Fault-Tolerance and Load Management in a Distributed Stream Processing System

by

Magdalena Balazinska

M.Sc.A. Electrical Engineering, École Polytechnique de Montréal (2000)
B.Ing. Computer Engineering, École Polytechnique de Montréal (2000)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 2005

Author . . . .
Department of Electrical Engineering and Computer Science
December 1, 2005

Certified by . . . .
Hari Balakrishnan
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Fault-Tolerance and Load Management in a Distributed Stream Processing System

by

Magdalena Balazinska

Submitted to the Department of Electrical Engineering and Computer Science
on December 1, 2005, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

Advances in monitoring technology (*e.g.*, sensors) and an increased demand for online information processing have given rise to a new class of applications that require continuous, low-latency processing of large-volume data streams. These "stream processing applications" arise in many areas such as sensor-based environment monitoring, financial services, network monitoring, and military applications. Because traditional database management systems are ill-suited for high-volume, low-latency stream processing, new systems, called stream processing engines (SPEs), have been developed. Furthermore, because stream processing applications are inherently distributed, and because distribution can improve performance and scalability, researchers have also proposed and developed *distributed SPEs*.

In this dissertation, we address two challenges faced by a distributed SPE: (1) fault-tolerant operation in the face of node failures, network failures, and network partitions, and (2) federated load management.

For fault-tolerance, we present a replication-based scheme, called *Delay, Process, and Correct* (DPC), that masks most node and network failures. When network partitions occur, DPC addresses the traditional availability-consistency trade-off by maintaining, when possible, a desired availability specified by the application or user, but eventually also delivering the correct results. While maintaining the desired availability bounds, DPC also strives to minimize the number of inaccurate results that must later be corrected. In contrast to previous proposals for fault tolerance in SPEs, DPC simultaneously supports a variety of applications that differ in their preferred trade-off between availability and consistency.

For load management, we present a *Bounded-Price Mechanism* (BPM) that enables autonomous participants to collaboratively handle their load without individually owning the resources necessary for peak operation. BPM is based on contracts that participants negotiate offline. At runtime, participants move load only to partners with whom they have a contract and pay each other the contracted price. We show that BPM provides incentives that foster participation and leads to good system-wide load distribution. In contrast to earlier proposals based on computational economies, BPM is lightweight, enables participants to develop and exploit preferential relationships, and provides stability and predictability. Although motivated by stream processing, BPM is general and can be applied to any federated system.

We have implemented both schemes in the Borealis distributed stream processing engine. They will be available with the next release of the system.

Thesis Supervisor: Hari Balakrishnan
Title: Associate Professor of Computer Science and Engineering

To my husband, Michel Goraczko.

Acknowledgments

First and foremost, I would like to thank Hari Balakrishnan for showing me the Cricket demo and getting me interested in building and studying systems. I greatly enjoyed all the projects he encouraged me to pursue from my first project on resource discovery to my dissertation on stream processing. I am profoundly indebted to Hari for all his guidance and help on each project. I was highly impressed by his patience, his knowledge, his accurate advice on any problem and any topic, and his ability to produce a clear and crisp description from the most confused and convoluted idea. Thanks, Hari, for everything!

My dissertation and graduate career would not have been the same without Mike Stonebraker. I deeply thank Mike for introducing me to the exciting database area and community, and most importantly for preventing me from building overly complex and impractical systems. In his very busy schedule, Mike always found the time to advise me both about research and about my career. Thank you Mike for all your kind advice, and for your very frank, direct, and always useful comments.

I thank Sam Madden for numerous helpful discussions, advice, and for always being ready to share his great expertise in database systems. I thank Barbara Liskov for being on my thesis committee and for providing highly insightful advice both on the thesis presentation and on the proposed algorithms.

I am also profoundly indebted to Rahul Sami for all his help on the load management scheme presented in this dissertation. Rahul always took the time to read and discuss my work related to game theory, and answer every single one of my numerous questions. Rahul provided invaluable comments, helping me improve the approach, fix errors, and tighten bounds. I would also like to thank Peyman Faratin for his helpful comments on negotiation techniques.

Jon Salz, one of the best programmers I have had the opportunity to meet, provided great help with the first versions of the Medusa system, which later became Borealis. Kwok Lee Tang and Bret Hull helped develop an early demonstration of Medusa which included RFID antennas, sensor networks, mobile phones, and teddy bears. I thank them all for their contributions to the project.

The fault-tolerance scheme presented in this dissertation gained in clarity and precision thanks to several intense discussions with Mehul Shah and Jeong-Hyon Hwang. Their attention to details made it a great joy to discuss the various corner cases and the general dynamics of failure and recovery in stream processing engines. Jeong-Hyon and I also worked together early on, studying process-pair based high availability techniques for stream processing engines. The operator taxonomy and buffer truncation scheme presented in this dissertation are based on that earlier work.

I would like to thank all members of the Borealis team: Daniel Abadi, Yanif Ahmad, Hari Balakrishnan, Bradley Berg, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Sam Madden, Anurag S. Maskey, Olga Papaemmanouil, Alex Rasin, Esther Ryvkina, Mike Stonebraker, Nesime Tatbul, Ying Xing, and Stan Zdonik, as well as all the members of the earlier Aurora project. Thank you for the great opportunity to discuss and reflect on so many different aspects of stream processing, and for all the fun we had at database conferences. In particular, I would like to thank Bradley for taking over the coordination of all student efforts on the Borealis project. I would also like to thank Bradley, Yanif, and all the students for their contributions to the most recent versions of Borealis. Jeong-Hyon also developed the lovely GUIs that I used at several occasions to demonstrate the load management and fault-tolerance schemes presented here.

Jaeyeon Jung and Dave Andersen from MIT, and Noah Case from Syptec, provided the network connection traces that I used in numerous demonstrations and experiments, some of which are presented in this dissertation. Thank you for giving me access to this data.

My research was supported at different times by a Microsoft Research Fellowship, the National Science Foundation, a scholarship from the Natural Sciences and Engineering Research Council of Canada, and MIT. I would like to thank all these institutions for their support.

I had the great privilege and pleasure of sharing my journey through graduate school with Nick Feamster, Kyle Jamieson, Jaeyeon Jung, Allen Miu, and Godfrey Tan. We started at about the same time. We had the opportunity to work together, drink and party together, and even build Lego robots together. From taking classes, discussing research, stressing about paper deadlines, to finally starting to plan life after graduate school, we have truly been together every step of the way. I'm not sure how I would have done it without such fabulous friends and first class colleagues. I would also like to thank all my other friends from NMS: Daniel Abadi, Dave Andersen, Vlad Bychkovsky, Bret Hull, Srikanth Kandula, Dina Katabi, Stan Rost, Eugene Shih, Asfandyar Qureshi, and all the others. Graduate school would definitely not have been both as interesting and fun without you.

In addition to all her help with various administrative tasks. Sheila Marian frequently brightened our graduate life with her most fantastic baking skills. From graduations to weddings, I can no longer imagine celebrating any office event without a few of her creative and delicious desserts.

From our discussions of Harry Potter, density estimation, and everything in between, I have learned so much from my great friend Chuck Blake that I could not even begin to enumerate it. I will always remember our fun coffee break expeditions and all our other activities and discussions. Thanks Chuck for your friendship and your help.

All 9th floor students and faculty have created an amazing environment to learn and thrive. I learned a lot by listening to the many practice talks, reading drafts of paper submissions, and having fun at systems conferences. I would also like to thank all those who patiently read my own submissions and came to my numerous practice talks. And to those in the milk pool, thanks for bringing milk on time!

I would like to thank my parents, Anna and Marek Balazinski, for encouraging me to pursue graduate studies and for always being there to share my successes and my worries. My older brother, Bartosz Balazinski, through his example, gave me the courage to even consider computer engineering as a major. I thank him for encouraging me to choose this most interesting field. My friends, Frederic Doucet, Eric Blaise, and Corinne Frasson inspired me to apply to graduate school at MIT by pursuing graduate studies at first-tier American universities themselves. I will always be grateful they opened this door for me. I would also like to thank Ettore Merlo, who advised me when I was just a master's student. After several years, I still remember our great discussions. Thanks for getting me started on the academic path.

Most importantly, I would like to thank my husband Michel Goraczko. His support and faith in my eventual graduation never wavered. He always found the best words to cheer me up, and the best equipment to run my experiments and demonstrations. Michel, I thank you for your love and support, and dedicate this work to you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, a new class of data-intensive applications has emerged. These applications, called *stream processing applications*, require *continuous* and low-latency processing of large volumes of information that "stream in" from data sources at high rates. Stream processing applications have emerged in several different domains motivated by different needs.

Advances in miniaturization and wireless networking have made it possible to deploy specialized devices capable of sensing the physical world and communicating information about that world. Examples of these devices include a wide variety of environmental sensors [161], miniature tags for object tracking [60], location-sensing devices [77, 79, 133], etc. Deploying these devices enables applications such as sensor-based environment monitoring (*e.g.*, building temperature monitoring, air quality monitoring), civil engineering applications (*e.g.*, highway monitoring, pipeline health monitoring), RFID-based equipment tracking, military applications (*e.g.*, platoon tracking, target detection), and medical applications (*e.g.*, sensor-based patient monitoring). All these applications must continuously process streams of information produced by the deployed devices.

In some areas, such as computer networks (*e.g.*, intrusion detection, network monitoring, tracking of worm propagation), web logs or clickstream monitoring, financial services (*e.g.*, market feed processing, ticker failure detection), applications have traditionally involved processing large volume data streams. These applications, however, typically store data persistently and mine it offline [51], which introduces a significant delay between the time when events occur and the time when they are analyzed. Alternatively, applications can process data online using specialized software [136], but that is expensive to implement.

As has been widely noted [2, 17, 33], traditional database management systems (DBMSs) based on the "store-then-process" model are inadequate for high-rate, low-latency stream processing. As a result, several new architectures have been proposed. The new engines are called *stream processing engines (SPEs)*, data stream management systems (DSMS) [2, 117], or continuous query processors [33]. Their goal is to offer data management services that meet the needs of all the above applications in a single framework. Because stream processing applications are inherently distributed, and because distribution can improve the performance and scalability of the processing engine, researchers have also proposed and developed *distributed SPEs* [33, 37].

Several challenges arise when developing a distributed SPE. In this dissertation, we focus on two challenges in particular: *fault-tolerance in a distributed SPE* and *load management in a federated environment.* The goal of our fault-tolerance mechanism is to enable a distributed SPE to survive processing node failures, network failures, and network partitions.

The goal of our load management approach is to create incentives and a mechanism for autonomous participants to collaboratively handle their load without having to individually own and administer the necessary resources for peak operation. Our load management technique is motivated by stream processing but is generally applicable to other federated systems. We implemented both techniques in the Borealis distributed SPE [1, 27], and evaluate their performance through analysis, simulations, and experiments.

In the rest of this chapter, we describe the main properties of stream processing applications and the key features of stream processing engines. We also present the fault-tolerance and load management problems addressed in this dissertation, outline our contributions, and discuss some of our key findings.

1.1 Stream Processing Applications

Stream processing applications differ significantly from traditional data management applications. The latter typically operate on bounded data-sets by executing one-time queries over persistently stored data. A typical example is a retail business that needs an application to keep track of its inventory, sales, equipment, and employees. DBMSs such as Oracle [125], IBM DB2 [148], and Microsoft SQL Server [114] are designed to support such traditional types of applications. In these systems, data is first stored and indexed. Only then are queries executed over the data.

In contrast, as illustrated in Figure 1-1, in a stream processing application, data sources produce unbounded streams of information and applications execute continuous, long-running queries over these streams. Network monitoring is an example of a stream processing application. In this application, the data sources are network monitors that produce information about the traffic originating in or destined to machines on monitored subnetworks. A possible goal of the application is to continuously compute statistics over these streams, enabling a network administrator to observe the state of the network as well as detect anomalies such as possible intrusion attempts.

Stream processing applications have the following properties [2, 9, 17, 21, 33]:

1. *A continuous-query processing model*: In a traditional DBMS, clients issue one-time queries against stored data (*e.g.*, "Did any source attempt more than 100 connections within a one minute period?"). In a stream processing application, clients submit long-duration monitoring queries that must be processed continuously as new input data arrives (*e.g.*, "Alert me if a source attempts more than 100 connections within a one minute period"). Clients submitting continuous queries expect periodic results, or alerts when specific combinations of inputs occur.
2. *A push-based processing model*: In a stream processing application, one or more data sources (*e.g.*, sensor networks, ticker feeds, network monitors) continuously produce information and push the data to the system for processing. Client applications passively wait for the system to push them periodic results or alerts. This processing model contrasts with the traditional model, where the DBMS processes locally stored data, and clients actively pull information about the data when they need it.
3. *Low latency processing*: Many stream processing applications monitor ongoing phenomena and require low latency processing of input data. For instance, in network monitoring, current information about ongoing intrusions is more valuable than stale information about earlier attacks. SPEs strive to provide low-latency processing but do not make any hard guarantees.

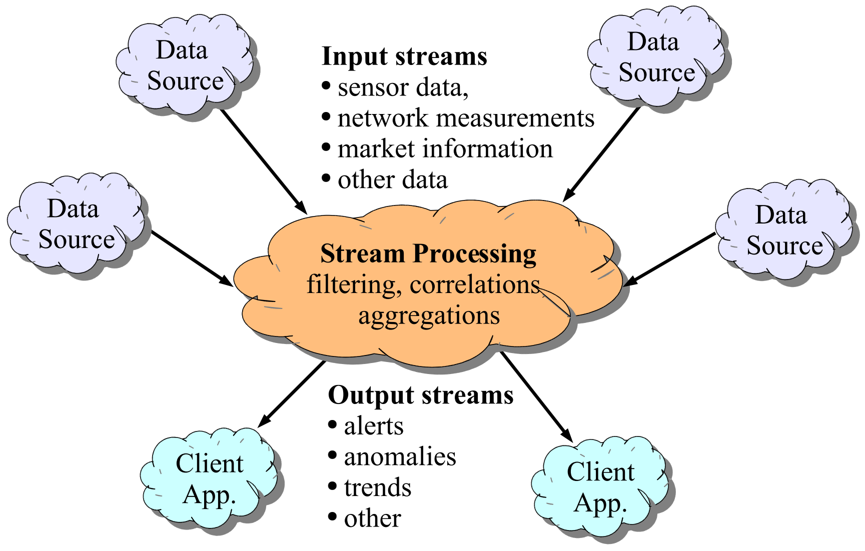


Figure 1-1: High-level view of stream processing. Data sources continuously produce streams of information. These streams are then continuously processed and results are pushed to client applications.

4. *High and variable input data rates*: In many stream processing applications, data sources produce large volumes of data. Input data rates may also vary greatly. For instance, a denial-of-service (DoS) attack may cause large numbers of connections to be initiated. If network monitors produce one data item per connection, the data rates on the streams they produce will increase during the attack. As the data rates vary, the load on an SPE also varies because query operators process data arriving at a higher rate.

1.2 Stream Processing Engines

To support stream processing applications, SPEs introduce new data models, operators, and query languages. Their internal architectures also differ from those of traditional DBMSs. They are geared toward processing new data as it arrives by pushing it through a set of continuous queries registered in the system.

1.2.1 Data and Processing Model

Several data models have been proposed for stream processing [2, 10, 11, 100, 168]. These models are based on the core idea that a stream is an append-only sequence of data items. Data items are composed of attribute values and are called *tuples*. All tuples in the same stream have the same set of attributes, which defines the type or *schema* of the stream. Typically, each stream is produced by a single data source. Figure 1-2 shows an example of streams, tuples, and schemas for a network monitoring application. In this example, network monitors are data sources. They produce input streams, where each tuple represents one connection and has the following schema: time when the connection was established,

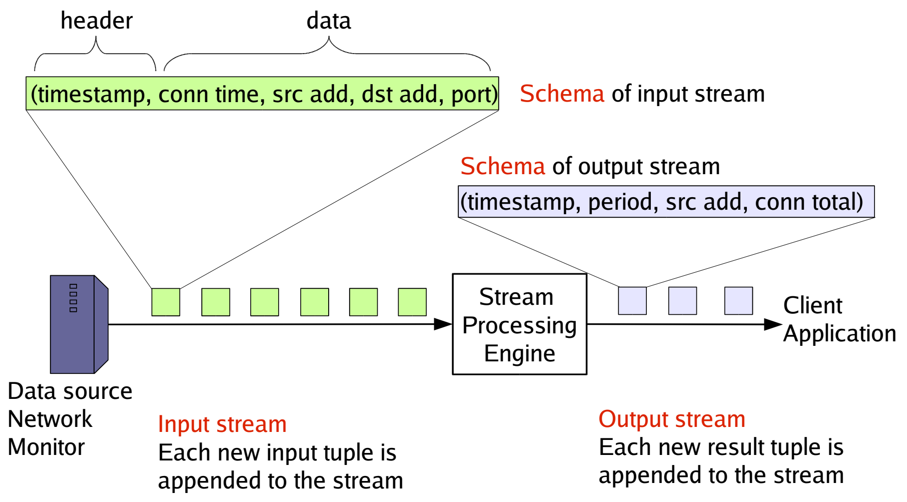

header
data
(timestamp, conn time, src add, dst add, port)
Schema of input stream
Schema of output stream
(timestamp, period, src add, conn total)
Stream Processing Engine
Client Application
Data source Network Monitor
Input stream
Each new input tuple is appended to the stream
Output stream
Each new result tuple is appended to the stream

Figure 1-2: Example streams and schemas, in a network monitoring application.

source address, destination address, and destination port number. The output stream has a different schema. Each tuple indicates how many connections were established by each distinct source address for each per-defined period of time.

In stream processing applications, data streams are filtered, correlated, and aggregated by *operators* to produce outputs of interest. An operator may be viewed as a function that transforms one or more input streams into one or more output streams. Because SPEs are inspired by traditional DBMSs, they support operators analogous to relational operators such as Select, Join, Project, Union, and Aggregate [2, 10, 33]. Some engines (*e.g.*, Aurora [2]/Borealis [1]) also support user-defined operators. However, our experience building various applications shows that most of application logic can be built directly using pre-defined operators [21]. Because streams are unbounded in size and because applications require timely output, operators can neither accumulate state that grows with the size of their inputs nor can they wait to see all their inputs before producing a value. For this reason, stream processing operators perform their computations over *windows of data* that move with time. These windows are defined either by assuming that tuples on a stream are ordered on their timestamp values [10, 100], or on one of their attributes [2], or by inserting explicit *punctuation* tuples that specify the end of a subset of data [166, 167, 168]. Window specifications make operators sensitive to the order of the input tuples.

In Aurora [2] and Borealis [1], applications determine how input streams should be processed by composing the pre-defined and the user-defined operators into a workflow-style loop-free, directed graph, called a *query diagram*. Other stream-processing engines use SQL-style declarative query languages [10, 33, 43, 100] and translate declarative queries into query diagrams. Figure 1-3 illustrates a simple query diagram. The query, inspired by the type of processing performed in Snort [136] and Autofocus [51], is a simple network monitoring application. Tuples in input streams each summarize one network connection: source and destination addresses, connection time, protocol used, etc. First, the streams from all the network monitors are *unioned* (operator `a`) into a single stream. The query then transforms that stream to identify sources that are either active (operators `b` and `c`), or that try to connect over too many distinct ports within a short time period (operators `d` and

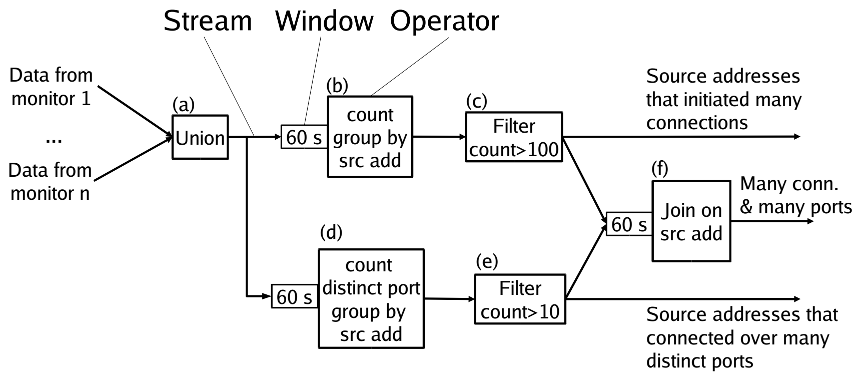


Figure 1-3: Example of a query diagram from the network monitoring application domain.

`e`), or both (operator `f`). To count the number of connections and ports the query applies windowed aggregate operators (`b` and `d`): these operators buffer connection information for a time period T. In the example, T is 60 seconds. The operators then group the information by source address and apply the desired aggregate function. Aggregate values are then *filtered* to identify the desired type of connections. In the example, a source is labeled as active if it establishes more than 100 connections within a 60-second interval or connects over more than 10 distinct ports. Finally, operator `f` *joins* active sources with those connecting over many ports on the source address field to identify sources that belong in both categories.

Some systems allow queries over both streams and stored relations [11, 69, 162]. We follow the model introduced in Aurora [2] and restrict processing to append-only data streams. We allow, however, read and write operators [21] that perform a "SQL update" command for every input tuple they receive, and may produce streams of tuples as output.

1.2.2 Distributed Operation

SPEs are an example of a naturally distributed system because data sources are often spread across many geographic locations and belong to different administrative entities. Distribution can also improve the performance and scalability of a stream processor, and enables high-availability because processing nodes can monitor and take over for each other when failures occur [2, 33].

A distributed SPE is composed of multiple physical machines. Each machine, also called a *processing node* (or simply *node*), runs an SPE. Each node processes input streams and produces result streams that are either sent to applications or to other nodes for further processing. When a stream goes from one node to another, the two nodes are called *upstream* and *downstream neighbors*, respectively. Figure 1-4 illustrates a possible deployment for the query from Figure 1-3. The data sources are at remote and distributed locations, where the network monitors are running. All input streams are unioned at Node 1 before being processed by operators running at Nodes 1, 2, and 3. Each operator receives input tuples through its *input queues*. When streams cross node boundaries, the output tuples are temporarily buffered in *output queues*.

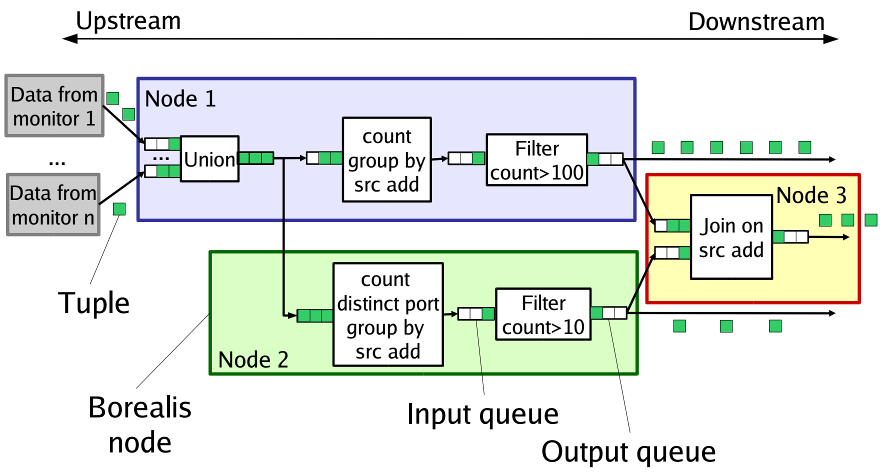


Figure 1-4: Example of distributed stream processing.

1.3 Challenges and Contributions

There are several challenges in building a distributed and possibly federated stream processing engine. In this dissertation, we address the following two problems: fault-tolerant distributed stream processing and load management in a federated environment. For each problem, we devise an approach, describe its implementation in the Borealis distributed SPE, and demonstrate its properties through analysis, simulations, and experiments.

1.3.1 Fault-Tolerance Challenges

In a distributed SPE, several types of failures can occur: (1) processing nodes can fail and stop, (2) the communication between nodes can be interrupted, and (3) the system can partition. All three types of failures can disrupt stream processing: they can affect the correctness of the output results and can even prevent the system from producing any results. We address the problem of protecting a distributed stream processing engine from these three types of failures. For node failures, we focus on crash failures. We do not address Byzantine failures, which cause nodes to produce erroneous results.

Ideally, even though failures occur, we would like client applications to receive the correct output results. For many node and network failures, the system can provide this property by replicating each processing node and ensuring that replicas remain mutually consistent, *i.e.*, that they process their inputs in the same order, progress at roughly the same pace, that their internal computational state is the same, and that they produce the same output in the same order. If a failure causes a node to lose one of its input streams, the node can then switch and continue processing from a replica of the failed or disconnected upstream neighbor. The state of replicas and the output seen by clients remain identical to a state and output that could have existed with a single processing node and without failures.

The challenge is that some types of failures, such as network partitions, can cause a node to lose access to all replicas that produce one of its input streams. To maintain consistency, the node must then block, making the system unavailable. Fault-tolerance through replication is widely studied and it is well known that it is not possible to provide both consistency and availability in the presence of network partitions [28, 68]. Many stream processing applications are geared toward monitoring tasks, and can benefit from early results even if these results are somewhat inaccurate because they are based on a subset of input streams. Some applications may even *prefer* to see approximate results early, rather than wait for correct results. For example, even if only a subset of network monitors is available, processing their data might suffice to identify some potential network intrusions, and low latency processing is critical to mitigate attacks. We therefore propose that nodes *continue processing available inputs in order to maintain availability.* Because client applications typically monitor some ongoing phenomenon and passively wait for the system to send them results, we define availability as a *low per-tuple processing latency.* This definition differs from that of traditional optimistic replication schemes [140].

We do not want to sacrifice consistency completely, however, because for many applications that favor availability, the correct results are also useful. In network monitoring, for example, it may be important for an administrator to eventually know about all attacks that occurred in the network or to know the exact values of different aggregate computations (*e.g.*, traffic rates per customer). Hence, the main fault-tolerance challenge that we address is to ensure that client applications always see the most recent results but *eventually* also receive the correct output results, *i.e.*, we would like the system to maintain availability and provide *eventual consistency.*[1]

To achieve the above goal, the SPE may have to allow replicas to become temporarily inconsistent. We measure inconsistency by counting the number of inaccurate results that must later be corrected, since that is often a reasonable proxy for replica inconsistency. Because it is expensive to process then correct results in an SPE, our second goal is to study techniques that minimize the number of inaccurate results, *i.e.*, we seek to *minimize inconsistency while maintaining a required availability.* We observe that stream processing applications differ in their preferred trade-offs between availability and consistency, and we propose to use these preferences to minimize inconsistency. Some applications (*e.g.*, sensor-based patient monitoring) may not tolerate any inconsistent results. Other applications (*e.g.*, network intrusion detection) may have to see the most recent data at any time. Other applications yet (*e.g.*, sensor-based environment monitoring) may tolerate bounded increase in processing latency if this helps reduce inconsistency. We propose to enable applications to set their desired trade-off between availability and consistency, by specifying how much longer than usual they are willing to wait for results, if this delay reduces inconsistency. The system should minimize inconsistency, while striving to always produce new results within the required time-bound. Providing a flexible trade-off between availability and consistency distinguishes DPC from previous techniques for fault-tolerant stream processing that handle only node failures [83] or strictly favor consistency over availability [146]. DPC is thus better suited for many monitoring applications, where high availability and near real-time response is preferable to perfect answers. At the same time, DPC supports, in a single framework, a wide variety of applications with different preferred trade-offs between availability and consistency.

Achieving the two main goals above entails a significant set of secondary challenges. First, we must devise a technique for replicas to remain mutually consistent in the absence of failures. In contrast to traditional databases and file systems, an SPE does not operate

[1]Correcting earlier results requires replicas to eventually reprocess the same input tuples in the same order, and by doing so reconcile their states. We thus call the eventual correctness guarantee *eventual consistency*, based on the same notion introduced by optimistic replication schemes for databases and file systems [140], although for an SPE, eventual consistency additionally includes the correction of previously produced results.

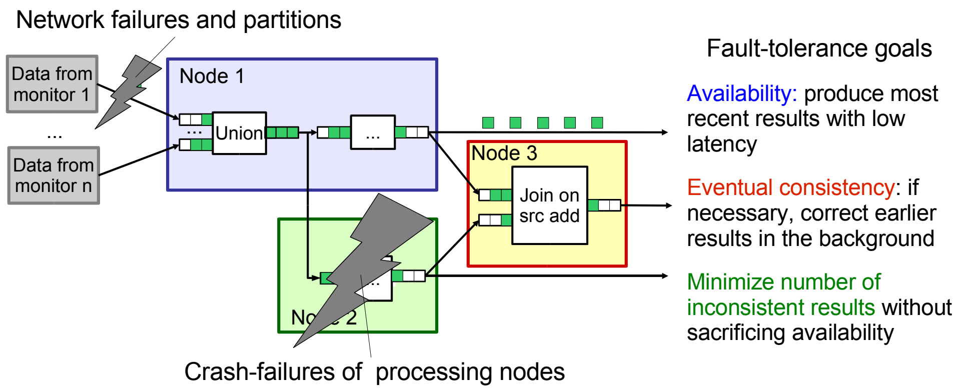

Network failures and partitions
Data from monitor 1
...
Data from monitor n
Node 1
Union
...
Node 2
Node 3
Join on src add
Crash-failures of processing nodes
Fault-tolerance goals
Availability: produce most recent results with low latency
Eventual consistency: if necessary, correct earlier results in the background
Minimize number of inconsistent results without sacrificing availability

Figure 1-5: Types of failures and fault-tolerance goals.

on a persistent state but rather on a large and rapidly changing transient state. Additionally, in a distributed SPE, the output produced by some nodes serves as input to other nodes, and must also be consistent. These two properties make the replication problem different from that addressed in previous work. Similarly, traditional approaches to record reconciliation [93, 169] are ill-suited to reconcile the state of an SPE because the SPE state is not persistent and depends on the order in which the SPE processed its input tuples. We thus need to investigate new techniques to reconcile SPE states. Third, to provide eventual consistency, processing nodes must buffer some intermediate tuples. For failures that last a long time, the system may not be able to buffer all the necessary data. We thus need a way to manage the intermediate buffers and handle long-duration failures. Finally, we need to enhance stream data models to support the distinction between a new tuple and a tuple correcting an earlier one.

In summary, fault-tolerance is a widely-studied problem, but the unique requirements and environment of stream processing applications create a new set of challenges. We propose to devise a new approach to fault-tolerance geared specifically toward distributed SPEs. The main challenge of the approach is to provide *eventual consistency while maintaining an application-defined level of availability.* The second challenge that we address is to study techniques that achieve the first goal in a manner that minimizes inconsistency. Figure 1-5 summarizes the types of failures that we would like a distributed SPE to handle and the fault-tolerance goals that we would like the SPE to achieve.

1.3.2 Fault-Tolerance Contributions

The main contribution of our work is to devise an approach, called *Delay, Process, and Correct* (DPC) [22] that achieves the goals outlined in the previous section.

The core idea of DPC is to favor replica autonomy. In DPC, nodes independently detect failures of their input streams and manage their own availability and consistency, by following a state-machine composed of three states: a *stable* state, an *upstream failure* state, and a *stabilization* state. To ensure replica consistency in the stable state, we define a simple data-serializing operator, called *SUnion*, that takes multiple streams as input and produces one output stream with deterministically ordered tuples. Additionally, we use a heartbeat-based mechanism that enables a node to decide, with respect to its inputs, when a failure has occurred. With these two techniques, a node need only hear from one replica of each one of its upstream neighbors to maintain consistency. Replicas do not need to communicate with each other. When a node detects the failure of one or more of its inputs, it transitions into the upstream failure state, where it guarantees that available inputs capable of being processed are processed within the application-defined time threshold. At the same time, the node tries to avoid or at least minimize the inconsistency that it introduces into the system by finding stable upstream replicas, and, when necessary, judiciously suspending or delaying the processing of new data. Once a failure heals and a node receives the previously missing or inconsistent input, it transitions into the stabilization state. During stabilization, nodes reconcile their states and correct earlier output results (thus ensuring eventual consistency), while maintaining the availability of output streams. To achieve consistency after a failure heals, we develop approaches based on checkpoint/redo and undo/redo techniques. To maintain availability during stabilization, we develop a simple inter-replica communication protocol that ensures at least one replica remains available at any time. Finally, to support DPC, we also introduce an enhanced streaming data model in which results based on partial inputs are marked as *tentative*, with the understanding that they may subsequently be modified; all other results are considered *stable* and immutable.

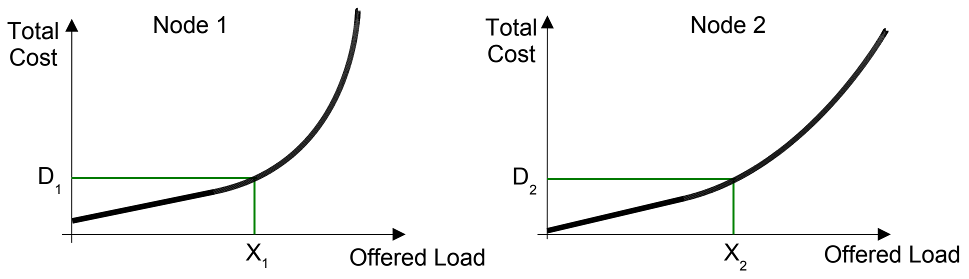
We implemented DPC in Borealis and evaluated it through analysis and experiments. We show that DPC achieves the desired availability/consistency trade-offs even in the face of diverse failure scenarios including multiple simultaneous failures and failures during recovery. We find that requiring each node to manage its own availability and consistency helps handle complex failure scenarios.

We also show that DPC has a low processing latency overhead. The main overhead of DPC comes from buffering tuples at the location of failures in order to reprocess them during stabilization. For certain common types of query diagrams, however, DPC supports arbitrarily-long failures, while still ensuring that, after failures heal, the system converges to a consistent state and the most recent tentative tuples get corrected.

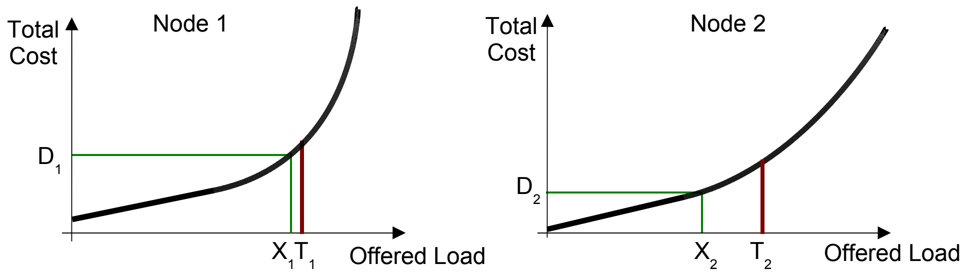
To reconcile the state of a node, we find that checkpoint/redo outperforms undo/redo. We also find that it is possible to reconcile the state of an SPE using checkpoints and redo, but limiting checkpoints and reconciliation to paths affected by failures and avoiding any overhead in the absence of failures.

We show that DPC performs especially well in the face of the non-uniform failure durations observed in empirical measurements of system failures: most failures are short, but most of the downtime of a system component is due to long-duration failures [54, 72]. DPC handles short failures by suspending processing and avoiding inconsistency. In fact, we find that is possible for the system to *avoid inconsistency for all failures shorter than the application defined maximum incremental latency*, independently of the size of the distributed system and the location of the failure. For long-duration failures, one of the greatest challenges of DPC is the guarantee to process all available tuples within a pre-defined time bound, while also ensuring eventual consistency. To maintain availability at all times, we show that it is necessary to perform all replays and corrections in the background. Every time a node needs to process or even simply receive old input tuples, it must nevertheless continue to process the most recent input tuples.

Overall, with DPC, we show that it is possible to build a single scheme that enables a distributed SPE to cope with a variety of failures, in a way that provides applications the freedom to chose the desired trade-off between availability and consistency.



(a) Minimal cost allocation. Total cost, $D_1 + D_2$, is less than cost of any other assignment.



(b) Acceptable allocation. For each node, the load, X_i is below a pre-defined threshold, T_i.

Figure 1-6: Different types of load allocations in a system of two nodes.

1.3.3 Load Management Challenges

The second problem that we address is the management of load variations in a distributed and federated system.

When a user submits a query, a distributed SPE assigns individual operators to different processing nodes. Because queries are continuous and long running, the amount of resources (memory, bandwidth, CPU) that each operator uses is likely to vary during the lifetime of the query. The variations are typically caused by changes in the data rates or data distributions on input streams. Users may also dynamically add and remove queries from the system, causing further load variations. In response to these changes, the system may eventually need to modify the original allocation of operators to nodes, in order to improve overall performance or at least avoid serious performance degradation.

The problem of dynamically managing load in a distributed system by continuously reallocating tasks or resources is well-studied (*e.g.*, [49, 64, 96]). Traditional techniques define a system-wide utility function, such as average total processing time [64, 96] or throughput [96] and aim to reach an allocation that optimizes this utility. Figure 1-6(a) shows an example of a system with two processing nodes. Each node has a function that gives the total processing cost for a given total load. The cost could, for example, represent the average queue size at the node. In this example, an optimal load allocation minimizes the sum of the processing costs (*i.e.*, minimizes $D_1 + D_2$). By doing so, the allocation optimizes system-wide utility given by the total number of messages queued in the system.

Most previous approaches assume a collaborative environment, where all nodes work together to maximize overall system performance. Distributed systems are now frequently deployed in federated environments, where different autonomous organizations own and administer subsets of processing nodes and resources. Examples of such federated systems include computational grids composed of computers situated in different domains [3, 29, 61, 164], overlay-based computing platforms such as Planetlab [131], Web-service-based cross-company workflows where end-to-end services require processing by different organizations [48, 94], and peer-to-peer systems [38, 45, 97, 120, 137, 153, 174]. Our goal is to enable load management in these new settings. Although motivated by stream processing, our approach is general and can be applied to any federated system.

Several issues make load management in a federated system challenging. Most importantly, participants do not strive to optimize system-wide utility but are driven by self-interest. They collaborate with others only if the collaboration improves their utility. Therefore, we need a scheme that provides *incentives* for participants to handle each other's load; we need to design a *mechanism* [86, 127] (*i.e.*, an incentive mechanism).

A *computational economy*, where participants provide resources and perform computing for each other in exchange for payment,[2] is a natural technique to facilitate collaborative load management between selfish participants. Several computational economies have been proposed in the past (*e.g.*, [3, 29, 154, 175]), but they have all failed to gain widespread acceptance in practice. This observation leads us to argue that traditional computational economies are not the correct solution. They provide a mechanism for participants to improve their utility by performing computation for others, but this facility appears to be insufficient to motivate *widespread* adoption.

In contrast to computational economies, bilateral agreements are frequently used in practice to enable collaborations between autonomous parties. Service level agreements (SLAs) routinely govern relationships between customers and service providers [81, 171, 178], between companies interconnecting through Web services [42, 94, 138], or between Internet Service Providers (ISPs) who agree to carry each other's traffic [53, 176]. Similarly to computational economies, bilateral agreements offer participants incentives to collaborate because participants agree to reward each other for the service they provide. Bilateral agreements, however, provide several additional features. They provide partners with privacy in their interactions with each other. They enable price and service discrimination [101], where a participant offers different qualities of service and different prices to different partners. Because they are based on long-term relationships, pairwise agreements also offer stability and predictability in interactions between participants. Finally, agreements make runtime interactions simpler and more lightweight than computational economies because most conditions are pre-negotiated, reducing runtime overhead.

Our goal is to develop a load management mechanism that enables a federated system to achieve good load distribution while providing the same benefits as bilateral agreements: incentives to collaborate, privacy in interactions, predictability in prices and load movements, stability in relationships, and runtime simplicity.

1.3.4 Load Management Contributions

Inspired by the successful use of contracts in practice, we propose a distributed mechanism, the *Bounded-Price Mechanism* (BPM) [23], for managing load in a federated system based on *private pairwise contracts*. Unlike computational economies that use auctions or implement global markets to set resource prices at runtime, our mechanism is based on *offline* contract negotiation. Contracts set tightly *bounded prices* for migrating each unit of load

[2]Non-payment models, such as bartering, are possible too. See Chapter 2 for details.

between two participants and may specify the set of tasks that each is willing to execute on behalf of the other. With BPM, runtime load transfers occur only between participants that have pre-negotiated contracts, and at a unit price within the contracted range. The load transfer mechanism is simple: a participant moves load to another if the expected local processing cost for the next time-period is larger than the payment it would have to make to the other participant for processing the same load (plus the migration cost).

We argue that optimal load balance is not an important goal in a federated system. Most participants will own sufficient resources to provide good service to their clients most of the time. Instead, our goal is to achieve an *acceptable allocation*, a load distribution where either *no* participant operates above its capacity, or, if the system as a whole is overloaded, *all* participants operate above their capacity. Figure 1-6(b) illustrates an acceptable allocation. The allocation does not minimize the sum of all processing costs but ensures that the load level, X_i, at each node is below the node's pre-defined capacity, T_i. Because we aim at achieving an acceptable allocation rather than an optimal load balance, our work significantly differs from other approaches geared toward selfish participants.

We have designed and implemented our approach in Borealis. Using analysis, simulations, and experiments, we show that the mechanism provides enough incentives for selfish participants to handle each other's excess load, improving the system's load distribution. We also show that the mechanism efficiently distributes excess load when the aggregate load both underloads and overloads total system capacity. The mechanism ensures good stability: it handles most load variations without reallocating any load. We show that it is sufficient for each participant to have a handful of contracts in order to get most of the benefits of the system. Additionally, contracts only need to specify a small price-range in order for the mechanism to always converge to acceptable allocations. With a price range, we show that the optimal strategy for participants is to quickly agree on a final price. Most of the time, the best approach is to not negotiate at all. We further show that the mechanism works well even when participants establish heterogeneous contracts at different unit prices with each other. Finally, we show that our approach has low runtime overhead.

Overall, we show with BPM that a simple, lightweight, and practical technique that leverages offline, long-term relationships between participants can lead to excellent load balance properties: acceptable and stable load allocations. At the same time, BPM provides participants with privacy, predictability in prices and possible runtime load movements, and possibility for service customization.

1.4 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we present an overview of related work on stream processing, fault-tolerance in SPEs and other data management systems, and load-management in federated environments. Because our implementation is in the Borealis SPE, in Chapter 3, we outline some aspects of the Borealis system architecture relevant to this work. In Chapter 4, we present and analyze DPC. In Chapter 5, we discuss the implementation of DPC in Borealis and evaluate DPC's performance through experiments with our prototype implementation. In Chapter 6, we present and analyze BPM. In Chapter 7, we complement the analysis with simulation results, describe BPM's implementation in Borealis, and present the results of some experiments with our prototype implementation. Finally, in Chapter 8, we conclude and discuss possible areas for future work.

Chapter 2

Background and Related Work

In this chapter, we discuss in more details the differences between traditional DBMSs and SPEs. We present related work on fault-tolerance in DBMSs, SPEs, and other data management systems, and related work on load management in federated environments.

More specifically, in Section 2.1, we discuss the limitations of traditional DBMSs to support stream processing applications, and present various extensions that have been proposed in the past to enhance DBMS capabilities. We also discuss traditional techniques to achieve fault-tolerance. In Section 2.2, we present the main research projects in the area of SPEs and discuss the specific efforts to achieve fault-tolerant stream processing. In Sections 2.3 through 2.7, we discuss fault-tolerance in workflow systems, publish/subscribe systems, sensor networks, state-machine replication, and rollback recovery. Finally, in Section 2.8, we present related work on load management in federated environments.

2.1 Traditional DBMSs and Extensions

Stream processing applications require continuous, low-latency processing of large volumes of streaming data. These requirements contrast with the capabilities of traditional DBMSs in three ways.

First, stream processing applications require new or at least extended data models, operators, and query languages. Law *et al.* [100] formally study the mismatch between SQL capabilities and the requirements of stream processing queries. The main problem is that the relational algebra assumes data sets are bounded in size, which is reflected in the operator design. For example, some relational operators (*e.g.*, aggregate operators such as min, max, or average) may need to process their whole input before producing a result. Stream processing applications typically monitor some ongoing phenomenon and want periodic updates rather than a single value at the end of a stream. Other relational operators (such as Join) accumulate state that grows with the size of their inputs. Such state management is inappropriate for streams, where tuples arrive continuously and the input can be potentially unbounded in size. To address these shortcoming, most SPE propose the use of some type of *windowed* specifications [2, 9, 33] as a way to process tuples in groups.

Second, the traditional data processing model requires that data be persistently stored and indexed before queries can be executed over that data. More generally, DBMSs assume they can dictate the movements of data to and from disk and can examine the same tuples multiple times, if necessary. These assumptions and this processing model break for continuously streaming inputs. SPEs propose an architecture where data is processed continuously as it arrives before or even without being stored [2, 9, 33, 43, 158]. Linear Road [12] is a benchmark developed for measuring and comparing the performance of stream processing engines. The authors of the benchmark show that, for stream processing workloads, a specialized SPE can outperform a traditional DBMSs by at least a factor of five.

Finally, traditional DBMSs have only limited support for alerting and continuous query capabilities, which are the main types of stream processing queries. A materialized view [73] is a type of continuous query. It is a derived stored relation defined in terms of base stored relations. A materialized view is thus like a cache that must be updated (incrementally or through re-execution) as the base relations change. Materialized views, however, are not designed for the push-based style of processing required by stream processing applications. The user must poll the state of a materialized view.

2.1.1 DBMS Extensions

Stream processing applications are not the first applications to break the traditional database management model. Before SPEs, various extensions to DBMS had already been proposed. Some of these extensions, such as triggers, main-memory processing, real-time processing, support for sequenced data, and the notion of time, bear similarities with the goals of SPEs. We now present and overview of some of these earlier proposals.

A traditional DBMS performs only one-time queries on the current state of persistently stored relations. Active databases [74, 130, 144, 155, 177] aim to enhance a DBMS with monitoring and alerting capabilities. In commercial databases, triggers [75, 95, 103] are the most commonly adopted technique to turn a passive database into an active one. A trigger is a pre-defined action executed by the DBMS when some pre-defined combination of events and conditions occurs. Both actions and events are database operations. In contrast to SPEs, active databases still operate on the current state of locally-stored relations. They simply monitor this state and react to changes in that state. A naive implementation of triggers does not scale, because every event requires the system to scan all relevant triggers to find those with matching conditions, and testing each condition may correspond to running an expensive query. To enable a greater scalability, triggers can be grouped into equivalence classes with an index built on their different conditions [75]. SPEs, however, claim that their new processing model is more scalable and better suited for stream processing applications than a trigger-based model [2].

In the relational model, an important design decision was to treat relations as unordered sets of tuples. This choice facilitates query optimization but fails to support applications that need to express queries over sequences of data rather than data sets. One example of such applications is in the financial services area, where applications need to examine stock price fluctuations. Sequence databases [87, 102, 112, 145] address this shortcoming. They propose new data models (*e.g.*, array data type [112], sequence enhanced abstract data type [145], or arrable [102]), new operators (*e.g.*, order preserving variants of relational operators), and query languages (*e.g.*, support for "ASSUMING ORDER on" clauses or definitions of windows of computation) to support data sequences in a relational engine. Sequence databases have greatly influenced the design of query languages for streams (*e.g.*, [2, 11]), but the latter face a few additional challenges that sequence databases ignore. In stream processing data is not stored and readily available. Instead, it streams continuously, possibly at a high rate. Tuples may arrive at least somewhat out of order but it is not possible to wait until the end of the stream to sort all the data and start processing it. SPEs thus either assume only bounded disorder [2, 11] and drop tuples out-of-order, or use

DBMS extension	Similarity to SPE	Key difference
Active	Monitoring and alerting capabilities	The traditional processing model of active databases is geared toward low event rates. Active databases operate on the current state of locally stored relations.
Sequence	Notion of tuple order	Sequence databases assume sequences are bounded in size and stored locally. In SPEs, data is pushed continuously into the system.
Temporal	Notion of time	Temporal databases enable queries over different physical or logical states of a database. In contrast, SPEs annotate tuples with timestamps primarily for ordering purposes.
Real-time	Process with deadlines	Real-time databases strive to complete one-time queries by given deadlines. SPEs provide low-latency processing of continuous queries without guarantees.
Main-memory	Keep data in-memory	Main-memory databases assume all tuples fit in memory. SPEs continuously receive new tuples and drop old tuples.
Adaptive	React to changing conditions during query execution	Adaptive databases are designed for one-time queries over data sets bounded in size rather than continuous queries.

Table 2.1: Extensions to DBMSs and their similarities with SPE features.

constructs such as punctuation [168] to tolerate disorder.

In traditional DBMSs, users query the *current* state of the database. There is no notion of a timeline showing when different updates occurred or when different tuples were valid. Temporal databases [126, 150] address this problem and support either one or both of the following notions of time. Transaction time (*aka*, physical time) is the time when information is stored in the database. With transaction time, an application can examine the state of the database at different points in time. For instance, an application could examine the state of the database as of last Friday. Valid time (*aka*, logical time) is the time when the stored information models reality. Valid time enables applications to record history and edit that history, as necessary. An application could record the history of an employee's salary over time and for example change the time when an employee received a raise if that time was initially incorrectly entered. All current databases also support a third notion of time, which is simply user-defined time in the form of a timestamp attribute. SPEs do not have such powerful support for time-based operations. Tuples are typically annotated with a timestamp value but these values serve ordering more than timing purposes. Recently, however, Borealis [1] started to explore more advanced notions of time in SPEs. Borealis studies possibilities for *time travel*, when an SPE goes back in time and re-processes some tuples or goes into the future and operates over predicted data.

Real-time databases [91, 98, 126] are databases where transactions have deadlines and the goal of the system is to complete transactions by their deadlines. SPEs strive for low-latency processing because they support monitoring applications, but they do not make any specific guarantees.

Traditional DBMSs operate on data that is stored on disk. In contrast, in a main-memory database [66], all data resides permanently in memory and only backups are written to disk. Because accessing memory is faster than accessing a disk, main-memory databases can achieve greater performance. These types of architectures are thus especially suitable for applications that have small data sets and real-time processing requirements [66]. Current SPEs are main-memory databases. In contrast to previous proposals, their architecture is geared toward streaming applications. They cannot assume that all data will fit in memory. They continuously receive new data and drop old data.

Adaptive databases address yet another shortcoming of traditional DBMSs. With the advent of the Internet, DBMSs no longer solely process data stored locally but also operate on data transferred from remote locations [85]. Remote data may have to be accessed over the wide area and may reside under a different administrative domain. For this reason, it is difficult if not impossible for the query optimizer to have access to accurate statistics about the data (*e.g.*, cardinality, arrival rate) when it decides on the query plan. The query processor must instead be able to react and adapt to changes or unexpected conditions [15, 84, 170]. This adaptation requires the ability to change the order in which tuples are processed as the query proceeds. SPEs must also be adaptive as they typically receive data from remote locations. The adaptation is particularly challenging because queries are continuous and long-running, making it likely that conditions will change during query execution.

Table 2.1 summarizes all these different extensions to traditional DBMSs, emphasizing their key similarities and differences with SPEs.

2.1.2 Fault-Tolerance in Traditional DBMSs

Fault-tolerance in traditional DBMSs and their different variants is usually achieved by running one or more replicas of the processing engine.

The simplest technique is for a DBMS to have a single backup server that waits, as standby, and takes over operations when the primary engine fails. This model is frequently called the process-pair model [25, 70]. There exist several variants of the process-pair model differing in runtime overhead and level of protection they provide. In the "cold-standby" variant, the primary periodically transmits a log of operations to the backup. The backup processes these operations asynchronously. With this approach, a failure of the primary can cause the loss of all operations not yet transmitted to the backup. Although logs of operations are commonly used, the primary can also send (incremental) checkpoints of its state. In the more expensive, "hot-standby", variant the primary and backup servers perform all operations synchronously, *i.e.*, they both perform every update before returning a result to the client. The process-pairs model is widely adopted by many existing DBMSs [40, 135].

The process-pair approach protects the system only against node failures, and with a single backup only one node can fail at any time for the system to remain available. To cope with network failures and larger numbers of node failures, a larger number of replicas spread across the wide-area network is needed. Fault-tolerance schemes with larger numbers of replicas can be categorized into one of two classes: eager or lazy replication. Eager replication favors consistency by having a majority of replicas perform every update as part of a single transaction [65, 67]. Eager replication, however, sacrifices availability because it forces minority partitions to block.

Lazy replication favors availability over consistency. With lazy replication all replicas process possibly conflicting updates even when disconnected and must later reconcile their state. They typically do so by applying system- or user-defined reconciliation rules [93, 169], such as preserving only the most recent version of a record [71]. It is unclear how one could define such rules for an SPE and reach a consistent state, because SPEs operate on a large transient state that depends on the exact order in which they processed their input tuples.

One class of lazy replication techniques uses tentative transactions during partitions and reprocesses transactions, possibly in a different order, during reconciliation [71, 163]. With these approaches, all replicas eventually have the same state, and that state corresponds to a single-node serializable execution. Our approach applies the ideas of tentative results to stream processing. We cannot, however, directly reuse the same concepts and techniques. Indeed, transactions are processed in isolation, while tuples in an SPE can all be part of a single query (there is no concept of a transaction). Tuples are aggregated and correlated rather than being processed in isolation. For this reason, we cannot reuse the notion of a tentative transaction. Instead, we introduce the notion of tentative and stable tuples. Additionally, we define availability as low-latency processing of the most recent input data. We must devise a protocol that maintains this property in spite of failures and reconciliations. Traditional lazy replication systems do not have such a constraint. Finally, in a distributed SPE, the output of processing nodes serves as input to other nodes and must also be eventually consistent.

Similar to our low-latency processing goal, there exist techniques for traditional query processing that enable trade-offs between result speed and consistency. The goal of most techniques is to produce meaningful partial results early during the execution of long-running queries. Previous work has addressed the problem of computing aggregates in an online fashion, producing results with increasingly greater precision [78, 159]. These schemes, however, assume that data is available locally and the query processor can chose the order in which it reads and processes tuples (for example, it can carefully sample inputs). This assumption does not hold for stream processing applications. Some approaches have been developed for querying data sources spread across the Internet. They process data as it arrives, focusing on materializing query outputs one row or even one cell at a time [134]. These schemes, however, require the query processor to buffer partial results until the query completes, and are thus unsuitable for unbounded data streams and long-duration failures.

Finally, some distributed query processing techniques offer users fine-grained control over the trade-off between query precision (*i.e.*, consistency) and performance (*i.e.*, resource utilization) [123, 124]. Users specify a consistency bound and the system strives to minimize resource utilization while ensuring the desired consistency. Users can also let the system optimize the consistency to within a given maximum resource utilization bound.

2.2 SPEs and Continuous Query Processors

SPEs and continuous query processors address the limitations of traditional DBMSs by proposing new architectures and investigating new techniques to provide low-latency processing of large numbers of continuous queries over high-rate data streams. Table 2.2 summarizes the main differences between DBMSs and SPEs.

2.2.1 Stream Processing Engines

We now present some of the main research projects in the area of stream processing and continuous query processing. Table 2.3 summarizes these main projects and their primary contributions. We present the projects in approximate chronological order of when they first started. After introducing these projects, we discuss specific efforts to achieve fault-tolerance in SPEs.

	DBMS	SPE
Data location	persistently stored	streaming
Data size	bounded	potentially unbounded input streams
Data model	unordered sets	append-only sequences
Operators	relational, can block	mostly windowed operators
Queries	primarily one-time	primarily continuous
Processing model	store, index, then process	process on arrival, storing optional

Table 2.2: Key differences between DBMSs and SPEs.

Tapestry [162] was one of the first systems to introduce the notion of stateful continuous queries. Tapestry is built on top of a regular DBMS, so it first stores and indexes data before including it in any continuous query. The advantage is that implementing Tapestry requires only small changes to a regular DBMS. The drawback is that Tapestry does not scale with the rate of the input streams nor the number of continuous queries. Tapestry focuses on append only data repositories such as databases of mail or bulletin board messages. The goal of the system is to enable users to issue continuous queries that identify some documents of interest and notify users when such documents are inserted into the repository. Tapestry supports only monotone queries: it returns all records that satisfied the query at any point in time. It never indicates that a record no longer satisfies a query. In Tapestry, users express queries in TQL, a language similar to SQL. Tapestry transforms user queries into incremental queries that it can execute periodically. Each execution produces a subset of the final results. Executing incremental queries is significantly more efficient than re-running complete queries.

Tribeca [157, 158] was an early SPE for network monitoring applications. Tribeca queries are expressed in the form of dataflows using a specific dataflow oriented query language. Queries can have only one input stream but can have multiple result streams. Indeed, Tribeca has limited support for join operations. It supports windowed aggregates and provides operators for splitting and merging streams (these operators are similar to a Group-By and a Union, respectively). Tribeca is thus a very application-specific engine.

NiagaraCQ [36] focuses on scalable continuous query processing over XML documents. The novelty in NiagaraCQ is its aim to support millions of simultaneous queries by grouping them together dynamically based on similarities in their structure. NiagaraCQ performs the grouping incrementally. As new queries arrive, they are added to existing groups. When needed, groups can be re-factored. Grouping queries improves performance because queries share computation, fewer execution plans exist and can thus all reside in memory, and when new data arrives, testing which queries need to be executed is performed at the group level. NiagaraCQ even groups queries that need to execute periodically with those that execute when a specific event occurs. For additional scalability, the system evaluates all queries incrementally. The NiagaraCQ also studies new query optimization techniques for queries over streams [173]. Traditional optimizers use information about the cardinality of relations. For streams, a more suitable metric is the rate of input streams, and the query optimizer should strive to maximize the output rate of the query.

TelegraphCQ [33] is one of the first engines to process data continuously as it arrives instead of storing it first. In TelegraphCQ, users express queries in SQL, except that each query has an extra clause defining the input windows over which the results should be computed. The emphasis of TelegraphCQ is on adaptive processing [15, 47, 108]. In

Project name	Main features
Tapestry [162]	One of first continuous query engines, based on a DBMS
Tribeca [157]	Early SPE effort geared solely toward network monitoring.
NiagaraCQ [36]	Continuous query processor for streaming XML documents. Focuses on scalability to millions of simultaneous queries.
TelegraphCQ [33]	Continuous and adaptive query processor. Also investigates support for querying historical stream data, as well as load management and fault-tolerance in parallel data flows.
STREAM [9]	SPE that enables queries over both streaming data and stored relations. Explores efficient processing at a single site.
Aurora [2]	SPE where users express queries directly by composing operators into data-flow diagrams. Explores efficient single-site processing optimizing quality-of-service on output streams.
Medusa [37]	Distributed and federated SPE built on top of Aurora.
Borealis [1]	Distributed SPE built on Aurora and Medusa. Focuses on distributed optimization, fault-tolerance, load management, tuple revisions, and integration with sensor networks.
Gigascope [44]	SPE for network monitoring. Queries are compiled into code. Focuses on techniques for high-rate stream processing.

Table 2.3: Continuous query processors and SPEs.

this system, query processing consists of routing tuples through query modules. Groups of query modules can be connected together with an Eddy. The Eddy intercepts tuples as they flow between modules and makes routing decisions based on changing conditions. This adaptivity enables better performance in the face of variable conditions during query execution. However, it comes at the price of a greater overhead because tuples need to hold lineage information and the Eddy must make routing decisions for each tuple. A second interesting aspect of TelegraphCQ is that it views stream processing as a join operation between a stream of data and a stream of queries. Queries and tuples are stored in separate state modules. When a new query arrives, its predicate is used to probe the data and vice versa [34]. This perspective enables TelegraphCQ to include old data in the result of a newly inserted continuous query. TelegraphCQ thus has good support for queries over both current and historical data [34, 35]. Finally, TelegraphCQ also addresses the problem of fault-tolerance and load management in stream processing [146, 147], although their focus is limited to parallel data flows. We discuss fault-tolerance further below.

The STREAM [9] project explores many aspects of stream processing: a new data model and query language for streams [11, 151], efficient single-site processing [16], resource management [18, 152], approximation and statistics computation over streams [13], and some distributed operation [19, 123]. STREAM also processes data as it arrives, without storing it. An important property of STREAM is its support for continuous queries over both streams and stored relations [11]. To achieve this, STREAM introduces three types of operators: stream-to-relation, relation-to-relation, and relation-to-stream. Stream-to-relation operators use window specifications to transform streams into relations that change with time. Relation-to-relation operators perform the bulk of the computations on snapshots of their input relations. Finally, relation-to-streams operators transform, when necessary, results back into a stream. Users express queries in CQL, a language based on SQL (and SQL-99 for the window specifications) with the addition of the three stream-to-relation operators. Mixing streams and relations enables interesting applications but it comes at the price of a somewhat greater complexity. Even simple filter queries over streams translate into a series of three operators: a stream-to-relation, relation-to-relation, and relation-back-to-stream operator. The distributed processing capabilities explored in the STREAM project [19, 123] are limited to the study of trade offs between resource utilization (bandwidth in particular) and precision of output results, when a set of distributed data sources contribute to a single aggregate computation.

The Aurora project [2, 14] also proposes a new data model for stream processing as well as a new architecture for the processing engine. In Aurora, users express queries directly by building boxes-and-arrows diagrams defining the way in which streams should be combined and transformed. In this model, boxes represent operators and arrows stand for streams. Boxes are based on relational operators. Diagrams are expressed using either a GUI or a textual description and must be directed acyclic graphs. Aurora adopted a dataflow interface instead of a SQL-style declarative query interface because a dataflow facilitates interspersion of pre-defined query operators with user-defined operators [21]. A drawback of the model is that clients can request any intermediate stream at any time, potentially limiting possible query optimizations. In terms of architecture, similarly to TelegraphCQ and STREAM, the Aurora engine processes data as it arrives, without storing it, so it can potentially support higher input rates than traditional DBMSs. The Aurora project also addresses the problem of efficient operator scheduling [30] and load shedding based on quality-of-service specifications [160]. Data tuples in Aurora are annotated with timestamps, used by the system to optimize the quality-of-service delivered on output streams.

The Medusa project [23, 37, 111] builds on Aurora and addresses distributed and federated operation. Medusa takes Aurora queries and distributes them across multiple processing nodes. Medusa also enables autonomous participants to compose their stream processing services into more complex end-to-end services and to perform processing on behalf of each other to balance their load. The Medusa approach to load management in a federated system is a contribution of this dissertation and we discuss it further in Chapter 6.

Borealis [1, 27] is a distributed SPE that builds on Aurora and Medusa. Borealis reuses the core data model and stream processing functionality of Aurora and the distribution capabilities of Medusa. Borealis explores advanced stream processing capabilities such as revision tuples that correct the value of earlier tuples, time travel where a running query diagram fragment can go back in time and reprocess some data, dynamic query modifications where operator parameters can change with the values of tuples on streams, various types of distributed query optimizations (operator placement and distributed load shedding), and fault-tolerant distributed stream processing. This last capability is a contribution of this dissertation and we discuss it further in Chapter 4.

Gigascope [43, 44] is an SPE designed specifically to support network monitoring applications. Users express queries in GSQL, a language based on a subset of SQL, but with support for user-defined operators. These operators enable Gigascope to leverage highly tuned functions that already exist in the network monitoring domain. Gigascope favors performance over flexibility. It is a compiled system. Queries are compiled into C and C++ modules, which are in turn compiled and linked into a runtime system. This choice can of course lead to better runtime performance but it makes it more difficult to add or delete queries at runtime. Gigascope has a two-tier architecture. Low-level queries, which run at the location of the data sources, perform aggressive data reduction (*e.g.*, aggregation, selection) before sending the data to high level query nodes that perform more complex queries on the lower-rate streams. Gigascope can be seen as a distributed system since each query node is also a process. Work in Gigascope focuses primarily on efficient stream processing [41, 89], and real-world deployments [43, 44].

2.2.2 Fault-Tolerance in SPEs

There has been only limited work on high availability in SPEs. Previous techniques focused primarily on fail-stop failures of processing nodes [83, 146]. These techniques either do not address network failures [83] or strictly favor consistency by requiring at least one fully connected copy of the query diagram to exist to continue processing at any time [146]. In contrast to previous work, we propose an approach that handles many types of system and network failures, giving applications the choice of trade-off between availability and consistency. With our approach, applications can request to see only correct results, but they can also choose to see early results quickly and correct results eventually.

Without focusing on fault-tolerance, some SPEs tolerate bounded disorder and delays on streams by using punctuations [166, 167, 168], heartbeats [151], or statically defined slacks [2]. An operator with a slack parameter accepts a pre-defined number of out-of-order tuples following a window boundary, before closing that window of computation [2]. Later out-of-order tuples are dropped. A punctuation is a “predicate on stream elements that must evaluate to false for every element following the punctuation” [168]. Punctuations are thus stream elements that unblock operators such as aggregate and join by allowing them to process all tuples that match the punctuation. If punctuations are delayed, operators block. STREAM’s Input Manager [151] uses periodic heartbeats with monotonically increasing timestamp values to sort tuples as they enter the SPE. It assumes, however, that heartbeats either always arrive within a bounded delay or can be assumed to occur within a pre-defined time-period. If a failure causes longer delays, late arriving tuples will miss their window of computation. These three approaches thus tolerate some disorder but they block or drop tuples when disorder or delay exceed expected bounds. Blocking reduces availability. Proceeding with missing tuples helps maintain availability but sacrifices consistency without even informing downstream nodes or client applications that failures are occurring.

2.3 Workflow Systems

Workflow management systems (WFMS) [6, 80, 82] share similarities with stream processing engines. In a workflow process, data travels through independent execution steps that together accomplish a business goal. This dataflow form of processing is similar to the way tuples flow through a query diagram. From the fault-tolerance perspective, a key difference between a WFMS and an SPE is that each activity in a workflow starts, processes its inputs, produces some outputs, and completes. In contrast, operators in an SPE process input tuples, update their transient states, and produce output tuples *continuously*. Operators in a query diagram are also significantly more fine-grained than tasks in a workflow. To achieve high availability, many WFMSs use a centralized storage server and commit the results of each execution step as it completes [90]. In case of failure, the committed information is sufficient to perform forward recovery. The storage servers themselves use one of the standard high-availability approaches: hot standby, cold standby, 1-safe, or 2-safe [90]. To abort a process instance, previously executed operations are undone using compensating operations [116]. For example, releasing one reserved seat on an airplane compensates the operation of reserving one seat. This fault-tolerance model is not suitable for SPEs because there is no notion of transactions, no moments during the execution of a query diagram fragment or even operator when inputs, outputs, and states should be updated and committed. The state is transient and the processing continuous. Also, persistently saving the state of each operator after it processes each tuple or even window of tuples would be prohibitive.

Instead of using a central storage server, more scalable approaches offer fault-tolerance in WFMSs by persistently logging messages and data exchanged by execution steps spread across processing nodes [7, 8]. Because the data transferred between execution steps can be large, some WFMSs use a separate data manager [8]. The data manager is a distributed and replicated DBMS and has thus the same properties as the eager or lazy replication schemes discussed above. Neither persistent queues nor a data manager are suitable for an SPE because they would require persistently storing tuples before letting downstream nodes process these tuples. The main goal of an SPE architecture is to process data as it arrives before or even without storing it, in order to keep up with high data input rates.

Finally, some WFMSs support disconnected operation by locking activities prior to disconnection [5]. This approach works well for planned disconnections, while our goal is to handle disconnections caused by failures. Also, an SPE processes data at a high rate. Any disconnected input causes the SPE to quickly run out of tuples to process on that input.

2.4 Publish/Subscribe Systems

Publish/subscribe [52] is a many-to-many communication paradigm that decouples the sources and destinations of events. Clients register their interest in an event or a pattern of events by specifying a topic (*e.g.*, [149]), an event type, or the desired content or properties of individual events (*e.g.*, Gryphon [24], Siena [31], and Java Message Service [76]). When a publisher generates an event that matches a client's interest, the client receives a notification of the event. The publish/subscribe middleware provides storage and management of subscriptions, accepts events from publishers, and delivers event notifications to clients. Publish/subscribe is thus a stateless message delivery system. An SPE can be used as a publish/subscribe system by transforming subscriptions into continuous queries composed solely of filters. More recent publish/subscribe systems, enable subscriptions to specify event correlations [20] or to be stateful [88]. In that respect, publish/subscribe systems are becoming increasingly similar to SPEs.

An approach developed for a stateful publish/subscribe systems tolerates failures and disorder by restricting all processing to "incremental monotonic transforms" [156]. The approach requires that each operator keep all windows of computation open at any time. The operator can thus produce, at any time and for any window, the current range for the final value. The range shrinks as new information arrives but bounds can remain arbitrarily large if the domain of the attribute is large. This approach thus works only for short failures because it causes the state of all operators to grow in proportion to the failure duration.

2.5 Sensor Networks

Work on SPEs addresses the problem of efficient and reliable processing of high-volume data streams *over a set of powerful servers.* Most projects view data sources, exactly as such: sources of data pre-configured to push tuples into the system either periodically or when events occur. Data sources are considered to be *outside of the system.* For many applications, such as network monitoring or financial services, this perspective is appropriate.

When data sources are sensors, however, performance can improve significantly if the system monitors and adjusts the sensor data rates dynamically. Potential gains are especially considerable when sensors are organized into an ad-hoc network rather than being all connected directly to the wired infrastructure. For this reason, in the past few years, significant efforts have been geared toward enabling stream processing in sensor networks [107, 106, 110, 180, 181].

The main similarity between efforts such as TinyDB [107, 106] or Cougar [180, 181], which run queries in sensor networks, and SPEs running on server nodes is their focus on processing streams of data. Both types of systems assume their inputs are unbounded and constrain processing to non-blocking windowed operators.

Several properties of sensor networks make stream processing in this environment different and particularly challenging. First, sensors usually run on batteries and, if used naively, have a very short lifetime, possibly as short as a couple of days. Second, sensors have limited resources: their processing speeds are low and their memories are small. Hence, they cannot perform arbitrary computation. Third, the communication model in a sensor network is different from that of a wired network. Sensors communicate over wireless and their radios allow them to communicate only with nearby nodes. Usually only a few sensors are within close proximity to a wired base-station and transmitting data from other sensors requires multi-hop communication: *i.e.*, sensors must forward data on behalf of other sensors. The communication inside the sensor network is also unreliable and low bandwidth.

The above challenges shape the work on stream processing in sensor networks. For example, query languages for sensor networks enable applications to adjust the trade-off between result accuracy, latency, and resource-utilization [107, 181]: *e.g.*, a query can specify a desired sampling rate or system life-time. Overall, query processing and optimization are forced to take into consideration all aspects of stream processing: when to collect readings and which readings to collect, what data to transmit and how to transmit it best, how to aggregate data as it travels through the network, what kind of topology to create for best data transmission, how to handle failed transmissions, etc. This holistic approach contrasts with SPEs running on powerful servers, which operate at a much higher level of abstraction and ignore low level data collection and transmission issues. In this dissertation, we focus only on stream processing in networks of powerful servers.

2.6 State Machine Replication

Because an SPE maintains state that it updates in response to the inputs it receives, replicating SPEs can be viewed as an instance of state machine replication [142]. Similarly to a state machine, our approach maintains consistency between replicas by ensuring that they process incoming messages in the same order. In contrast to a typical state machine, however, messages are aggregated rather than being processed atomically and in isolation from one another, so we do not have a direct relationship between input messages and output responses. More importantly, traditional techniques for fault-tolerance using state-machine replication strictly favor consistency over availability. Only replicas that belong to the majority partition continue processing at any time [32] and a voter (*e.g.*, the output client) combines the output of all replicas into a final result [142]. In contrast, our approach favors availability, allowing all replicas to continue processing even when some of their input streams are missing. The state machine approach is particularly well-suited to handle Byzantine failures, where a failed machine produces erroneous results rather than stopping. Our approach handles only crash failures of processing nodes, network failures, and network partitions.

2.7 Rollback Recovery

Approaches that reconcile state of a processing node using combinations of checkpoints, undo, and redo are well known [50, 72, 104, 163]. We adapt and use these techniques in the context of fault-tolerance and state reconciliation in an SPE and evaluate their overhead and performance in these environments.

2.8 Load Management

We now present an overview of related work on load management in cooperative and more importantly competitive environments. We also briefly discuss SLA management schemes.

2.8.1 Cooperative Load Management Algorithms

Most previous work on load management in an SPE addresses the problems of efficient or resource-constrained operator scheduling [16, 30] and load-shedding [18, 46, 152, 160]. A few schemes for distributed stream processing have examined the problem of allocating operators to processing nodes in a manner that minimizes resource utilization (network bandwidth in particular) or processing latency [4, 132]. More recently, some efforts have also started to address dynamic reallocations of operators between processing nodes in response to load variations [179]. All these techniques assume a collaborative environment and optimize either quality-of-service or a single system-wide utility.

In distributed systems in general, cooperative load and resource sharing has been widely studied (see, *e.g.*, [49, 64, 96, 109, 147]). Approaches most similar to the one that we propose produce optimal or near-optimal allocations using *gradient-descent*, where nodes exchange load or resources among themselves producing successively less costly allocations. An interesting result comes from Eager *et al.*, [49], who comparatively studied load balancing algorithms of varying degrees of complexity. They found that a simple threshold-based approach, where nodes with more than T queued tasks transfer newly arriving tasks to other nodes with queues shorter than T, performs almost as well as more complex approaches such as transferring tasks to least loaded nodes, although none of the approaches they investigated performed optimally. The contract-based approach that we propose is a type of threshold-based approach adapted for continuous tasks and heterogeneous environments with selfish agents. Our approach allows some bounded threshold variations (*i.e.*, bounded-price contracts) and assumes that different nodes use different thresholds (*i.e.*, nodes have different contracts). Similarly to Eager *et al.*, however, we find that, in many cases, even the fixed-threshold scheme leads to good overall load distribution.

In contrast to work on cooperative load management, the main challenge of our approach is its focus on competitive environments. In cooperative environments, unless they are faulty or malicious, nodes follow a pre-defined algorithm and truthfully reveal their current load conditions to other nodes in order to optimize system-wide performance. In a competitive environment, participants are directed by self-interest and adopt strategies that optimize their own utility. Participants may even choose not to collaborate at all.

2.8.2 Distributed Algorithmic Mechanism Design

With the increasing popularity of federated systems, recent approaches to load balancing and resource allocation have started to consider participant selfishness. The typical solution is to use techniques from microeconomics and game theory to create the right *incentives* for selfish participants to act in a way that benefits the whole system. These new schemes can be grouped into two broad areas: mechanism design and computational economies.

The goal of mechanism design (MD) [86, 127] is to implement a system-wide solution to a distributed optimization problem, where each selfish participant holds some private information that is a parameter to the global optimization. For example, each participant's processing cost is a parameter to a global load distribution problem. Participants are considered to be selfish, yet *rational*: they seek to maximize their utility computed as the difference between the payments they receive to perform some computation and the processing cost they incur. Participants may lie about their private information if this improves their utility. The mechanism defines the "rules of the game" that constrain the actions that participants can take. Direct-revelation mechanisms, are most widely studied. In these types of mechanisms, participants are asked to reveal their private information directly to a central entity that computes the optimal allocation and a vector of compensating payments. Algorithms that compute the load allocation and the payments to participants are designed to optimize participant utility when the latter reveal their private information truthfully.

In contrast to pure mechanism design, algorithmic mechanism design (AMD) [119, 122] additionally considers the computational complexity of mechanism implementations, usually at the expense of finding an optimal solution. Distributed algorithmic mechanism design (DAMD) [55, 57] focuses on distributed implementations of mechanisms, since in practice a central optimizer may not be implementable. Previous work on DAMD schemes includes BGP-based routing [55] and cost-sharing of multicast trees [56]. These schemes assume that participants correctly execute payment computations. In contrast, our load management mechanism is an example of a DAMD scheme that does not make any such assumption because it is based on bilateral contracts.

2.8.3 Economic-Based Load Management

Researchers have also proposed the use of economic principles and market models for developing complex distributed systems [115]. Computational economies have been developed in application areas such as distributed databases [154], concurrent applications [175], and grid computing [3, 29].

Most computational economies use a price-based model [29, 39, 59, 141, 154, 175], where consumers have different price to performance preferences, are allocated a budget, and pay resource providers. Different techniques can be used to price resources [29]. Frequently, resource providers hold auctions to determine the price and allocation of their resources [39, 59, 175]. Alternatively, resource providers bid for tasks [141, 154], or adjust their prices iteratively until demand matches supply [59]. These approaches to computational economies require participants to hold and participate in auctions for every load movement, thus inducing a large overhead. Variable load may also make prices vary greatly and lead to frequent reallocations [59]. If the cost of processing clusters of tasks is different from the cumulative cost of independent tasks, auctions become combinatorial [121, 128][1], complicating the allocation problem. If auctions are held by overloaded agents, underloaded agents have the choice to participate in one or many auctions simultaneously, leading to complex market clearance and exchange mechanisms [119]. We avoid these complexities by bounding the variability of runtime resource prices and serializing communications between partners. In contrast to our approach, computational economies also make it significantly more difficult for participants to offer different prices and service levels to different partners.

As an alternative to pricing, computational economies can also be based on *bartering* [29, 38, 59]. SHARP [62] is an infrastructure that enables peers to securely exchange tickets that provide access to resources. SHARP does not address the policies that define how the resources should be exchanged. Chun *et al.* [38] propose a computational economy based on SHARP. In their system, peers discover required resources at runtime and trade resource tickets. A ticket is a soft claim on resources and can be rejected resulting in zero value for the holder. In contrast, our pairwise agreements do not specify any resource amounts and peers pay each other only for the resources they actually use.

2.8.4 Resource Sharing in Peer-to-Peer Systems

In peer-to-peer systems, participants offer their resources to each other for free. Schemes to promote collaboration use reputation [97], accounting [174], auditing [120], or strategyproof computing [118] to eliminate "free-riders" who use resources without offering any in return. The challenge in peer-to-peer systems is that most interactions involve strangers. It is rare fore the same participants to interact with each other multiple times. Our work addresses a different environment. We focus on participants who want to control who they interact with and develop long-term relationships with each other.

2.8.5 Service Level Agreements

Service level agreements (SLAs) are widely used to enable interactions between autonomous participants [42, 81, 94, 101, 138, 171, 178]. Significant recent work addresses the problem of automatically monitoring and enforcing SLAs [94, 138]. These schemes enable SLAs to include specifications of the measurements that should be taken for monitoring purposes, the time and manner in which the measurements should be performed, and the party that will perform the measurements. The contract model that we propose fits well with such SLA infrastructures.

In this chapter, we presented background and related work. We discussed SPEs and their motivation. We presented schemes that enable fault-tolerance in traditional DBMSs, SPEs, and other related systems. We also presented an overview of techniques for load management in federated environments. In the next chapter, we focus more specifically on the design of the Borealis SPE, which we use as a basis for our implementation and evaluation.

[1]In a combinatorial auction, multiple items are sold concurrently. For each bidder, each subset of these items represents a different value.

Chapter 3

Borealis Architecture

Borealis [27] is a second-generation distributed stream processing engine developed as part of a collaboration between Brandeis University, Brown University, and MIT. In this chapter, we present the high level architecture of the Borealis engine to set the context for the main contributions of this dissertation. In Chapters 4 through 7 we discuss the detailed design and implementation of our fault-tolerance and load management mechanisms.

Borealis inherits and builds on two earlier stream processing projects: Aurora [14] and Medusa [111]. Aurora is a single-site SPE. Medusa is a distributed SPE built using Aurora as the single-site processor. Medusa takes Aurora queries and distributes them across multiple nodes. These nodes can all be under the control of one entity or can be organized as a loosely coupled federation under the control of different autonomous participants. Borealis inherits core stream processing functionality from Aurora and distribution capabilities from Medusa. Borealis does, however, modify and extend both systems with various features and mechanisms [1].

The rest of this chapter is organized as follows. In Section 3.1, we first present the Borealis stream data model. In Section 3.2, we discuss Borealis operators and query diagrams. In Section 3.3, we present the main system components and the system interface. We discuss the software architecture of each Borealis node in Section 3.4, and present the data flow during stream processing in Section 3.5.

3.1 Stream Data Model

The Borealis stream data model is based on the model introduced in Aurora, which defines a stream as an append-only sequence of data items. Data items are composed of attribute values and are called *tuples*. All tuples on the same stream have the same set of attributes. This set of attributes defines the type or *schema* of the stream. As an example, in an application that monitors the environmental conditions inside a building, suppose that sensors produce a stream of temperature measurements. A possible schema for the stream is (t.time, t.location, t.temp), where t.time is a timestamp field indicating the time when the measurement was taken, t.location is a string indicating the location of the sensor, and t.temp is an integer indicating the temperature value.

More specifically, Aurora uses the following definition of a tuple:

Definition 1 *(paraphrased from Abadi* et al.*, [2]) A* tuple *is a data item on a stream. A tuple takes the form:* $(\texttt{timestamp}, a_1, \ldots, a_m)$*, where* timestamp *is a timestamp value and*

$a_1, \ldots, a_m$ *are attribute values. All tuples on the same stream have the same* schema*:* i.e.*, they have the same set of attributes. The schema has the form* $(\mathtt{TS}, A_1, \ldots, A_m)$*.*

Aurora uses timestamps only for internal purposes such as quality-of-service (QoS) [2]. Timestamps can thus be considered as part of the *header* of the tuple, while the other attributes form the *data* part of the tuple.

Borealis extends the Aurora data model by introducing additional fields into tuple headers [1]. Our fault-tolerance scheme, DPC, uses only two of those fields: the tuple type (tuple_type), and the tuple identifier (tuple_id). The tuple_id uniquely identifies a tuple in a stream. Timestamps cannot serve as identifiers because they are not unique. The tuple_type enables the system to distinguish between different types of tuples, and DPC introduces a few new tuple types. DPC also extends the tuple headers with a separate timestamp called tuple_stime that serves to deterministically order tuples before processing them. We present the details of the DPC enhanced data model in Chapter 4. This model is based on the following definition:

Definition 1 (new)[1] *A* tuple *is a data item on a stream. A tuple takes the form:* $(\mathtt{tuple\_type}, \mathtt{tuple\_id}, \mathtt{tuple\_stime}, a_1, \ldots, a_m)$*, where* tuple_type *is the type of the tuple,* tuple_id *is the unique identifier of the tuple on the stream, and* tuple_stime *is a timestamp value used to determine the tuple processing order.* $a_1, \ldots, a_m$ *are attribute values. All tuples on the same stream have the same* schema*:* i.e.*, they have the same set of attributes. The schema has the form* $(\mathtt{TYPE}, \mathtt{ID}, \mathtt{STIME}, A_1, \ldots, A_m)$*.*

We also use the following definitions:

Definition 2 *A* data stream *is a uniquely named append-only sequence of tuples that all conform to the same pre-defined schema.*

A data stream typically originates at a single data source, although Borealis does not enforce this rule. A data source can produce multiple streams with the same schema, but it must assign a different name to each stream.

Definition 3 *A* data source *is any application, device, or operator that continuously produces tuples and pushes them to client applications or to other operators for additional processing.*

As we present DPC in the next chapter, we frequently need to refer to subsequences of tuples on a stream. We use the following definitions:

Definition 4 Prefix of a sequence of tuples*: Subsequence of tuples starting from the oldest tuple in the sequence and extending until some tuple, t, within the sequence.*

Definition 5 Suffix of a sequence of tuples*: Subsequence of tuples starting at some tuple, t, within the sequence and extending until the most recent tuple in the sequence.*

We often consider the entire stream as the sequence of tuples and talk about the prefix and suffix of the stream, referring to all tuples that either precede or succeed a given tuple on the stream. We also talk about the prefix and suffix of a sequence of tentative tuples. In this case, we consider the tentative tuples in isolation from the rest of the stream.

[1]This definition is a refinement of the Borealis data model definition originally presented by Abadi *et al.*, [1]. We ignored the header fields that DPC does not require (including the original Aurora timestamp field), we added the tuple_stime field, and we aligned the definition more closely with the original Aurora definition.

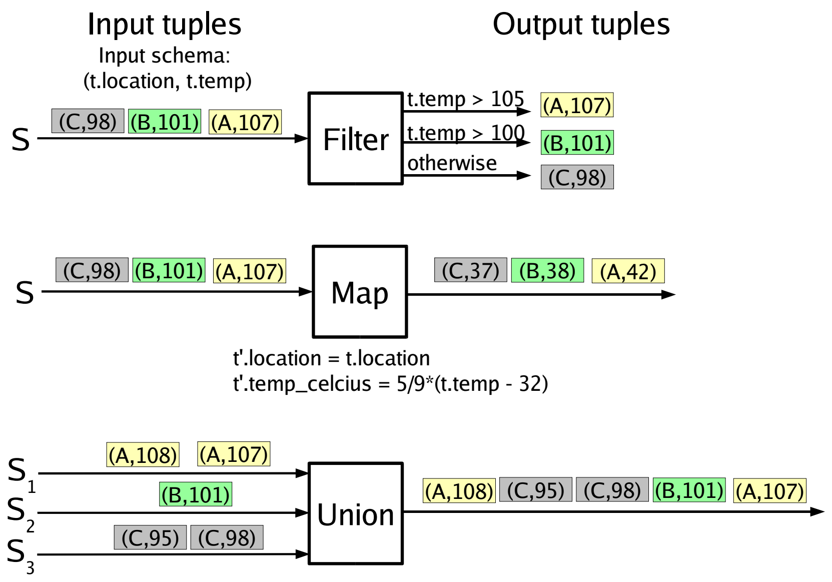


Figure 3-1: Sample outputs from stateless operators. Tuples shown on the right of each operator are the output tuples produced after processing the tuples shown on the left.

3.2 Operators

The goal of a stream processing engine is to filter, correlate, aggregate, and otherwise transform input streams to produce outputs of interest to applications, making the applications themselves easier to write. For instance, a stream processing engine could produce an alert when the combination of temperature and humidity inside a room goes outside a range of comfort. The application might then simply transform the alert into an email and send it to the appropriate party.

Inside an SPE, input streams are transformed into output streams by traversing a series of operators (*a.k.a.*, boxes). We now describe the core Borealis operators. A detailed description of these operators appears in Abadi *et al.* [2].

3.2.1 Stateless Operators

Stateless operators perform their computation on one tuple at a time without holding any state between tuples. There are three stateless operators in Borealis: *Filter*, *Map*, and *Union*.

Filter is the equivalent of a relational selection operator. Filter applies a predicate to every input tuple, and forwards tuples that satisfy the predicate on its output stream. For example, a Filter applied to a stream of temperature measurements may forward only tuples with a temperature value greater than some threshold (*e.g.*, "temperature $> 101°\texttt{F}$"). Tuples that do not satisfy the predicate are either dropped or forwarded on a second output stream. A Filter can have multiple predicates. In that case, the Filter acts as a case statement and propagates each tuple on the output stream corresponding to the first matched predicate.

A Map operator extends the Projection operator. Map transforms input tuples into output tuples by applying a set of functions on the tuple attributes. For example, Map could transform a stream of temperature readings expressed in Fahrenheit into a stream of Celsius temperature readings. As a more complex example, given an input tuple with two attributes, d and t, indicating a distance and a time period, Map could produce an output tuple with a single attribute indicating a speed, $v = \frac{d}{t}$.

A Union operator simply merges a set of input streams (all with the same schema) into a single output stream. Union merges tuples in arrival order without enforcing any order on output tuples. Output tuples can later be approximately sorted with a *BSort* operator. The latter operator maintains a buffer of a parameterizable size $n + 1$. Every time a new tuple arrives, BSort outputs the lowest-valued tuple from the buffer.

Figure 3-1 shows examples of executing Filter, Map, and Union on a set of input tuples. In the example, all input tuples have two attributes, a room number indicated with a letter and an integer temperature reading. The Filter has two predicates and a third output stream for tuples that match neither predicate. The Map converts Fahrenheit temperatures into Celsius. Union merges tuples in arrival order.

3.2.2 Stateful Operators

Rather than processing tuples in isolation, stateful operators perform computations over groups of input tuples. Borealis has a few stateful operators but we present only Join and Aggregate, the most fundamental and most frequently used stateful operators.

An aggregate operator computes an aggregate function such as average, maximum, or count. The function is computed over the values of one attribute of the input tuples (*e.g.*, produce the average temperature from a stream of temperature readings). Before applying the function, the aggregate operator can optionally partition the input stream using the values of one or more other attributes (*e.g.*, produce the average temperature for each room). The relational version of aggregate is typically *blocking*: the operator may have to wait to read all its input data before producing a result. This approach is not suitable for unbounded input streams. Instead, stream processing aggregates perform their computations over *windows* of data that move with time (*e.g.*, produce the average temperature every minute). These windows are defined over the values of one attribute of the input tuples, such as the time when the temperature measurement was taken. Both the window size and the amount by which the window slides are parameterizable. The operator does not keep any history from one window to the next, but windows can overlap. As an example, suppose an aggregate operator computes the average temperature in a room, and receives the following input. Each tuple has two attributes, the time of the measurement and the measured temperature.

Input: (1:16, 68), (1:21, 69), (1:25, 70), (1:29, 68), (1:35, 67), (1:41, 67), ...

The aggregate could perform this computation using many different window specifications. As a first example, the aggregate could use a landmark window [33], keeping a running average of the temperature starting from the landmark value, and producing an updated average for every input tuple. The following is a possible sequence of outputs, assuming 1:00 is the landmark.

Output 1: (1:16, 68), (1:21, 68.5), (1:25, 69), (1:29, 68.75), (1:35, 68.4), (1:41, 68.17), ...

Alternatively, the aggregate could use a sliding window [2]. Assuming a 10-minute-long

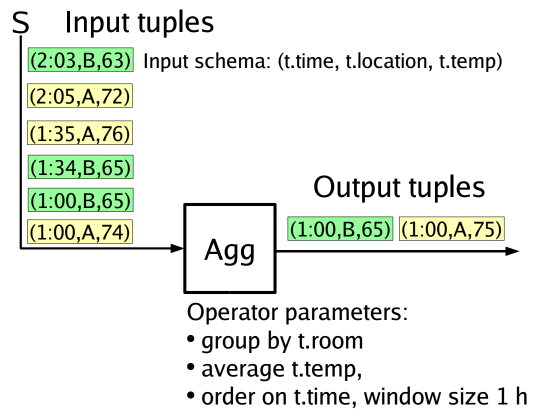


Figure 3-2: Sample output from an aggregate operator.

window advancing by 10 minutes, the aggregate could compute averages for windows [1:16,1:26), [1:26,1:36), etc. producing the following output:

Output 2: (1:16, 69), (1:26, 67.5), ...

In this example, the aggregate used the value of the first input tuple (1:16) to set the window boundaries. If the operator started from tuple (1:29, 68), the windows would have been [1:29,1:39), [1:39,1:49), etc. The operator could also round down the initial window boundary to the closest multiple of 10 minutes (the value of the advance), to make these boundaries independent of the tuple values. With this approach, the aggregate would have computed averages for windows [1:10,1:20), [1:20,1:30), [1:30,1:40), etc., producing the following output:

Output 3: (1:10, 68), (1:20, 69), (1:30, 67), ...

Borealis supports only sliding windows. In Borealis, windows can also be defined directly on a static number of input tuples (*e.g.*, produce an average temperature for every 60 measurements).

In general, window specifications render a stateful operator sensitive to the order of its input tuples. Each operator assumes that tuples arrive ordered on the attribute used in its window specification. The order then affects the state and output of the operator. For example, when an operator with a 10-minute sliding window computes the average temperature for 1:10 pm and receives a tuple with a measurement time of 1:21 pm, the operator *closes the window*, computes the average temperature for 1:10 pm, produces an output tuple, and *advances* the window to the next ten-minute interval. If a tuple with a measurement time of 1:09 pm arrives after the window closed, the tuple is dropped. Hence, applying an aggregate operator to the output of a Union produces approximate results, since the Union does not sort tuples while merging its input streams. Figure 3-2 illustrates a simple aggregate computation. In the example, tuples have three attributes: a measurement time, a location (room identified with a letter), and a temperature value. The aggregate produces separately for each room, the average temperature every hour.

Join is another stateful operator. Join has two input streams and, for every pair of

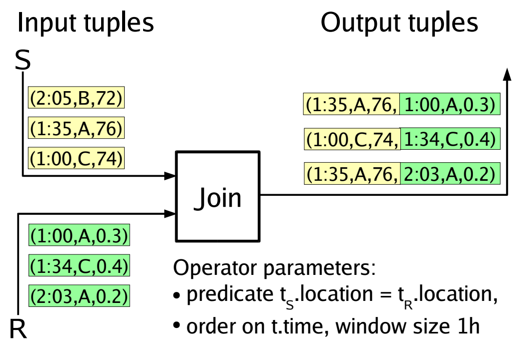


Figure 3-3: Sample output from a join operator.

input tuples (each tuple from a different stream), Join applies a predicate over the tuple attributes. When the predicate is satisfied, Join concatenates the two tuples and forwards the resulting tuple on its output stream. For example, a Join operator could concatenate a tuple carrying a temperature reading with a tuple carrying a humidity reading, every time the location of the two readings is the same. The relational Join operator accumulates *state that grows linearly with the size of its inputs*, matching every input tuple from one relation with every input tuple from the other relation. With unbounded streams, it is not possible to accumulate state continuously and match all tuples. Instead, the stream-based Join operator matches only tuples that fall *within the same window*. For two input streams, R and S, both with a `time` attribute, and a window size, w, Join matches tuples that satisfy $|r.\texttt{time} - s.\texttt{time}| \leq w$, although other window specifications are possible. Figure 3-3 illustrates a simple Join operation. In the example, tuples on stream S have three attributes: a measurement time, a measurement location (room identified with a letter), and a temperature value. Tuples on stream R have a humidity attribute instead of a temperature attribute. The Join matches temperature measurements with humidity measurements taken within one hour of each other in the same room.

In summary, stateful operators, in Borealis, perform their computations over windows of data. Because operators do not keep any history between windows, at any point in time, the *state of an operator consists of the current window boundaries and the set of tuples in the current window.* Operators can keep their state in aggregate form. For instance, the state of an average operator can be summarized with a sum and a number of tuples.

Stateful operators can have a *slack* parameter forcing them to wait for a few extra tuples before closing a window. Slack enables operators to support bounded disorder on their input streams. Operators can also have a timeout parameter. When set, a timeout forces an operator to produce a value and advance its window even when no new tuples arrived. Timeouts use the local time at the processing node. When a window of computation first opens, the operator starts a local timer. If the local timer expires before the window closes, the operator produces a value.

3.2.3 Persistent Operators

Most Borealis operators perform their computations only on transient state. Borealis has, however, two operators that enable access to persistent storage: a Read operator and an Update operator. Input tuples to these operators are SQL queries that either read or update

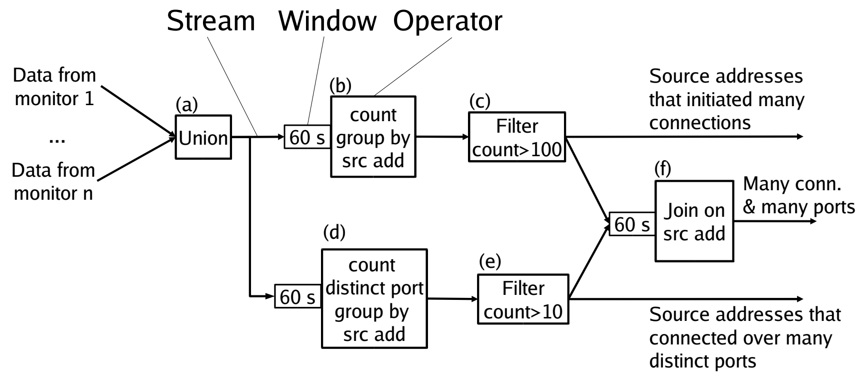


Figure 3-4: Example of a query diagram from the network monitoring application domain.

the state of a DBMS. The results of the SQL operations are output as a stream.

3.2.4 Query Diagrams

In Borealis, the application logic takes the form of a dataflow. To express queries over streams, users or applications compose operators together into a "boxes and arrows" diagram, called a *query diagram*. Most other stream-processing engines use SQL-style declarative query interfaces [10, 33, 43, 100] and the system converts the declarative query into a boxes-and-arrows query plan. Letting applications directly compose the dataflow facilitates intertwining query activity with used-defined stream processing operations. Figure 3-4 reproduces the query diagram example from Chapter 1.

3.3 System Architecture

Borealis is a distributed SPE composed of multiple physical machines, called *processing nodes* (or simply *nodes*), and a global catalog:

- The *global catalog* holds information about all components in the system. The information stored in the global catalog includes the set of processing nodes, the complete query diagram, the current assignment of operators to nodes (*i.e.*, the current diagram deployment), and other configuration information for processing nodes and other components. The global catalog is a single logical entity currently implemented as a central process. The catalog implementation, however, could also be distributed.

  The global catalog starts empty. Client applications communicate directly with the global catalog to change the system configuration, create the query diagram, define the initial deployment for that diagram, modify the diagram or deployment, and subscribe to streams. Subscribing to a stream enables a client to receive the tuples produced on that stream. The catalog methods expect arguments in XML. The choice of XML itself is not important, but allowing clients to specify textual descriptions of their requests makes it easy to develop applications. Most applications only require a developer to write two XML files: one file describing the query diagram and one specifying its deployment (*i.e.*, which groups of operators should run at each node).

- *Nodes* perform the actual stream processing. Each node runs a fragment of the query diagram and stores information about that fragment in a local catalog. The local catalog also holds information about other nodes that either send input streams into the node or receive locally produced outputs streams. Nodes collect statistics about their load conditions and processing performance (*e.g.*, processing latency). Each node also performs the tasks necessary to manage its load and ensure fault-tolerant stream processing. We present the software architecture of a Borealis node in Section 3.4.

- *Client applications*: Application developers can write and deploy applications that provide various services within a Borealis system. A client application can create and modify pieces of the query diagram, assign operators to processing nodes, and later request that operators move between nodes. A client application can also act as a data source or a data sink, producing or consuming data streams. A Borealis system comes by default with two client applications. One client provides a graphical user interface (GUI) for an administrator to modify the running query diagram graphically. The GUI generates the corresponding XML descriptions and transmits them to the global catalog. A second client, Monitor, is a monitoring tool that displays the current deployment and load conditions.

- *Data sources*: These are client applications that produce streams of data and send them to processing nodes.

3.4 Borealis Node Software Architecture

Each processing node runs a Borealis server whose major software components are shown in Figure 3-5. We now briefly present each one of these components.

The *Query Processor (QP)* forms the core piece where actual stream processing takes place. The QP is a single-site processor, composed of:

- An *Admin* interface for handling all incoming requests. These requests can modify the locally running query diagram fragment or ask the QP to move some operators to remote QPs. The Admin handles the detailed steps of these movements. Requests can also set-up or tear down subscriptions to locally produced streams.

- A *Local Catalog* that holds the current information about the local query diagram fragment. The local catalog includes information about local operators, streams, and subscriptions.

- The input streams are fed into the QP and results are pulled through the *DataPath* component that routes tuples to and from remote Borealis nodes and clients.

- *AuroraNode* is the actual local stream processing engine. AuroraNode receives input streams through the DataPath, processes these streams, and produces output streams that go back to the DataPath. To perform the processing, AuroraNode instantiates operators and schedules their execution. Each operator receives input tuples through its input queues (one per input stream) and produces output results on its output queues. AuroraNode also collects statistics about runtime performance, data rates on streams, CPU utilization of various operators, etc. These statistics are made available to other modules through the Admin interface.

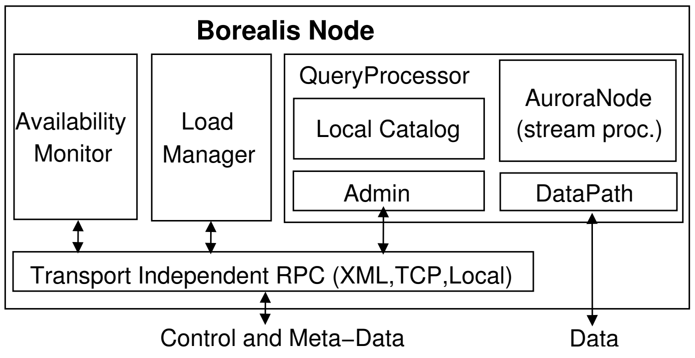


Figure 3-5: Software architecture of a Borealis node.

- The QP has a few additional components (such as a Consistency Manager) that enable fault-tolerant stream processing. We discuss their implementation in Chapters 4 and 5.

Other than the QP, a Borealis node has modules that communicate with their peers on other Borealis nodes to take collaborative actions:

- The *Availability Monitor* monitors the state of other Borealis nodes and notifies the query processor when any of these states change. The Availability Monitor is a generic monitoring component that we use as part of the fault-tolerance protocols.
- The *Load Manager* uses local load information as well as information from other Load Managers to improve load balance between nodes. We discuss the load management algorithms in Chapters 6 and 7.

Control messages between components and between nodes go through a transport independent Remote Procedure Call (RPC) layer that enables components to communicate in the same manner whether they are local or remote. Calls are automatically translated into local or remote messages. Communication is asynchronous. The RPC layer also seamlessly supports remote clients that use different RPC protocols to communicate with Borealis.

3.5 Data Flow

In Borealis, applications can modify the query diagram at runtime and can request operators to move between nodes. In this dissertation, however, we assume a static query diagram deployed over a set of processing nodes. We ignore the steps involved in deploying or modifying the query diagram and describe only the interactions between components during stream processing.

The data flow is the continuous flow of tuples from data sources to client applications going through the query diagram operators. The components involved in the data flow are thus the data sources, the processing nodes, and the client applications that subscribe to output streams. Figure 3-6 illustrates the data flow. (1) The data sources produce streams and push them to processing nodes. (2) The nodes process their input streams and produce streams of results that they push to other nodes for more processing, or (3) to client applications. When a stream goes from one node to another, the nodes are called *upstream and downstream neighbors.* More specifically, we use the following definition:

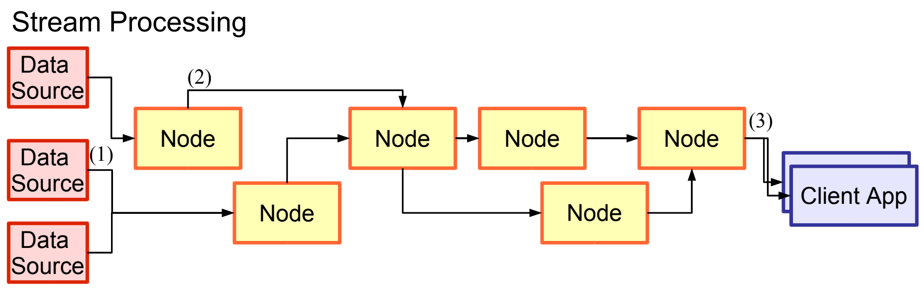


Figure 3-6: Data flow (stream flow) in a Borealis system.

Definition 6 Upstream/Downstream neighbors*: If a stream from a processing node, N_u, is pushed to node, N_d, where it may undergo further processing, then N_u is said to be an upstream neighbor of N_d, and N_d is a downstream neighbor of N_u.*

Applications that produce an input stream must open a TCP connection directly to one of the Borealis processing nodes and must send data in the appropriate binary format. Applications that receive an output stream must listen for incoming TCP connections and parse the incoming data. Borealis provides a library that facilitates these tasks.

When a query diagram is set up, each node that runs a query diagram fragment receives, from the global catalog, information about the location of its input streams. For each stream, this information includes the identifier (address and port) of the processing node that produces the data stream or receives it from a data source. Given this information, the node sends a subscription request to each node that produces one of the input streams. Upon receiving a subscription, an upstream node opens a TCP connection to the new downstream neighbor and pushes tuples as they become available. In Borealis, the data flow is thus push-based, which avoids having downstream nodes continuously poll for available data. Each node also buffers the most recent output tuples it produces. These buffers are necessary to achieve fault-tolerance, as we discuss in Chapter 4.

3.5.1 Summary

In this chapter, we presented the high-level architecture of the Borealis engine to set the context for our work. We presented the Borealis stream data model, the operators, the main system and software components, and the data flow during stream processing. In the next four chapters, we present our fault-tolerance and load-management schemes, their implementation in Borealis, and their evaluation.

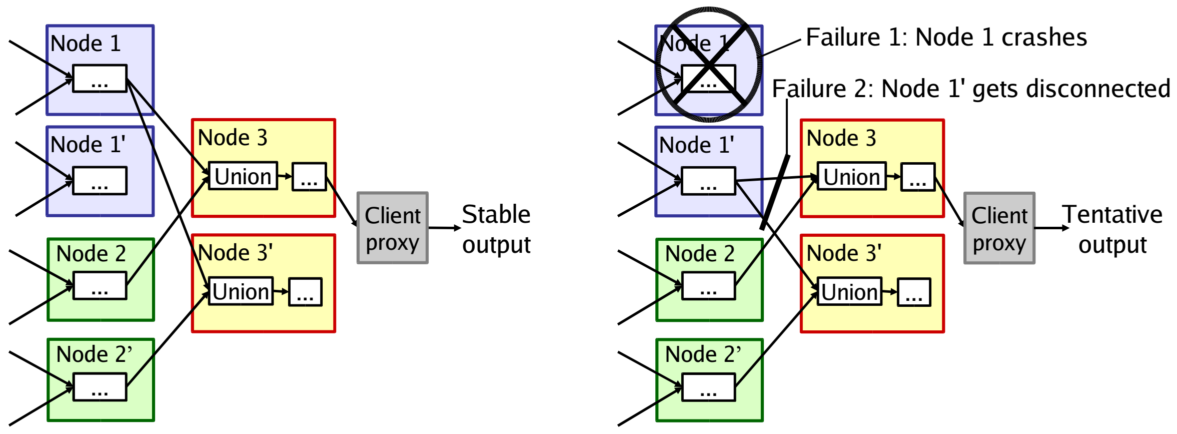
Chapter 4

Fault-Tolerance

In this chapter, we present *Delay, Process, and Correct* (DPC), an approach to fault-tolerant stream processing that enables a distributed SPE to cope with a variety of network and node failures in a manner that accommodates applications with different required trade-offs between availability and consistency. DPC addresses the problem of minimizing the number of tentative tuples while guaranteeing that the results corresponding to any new tuple are sent downstream within a specified time threshold, and that applications eventually receive the complete and correct output streams.

Figure 4-1 illustrates the output produced by a system using DPC. As shown in Figure 4-1(a), in DPC, all replicas continuously process data, and a node can use any replica of its upstream neighbor to get its input streams. In the absence of failures, the client receives a stable output. In the example, if Node 1 fails, then Node 3 (and its replica Node $3'$) can re-connect with Node $1'$, ensuring that the client application continues to receive the current and correct information. Output tuples continue to be labeled as "stable". The failure is masked by the system. If, however, Node $1'$ becomes disconnected while Node 1 is still down, the system will be unable to mask the failure. For a pre-defined time period, the system may suspend processing. The client will not receive any data. If the failure persists, however, processing of the inputs that remain available (streams produced by Node 2) must continue in order to ensure their availability. Processing thus restarts, but because some input streams are missing, output tuples are labeled as "tentative". Once the failure heals, the client application continues to receive results based on the most recent input data, still labeled as "tentative", while *receiving corrections to earlier tentative results.* This process continues until corrections catch up with most recent output results. At this point, the client application receives only the most recent and stable information.

The rest of this chapter is organized as follows. In Section 4.1, we present the details of our problem and design goals. We present an overview of the DPC protocol in Section 4.2. In Section 4.3, we discuss the high-level software architectural changes to an SPE required by DPC. In Section 4.4, we present the enhanced data model. The remaining sections present details of various aspects of DPC. Sections 4.5 through 4.7 present the algorithms and protocols that nodes follow in the absence of failures, when some of their inputs fail, and after failures heal, respectively. We address recovery of failed nodes separately in Section 4.8. Because DPC requires buffering tuples at various locations in the query diagram, we discuss algorithms to manage these buffers in Section 4.9. Finally, we discuss the impact of DPC on data sources and clients in Section 4.10. We present a few properties of DPC in Section 4.11.



(a) Distributed and replicated query diagram. (b) Failures causing tentative results.

Figure 4-1: Sample deployment with replication and failures.

4.1 Problem Definition

To define the problem, we first identify the fault-tolerance requirements of stream processing applications, emphasizing the similarities and differences in their availability and consistency goals. Second, we outline our detailed design goals: desired system properties and performance metrics. We then present the assumptions that we make about the system and the types of failures that we intend the system to tolerate. Finally, because some operator characteristics affect the properties and overhead of DPC, we present an operator classification. We start by considering application requirements.

Many stream processing applications monitor some ongoing phenomenon and require results to be produced with low latency. These applications often even value low latency processing more than result accuracy, although the maximum processing delay they can tolerate differs from one application to the next. There are several examples of such applications:

- Network monitoring: An administrator relies on network monitors to continuously observe the state of its network and detect anomalies such as possible intrusion attempts, worm propagation events, or simply overload situations. Administrators from different domains may even share some of their information streams to improve their detection capabilities [92]. In this application, even if data from only a subset of monitors is available, processing that data might suffice to identify at least some of the anomalies. Furthermore, low latency processing is critical to handle anomalous conditions in a timely manner: mitigate attacks, stop worm propagations as they occur, or react to overload situations quickly.

- Sensor-based environment monitoring. As they get cheaper and smaller, sensors are increasingly being deployed inside structures such as gas pipelines, air or water supplies, rooms in a building, etc. These sensors monitor the health of the infrastructures continuously. If a failure occurs and prevents data from a subset of sensors from being processed, continuing with the remaining information may help detect problems at least tentatively. For instance, the air temperature might be a little high in one part of a building. This may signify either a failure of the air conditioning system or simply that the sun is currently heating that side of the building. If there is no information for adjacent rooms, the system may tentatively declare that a failure occurred. A technician may be dispatched to make the final assessment. In contrast to network monitoring, this application may tolerate processing to be suspended for a few minutes, if this helps reduce the number of potential problems that later turn out to be benign.

- RFID-based equipment tracking: In this application, an RFID tag is attached to each piece of equipment and antennas are deployed through the facility. Antennas detect tags that appear in their vicinity and transmit the readings in the form of a stream, enabling the system to track the equipment as it moves. Failures may cause the system to miss some readings because information produced by a subset of antennas might be temporarily unavailable. Continuing with the remaining data, however, might suffice to satisfy a subset of queries about equipment locations. For instance, a user may be able to locate a needed piece of equipment quickly, even if no readings are currently available for one floor of the building. In this application, individual queries may have different levels of tolerance to processing delays.

While the above applications need to receive results quickly, they also need to receive the correct results eventually:

- Network monitoring: When processing data from only a subset of network monitors, some events might go undetected, and some aggregate results may be incorrect. Previously missed events are important, however, because they may still require some action, such are cleaning an infected machine. Corrected final values of aggregate results, such as per costumer bandwidth utilization, may also be needed for future analysis. The network administrator thus eventually needs to see the complete and correct output.

- Sensor-based environment monitoring: When a failure heals, some of the tentative alarms may be determined to have been false positives while other alarms were actual problems. Final correct values, especially if provided soon after a failure healed, may help re-assign technicians more efficiently.

- RFID-based equipment tracking. Eventually re-processing the complete and correct input data may help determine the accurate utilization information for each piece of equipment. Such information may, for instance, be required later for maintenance purposes.

Many other stream processing applications share similar requirements: *e.g.*, financial services, military applications, GPS-based traffic monitoring, etc. These applications need new results within a bounded maximum delay, and they eventually need the correct data. Of course, some stream processing applications require absolute consistency. Sensor-based patient monitoring is one example. DPC supports these applications as well since it allows applications to adjust the trade-off between availability and consistency. An application may thus set an infinite threshold indicating that inconsistent results should never be produced. An application may also drop all tentative results and wait for the stable ones.

4.1.1 Design Goals

Given the above application requirements, we now present the fault-tolerance goals, metrics, and desired properties of DPC.

Consistency Goal

In a replicated system, the strongest notion of consistency is *atomic consistency* [68, 105], also called *single-copy consistency* [140], *linearizable consistency* [68, 140] or *serializability* [71]. To provide atomic consistency, all accesses to a replicated object must appear as if they were executed at a single location following some serial execution. In an SPE, atomic consistency would ensure that clients receive only correct output tuples. As we discussed in Chapter 1, maintaining atomic consistency requires that the system sacrifices availability when certain types of failures, such as network partitions, occur [68].

To maintain availability, optimistic replication schemes often guarantee a weaker notion of consistency, called *eventual consistency* [140]. With eventual consistency, in order to provide availability, replicas can process client requests even if they do not know their final order yet, letting their states diverge. However, if all update operations stop, replicas must eventually converge to the same state. To achieve eventual consistency, all replicas of the same object must thus eventually process all operations in an equivalent order [140]. If operations are submitted continuously, eventual consistency requires that the prefix of operations in the final order grows monotonically over time at all replicas [140]. A data service that offers eventual consistency can be modeled as *eventually-serializable* (*i.e.*, maintaining requested operations "in a partial order that gravitates over time towards a total order" [58]).

Because many stream processing applications favor availability over consistency but need to receive the correct results eventually, our goal is for DPC to provide eventual consistency. In an SPE, the state of processing nodes is transient and the output stream continuous. We thus translate eventual consistency as requiring that all replicas of the same query diagram fragment eventually process the same input tuples in the same order, and that order should be one that could have been produced by a single processing node without failure.

Eventual consistency is a property of a replicated object. Responses to operations performed on the object do not have to be corrected after operations are reprocessed in their final order [58]. In an SPE, because the output of processing nodes serves as input to their downstream neighbors, we extend the notion of eventual consistency to include output streams. We require that each replica eventually processes the same input tuples in the same order and *produces the same output tuples in the same order.*

In summary, the first goal of DPC is:

Property 1 *Assuming sufficiently large buffers,*[1] *ensure* eventual consistency*.*

where eventual consistency is defined as:

Definition 7 *A replicated SPE maintains* eventual consistency *if all replicas of the same query diagram fragment eventually process the same input tuples in the same order and produce the same output tuples in the same order, and that order could have been produced by a single processing node without failure.*

[1]We discuss buffer management and long-duration failures in Section 4.9.

Once the final processing order of some operations is known, the operations are said to be *stable* [58]. We use the same definition for tuples. An input tuple is stable once its final processing order is known. When a replica processes stable input tuples, it produces stable output tuples because these output tuples have final values and appear in final order. Eventual consistency ensures that clients eventually receive stable versions of all results.

All intermediate results that are produced in order to provide availability, and are not stable, are called *tentative*. At any point in time, as a measure of inconsistency, we use, $\texttt{N}_{\texttt{tentative}}$, the *number of tentative tuples produced on all output streams* of a query diagram. $\texttt{N}_{\texttt{tentative}}$ may also be thought of as a (crude) substitute for the degree of divergence between replicas of the same query diagram when the set of input streams is not the same at the replicas. More specifically, we use the following definition:

Definition 8 *The* inconsistency *of a stream s,* $\texttt{N}_{\texttt{tentative}}(s)$*, is the number of tentative tuples produced on s since the last stable tuple. The inconsistency,* $\texttt{N}_{\texttt{tentative}}$*, of a query diagram is the sum of tentative tuples produced on all output streams of the query diagram since the last stable tuples produced.*

Availability Goal

The traditional definition of availability requires only that the system eventually produces a response for each request [68]. Availability may also measure the fraction of time that the system is operational and servicing requests (*i.e.*, the time between failures divided by the sum of the failure duration, recovery duration, and the time between failures) [72]. In an SPE, however, because client applications passively wait to receive output results, we define availability in terms of processing latency, where a low processing latency indicates a high level of availability.

To simplify our problem, we measure availability in terms of *incremental processing latency*. When an application submits a query to the system, DPC allows the application to specify a desired availability, X, as a maximum incremental processing latency that the application can tolerate on its output streams (the same threshold applies to all output streams within the query). For example, in a query diagram that takes 60 seconds to transform a set of input tuples into an output result, a client can request "no more than 30 seconds of added delay", and DPC should ensure that output results are produced within 90 seconds.

With the above definition, to determine if the system meets a given availability requirement, we only need to measure the *extra* buffering and delaying imposed on top of normal processing. We define $\texttt{Delay}_{\texttt{new}}$ as the maximum *incremental* processing latency for *any* output tuple and express the availability goal as $\texttt{Delay}_{\texttt{new}} < X$. With this definition, we express the second goal of DPC as:

Property 2 *DPC ensures that as long as some path of non-blocking operators*[2] *is available between one or more data sources and a client application, the client receives results within the desired availability requirement: the system ensures that* $\texttt{Delay}_{\texttt{new}} < X$*.*

As we discuss later, DPC divides X between processing nodes. To ensure Property 2, a node that experiences a failure on an input stream must switch to another replicas of its upstream neighbor, if such replica exists, within D time-units of arrival of the oldest

[2]We discuss blocking and non-blocking operators in Section 4.1.3.

unprocessed input tuples. If no replica exists, the node must process all tuples that are still available, within D time-units of their arrival, where D is the maximum incremental processing latency assigned to the node.

$\texttt{Delay}_{\texttt{new}}$ only measures the availability of result tuples that carry new information. We denote this set of tuples with `NewOutput`. These tuples exclude any stable result that correct a previously tentative one.

Even though we only measure incremental latencies, we can show how $\texttt{Delay}_{\texttt{new}}$ relates to normal processing latency. We define $\texttt{proc}(t)$ as the normal processing latency of an *output* tuple, t, in the absence of failure. $\texttt{proc}(t)$ is the difference between the time when the SPE produces t and the time when the oldest input tuple that contributed to the value of t entered the SPE. Given $\texttt{proc}(t)$ and the actual processing latency of a tuple, $\texttt{delay}(t)$, $\texttt{Delay}_{\texttt{new}} = \max_{t \in \texttt{NewOutput}} (\texttt{delay}(t) - \texttt{proc}(t))$.

Minimizing Inconsistency Goal

The main goal of DPC is to ensure that the system meets, if possible, a pre-defined availability level while ensuring eventual consistency. To maintain availability, the system may produce tentative tuples. To ensure eventual consistency, tentative tuples are later corrected with stable ones. Because it is expensive to correct earlier results in an SPE, we seek to minimize the number of tentative tuples. In the absence of failures, we would like replicas to remain mutually consistent, maintaining linearizable consistency, and ensuring that all results are stable. If a failure occurs, we would like the system to mask the failure, if possible, without introducing inconsistency. Finally, if a failure cannot be masked, we would like the system to minimize the number of tentative results. We summarize these requirements with the following two properties that we would like DPC to provide:

Property 3 *DPC favors stable results over tentative results when both are available.*

Property 4 *Among possible ways to achieve Properties 1 and 2, we seek methods that minimize* $\texttt{N}_{\texttt{tentative}}$.

4.1.2 Failure Model and Assumptions

Before presenting DPC, we identify the types of failures that we would like the system to support, and the fundamental and simplifying assumptions that we make about the system.

Fundamental Assumptions

We assume that the query diagram and its deployment (*i.e.*, the assignment of operators to processing nodes) are static. We also assume that the set of replicas for each processing node is static. We consider dynamic changes to the diagram or the deployment outside of the scope of this dissertation.

We assume that data sources (or proxies acting on their behalf) have loosely synchronized clocks and can timestamp the tuples they push into the system. Every time two or more streams are joined, unioned, or otherwise combined by an operator, DPC will delay tuples until timestamps match on all streams. The clocks at data sources must therefore be sufficiently synchronized to ensure these buffering delays are smaller than the maximum incremental processing latency, X. Similarly, when operators process tuples they assign timestamps to output tuples. Different algorithms are possible, but we assume that the timestamp assignment algorithm combined with the structure of the query diagram ensure tuple timestamps approximately match every time multiple streams are processed by the same operator. Once again, delaying tuples until their timestamp match should cause delays within X. These requirements are similar to those one would expect from an application-defined attribute serving in window specifications. Borealis applications already use such attributes.

We further assume that each processing node has sufficient resources (CPU, memory, and network bandwidth) to keep up with tuple input rates ensuring that queues do not form in the absence of failures. We assume that the network latency between any pair of nodes is small compared with the maximum incremental processing latency, X.

DPC handles crash failures [143] of processing nodes: when a processing node fails it halts without producing erroneous results, but the fact that the SPE crashed may not be detectable by other SPEs. In particular, a crash may not be distinguishable from a network failure that causes message losses or delays. DPC also handles network failures and network partitions. A network failure can cause message losses and delays, preventing any subset of nodes from communicating with one another. DPC treats long delays as failures. Because our system currently handles only static query diagram deployments, it also handles only transient failures. We assume that if a processing node fails it is not permanently removed from the system, but eventually restarts and rejoins the system with an empty state.[3]

At the beginning of this chapter, we assume that all tuples ever produced by a processing node are buffered. We revisit this assumption in Section 4.9, where we discuss buffer management and failures that last a long time. Except for data sources, we assume that buffers can be lost when a processing node fails. We assume that data sources and clients implement DPC, and that data sources can persistently log (or otherwise backup) the data they produce before pushing it into the system. With this assumption, even after failing and recovering, data sources can ensure that all replicas of the first processing node receive the same sequence of inputs. We discuss how this can be achieved in Section 4.10.

If tuples are logged forever, DPC can cope with the failure of all processing nodes. While all replicas of a processing node are unavailable, clients do not receive any data. After failed nodes recover with an empty state, they can reprocess all tuples logged upstream ensuring eventual consistency. If buffers are truncated, however, at any time, at least one replica of each processing node must hold the current consistent state. This state comprises the set of stable input tuples no longer buffered upstream and the set of stable output tuples not yet received by all replica of all downstream neighbors. Hence with buffer truncation, DPC handles the simultaneous crash failure of at most $R-1$ of the R replicas of each processing node, but we show that it handles both single failures and multiple overlapping (in time) failures.

DPC currently handles data source failures as follows. To maintain availability, when a data source fails, the system processes the remaining input streams as tentative. Once the data source recovers, the SPE reprocesses, as stable, all inputs including the previously missing ones if the data source produced any data during the failure and is able to replay that data. As in the case of node failures, DPC tolerates only transient failures of data sources. A permanent failure would be equivalent to a change in the query diagram.

[3]We discuss recovery of failed nodes in Section 4.8.

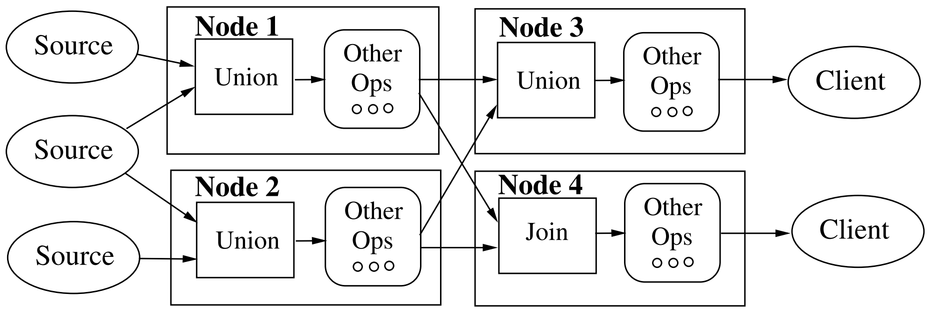


Figure 4-2: Query diagram composed of blocking (Join) and non-blocking (Union) operators.

Simplifying Assumptions

We assume that replicas communicate using a reliable, in-order protocol like TCP. With this assumption, nodes can rely on the fact that tuples transmitted from upstream arrive in the order in which they were produced. A downstream node can, for example, indicate with a single tuple identifier the exact data it has received so far.

In general, DPC is designed for a low level of replication and a low failure frequency.

4.1.3 Operator Classification

Several operator properties constrain the guarantees that DPC can provide and affect its overhead. In this section, we categorize operators along two axes: whether they block or not when some of their input streams are missing and how they update their state while processing input tuples.

Blocking and NonBlocking Operators

Stream processing operators perform their computations over windows of data that slide as new tuples arrive. However, some operators, such as *Join*, still block when some of their inputs are missing. Indeed, if no tuples arrive on one input stream, there are eventually no tuples to join with the most recent tuples on the other stream. In contrast, a Union is an example of a *non-blocking* operator because it can perform meaningful processing even when some of its input streams are missing. Of course, all operators block if all their inputs are missing.

Figure 4-2 illustrates the impact of blocking and non-blocking operators on fault-tolerance. The figure shows a query diagram deployed on four nodes. In this example, the failure of a data source does not prevent the system from processing the remaining streams. The failure of Nodes 1 or 2 does not block Node 3, but blocks Node 4. Only non-blocking operators maintain availability when some of their inputs are missing.

Determinism and Convergence

The manner in which operators update their state and produce output tuples in response to input tuples affects fault-tolerance algorithms. In this section, we present an operator taxonomy based on this property. This taxonomy is the same as that presented by Hwang *et al.* [83] except for our definition of determinism.

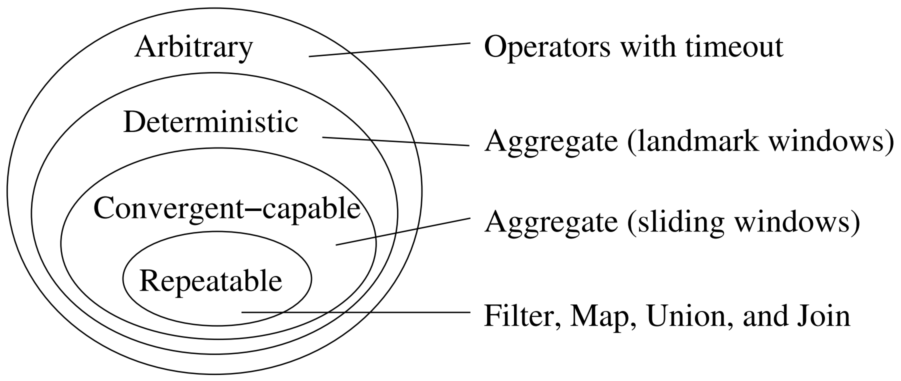

Arbitrary
Deterministic
Convergent−capable
Repeatable
Operators with timeout
Aggregate (landmark windows)
Aggregate (sliding windows)
Filter, Map, Union, and Join

Figure 4-3: Taxonomy of stream processing operators, with examples of operators in each category.

We distinguish four types of operators: *arbitrary* (including non-deterministic), *deterministic*, *convergent-capable*, and *repeatable*. Figure 4-3 depicts the containment relationship among these operator types and example operators in each category.

An operator is *deterministic* if it produces the same output stream every time it starts from the same initial state and processes the same sequence of input tuples. The sequence must not only define the order of tuples on each input stream separately, but also the absolute processing order of *all* input tuples. With this definition, there are only two possible causes of non-determinism in operators: dependence on execution time or input tuple arrival times (*e.g.*, operators with a timeout parameter produce an output tuple when no inputs arrive for a pre-defined time period) and use of randomization (*e.g.*, an operator that randomly drops tuples to shed load [160]). Currently, DPC supports only deterministic operators.

It is important to note that a query diagram composed of deterministic operators is not itself automatically deterministic. For the digram to be deterministic, we must ensure that all operators process input tuples in a deterministic order. In Section 4.5, we present a technique to achieve query determinism in general.

A deterministic operator is called *convergent-capable* if it can start processing input tuples from any point in time, yet it always converges to the same consistent state after processing sufficiently many input tuples (assuming, of course, that in all executions input tuples have the same values and arrive in the same order). Convergence enables a greater choice of techniques for a failed operator or node to rebuild a consistent state. We distinguish between convergent-capable and other deterministic operators as we present DPC.

Because the state of an operator is defined by the window of input tuples that it processes, the manner in which these windows move determine if the operator is convergent-capable or not. To ensure convergence, any input tuple must affect the state of the operator for a limited amount of time, and the operator must always converge to processing the same groups of input tuples. In Section 3.2.2, we discussed different types of window specifications. Typically, an operator with a sliding window is convergent-capable. For example, for a window size of 100, an advance of 10, and a first tuple 217, an aggregate may always converge to computing windows with boundaries that are multiples of 10: [210,310), [220,320), etc. The operator is not convergent capable, however, if all window boundaries are completely defined by the value of the first input tuple, *e.g.*, [217,317), [227,327), etc. Landmark windows may also prevent convergence because one end of the window may never move.

Finally, we classify Joins as convergent-capable because they typically align their window with respect to *each* input tuple.

A convergent-capable operator is also *repeatable* if it can re-start processing input tuples from an empty state and an earlier point in time, yet produces only tuples with the same values (identical duplicates) and in the same order. A necessary condition for an operator to be repeatable is for the operator to use at most one tuple from each input stream to produce an output tuple. If a sequence of multiple tuples contributes to an output tuple, then restarting the operator from the middle of that sequence may yield at least one different output tuple. Aggregates are thus not repeatable in general, whereas Filter (which simply drops tuples that do not match a given predicate) and Map (which transforms tuples by applying functions to their attributes) are repeatable as they have one input stream and process each tuple independently of others. Join (without timeout) is also repeatable if it aligns windows relative to the latest input tuple being processed. Repeatability affects the ease with which duplicate tuples can be eliminated if an operator restarts processing from an empty state and an earlier point in the stream. As such, this property affects fault-tolerance in general but our schemes do not need to distinguish between convergent-capable and repeatable operators.

In summary, two operator characteristics affect DPC the most. The blocking nature of some operators affects availability during failures. The convergent-capable nature of some operators enables greater flexibility in recovering failed nodes and may help reduce overhead as we discuss in Section 4.9.

4.2 DPC Overview

In this section, we present an overview of DPC by describing the expected high-level behavior of a processing node. The key idea behind DPC is to favor replica autonomy. Each replica is considered to be an independent processing node. Each node processes data. Each node also monitors the state of its input streams, monitors its availability, and manages its consistency, by implementing the DPC protocol that follows the state machine shown in Figure 4-4 with three states: STABLE, UPSTREAM_FAILURE (UP_FAILURE), and STABILIZATION.

As long as all upstream neighbors of a node are producing stable tuples, the node is in the STABLE state. In this state, the node processes tuples as they arrive and passes stable results to downstream neighbors. To maintain consistency between replicas that may receive inputs in different orders, we define a simple data-serializing operator that we call *SUnion*. Section 4.5 discusses the STABLE state and the SUnion operator.

If one input stream becomes unavailable or starts carrying tentative tuples, a node goes into the UP_FAILURE state, where it tries to find another stable source for the input stream. If no such source is available, the node has three choices to process the remaining available input tuples:

1. *Suspend* processing until the failure heals and one of the failed upstream neighbors starts producing stable data again.
2. *Delay* each new tuple for a short period of time before processing it.
3. *Process* each new available tuple without any delay.

The first option favors consistency. It does not produce any tentative tuples and may be used only for short failures given our goal to process new tuples with bounded delay. The latter two options both produce result tuples that are marked "tentative"; the difference

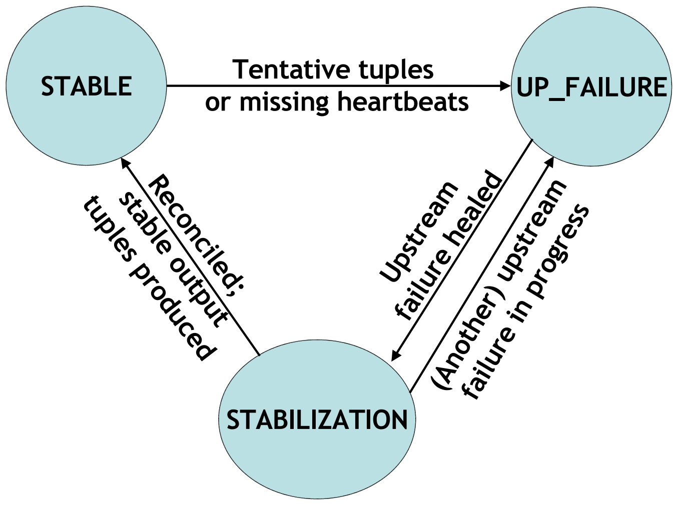

STABLE
Tentative tuples or missing heartbeats
UP_FAILURE
Reconciled; stable output tuples produced
Upstream failure healed
(Another) upstream failure in progress
STABILIZATION

Figure 4-4: DPC's state-machine.

between the options is in the latency of results and the number of tentative tuples produced. Section 4.6 discusses the UP_FAILURE state.

A failure *heals* when a previously unavailable upstream neighbor starts producing stable tuples again or when a node finds another replica of the upstream neighbor that can provide the stable version of the stream. Once a node receives the stable versions of all previously missing or tentative input tuples, it transitions into the STABILIZATION state. In this state, if the node processed any tentative tuples during UP_FAILURE it must now reconcile its state and stabilize its outputs (*i.e.*, correct the output tuples it produced during the failure). We explore two approaches for state reconciliation: a checkpoint/redo scheme and an undo/redo scheme. While reconciling, new input tuples are likely to continue to arrive. The node has the same three options mentioned above for processing these tuples: suspend, delay, or process without delay. DPC enables a node to reconcile its state and correct its outputs while ensuring that new tuples continue to be processed. We discuss the STABILIZATION state in Section 4.7. Once stabilization completes, the node transitions to the STABLE state if there are no other current failures, or back to the UP_FAILURE state otherwise.

At any time, a node can also fail and stop. When it recovers, the node restarts with an empty state and must rebuild a consistent state. We discuss node failures and recovery in Section 4.8.

DPC requires nodes to buffer some tuples. Because buffers cannot grow without bounds, nodes must communicate with one another to truncate these buffers periodically. We discuss buffer management in Section 4.9.

Hence, overall, DPC modifies the function of an SPE at three levels. DPC affects interactions between processing nodes (*e.g.*, switching between replicas of an upstream neighbor). DPC requires more extensive management of input and output tuples (*e.g.*, buffering and replaying tuples). Finally, DPC affects stream processing itself by requiring delaying and correcting tentative tuples as necessary. We present this functionality break down next, before presenting the enhanced data model and the details of DPC in the subsequent sections.

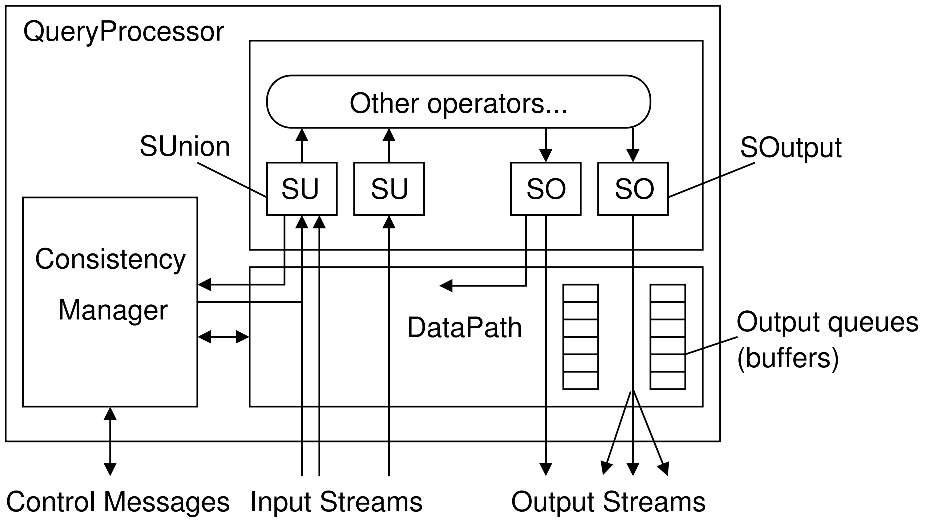


Figure 4-5: SPE software architecture extensions to support DPC. Arrows indicate communication between components (either control or data messages).

4.3 Extended Software Architecture

To run DPC, the software architecture of an SPE must be extended as illustrated in Figure 4-5. A new component, the *Consistency Manager*, is added to control all communication between processing nodes. The *Data Path*, which keeps track of and manages the data entering and exiting the node, is extended with extra monitoring and buffering capabilities. Finally, two new operators, *SUnion* and *SOutput*, are added to the query diagram to modifying the processing itself. We now present the main role of each component. We discuss these components further as we present the details of DPC in the next sections.

The Consistency Manager keeps a global perspective on the situation of the processing node within the system. It knows about the node's replicas, the upstream neighbors and their replicas, as well as the downstream neighbors and their replicas. The Consistency Manager therefore handles all interactions between processing nodes. It periodically requests state information from upstream neighbors and their replicas, and decides when to switch from one replica to another. Because of its global role, the Consistency Manager also makes decisions that affect the processing node as a whole. For example, it decides when to perform or suspend periodic checkpoints and when to enter the STABILIZATION state. As we discuss in the next chapter, in our prototype implementation, the Consistency Manager delegates some of the above functions to other modules in Borealis. In this chapter, we view the Consistency Manager as the logical entity that performs all the above tasks.

The Data Path establishes and monitors the input and output data streams. The Data Path knows only about the *current* upstream and downstream neighbors. For input streams, the Data Path keeps track of the input received and its origin, ensuring that no unwanted input tuples enter the SPE. For output streams, the Data Path ensures that each downstream client receives the information it needs, possibly replaying buffered data.

Finally, to enable fine grained control of stream processing, we introduce two new operators: SUnion and SOutput. SUnion ensures that replicas remain mutually consistent in the absence of failures, and manages trade-offs between availability and consistency. It buffers tuples when necessary, and delays or suspends their processing as needed. SUnion

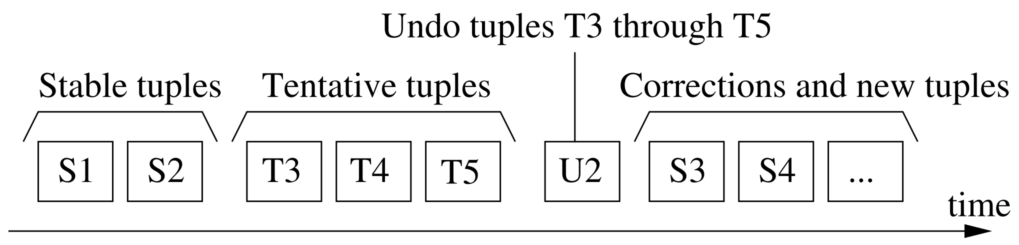


Figure 4-6: Example of using TENTATIVE and UNDO tuples. U2 indicates that all tuples following tuple with tuple_id 2 (S2 in this case) should be undone.

also participates in STABILIZATION. SOutput only monitors output streams, dropping possible duplicates during state reconciliation. Both SUnion and SOutput send signals to the Consistency Manager when interesting events occur (*e.g.*, the first tentative tuple is processed, the last correction tuple is sent downstream). DPC also requires small changes to all operators in the SPE. We discuss these changes in Chapter 5.

4.4 Enhanced Data Model

With DPC, nodes and applications must distinguish between stable and tentative results. Stable tuples produced after stabilization may override previous tentative ones, requiring a node to correctly process these amendments. To support tentative tuples and corrections, we extend the traditional stream data model by introducing new types of tuples.

As discussed in Chapter 3, traditionally, a stream is an append-only sequence of tuples of the form: $(t, a_1, \ldots, a_m)$, where t is a timestamp value and $a_1, \ldots, a_m$ are attribute values [2]. To accommodate our new tuple semantics, we adopt and extend the Borealis data model [1]. In Borealis, tuples take the following form (we ignore header fields that DPC does not use):

$$(\texttt{tuple\_type}, \texttt{tuple\_id}, \texttt{tuple\_stime}, a_1, \ldots, a_m).$$

1. **tuple_type** indicates the type of the tuple.
2. **tuple_id** uniquely identifies the tuple in the stream.
3. **tuple_stime** is a new tuple timestamp. [4]

Traditionally, all tuples are immutable stable insertions. We introduce two new types of tuples: TENTATIVE and UNDO. A tentative tuple is one that *results from processing a subset of inputs and may subsequently be amended with a stable version.* An UNDO tuple indicates that a suffix of tuples on a stream should be deleted and the associated state of any operators rolled back. As illustrated in Figure 4-6, the UNDO tuple indicates the suffix to delete with the tuple_id of the last tuple that should *not* be undone. Stable tuples that follow an UNDO replace the undone tentative tuples. Applications that do not tolerate inconsistency may thus simply drop TENTATIVE and UNDO tuples. We use a few additional tuple types in DPC but they do not fundamentally change the data model. Table 4.1 summarizes the new tuple types.

DPC also requires tuples to have a new timestamp field, tuple_stime for establishing a deterministic serial order for tuple processing as we discuss next.

[4]In Borealis, tuples also have a *separate* timestamp field used for quality of service purposes.

Tuple type	Description
Data streams	
INSERTION	Regular stable tuple.
TENTATIVE	Tuple that results from processing a subset of inputs and may later be corrected.
BOUNDARY	All following tuples will have a timestamp equal or greater to the one indicated. Only the tuple_type and tuple_stime need to be set on this tuple.
UNDO	Suffix of tuples should be deleted and the associated state should be rolled back. Except for its tuple_type, an UNDO tuple is a copy of the last tuple that should not be undone.
REC_DONE	Tuple that indicates the end of state reconciliation. Only the tuple_type needs to be set.
Control streams	Signals from SUnion or SOutput.
UP_FAILURE	Detected an upstream failure. Only the tuple_type needs to be set, although we also set the tuple_stime to the beginning of the failure.
REC_REQUEST	Received corrected input stream, ready to reconcile state. Only the tuple_type needs to be set.
REC_DONE	Same as above.

Table 4.1: New tuple types.

4.5 Stable State

The STABLE state defines a node's operations in the absence of failures. To minimize inconsistency and facilitate failure handling, DPC ensures that all replicas remain mutually consistent in the absence of failures: that they process the same input in the same order, go through the same internal computational states, and produce the same output in the same order. We now present the DPC mechanism to maintain such consistency. In the STABLE state, nodes must also detect failures of their input streams in order to transition to the UP_FAILURE state. We discuss failure detection second.

4.5.1 Serializing Input Tuples

We restrict DPC to deterministic operators. These are the operators that update their state and produce outputs only based on the values and the order of their input tuples (no timeouts, no randomization). If all operators are deterministic, to maintain replica consistency, DPC needs to ensure that replicas of the same operator process data in the same order; otherwise, the replicas will diverge even without failures.

Since we assume that nodes communicate using a reliable, in-order protocol like TCP, tuples never get re-ordered within a stream. Because each stream is also produced by a single data source, all replicas of an operator with a single input stream process their inputs in the same order without additional machinery. To ensure consistency, we thus only need a way to order tuples deterministically *across multiple input streams that feed the same operator* (*e.g.*, Union and Join).

If tuples on streams were always ordered on one of their attribute values and arrived at a constant rate, the problem of deterministically ordering them would be easy. Each operator could buffer tuples and process them in increasing attribute value, breaking ties in a deterministic fashion. The challenge is that tuples on streams are not necessarily sorted on any attribute and they may arrive at significantly different rates.

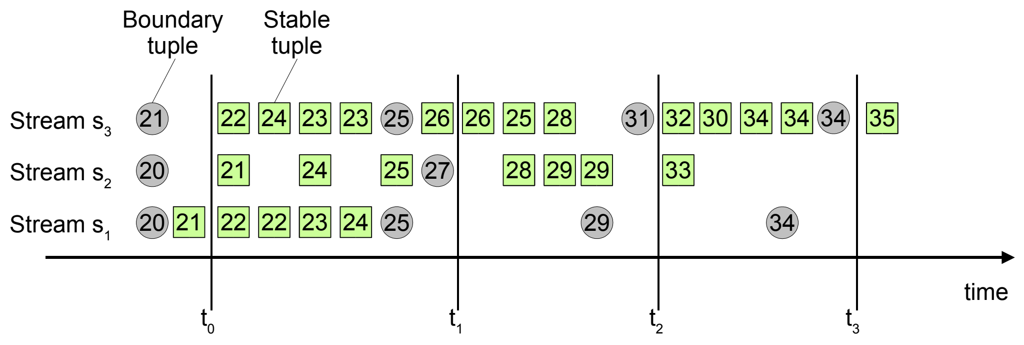


Figure 4-7: Boundaries enable sorting tuples deterministically across streams.

To compute an order without the overhead of inter-replica communication, we introduce additional *boundary tuples* on streams. Boundary tuples have tuple_type = BOUNDARY and serve the role of both punctuation tuples [166] and heartbeats [151]. The punctuation property of boundary tuples requires that no tuples with tuple_stime smaller than the boundary's tuple_stime appear after the boundary on the stream.[5] Boundary tuples enable an operator to deterministically order all (previously received) tuples with a lower tuple_stime because they ensure that the operator has received all such tuples. More specifically, an operator with i input streams can order deterministically all tuples that satisfy:

$$\texttt{tuple\_stime} < \min_{\forall i}(b_i), \tag{4.1}$$

where b_i is the tuple_stime value of the latest boundary tuple received on stream i. Figure 4-7 illustrates the approach for three streams. In the example, at time t_0, $\min(b_i) = 20$, and all tuples with tuple_stime values strictly below 20 can be sorted. Similarly, at time t_1, tuples below 25 can be sorted. At t_2 and t_3 only tuples below 27 can be sorted, since the last boundary seen on streams s_2 had value 27.

Requiring that clients make periodic, possibly null requests, in order for replicas to order requests deterministically is a standard approach used with state-machine replication [142]. Boundary tuples play the role of null requests in an SPE. Because boundary tuples are periodic, they ensure continuous and steady progress even in the absence of actual data on one or more input streams.

Rather than modifying operators to sort tuples before processing them, we introduce *SUnion*, a simple data-serializing operator that takes multiple streams as inputs and orders all tuples deterministically into a single sequence. Using a separate operator enables the sorting logic to be contained within a single operator. SUnion also manages trade-offs between availability and consistency by deciding when tuples should be processed, as we discuss in later sections. To enable greater flexibility in selecting the sort function and to manage availability-consistency trade-offs at a courser granularity, SUnion processes tuples at the granularity of fixed-size buckets. SUnion uses tuple_stime values to place tuples in buckets of statically defined sizes. It then uses boundary tuples to determine when a bucket is *stable* (no more tuples will ever arrive for that bucket), at which time it is safe to order tuples in this bucket and output them. SUnion's sort function typically orders tuples by

[5]If a data source cannot produce boundary tuples or set tuple_stime values, the first processing node to see the data can act as a proxy for the data source, setting tuple headers and producing boundary tuples (see Section 4.10).

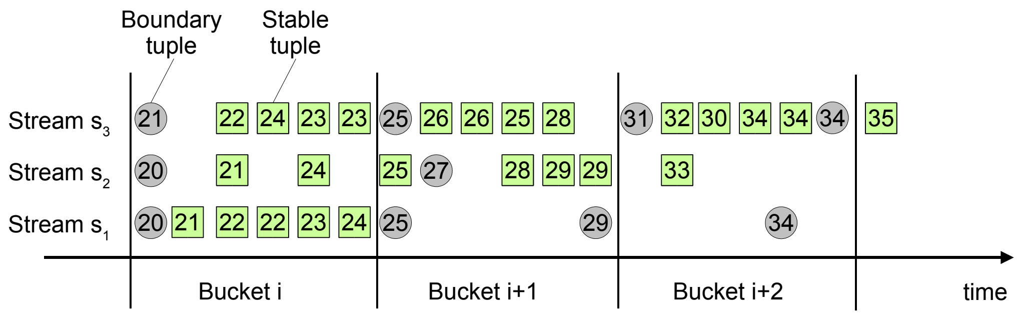


Figure 4-8: Example of organizing tuples into buckets with boundary interval $d = 5$**.** Only tuples in bucket i can be sorted and forwarded as stable.

increasing tuple_stime values, but other functions are possible. Figure 4-8 illustrates how buckets are used to determine what tuples can be processed. In this example, only tuples in bucket i can be sorted and forwarded as stable because boundary tuples with timestamps greater than the bucket boundary have arrived (they are in bucket $i + 1$). These boundary tuples make bucket i stable as they guarantee that no tuples are missing from the bucket. Neither bucket $i + 1$ nor $i + 2$ can be processed, since both buckets are missing boundary tuples, making it still possible that tuples will arrive for these buckets.

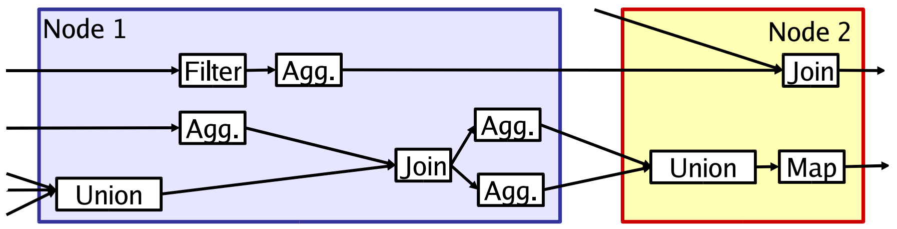
To maintain replica consistency, an SUnion operator must appear in front of every operator with more than one input stream. Figures 4-9(a) and (b) show a query diagram and its modified version, where all Union operators are replaced with SUnions and an SUnion is placed in front of every Join. Union and Join are the only operators that have more than one input stream.

As illustrated in Figure 4-9(b), SUnion operators may appear at any location in a query diagram. Therefore, all operators must deterministically set tuple_stime values on their output tuples and produce periodic boundary tuples with monotonically increasing tuple_stime values. Operators can ensure tuple_stime determinism by using, for example, tuple_stime values of input tuples to compute the tuple_stime value of the output tuples. To enable downstream operators to produce correct boundary tuples even in the absence of input data or even when tuples are not strictly ordered on their tuple_stime values, boundary tuples must propagate through the query diagram.

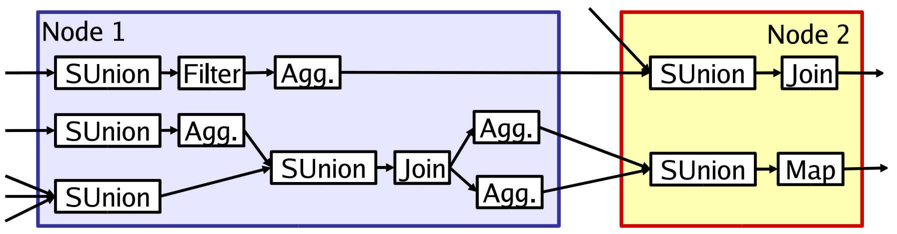
SUnion is similar to the Input Manager in STREAM [151], which sorts tuples by increasing timestamp order. SUnion, in contrast, ensures that replicas process tuples in the same order. SUnions need to appear in front of every operator with more than one input and not just on the inputs to the system. SUnion is also more general than the Input Manager. It can support different serialization functions but must break ties deterministically. More importantly, the Input Manager is not fault-tolerant. It assumes that delays are bounded and it uses that assumption to compute heartbeats if applications do not provide them. As we discuss below, we use the SUnion operator to provide the parameterizable availability/consistency trade-off.

4.5.2 Impact of tuple_stime Values Selection

A natural choice for tuple_stime is to use the local time at data sources. By synchronizing data source clocks, tuples will get processed approximately in the order in which they are



(a) Initial query diagram.



(b) Diagram modified to maintain consistency between replicas in the absence of failures and enable control over availability and consistency as failures occur.

Figure 4-9: SUnion placements in a query diagram.

produced. The Network Time Protocol (NTP) [165] is standard today and implemented on most computers and essentially all servers, synchronizing clocks to within 10 ms. Wall-clock time is not the only possible choice, though. In Borealis, any integer attribute can define the windows that delimit operator computations. When this is the case, operators also assume that input tuples are sorted on that attribute [2]. Using the same attribute for tuple_stime as for windows helps enforce the ordering requirement.

SUnion delays tuples because it buffers and sorts them. This delay depends on three properties of boundary tuples. First, the interval between boundary tuples with increasing tuple_stime values and the bucket size determine the basic buffering delay. Second, the basic delay further increases with disorder. The increase is bounded above by the maximum delay between a tuple with a tuple_stime, t, and a boundary tuple with a tuple_stime $> t$. Third, a bucket is stable only when boundary tuples with sufficiently high tuple_stime values appear on all streams input to the same SUnion. The maximum differences in tuple_stime values across these streams bounds the added delay. Because the query diagram typically assumes tuples are ordered on the attribute selected for tuple_stime, we can expect serialization delays to be small in practice. In particular, these delays should be significantly smaller than the maximum added processing delay, X.

In summary, the combination of SUnion operators and boundary tuples enables replicas of the same processing node to process tuples in the same order and remain mutually consistent. SUnions may increase processing latency because they buffer tuples before sorting and processing them, but this extra delay is small.

4.5.3 Detecting Failures

The heartbeat property of boundary tuples enables an SUnion to distinguish between the lack of data on a stream and a failure. When a failure occurs, an SUnion stops receiving boundary tuples on one or more input streams. SUnion may also start to receive tentative tuples. Because we do not want to propagate tentative tuples through the query diagram as soon as they arrive but rather delay them based on the current availability requirement, we need to place SUnion operators on each input stream, even when the stream is the only input to an operator. Figure 4-9(b) also illustrates the modification to the query diagram necessary to control availability and consistency when failures occur. In the diagram, an SUnion operator is added to each input stream.

In STABLE state, SUnion does most of the work. Other components, however, are also active. The Data Path buffers the most recent output tuples and the Consistency Manager monitors upstream neighbors and their replicas. Indeed, in addition to relying on boundary tuples to detect failures, the Consistency Manager periodically requests heartbeat responses from *each replica* of each upstream neighbor. By doing so, if an upstream neighbor fails, the Consistency Manager knows the states of all replicas of that neighbor and can switch to using another replica. Heartbeat responses not only indicate if a replica is reachable, but also include the states (STABLE, UP_FAILURE, FAILURE, or STABILIZATION) of its *output streams*. The Consistency Manager uses this detailed information when selecting a new upstream neighbor. Figure 4-10 shows the exact algorithm for monitoring input streams. The Consistency Manager periodically (every P_2 time units) sends a message to each replica of each upstream neighbor. Upon receiving a response from a replica, r, the Consistency Manager updates the last known state for each stream produced by r. If more than P_1 requests go unanswered, the Consistency Manager considers a replica as failed. The values of parameters, P_1 and P_2, trade off between the failure detection delay and the sensitivity of the monitoring algorithm to transient failures that may cause a node to drop one or more consecutive requests. In our implementation, we use $P_1 = 3$ and $P_2 = 100$ ms.

In addition to monitoring input streams, the Consistency Manager must also advertise the correct state of all *output streams*. To determine the state of an output stream, the SOutput operator of even the Data Path could simply send a message to the Consistency Manager every time they detect a state change on the stream. There are two drawbacks to this approach, though. First, from the moment a failure occurs on an input, it may take time for tentative tuples to propagate to the output. Second, by observing a stream, SOutput would not be able to distinguish between a partial failure that results in tentative tuples and a total failure that blocks the output. Instead, the Consistency Manager can compute the state of output streams directly from the state of input streams. We present two possible algorithms

Algorithm 1: The simplest algorithm is to equate the state of all output streams with the state of the processing node. We define the *state of the processing node* as follows. If one or more input streams are in the UP_FAILURE state, the node is in the UP_FAILURE state. The node remains in that state until it finds alternate replicas for the failed input streams or the failures heal and the node starts reconciling its state. During state reconciliation, the node is in the STABILIZATION state. After reconciling its state, if no new failures occurred, the node goes back to the STABLE state. Otherwise, it goes back to the UP_FAILURE state. In the rest of this document, for simplicity of presentation, we use this algorithm and always equate the state of output streams with the state of the node.

Algorithm 2: The above algorithm is simple but it only provides approximate in-

```
// Send heartbeat requests
PROCEDURE REQUEST_STATE:
Input:
    InputStreams: set of all input streams to the node.
    Replicas: upstream neighbors and their replicas.
      $\forall s \in$ InputStreams, Replicas$[s] = \{r_1, r_2, ...r_n\} \mid \forall i \in [1, n],\ r_i \in$ Nodes produces $s$.
Both Input and Output:
    Pending: number of unanswered requests sent to different nodes.
      $\forall r \in$ Nodes, Pending$[r] = i \mid i$ is the number of unanswered requests sent to $r$ by this node.
    InState: states of streams produced by different nodes.
      $\forall s \in$ InputStreams, $\forall r \in$ Replicas$[s]$,
      InState$[r][s] = x \mid x \in$ States is the state of stream $s$ produced by $r$.

01. foreach $r$ in Replicas
      // If more than $P_1$ pending messages, consider all streams as failed.
02.   if Pending$[r] > P_1$
03.     foreach $s$ in InputStreams $\mid r \in$ Replicas$[s]$
04.       InState$[r][s] \leftarrow$ FAILURE
05.   else
06.     send heartbeat request to r
07.     Pending$[r] \leftarrow$ Pending$[r] + 1$
08. sleep for $P_2$


// Receive a response, Response[r], from $r$
PROCEDURE RECEIVE_RESPONSE:
Input:
    Response$[r]$: response to heartbeat request from replica $r$
      $\forall s \in$ InputStreams $\mid r \in$ Replicas$[s]$
      Response$[r][s] = x \mid x \in$ States is the state of stream $s$ produced by $r$
Both Input and Output:
    Pending: number of unanswered requests sent to a replica
    InState: states of input streams produced by upstream neighbors and their replicas

01. Pending$[r] \leftarrow 0$
02. foreach $s$ in Response$[r]$
03.   InState$[r][s] \leftarrow$ Response$[r][s]$
```

Figure 4-10: Algorithm for monitoring the availability and consistency of input streams by monitoring all replicas of all upstream neighbors. Nodes denotes the set of all processing nodes in the system. States = {STABLE, UP_FAILURE, FAILURE, STABILIZATION}. The algorithm continuously updates InState, the state of input streams produced by each replica of each upstream neighbor. All data structures are local to each node.

formation about the state of output streams. In many cases, even though a node is in UP_FAILURE (as per the algorithm above), a subset of its outputs may be unaffected by the failure and may remain in STABLE state. We can also distinguish between the UP_FAILURE state, where the output stream may produce tentative tuples and the FAILURE state where the output stream is blocked or unreachable. This distinction improves the replica choices of downstream neighbors. Appendix A presents the algorithm for computing the detailed states of output streams.

4.6 Upstream Failure

In this section, we present the algorithms that each node uses to handle failures of its upstream neighbors in a manner that meets the application-required availability and ensured eventual consistency. There are two components to these algorithms: switching between replicas when failures occur, and suspending or delaying processing new tuples to reduce inconsistency.

4.6.1 Switching Upstream Neighbors

Because the Consistency Manager continuously monitors input streams, as soon as an upstream neighbor is no longer in the STABLE state (*i.e.*, it is either unreachable or experiencing a failure), the node can switch to another STABLE replica of that neighbor. By performing such a switch, the node can maintain both availability and consistency in spite of the failure. To enable the new upstream neighbor to continue sending data from the correct point in the stream, when a node switches replicas of an upstream neighbor, it indicates the *last stable tuple it received and whether it received tentative tuples after stable ones*. This information is provided by the Data Path to the Consistency Manager, which sends it to the new upstream neighbor in a subscribe message. The new upstream neighbor can then replay previously missing tuples or even correct previously tentative tuples. Data Paths at upstream neighbors must, of course, buffer their output tuples to perform such replays and corrections. We discuss buffer management in Section 4.9.

If the Consistency Manager is unable to find a STABLE replica to replace an upstream neighbor, it should at least try to connect to a replica in the UP_FAILURE state because processing tuples from such a replica helps the node maintain availability. If the Consistency Manager cannot find a replica in either STABLE or UP_FAILURE states, the node cannot maintain the availability of the missing stream. Connecting to a replica in the STABILIZATION state allows the node to at least start correcting data on the failed stream. Table 4.2 presents the algorithm that nodes use to switch upstream replicas. In this algorithm, a node simply prefers upstream neighbors in STABLE state over those in UP_FAILURE, which it prefers over all others. As illustrated in Figure 4-11, the result of these switches is that any replica can forward data streams to any downstream replica or client and the outputs of some replicas may not be used. We refine the switching algorithm further after presenting the STABILIZATION state in Section 4.7.3.

4.6.2 Managing Availability and Consistency during Failures

If a node fails to find an upstream replica that can provide the most recent stable tuples on a stream, the node can either suspend processing new tuples for the duration of the failure or it can continue processing the (possibly tentative) inputs that remain available.

State	Condition		Action
	`InState(Curr(s), s)`	$R =$ `Replicas(s)` $-$ `Curr(s)`	
1	`STABLE`	—	Remain in State 1
2	`! STABLE`	$\exists r \in R$, `InState`$(r, s) =$ `STABLE`	Unsubscribe from `Curr`(s) `Curr`$(s) \leftarrow r$ Subscribe to `Curr`(s) Go to state 1
3	`UP_FAILURE`	$\nexists r \in R$, `InState`$(r, s) =$ `STABLE`	Remain in State 3
4	$\in$ {`FAILURE`, `STABILIZATION`}	$\nexists r \in R$, `InState`$(r, s) =$ `STABLE` and $\exists r' \in R$, `InState`$(r', s) =$ `UP_FAILURE`	Unsubscribe from `Curr`(s) `Curr`$(s) \leftarrow r'$ Subscribe to `Curr`(s) Go to state 3
5	$\in$ {`FAILURE`, `STABILIZATION`}	$\nexists r \in R$, `InState`$(r, s) =$ `STABLE` and $\nexists r' \in R$, `InState`$(r', s) =$ `UP_FAILURE`	Reamin in state 5

Table 4.2: Algorithm for switching replicas of an upstream neighbor in order to maintain availability. Replicas(s) is the set of all replicas producing stream s. Curr(s) is the current upstream neighbor for s. The state of Curr(s) and the states of nodes in Replicas(s) define the conditions that can trigger an upstream neighbor switch. These switches in turn cause the state of Curr(s) to change. States also change as failures occur or heal.

Because suspending avoids inconsistency, it is the best approach for short-duration failures. For long-duration failures, the node must eventually process the most recent input tuples to ensure the required availability. If a node processes tentative tuples during a failure, either by receiving tentative tuples or proceeding with missing inputs, its state may start to diverge from the other replicas.

To minimize inconsistency while maintaining the required availability, a node can try to *continuously delay* new tuples up to its maximum pre-defined incremental processing latency. Delaying new tuples reduces the number of tentative tuples produced during the failure, but processing them as they arrive enables the node to delay or suspend processing new tuples later during STABILIZATION. We discuss trade-offs between processing tentative tuples with or without delay in Chapter 5. We now only present the mechanics of controlling availability and consistency.

SUnion operators manage the trade-off between availability and consistency by suspending or continuously delaying tuples when failures occur. More specifically, the client application specifies a total incremental processing latency, X, which we divide among SUnion operators.[6] We assign a maximum latency, D, to each *input stream of each SUnion operator*. At runtime, when it receives the first tuple for a given bucket and stream combination, SUnion starts a timer. If the timer expires before boundary tuples ensure that the complete bucket is stable, SUnion serializes the available tuples, labeling them as TENTATIVE, and buffering them in preparation for future reconciliation. In the example from Figure 4-7, if the boundary for stream s_2 does not arrive within D time-units of the moment when the first tuple entered bucket $i+1$, then SUnion will forward the remaining tuples

[6]We analyze how to divide X between SUnion operators in Chapter 5.

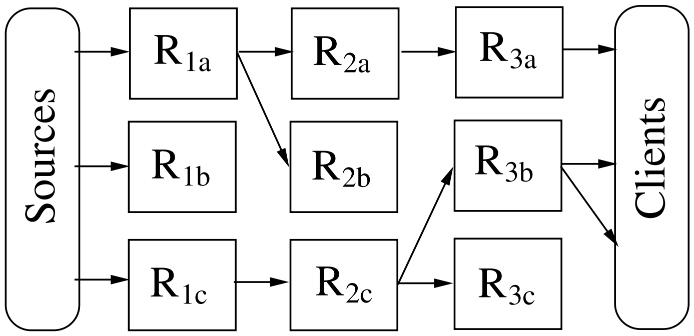


Figure 4-11: Example of a distributed and replicated SPE. R_{ij} is the j'th replica of processing node i.

from that bucket as tentative. It will also buffer these tuples in preparation for future reconciliation. In the UP_FAILURE state, SUnions and operators should continue processing and producing boundary tuples, but these tuples should also be labeled as TENTATIVE. Tentative boundaries can help a downstream SUnion determine how soon it can process a tentative bucket.

To ensure that a node has time to detect and react to an upstream failure before SUnion starts processing inputs as tentative, the parameters of the monitoring algorithm in Figure 4-10 must satisfy: $P_2 * P_1 << D$, where P_2 is the interval between requests for state information and P_1 is the number of consecutive requests that must go unanswered before the downstream node declares the upstream neighbor as failed.

In summary, when an input stream fails, a node transitions to the UP_FAILURE state and uses previously collected information to find and continue processing from the best available replica of the failed stream. If no replica in the STABLE state exists, the node must continue processing the (possibly tentative) inputs that remain available in order to maintain their low processing latency. SUnions can, however, suspend or delay most recent tuples to minimize inconsistency. After the failure heals, the node transition into the STABILIZATION state, which we discuss next.

4.7 Stabilization

In this section, we present the algorithms that each node follows after a failure heals. For simplicity, we focus on recovery from a single failure; DPC handles multiple simultaneous failures as well as failures during recovery. We discuss these more complex failure scenarios in Section 4.11.

An SUnion determines that a failure has healed when it receives corrections to previously tentative tuples or a replay of previously missing inputs. Corrections arrive in the form of a single UNDO tuple followed by stable tuples. When it receives an UNDO tuple, SUnion stabilizes the corresponding input stream by replacing, in its buffer, undone tuples with their stable counterparts. Once at least one bucket worth of input tuples is corrected and stable, SUnion notifies the Consistency Manager that it is ready for state reconciliation. To avoid possibly correcting tentative tuples with other tentative tuples when only one of several streams has healed, the Consistency Manager wait for notifications from all previously failed input SUnions before entering the STABILIZATION state.

In the STABILIZATION state, to ensure eventual consistency, a node that processed tentative tuples must reconcile its state and stabilize its outputs, by replacing previously tentative output tuples with stable tuples. Stabilizing output streams allows downstream neighbors to reconcile their states in turn. We present state reconciliation and output stabilization techniques in this section. We also describe how each node maintains availability while reconciling its state.

4.7.1 Reconciling the Node's State

Because no replica may have the correct state after a failure and because the state of a node depends on the exact sequence of tuples it processed, we propose that a node reconcile its state by reverting to a pre-failure state and reprocessing all input tuples that arrived since then. To revert to an earlier state, we explore two approaches: reverting to a checkpointed state, or undoing the effects of tentative tuples. Both approaches require that the node suspend processing new input tuples while reconciling its state. For clarity, we only present the checkpoint/redo scheme in this chapter. We present the undo-based technique in Appendix B.

With checkpoint/redo reconciliation, a node periodically checkpoints the state of its query diagram when it is in STABLE state. The Consistency Manager determines when the node should checkpoint its state. To perform a checkpoint, a node suspends processing any tuples and iterates through all operators and intermediate queues making copies of their states. To enable this approach, operators must thus be extended with a method that takes a snapshot of their state. Checkpoints could be optimized to copy only differences in states since the last checkpoint.

The Consistency Manager determines when the node should reconcile its state. To reconcile its state, a node restarts from the last checkpoint before the failure and reprocesses all tuples received since then. To re-initialize its state from the checkpoint, a node suspends processing all tuples and iterates through all operators and intermediate queues re-initializing their states from the checkpointed state. Operators must thus be modified to include a method to re-initialize their state from a checkpoint. After re-initializing its state, the node reprocesses all input tuples received since the checkpoint. To enable these replays, SUnions on input streams must thus buffer input tuples between checkpoints. They must buffer all tuples that arrive before, during, and after the failure. When a checkpoint occurs, however, SUnion operators truncate all buckets that were processed before the checkpoint.

4.7.2 Stabilizing Output Streams

Independent of the approach chosen to reconcile the state, a node stabilizes each output stream by deleting a suffix of the stream with a single UNDO tuple and forwarding corrections in the form of stable tuples. A node must typically undo and correct all tentative tuples produced during a failure. As a possible optimization, a node could first determine if a prefix of tentative tuples was unaffected by the failure and need not be corrected. However, as we discuss in Section 4.9, planning to always correct all tentative tuples enables significant savings in buffer space.

With undo/redo state reconciliation, operators process and produce UNDO tuples, which simply propagate to output streams. To generate an UNDO tuple with checkpoint/redo, we introduce a new "serializing output" operator, *SOutput*, that we place on each output stream that crosses a node boundary. At runtime, SOutput acts as a pass-through filter that also remembers the last stable tuple it produced. After restarting from a checkpoint, SOutput drops duplicate stable tuples and produces the UNDO tuple. The Data Path could monitor the output stream and produce the appropriate UNDO tuple instead of SOutput. SOutput, however, nicely encapsulates the small state machine needed to properly delete duplicates under various, possibly complex, failure and recovery scenarios.

Before forwarding the UNDO downstream, the Data Path must look up, for each downstream neighbor, the identifier of the last stable tuple the neighbor received. Indeed, a node can switch upstream neighbors any time before or *during* a failure. An upstream neighbor must therefore adjust which tuples it is correcting for each downstream neighbor. Stabilizing an output stream thus occurs in two steps. First, the Data Path corrects tuples in the output buffers such that these buffers contain only stable tuples. Second, the Data Path looks up the last stable tuple received by each downstream neighbor and sends the appropriate UNDO and corrections.

Stabilization completes when either the node reprocesses all previously tentative input tuples *and* catches up with normal execution (*i.e.*, it clears its queues) or when another failure occurs and the node goes back into UP_FAILURE. Once stabilization completes, a node transmits a REC_DONE tuple to its downstream neighbors. SUnions placed on input streams produce REC_DONE tuples once they clear their input buffers. REC_DONE tuples propagate through the query diagram to SOutput operators, which forward these tuples downstream. To avoid duplicate REC_DONE tuples, each SUnion operator placed in the middle of the diagram waits for a REC_DONE on all its inputs before forwarding a single such tuple downstream.

The above algorithms for state reconciliation and output stream stabilization enable a node to ensure eventual consistency: the node's state becomes once again consistent and downstream neighbors receive the complete and correct output streams. The problem, however, is that stabilization takes time and while reconciling its state and correcting its output, the node is not processing new input tuples. A long stabilization may cause the system to break the availability requirement. We discuss how to address this problem next.

4.7.3 Processing New Tuples During Reconciliation

During stabilization, we have, once again, a trade-off between availability and consistency. Suspending new tuples during state reconciliation reduces the number of tentative tuples but may eventually violate the availability requirement if state reconciliation takes a long time. To maintain availability when reconciling after a long-duration failure, a node must produce both corrected stable tuples and new tentative tuples. Hence, DPC uses two replicas of a query diagram: one replica remains in UP_FAILURE state and continues processing new input tuples, while the other replica performs the reconciliation. A node could run both versions locally, but DPC *already uses replication* and downstream clients must know about all existing replicas to properly manage their availability and consistency. Therefore, to avoid duplicating the number of replicas, we propose that existing replicas use each other as the two versions, when ever possible.

To enable a pair of replicas to decide which one should reconcile its state while the other remains available, we propose that Consistency Managers run the following simple inter-replica communication protocol. Before entering STABILIZATION, the Consistency Manager sends a message to one of its randomly selected replicas. The message requests authorization to enter the STABILIZATION state. If the partner grants the authorization, it promises that it will not enter STABILIZATION itself. Upon receiving authorization to

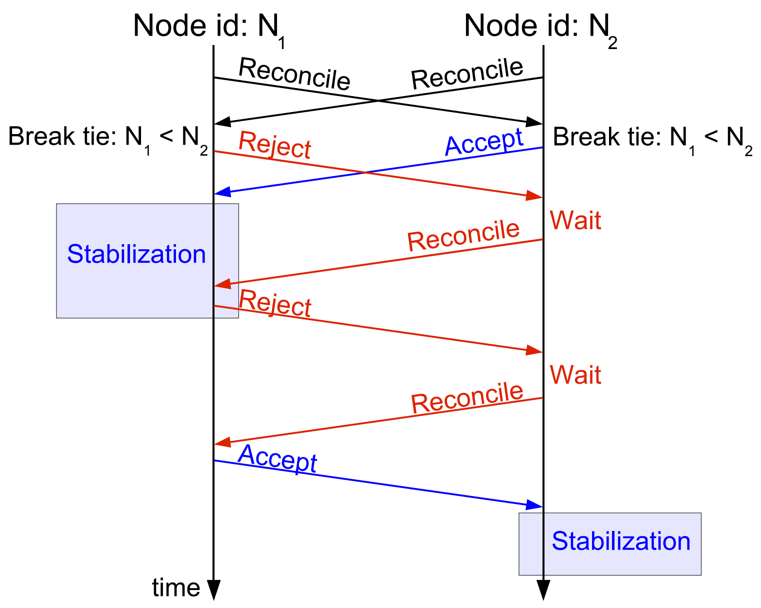


Figure 4-12: Inter-replica communication protocol to stagger replica stabilizations.

reconcile, the Consistency Manager triggers state reconciliation. However, if the replica rejects the authorization, the node cannot enter the STABILIZATION state. Instead, the Consistency Manager waits for a short time-period and tries to request authorization again, possibly communicating with a different replica. A Consistency Manager always accepts reconciliation requests from its replicas except if it is already in the STABILIZATION state or it needs to reconcile its own state and its identifier is lower than that of the requesting node. The latter condition is a simple tie breaker when multiple nodes need to reconcile their states at the same time. Figure 4-12 illustrates the communication taking place between two replicas that both need to reconcile their states.

Processing new tuples during STABILIZATION increases the number of tentative tuples compared to suspending their processing. A node may still attempt to reduce inconsistency by delaying new tuples as much as possible. In Chapter 5, we compare the alternatives of suspending, delaying, or processing new tuples without delay during STABILIZATION.

It is up to each downstream node to detect when any one of its upstream neighbors goes into the STABILIZATION state and stops producing recent tuples in order to produce corrections. Downstream nodes detect this condition through the explicit heartbeat requests and also because they start receiving corrections. In the replica switching algorithm of Table 4.2, if one of its upstream neighbors goes into the STABILIZATION state, a node switches to a replica that is still producing tentative tuples until a replica finally enters the STABLE state. At that time, the node switches back to the STABLE replica.

4.7.4 Background Corrections of Input Tuples

After a failure heals, when a node finally switches over to a STABLE replica of an upstream neighbor, it indicates the identifier of the last stable tuple that it received and a flag indicating whether it has processed tentative tuples after that last stable one. This information enables the new STABLE upstream neighbor to send corrections to previously tentative tuples before sending the most recent stable tuples. If the failure lasted a long time, simply sending corrections can take a significant amount of time and may cause the downstream node to break its availability requirement.

One solution to this problem would be for a node to enter the STABILIZATION state before switching over to a STABLE replica and requesting corrections to previously tentative inputs. The problem with this approach is that it would require the node to be able to communicate simultaneously with STABLE replicas of *all* its upstream neighbors in order to enter the STABILIZATION state. To avoid this requirement and also to speed-up reconciliation through a chain of nodes, we choose to enable downstream nodes to *correct their inputs in the background*, while they continue processing the most recent tentative tuples.

A node performs background corrections on an input stream by connecting to a *second replica* in the STABILIZATION state, while remaining connected to a replica in the UP_FAILURE state. With these two input streams, SUnions receive tentative and stable tuples in parallel. SUnions support background corrections by considering that the tentative tuples they receive between an UNDO and a REC_DONE are part of the ongoing failure and stable tuples are corrections to earlier tentative ones. Tentative tuples that appear after a REC_DONE belong to a new failure. The Data Path ensures these semantics by monitoring tuples that are entering the SPE and disconnecting the tentative input stream as soon as a REC_DONE tuple appears on the stream with the background corrections.

Table 4.3 shows the extended algorithm for switching between upstream neighbors using both a Curr(s) neighbor to get the most recent inputs and a Bck(s) neighbor to get background corrections. The algorithm is actually simpler than it appears. The basic idea is for a node to first select the Curr(s) neighbor as before using the algorithm from Table 4.2. The selected replica is the best possible replica from the point of view of availability. Given the choice for Curr(s), the node selects a replica to serve as Bck(s), if possible and useful. If Curr(s) is in UP_FAILURE, the node connects to a Bck(s) neighbor in the STABILIZATION state, if such a replica exists. If choosing the best possible replica for Curr(s) yields a replica in STABLE, STABILIZATION, or FAILURE state, a background corrections streams either does not exist or would not be helpful, and the node disconnects from any Bck(s) neighbor.

Supporting background corrections also requires a change in the upstream neighbor monitoring algorithm (Figure 4-10). If a node remains longer than a given time P_3 in the UP_FAILURE state, the Consistency Manager should consider all STABLE upstream neighbors as being in the STABILIZATION state instead. Indeed, if the node connects to one of these STABLE replicas, it will take time to receive and process the replay or corrections. As we discuss above, once an upstream node finishes sending all corrections and catches up with current execution, it sends a REC_DONE tuple. As soon as the Data Path at the downstream node receives such a tuple, it disconnects the background stream ensuring that no tentative tuples follow the REC_DONE. The Consistency Manager then marks the corresponding upstream neighbor as being in the STABLE state again.

We further discuss switching between upstream neighbors in various consistency states when we discuss the properties of DPC in Section 4.11.

In summary, once failures heal, to ensure eventual consistency, a node transitions into the STABILIZATION state when it reconciles its state and corrects the output it produced during the failure. To maintain availability while performing these corrections, the node engages in a simple inter-replica communication protocol to stagger its reconciliation with respect to that of other replicas. Because the number of corrected tuples on a stream may be large, a node corrects its inputs in the background, while continuing to process the most

State	Condition			Action
	`InState(Curr(s), s)`	`InState(Bck(s), s)`	R = `Replicas(s)` − `Curr(s)` − `Bck(s)`	
1	`STABLE`	—	—	Unsubscribe from `Bck`(s) `Bck`(s) ← `NIL` Remain in state 1
2	! `STABLE`	`NIL` or ! `STABLE`	$\exists r \in R$, `InState`(r, s) = `STABLE`	`Curr(s)` ← r Go to state 1
3	`UP_FAILURE`	`STABILIZATION`	$\forall r \in R$, `InState`$(r, s) \neq$ `STABLE`	Remain in state 3
4	`UP_FAILURE`	`NIL` or `UP_FAILURE` or `FAILURE`	$\forall r \in R$, `InState`$(r, s) \neq$ `STABLE` and $\forall r' \in R$, `InState`$(r', s) \neq$ `STABILIZATION`	Unsubscribe from `Bck`(s) `Bck(s)` ← `NIL` Remain in state 4
5	`UP_FAILURE`	`NIL` or `UP_FAILURE` or `FAILURE`	$\forall r \in R$, `InState`$(r, s) \neq$ `STABLE` and $\exists r' \in R$, `InState`(r', s) = `STABILIZATION`	`Bck(s)` ← r' Subscribe to `Bck`(s) Go to state 3
6	`STABILIZATION`	`NIL` or `FAILURE`	$\forall r \in R$, `InState`$(r, s) \neq$ `STABLE` and $\forall r \in R$, `InState`$(r, s) \neq$ `UP_FAILURE`	Unsubscribe from `Bck(s)` `Bck(s)` ← `NIL` Remain in state 6
7	`STABILIZATION`	`NIL` or `FAILURE`	$\forall r \in R$, `InState`$(r, s) \neq$ `STABLE` and $\exists r' \in R$, `InState`(r', s) = `UP_FAILURE`	`Bck(s)` ← r' Subscribe to `Bck(s)` `Bck(s)` ↔ `Curr(s)` Go to state 3
8	`STABILIZATION`	`STABILIZATION`	—	Most advanced of the two should become `Curr(s)` Unsubscribe from `Bck(s)` `Bck(s)` ← `NIL` Go to state 6 or 7
9	`FAILURE`	`NIL` or `FAILURE`	$\forall r \in R$, `InState`$(r, s) \neq$ `STABLE` and $\exists r' \in R$, `InState`(r', s) = `UP_FAILURE`	`Curr(s)` ← r' Subscribe to `Curr(s)` Go to state 4, or 5
10	`FAILURE`	`NIL` or `FAILURE`	$\forall r \in R$, `InState`$(r, s) \neq$ `STABLE` and $\exists r' \in R$, `InState`(r', s) = `STABILIZATION`	`Curr(s)` ← r' Subscribe to `Curr(s)` Go to state 6, or 7
11	`FAILURE`	`NIL` or `FAILURE`	$\forall r \in R$, `InState`(r, s) = `FAILURE`	Unsubscribe from `Bck(s)` `Bck(s)` ← `NIL` Remain in state 11

Table 4.3: Algorithm for switching between replicas of an upstream neighbor in order to maintain availability, while enabling background corrections of input streams. Replicas(s) is the set of replicas producing stream s. Curr(s) and Bck(s) are the upstream neighbor replicas used to get the most recent tentative input data for s or the background corrections, respectively. The state of Curr(s), Bck(s), and the states of nodes in Replicas(s) is given by InState and is thus updated periodically. These states define the conditions that can trigger switches between upstream neighbors. These switches in turn cause the states of Curr(s) and Bck(s) to change. For the following sequence of possible states: STABLE, UP_FAILURE, STABILIZATION, FAILURE, we only examine states where State(Curr(s),s) is on the left of State(Bck(s),s). We assume the two are switched automatically otherwise.

recent tentative data. After correcting its state and catching up with current execution, if no new failures occurred during STABILIZATION, the node transition into the STABLE state. Otherwise, it transition back into the UP_FAILURE state.

4.8 Failed Node Recovery

In the above sections, we described failure and recovery when a node temporarily loses one or more of its input streams. Independent of these failures, a node can also crash. A crashed node restarts from an empty state and must re-build a consistent state before it can consider itself in the STABLE state again. When a node recovers, it must not reply to any requests (including heartbeat requests) until it reaches the STABLE state. Rebuilding the state of a crashed node has been investigated elsewhere [83, 146] and we only summarize how it can be achieved.

Rebuilding the state can occur in one of two ways, depending on the types of operators in the query diagram. The state of convergent-capable operators depends only on a finite window of input tuples and is updated in a manner that always converges back to a consistent state. If a query diagram consists only of these types of operators, the state of the node will always converge to a consistent state, after the node processes sufficiently many input tuples. To rebuild a consistent state, a node simply needs to process tuples and monitor their progress through the operators' states [83]. Once the first set of processed tuples no longer affects the state of any operator, the state has converged. To speed-up recovery, rather than processing only new input tuples, a node can also reprocess some of the tuples already buffered at upstream nodes [83].

If operators are deterministic but not convergent-capable their states may depend on arbitrarily old input tuples. To rebuild a consistent state a node must either reprocess all input tuples ever processed or it must acquire a copy of the current consistent state. Because we assume that at least one replica of each processing node holds the current consistent state at any time (*i.e.*, at most $R-1$ replicas out of R ever *crash* at the same time, although all of them can be in the UP_FAILURE state at the same time), when a node recovers, it can always request a copy of the current state from one of the other replicas. This specific recovery technique is similar to that investigated by Shah *et al.* [146], and we do not investigate it further.

4.9 Buffer Management

In the previous sections, we presented the details of the DPC fault-tolerance protocol. For DPC to work, Data Paths must buffer output tuples and SUnions must buffer input tuples. In Section 4.7.1, we discussed how SUnions can truncate their buffers after each checkpoint. Until now, we assumed, however, that output buffers could grow without bounds. Even if old tuples are written to disk, it is undesirable to let buffers grow without bounds.

We now present algorithms for managing these input and output buffers. We first review which tuples must be buffered and where they need to be buffered in order for DPC to work. We then show that correcting all tentative tuples after failures heal (even those that did not change) significantly reduces buffering requirements. We present a simple algorithm for periodically truncating output buffers in the absence of failures. Because input or output buffers may not always be truncated during failures, we also show how the system can handle long-duration failures with only bounded buffer space: for convergent-capable query

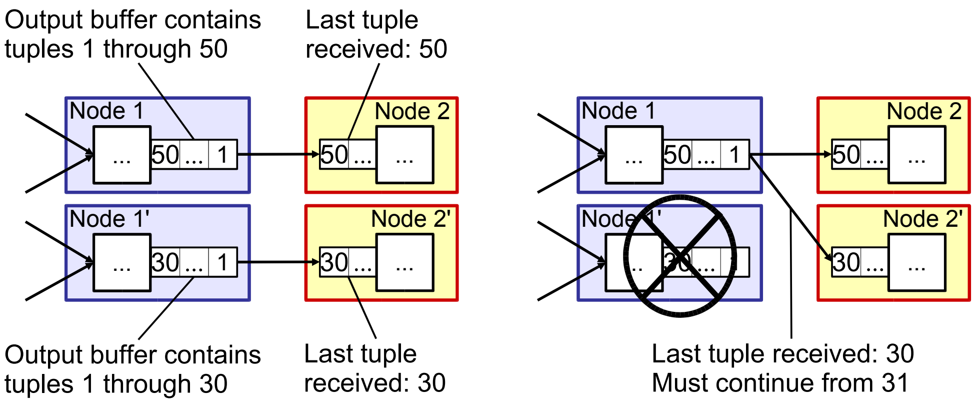

Output buffer contains tuples 1 through 50
Last tuple received: 50
Node 1
Node 2
Node 1'
Node 2'
50 ... 1
30 ... 1
Output buffer contains tuples 1 through 30
Last tuple received: 30
Last tuple received: 30 Must continue from 31

Figure 4-13: Example use of output buffers.

diagrams, the system can maintain availability but may be able to correct only the most recent tentative tuples. For other deterministic query diagrams, the system may have to block when failures last long enough to fill up buffers.

4.9.1 Buffering Requirements

To support DPC, the Data Path must buffer output tuples and SUnions must buffer input tuples. We now review when each type of buffer is necessary.

Output Buffers: A node must buffer the output tuples it produces until all replicas of all downstream neighbors receive these tuples. Indeed, at any point in time, any replica of a downstream neighbor may connect to any replica of an upstream neighbor and request all input tuples it has not yet received. Output buffers are necessary during failures. Upstream nodes must buffer output tuples as long as downstream nodes are unable to receive them. Output buffers are also needed in the absence of failures, because nodes do not process and transmit data at the same time. We assume that nodes have spare processing and bandwidth capacity (see Section 4.1.2). Therefore, nodes can keep up with input rates without falling behind. Nevertheless, some nodes may run somewhat ahead of others nodes. The nodes that are ahead must buffer their output tuples until the other nodes produce the same output tuples and forward them downstream. Figure 4-13 illustrates the need for buffering output tuples. In the example, Node 1 is ahead of its replica Node $1'$. Node 1 already produced tuple with tuple_id= 50. Node $1'$ only produced tuple 30. If Node $1'$ fails, Node $2'$ will request from Node 1 all tuples since tuple 30. These tuples must still be buffered at Node 1.

Input Buffers: When failures occur, nodes need to buffer the input tuples they receive in order to reprocess them later during STABILIZATION. Exact buffering needs depend on the technique used for state reconciliation. As illustrated in Figure 4-14, with checkpoint/redo, SUnions placed on input streams need to buffer tuples they receive between checkpoint, because reconciliation involves restarting from the last checkpoint and reprocessing all input tuples received since then. We discuss the buffering requirements of undo/redo in Appendix B.

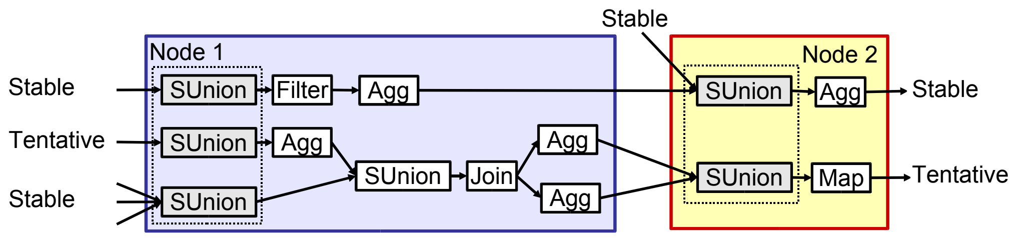


Figure 4-14: Locations where tuples are buffered with checkpoint/redo. SUnions on input streams buffer all tuples received between checkpoints. Operators that buffer tuples are outlined.

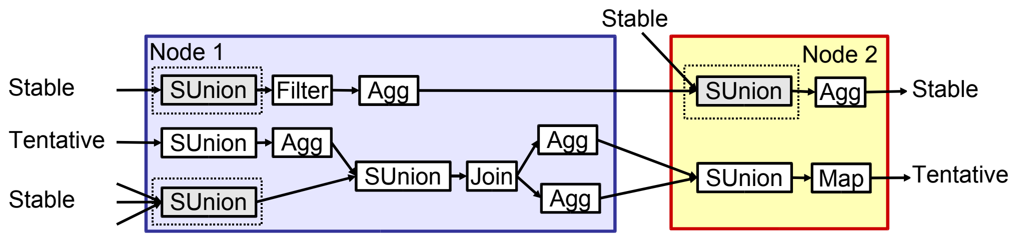


Figure 4-15: Locations where tuples are buffered with checkpoint/redo when all tentative tuples are always corrected. SUnions on input streams buffer only *stable* tuples received between checkpoints. Operators that buffer tuples are outlined.

4.9.2 Avoiding Buffering Tentative Tuples

For long failures, nodes can produce significant amounts of tentative tuples that are later corrected. The first step in buffer management is to decide whether nodes need to buffer these tentative tuples or not.

After a failure heals, nodes stabilize their outputs. Frequently, however, the oldest tentative tuples produced at the beginning of a failure are replaced with identical stable tuples. This may occur, for example, when the oldest tentative buckets on input streams already received all their data and were only missing boundary tuples. DPC's data model and protocols enable nodes to correct only the suffix of tentative tuples that actually changed. Such an optimization, however, requires that tentative tuples be buffered both on output streams and at downstream nodes because any subset of them may never be corrected.

In contrast, if we force nodes to correct *all* tentative tuples, tentative tuples need never be buffered, which results in significant savings in buffer space:

1. The Data Path must still buffer output tuples until all replicas of all downstream neighbors receive these tuples. However, it only needs to buffer *stable* tuples. Tentative tuples are discarded after being sent.
2. For checkpoint/redo, SUnions placed on input streams buffer only *stable* tuples received since the last checkpoint before the failure (Figure 4-15).

4.9.3 Basic Buffer Management Algorithm

Buffers inside SUnions grow only during failures. Data Path output buffers, however, must explicitly be truncated. A possible technique to truncate output buffers is for downstream nodes to send acknowledgments to all replicas of their upstream neighbors after they receive input tuples and for upstream nodes to delete tuples from output buffers once they are acknowledged by all replicas of all downstream neighbors [83, 146].

4.9.4 Handling Long-Duration Failures

With the above buffer truncation algorithm, tuples are deleted from buffers periodically as long as no failures occur. If downstream nodes crash or become disconnected they may no longer send acknowledgments, forcing *buffers to grow with the duration of the failure*. Tuples in buffers can be written to disk so even without additional mechanisms, DPC can tolerate relatively long failures. We want, however, to bound the size of all buffers. We now discuss how the system can cope with long-duration failures given bounded buffer sizes.

When nodes buffer and reprocess all tuples since the beginning of a failure, they ensure that clients eventually receive corrections to all previously tentative results, but even more importantly, they are able to reconcile their states. Limiting buffer sizes may prevent nodes from reconciling their states after a sufficiently long failure. The best approach to handling long-duration failures depends on the type of operators in the query diagram.

Deterministic Operators: The state of a deterministic operator can, in the worst case, depend on all tuples that the operator ever processed. With such operators, any tuple loss during a failure may prevent nodes from becoming consistent again. Such a situation is called system delusion [71]: replicas are inconsistent and there is no obvious way to repair the system. To avoid system delusion, when operators are not convergent-capable, we propose to maintain availability only as long as there remains space in buffers. Once a node's buffers fill up, the node blocks. Blocking creates back pressure all the way up to the data sources, which start dropping tuples without pushing them into the system. This technique maintains availability only as long as buffer sizes permit but it ensures eventual consistency. It avoids system delusion.

Convergent-Capable Operators: Convergent-capable operators have the nice property that any input tuple affects their state only for a finite amount of time. When a query diagram consists only of these types of operators, we can compute, for any location in the query diagram, a maximum buffer size, S, that guarantees enough tuples are being buffered to *rebuild the latest consistent state and correct the most recent tentative tuples*. With this approach, the system can support failures of arbitrarily long duration with bounded-size buffers.

Convergent-capable operators perform computations over windows of data that slide as new tuples arrive. To obtain S at any point in the query diagram, in the worst case, we need to sum up the window sizes of all downstream operators. If we also want to correct a window W of latest tentative tuple, we can add this value to the sum of all the windows. When the same attribute is used for both window specifications and tuple_stime values (Section 4.5), S translates into a maximum range of tuple_stime values. Given S and a boundary tuple with value B, the Data Path or SUnions need only keep tuples that satisfy:

$$\texttt{tuple\_stime} > B - S. \qquad (4.2)$$

Since only stable tuples are buffered, buffer truncation is possible even during failures.

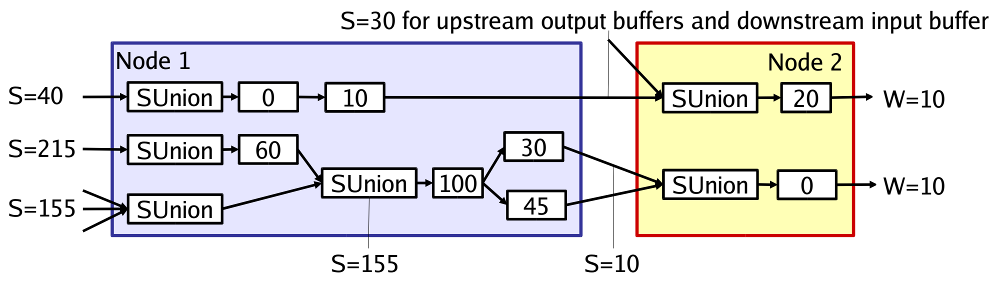


Figure 4-16: Example buffer size allocation with convergent-capable operators. Buffer sizes are indicated with S. The query diagram comes from Figure 4-9 but operator labels have been replaced with example window sizes. The user wants to receive stable versions of the most recent $W = 10$ tentative tuples.

Figure 4-16 shows an example of buffer size allocations.

Hence, with bounded buffers and convergent-capable operators, DPC maintains availability *for failures of arbitrarily long duration.* DPC no longer ensures that *all* tentative tuples are corrected after a failure heals. Instead, an application specifies a window, W, and only the latest W worth of tentative tuples are corrected. For instance, a network monitoring application may specify that it needs to see at least all intrusions and other anomalies that occurred in the last hour. Convergent-capable query diagrams are thus more suitable for applications that need to maintain availability during failures of arbitrarily long duration, yet also need to reach a consistent state after failure heals. For other deterministic operators, buffer sizes determine the duration of failures that the system can support while maintaining availability.

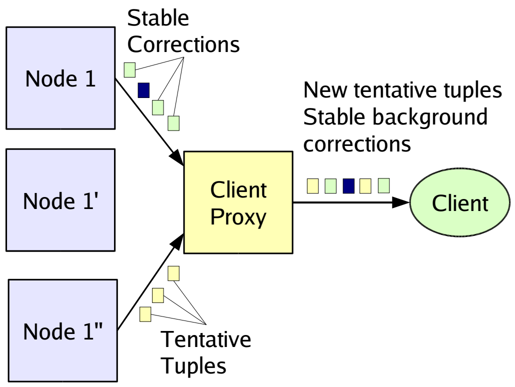
Buffer management is the last component of DPC. In the next section, we briefly discuss the impact of DPC on client applications and data sources.

4.10 Client Applications and Data Sources

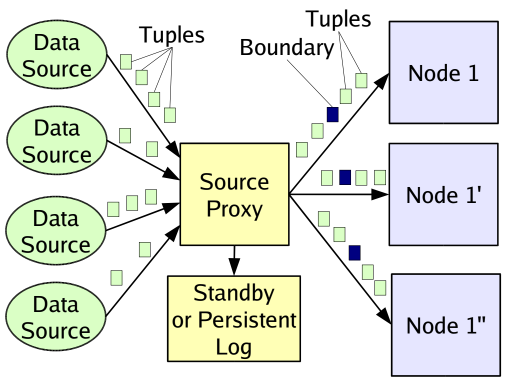
DPC requires client applications and more importantly data sources to participate in the fault-tolerance protocol. This can be achieved by having clients and data sources use a fault-tolerance library or by having them communicate with the system through proxies (or nearby processing nodes) that implement the required functionality. We now describe the latter solution. Figure 4-17 illustrates the use of proxies for data sources and client applications.

4.10.1 Client Proxies

If a client application communicates with the system through a proxy, when the client subscribes to a stream, the proxy receives the request and subscribes itself to the stream instead. Because the proxy runs the fault-tolerance protocol, it monitors all replicas of its upstream neighbors and switches between them in a manner that maintains availability and ensures eventual consistency. Instead of an SUnion, however, the proxy only runs a pass-through filter and sends all output tuples directly to the client application. As illustrated in Figure 4-17(a), the client receives both new tentative tuples and corrections to previous tentative tuples intertwined on the same stream. The proxy ensures that no



(a) Client proxy



(b) Data source proxy

Figure 4-17: Client and data source proxies.

tentative tuples appear after a REC_DONE unless they indicate the beginning of a new failure. Ideally, the proxy should run on the same machine as the client, such that if the proxy becomes disconnected from the rest of the system, the client becomes disconnected as well. A proxy failure is considered equivalent to a client failure.

4.10.2 Data Source Proxies

DPC requires data sources to perform the following functions:

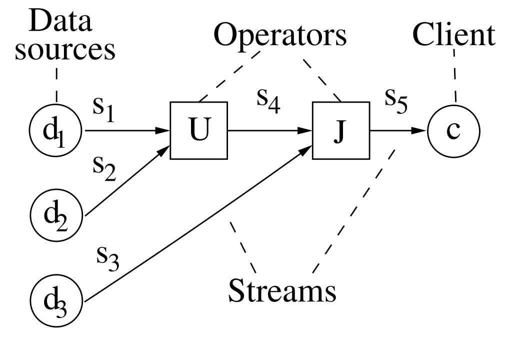
1. Send their streams to *all* replicas of the nodes that process these streams.
2. Buffer output tuples until all replicas of all downstream neighbors acknowledge receiving them.
3. Set tuple tuple_stime values.
4. Insert periodic boundary tuples.

As illustrated in Figure 4-17(b), a proxy can perform the above tasks on behalf of one or more data sources. The proxy should be located near data sources because any network failure between the data source and the proxy is considered to be a data source failure. The failure of the proxy is also considered to be a data source failure. As part of the fault-tolerance protocol, a proxy can set tuple tuple_stime values and can insert periodic boundary tuples on behalf of its data sources, if the latter cannot perform these tasks. Because the proxy is co-located with the data sources, it assumes that if it doesn't hear from these sources, they have no data to send (either because no data is available or because the sources have failed). Because the proxy can fail, it must log tuples persistently before transmitting them to the system, in order to ensures that all replicas eventually see the same input tuples. An alternate technique is for proxy to have a hot-standby replica that takes over when the primary fails.

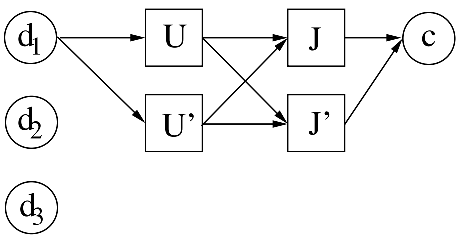
As mentioned in Section 4.1.2, DPC currently supports only transient failures of data sources or their proxies. Permanent failures would be equivalent to a change in the query diagram.

Property #	Brief description
1	Process available inputs in spite of failures.
2	Maintain low-latency processing of available inputs.
3	Choose techniques that reduce inconsistency.
4	Ensure eventual consistency (conditional).
5	Handle multiple simultaneous failures.
6	Handle failures during recovery.
7	Handle frequent failures.

Table 4.4: Summary of properties.



(a) No replication: only one path exists between each source and the client.



(b) With replication: multiple paths exist between each source and the client (only paths from d_1 to c are shown).

Figure 4-18: Paths in a query diagram.

4.11 Properties of DPC

In Section 4.1, we outlined four properties that we wanted DPC to meet (Properties 1 through 4). We now revisit these properties, refine them, and show how DPC meets them. Table 4.4 summarizes the revised properties.

To help us state these properties, we start with a few definitions. We say that a data source *contributes to a stream*, s, if it produces a stream that becomes s after traversing some sequence of operators, called a *path*. Figure 4-18 shows examples of paths in a query diagram with and without replication. In the example, source d_1 contributes to streams $\{s_1, s_4, s_5\}$. Source d_2 contributes to $\{s_2, s_4, s_5\}$. Source d_3 contributes only to $\{s_3, s_5\}$. Without replication (Figure 4-18(a)), there is one path between each source and the client c. By replicating the Union and the Join (Figure 4-18 (b)), there are now four paths from d_1 to c. Other paths are not represented but there are also four paths from d_2 to c and two paths from d_3 to c.

The union of paths that connect a set of sources to a destination (a client or an operator), forms a *tree*. A tree is valid if paths that traverse the same operator also traverse the same replica of that operator. Otherwise the tree is *not valid*. A valid tree is *stable* if the set of data sources in the tree includes *all* sources that contribute to the stream received by the destination. A stable tree produces stable tuples during execution. If the set of sources does not include all those that contribute to the stream, and any of the missing sources

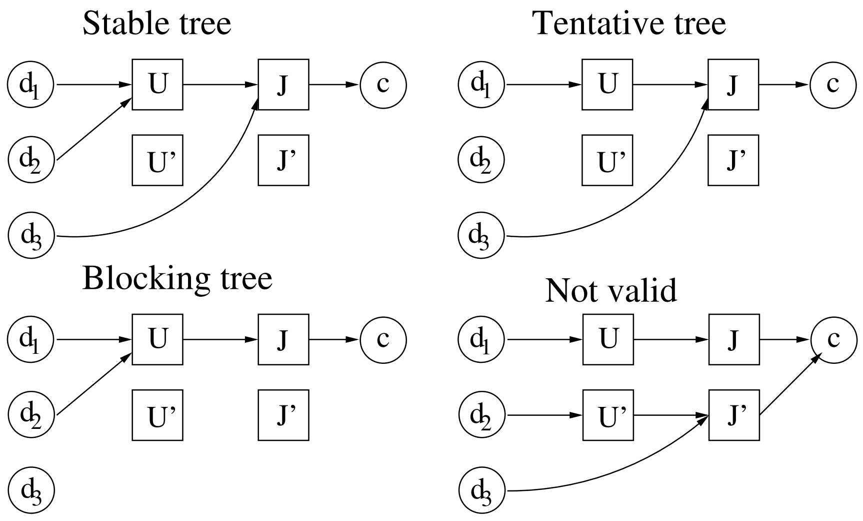


Figure 4-19: Possible types of trees in a distributed and replicated query-diagram.

would connect to the tree through non-blocking operators, the tree is *tentative*. Otherwise, the tree is *blocking*. Figure 4-19 shows an example of each type of tree using the replicated query diagram from Figure 4-18.

4.11.1 Availability and Consistency Properties

The main goal of DPC is to maintain availability, while ensuring eventual consistency and minimizing inconsistency at any time. In this section, we show that DPC meets these goals.

We first focus on availability. In Section 4.1, Property 2 stated that if a path of non-blocking operators exists between a data source and a client application, the client receives results within the desired time-bound. Property 3 stated that DPC favors stable results over tentative one. With the more precise notion of trees, we now re-state these properties more precisely and give either a proof or a qualitative argument for each one of them.

Property 1 Process available inputs in spite of failures*: In a static failure state, if there exists a stable tree, a destination receives stable tuples. If only tentative trees exist, the destination receives tentative tuples from one of the tentative trees. In other cases, the destination may block.*

Precondition*: We assume that all nodes start in the* STABLE *state and are able to communicate with each other when a set of failures occur. Afterward, no other failure occurs and none of the failures heal.*

Proof. This property involves three algorithms: the algorithm for determining the state of output streams from the state of input streams (we assume nodes use the more detailed algorithm from Figure A-1), the algorithm for monitoring input streams (Figure 4-10), and our upstream neighbor switching algorithm from Table 4.3. We prove this property by induction. The induction is on the depth of the tree.

1. Base case: Nodes that communicate directly with a data source receive stable tuples if and only if the source is available (and thus a stable tree exists). If the data source

fails or gets disconnected, neither a stable nor a tentative tree exists, and the nodes do not receive any tuples. Hence the property holds for the base case.

2. Induction hypothesis (part 1): If a node is downstream from a stable tree, it uses the stable stream as input. If a node is downstream from a tentative tree and no stable trees exist, the node uses the tentative inputs. Otherwise, the node may block. We assume, for the moment (and argue in part 2), that nodes label their outputs properly to reflect the type of tree to which each stream belongs: the output of a stable tree is STABLE, the output of a tentative tree is labeled as UP_FAILURE and the output of a blocking tree is labeled as FAILURE.
3. Argument for the induction step (part 1): Downstream nodes periodically request the state of upstream neighbors' replicas, using the algorithm in Figure 4-10. Using that information, nodes switch between replicas of upstream neighbors following the algorithm in Table 4.3. If there exists a stable replica of an upstream neighbor, a node switches to using that replica (Table 4.3, state 2). If no stable replica exists but a replica in UP_FAILURE exists, node switches to that replica instead (Table 4.3, state 7 and 9). Hence if there exists a stable tree, nodes choose the output of that tree as their input. If no such tree exists but a tentative tree exists, nodes choose the output of the tentative tree as input. Otherwise only blocking trees exist and the node may block.
4. Induction hypothesis (part 2): Nodes label their outputs correctly as per the type of tree to which they belong.
5. Argument for the induction step (part 2). Nodes use algorithm A-1 to label their output streams. If all inputs are stable, all outputs are labeled as also stable (lines 5 and 7). If a node itself is blocked because it is reconciling its state, the affected outputs are blocked (line 3). For the other cases, we must determine if an output stream affected by a failure is the output of a blocking tree or a tentative tree. For a stream to be the output of a blocking tree, it must be either downstream from a blocking operator with at least one blocked input (line 9) or downstream from an operator with all its inputs blocked (line 11). Otherwise, the output is tentative (line 14).

Hence, since the property holds for nodes that receive their inputs directly from data sources, nodes label their outputs in a way that reflects the type of tree to which they belong, and downstream nodes always switch their inputs to the best available tree, the property holds for all nodes. ∎

While a node maintains the above property, it may also correct tentative input tuples using a second input stream that we called Bck(s) but these background corrections do not affect availability.

Our goal is not only to ensure that clients receive the best results possible but also that these results arrive within a bounded processing latency.

Property 2 Maintain low-latency processing of available inputs*: If a stable or tentative tree exists, the destination receives results with a delay that satisfies* $\texttt{Delay}_{\texttt{new}} < kD$*, where* D *is the delay assigned to each SUnion operator and* k *is the number of SUnions on the longest path in the tree.*

Proof. Property 1 states that if a stable or tentative tree exists, a node always chooses the output of that tree over an output of a blocking tree. In DPC, an SUnion never buffers an input tuple longer than its assigned delay D. Since there are at most k SUnions in a path, the added processing delay for a tuple cannot exceed kD. ▮

We further study the delay properties in Chapter 5, where we also discuss how delays are assigned to SUnions.

Another desired property of DPC is to minimize inconsistency. More specifically, we strive to achieve the following property:

Property 3 Choose techniques that reduce inconsistency*: Among possible ways to achieve Properties 1 and 2, we seek methods that produce the fewest tentative tuples:* i.e.*,* $\texttt{N}_{\texttt{tentative}}$ *produced by the chosen method is less than* $\texttt{N}_{\texttt{tentative}}$ *of any other method.*

To achieve this property, we experiment with different variants of DPC, measuring both the $\texttt{Delay}_{\texttt{new}}$ and $\texttt{N}_{\texttt{tentative}}$ of each one. We show that the best approach is adaptive: it handles differently failures of different durations. We defer this discussion to Chapter 5.

The above properties address one of the main goals of DPC, which is to maintain availability in spite of failures and try to minimize inconsistency. The other main goal of DPC is to ensure eventual consistency. We first present the case of a single failure. We discuss simultaneous failures in Properties 5 through 7.

Property 4 Ensure eventual consistency*: Suppose a set of failures causes some nodes to crash and some nodes to lose communication with each other, and no other failure occurs. If at least one replica of each processing node does not crash, when all failures heal, the destination receives the complete stable output stream.*

Precondition*: We assume that all nodes start in the* STABLE *state and are able to communicate with each other when the set of failures occur. Afterward, no other failure occurs. Nodes buffer tuples and remove them from buffers as described in Section 4.9. All buffers are sufficiently large (respectively failures are sufficiently short) to ensure no tuples are dropped due to lack of space.*

Proof. We prove the above property by induction. The induction is on the depth of the tree and uses the properties of the upstream neighbor switching algorithm from Table 4.3.

1. Base case: Because we assume all tuples produced during the failure are buffered, when a failure heals, all nodes that communicate directly with data sources receive a replay of *all* previously missing tuples. At least one of these nodes must have remained up and running during the whole failure. This node goes into STABILIZATION: it reconciles its state and stabilizes its outputs. Because the node re-processes all tuples from the beginning of the failure, it can also correct *all* tentative output tuples it produced during the failure. Hence the downstream neighbors of the node now have access to stable tuples that correct *all* previously tentative tuples.
2. Induction hypothesis: When at least one upstream neighbor corrects all tentative tuples it previously produced, a downstream node that remained up and running during the failure can reconcile its state and stabilize its output, correcting in turn, all tentative tuples that it produced.
3. Argument for induction step: When at least one replica of an upstream neighbor corrects all previously tentative tuples, the replica goes back to the STABLE state. Because all failures healed, downstream nodes detect the transition and switch to using the stable replica as upstream neighbor (algorithm from Table 4.3, state 2). When a

node switches to a stable replica of an upstream neighbor, it sends the identifier of the last stable tuple it received. Because we assume that buffers could hold all tuples produced during the failure, the new upstream neighbor can correct all tentative tuples following the one identified by the downstream node. The downstream node will now have received the complete and correct input streams. (Nodes can also correct their inputs in the background as the upstream node is still in STABILIZATION, shown in Table 4.3, states 3 and 6).

Once a node receives the stable version of all previously tentative or missing inputs, it can go into STABILIZATION. The node can re-process all tuples since the beginning of the failure, reconciling its state and correcting *all* tentative tuples it produced.

Hence, after a failure heals, nodes that communicate with data sources receive a replay of all previously missing tuples. Each node reconciles its state and stabilizes its output, correcting *all* tentative tuples it produced during the failure. Every time a node stabilizes its output, its downstream neighbors can correct their inputs, reconcile their states, and stabilize their outputs in turn. Since we assume that at least one replica of each processing node remained up and running during the failure, this process propagates all the way to the client applications. ▮

As discussed in Section 4.9, when failures exceed buffer capacity, for convergent-capable query diagrams, nodes drop old tuples from buffers and no longer correct *all* tentative tuples, but only the most recent ones. For other query diagrams, once buffers fill up DPC blocks and prevents new inputs from entering the system. The system no longer maintains availability in order to ensure eventual consistency and avoid system delusion.

4.11.2 Multiple Failure Properties

In addition to the basic properties above, we show that DPC handles simultaneous failures as well as failures during recovery. In Property 5, we show that as failures occur and a node switches multiple times between replicas of an upstream neighbor, stable tuples on its input streams are never undone, dropped, nor duplicated. In Property 6, we show that when a node reconciles its state, it never drops, duplicates, nor undoes stable output tuples in spite of simultaneous failures and failures during recovery. Finally, in Property 7, we show that a node periodically produces stable output tuples, even when its input streams frequently fail and recover.

Property 5 Handle multiple simultaneous failures*: Switching between trees never causes stable input tuples to be dropped, duplicated, or undone.*

Precondition*: Nodes buffer tuples and remove them from buffers as described in Section 4.9. All buffers are sufficiently large (respectively failures are sufficiently short) to ensure no tuples are dropped due to lack of space.*

We argue this property by studying each possible neighbor-switching scenario:

1. *Switching from an upstream neighbor that was in* STABLE *state before the failure occurred to an upstream neighbor still in* STABLE *state*: Because the downstream node indicates the identifier of the last stable tuple it received, a new STABLE replica can continue from the correct point in the stream either by waiting to produce the identified tuple or replaying its output buffer.

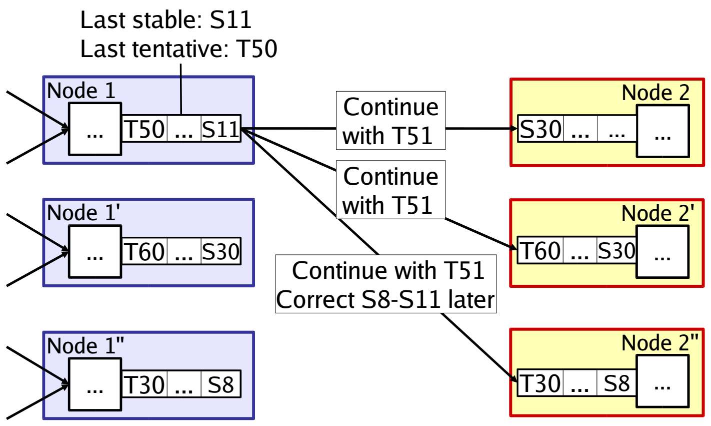


Figure 4-20: Switching between replicas of upstream neighbors in different consistency states. Node 2 and all its replicas are switching to using Node 1. Node 1 always continues with the most recent tentative tuples because it is in the UP_FAILURE state.

2. *Switching from a neighbor in the* UP_FAILURE *state to one in the* STABLE *state*: In this situation, the downstream node indicates the identifiers of the last stable tuple it received and a flag indicating that it processed tentative tuples since that last stable one. The new upstream neighbor thus knows that it needs to stabilize the output and the exact point in the stream where to start the corrections.
3. *Switching to an upstream neighbor in the* UP_FAILURE *state.* In this situation the downstream node and its new upstream neighbor are most likely to have to continue in mutually inconsistent states. Indeed, the last stable tuple produced by the new neighbor (before it went into the UP_FAILURE state) occurred on the stream either before or after the last stable tuple received by the downstream node, as illustrated in Figure 4-20.
   If the last stable tuple produced by the new neighbor appears earlier on the stream (Figure 4-20 Node 2 and Node 2'), the new upstream neighbor has been in UP_FAILURE state longer than the previous upstream neighbor. Because nodes cannot undo stable tuples, the new upstream and downstream pair must continue processing tuples while in mutually inconsistent states.
   If the last stable tuple produced by the new neighbor appears later on the stream (as in the case of Node 2" in Figure 4-20), the new upstream neighbor could stabilize the oldest tentative tuples before continuing with the most recent tentative ones. The downstream node, however, is connecting to the replica in UP_FAILURE state in order to get the most recent input data rather than corrections. Therefore, the new replica continues directly with the most recent tentative tuples.
   In both cases, the upstream neighbor remembers the last stable tuples received by each new downstream neighbor but sends them directly the most recent tentative data. After failures heal and the upstream node stabilizes its output, it looks up the last stable tuple received by each downstream client and produces the appropriate undo and correction for each one of them.
4. Finally, a node may connect to a replica in the STABILIZATION state to correct input streams in the background. Because the node indicates the last stable tuple it received, background corrections always start at the appropriate point in the stream.

Additionally, when a node receives both new tentative input tuples and corrections, DPC requires that the Data Path monitors these incoming streams ensuring that only the background stream carries stable data.

We have shown that when a node switches between replicas of upstream neighbors, for all consistency combinations of these neighbors, the downstream node receives the correct input tuples: stable tuples are never undone, dropped, or duplicated. Tentative tuples, however, can be duplicated and more likely dropped while a node switches between upstream neighbors but these duplications or losses affect neither availability nor eventual consistency.

The second problem with multiple failures is that input streams become tentative at different points in time and may not all get corrected at the same time. In the extreme case, there can always be at least one failed input. Additionally, new failures can occur at any time, which includes the time when a node is in the process of recovering from earlier failures. To maintain consistency, we must ensure that stabilization never causes output tuples to be duplicated, dropped, or undone.

Property 6 Handle failures during failures and recovery*: Stabilization never causes stable output tuples to be dropped, duplicated, or undone.*

Precondition*: Nodes buffer tuples and remove them from buffers as described in Section 4.9. All buffers are sufficiently large (respectively failures are sufficiently short) to ensure no tuples are dropped due to lack of space.*

We argue the above property by examining each possible failure scenario for each state reconciliation technique. We present checkpoint/redo here. We discuss undo/redo in Appendix B. We show that DPC handles failures during failures and recovery without the risk of dropping, duplicating, or undoing stable tuples.

Because our goal is to produce few tentative tuples, for a node to enter the STABILIZATION state, it must have received corrections to previously tentative tuples on *all* its input streams. Indeed, if a node launched reconciliation when only one stream had been corrected, if the stream was unioned or joined with another still tentative stream, it is likely that only new tentative tuples would have resulted from the reconciliation. Hence, when a node starts to reconcile its state it has old stable tuples on all its input streams. If a failure occurs during stabilization, the new tentative tuples appear *after* these stable tuples on the input streams.

Because SUnions are placed on all input streams and they buffer all stable tuples that arrive between checkpoints, restarting from a checkpoint cannot cause any tuple losses.

SOutput operators guarantee that stable tuples are neither undone nor duplicated. When restarting from a checkpoint, SOutput enters a “duplicate elimination” mode. It remains in that state and continues waiting for the same last duplicate tuple until it produces the UNDO tuple, even if another checkpoint or recovery occur. After producing the UNDO, SOutput goes back to its normal state, where it remembers the last stable tuple that it sees and saves the identifier of that tuple during checkpoints. If a new failure occurs before the node had time to catch up and produce a REC_DONE tuple, SOutput forces a REC_DONE tuple between the last stable and first tentative tuples that it sees.

Hence, with checkpoint/redo recovery, failures can occur during failures or reconciliation and the system still guarantees that stable tuples are not undone, dropped, nor duplicated.

Finally, we address the problem of frequent failures, and show that even if there is no time when all components and communication in a system are fully functional, the system can make forward progress. We define forward progress as producing new stable tuples. To enable forward progress on output streams, each stream in the query diagram must periodically make forward progress.

Property 7 Handle frequent failures*: Even if inputs to a processing node fail so frequently that at least one is down at any time, as long as each input stream recovers between failures and makes forward progress, the node itself makes forward progress.*

Precondition*: Nodes buffer tuples and remove them from buffers as described in Section 4.9. All buffers are sufficiently large (respectively failures are sufficiently short) to ensure no tuples are dropped due to lack of space.*

An input stream makes forward progress when a node is able to receive a replay of previously missing inputs or corrections to previously tentative inputs. A node makes forward progress when it reconciles its state and stabilizes its output.

We argue this property through a set of assertions about the processing node with the failing inputs. These assertions lead to the main conclusion. For clarity, we assume that SUnion operators placed on input streams buffer tuples in buckets identified with sequence numbers that increase with time. We also assume that, for all SUnions, buckets with the same sequence number correspond to roughly the same point in time.

1. Assertion 1: For any stream s and bucket i, the node eventually receives stable tuples for that bucket on that stream. Argument: A stream that fails always eventually recovers and makes forward progress before failing again. Each recovery enables the downstream node to correct (possibly in the background) at least a subset of tuples previously missing or tentative on that stream. Hence, a node eventually receives stable input tuples for all buckets on all input streams.
2. Assertion 2: For any i, all SUnion operators eventually have only stable tuples in their bucket i. Argument: Follows directly from Assertion 1.
3. Assertion 3: The node eventually goes into the stabilization state. Argument: Let j be the first bucket processed as tentative by at least one SUnion. By Assertion 2, eventually, all SUnions will have stable tuples in their bucket j. The node will then be able to go into STABILIZATION.
4. Assertion 4: A node makes forward progress processing stable tuples, in spite of failures. Argument: By assertion 3, a node eventually goes into STABILIZATION. Assuming the node uses checkpoint/redo,[7] at the first state reconciliation, the node restarts from a checkpoint taken before processing bucket j. It then processes bucket j and following. Tentative tuples eventually appear in bucket $k > j$. At the next stabilization, even if the node restarts from the same checkpoint, it processes bucket j through k before starting to process new tentative tuples. Once again, the node makes forward progress. Of course, to avoid re-processing the same tuples multiple times and having SOutput filter duplicates, a node should checkpoint its state before starting to process any tuples from bucket k.
5. Conclusion: Because a node can periodically enter the STABILIZATION state and make forward progress processing stable tuples, it periodically produces new stable tuples on its output streams.

[7]The argument is similar for undo/redo reconciliation.

4.12 Summary

In this chapter, we presented DPC, an approach to fault-tolerant distributed stream processing. DPC introduces an enhanced data model, where tuples are explicitly labeled as either tentative or stable. DPC is based on replication: each query diagram fragment is replicated on multiple processing nodes. Replication enables the system to mask many node and network failures, and reduces the probability of total system failure. DPC is based on the principle that each replica should manage its own availability and consistency. With DPC, each node implements a fault-tolerance protocol based on a three-state state machine. In the STABLE state, replicas maintain mutual consistency by using SUnion, a simple data serializing operator, coupled with periodic boundary tuples. Nodes also detect failures on their input streams by using these boundary tuples and an explicit upstream neighbor monitoring mechanism. When failures occur, nodes transition into the UP_FAILURE state, where they maintain the required availability, while trying to minimize inconsistency by switching between replicas of upstream neighbors, and possibly delaying processing new input tuples. Once failures heal, nodes transition into the STABILIZATION state, where they reconcile their states using either undo/redo or checkpoint/redo. To maintain availability during STABILIZATION, nodes run an inter-replica communication protocol to stagger their reconciliation. DPC requires that nodes buffer output tuples and input tuples, and we presented an approach for managing these buffers. To benefit from fault-tolerance without implementing any additional protocols, client applications and data sources can communicate with the system through a nearby processing node that acts as their proxy. We have shown that DPC provides the required availability and consistency and supports both single failures and multiple simultaneous failures. DPC is designed for a low failure frequency, but it also works when failures occur frequently. DPC works particularly well when query diagrams are convergent-capable. In the next chapter, we present the implementation of DPC in Borealis and its evaluation through experiments with our prototype implementation.

Chapter 5

Fault-Tolerance: Evaluation

In this chapter, we present the implementation of DPC in Borealis and evaluate its performance through experiments with our prototype implementation.

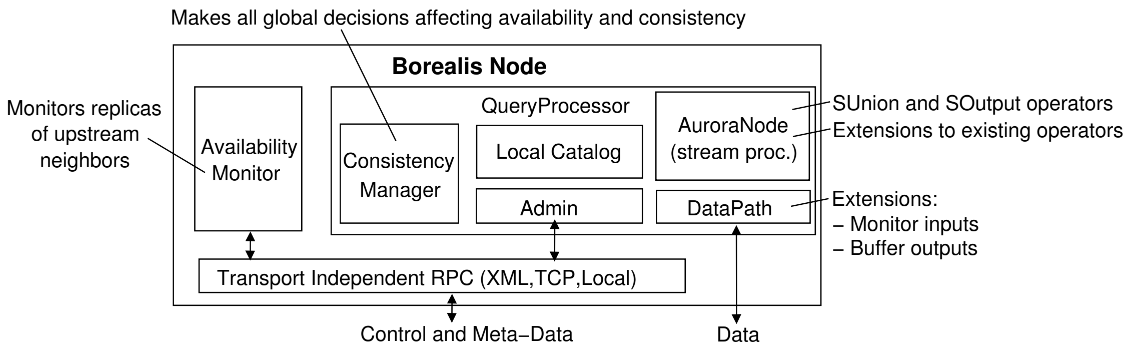
Our evaluation has two main goals. First, we show that DPC can ensure eventual consistency while maintaining, at all times, a required level of availability, both in a single-node and a distributed deployment. Second, we study techniques that enable a SPE to minimize the number of tentative tuples it produces, while meeting specific availability and consistency goals. In this second part, we find that the best strategy is for any SUnion to block for the maximum incremental processing latency when it first detects a failure. If the failure persists, SUnions should process new tuples without delay because later delays are not helpful.

Another important goal of our evaluation is to study and compare techniques to efficiently reconcile the state of an SPE. We find that checkpoint/redo outperforms undo/redo. We show how to enhance the checkpoint-based approach to avoid any overhead in the absence of failures and, when failures occur, limit reconciliation and even overhead to those operators affected by the failures.

The rest of this chapter is organized as follows. In Sections 5.1 and 5.2, we present the details of DPC's implementation and discuss how a user can write a fault-tolerant application, respectively. We also show an illustrative example of the input received by an application when the SPE experiences a temporary failure that it cannot mask. In Section 5.3, we run a few experiments demonstrating that DPC properly handles multiple simultaneous failures including failures during recovery. We show that regardless of the failure scenario, DPC always correctly stabilizes the output stream. In Section 5.4, we turn to the quantitative evaluation of DPC and study the trade-offs between availability and consistency. In Section 5.5, we compare the performance of state reconciliation using either checkpoint/redo or undo/redo. In Section 5.6, we study the overhead of DPC. We summarize the main conclusions from the evaluation in Section 5.7 and discuss the limitations of DPC in Section 5.8.

5.1 Implementation

In this section, we present the implementation of DPC in Borealis. We already presented the overall software architecture in Section 4.3. We now summarize the main functions of each component and add a few implementation details. Figure 5-1 presents the complete extended software architecture.



Makes all global decisions affecting availability and consistency
Borealis Node
Monitors replicas of upstream neighbors
Availability Monitor
QueryProcessor
Consistency Manager
Local Catalog
AuroraNode (stream proc.)
Admin
DataPath
SUnion and SOutput operators
Extensions to existing operators
Extensions:
– Monitor inputs
– Buffer outputs
Transport Independent RPC (XML,TCP,Local)
Control and Meta–Data
Data

Figure 5-1: Extended software node architecture for fault-tolerant stream processing.

5.1.1 Consistency Manager and Availability Monitor

The Consistency Manager makes all global decisions related to failure handling:

1. For each input stream, the Consistency Manager selects, by implementing the algorithm from Table 4.3, the replica that should serve as upstream neighbor and optionally the one to serve as background corrections.
2. The Consistency Manager computes the states of output streams. In the current implementation, the Consistency Manager assigns the state of the node as the advertised state of all output streams, rather than using the more sophisticated algorithm from Figure A-1. To determine the state of the node, the Consistency Manager uses information from SUnion and SOutput operators. These operators send messages to the Consistency Manager in the form of tuples produced on separate *control output streams*. When an SUnion starts processing tentative data, it produces an UP_FAILURE message. As soon as one SUnion goes into UP_FAILURE, the whole node is considered to be in that state. When an SUnion receives sufficiently many corrections to reconcile its state, it produces a REC_REQUEST message. Once all previously failed SUnions on input streams are ready to reconcile, the Consistency Manager triggers state reconciliation (either checkpoint/redo or undo/redo), taking the node into the STABILIZATION state. Once reconciliation finishes or another failure occurs, each SOutput sends a REC_DONE tuple downstream and to the Consistency Manager. After receiving one such tuple from each SOutput, the Consistency Manager takes the node back to the STABLE or the UP_FAILURE state.
3. To ensure that replicas stagger their state reconciliations, the Consistency Manager implements the inter-replica communication protocol from Section 4.7.3.

The Consistency Manager uses a separate component, the Availability Monitor, to implement the upstream neighbor monitoring protocol from Figure 4-10. The Availability Monitor informs the Consistency Manager only when the state of any output stream at any replica changes. The Availability Monitor is a generic monitoring component that could easily be extended to provide monitoring for purposes other than fault-tolerance.

5.1.2 The Data Path

The Data Path handles the data coming into the SPE and the data sent to downstream neighbors. DPC requires a few extensions to this component:

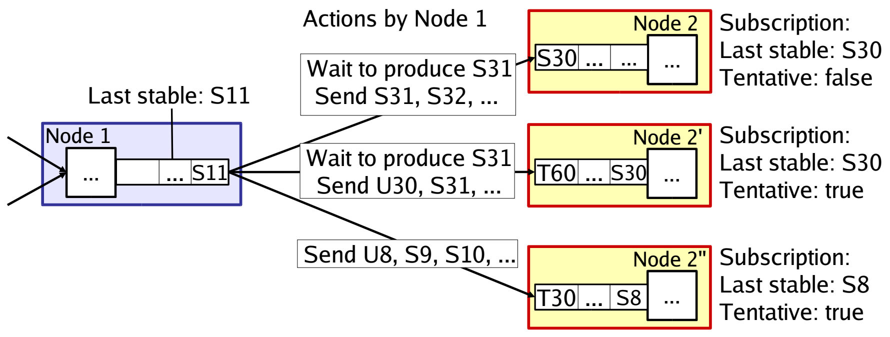
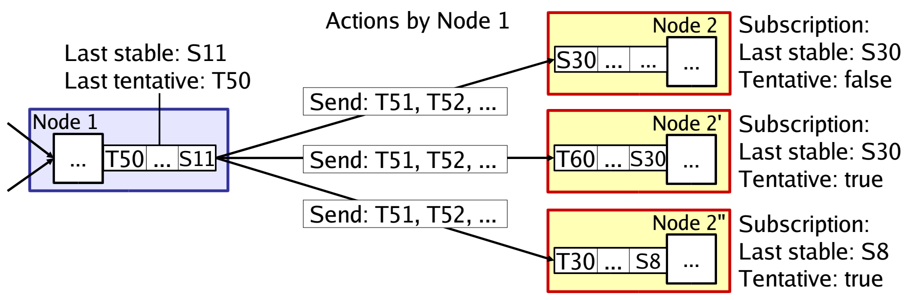


Figure 5-2: A node in STABLE state stabilizes the input streams of new downstream neighbors as per the information they provide in their subscriptions. Subscription information is shown on the right of each downstream node (Node 2, Node 2′, and Node 2″). Actions taken by upstream Node 1 are shown on the arrows.

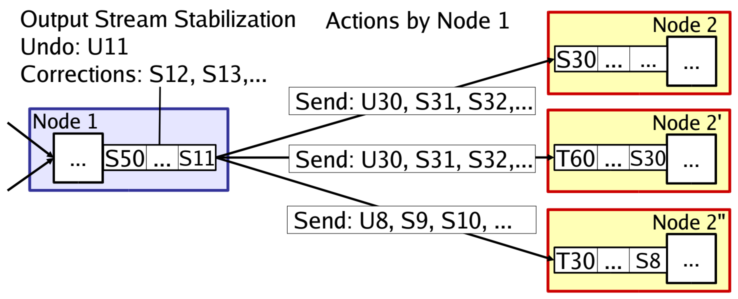
1. The Data Path monitors the input tuples entering the SPE. For each input stream, the Data Path remembers the most recent stable tuple and whether any tentative tuples followed the last stable one. The Data Path provides this information to the Consistency Manager when the latter decides to switch to another replica of an upstream neighbor. The information is sent to the new replica in a susbscribe message.
2. The greater challenge in managing input streams is for the Data Path to handle both a main input stream and a second input stream with background corrections. If a stream of corrections exists, the Data Path must verify that the main input carries only tentative tuples. Other tuples must be dropped. Additionally, the main input stream must be disconnected as soon as a REC_DONE tuple appears on the background corrections stream. The latter then takes the place of the main input stream.
3. The Data Path must also buffer stable output tuples (tentative and UNDO tuples are never buffered), and stabilize, when appropriate, the data sent to downstream neighbors. Figure 5-2 illustrates the tuples sent by a node in STABLE state to new downstream neighbors in different consistency states. Each downstream neighbor provides information about the most recent data it received in the subscribe message. The Data Path uses that information to decides what tuples to send. The upstream node also sends a REC_DONE right after sending the missing stable tuples.
   If the upstream node itself is experiencing a failure, the Data Path simply sends the most recent tentative data to all new downstream neighbors. Once it starts stabilizing its output streams, however, the Data Path uses the information from the subscription messages to produce and send the appropriate UNDO tuple and corrections to each downstream neighbor. Figure 5-3 illustrates the tuples sent to downstream neighbors when they first connect and later when the node goes from UP_FAILURE to STABILIZATION. After stabilizing its state, the Data Path propagates, to all downstream clients, the REC_DONE tuple that appears on its output stream.

5.1.3 Query Diagram Modifications

DPC requires three modifications to the query diagram: the addition of SUnion operators, the addition of SOutput operators, and small changes to all other operators.



(a) When a node in UP_FAILURE acquires new downstream neighbors, it sends them the most recent tentative data.



(b) The node stabilizes the input streams of downstream neighbors only after stabilizing its own output streams.

Figure 5-3: Stream stabilization by a node in the UP_FAILURE state.

SUnion

SUnion operators appear on all input streams of an SPE node but they can also appear in the middle of the local query diagram fragment. SUnions in the middle behave somewhat differently than SUnions on inputs. We now outline these differences.

SUnions on input streams can be seen as performing two roles: the role of an SUnion and the role of an input buffer. This difference is visible during checkpoints when an SUnion in the middle of the diagram behaves like an ordinary operator. It makes a copy of its state or reinitializes its state (*i.e.*, its buckets of tuples) from a checkpoint. An SUnion on an input does not make a copy of its state during checkpoints. Instead, it only remembers the identifier of the bucket it is currently processing. Indeed, an input SUnion does not receive a replay of earlier stable tuples. It is the one to perform the replay.

The location of an SUnion also determines the manner in which it processes and produces control tuples. Since failures can only occur on inputs, only input SUnions need to communicate with the Consistency Manager, although in our current implementation, all SUnions signal failures and readiness to reconcile.

Besides differences between an SUnion placed on an input stream and one in the middle of the diagram, the behavior of an SUnion also depends on the operator that follows it in the query diagram. If an SUnion simply monitors an input stream or replaces a Union operator, it produces all tuples on a single output stream. This is not the case for an

SUnion that precedes a Join operator. In Borealis, a Join operator has two distinct input streams. To enable a Join to process input tuples deterministically, the SUnion placed in front of the Join orders tuples by increasing tuple_stime values but it outputs these tuples on two separate streams that feed the Join operator. We modify the Join to process tuples by increasing tuple_stime values, breaking ties between streams deterministically. We call the modified Join a *Serializing Join* or *SJoin*.

SOutput

SOutput operators monitor output streams. Independently of the sequence of checkpoints and recovery, SOutputs ensure they produce UNDO and REC_DONE tuples when appropriate. When restarting from a checkpoint, the normal case is for SOutput to produce an UNDO tuple, drop duplicate stable tuples until the first new tuple appears, and forward the REC_DONE tuple when it finally makes its way to the output. There are three other interesting cases. First, if SOutput did not see any tentative tuples after a checkpoint, it only drops duplicate stable tuples without producing an UNDO. Second, if another checkpoint and another recovery occur *while* SOutput is dropping duplicates, after the recovery, SOutput goes back to dropping duplicates and expecting the same last stable tuple as before. Finally, if a failure occurs during recovery SOutput forces a REC_DONE tuple before the first new tentative tuple. All SOutputs propagate REC_DONE tuples to the Consistency Manager in addition to sending them downstream.

Operator Modifications

DPC also requires a few changes to stream processing operators. In the current implementation, we have modified Filter, Map, Join, and Aggregate to support DPC, and SUnion replaces the Union operator. We already mentioned the modification of the Join operator to make it process tuples in the deterministic order prepared by a preceding SUnion operator. We now outline the other necessary changes.

For checkpoint/redo, operators need the ability to take snapshots of their state and recover their state from a snapshot. Operators perform these functions by implementing a packState and an unpackState method.

For undo/redo, the changes are significantly more involved. Operators must keep an undo buffer. They must compute stream markers and remember the last tuple they output. They must understand and process UNDO tuples. The undo handling functionality can be implemented with a wrapper, requiring that the operator itself only implements two methods: clear() clears the operator's state and findOldestTuple(int stream_id) returns the oldest tuple from input stream, stream_id, that is currently in the operator's state. In the prototype, we only implemented undo/redo directly inside an experimental Join operator. Because the approach does not perform as well as checkpoint/redo, we did not implement it for other operators.

Operators must also be modified to set the tuple_stime value on output tuples, in addition to other fields in the tuple headers. Different algorithms are possible. Stateless operators can simply copy the value from the input tuples. Aggregate operators can use the most recent tuple_stime value in the window of computation. Join can use the most recent value among those present in the two tuples being joined. The chosen algorithm determines how an operator processes boundary tuples. For this reason, we chose to implement both functionalities directly inside operators. To produce a boundary tuple, an operator must implement the method findOldestTimestamp() that returns the oldest tuple_stime value that the operator *can still produce*. This value depends on the algorithm used to assign tuple_stime values. In the best case, the operator can simply propagate the boundaries it receives. In the worst case, this boundary value is the minimum of all tuple_stime values in the operator's state and the latest boundary tuples received on each input stream.

Finally, upon receiving a tentative tuple, operators must start to label their output tuples as TENTATIVE as well.

5.1.4 Implementation Limitations

The implementation currently lacks the following features:

1. We did not implement tentative boundaries. Currently, the SUnion operator always uses a timeout to decide when to process a tentative bucket. Even when an SUnion should process tuples as soon as they arrive, we set a minimum timeout of 300 ms. SUnion would not need the timeout with tentative boundaries, but the SUnion logic would have to be changed to properly handle failed input streams that do not even carry tentative boundaries.
2. We did not implement data source proxies. The current system does not tolerate data source failures, and assumes data sources produce boundary tuples and set tuple_stime values. Data sources can use an intermediate Borealis node to send data to multiple replicas, replaying it as necessary. The system will not support a failure of the proxy.
3. We did not implement the buffer management algorithm. For each output stream, the Data Path uses a circular buffer of a pre-defined size.
4. As we mentioned above, we have a full implementation of checkpoint and redo. We only implemented undo capabilities inside SUnions and inside one experimental Join operator. Focusing on a single technique, however, makes the implementation of the Consistency Manager cleaner because the logic inside the latter must be a little different for undo-based recovery.
5. The Consistency Manager does not distinguish the exact states of output streams but equates them with the state of the node itself.
6. A failed node can currently re-join the system only during a time-period without failures such that its state has time to converge to a consistent state before any node subscribes as downstream neighbor. Because the node joins with an empty state, the scheme only supports convergent-capable query diagrams at the moment.
7. The Borealis operator scheduler is quite simple. A constant defines how many input tuples an operator reads every time it gets scheduled. All operators use the same constant independent of their selectivities. SUnion operators are sensitive to this constant because they process and potentially produce one bucket of data for every iteration. This sensitivity may manifest itself during state reconciliation. It affects reconciliation times and the way queues build up then clear during recovery.

5.2 Fault-Tolerant Applications

We now discuss how users can write fault-tolerant applications, and show the sample output of an application during a failure that the SPE cannot mask.

To make an application fault-tolerant, it is currently necessary for the application writer to manually include all necessary SUnion and SOutput operators, although it would not be difficult to add these operators automatically. Application writers must also use the modified Join operator called SJoin. For aggregates, to ensure that they are convergent-capable, an option called "independent window alignment" must be set. This option ensures that window boundaries are independent of the exact value inside the first tuple processed by the operator. When deploying a query diagram onto processing nodes, a user can simply label groups of nodes as *replica sets*. Assigning a query diagram fragment to a replica set automatically replicates the fragment. For the client proxy, the deployment should include, as the last hop, a Borealis node running a pass-through filter.

Figure 5-4 shows two screenshots from an application that illustrates the output of a query diagram. Each screen is composed of two panels. The top panel shows the types of tuples that appear on the output stream. Each tuple is displayed as one square. Tuples are displayed in four columns and are drawn from left to right and from top to bottom. Hence the oldest tuple is the square in the top left corner and the most recent tuple appears at the bottom right. Columns shift left by one when all four columns fill up. Stable tuples are shown in green. Tentative tuples are shows in yellow, control tuples (UNDO and REC_DONE) are shown in red, and missing tuples leave gray holes. The bottom panel shows the end-to-end processing latency for tuples. The x-axis shows the time when a tuple was received and the y-axis shows the processing latency for that tuple. In the example, the diagram unions three streams and performs some processing equivalent to a pass-through filter. The maximum incremental processing latency is set to 3 seconds and one of the three input streams becomes temporarily disconnected for 20 seconds.

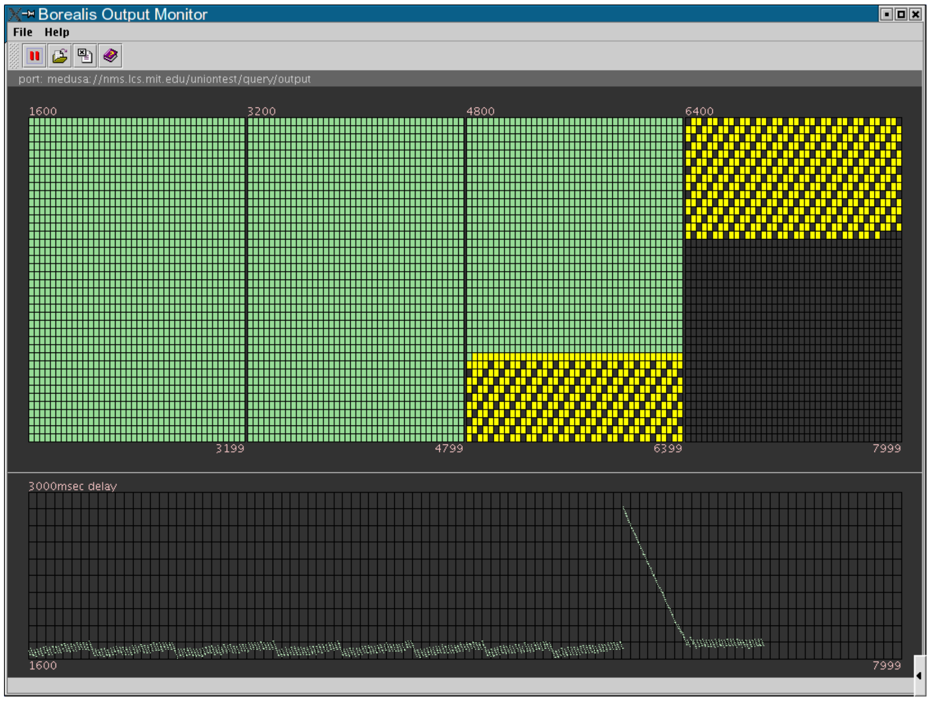
Figure 5-4(a) shows the output during the failure. The figure shows that the output is stable at first. Green squares appear in the first two and a half columns. The processing latency, shown in the bottom window, is also small for these tuples. As the failure occurs, the processing latency increases to almost 3 seconds as the SPE suspends processing. Because the failure persists, however, in order to meet the required 3-second availability, the system starts producing tentative results. These results appear as yellow squares. In this example, each input stream contributes one third of the output tuples, so a third of the output tuples are also missing during the failure.

Figure 5-4(b) shows the output after the failure heals. The previously tentative results are replaced with a red UNDO tuple, a sequence of stable corrections, and a red REC_DONE tuple. Note that the output of Figure 5-4(b) is shifted by two columns compared with that of Figure 5-4(a) because time has passed between the two screenshots. In this example, we used a single SPE node to show how the processing latency increases during state reconciliation. The bottom panel of Figure 5-4(b) shows a second peak in processing latency corresponding to the reconciliation.

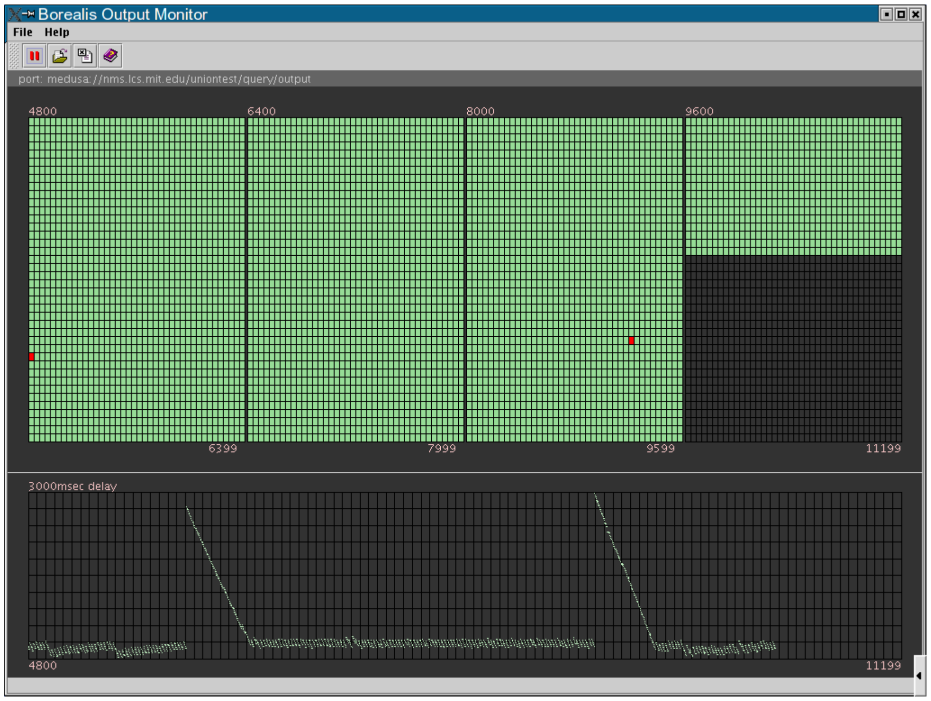
5.3 Multiple Failures

In this section, we show examples of how DPC with checkpoint/redo handles simultaneous failures and failures during recovery. We show that client applications eventually receive the stable version of all result tuples and that no stable tuples are duplicated.

We run a simple query diagram (Figure 5-5) that produces, on the output stream, tuples with sequentially increasing identifiers. We first cause a failure on input stream 1. We label this failure as "Failure 1". We then cause a failure on input stream 3. We label this failure as "Failure 2". We plot the sequence numbers received by the client application over time as the two failures occur and heal. Figure 5-6(a) shows the output when the two failures overlap in time. Figure 5-6(b) shows the output when Failure 2 occurs exactly at the



(a) When the failure occurs, the SPE node first suspends processing tuples then produces tentative results.



(b) When the failure heals, previously tentative data is replaced with the final stable results.

Figure 5-4: Screenshots from a client application during a partial failure of the SPE.

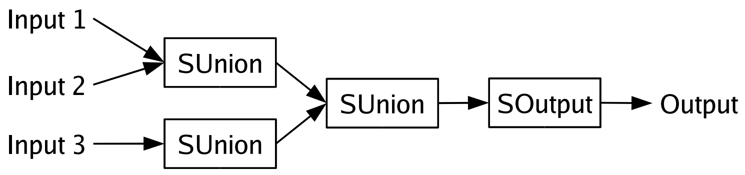
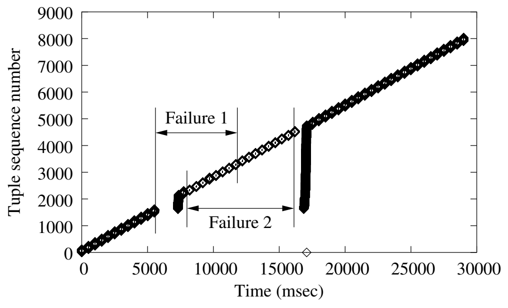
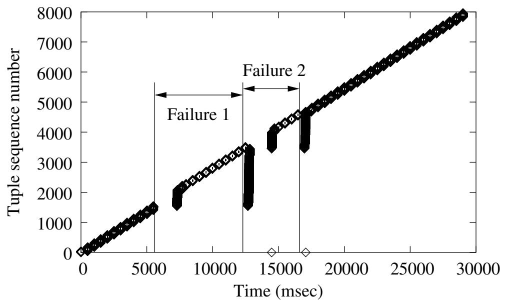


Figure 5-5: Query diagram used in simultaneous failures experiments.



(a) Overlapping failures



(b) Failure during recovery

Figure 5-6: Example outputs with simultaneous failures.

moment when Failure 1 heals and the node starts reconciling its state.

In the case of simultaneous failures (Figure 5-6(a)), as Failure 1 occurs, the output first stops because the node suspends all processing. All tuples following the pause are tentative tuples. Nothing special happens when the second failure occurs because the output is already tentative. Nothing happens either when the first failure heals because the second failure is still occurring. It is only when all failures heal, that the node enters the STABILIZATION state and sends the stable version of previously tentative data. As the node finishes producing corrections and catches up with normal execution, it produces a REC_DONE tuple that we show on the figure as a tuple with identifier zero (tuple that appears on the x-axis). As the experiment shows, in our implementation, we chose the simpler approach of waiting for all failures to heal before reconciling the state. This approach works well when failures are infrequent. In the case of a large number of input streams and frequent failures, we could change the implementation to reconcile the state as soon as the first failure heals and the node can reprocess a few buckets of stable input tuples on all input streams.

In the example with a failure during recovery (Figure 5-6(b)), as the first failure heals, the node enters STABILIZATION and starts producing corrections to previously tentative results. Before the node has time to catch-up with normal execution and produce a REC_DONE the second failure occurs and the node suspends processing once again. The node then produces a REC_DONE to indicate the end of the sequence of corrections before going back to processing tentative tuples once again. After the second failure heals, the node corrects *only those tentative tuples produced during the second failure.* The corrections are followed by a REC_DONE. Hence all tentative tuples get corrected and no stable tuple is duplicated.

5.4 Availability and Consistency Trade-Offs

In the previous sections, we described the implementation of DPC in Borealis and showed sample outputs produced during executions with failures. We also showed examples of how DPC handles simultaneous failures and failures during recovery. In this section, we turn toward a quantitative evaluation of DPC and study trade-offs between availability and consistency.

Our first goal is to show that DPC meets a pre-defined level of availability, measured as a maximum incremental processing latency, while ensuring eventual consistency. For long failures, we show that it is necessary for nodes to process tentative tuples *both* during failures and stabilization in order to meet a given availability goal. However, nodes can sometimes suspend or delay processing new tuples, yet still provide the required availability. Our second goal is to study which such DPC variant produces the fewest tentative tuples.

In Section 5.4.1, we show results for a single-node deployment. Most results are straightforward, showing that suspending processing enables the system to mask short failures without introducing any inconsistency. For long failures, suspending breaks the availability requirement, but continuously delaying new tuples as much as possible produces fewer tentative tuples than processing them without delay. In Section 5.4.2, we compare the same techniques but in a distributed setting and show that, interestingly, delaying new tuples hurts availability often *without improving consistency.*

Finally, because many failures are short, it is always best for a node to suspend processing new tuples for some time-period, D, when a failure first occurs. For a single node and SUnion, D equals the maximum incremental processing latency. For a sequence of SUnions, however, the optimal value of D that each SUnion should use depends on the overall failure handling dynamics. In Section 5.4.3, we study different algorithms for assigning delays to SUnions and show the availability and consistency that result from each assignment. We find that *assigning the maximum incremental delay to each SUnion* yields the best consistency, while still meeting the availability requirement.

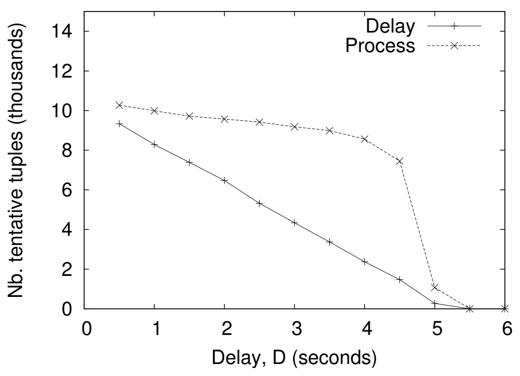
All single-node experiments are performed on a 3 GHz Pentium IV with 2 GB of memory running Linux (Fedora Core 2). Multi-node experiments are performed by running each pair of node replicas on a different machine. All machines have 1.8 GHz Pentium IV processors or faster with at least 1 GB of memory.

The basic experimental setup is the following. We run a query diagram composed of three input streams, an SUnion that merges these streams into one, an SJoin that serves as a generic query diagram with a 100 tuple state size, and an SOutput. The aggregate input rate is 3000 tuples/s. We create a failure by temporarily disconnecting one of the input streams without stopping the data source. After the failure heals, the data source replays all missing tuples while continuing to produce new tuples. Unless stated otherwise, nodes use checkpoint/redo to reconcile their state.

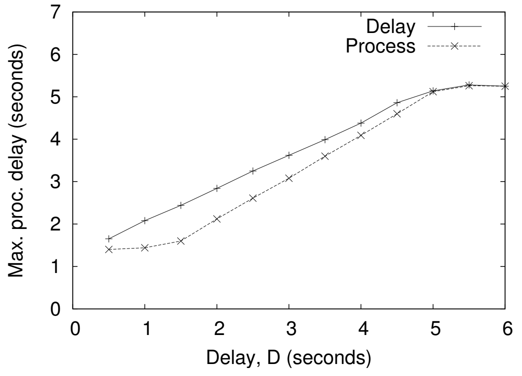
5.4.1 Single-Node Performance

We first examine the availability and consistency of a single Borealis node during UP_FAILURE and STABILIZATION. Because the experiments in this section are highly deterministic, each result is an average of three experiments.

Our measure of availability is $\texttt{Delay}_{\texttt{new}}$, the maximum *added* processing latency for any new output tuple. In most experiments, however, we use a single output stream and plot directly the maximum processing latency $\texttt{Proc}_{\texttt{new}} = \texttt{Delay}_{\texttt{new}} + \texttt{proc}(t)$. Our measure of



(a) $\mathtt{N_{tentative}}$.



(b) $\mathtt{Proc_{new}}$.

Figure 5-7: Benefits of continuously delaying new tuples during failures. During a 5-second failure, the SPE first suspends processing new tuples for a variable time, D, then either processes them as they arrive (Process) or continuously delaying them (Delay). Continuously delaying tuples (Delay) during UP_FAILURE reduces $\mathtt{N_{tentative}}$.

consistency is $\mathtt{N_{tentative}}$, the number of tentative tuples received by the client application.[1]

Handling Most Recent Input Tuples During Failures

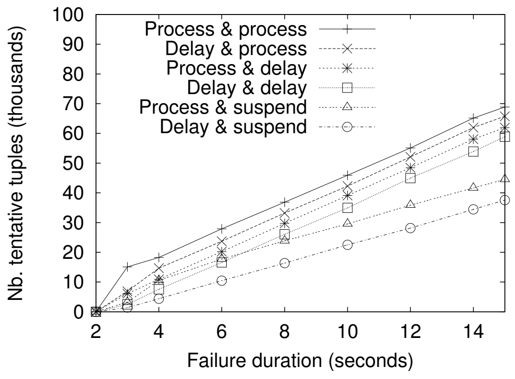
In DPC, each SUnion is given a maximum incremental latency, D, and it must ensure that it processes available input tuples within time D of their arrival. To minimize the number of tentative result tuples, while meeting the above availability requirement, SUnion can handle all failures with a duration shorter than D by suspending processing new tuples until the failure heals. When a failure lasts longer than D, however, the SUnion can no longer suspend processing available input tuples without breaking the low-latency processing requirement. SUnion can, however, either continuously delay new tuples by D or catch-up and process new tuples soon after they arrive. We call these alternatives “Delay” and “Process” and examine their impact on $\mathtt{Proc_{new}}$ and $\mathtt{N_{tentative}}$.

We cause a 5-second failure, vary D from 500 ms to 6 seconds (this set-up is equivalent to holding D fixed and varying the failure duration), and observe $\mathtt{Proc_{new}}$ and $\mathtt{N_{tentative}}$ until after STABILIZATION completes. Figure 5-7 shows the results. From the perspective of DPC's optimization, Delay is better than Process as it leads to fewer tentative tuples. Indeed, with Process, as soon as the initial delay is small compared with the failure duration ($D \leq 4$ s for a 5-second failure), the node has time to catch-up and produce a number of tentative tuples almost proportional to the failure duration. The $\mathtt{N_{tentative}}$ graph approximates a step function. In contrast, Delay reduces the number of tentative tuples proportionally to D. With both approaches, $\mathtt{Proc_{new}}$ increases approximately linearly with D.

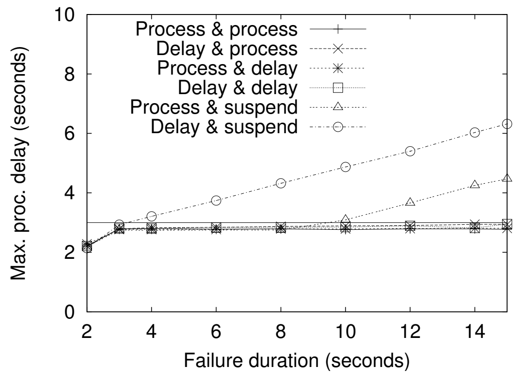
Handling Most Recent Input Tuples During Stabilization

In UP_FAILURE, a node should thus continuously delay new input tuples up to the maximum delay, D. We now examine how the node should process new tuples during STABILIZATION. Three techniques are possible. The node can either suspend new tuples (Suspend), or have a second version of the SPE continue processing them with or without delay (Delay or

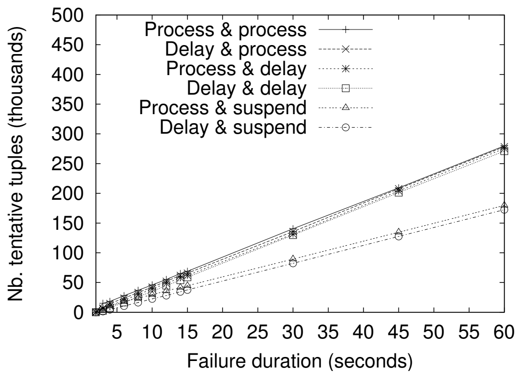
[1] $\mathtt{proc}(t)$, $\mathtt{Delay_{new}}$, and $\mathtt{N_{tentative}}$ are defined more precisely is Section 4.1.



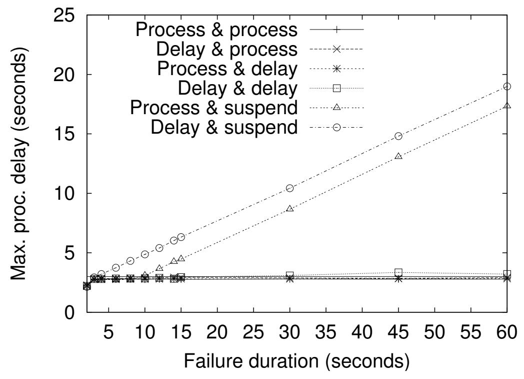
(a) $\mathtt{N_{tentative}}$, short failures.



(b) $\mathtt{Proc_{new}}$, short failures.



(c) $\mathtt{N_{tentative}}$, long failures.



(d) $\mathtt{Proc_{new}}$, long failures.

Figure 5-8: Availability and consistency resulting from delaying, processing, or suspending new tuples during UP_FAILURE and STABILIZATION. X-axis starts at 2 s. Delay & Delay meets the availability requirement for all failure durations, while producing the fewest tentative results.

Process). Because delaying tuples during UP_FAILURE affects the results of suspending or delaying tuples during STABILIZATION, we examine all six possible combinations: (1) delaying or (2) processing new tuples during UP_FAILURE then (a) suspending, (b) delaying, or (c) processing new tuples during STABILIZATION. Our goal is to determine the failure durations when each combination of techniques produces the fewest tentative tuples without breaking the availability requirement.

In this experiment, we use $D = 2.7$ s. Since $X = D = 2.7$ s, and the normal processing latency is below 300 ms, $\mathtt{Delay_{new}} < X$ when $\mathtt{Proc_{new}} < 3$ s. We increase the input rate to 4500 tuples/s to emphasize differences between approaches.

Figure 5-8 shows $\mathtt{Proc_{new}}$ and $\mathtt{N_{tentative}}$ for each combination and for increasing failure durations. We only show results for failures up to 1 minute. Longer failures continue the same trends. Because suspending is optimal for short failures, all approaches suspend for time-period D, and produce no tentative tuples for failures below this threshold.

Processing new tuples without delay during UP_FAILURE and STABILIZATION (Process & Process) ensures that the maximum delay remains below D independent of failure duration. This baseline combination, however, produces the most tentative tuples as it produces them for the duration of the whole failure and reconciliation. The SPE can reduce the number of tentative tuples without hurting $\mathtt{Proc_{new}}$, by delaying new tuples during STA-

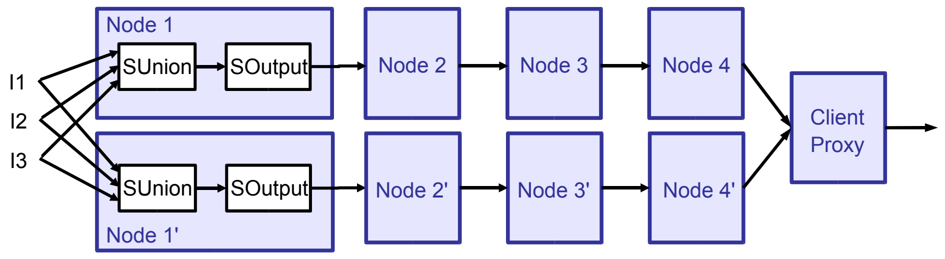


Figure 5-9: Setup for experiments with a distributed SPE.

BILIZATION (Process & Delay), during UP_FAILURE, or in both states (Delay & Delay). This last combination produces the fewest tentative tuples.

Delaying tuples in UP_FAILURE then suspending them during STABILIZATION (Delay & Suspend) is unviable because delaying during failures brings the system on the verge of breaking the availability requirement. Suspending during reconciliation then adds a delay proportional to the reconciliation time. It is, however, possible to process tuples as they arrive during failures in order to have time to suspend processing new tuples during reconciliation (Process & Suspend). This approach is viable only as long as reconciliation is shorter than D. Once reconciliation becomes longer than D (for a failure duration around 8 seconds), Process & Suspend causes $\texttt{Delay}_{\texttt{new}}$ to exceed D. With Process & Suspend, the savings in terms of $\texttt{N}_{\texttt{tentative}}$ is proportional to the reconciliation time, which is thus always less then D, otherwise the approach breaks the availability requirement. With Delay & Delay, the savings is always equal to D. Delay & Delay is thus always equivalent or better than Process & Suspend.

In summary, to meet the availability requirement for long failures, nodes *must* process new tuples not only during UP_FAILURE but also during STABILIZATION. Nodes can produce fewer tentative tuples, however, by always running on the verge of breaking that requirement (Delay & Delay).

5.4.2 Multiple Nodes

We now examine the performance of the above techniques in a distributed setting. Because it is never advantageous to suspend processing during reconciliation and because suspending breaks the availability requirement for long failures, we only compare two techniques: continuously delaying new tentative tuples (Delay & Delay) and processing tentative tuples almost as they arrive (Process & Process). We expect that delaying tuples as much as possible will result in a longer processing latency but will lead to fewer tentative tuples on the output stream. We show that in contrast to our expectations, delaying tentative tuples does not improve consistency, except for short failures.

Figure 5-9 shows the experimental setup: a chain of up to four processing nodes. Each node runs a trivial diagram composed of just an SUnion and an SOutput. Such a diagram ensures that when we measure processing latencies, we do not measure any effects of the operator scheduler. More complex diagrams would simply have longer reconciliation times. As in the previous section, the first SUnion merges the inputs from three streams. Subsequent SUnions process a single input stream. Each pair of processing nodes runs on a

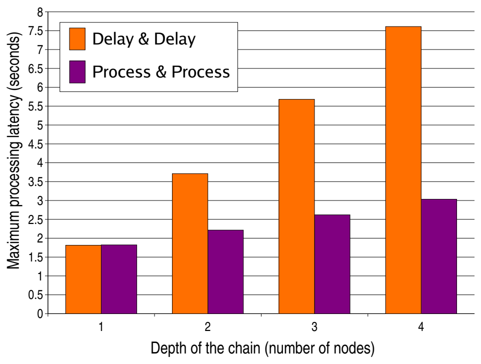


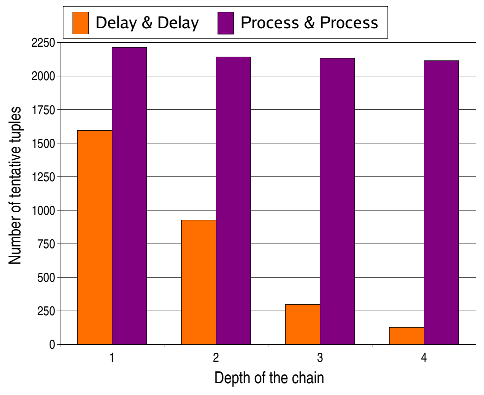
Figure 5-10: `Proc`$_{\texttt{new}}$ for a sequence of processing nodes. Each node runs a single SUnion with $D = 2$ s. Results are independent of failure durations. Each result is the average of 10 experiments (Standard deviations, σ, are within 3% of means). Both techniques meet the required availability of 2 seconds per-node. Process & Process provides a significantly better availability.

different physical machine. The data sources, first processing nodes, proxy, and client all run on the same machine.

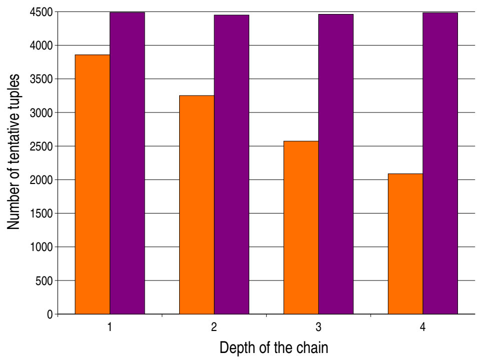
To cause a failure, we temporarily prevent one of the input streams from producing boundary tuples. We chose this technique rather than disconnecting the stream to ensure that the output rate at the end of the chain is the same with and without the failure. Keeping the output rate the same makes it easier to understand the dynamics of the approach. The aggregate input rate is 500 tuples/second. We cause failures of different durations between 5 seconds and 60 seconds.

Figure 5-10 shows the measured availability of the output stream during UP_FAILURE and STABILIZATION as we increase the depth of the chain. We show results only for a 30-second failure because the end-to-end processing latency is independent of failure duration, except for very short failures. In this experiment, we assign a maximum delay, $D = 2$ s, to each SUnion. The end-to-end processing latency requirement thus increases linearly with the depth of the chain. It is $2n$ seconds for a chain of n nodes. Both Delay & Delay and Process & Process provide the required availability, but the availability is significantly better with Process & Process.

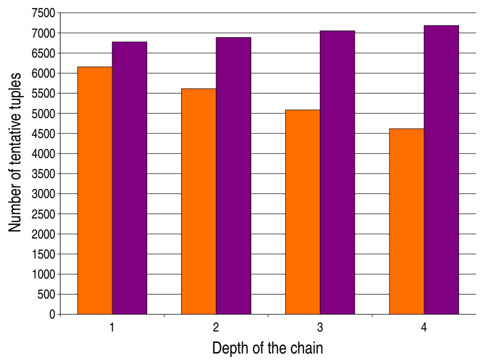
We now explain the results in more detail. For Delay & Delay, each node in the chain delays its input tuples by D before processing them. The processing latency thus increases by a fixed amount for every consecutive processing node. For Process & Process, we would expect the *maximum* processing latency to be the same. As a failure occurs and propagates through the chain, each node that sees the failure first delays tuples by D, before processing *subsequent* tuples almost as they arrive. The first bucket without a boundary tuple should thus be delayed by each node in sequence. Instead, for Process & Process, the end-to-end processing latency is closer to the delay imposed by a single processing node. Indeed, as the failure occurs, *all nodes suspend processing at the same time* because when the first node suspends processing it also suspends producing boundary tuples. All nodes stop receiving boundary tuples at the same time. Thus, after the initial 2 second delay, tuples stream through the rest of the chain with only a small extra delay per node. This observation is important because it affects the assignment of delays to SUnions, as we see later in this



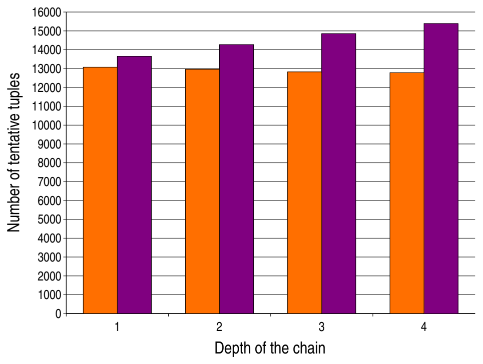
(a) 5-second failure.



(b) 10-second failure.



(c) 15-second failure.



(d) 30-second failure.

Figure 5-11: $\texttt{N}_{\texttt{tentative}}$ during *short* failures and reconciliations. Each node runs a single SUnion with $D = 2$ s. Each result is the average of 10 experiments. (Standard deviations, σ, are within 4 % of means except for 5-second failure with Delay & Delay, where σ is about 16 % of the mean for deep chains). Delaying improves consistency only by a fixed amount approximately proportional to the total delay through the chain.

section. In summary, Process & Process achieves a significantly better availability (latency), although both techniques meet our requirement.[2]

We now compare the number of tentative tuples produced on the output stream to examine how much delaying helps improve consistency (*i.e.*, reduces $\texttt{N}_{\texttt{tentative}}$).

Figure 5-11 shows $\texttt{N}_{\texttt{tentative}}$ measured on the output stream for each technique and failures up to 30 seconds in duration. In these examples, failures are relatively short and reconciliation is relatively fast. The main conclusion we can draw is that delaying tentative tuples reduces inconsistency. The gain is constant and approximately proportional to the total delay through the chain of nodes (*i.e.*, the gain increases with the depth of the chain). However, this also means that the relative gains actually *decrease* with failure duration (*i.e.*,

[2]The processing latency increases by a small amount per-node even with Process & Process because, in the current implementation, SUnions do not produce tentative boundaries. Without boundaries, an SUnion does not know how soon a bucket of tentative tuples can be processed. We currently require SUnions to wait for a minimum, 300 ms, delay before processing a tentative bucket. Using tentative boundaries would enable even faster processing of tentative data and latency would remain approximately constant with the depth of the chain (increasing only as much as the actual processing latency).

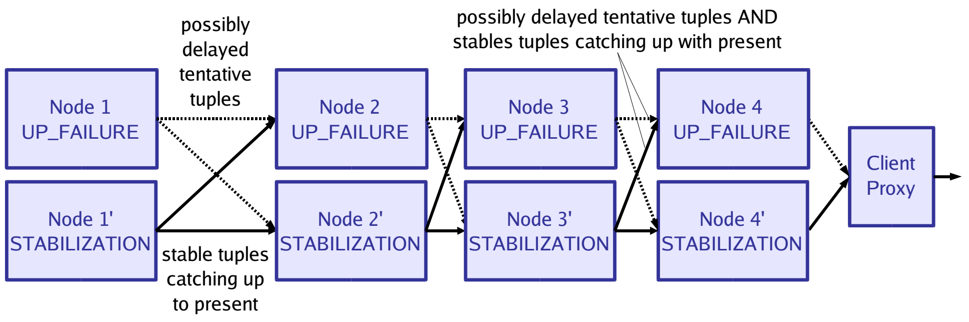


Figure 5-12: Dynamics of state reconciliation through a chain of nodes. When a failure heals, at least one replica from each set enters the STABILIZATION state. Every node thus receives simultaneously potentially delayed tentative tuples and stable tuples, which slowly catch-up with current execution.

as reconciliation gets longer). At 30 seconds, the gains start to become insignificant.

To discuss the results in more detail, we must first examine the dynamics of state reconciliation through a chain of nodes. In the implementation, a node enters the STABILIZATION state as soon as one previously tentative bucket becomes stable. As a failure heals, the first node starts reconciling its state and starts producing corrections to previously tentative tuples. Almost immediately, the downstream neighbors receive sufficiently many corrections for at least one bucket to become stable, and one of the downstream neighbors enters the STABILIZATION state. Hence, in a chain of nodes, as soon as a failure heals, *at least one replica of each node starts to reconcile its state.* These replicas form a chain of nodes in the STABILIZATION state. The other replicas form a parallel chain of nodes that remain in the UP_FAILURE state, as illustrated in Figure 5-12. This approach has two important effects.

First, the total reconciliation time increases only slightly with the depth of the chain because all nodes reconcile roughly at the same time. For Process & Process, the number of tentative tuples is proportional to the failure duration plus the stabilization time. For 30-second and 15-second failures, we clearly see the number of tentative tuples increase slightly with the depth of the chain.[3]

Second, because a whole sequence of nodes reconciles at the same time, the last replica in the chain that remains in UP_FAILURE state receives both delayed tentative tuples from its upstream neighbor in UP_FAILURE state and *most recent corrected stable tuples* from the upstream neighbor in the STABILIZATION state. The last node in the chain processes these most recent stable tuples as tentative because it is still in the UP_FAILURE state. For short failures, reconciliation is sufficiently fast that the last node in UP_FAILURE state does not have time to get to these most recent tuples before the end of STABILIZATION at its replica. The last node in the chain only processes the delayed tentative tuples. Therefore, for Delay & Delay and short failures, the number of tentative tuples decreases with the depth of the chain. It decreases proportionally to the total delay imposed on tentative tuples.

The result is different for long failures, though. Figure 5-13 shows $\mathtt{N_{tentative}}$ on the output stream when the failure lasts 60 seconds. The figure shows that the benefits of

[3]For short 5-second failures, the number of tentative tuples decreases with the depth of the chain even for Process & Process, because a small number of tentative tuples is dropped every time a node switches upstream neighbors. For short failures, these small drops are not yet offset by increasing reconciliation times.

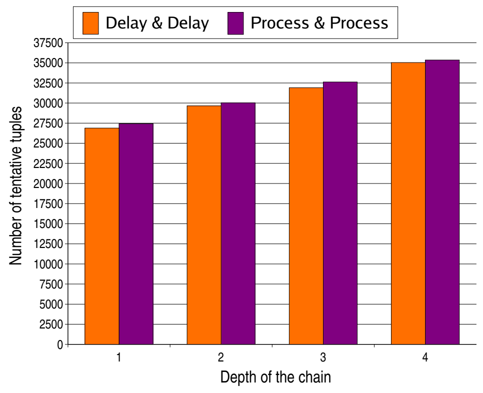


Figure 5-13: $\mathtt{N_{tentative}}$ during a *long* failure and reconciliation. Each node runs a single SUnion with $D = 2$ s. Each result is the average of 10 experiments (Standard deviations, σ, are within 3% of means). For long failures, delaying does not improve consistency.

delaying almost disappear. Delay & Delay can still produce a little fewer tentative tuples than Process & Process but the gain is negligible and independent of the depth of the chain. The gain is equal to only the delay, D, imposed by the last node in the chain. Therefore, for long failures, delaying sacrifices availability without benefits to consistency.

In summary, in a distributed SPE, the best strategy for processing tentative tuples during failure and reconciliation is to first suspend all processing hoping that failures are short. If failures persist past the maximum delay, D, the new best strategy is to continuously delay new input tuples as much as possible. As the failure persists and an increasingly large number of tuples will have to be re-processed during state reconciliation, delaying is no longer beneficial, and nodes might as well improve availability by processing tuples without any delay. This technique could easily be automated, but as we show later an even better strategy exists.

5.4.3 Assigning Delays to SUnions

In the previous sections, we assumed that each SUnion was assigned a fixed delay, D. We now examine how to divide an end-to-end maximum added processing latency, X, among the many SUnions in a query diagram.

We first study a chain configuration and examine two different delay assignment techniques: uniformly dividing the available delay among the SUnions in the chain or assigning each SUnion the *total incremental delay*. For a total incremental delay of 8 seconds, and a chain of four processing nodes, the first technique assigns a delay of 2 seconds to each node. This is the technique we have been using until now. In contrast, with the second technique, we assign the complete 8 second delay to each SUnion. In the experiments, we actually use 6.5 s instead of 8 s because queues start to form when the initial delay is long.

Figure 5-14 shows the maximum processing latency for a chain of 4 nodes and the two different delay assignment techniques. Figure 5-15(a) shows $\mathtt{N_{tentative}}$ for the same configurations. The left and middle graphs repeat the results from the previous section: each SUnion has $D = 2$ s and either delays tentative tuples or processes them without delay. The graphs on the right show the results when each SUnion has $D = 6.5$ s and processes tuples without delay.

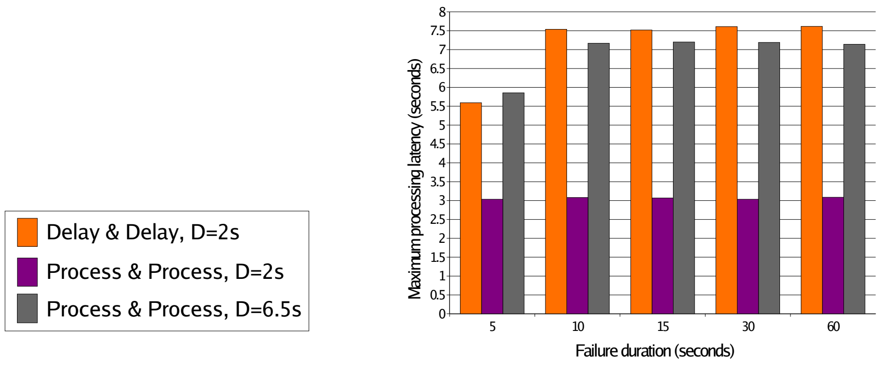
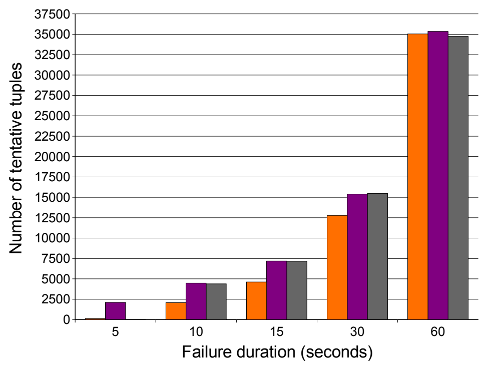


Figure 5-14: $\texttt{Proc}_{\texttt{new}}$ **for a sequence of four processing nodes and different failure durations.** Each result is the average of 10 experiments. (Standard deviations, σ, are within 3% of means). It is possible to assign the full incremental delay to each SUnion in a chain and still meet the availability requirement.

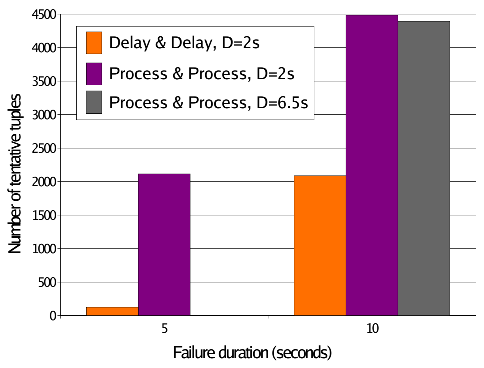
Interestingly, assigning the total delay to each SUnion meets the required availability independently of the depth of the chain. Indeed, when a failure occurs, *all SUnions downstream from the failure suspend at the same time*. After the initial delay, however, nodes must process tuples as they arrive to meet the availability requirement (alternatively, they could revert to delaying by only 2 s). At a first glance, though, assigning the total delay to each SUnion appears to have the worse availability of Delay & Delay and the worse number of tentative tuples of Process & Process. Figure 5-15(b) shows a close-up on the results for only the 5-second and 10-second failures. For the 5-second failure, when $D = 6.5$ s, the system produces not a single tentative tuple. Since we can expect such short, intermittent failures to be frequent, assigning the maximum incremental processing latency to each SUnion thus enables a system to cope with the longest possible failures without introducing inconsistency and still meeting the required availability. Additionally, the high maximum processing latency when assigning the whole delay to each SUnion affects only the tuples that enter the system *as the failure first occurs*. After the initial delay, nodes process subsequent tuples without any delay. The availability thus goes back to the low processing latency of Process & Process.

Hence, the best delay assignment strategy is to assign the total incremental processing latency to each SUnion in the query diagram. Such a delay assignment meets the required availability, while masking the longest failures without introducing inconsistency. The approach thus minimizes $\texttt{N}_{\texttt{tentative}}$ for all failures shorter than D.

In a more complex graph configuration, the same result holds. When an SUnion first detects a failure, all downstream SUnions suspend processing at the same time. The initial delay can thus be equal to X. After the initial delay, however, SUnions can either process tuples without further delay, or they can continuously delay new tuples for a shorter incremental time, D. We have shown that additive delays are not useful for long-duration failures. In a graph configuration, they are also difficult to optimize. Figure 5-16 illustrates the problem. Every time two streams meet at an operator, their respective accumulated delays depend on the location of upstream failures and the number of SUnions they traversed. There is therefore no single optimal incremental delay that SUnions can impose *individually*, even when delays are assigned separately to each input stream at each SUnion. As shown



(a) Various failure durations.



(b) Short failures only.

Figure 5-15: $\mathtt{N_{tentative}}$ for different delay assignments to SUnions. Process & Process with $D = 6.5$ s is the only technique that can mask the 5-second failure while performing as well as the other approaches for longer failures. Each result is the average of 10 experiments (Standard deviations, σ, are within 11% of means).

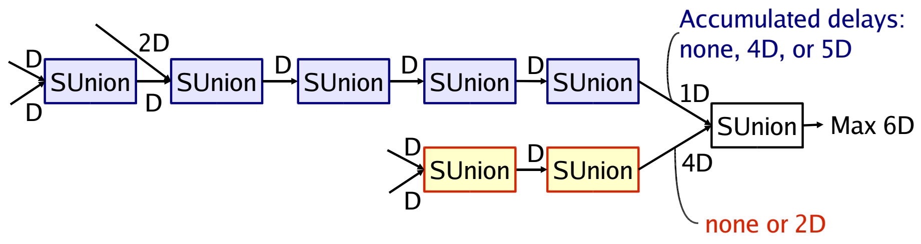


Figure 5-16: Assigning incremental delays to SUnions in a query diagram. The assignment is difficult because accumulated delays depend on the diagram structure and on the failure location.

on the figure, it is possible to produce an assignment guaranteeing that the SPE meets a required availability, but some failure configurations can cause some tentative tuples to be systematically dropped by an SUnion. Pre-defined additive delays are thus undesirable.

To circumvent this problem, the SPE could encode accumulated delays inside tuples and SUnions could dynamically adjust the incremental delays they impose based on the total accumulated delays so far. This type of assignment would lead to a new set of possible delay assignment optimizations, and we leave it for future work.

In this section, we investigated trade-offs between availability and consistency for a single SPE and for simple distributed deployments. We showed that in order to minimize inconsistency while meeting a desired availability level, when first detecting a failure, each SUnion should suspend processing for as long as the total incremental latency specified by the application. After this initial delay, SUnions should process tuples as they arrive because independent incremental delays do not improve consistency after long-duration failures and, for complex query diagrams, may cause a greater number of tuples to be dropped because of mismatches in accumulated delays on different streams.

Approach	$\texttt{Delay}_{\texttt{new}}$ (*i.e.*, reconciliation time)	CPU Overhead	Memory Overhead
Checkpoint	$Sp_{\texttt{copy}} + (F + 0.5l)\lambda p_{\texttt{proc}}$	$\frac{Sp_{\texttt{copy}}}{l}$	$S + (l + F)\lambda_{in}$
Undo	$S(p_{\texttt{comp}} + p_{\texttt{proc}}) + (F + 0.5l)\lambda(p_{\texttt{comp}} + p_{\texttt{proc}})$	$\frac{Sp_{\texttt{comp}}}{l}$	$S + (l + F)\lambda$

Table 5.1: Performance and overhead of checkpoint/redo and undo/redo reconciliations.

5.5 State Reconciliation

We now study the performance of state reconciliation through checkpoint/redo and undo/redo. We compare the time it takes for an SPE to reconcile its state using each approach. We also compare their CPU and memory overhead. We introduced state reconciliation with checkpoint/redo in Section 4.7.1. We describe undo/redo in Appendix B.

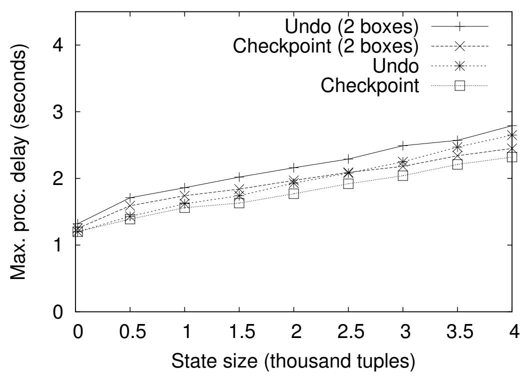
Independently of the state reconciliation technique, nodes must buffer their output tuples until all replicas of all their downstream neighbors receive these tuples. We discussed the overhead of these buffers in Section 4.9. Since it is the same overhead for both techniques, we ignore it in this section. Similarly, DPC also introduces some additional minor overheads, which we discuss in the next section. In this section, we focus only on comparing checkpoint/redo with undo/redo.

Table 5.1 summarizes the analytical reconciliation times and overheads. In the table, $p_{\texttt{comp}}$ is the time to read and compare a tuple. $p_{\texttt{copy}}$ is the time to copy a tuple. $p_{\texttt{proc}}$ is the time an operator takes to process a tuple. We assume $p_{\texttt{proc}}$ is constant but it may increase with an operator's state size. S is the size of the state of the query diagram fragment (*i.e.*, the sum of the state sizes of all operators including their input queues). F is the failure duration. l is the interval between checkpoints or between stream marker computations. λ is the average aggregate tuple rate on all input and intermediate streams. λ_{in} is the aggregate rate only on input streams to the node.

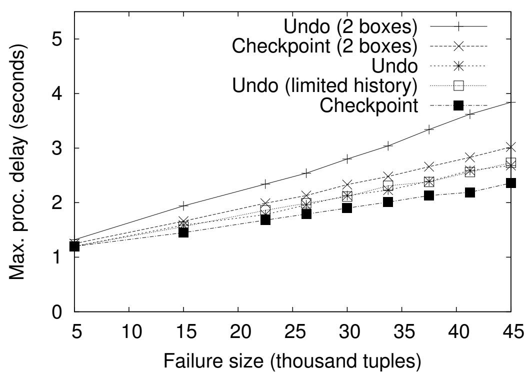
5.5.1 Reconciliation Time

For checkpoint/redo, reconciliation consists of reinitializing the state of the query diagram from the last checkpoint taken before the failure, and reprocessing all tuples since then. The average reconciliation time is thus the sum of $Sp_{\texttt{copy}}$, the time to copy the state of size S, and $(F + 0.5l)\lambda p_{\texttt{proc}}$, the average time to reprocess all tuples since the last checkpoint before failure. In the experiments, we use approximately constant input rates.

For undo/redo, reconciliation consists of processing the undo history up to the correct stream markers and reprocessing all tuples since then. Producing an UNDO tuple takes a negligible time. We assume that the number of tuples necessary to rebuild an operator's state is equal to the state size. This assumption is realistic only for convergent-capable operators. We also assume that stream markers are computed ever l time units. The average number of tuples in the undo log that must be processed backward then forward is thus: $(F + 0.5l)\lambda + S$. The first term accounts for tuples that accumulate during the failure, since the last marker computation. The second term accounts for the tuples that are needed to rebuild the pre-failure state. In the backward direction, each tuple must be read and compared to the UNDO tuple. In the forward direction, each tuple must actually be processed. The average reconciliation time is thus: $S(p_{\texttt{comp}} + p_{\texttt{proc}}) + (F + 0.5l)\lambda(p_{\texttt{comp}} + p_{\texttt{proc}})$.



(a) $\texttt{Proc}_{\texttt{new}}$ for increasing state size.



(b) $\texttt{Proc}_{\texttt{new}}$ for increasing failure size starting at 5000 tuples.

Figure 5-17: Performance of checkpoint/redo and undo/redo reconciliations. Checkpoint/redo outperforms undo/redo in all configurations.

Because $p_{\texttt{comp}}$ is small compared with $p_{\texttt{copy}}$ and $p_{\texttt{proc}}$, and because $p_{\texttt{proc}} > p_{\texttt{copy}}$, we expect checkpoint/redo to perform better than undo/redo, but the difference should appear only when the state of the query diagram is large.

Figure 5-17 shows the experimental $\texttt{Proc}_{\texttt{new}}$ as we increase the state size, S, of the query diagram (Figure 5-17(a)) or the number of tuples to re-process, $F\lambda$ (Figure 5-17(b)). In this experiment, F is 5 seconds and we vary λ. For both approaches, the time to reconcile increases linearly with S and $F\lambda$. When we vary the state size, we keep the tuple rate low at 1000 tuples/s. When we vary the tuple rate, we keep the state size at only 20 tuples. As expected and as shown in Figure 5-17(a), undo/redo takes longer to reconcile primarily because it must rebuild the state of the query diagram ($Sp_{\texttt{proc}}$) rather than recopy it ($Sp_{\texttt{copy}}$). Interestingly, even when we keep the state size small and vary the number of tuples to reprocess (Figure 5-17(b)), checkpoint/redo beats undo/redo, while we would expect the approaches to perform the same ($\propto (F + 0.5l)\lambda p_{\texttt{proc}}$). The difference is not due to the undo history (*i.e.*, to $(F + 0.5l)\lambda p_{\texttt{comp}}$), because the difference remains even when operators do not buffer any tentative tuples in the undo buffer (Undo "limited history" curve). In fact, an SPE always blocks for D (1 s in this experiment) before going into UP_FAILURE. For checkpoint/redo, because the node checkpoints its state every 200 ms, it always checkpoints the pre-failure state and avoids reprocessing on average $0.5l\lambda$ tuples, which corresponds to tuples that accumulate between the checkpoint and the beginning of the failure. Undo/redo always pays this penalty, as stream markers are computed only when an operator processes new tuples. Although, we could modify operators to always compute stream markers just before processing their first tentative input tuple.

As shown in Figure 5-17, for both approaches, splitting the state across two operators in series (curves labeled "2 boxes"), simply doubles λ and increases curve slopes.

5.5.2 CPU Overhead

In theory, checkpoint/redo has higher CPU overhead than undo/redo because checkpoints are more expensive than scanning the state of an operator to compute stream markers (Figure 5-18). However, because a node has time to checkpoint its state when going into UP_FAILURE state, it can *perform checkpoints only at that point* and avoid the overhead of

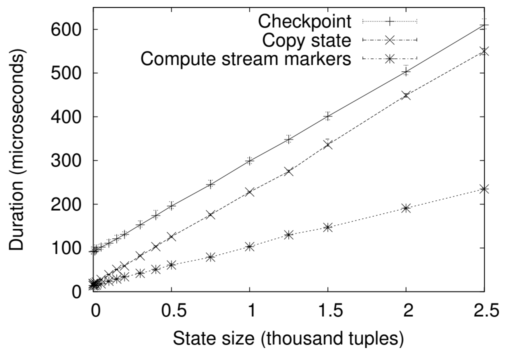


Figure 5-18: CPU overhead of checkpoint/redo and undo/redo reconciliations. Checkpoints are more expensive than stream marker computations because they involve recopying the state of all operators and their input queues.

periodic checkpoints at runtime. Stream markers can also be computed only once a failure occurs. Hence, both schemes can avoid CPU overhead in the absence of failures.

5.5.3 Memory Overhead

To compare the memory overhead of the approaches, we first assume that nodes buffer all tuples (stable and tentative) during failures.

With checkpoint/redo, a node must keep in memory its checkpointed state and all tuples that accumulate on input streams since the last checkpoint. The overhead is thus: $S+(l+F)\lambda_{in}$, where λ_{in} is the aggregate *input* rate. However, since nodes always checkpoint their state when entering UP_FAILURE, $l = 0$, and the overhead is actually only $S + F\lambda_{in}$,

Checkpoint/redo must always reprocess all tuples since the beginning of the failure. With undo/redo, it is possible to undo only the suffix of tentative tuples that actually changed. Correcting a suffix of tentative tuples is equivalent to recovering from a shorter failure than F. This possibility, however, causes undo/redo to incur a high memory overhead, because all operators must keep an undo buffer. Even stateless operators (*e.g.*, Filter) need an undo buffer because they must remember when they produced each output tuple in order to produce the appropriate UNDO tuple. Only operators that produce exactly one output tuple for each input tuple (*e.g.*, Map) need not keep an undo buffer, because they can produce the correct UNDO from the one they receive. Even if we assume that a query diagram needs no more than S tuples to rebuild its state, the total overhead is then: $S + (l + F)\lambda$.

The situation is different if nodes buffer only stable tuples. With this approach, all tentative tuples must always be corrected (not just the suffix that actually changed). However, both techniques realize great savings in memory overhead, making the approach preferable.

When a failure occurs, suppose a fraction α of input streams becomes tentative. With checkpoint/redo, a node only needs to buffer tuples that arrive on the $(1 - \alpha)$ stable input streams. The overhead is then:

$$S + (1 - \alpha)F\lambda_{in}. \tag{5.1}$$

With undo/redo, all stateful operators must still buffer sufficiently many tuples to rebuild their pre-failure states, which remains a significant memory overhead. However, only operators that have both stable and tentative inputs during the failure need to buffer any additional tuples. They must buffer tuples on their stable inputs. The overhead of undo/redo can thus vary greatly between query diagrams.

In summary, both schemes save significantly on overhead when only stable tuples are buffered during failures. Undo/redo has the potential of even greater savings compared with checkpoint/redo, but keeping sufficiently many tuples to rebuild the pre-failure state continues to impose a very high overhead. When all tentative tuples are corrected after each failure, checkpoint/redo always significantly outperforms undo/redo in terms of reconciliation time. Finally, an interesting advantage of the undo-based approach, which we have ignored in the above analysis, is that reconciliation propagates *only* on paths affected by failures. It would be interesting to enhance the checkpoint-based approach with the same feature.

5.5.4 Optimized Checkpoint/Redo Approach

The above results lead us to conclude that the best state-reconciliation technique should combine the benefits of (1) recovering the SPE state from a checkpoint, (2) reconciling only operators affected by failures, and (3) buffering tuples only at operators with at least one stable and one tentative input.

We propose an approach based on checkpoint/redo that provides all these benefits. This approach is not yet implemented in Borealis. The idea is for operators to use checkpoints to save and recover their pre-failure states. Instead of checkpointing and recovering the state of the whole query diagram, however, operators checkpoint and recover their states individually. To perform such localized checkpoints, an operator checkpoints its state only when it is about to process its first tentative tuple. To recover only on paths affected by a failure, just as in the undo-based technique, SUnions on previously failed inputs push an UNDO tuple into the query diagram before replaying the stable tuples. When an operator receives an UNDO tuple, it recovers its state from its checkpoint and reprocesses all stable tuples that follow the UNDO. Interestingly, with this technique, UNDO tuples no longer need to explicitly identify a tuple because all tentative tuples are always undone.

With this approach, only operators that see a failure checkpoint their state. The only SUnions that need to buffer tuples are those with both tentative and stable inputs, and they only need to buffer tuples on the stable inputs (the only tuples they are in charge of replaying). Finally, only those operators that previously experienced a failure participate in the state reconciliation.

5.6 Overhead and Scalability

Buffering tuples during failures in order to replay them during state reconciliation is the main overhead of DPC. There are, however, a few additional sources of overhead. We now discuss these secondary overheads.

Tuple serialization is the main additional cause of overhead. If the sort function requires an SUnion to wait until a bucket is stable before processing tuples in that bucket, the processing delay of each SUnion increases linearly with the boundary interval (we assume this interval is equal to the bucket size). Table 5.2 shows the average end-to-end delay from nine 20 s experiments and increasing bucket sizes. Tuple serialization also introduces memory overhead because SUnions must buffer tuples before sorting them. This memory overhead increases proportionally to the number of SUnion operators, their bucket sizes, and the rate of tuples that arrive into each SUnion. However, because we typically chose

Boundary interval (ms)	50	100	150	200	250	300
Average processing delay	69	120	174	234	298	327
Stddev of the averages	0.5	4	10	28	55	70

Table 5.2: Latency overhead of serialization.

the tuple_stime to correspond to the expected order of tuples on input streams, we could significantly reduce overhead by allowing SUnions to output tuples as soon as they receive input tuples with higher tuple_stime values on all their input streams, reducing both memory and latency overhead. This optimization requires that all operators process and produce tuples in increasing tuple_stime values.

Other overheads imposed by DPC are negligible. Operators must check tuple types and must process boundary tuples. The former is negligible while the latter requires, in the worst case, a scan of all tuples in the operator's state, and, in the best case, requires simply that the operator propagates the boundary tuple. Each SOutput must also save the last stable tuple that it sees in every burst of tuples that it processes.

Finally, DPC relies on replication. It increases resource utilization proportionally to the number of replicas. These replicas, however, can actually improve runtime performance by forming a content distribution network, where clients and nodes connect to nearby upstream neighbors rather than a single, possibly remote, location.

5.7 Lessons Learned

Our analysis and experiments lead us to draw the following conclusions about fault-tolerance in a distributed SPE.

First, even when operators are deterministic, to ensure that all replicas of the same operator remain mutually consistent (*i.e.*, they go through the same states and produce the same output tuples), it is necessary to introduce boundary tuples into streams. These tuples serve both as punctuations and heartbeats. The punctuation property enables all replicas of an operator to sort tuples in the same deterministic order. The heartbeat property enables replicas to process tuples without much delay, even in the absence of data on some input streams.

Second, our design decision to put each node in charge of maintaining its own availability and consistency works well for complex failure scenarios involving concurrent failures and failures during recovery. Each node monitors the availability and consistency of its input streams and keeps track of the data it receives. No matter when failures occur, a node can readily request the missing data from the most appropriate replica. DPC thus avoids complex communication protocols between nodes.

Third, one of the greatest challenges of DPC is the requirement to continuously maintain availability. Because DPC guarantees that tuples are never delayed more than a pre-defined bound, every time a node needs to process or even simply receive old input tuples, it must continue processing the most recent input tuples. To maintain availability at all times, it is thus necessary to perform all replays and corrections in the background.

Fourth, to ensure eventual consistency nodes must buffer tuples during failures. Nodes can avoid buffering tentative tuples if all these tuples are always corrected after failures heal. Because buffers cannot grow without bound, even when buffering only stable tuples, sufficiently long failures require nodes to truncate their buffers or block. Convergent-capable query diagrams are therefore most suitable for applications that favor availability over consistency. With these types of query diagrams, nodes can keep fixed-sized buffers, yet, after failures heal, tuples in these buffers suffice to rebuild a consistent state and correct the latest tentative tuples.

Fifth, to reconcile the state of a node, it is faster for operators to restart from a checkpoint rather than rebuild their state dynamically. Additionally, runtime overhead reduces significantly when all tentative tuples are always corrected and nodes buffer only stable tuples. Because reconciling only the state of operators affected by failures further reduces overhead and improves availability, we proposed an approach based on checkpoint/redo that limits checkpoints and reconciliation to paths affected by failures.

Sixth, DPC enables users to set the trade-off between availability and consistency, by specifying a maximum incremental processing latency that gets distributed across the SUnions in the query diagram. We find that giving the total added delay to each SUnion and requiring them to use it all when a failure first occurs, enables the system to handle longer failures without introducing inconsistency, while still achieving the required availability.

Finally, in a distributed SPE, the deployment of operators to processing nodes affects fault-tolerance. With non-blocking operators, every additional input stream to a node increases the chances of an upstream failure cannot be masked. With blocking operators, every additional input stream increases the chance of a total failure, especially in deployments over deep chains of nodes. In all cases, increasing the degree of replication improves the probability that the system masks a set of simultaneous failures, but the replicas must be located at different sites to increase the likelihood that their failures are not correlated.

5.8 Limitations and Extensions

We now discuss the limitations of DPC and possible extensions to address these limitations.

5.8.1 Non-determinism

DPC handles only deterministic operators. This restriction enables us to ensure that all replicas of the same operator remain mutually consistent simply by ensuring that they all process tuples in the same order. DPC maintains the latter property by inserting SUnion operators into the query diagram and adding boundary tuples to streams.

With non-deterministic operators, replicas states can diverge even if they process tuples in the same order. A possible approach to handling non-deterministic operators is to designate one replica as a primary. The primary produces a stream of tuples, called determinants [50], each tuple encoding the result of one non-deterministic event. This stream of tuples forms an additional input at other replicas. The challenge is that all replicas go into UP_FAILURE state every time the primary experiences a failure. An alternate technique is to add extra machinery to elect a new primary, every time an old primary fails. To ensure that replicas agree, at any time, on which node is the primary, the election should use a consensus protocol such as Paxos [99], and this approach has high overhead.

5.8.2 Dynamic Modifications of the Query Diagram

Currently, DPC assumes a static query diagram and a static deployment of the diagram: *I.e.*, we assume that all data sources, operators, and clients remain fixed through the execution. We also assume a static assignment of operators to processing nodes and a static choice of replicas for each node. An example of a dynamic change to the query diagram is the addition or removal of operators or data sources. An example of a dynamic change to the deployment is the addition or removal of some replicas of a node or the movement of an operator from one processing node to another. Because we assume a static assignment, we further assume that all failures are transient. A failed node, connection, or data source eventually recovers and re-joins the system. We consider it to be a dynamic change when a failed node or data source does not re-join the system. A direct area of future work is to extend DPC to support dynamic query diagram and deployment modifications.

A related area of future work is to study operator placement strategies maximizing the probability that a system masks failures or produces at least tentative outputs.

5.8.3 Precision Information

We currently do not include any accuracy information in tentative tuples. Tentative tuples only indicate that a failure occurred and the current value is not the final one. A user cannot distinguish between an upstream failure where 80% of inputs are being processed and one where only 10% of tuples are being processed. An interesting area of future work is to add such precision information to tentative tuples. We could either encode precision bounds (*e.g.*, the value of this average is accurate within 5%) or we could encode information about the failure itself (*e.g.*, the result is the average from processing only streams 1 through 3).

It might also be interesting to enable applications to specify some integrity constraints that determine the level of failures when streams still carry interesting information or the conditions necessary for two tentative streams that experienced different upstream failures to still be correlated or otherwise merged.

5.8.4 Resource Utilization

More importantly, DPC requires the system to buffer tuples when failures occur and re-process them after failures heal. DPC is thus particularly well suited for convergent-capable operators. With these operators, DPC can bound buffers and still ensure that nodes re-build a consistent state and correct the most recent tentative tuples. With other deterministic operators, however, once buffers fill-up, DPC blocks. DPC no longer maintains availability. An interesting area for future work would be to allow these operators to continue processing and later use a separate mechanism to allow replicas to become at least mutually consistent again. This problem is difficult because reconciliation is expensive and disruptive. Ideally, the system should quickly converge to some useful state after a failure heals.

5.8.5 Failures of all Replicas

To ensure eventual consistency, DPC assumes that at least one replica of each processing node remains available at any time. If all replicas of a node fail, the system can still continue processing and can produce tentative tuples. The problem occurs when the failed nodes recover and re-join the system. When query diagrams are convergent-capable, nodes can buffer sufficiently many tuples in their output buffers to enable failed downstream nodes to rebuild a consistent state and correct a pre-defined window of output tuples. With deterministic query diagrams, to support the failure of all replicas of a processing node, replicas would have to take periodic checkpoints and save these checkpoints persistently. Persistent checkpoints would add significant runtime overhead.

5.8.6 Persistent Storage

DPC is geared toward convergent-capable query diagrams that operate on finite transient state. We do not address, for instance, query diagrams that read and write from a persistent store. DPC could handle operators that perform computation over a persistent store by treating the persistent store as a deterministic operator with a state that depends on every single tuples the operator ever processed. Of course, with these types of operators checkpoints become more expensive. We also need to serialize all reads and write operations handled by the persistent store in the same way we serialize tuples processed by other deterministic operators with multiple input streams. We could therefore place an SUnion operator in front of the persistent store. Because the reads and writes would come from different streams that may have boundary tuples with significantly different values, we would have to use a different serialization function in the SUnion to avoid delaying tuples. SUnion operators, however, are general enough to support different serialization functions.

5.9 Summary

In this chapter, we presented the implementation and evaluation of DPC. We showed that DPC handles both single failures and multiple concurrent failures — DPC maintains availability at any time and corrects all tentative tuples in the background without dropping nor duplicating stable results (assuming sufficiently large buffers).

Given a maximum incremental processing latency, DPC handles all failures shorter than the given bound without introducing any inconsistency. DPC achieves this by having all SUnion operators affected by a failure simultaneously suspend processing new tuples, when a failure first occurs. For long failures, we showed that at least one replica of each processing node must continue processing the most recent input data at any time, in order to meet the required availability. DPC achieves this by having nodes correct their inputs in the background and communicate with each other to decide when each node reconciles its state. Furthermore, for long failures, we found that processing tentative tuples without added delay achieves the best availability without hurting consistency compared with delaying tuples as much as possible.

To reconcile the state of an SPE, we compared the performance and overhead of checkpoint/redo with that of undo/redo and proposed a checkpoint-based scheme that combines the benefits of both techniques. To achieve the fast recovery time of checkpoint/redo, we proposed that operators recover their states from checkpoints. To avoid the overhead of periodic checkpoints, we proposed that operators checkpoint their state upon receiving their first tentative input tuples. As in undo/redo, to constrain recovery to query-diagram paths affected by the failure, we proposed to propagate an UNDO tuple through the query diagram, letting each operator recover its state from its checkpoint upon receiving and forwarding the UNDO tuple.

Finally, we have shown that the processing latency overhead of DPC is small, especially when tuples are ordered by increasing timestamp values on all streams, and SUnions simply interleave the tuples that arrive on their many inputs. The main overhead of DPC is due to buffering tuples during failures in order to reprocess them when failures heal. We have shown, in the previous chapter, that this overhead can be bounded when query diagrams are convergent-capable. With bounded buffers and convergent-capable query diagrams, DPC can ensure that all replicas converge to a mutually consistent state and correct the last pre-defined window of tentative tuples produced during the failure.

Overall, we have shown that it is possible to build a single scheme for fault-tolerant distributed stream processing that enables an SPE to cope with a variety of node and network failures, while giving applications the freedom to set their desired trade-off between availability and consistency. This chapter concludes our investigation of fault-tolerant distributed stream processing. In the next chapter, we turn our attention to load management, another important problem in federated, distributed systems.

Chapter 6

Load Management

In this chapter, we study load management, a second important problem that arises in a distributed SPE. We present the *Bounded-Price Mechanism* (BPM), an approach that enables autonomous participants to manage their load variations using offline-negotiated pairwise contracts. Although motivated by stream processing, BPM is applicable to a variety of distributed systems.

In an SPE, as users add and remove operators from the query diagram, and as the input rates and input data distributions vary, the load on the system varies. To improve performance or at least avoid significant performance degradation, the system may have to periodically change the assignment of operators (tasks) to processing nodes.

As we discussed in Chapter 1, dynamic load management is a widely studied problem (*e.g.*, [49, 64, 96]) and various techniques have been proposed to enable a system to reach load distributions that optimize some overall utility, such as throughput, processing latency, or queue sizes. These approaches usually assume a collaborative environment, where all nodes work together to maximize overall system performance. Many distributed systems, however, are now deployed in *federated* environments, where different autonomous organizations own and administer sets of processing nodes and resources.

Federated systems arise when individual participants benefit from collaborating with others. For example, participants may collaborate in order to compose the services they provide into more complete end-to-end services. Cross-company workflows based on Web services [48, 94] and peer-to-peer systems [38, 45, 97, 120, 137, 153, 174] are examples of such federated systems. Another benefit of federation is that organizations can pool their resources together to cope with periods of heavy load (load spikes) without individually having to maintain and administer the computing, network, and storage resources required for peak operation. Examples of such federated systems include computational grids composed of computers situated in different domains [3, 29, 61, 164] and overlay-based computing platforms such as Planetlab [131]. Stream processing applications are naturally distributed and federated because data streams often come from remote geographic locations (*e.g.*, sensor networks deployed in remote areas) and even from different organizations (*e.g.*, market feeds). Data streams can also be composed in different ways by different organizations to create various services.

Load management in a federated environment is challenging because participants are driven by self-interest. Some earlier efforts to enable load management between autonomous participants have proposed the use of computational economies (*e.g.*, [3, 29, 154, 175]). Because none of these schemes has seen a widespread deployment, we posit that these techniques do not entirely solve the problem. Instead, we note that in practice, autonomous participants tend to collaborate by establishing *pairwise agreements* [42, 53, 94, 81, 138, 171, 176, 178].

Inspired by the successful use of pairwise agreements in practice, we propose BPM, a distributed mechanism for managing load in a federated system based on *private pairwise contracts*. Unlike computational economies that use auctions or implement global markets to set resource prices at runtime, BPM is based on *offline* contract negotiation. Contracts set tightly *bounded prices* for migrating each unit of load between two participants and may specify the set of tasks that each is willing to execute on behalf of the other.

With BPM, runtime load transfers occur only between participants that have pre-negotiated contracts, and at a unit price within the contracted range. The load transfer mechanism is simple: a participant moves load to another if the expected local processing cost for the next time-period is larger than the expected payment it would have to make to the other participant for processing the same load (plus the migration cost).

Hence, in contrast to previous proposals, BPM (1) provides privacy to all participants regarding the details of their interactions with others, (2) facilitates service customization and price discrimination, (3) provides simple and lightweight runtime load management using pre-negotiated prices, and as we show, (4) has good system-wide load balance properties. We envision that contracts will be extended to contain additional clauses further customizing the offered services (*e.g.*, performance, security, and availability guarantees), but we leave such extensions for future work.

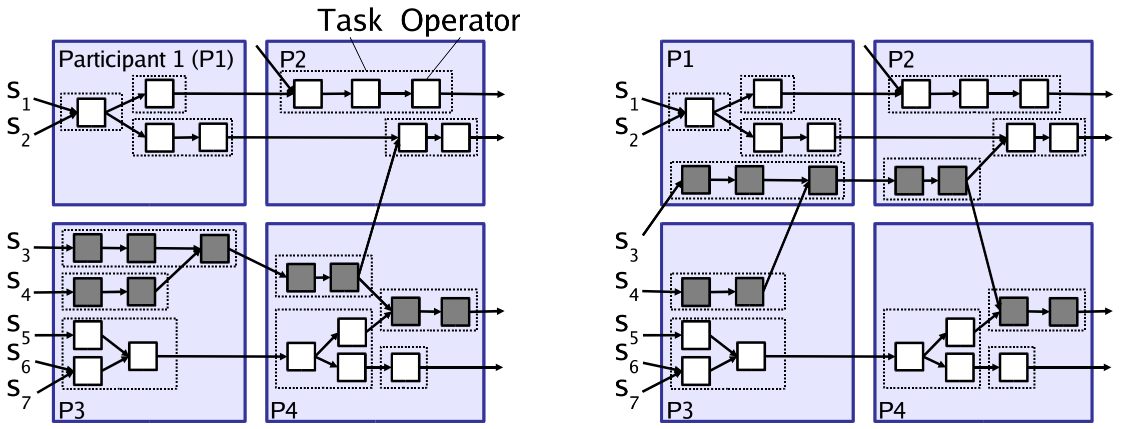
In this chapter, we present BPM and some of its properties. We start by formalizing our problem in Section 6.1 and overview the approach in Section 6.2. Each of the following sections presents one component of BPM. In Section 6.3, we first present the strategies and algorithms for runtime load movements assuming each participant has a set of fixed-price contracts and assuming that load is fixed. We enhance BPM to handle dynamic load variations in Section 6.4. In Section 6.5, we discuss strategies for establishing fixed-price contracts offline. Because fixed-price contracts do not always lead to acceptable allocation, in Section 6.6, we propose to relax the fixed-price constraint and allow contracts to specify small pre-negotiated price ranges. Finally, we discuss how BPM applies to federated SPEs in Section 6.7, and present some properties of BPM in Section 6.8. We describe the implementation of BPM in Borealis and its evaluation through simulations and experiments in the next chapter.

6.1 Problem Definition

In this section, we present the load management problem: we introduce the system model and define the type of load allocation that we would like a load-management mechanism to achieve. We use a federated stream processing engine as an illustrative example and later to evaluate our approach but our goal is to support any federated system.

6.1.1 Tasks and Load

We assume a system composed of a set N of autonomous *participants*. Each participant owns and administers a set of resources that it uses to run tasks on behalf of its own clients. We assume a total set, K, of time-varying tasks. Each task in K originates at some participant in N, where it is submitted by a client. Since we only examine interactions between participants, we use the terms *participant* and *node* interchangeably.



(a) Participants P_2 and P_3 are overloaded.

(b) Some tasks move to the more lightly loaded participants P_1 and P_2.

Figure 6-1: Example of a desired load reallocation in a federated SPE composed of four participants.

A *task* is a long-running computation that requires one or more resources (*e.g.*, memory, CPU, storage, and network bandwidth). In an SPE, a task comprises one or more interconnected operators, as illustrated in Figure 6-1. A query diagram is made up of one or more tasks. We do not address the problem of optimally partitioning a query diagram into tasks. We assume that we are given a set of tasks. We also assume that each task is sufficiently fine-grained that is uses a relatively small fraction of a node's resources. We use a task as the unit of load movement. In an SPE, if a single operator uses a large amount of resources (*e.g.*, a large aggregate), the operator may frequently be partitioned into multiple smaller tasks [147].

6.1.2 Utility

The total amount of resources used by tasks forms the *load* experienced by participants. For each participant, the load imposed on its resources represents a cost. Indeed, when load is low, the participant can easily handle all its tasks. As load increases, it may become increasingly more difficult for the participant to provide a good service to its clients. This, in turn, may cause clients to become dissatisfied: clients can seek a different service provider or they can demand monetary compensation. We assume that a participant is thus able to quantify the processing cost for a given load.

More specifically, we define a real-valued *cost function* of participant, i, as:

$$\forall i \in N, \quad D_i : \{taskset_i \subseteq K\} \to \mathbb{R}, \tag{6.1}$$

where $\texttt{taskset}_i$ is the subset of tasks in K running at i. D_i gives the total cost incurred by i for running its `taskset`. This cost depends on the load, $\texttt{load}(\texttt{taskset}_\texttt{i})$, imposed by the tasks. From a game-theoretic perspective, the cost function can be viewed as the *type* of each participant. It is the private information that fully determines a participant's preference for different load allocations.

Each participant monitors its own load and computes its processing cost. There are

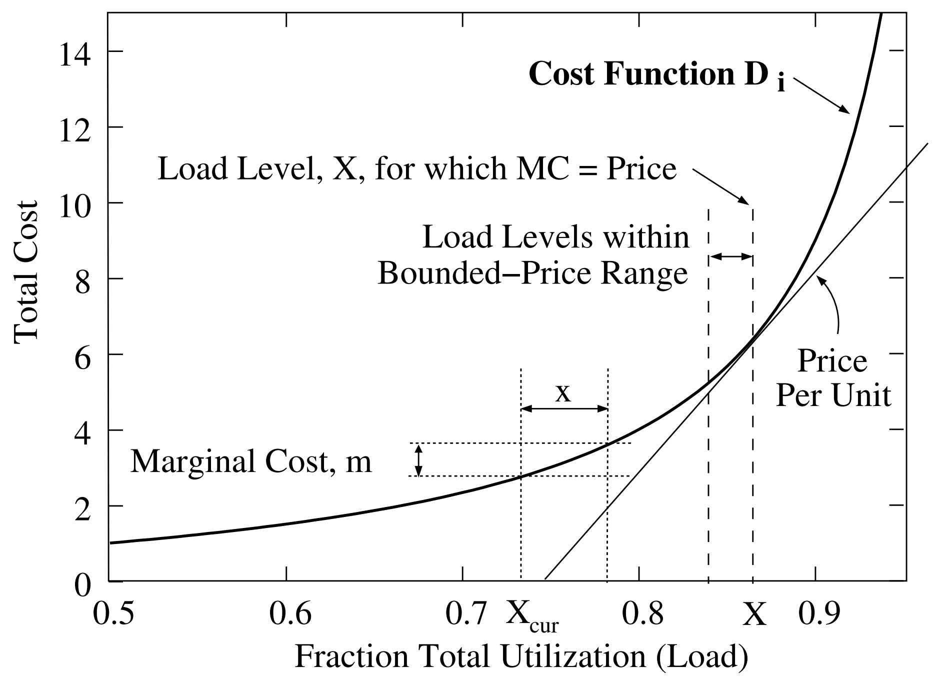


Figure 6-2: Prices and processing costs.

an unlimited number of possible cost functions and each participant may have a different one. We assume, however, that this cost is a *monotonically increasing* and *convex* function. Indeed, for many applications that process messages (*e.g.*, streams of tuples), an important cost metric is the per-message processing delay. For most scheduling disciplines this cost is a monotonically increasing, convex function of the offered load, reflecting the increased difficulty in offering low-delay service at higher load. Figure 6-2 illustrates such a cost function for a single resource. We will use Figure 6-2 repeatedly in this chapter.

We denote the incremental cost or *marginal cost* for node i of running task u given its current $\texttt{taskset}_\texttt{i}$ as:

$$\forall i \in N,\ \ \texttt{MC}_i : \{(u, \texttt{taskset}_i) \mid \texttt{taskset}_i \subseteq K, u \in \{K - \texttt{taskset}_i\}\} \to \mathbb{R} \tag{6.2}$$

Figure 6-2 shows the marginal cost, m, caused by adding load x, when the current load is X_{cur}. Assuming that the set of tasks in $\texttt{taskset}_\texttt{i}$ imposes a total load X_{cur} and u imposes load x, then $\texttt{MC}(u, \texttt{taskset}_\texttt{i}) = m$. If x is one unit of load, we call m the *unit marginal cost.* As we discuss later, nodes compute marginal costs to determine when it is profitable to offer or accept a set of tasks.

We assume that participants are selfish. They aim to maximize their utility, $u_i(\texttt{taskset}_\texttt{i})$, computed as the difference between the payment, $p_i(\texttt{taskset}_\texttt{i})$, that participant i receives for processing some tasks, and the processing cost, $D_i(\texttt{taskset}_\texttt{i})$, it incurs:

$$u_i(\texttt{taskset}_\texttt{i}) = p_i(\texttt{taskset}_\texttt{i}) - D_i(\texttt{taskset}_\texttt{i}). \tag{6.3}$$

When a task originates at a participant, we assume that the client who issued the task pays the participant. As we discuss later, when participants move load from one to the other, they pay each other for the processing.

We also assume that each participant has a pre-defined maximum load level, T_i, that corresponds to a maximum processing cost, $D_i(T_i)$, above which participant i considers itself overloaded. We often speak of T_i as the participant's capacity, although participants can select any load level below their capacity as their maximum desired load.

6.1.3 Social Choice Correspondence

In contrast to previous proposals, we argue that optimal load balance is not needed in a federated system. When a participant is lightly loaded, it can provide good service to its clients and load movements will not improve overall system performance much. They will mostly add overhead. If a participant is heavily loaded, however, performance may significantly degrade. Participants usually acquire sufficient resources to handle their own load most of the time. They experience flash crowds, where the total load significantly exceeds the usual load, only from time to time. A participant can either over-provision to handle such rare peak loads, or it can collaborate with other participants during overload. Our goal is to enable participants to re-distribute such excess load. The goal of BPM is thus to ensure that no participant is overloaded, when spare capacity exists. If the whole system is overloaded, the goal is to use as much of the available capacity as possible. We call an allocation *acceptable* if it satisfies these properties. In summary, our goal is for BPM to implement a social choice correspondence[1] whose outcomes are always *acceptable allocations.*

Definition 9 *An* acceptable allocation *is a task distribution where (1)* no *participant is above its capacity threshold,* or *(2)* all *participants are at or above their capacity thresholds if the total offered load exceeds the sum of the capacity thresholds.*

More formally, the acceptable allocation satisfies:

$$D_i(\text{taskset}_i)\begin{cases} \leq D_i(T_i) \ : \ \forall i \in N, & \text{if the federated system is underloaded, or} \\ \geq D_i(T_i) \ : \ \forall i \in N, & \text{if the federated system is overloaded.} \end{cases} \quad (6.4)$$

As a concrete example, imagine the query diagram deployment shown in Figure 6-1(a). Most of the time, each of the four participants handles its own load. If the input rate increases significantly on input streams, S_3 and S_4, the total load may exceed the capacity of participants P_3 and P_4. In that case, we would like other participants to handle some of the excess load for the duration of the overload, as shown in Figure 6-1(b).

6.2 BPM Overview

We now present and overview of BPM. In the following sections, we discuss the details of each component of the approach.

The goal of mechanism design [127] is to implement an optimal system-wide solution to a decentralized optimization problem, where each agent holds an input parameter to the problem and prefers certain solutions over others. In our case, agents are participants and their cost functions, capacities, and tasks are the optimization parameters. The system-wide goal is to achieve an acceptable allocation, while each participant tries to optimize

[1]Given agent types (*i.e.*, their cost functions), the social choice correspondence selects a set of alternative load allocations and participant payments.

its utility within its pre-defined capacity. Because the set of tasks changes with time, the allocation problem is an online optimization. Since the system is a federation of loosely coupled participants, no single entity can play the role of a central optimizer and the implementation of the mechanism must be distributed. We first present the fixed-price mechanism and also assume that load is fixed. We extend the approach to dynamic load in Section 6.4 and to bounded prices in Section 6.6.

In a mechanism implementation, we must define: (1) a set, S, of *strategies* available to participants (*i.e.*, the sequence of actions they can follow), and (2) a method to select the outcome given a set of strategies chosen by participants. The outcome is the final allocation of tasks to participants. The method is an outcome rule, $g : S^N \rightarrow O$, that maps each possible combination of strategies adopted by the N participants to an outcome O.

We propose an *indirect* mechanism: participants reveal their costs and tasks indirectly by offering and accepting tasks rather than announcing their costs directly to a central optimizer or to other participants. Additionally, agents *pay each other for the load they process.* Our mechanism is based on contracts, which we define as follows:

Definition 10 *A* fixed-price contract $\mathcal{C}_{i,j}$ *between participants i and j defines a price,* `FixedPrice`$(\mathcal{C}_{i,j})$*, that participant i must pay j for each unit of resource i purchases at runtime (*i.e.*, for each unit of load moved from i to j).*

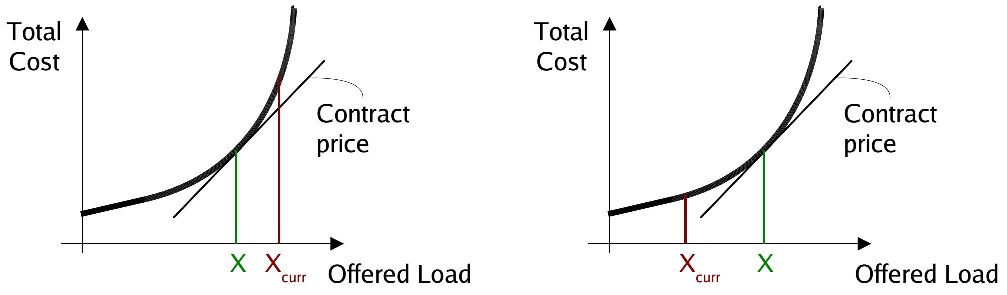
Participants establish contracts offline. At runtime, participants that have a contract with each other may perform *load transfers.* Based on their load levels, they agree on a set of tasks, the `moveset`, that will be transferred from one partner to the other. The participant offering load also pays its partner a sum of `FixedPrice`$(\mathcal{C}_{i,j}) *$ `load`(`moveset`). The payment is proportional to the amount of resources that the task requires. The idea is that i purchases resources from j, but it indicates the specific tasks that need these resources. Hence, partners determine a price offline, but they negotiate at runtime the amount of resources that one partner purchases from the other.

What we propose is thus that participants play two games on different timescales: an offline game of contract negotiation and a runtime game of load movements.

In the first, offline game, participants establish contracts. Assuming for simplicity that all participants are identical, the strategy of a participant is the number of contracts it chooses to establish and the price it negotiates for each contract. The outcome of this game is a contract graph in which the nodes are the participants and an edge between two nodes signifies the existence of a pairwise contract.

We establish the following rules for contract negotiations. As part of the negotiation, we require that participants mutually agree on what one unit of processing, bandwidth, or other resource represent. Different pairs of participants may have contracts specifying different unit prices. Each contract applies only to one direction. There is at most one contract for each pair of participants in each direction. Participants may periodically renegotiate, establish, or terminate contracts offline. We assume the contract graph is connected. The set of contracts at a participant is called its `contractset`. We use C to denote the maximum number of contracts that any participant has. Contracts may contain additional clauses such as the set of tasks that can be moved or a required minimum performance, security, availability, etc. We ignore these additional clauses in our discussion. We further discuss establishing contracts in Sections 6.5 and 6.6.

In the second, runtime game, participants move load to partners. With fixed-price contracts, the set of actions available to participants comprises only the following three



(a) Offering load can improve utility (b) Accepting load can improve utility

Figure 6-3: Load movement decisions based on marginal costs.

actions: (1) offer load at the pre-negotiated price, (2) accept load at the pre-negotiated price, or (3) do neither. The strategy of each participant determines when it performs each one of the above actions.[2] The desired system-wide outcome is an acceptable allocation. The sequence of runtime load movements defines the final task allocation, or outcome of the game. We establish the following rules for the online game. Participants may only move load to partners with whom they have a contract and must pay each other the contracted price. Every time a participant offers a set of tasks at a given price, the offer is *binding*. If the partner accepts, the participant that initiated the movement must pay the offered price. As we show in Section 6.3, this choice can help participants avoid overbooking their resources. We further discuss runtime load movements in Sections 6.3 and 6.4.

Participants may be unwilling to move certain tasks to some partners due to the sensitive nature of the data processed or because the tasks themselves are valuable intellectual property. For this purpose, contracts can also specify the set of tasks (or types of tasks) that may be moved, constraining the runtime task selection. In offline agreements, participants may also prevent their partners from moving their operators further thus constraining the partner's task selections. BPM can handle these constraints as it is based on pairwise contracts; for simplicity, we ignore them in the rest of the discussion.

To ensure that a partner taking over a task provides enough resources for it, contracts may also specify a minimum per-message processing delay (or other performance metric). A partner must meet these constraints or pay a monetary penalty. Such constraints are commonplace in SLAs used, for example, for Grid computing [139], Web and application hosting services [26, 63], and Web services [138]. Infrastructures exist to enforce them through automatic verification [26, 94, 138, 139]. We assume such infrastructure exists and participants provide sufficient resources for the tasks that they run. To avoid breaching contracts when load increases, participants may prioritize on-going tasks over newly arriving tasks.

In summary, we propose a set of two games. Offline, participants negotiate contracts that specify the price they are going to pay each other for processing load. At runtime, given a set of contracts, each participant offers and accepts load paying or receiving the contracted

[2]Our approach works independently of the strategy that agents use to select the set of tasks they offer or accept. To simplify our analysis, we exclude the task selection problem from the strategy space (*i.e.*, from the set of all possible combinations of strategies selected by participants). We also exclude the problem of ordering contracts (*i.e.*, selecting the partner to whom offer load first). This order is typically defined offline by the relationships between participants.

```
00. PROCEDURE OFFER_LOAD:
01. repeat forever:
02.   sort(contractset on price(contractset_j) ascending)
03.   foreach contract $C_j \in$ contractset:
04.     offerset $\leftarrow \emptyset$
05.     foreach task $u \in$ taskset
06.       total_load $\leftarrow$ taskset $-$ offerset $- \{u\}$
07.       if MC($u$, total_load) $>$ load($u$) $*$ price($\mathcal{C}_j$)
08.         offerset $\leftarrow$ offerset $\cup \{u\}$
09.     if offerset $\neq \emptyset$
10.       offer $\leftarrow$ (price($\mathcal{C}_j$), offerset)
11.       (resp, acceptset) $\leftarrow$ send_offer($j$, offer)
12.       if resp = accept and acceptset $\neq \emptyset$
13.         transfer($j$, price($\mathcal{C}_j$), acceptset)
14.         break foreach contract
15.   wait $\Omega_1$ time units
```

Figure 6-4: Algorithm for shedding excess load.

price. In the following sections, we present the different components of the bounded-price mechanism and analyze its properties. We show that the bounded-price mechanism implements the acceptable allocation in a *Bayesian-Nash* equilibrium —*i.e.*, when participants adopt strategies that optimize their *expected utilities* given prior assumptions about the load and contracts of other participants.

6.3 Runtime Load Movements

Given a set of contracts, a participant can use different strategies to improve its utility at runtime. We now present the strategy or algorithm that we would like each participant to follow. Parkes and Shneidman [129] call such algorithm the *intended implementation.* The proposed algorithm is quite natural —we want a participant to offer load or accept load when doing so improves its utility. In Section 6.8, we show that, under certain conditions, the proposed algorithm is indeed the optimal strategy for participants, in a Bayesian-Nash equilibrium.

The strategy that we propose is based on the following simple observation. With fixed-price contracts, if the marginal cost per unit of load of a task is higher than the price in a contract, then processing that task locally is more expensive than paying the partner for the processing. As a result, offering the task to the partner can potentially improve utility. Conversely, when a task's marginal cost per unit of load is below the price specified in a contract, then accepting that task results in a greater payment than cost increase and can improve utility. Figure 6-3 illustrates both cases. In the example, $X_{\texttt{curr}}$ is the current load level at the node. X is the load level for which the unit marginal cost equals the contract price. When $X_{\texttt{curr}} > X$, the unit marginal cost is above the contract price and offering load can improve utility (Figure 6-3(a)). When $X_{\texttt{curr}} < X$, the unit marginal cost is below the contract price and accepting load can improve utility (Figure 6-3(b)). Moving load using marginal costs or marginal utilities is a natural choice and many previous schemes have used this approach (*e.g.*, [96, 141]). When the cost function is convex, moving tasks from a participant with a higher marginal cost to one with a lower marginal cost leads to a gradient descent: each load movement strictly decreases the total cost of the allocation.

Given a set of contracts, we propose that each participant concurrently run one algorithm

```
PROCEDURE ACCEPT_LOAD:
repeat forever:
  offers $\leftarrow \emptyset$
  for $\Omega_2$ time units or while (movement = true)
    foreach new offer received, new_offer:
      offers $\leftarrow$ offers $\cup$ {new_offer}
  sort(offers on price($\mathtt{offers_i}$) descending)
  potentialset $\leftarrow \emptyset$
  foreach offer $o_i \in$ offers
    acceptset $\leftarrow \emptyset$
    foreach task $u \in$ offerset($o_i$)
      total_load $\leftarrow$ taskset $\cup$ potentialset $\cup$ acceptset
      if MC($u$, total_load) $<$ load($u$) $*$ price($o_i$)
        acceptset $\leftarrow$ acceptset $\cup \{u\}$
    if acceptset $\neq \emptyset$
      potentialset $\leftarrow$ potentialset $\cup$ acceptset
      resp $\leftarrow$ (accept, acceptset)
      movement $\leftarrow$ true
    else resp $\leftarrow$ (reject, $\emptyset$)
    respond($o_i$, resp)
```

Figure 6-5: Algorithm for taking additional load.

for shedding excess load (Figure 6-4) and one for taking on new load (Figure 6-5).

The basic idea in shedding excess load is for an overloaded participant to select a maximal set of tasks from its $\mathtt{taskset_i}$ that cost more to process locally than they would cost if processed by one of its partners and offer them to that partner. Participants can use various algorithms and policies for selecting these tasks. We present a general algorithm in Figure 6-4. Because our mechanism makes offers binding, if the partner accepts even a subset of the offered tasks, the accepted tasks are transferred, and the participant shedding load must pay its partner. An overloaded participant could consider its contracts in any order. We assume that order is established offline. One approach is to exercise the lower-priced contracts first with the hope of paying less and moving more tasks. Procedure OFFER_LOAD waits between load transfers to let local load level estimations (*e.g.*, exponentially weighted moving averages) catch-up with the new average load level. If no transfer is possible, a participant retries to shed load periodically. Alternatively, the participant may ask its partners to notify it when their loads decrease sufficiently to accept new tasks.

The basic idea in accepting load is for a participant to accept all tasks that are less costly to process locally than the offered payment from the partner. These offers improve utility as they increase the total payment more than the total processing cost. Because multiple offers can arrive approximately at the same time, participants can accumulate offers for a short time period before examining them (although BPM does not require this). More specifically, in procedure ACCEPT_LOAD (Figure 6-5), each participant continuously accumulates load offers and periodically accepts subsets of offered tasks, examining the higher unit-price offers first. Another approach would be to accept the offers with the highest expected utility increase per unit of load. Since accepting an offer results in a load movement (because offers are sent to one partner at the time), the participant keeps track of all accepted tasks in the potentialset and responds to both accepted and rejected offers. Participants that accept a load offer cannot cancel transfers and move tasks back. They can, however, use their own contracts to move load further or to move it back.

Figure 6-6 illustrates three load movement scenarios. In a single load movement, A can

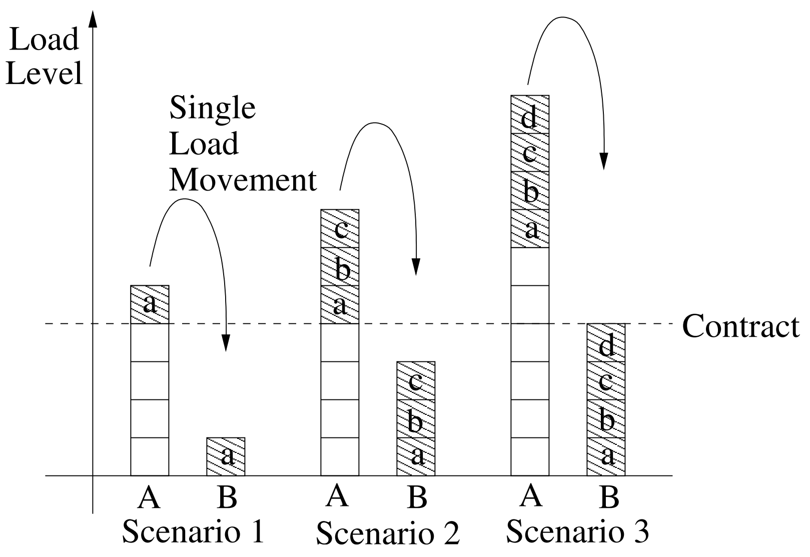


Figure 6-6: Three load movement scenarios for two partners.

transfer to B all tasks with a unit marginal cost greater than the contract price (scenarios 1 and 2). Only those tasks are transferred for which the marginal cost per unit of load at B does not exceed the price in the contract (scenario 3).

Other protocols can also be designed based on fixed-price contracts. Participants could, for instance, offer their load simultaneously to all their partners. For a single overloaded node, this approach would converge faster in the worst-case. This scheme, however, creates a large communication overhead since C (the number of contracts) offers are made for every load movement. Offering load simultaneously to many partners also prevents these partners from knowing which tasks they will actually receive from among all the tasks they accept. This in turn makes it difficult for them to determine how many profitable movements to accept without the risk of overbooking their resources. In contrast, under static load, BPM, allows participants to accept many offers at once without the risk of overbooking. BPM also avoids the extra communication overhead because participants send offers to one partner at the time.

In this section, we showed one strategy for runtime load movements assuming a static system load. The strategy is based on two simple protocols for using contracts at runtime in a way that improves utility. In the next section, we enhance the algorithm to account for dynamic load variations.

6.4 Dynamic Load Conditions

Because load varies with time, and because such variations affect participant load movement decisions, we extend fixed-price contracts to include a *unit load movement duration* in addition to a price. We show how participants can fine-tune their decisions to offer or accept load, given such contracted durations.

With the approach that we introduced, given some load distribution, participants move load to their partners improving their individual utilities and producing successively less costly allocations. A sudden change in total system load will abruptly modify the utility of one or more participants. From that point on, however, participants can again improve their utilities by moving load. A sudden change in load thus translates into restarting the convergence toward acceptable allocation from a different initial load distribution.

Although it may appear that the approach presented so far does naturally handle load variations, there are two problems. In the current mechanism, once load moves, it moves forever. With this constraint, successive load variations may cause a situation where all participants eventually run other participants' tasks rather than their own tasks, which may not be desirable. Furthermore, participants may have some expectations about the duration of their overload and may be unwilling to move load if they expect the overload to be of short duration. We could lessen these problems by modeling load increases and decreases as the arrival and departure of tasks. Such a model is difficult to implement in practice because it requires extra machinery to isolate and move only excess load, and may not always be possible. Additionally, operators accumulate transient state that may have to be moved back after the load subsides.

To address these issues, we propose instead to extend our definition of a contract to include a unit duration, d, for every load movement:

Definition 11 *(***revisited***) A* fixed-price contract $\mathcal{C}_{i,j}$ *between participants i and j defines* a unit duration, d, for each load movement *and a price,* $\texttt{FixedPrice}(\mathcal{C}_{i,j})$*, that participant i must pay j for each unit of resource i purchases at runtime (*i.e.*, for each unit of load moved from i to j).*

Load always moves for the pre-defined time d. After this period, either partner can request that the load move back. Of course, partners can also agree that it is still beneficial for both of them to leave the load where it is for another time-period, d. To limit the frequency of load offers, we also impose the constraint that if a participant refuses load, its partner can retry offering load only after a time-period d. We discuss this latter constraint further in Section 6.8.

Under static load, it is beneficial for a participant to move a task, u, to a partner j, when the local per-unit processing cost exceeds the contract price:

$$\texttt{MC}(u, \texttt{total\_load}) > \texttt{load}(u) * \texttt{price}(\mathcal{C}_j). \tag{6.5}$$

When the offered load (*i.e.*, the set of tasks in the system and the load imposed by any task) changes dynamically, both load and MC become function of time. For example, $\texttt{load}(u, t)$ is the load imposed by task u at time t, and $\texttt{MC}(u, \texttt{total\_load}, t)$ is the marginal cost of task u at time t.

Under dynamic load, to decide whether to offer or accept load, a participant should compare its *expected* marginal cost for a task over the next time-period, d, against the *expected* payment for that task. A participant should perform a load movement only when the movement improves the *expected utility.* For instance, the participant should offer load when:

$$\int_{t=\texttt{now}}^{t=\texttt{now}+d} E[\texttt{MC}(u, \texttt{total\_load}, t)]\, dt > \int_{t=\texttt{now}}^{t=\texttt{now}+d} E[\texttt{load}(u, t)] * \texttt{price}(\mathcal{C}_j)\, dt. \tag{6.6}$$

Similarly, a participant should accept load when, for the duration of a movement, the expected marginal cost for that load is below the expected payment for the same load:

$$\int_{t=\texttt{now}}^{t=\texttt{now}+d} E[\texttt{MC}(u, \texttt{total\_load}, t)]\, dt < \int_{t=\texttt{now}}^{t=\texttt{now}+d} E[\texttt{load}(u, t)] * \texttt{price}(\mathcal{C}_j)\, dt. \tag{6.7}$$

In all cases, even though load varies, the price, $\texttt{price}(\mathcal{C}_j)$, remains fixed.

Participants can use different techniques to estimate expected load levels and marginal costs. If the unit time-period, d, is short compared to the frequency of load variations, participants can assume, for example, that the current load level will remain fixed over the whole time period, d.

Moving load for a bounded time interval also makes it easier to take into account the cost of load migration (which we ignored until now). If the expected overload for the next time period, d, is greater than the price a participant would have to pay a partner *plus twice the migration cost*, then it is beneficial to perform the load movement.

The choice of the proper value of d is an interesting problem. d must be long enough for the gains from a load movement to cover the overhead of the actual migration. Otherwise, it may never be worth to move any load. d should not be too long, though, because the probability that load will not persist for the whole interval increases with d. In this dissertation, we simply assume that d is some fixed value.

6.5 Establishing Fixed-Price Contracts Offline

In the previous sections, we presented strategies for runtime load movements assuming that each participant had a set of fixed-price contracts. We now analyze the set of contracts that a participant should try to establish in order to avoid overload and improve utility.

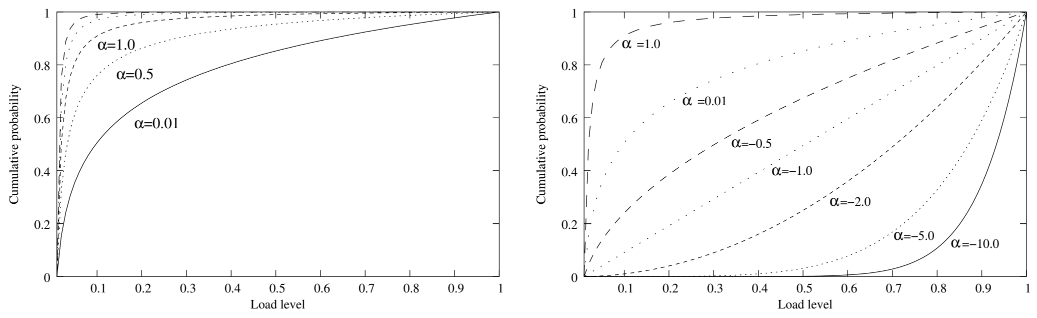
To simplify the analysis, we assume a system of identical participants with independent load levels that all follow the same distributions. We also assume that all participants wish not to exceed the same load threshold T. We analyze heterogeneous settings through simulations, in Chapter 7. We show that a small number of contracts suffice for participants to get most of the benefits from the system.

When a participant negotiates a contract to shed load, it must first determine its maximum desired load level T, and the corresponding marginal cost per unit load. This marginal cost is also the maximum unit price that the participant should accept for a contract. For any higher price, the participant risks being overloaded and yet unable to shed load. A higher price is also equivalent to a processing cost greater than T, a cost that the participant does not want to incur. Any price below that maximum is acceptable but, in general, *higher contract prices increase the probability that a partner will find it cost-effective to accept load.* Therefore, the *optimal strategy for a participant to minimize its chances of being overloaded*, is to negotiate contracts at a price just below T. Figure 6-2 illustrates, for a single resource and a strictly convex function, how a load level X maps to a unit price. In general, this price is the gradient of the cost function evaluated at X.

A participant could try to establish as many contracts as possible, but maintaining a contract comes with recurring costs due to periodic offline re-negotiations. It is only cost effective to establish a contract when the benefits of the contract offset its recurring maintenance costs. We compute the optimal number of contracts by computing the cumulative benefit of contracts and comparing it with the maintenance costs.

Let buyer B be a participant who is shedding load by buying resources from others. To B, all other participants in the system are potential resource sellers. We first study the optimal number of contracts that B should establish.

From B's perspective, the load of each seller follows some probability density function, $p(x)$, within $[k, T]$. The buyer does not distinguish between a seller with load T and a seller with load greater than T because both have no spare resources to offer. We denote



(a) Bounded Pareto load distributions. (b) Various skewed load distributions

Figure 6-7: Illustration of various bounded load distributions with $k = 0.01$, $T = 1.0$, and different values of α.

the cumulative distribution function for a given seller i with $F(x) = P(X_i < x)$. The buyer has a second probability density function, $g(x)$, for its expected load spikes, when its local load exceeds T. We assume that load spikes can exceed T by orders of magnitude. We model the load spikes and the normal load distributions with two separate functions because sellers also exchange load among themselves and may therefore have a more uniform load distribution within $[k, T]$, than an isolated node.

We analyze different types of density functions. With a uniform distribution over an interval $[k, T]$, any value within the interval is equally likely to occur: $p_{\texttt{uniform}}(x) = \frac{1}{T-k}$ for $k \leq x \leq T$ and 0 otherwise. The corresponding cumulative distribution is:

$$F_{\texttt{uniform}}(x) = P_{\texttt{uniform}}(k \leq X \leq x) = \frac{x-k}{T-k} \quad \texttt{for} \quad k \leq x \leq T. \tag{6.8}$$

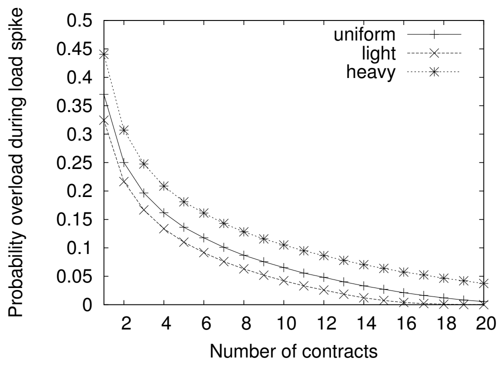
Frequently, however, a participant is likely to be either lightly of heavily loaded most of the time, while staying within a bounded range $[k, T]$. In some cases, especially for load spikes, the distribution can even be heavy-tail. We could use various distributions to model such load conditions. The bounded Pareto density function is commonly used: $p(x) = \frac{\alpha}{x^{\alpha+1}} \frac{k^\alpha}{1-\left(\frac{k}{T}\right)^\alpha}$ for $k \leq x \leq T$, where $0 < \alpha < 2$ is the "shape" parameter. The corresponding cumulative distribution is:

$$\begin{aligned} F_{\texttt{pareto}}(x) &= P_{\texttt{pareto}}(k \leq X \leq x) \\ &= \int_{X=k}^{X=x} \alpha X^{-\alpha-1} \frac{k^\alpha}{1-\left(\frac{k}{T}\right)^\alpha}\, dX. \\ &= \left(1 - \left(\frac{k}{x}\right)^\alpha\right)\left(\frac{1}{1-\left(\frac{k}{T}\right)^\alpha}\right). \end{aligned} \tag{6.9}$$

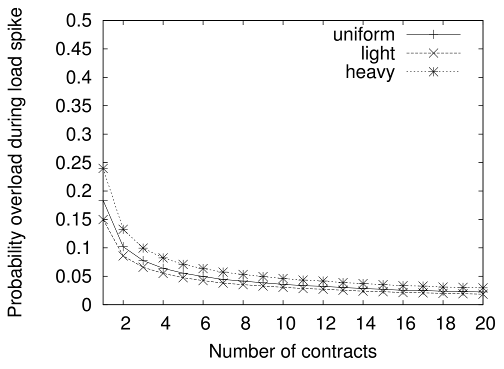
Figure 6-7(a) illustrates the load distribution for $k = 0.01$, $T = 1.0$, and different values of α. The mean of the bounded Pareto distribution is:

$$E(x) = \Big(\frac{k^\alpha}{1-\left(\frac{k}{T}\right)^\alpha}\Big)\Big(\frac{\alpha}{\alpha-1}\Big)\Big(\frac{1}{k^{\alpha-1}} - \frac{1}{T^{\alpha-1}}\Big). \tag{6.10}$$

We use the Pareto distribution to model load spikes. In many cases, we assume that spikes can exceed the threshold T by a few orders of magnitude.



(a) Spikes follow a bounded Pareto with $\alpha = 0.14$ and range $[0.01, 10T]$.



(b) Spikes follow a bounded Pareto with $\alpha = 0.5$ and range $[0.01, 100T]$.

Figure 6-8: Probability of overload when a load spike occurs. In all six configurations, the first few contracts provide most benefits.

To analyze various non-heavy-tail distributions, we use the same cumulative distribution function as above, but we vary α within a wider range. Figure 6-7(b) illustrates the various resulting distributions. For $\alpha = -1.0$, the distribution is uniform. Decreasing α, increases the fraction of nodes that have a heavier load than in the uniform distribution. Increasing α, increases the fraction of nodes that have a lighter load than in the uniform distribution.

In the absence of contracts, when a load spike occurs, the participant is overloaded with probability 1.0.

$$P_{overload}(0,T) = G(X > 0) = 1. \tag{6.11}$$

With one contract at threshold T, a participant is overloaded only when it experiences a load spike that its partner cannot absorb. Assuming highly fine-grained tasks, the probability is:

$$P_{\texttt{overload}}(1,T) \approx G(X > T) + \int_{x=0}^{T} G(X = x) P(X_1 > T - x)\, dx. \tag{6.12}$$

where $G(X > T)$ is the probability of a load spike that exceeds the partner's total capacity, T. The integral computes the probability that a spike occurs within the partner's capacity, T, but the partner's load is too high to absorb it (given by $P(X_1 > T - x)$). We denote the partner's load with X_1.

We can generalize this relation to N contracts. A participant is overloaded when it experiences a load spike greater than the total capacity of its C partners, $G(X > CT)$, or when it experiences a smaller load spike, but the aggregate available capacity at its partners is less that the value of the spike:

$$P_{\texttt{overload}}(C,T) \approx G(X > CT) + \int_{x=0}^{CT} G(X = x) P(X_1 + X_2 + ...X_C > CT - x)\, dx. \tag{6.13}$$

Figure 6-8 shows the probability of overload in six concrete configurations. Each subfigure shows results for a different distribution used for load spikes, although both distributions have the same mean, T. Each curve in each graph shows the results for a different distribution for the load at the sellers. These distributions have different means, but in

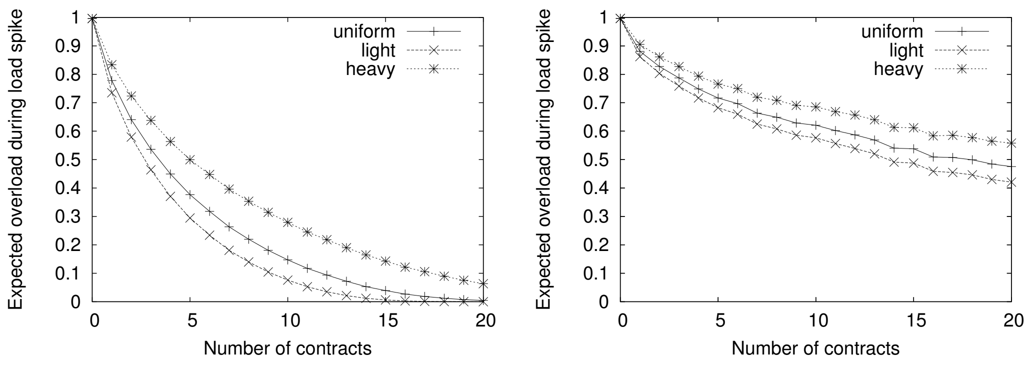


Figure 6-9: Expected magnitude of overload when a load spike occurs. Without contracts the expected magnitude of the overload is 1.0. It decreases with each additional contract, but the decrease is sharpest for the first few contracts.

all cases the load varies within range $[0.01, 1.0]$. The "light" distribution has mean 0.37 ($\alpha = -0.5$), "the uniform" has mean 0.505 ($\alpha = -1.0$), and the "heavy" has mean 0.67 ($\alpha = -2.0$). We run Monte-Carlo simulations to compute $P_{\texttt{overload}}(C, T)$ for increasing values of C. Each point is the result from running 500,000 simulations. In all configurations, increasing the number of contracts reduces the probability of overload during a spike. Because load spikes can be orders of magnitude higher than capacity, the probability remains above zero even with many contracts. Interestingly, the first few contracts provide most of the benefit (3 to 5 in the examples). When load spikes are more uniformly distributed (Figure 6-8(a)), larger numbers of contracts are helpful, although each additional contract brings an increasingly smaller improvement.

The benefit of a set of C contracts, denoted with $\texttt{Benefit}(C)$, is not only the savings realized from reducing the frequency of overload, but also a function of reducing the expected magnitude of overload. Figure 6-9 shows the expected magnitude of overload for the same experiments as in Figure 6-8. Given the expected magnitude of overload, we compute the benefit of a set of C contracts as follows:

$$\texttt{Benefit(C)} = P_{\texttt{spike}} \;\; f\big(E[\texttt{Overload}(0)] - E[\texttt{Overload}(C)]\big), \tag{6.14}$$

where $E[\texttt{Overload}(0)]$ is the expected magnitude of the overload when a load spike occurs and the participant has zero contracts. $E[\texttt{Overload}(C)]$ is the expected magnitude of the overload with C contracts. The difference represents the savings. f is a function of the savings realized by moving excess load to a partner rather than incurring processing costs locally. $P_{\texttt{spike}}$ is the probability that a load spike occurs at all. It is cost-effective to establish a contract only when the added benefit of the contract offsets the cost of negotiating and maintaining that contract. Figure 6-10 shows $\texttt{Benefits}(C)$ for an increasing number of contracts for the set of experiments from Figure 6-8. In the figure, $P_{\texttt{spike}} = 0.05$, f is the identity function, and the total cost of a set of contracts grows linearly with the number of contracts in the set. Once again, each additional contract is beneficial. When load spikes can exceed capacity by orders of magnitude, the benefits increase almost linearly with the number of contracts. If contracts are extremely cheap, then it is beneficial for a buyer to establish large numbers of them. With smaller load spikes, the first few contracts provide most benefits and, as the cost of maintaining a contract increases, it quickly becomes cost-

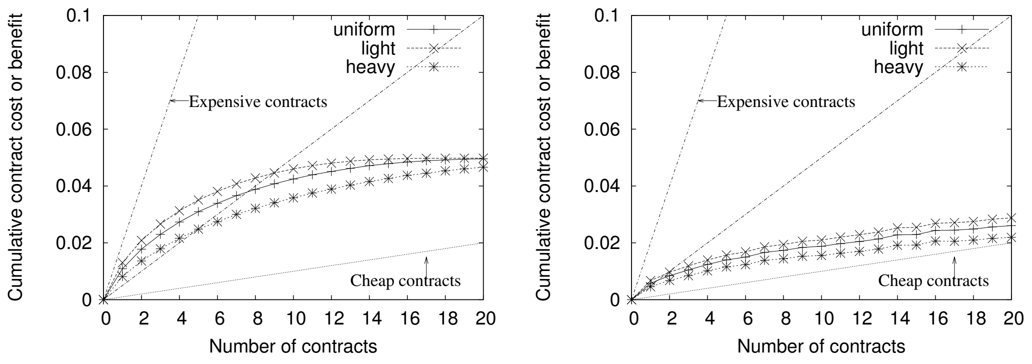


Figure 6-10: Example of cost and benefit of an increasing number of fixed-price contracts for the same experiments as in Figure 6-8. Unless contracts are very inexpensive, it is only cost-effective to establish a few of them.

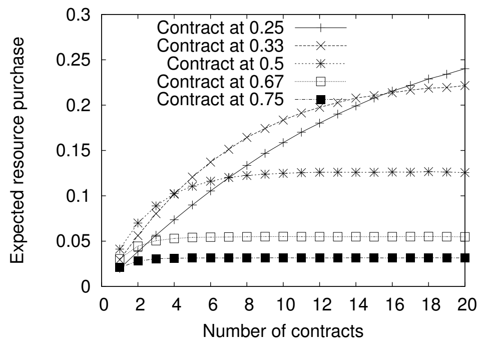
effective to maintain only a few contracts (fewer than 10 in the experiment). Of course, if contracts are very expensive, it may not be beneficial to establish any contract at all.

The computation is different for *contracts in the reverse direction.* When a participant acts as a seller, every contract that it establishes causes it to receive some extra load from time to time. Because any given buyer experiences overload with small probability, $P_{\texttt{spike}}$, and because the buyer does not use all its partners for every spike, a seller receives extra load with probability, $P_{\texttt{extra\_load}} < P_{\texttt{spike}}$. Because the probability of extra load from a single contract is small, the cumulative benefit of contracts grows linearly with their number. If the benefit of one contract is greater than the cost of maintaining that contract, a participant can profit from establishing many contracts as a seller. Of course, when the number of contracts is very large, eventually, the seller is always heavily loaded and the benefit of additional contracts starts to decrease.

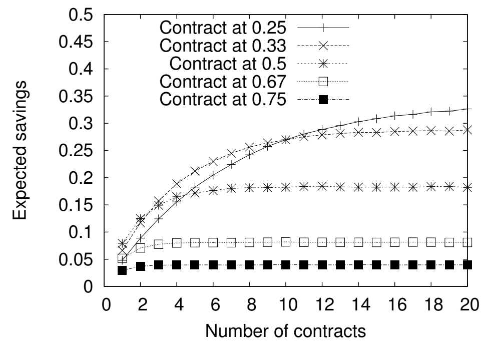
A participant may establish additional contracts to buy resources at a price below T. The participant will use these contracts not to absorb its load spikes but rather to reduce its overall processing costs. Similarly, a participant may establish additional contracts to sell resources at a price below T. In both cases, participants maximize contract benefits by maximizing the profit on each load movement and the load movement frequency. Figure 6-11 shows the expected amount of resources a buyer will purchase and the expected savings it will make from a set of contracts at load levels βT with $T = 1.0$ and $\beta \in \{0.25, 0.33, 0.5, 0.66, 0.75\}$. The figure shows results for three different load distribution functions: uniform ($\alpha = -1.0$, range $[0.01, 1.0]$), light ($\alpha = -0.5$, range $[0.01, 1.0]$), and heavy ($\beta = -2.0$, range $[0.01, 1.0]$). For the savings computation, we use a simple monotonically increasing cost function: $\frac{\rho - 0.2}{1 - \rho + 0.2}$, as an illustrative example. Each point is the average of $500,000$ simulations.

For a very small number of contracts (up to 3), the median price (0.5 for uniform, 0.3 for light, and 0.7 for heavy) yields the greatest amount of resources moved. With more contracts, a price a little below the median results in more resources moved because the buyer is more frequently in a position to purchase resources while at least one of the sellers has spare resources. Prices a little below the median also yield the greatest *utility* increase for the buyer. When prices are too low, the benefit is lower because fewer resources can be moved.

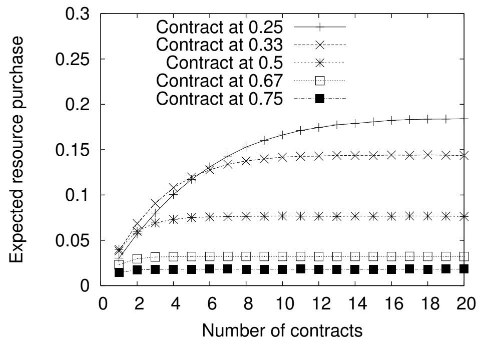
For a seller, the graphs would be reversed. Prices just a little above the median would



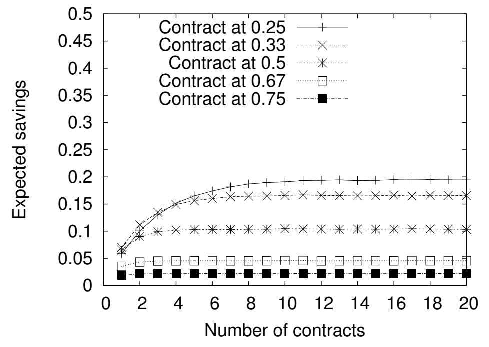
(a)



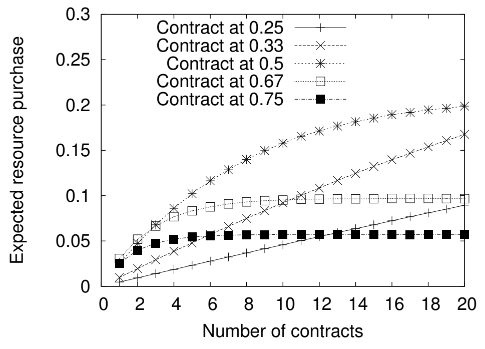
(b)



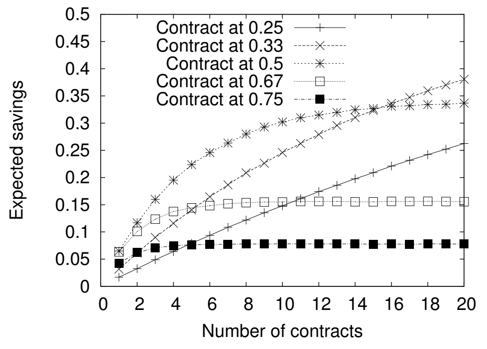
(c)



(d)



(e)



(f)

Figure 6-11: Buyer benefits for contracts below threshold T. (left) Expected resource purchases by a buyer and (right) buyer utility gains for various contracts below T and various load distributions: (a) and (b) uniform, (c) and (d) light load ($\alpha = -0.5$), (e) and (f) heavy load ($\alpha = -2.0$).

Load level corresponding to the contract price	Contract direction	Number of contracts
Maximum load	Buyer	A few
Median load	Buyer	A few
Maximum load	Seller	Many
Median load	Seller	A few

Table 6.1: Heuristics for establishing offline contracts.

yield the largest benefit. Because neither the buyer nor the seller has an incentive to concede faster during a negotiation, and because both will benefit greatly from a contract at the median price, we can expect contracts to be established at that price. With a price around the median, 5 to 6 contracts suffice to get nearly the maximum benefit.

Table 6.1 summarizes the optimal number of fixed-price contracts that a participant should try to establish. For resource buyers, the first few contracts (approximately five) suffice to attain most of the possible benefits. Each additional contract provides only a small incremental improvement in utility. If, however, load spikes are several orders of magnitude greater than capacity while contract prices are extremely cheap, then the buyer should establish as many contracts as possible. For resource sellers, it is beneficial to establish only a few contracts if these contracts have low prices. For contracts at a price corresponding to the maximum capacity, the utility increases almost linearly with the number of contracts, making larger numbers of contracts profitable.

6.6 Bounded-Price Contracts

In the previous sections, we presented the basic components of BPM. We showed how to establish and use fixed-price contracts for moving load, possibly for only a limited amount of time. Fixed-price contracts, however, do not always produce acceptable allocations. In this section, we show that by extending fixed-price contracts to cover a small price range, we can guarantee that a system of uniform nodes and contracts converges to acceptable allocations in all contract and load configurations. Because the final price is not fixed anymore, participants must negotiate the final price within range. We present a simple negotiation protocol and show that in most cases, participants agree on the final price even without negotiating.

Fixed-price contracts do not necessarily lead to acceptable allocation because load cannot always propagate through a sequence of nodes. We have shown in Section 6.5 that it is not cost-effective for a participant to establish enough contracts to ensure that its direct partners can always absorb its excess load. Partners can always use their own contracts to move load further but such movements are not possible in all configurations of the contract graph. A chain of identical contracts is one example of a configuration that prevents load from spreading. As illustrated in Figure 6-12, a lightly loaded node in the middle of a chain accepts new tasks as long as its marginal cost is *strictly below* the contract price. The node eventually reaches maximum capacity (as defined by the contract price) and refuses additional load. It does not offer load to partners that might have spare capacity because its unit marginal cost is still lower than any of its contract prices. Hence, if all contracts are identical, a task can only propagate one hop away from its origin.

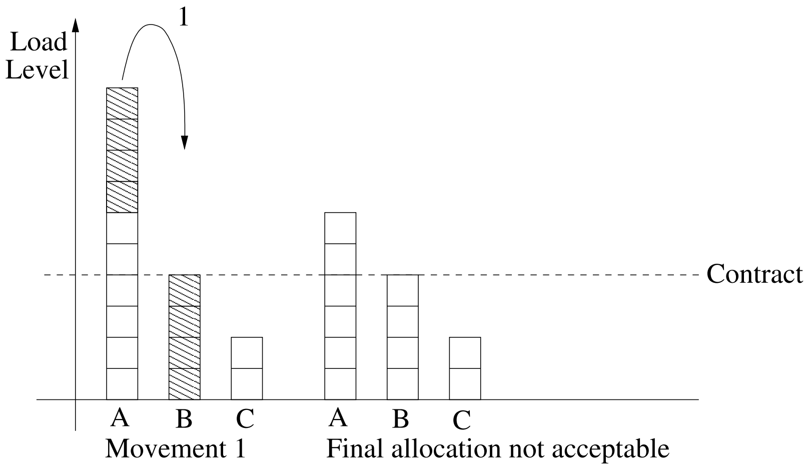


Figure 6-12: Fixed-price contracts do not lead to acceptable allocation in certain configurations. In a chain of three nodes with two identical contracts, the middle node has no incentive to propagate load from an overloaded to a lightly loaded partner.

6.6.1 Minimal Price Range

To achieve acceptable allocations for *all* configurations, participants need to specify a *small range of prices*, [`FixedPrice` $-\Delta$; `FixedPrice`], in their contracts. With a contracted price-range, partners negotiate the final price for each load movement at runtime. By allowing prices to vary, we enable load to propagate through chains of nodes. Indeed, a participant can now forward load from an overloaded partner to a more lightly loaded one by accepting tasks at a higher price and offering them at a lower price (*i.e.*, selling resources at a high price and buying resources at a lower price). We call *bounded-price contracts* those contracts that specify a range of prices rather than a fixed price.

Definition 12 *A* bounded-price contract $\mathcal{C}_{i,j}$ *between participants* i *and* j *defines a unit duration,* d*, for each load movement and a price range:* [`FixedPrice`$(\mathcal{C}_{i,j}) - \Delta(\mathcal{C}_{i,j})$, `FixedPrice`$(\mathcal{C}_{i,j})$]*, that bounds the runtime price paid by participant* i *for each unit of resource it purchases from* j *at runtime (*i.e.*, for each unit of load moved from* i *to* j*).*

Since a fixed unit price equals the gradient (or derivative) of the cost curve at some load level, a price range converts into a load level interval as illustrated in Figure 6-2. The price range is the difference in the gradients of the cost curve at interval boundaries.

With a larger price range, the unit marginal costs of nodes are more likely to fall within the dynamic range. Because load variations within the contracted price range easily create load movement opportunities, a large price range increases price volatility and the number of reallocations caused by small load variations. Our goal is therefore to keep the range as small as possible and extend it only enough to ensure convergence to acceptable allocations.

We now derive the minimal price range that ensures convergence to acceptable allocations. We analyze a network of homogeneous nodes with identical contracts. We explore heterogeneous contracts through simulations in Chapter 7. For clarity of exposition, we also assume in the analysis that all tasks are identical to the smallest migratable task, u and *impose the same load.*

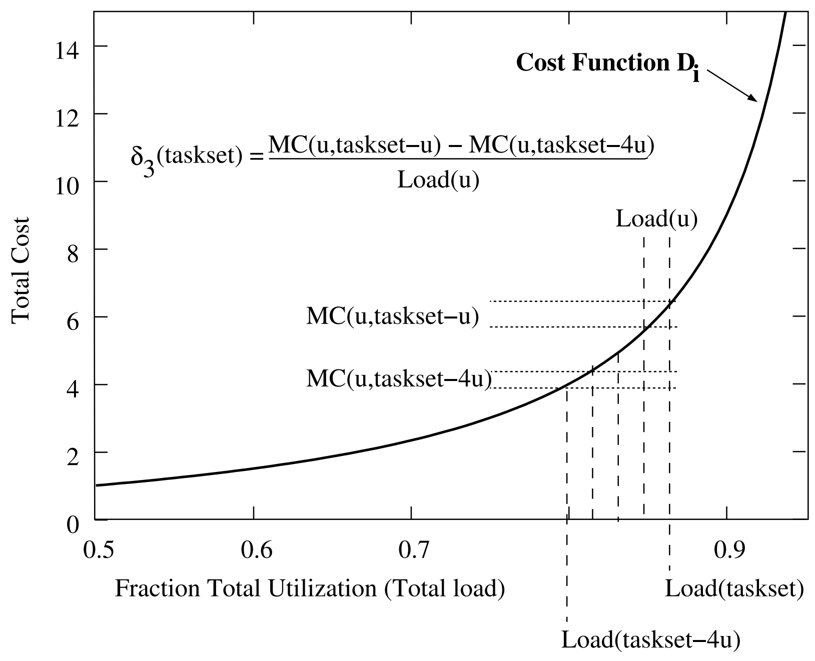


Figure 6-13: Example of $\delta_k(\texttt{taskset})$ computation, for $k = 3$.

We define δ_k as the decrease in unit marginal cost due to removing k tasks from a node's `taskset`:

$$\delta_k(\texttt{taskset}) = \frac{\texttt{MC}(u, \texttt{taskset} - u) - \texttt{MC}(u, \texttt{taskset} - (k+1)u)}{\texttt{load}(u)}. \tag{6.15}$$

δ_k is thus approximately the difference in the cost function gradient evaluated at the load level including and excluding the k tasks. Figure 6-13 illustrates the concept of δ_k.

Given a contract with price, `FixedPrice`, we define $\texttt{taskset}^{\texttt{F}}$ as the maximal set of identical tasks u that a node can handle before its unit marginal cost exceeds `FixedPrice` and triggers a load movement. *I.e.*, $\texttt{taskset}^{\texttt{F}}$ satisfies: $\texttt{MC}(u, \texttt{taskset}^{\texttt{F}} - u) \leq \texttt{load(u)} * \texttt{FixedPrice}$ and $\texttt{MC}(u, \texttt{taskset}^{\texttt{F}}) > \texttt{load(u)} * \texttt{FixedPrice}$.

If all contracts in the system specify the same price range, $[\texttt{FixedPrice} - \Delta, \texttt{FixedPrice}]$ such that $\Delta = \delta_1(\texttt{taskset}^{\texttt{F}})$, any task can now travel two hops away from the node where it originated. As illustrated in Figure 6-14, a lightly loaded node accepts tasks at $\texttt{FixedPrice} - \delta_1(\texttt{taskset}^{\texttt{F}})$ until its load reaches that of $\texttt{taskset}^{\texttt{F}} - u$. The node then alternates between accepting one task, u, at `FixedPrice` and offering one task at price $\texttt{FixedPrice} - \delta_1(\texttt{taskset}^{\texttt{F}})$. Similarly, for load to travel through a chain of $M + 1$ nodes (or M transfers) the price range must be at least $\delta_{M-1}(\texttt{taskset}^{\texttt{F}})$. The jth node in such a chain alternates between accepting a task at price $\texttt{FixedPrice} - \delta_{j-1}(\texttt{taskset}^{\texttt{F}})$ and offering it at price $\texttt{FixedPrice} - \delta_j(\texttt{taskset}^{\texttt{F}})$.

We now show the following lemma:

Lemma 1 *In a network of homogeneous nodes, tasks, and contracts, to ensure convergence to acceptable allocations in an underloaded system, the price range in contracts must be at least* $[\texttt{FixedPrice} - \delta_{M-1}(\texttt{taskset}^{\texttt{F}}), \texttt{FixedPrice}]$*, where M is the diameter of the network of contracts and* $\texttt{taskset}^{\texttt{F}}$ *is the set of tasks that satisfies* $\texttt{MC}(u, \texttt{taskset}^{\texttt{F}} - u) \leq \texttt{load(u)} * \texttt{FixedPrice}$ *and* $\texttt{MC}(u, \texttt{taskset}^{\texttt{F}}) > \texttt{load(u)} * \texttt{FixedPrice}$.

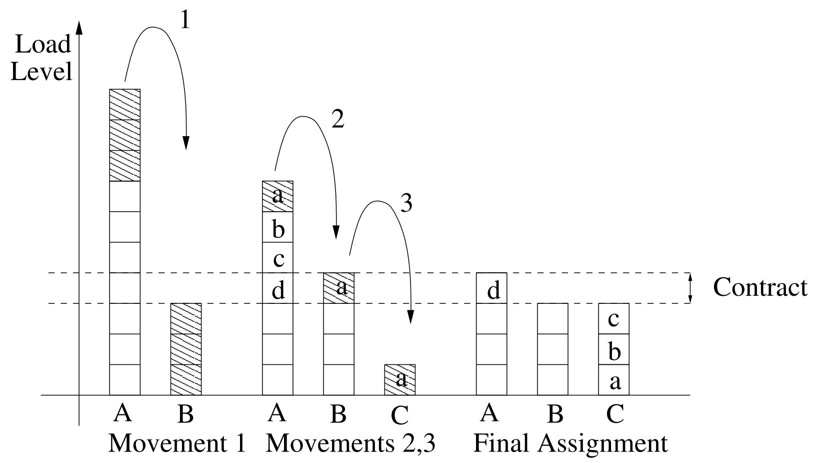


Figure 6-14: Load movements between three nodes using a small price range.

Conditions for the Lemma: *All participants have cost functions that are monotonically increasing and convex. Any node can run any task.*

We consider a system to be underloaded when the total load is less than the sum of all node capacities at the *lowest* price bound. For a price range $[\texttt{FixedPrice} - \delta_{M-1}(\texttt{taskset}^{\texttt{F}}), \texttt{FixedPrice}]$, the capacity of a node at the lowest price bound is $\texttt{taskset}^{\texttt{F}} - (M-1)u$.

Proof. We prove Lemma 1 by contradiction. Suppose that node N_0 has a load level above capacity in the final allocation. We show that N_0 cannot exist in an underloaded system with the properties outlined in the Lemma.

By definition of an underloaded system, as long as N_0 is above capacity (and therefore has an above average load), there exists at least one node N_M in the system that has a load level below $\texttt{taskset}^{\texttt{F}} - (M-1)u$ (below average) and can thus accept load. N_M is at most M, the diameter of the contract network, hops away from N_0. Because the price range is $\delta_{M-1}(\texttt{taskset}^{\texttt{F}})$ load can propagate for M hops. Load can thus propagate from N_0 to the closest underloaded node, N_M. Load movements continue until the load at N_M reaches $\texttt{taskset}^{\texttt{F}} - (M-1)u$. At this point, if the load at N_0 is still above capacity, there must exist another node N_M with a load level below $\texttt{taskset}^{\texttt{F}} - (M-1)u$. Hence, load movements do not stop until the load level at N_0 falls below capacity. Because convergence follows a gradient descent, load movements eventually stop. Therefore, N_0 cannot exist in the final allocation. ▮

We consider a system to be overloaded when the total load is greater than the sum of node capacities at the *lowest* price bound. When the system is overloaded, a price range does not lead to an acceptable allocation (where $\forall i \in N,\ D_i(\texttt{taskset}_\texttt{i}) \geq D_i(\texttt{taskset}^{\texttt{F}}_\texttt{i})$). Indeed, because of the dynamics of load movements through a chain, in the final allocation, some participants may have a marginal cost as low as $\texttt{FixedPrice} - \delta_M(\texttt{taskset}^{\texttt{F}}_\texttt{i})$ (wider ranges do not improve this bound). For overloaded systems, price-range contracts achieve instead *nearly acceptable allocations* defined as:

Definition 13 *A* nearly acceptable allocation *satisfies* $\forall i \in N,\ D_i(\texttt{taskset}_\texttt{i}) > D_i(\texttt{taskset}^{\texttt{F}}_\texttt{i} - Mu)$. I.e., *all participants operate above or only slightly below their maximum capacity.*

In summary, a small price range proportional to the diameter of the network of contracts suffices to ensure that underloaded systems always converge to acceptable allocation, and overloaded systems converge to nearly acceptable allocation. We now examine how participants can negotiate the final price at runtime.

6.6.2 Negotiating the Final Price

With a bounded-price contract, participants must negotiate the final price within the contracted range at runtime. The negotiation is performed automatically by agents that represent participants. In this section, we propose a simple and efficient negotiation protocol.

The negotiation protocol is based on three observations. First, reaching an agreement is *individual rational* [3], *i.e.*, both participants are interested in reaching an agreement because a load movement at a unit price within their respective marginal costs increases both their utilities. Second, the price range is small, limiting the maximum gain that a participant can achieve from negotiating compared to accepting the worst price within the range. Third, in Section 6.5, we showed that participants improve their expected utilities by establishing multiple contracts at the same price. Multiple equivalent contracts create competition between both resource buyers and resource sellers. The competition is greater between resource sellers for contracts corresponding to the pre-defined capacity load, T, because buyers are rarely overloaded. For contracts at lower prices, the competition is the same for both types of participants. Given the above observations, we propose a simple negotiation protocol that ensures an efficient, often one-step, negotiation. The protocol favors resource buyers by leveraging the competition between resource sellers to improve negotiation efficiency (*i.e.*, avoid lengthy negotiations).

Assuming a price range $[p_L = \texttt{FixedPrice} - \Delta, p_H = \texttt{FixedPrice}]$, with $0 \leq p_L \leq p_H < 1$, and $p_H - p_L << 1$, negotiation proceeds as follows:

1. As before, when the buyer wants to purchase some resources, it offers a set of tasks to a seller at a price $p_1 = p_L$, the *lowest price within the contracted range*.
2. If the seller accepts the offered load (or subset of the load), the negotiation completes. The load moves and the buyer pays p_1.
3. If the seller refuses the lowest price, it can reply with a counter-offer, $p_2 \in [p_1, p_H]$, for a *subset of the offered tasks* or it can terminate the negotiation. The seller must respond within a short pre-defined time-period or the buyer concludes that the seller rejected the offer.
4. The seller's counter-offer is also binding. If the movement at p_2 is profitable for the buyer, the buyer can move a subset of the accepted load and pay p_2. Otherwise, the buyer rejects the counter-offer and the negotiation terminates. The buyer must also respond within a bounded amount of time or the seller concludes that the buyer rejected the counter-offer.
5. At least d time-units after the initial offer, the buyer can try to offer load at $p_1 = p_L$ again. At that point, a new negotiation starts.

All offers and counter-offers are binding for a limited time period. We propose to make seller's counter-offers binding for a sufficiently long time to enable the buyer to offer load to all its partners in turn before the first counter-offer expires. If a buyer's offer is binding for a duration, t, and the buyer has C contracts, the seller's counter-offers must be binding for a time Ct. With these constraints, upon receiving a counter-offer, the buyer can offer load

[3]A mechanism is individual rational if participation leads to a higher expected utility than non-participation.

to another partner that it has not tried yet or if all partners are heavily loaded, it can move load to the partner that replied with the lowest counter-offer and pay the counter-offered price. Because there are multiple sellers, because sellers do not see each other's counter-offers, and because these counter-offers are binding, the negotiation is thus equivalent to a *first-price reverse auction* with *sealed-bids*. The auction is a reverse auction because it is conducted by the buyer rather than the seller.

We now examine the strategies available to the buyer and the seller, starting with the buyer. For a given contract price, if the load distributions at the sellers are all the same or they are unknown, the buyer can send a load offer to a randomly selected seller (or it can use a pre-defined order). If the seller refuses the lowest price, the buyer should send the same offer to another seller, as it is possible the other will accept it at the lowest price p_1. If all sellers refuse p_1, however, the optimal strategy for the buyer is to move load to the partner with the lowest counter-offer, paying the lowest possible price for the load. In case of a tie, the buyer can randomly pick the winner or it can pick the preferred partner based on offline relationships. The best strategy for the buyer is thus to act as an auctioneer in a reverse auction.

We now examine the strategies available to a seller. A seller can either accept the lowest price p_1, or reply with a higher price p_2, hoping to make a greater profit. Because each participant establishes C contracts, there is a probability, $1 - p_1^{(C-1)}$, that another seller has a marginal cost below p_1 and can accept p_1 directly. This probability is high when p_1 is close to 1 or C is large. Additionally, because the price range is small, potential extra gains from a counter-offer are small as well. Therefore, as long as selling resources at p_1 increases utility, accepting p_1 directly is often the optimal strategy for the seller (we discuss this scenario further in Section 6.8, Property 2).

If a seller is heavily loaded, however, and its marginal cost is close to or above p_1, the seller maximizes its expected utility by counter-offering a higher price. If there exists at least one other seller who can accept p_1, then the value of the counter-offer p_2 does not matter, but if all other sellers are also loaded, they will all make counter-offers (or they will reject the offer). In that situation, a high p_2 increases the possible profit but reduces the chances of winning the load because of competition and because the buyer itself may not be able to accept the higher price. The seller must thus counter-offer, p_2, that maximizes:

$$E[\texttt{Profit}_{\texttt{Seller}}] = (p_2 - S)P(p_2). \tag{6.16}$$

where S is the seller's valuation for the offered tasks (*i.e.*, the average per-unit marginal cost), $p_2 - S$ is the seller's profit given its valuation, S, and $P(p_2)$ is the probability of winning the load with a counter-offer p_2.

As an example, let's assume a price range $[0, 1]$ and a uniform distribution of competing offers in the range. The seller needs to select its counter-offer p_2 to maximize its expected profit given by:

$$E[\texttt{Profit}_{\texttt{Seller}}] = (p_2 - S)(1 - p_2)^{(C-1)}(1 - p_2). \tag{6.17}$$

where $p_2 - S$ is the profit realized by the seller, $(1 - p_2)^{(C-1)}$ is the probability that none of $C - 1$ competing sellers places a bid below p_2. Because the buyer can have a valuation within the range as well, we use $1 - p_2$ to model a linearly decreasing probability that the buyer can accept the price. Table 6.2 shows examples of counter-offers assuming between 1 and 7 competing sellers. With as few as 5 competitors, it is never profitable to over-price by more than 15% of the total price range, showing that sellers will produce counter-offers

Seller's valuation, S	0	0.25	0.5	0.75
1 competitor	0.33 (33%)	0.50 (25%)	0.67 (17%)	0.83 (8%)
3 competitors	0.20 (20%)	0.40 (15%)	0.60 (10%)	0.8 (5%)
5 competitors	0.14 (14%)	0.36 (11%)	0.57 (7%)	0.79 (4%)
7 competitors	0.11 (11%)	0.33 (8%)	0.55 (5%)	0.78 (3%)

Table 6.2: Example of counter-offers (and the corresponding over-pricing expressed as a percent of total price range) by sellers with different valuations for resources and different numbers of competitors.

only a little over their true valuations. In practice, of course, all rational sellers would follow the over-pricing strategy and would produce counter-offers higher than their valuations.

A well-known result from auction theory [172] is that if a set of C risk-neutral participants[4] bid for a single item, and if their valuations are uniformly distributed in the range $[0, 1]$, then a Nash equilibrium strategy (*i.e.*, the best strategy for each agent assuming all other agents follow that strategy) is to bid $\alpha B = \frac{C-1}{C}B$, where B is the buyer's valuation for the item.

In a reverse auction, the sellers bid rather than the buyers. The situation, however, can also be viewed as a regular auction where the sellers bid for the *discount* they will offer to the buyer. If a seller bids its true valuation $S \in [0, 1]$, the discount to the buyer is $1 - S$. It is the savings compared to paying the maximum price 1. The Nash equilibrium bids for the sellers can then be expressed as: $1 - \frac{C-1}{C}(1 - S)$. In our case, counter offers must be within the contracted range $[p_L, p_H]$. For a valuation, $S \in [p_L, p_H]$, and assuming a fraction, β, of sellers has a unit marginal cost within the contracted range, the equilibrium counter offer should thus be:

$$p_2 = p_H - \frac{\beta C - 1}{\beta C}(p_H - S). \tag{6.18}$$

The above result does not consider the fact that the buyer itself may not be able to accept a higher price, but we could model the buyer as an extra seller. The above result uses the assumption that the buyer breaks ties randomly.

In summary, assuming the buyer has a set of C equivalent contracts, the optimal strategy for a seller when its average unit marginal cost, S, for the offered tasks falls below the price range is to accept the lowest price p_1. When its average unit marginal cost falls within the contracted range, the optimal strategy for the seller is to counter-offer a price p_2 a little above S. Otherwise, the seller must reject the offer. We prove this property and discuss possible values of β in Section 6.8.

The above strategy analysis assumes that the buyer offers a single task to sellers. In BPM, the buyer offers groups of tasks. As long as a seller's unit marginal cost is below p_L, the seller can accept the complete set of tasks or a subset of these tasks at the lowest price. Similarly, if its counter-offer is equal to the highest price within range, a seller should accept all tasks that improve utility. Within the price range, however, a participant can choose between counter-offering a lower price for a single task or a higher price for a group of tasks. To simplify the approach, we assume that within range, counter-offers are made for a single task.

In most cases, even when participants' counter-offers are a little higher than their actual

[4]A risk-neutral participant is neither risk-averse nor risk-seeking. A risk-neutral participant is, for example, willing to bet up to \$50 for a 0.5 chance of winning \$100 and a 0.5 chance of winning \$0.

valuations, a load movement occurs and both the buyer and the seller improve their utilities. A possible load movement fails only when the buyer's valuation, B, is within $[S, p_2]$, where p_2 is the counter-offered price. Because p_2 is only a little higher than S, this situation occurs rarely. Additionally, the failure is not permanent. After a time-period, d, the buyer can retry to move load at the minimum price. Based on the failure of the previous load movement, the seller can update its estimate of the buyer's load and counter-offer a price $p_2' < p_2$. With this approach, if a load movement is possible, it occurs eventually.

The negotiation protocol that we presented is one possible approach. BPM works independently of the protocol used in the final price negotiation, but we showed that it is possible to make the final price negotiation efficient. We further discuss the properties of the price negotiation in Section 6.8. We now discuss how BPM can apply to federated SPEs.

6.7 Application to Federated SPEs

As we outlined at the beginning of this chapter, BPM can be directly applied to a stream processing engine. Tasks in an SPE have an interesting property, though: the output streams of some tasks serve as input streams to others.

These relationships between tasks affect marginal costs. For the same additional task, a participant already running related tasks incurs a lower processing cost increase than a participant running only unrelated tasks. Hence, participants can lower their processing costs by clustering related tasks together, preferring to offer more loosely coupled tasks during overload. Clustering inter-dependent tasks can help improve overall performance.

Relationships between tasks have a second impact on the system. A load movement by a participant may affect the processing cost of upstream participants because the latter may have to send their output streams to a different location. Such a change may cause an upstream participant to perform a load movement, in turn. These dependencies do not cause oscillations, however, because there are no cycles in query diagrams and dependencies are unidirectional —downstream movements affect only upstream participants.

6.8 Properties of BPM

In this section, we present BPM's properties. These properties are summarized in Table 6.3. For each property, we give an intuitive description, state the property, and either follow with a proof (under certain conditions) or a qualitative argument.

The main property of BPM is to enable autonomous participants to collaboratively handle their peak load by using pre-negotiated, private, pairwise contracts. Contracts enable participants to develop or leverage preferential, long-term relationships with each other. Such relationships, coupled with pre-negotiated bounded prices provide predictability and stability at runtime. Through these properties, BPM differs from other efforts that propose the use of open markets and auctions, where prices and collaborators typically change much more frequently. We believe that privacy, predictability, stability, and reliance on offline-established relationships make BPM more acceptable in practice. The questions are whether BPM redistributes load between participants well and whether the runtime overhead is not too high. In this section, we answer these questions by analyzing a few properties of BPM. In the following chapter, we further evaluate BPM in different settings by running simulations and experiments.

Property #	Brief description (not the actual statements)
Property 1	The bounded-price mechanism is individual rational
Theorem 1	Following the intended implementation is a Bayesian-Nash equilibrium.
Property 2	Negotiating the final price only when valuation falls within price range and counter-offering with a price slightly higher than the valuation is nearly always an optimal strategy for a seller.
Theorem 2	A small price range suffices to achieve acceptable allocation for under-loaded systems and a nearly acceptable allocation for overloaded systems.
Property 3	The mechanism has low complexity: convergence occurs quickly.
Property 4	The mechanism imposes only a low communication overhead.

Table 6.3: Summary of the bounded-price mechanism properties.

We show that our approach provides sufficient incentives for participants to engage in the mechanism that we have designed (Property 1). We also show that it is in a participant's best interest to follow the intended implementation that we have outlined in the previous sections (Theorem 1 and Property 2). Given that participants follow the intended implementation, we show that a small price range suffices to ensure that a uniform system always converges to acceptable allocation (Theorem 2). BPM is also indirect, algorithmically tractable, and distributed. BPM is indirect because participants reveal their costs only indirectly by accepting or offering load to their partners. The implementation is distributed as there is no central optimizer. Participants negotiate directly with one another. BPM is also algorithmically tractable because it has, as we show in Property 3, a polynomial-time computational complexity (*i.e.*, the convergence to final load allocation occurs quickly) and a worst-case communication overhead polynomial in the amount of load that needs reallocating (Property 4).

6.8.1 Additional Assumptions

To facilitate our analysis, and simplify the stated properties, we make three assumptions. First, we assume that participants are *risk neutral*, *i.e.*, they are willing to pay (or incur a cost) for a load movement up to the amount they expect to gain from that movement. BPM does generalize to participants with different attitudes toward risk. Such participants simply use different thresholds to decide when to engage in a load movement. Risk-averse participants need a risk premium: they engage only in load movements with expected gains exceeding costs by some pre-defined margin (proportional to the expected costs). Risk-seeking participants may gamble and move load even when expected gains minus costs are negative. The attitude toward risk also affects the counter-offers that a participant makes when negotiating the final price within a contracted range. We ignore these risk adjustments to simplify the properties.

Second, we assume that participants evaluate expected costs and gains without considering possible future movements. With this simplifying assumption, a participant never accepts a load movement that is expected to lower its utility hoping to offset the movement later with another more profitable one. This assumption enables us to significantly reduce and simplify the strategy space of participants.

Third, we assume, that each participant can correctly estimate its expected load conditions for the *current* time interval d_i, and that actual load conditions match the expected ones. Participants use these estimates to make load movement decisions. We also assume, however, that load distributions are memoryless at all participants between these time intervals, *i.e.*, the load at time interval d_{i+1} does not depend on the load during interval d_i. This assumption enables us to avoid considering strategies where participants try to anticipate their future load conditions or those of their partners. In practice, this property may hold for large enough time intervals.

6.8.2 Properties

Because participants are not forced to use our approach, we start by showing that participation in the bounded-price mechanism is beneficial. Intuitively, participation is beneficial because each participant has the option at runtime to accept only those movements expected to improve its utility. More specifically, our mechanism has the following property:

Property 1 *The bounded-price mechanism is* individual rational*: i.e., the* expected *utility from participation is greater than the utility from non-participation.*

Precondition*: For each participant, the actual load for the current time period, d_i, equals the expected load for that time period.*

Proof. If participants choose to take part in the mechanism, their runtime strategies are constrained to accepting and offering load at a price within the contracted range, after possible negotiation.

Under static load, if participants move load to a partner or accept load from a partner only when doing so increases their utility (*i.e.*, when the marginal processing cost is strictly higher or strictly lower than the contract price), then it is straightforward that participation improves utility.

Under variable load, participants move load only for a time-period d (*i.e.*, for the current interval d_i). When the time expires, either participant can request that the load moves back, reverting to an allocation that would have existed if they did not participate in the mechanism. Assuming participants can correctly estimate their expected load level for the current interval, d_i (such that the actual load level equals the expected one), moving load only when doing so improves the expected utility for that time period results in a higher expected utility than none participation.

Hence, under both static and dynamic load participants benefit from the bounded-price mechanism. ▮

We now explore the strategies that participants should adopt to maximize their utilities. We show that following the intended implementation presented in Section 6.3 is the best-response strategy[5] of a participant given the knowledge of memoryless load distributions at other participants and assuming that participants are, in general, rarely overloaded. We first consider the fixed-price runtime game.

In the fixed-price game, BPM's implementation is a simple and natural strategy. A participant offers load to its partners when its marginal cost is above the contracted price and it accepts load offers when its marginal cost is below that price. Intuitively, this strategy is optimal for a buyer when the latter tries to shed load through its least expensive contracts first. This strategy is also optimal for a seller when participants are, in general, lightly

[5]The best-response strategy is the strategy that maximizes the expected utility of a participant given the (expected) strategies adopted by other participants

loaded, because refusing a load offer is equivalent to losing an opportunity to improve utility over the current time-period, d_i without a good chance that a more profitable opportunity will present itself during the same time period. It is possible to formalize the above reasoning into a argument under our simplifying, yet reasonable assumptions about players and their load distributions. We now present this formalization.

In the fixed-price contract game, with only two participants, accepting or offering load when doing so improves the expected utility is a *dominant strategy*. It is the strategy that leads to the highest expected utility, independent of what the other participant does. For multiple participants and contracts, even though participants can only accept or offer load at the contracted price, the strategy space is richer. Participants may, for instance, try to optimize their utility by accepting too much load with the hope of passing it on at a lower price. Participants can also postpone accepting or offering load with the hope of performing a more profitable movement later. The first strategy violates our assumption that participants never accept load movements that lower their utility. We show that the second strategy does not improve utility when load distributions are memoryless and participants are generally lightly load. More specifically, we show the following theorem:

Theorem 1 *Suppose load distributions are independent and memoryless, and the pre-defined contract order sorts contracts by increasing contract price. In the fixed-price contract game, offering load through any contract that improves utility and accepting all offers that improve utility is a* Bayesian-Nash equilibrium*; i.e., it is the best-response strategy against the expected strategies adopted by other participants.*

Precondition*: For each participant, the actual load for the current time period, d_i, equals the expected load for that time period. Participants must be generally lightly loaded, producing load offers with a sufficiently small probability to ensure that expected gains from an expected load offer are small. These expected gains should be smaller than potential gains from an actual load movement of a single task through any other contract.*

Proof. We prove the above property by showing that other strategies do not produce a higher expected utility. Two other strategies are possible: performing load movements that potentially decrease utility, or refusing load movements that potentially increase utility.

Because we assumed that a participant never performs a load movement that decreases its expected utility, the only other possible strategy is to delay offering or accepting load in the hope of performing a more beneficial movement later. We show that delaying a load movement does not improve expected utility at a buyer nor a seller. We start by examining the strategy of a buyer.

First, it is clear that offering load when doing so improves expected utility is a better strategy than processing load locally. Assuming a buyer tries its contracts by increasing contract price, the optimal strategy for a buyer is to always try to move load through the first profitable contract. If the partner refuses the load, however, given that prices are fixed, the buyer has two choices: try the next, potentially more expensive contract or wait for load to decrease at the first partner. Because BPM requires that participants retry an offer no sooner than d time-units after the previous one, the buyer has the choice to move load for current time interval d_i using a potentially more expensive contract or keep processing load until time interval d_{i+1}. If the price of the more expensive contract is below the current unit marginal cost, using the more expensive contract increases utility compared with processing the load locally. Hence, the best strategy for the buyer is to try all its contracts in turn rather than delaying a load movement.

We now examine the strategies of the seller. When a seller receives a load offer that improves its utility, it can either accept the load for a time-period d (*i.e.*, for the current interval d_i) or reject it in the hope that an offer will come through a more expensive contract within that same time-period. More precisely, the offer should be rejected if the expected utility increase from a probable offer is greater than the expected utility increase from the given offer. A seller could anticipate a load offer if it knows the load conditions at its partners. Because load distributions are memoryless, the seller cannot deduce these conditions from earlier conditions it may have observed. Because BPM prevents a buyer from making more than one offer within an interval d_i, the seller could not have observed the current conditions without already having received an offer from this partner. The seller thus cannot anticipate the load conditions at its partners. Because we assume (in the pre-condition of the theorem) that participants are generally lightly loaded such that potential gains from expected load offers are below potential gains from concrete load movements, the best strategy for the seller is to accept the load offer.

A seller could have a partner that offered load in a deterministic manner, independently of its load level. With such a partner it could be possible for the seller to anticipate load movements, and reject profitable offers. Because such partner's strategy is not a best-response strategy, however, the seller cannot expect the existence of such a partner. By definition of a Bayesian-Nash equilibrium, the strategy of the seller must be a best-response to the *expected* strategies of other nodes.

Therefore, following our intended implementation of accepting and offering load whenever doing so improves utility, is the best-response strategy of both buyers and sellers, assuming that the load distributions at all participants are known to be independent and memoryless, and assuming that, at any time, the probability of any participant buying resources is small. ▮

With bounded-price contracts, participants must negotiate the final price at runtime. In Section 6.6, we proposed a negotiation protocol where a seller either accepts the lowest price within the range or counter-offers a higher price. We discussed different conditions when either accepting the lowest price or counter-offering a higher price yields a potentially greater increase in utility. We now show the following, more specific, property:

Property 2 *Suppose the distribution of unit marginal costs at participants is independent and uniform within range* $[0, 1]$*, and that each participant has* C *contracts with a price range* $[p_L, p_H]$ *such that* $0 < p_L < p_H < 1$ *and* $p_H - p_L << 1.0$*. In the bounded-price contract game, it is nearly always optimal for a seller to accept the lowest price when the unit marginal cost for the offered task is below the price range and to counter-offer with a price slightly higher than its valuation, when its marginal cost is within the price range. When the marginal cost exceeds the range, the seller must reject the offer.*

We only give a sketch for the argument as we do not compute the exact value for the factor by which a seller should increase its valuation when its marginal cost falls within the contracted range. This value depends on the overall load of the system (*i.e.*, it depends on the actual load distributions).

We first examine the case when a seller's valuation, S (*i.e.*, average unit marginal cost for the offered task), is below the price range. We show that the best strategy is to accept the lowest price. In BPM, a seller does not know about other competitors with certainty but it expects the buyer to have a set of C contracts. The seller maximizes its utility by accepting the lowest price instead of counter-offering a higher price, p_2, when the certain profit, $p_L - S$, is greater than the expected profit, $(p_2 - S)P(p_2)$. A very simple upper bound for $P(p_2)$ is the probability, $(1 - p_L)^{C-1}$, that all other participants have a unit marginal cost above p_1 and simply cannot accept the lowest price, p_1. Even with this approximation, accepting the lowest price is beneficial as soon as:

$$\begin{aligned} p_L - S &> (p_2 - S)(1 - p_L)^{C-1}. \\ S &< \frac{p_L - p_2(1 - p_L)^{C-1}}{1 - (1 - p_L)^{C-1}}. \end{aligned} \tag{6.19}$$

Because p_2 can be at most p_H:

$$S < p_L\left(\frac{1 - \frac{p_H}{p_L}(1 - p_L)^{C-1}}{1 - (1 - p_L)^{C-1}}\right). \tag{6.20}$$

For example, if the price range is $[0.75, 0.8]$ and $C = 5$, the seller should accept the lowest price as long as $S < 0.9997p_L$. Hence, when the price range is small or when the number of contracts is large, the seller should accept the lowest offered price as long as its valuation, S is below p_L.

We now examine the seller's strategy when $S \in [p_L, p_H]$. As discussed in Section 6.6.2, in a sealed-bid first-price auction, where C buyers compete for a good, and their valuations are uniformly distributed within range $[0, 1]$, the strategy to submit a bid that is a fraction of the valuation by a factor $\frac{C-1}{C}$ is a Nash equilibrium strategy [172]. In a reverse auction, and with a price range, the factor applied to the discount offered by the seller translates into the following counter-offer: $p_2 = p_H - \frac{\beta C-1}{\beta C}(p_H - S)$, where $\beta C - 1$ is the number of competing sellers producing counter-offers within the price range. To determine the strategy of the seller, we thus need to compute this number of competing sellers. Because a counter-offer may result in receiving the load *only if* all competing sellers have a marginal cost above p_L, the seller should always consider that this is the case when producing a counter offer. This situation is more likely to arise when the total load on the system is high. If all competitors have a marginal cost above p_L, only those with a marginal cost also below p_H are actually competing for the load. Because the price range $[p_L, p_H]$ is small, if the total offered load is high, many participants will have a load above p_H. However, *each* participant has a set of C contracts that it uses to redistribute any load above p_H, increasing the number of participants with a unit marginal cost below p_H. Thus, if a seller's marginal cost is within price-range, it should counter-offer a price based on the assumption that β is close to one.

In summary, it is nearly always optimal for a seller to accept the offered price when its valuation is below p_L. For a valuation within $[p_L, p_H]$, counter-offering a price slightly higher than the valuation maximizes expected utility. When the seller's valuation is above p_H, the seller should reject the offer.

The negotiation that we propose is thus efficient (*i.e.*, participants do not waste much time or resources on negotiation). Typically, lightly loaded sellers accept the lowest price directly, for a one-step negotiation. In the rare case when all sellers are overloaded, the buyer directly accepts the lowest counter-offer. In the even less frequent scenario when all sellers and the buyer have marginal costs within the range, and their marginal costs are similar, participants may fail to move load and may have to wait for a time period d

before trying again. If a seller loses an offer and the same offer comes back, it is likely that the buyer could not accept the offer and had to wait before retrying. The seller should then counter-offer a lower value to improve the chances of a successful, mutually beneficial load movement. By re-adjusting their counter-offers, participants can quickly agree on a mutually beneficial final price.

Overall, we have shown that it is beneficial for participants to take part in BPM and that their best strategy (under some simplifying assumptions) is to follow the intended implementation. We now discuss the algorithmic properties of BPM. To simplify our analysis, we assume a system with homogeneous nodes and contracts. We study heterogeneous systems in Chapter 7. Before examining the convergence time and communication overhead, we first consider the conditions necessary to guarantee convergence to acceptable allocations, the main goal of BPM.

In BPM, a load transfer takes place only if the marginal cost of the node offering load is strictly greater than that of the node accepting the load. Because cost functions are convex, successive allocations strictly decrease the sum of all costs and movements eventually stop under constant load. If all participants could talk to each other, the final allocation would always be acceptable and *Pareto-optimal*, *i.e.*, no agent could improve its utility without another agent decreasing its own utility. In BPM, however, because participants establish only a few contracts, and exchange load only with their direct partners, this property does not hold. Instead, for a given load, BPM limits the maximum difference in load levels that can exist between participants once the system converges to a final allocation. This property and the computation of the minimal price range yield the following theorem:

Theorem 2 *If nodes, contracts and tasks are homogeneous, and contracts are set according to Lemma 1, the final allocation under static load is an acceptable allocation for underloaded systems and a nearly acceptable allocation for overloaded systems.*

Precondition*: All participants have cost functions that are monotonically increasing and convex. Any node can run any task.*

For convenience, we repeat Lemma 1 here:

Lemma 1 *In a network of homogeneous nodes, tasks, and contracts, to ensure convergence to acceptable allocations in underloaded systems, the price-range in contracts must be at least:* $[\texttt{FixedPrice}-\delta_{M-1}(\texttt{taskset}^{\texttt{F}}), \texttt{FixedPrice}]$*, where* M *is the diameter of the network of contracts and* $\texttt{taskset}^{\texttt{F}}$ *is the set of tasks that satisfies* $\texttt{MC}(u, \texttt{taskset}^{\texttt{F}} - u) \leq \texttt{load(u)} * \texttt{FixedPrice}$ *and* $\texttt{MC}(u, \texttt{taskset}^{\texttt{F}}) > \texttt{load(u)} * \texttt{FixedPrice}$*.*

Proof. We separately examine an overloaded and an underloaded system. For an overloaded system, we show that if at least one node remains overloaded in the final allocation, then all nodes operate at or above their lower contract bound, and the system is in nearly acceptable allocation. For an underloaded system, we show that it is impossible for any node to have a unit marginal cost above the contracted range in the final load allocation, making the allocation acceptable.

We first examine the conditions necessary for convergence to stop under static load. In BPM, because cost functions are monotonically increasing and convex, and because load moves only in the direction of decreasing marginal costs, convergence follows a gradient descent. Convergence stops only when it is no longer possible to move any task between any pair of partners. Suppose a node N_i has a load level $\texttt{taskset}_i$. N_i, cannot move any load to any partner if and only if $\texttt{taskset}_i$ is below the contracted range, the load at each partner, N_{i+1}, is above $\texttt{taskset}_i - u$, or the load at each partner is above the contracted range.

Using the conditions necessary for convergence to stop, we show that in the final load allocation in an overloaded system, all nodes have a load level at or above $\texttt{taskset}^F - (M-1)u$, assuming the contract and system conditions from Lemma 1. Suppose, in the final allocation, a node, N_0, exists with a unit marginal cost greater than FixedPrice. By definition of $\texttt{taskset}^F$, the load at N_0 is at least $\texttt{taskset}^F + u$ (one task above capacity). Because the system has reached its final load allocation, no additional load movement is possible, and none of the excess tasks from N_0 can move to any of its partners. In this case, no movement occurs only if each partner N_1 of N_0 has a load level of at least $\texttt{taskset}^F$. If the load level at any N_1 was even one task lower, by definition of $\texttt{taskset}^F$, the partner could accept at least one excess task from N_0 and the allocation would not be final. Similarly, any partner N_2 of N_1 must have a load level of at least $\texttt{taskset}^F - u$. Otherwise, a load movement would still be possible from N_1 to N_2. By induction on the path length, any node N_i, i hops away from N_0 must have a load level of at least $\texttt{taskset}^F - (i-1)u$, as long as the load at N_{i-1} is within price range. Because the diameter of the contract network is M, each node in the system is at most M hops away from N_0. The load level at such node, N_M is at least $\texttt{taskset}^F - (M-1)u$ because the load at their direct partners, one hop closer to N_0, is at least $\texttt{taskset}^F - (M-2)u$, and that higher load is still within the price range. Therefore, all nodes in the system have a load level of at least $\texttt{taskset}^F - (M-1)u$, and by definition, the system is in nearly acceptable allocation.

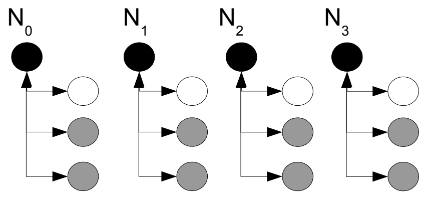
We now consider an underloaded system and show, by contradiction, that no node can have a final load above capacity. The argument for overloaded systems shows that, if at least one node is above its capacity, all nodes operate at or above their capacity at the lower bound of the contract price-range. By definition, such a load distribution indicates that the system is overloaded. By contradiction, a node, N_0, with a load level above its capacity cannot exist in an underloaded system. ▮

According to the above theorem and lemma, a small price range suffices to achieve acceptable allocation for underloaded systems and a nearly acceptable allocation for overloaded systems. For a network diameter of M, the width of the price range is only: $\delta_{M-1}(\texttt{taskset}^{\mathsf{F}})$. In Chapter 7 we show that in randomly selected configurations, an even smaller price range enables the system to reach nearly acceptable allocations.

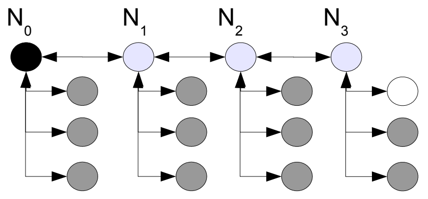
We now examine the computational complexity (*i.e.*, convergence time) and communication overhead of the bounded-price mechanism. In general, offline-negotiated contracts, especially fixed-price contracts, make the runtime game much simpler. They lead to a low computational complexity in most configurations, and a communication overhead significantly smaller than with auctions. We first examine the computational complexity and show the following property:

Property 3 *Suppose each participant, i, has C contracts at a price corresponding to its pre-defined capacity, T_i, and the total fixed excess load is K tasks. BPM has a best-case convergence time of $O(1)$ and a worst-case convergence time of $O(KC)$.*

In this property, K, is the sum of the excess load at all participants. An absolute total of K tasks must be reallocated.



(a) Best-case



(b) Worst case

Figure 6-15: Examples of best-case and worst-case load distributions. Overloaded nodes are black, nodes loaded to upper price-bound are dark gray, nodes loaded to lower price-bound are light gray, and nodes with significant spare capacity are white. Arrows indicate bilateral contracts. In the best case, overloaded nodes move their excess load in parallel to their first partner. In the worst case, excess load must propagate through a chain of contracts.

The cost of moving load is proportional to the number of tasks moved and the size of their state, but we do not consider state size in our complexity analysis. A node selects the set of tasks to offer or to accept and computes their marginal cost at most once for each load offer it produces or receives. Marginal cost computations can increase with the number of offered tasks, but we assume that the increase is negligible (this may not be the case in all applications). In the best case, overloaded participants simultaneously contact different lightly loaded nodes, which accept all excess load at the offered price. All offers and responses proceed in parallel as do the movements of the K tasks, resulting in an $O(1)$ convergence time. Figure 6-15(a) illustrates the best-case configuration. Auctions are computationally more complex, and, during convergence, each node is likely to participate in multiple auctions at the same time, risking overbooking or underutilizing its resources. Overbooking does not occur with BPM when actual load conditions match expected ones.

In the worst case, fixed-price contracts do not lead to acceptable allocation. With bounded-price contracts, because convergence follows a gradient descent, the worst-case is when convergence requires load to propagate through a long chain of nodes. Figure 6-15(b) illustrates this worst-case configuration. In the example, all excess load is located at one node, and all other nodes in the system are exactly at capacity, except for one node that has significant spare capacity. If nodes counter-offer a price for one task, each load movement reallocates exactly one task. In each iteration, half the nodes in the chain offer a task and the other half accepts a task. Because a total of K tasks must move, the convergence time is thus $2K$, independent of the depth of the chain. Additionally, because each node has C contracts, it must contact all C partners in turn every time it negotiates a final price within range. The worst-case convergence time is thus $O(KC)$.

With auctions, the worst-case convergence time is a little lower because nodes communicate with their C partners in parallel and nodes can also potentially move more tasks at each iteration. The depth of the chain, however, bounds the number of tasks moving together. The size of these groups also decreases as the system gets closer to an even load distribution.

The bounded-price mechanism has thus a lower complexity than auctions in the best case, and does not necessarily perform much worse in the worst case. The best-case and worst-case complexities are extreme examples, though. In general, with bounded-price contracts, load may neither be absorbed by direct neighbors nor has to propagate through a single chain. Rather, load propagates on parallel paths created by the contract network. We can thus expect convergence times to decrease quickly with each additional contract that nodes establish. We investigate these situations in Chapter 7, confirming these hypotheses.

We now examine the communication overhead of our approach and compare to that imposed by auctions. We show the following property:

Property 4 *Suppose each participant, i, has C contracts at a price corresponding to its predefined capacity, T_i, and the total fixed excess load is K tasks. To converge, BPM imposes a best-case communication overhead of $O(1)$ and a worst-case overhead of $O(MKC)$, where M is the diameter of the contract network.*

With our mechanism, most load movements require just three messages: an offer, a response, and a load movement. Load movement messages impose the greatest overhead when tasks move with their state. We do not consider this overhead here. Offer and response messages are small: they only contain lists of tasks, statistics about resource utilization, and prices. We consider that the per-message overhead is constant.

The best-case communication overhead for the bounded-price mechanism is thus $O(1)$. In contrast, an approach based on auctions has a best-case per-movement communication overhead of $O(C)$. Because with auctions the system converges to uniform load distributions, the overall overhead can grow to be much larger than $O(C)$. If participants must move K tasks through a chain of M nodes, the worst-case overhead of BPM may be as high as $O(MKC)$. The overhead could decrease to $O(M(K + C))$ if overloaded nodes told their partners to stop sending them offers after the first failed offer. With auctions, nodes always send each offer to their C partners, but they can potentially move more tasks at once through a chain.

Compared with auctions, BPM thus significantly reduces both communication overhead and computational complexity in the best case. In the worst-case, both techniques can have a high complexity and overhead. BPM can potentially do a little worse. One of the main advantages or BPM, however, is the overall smaller number of load movements. Load moves only when a participant's load exceeds a contract price while some of its partners have spare capacity, or a participant's load falls below a contract price while some of its partners are overloaded. Load variations away from contract boundaries do not cause load movements, as we discuss further in the next chapter.

6.9 Summary

In this chapter, we presented BPM and some of its properties. The basic idea of BPM is for participants to establish bilateral contracts offline. These contracts specify a small range of prices that participants must pay each other at runtime for processing each other's load. Contracts can also specify a load movement duration, which determines the minimum amount of time before either partner can cancel an earlier load movement. We showed that the best strategy for a participant (under some conditions) is to follow our intended implementation of offering and accepting load when doing so improves its utility, and that a small price range ensures that a homogeneous system always converges to either an acceptable or a nearly acceptable allocation. In the next chapter, we complete BPM's analysis with simulations, describe its implementation in Borealis, and show results from experiments with our prototype.

Chapter 7

Load Management: Evaluation

In the previous chapter, we presented BPM and some of its properties. In this chapter, we complete the analysis and evaluation, showing simulation results, implementation details, and a few experiments on a real workload.

Simulating random networks of contracts and random initial load allocations, where the load at each node is drawn from the same skewed distribution, we find that BPM works well in heterogeneous environments, where nodes with different capacities establish contracts at different prices. We find that a small number of contracts per node suffices to ensure the final allocation is nearly acceptable for both underloaded and overloaded heterogeneous systems. Interestingly, even though they do not always lead to an acceptable allocation, we find that fixed-price contracts often lead to good load distributions. A small number of fixed-price contracts per node enables these nodes to reallocate most excess load and use most of the capacity available in the system. Additionally, fixed-price contracts lead to extremely rapid convergence to a final load allocation. With price-ranges, convergence can sometimes take a long time, but 95% of its benefits occur approximately within the first 15% of convergence time in the simulated configurations. Under dynamic load, we show that even with small price ranges, BPM absorbs a large fraction of load variations. Finally, we show that BPM works well on load variations that may appear in a real network monitoring application.

In Section 7.1, we present and discuss all simulation results. In Sections 7.2 and 7.3, we present the implementation of BPM in Borealis and show results from experiments with our prototype implementation on a real workload. Finally, we discuss some limitations of BPM before concluding with the main lessons learned in Section 7.4.

7.1 Simulations

In this section, we present the simulation results. We first describe our simulator and the system configurations that we simulate (Sections 7.1.1 and 7.1.2). We show that BPM enables a system to converge to good load distributions (Section 7.1.3), reallocating most load quickly, even when nodes produce counter-offers higher than their valuations (Section 7.1.4). Finally, we show that BPM efficiently absorbs a large fraction of load variations, even when nodes are allowed to cancel those load movements that are no longer profitable (Section 7.1.5).

min # of contracts at any node	1	2	3	4	5	6	7	8	9	10
max # of contracts at any node	13	14	15	16	16	17	18	18	18	18
avg diameter of contract topology	20	11	8	7	6	5	5	5	4	4
min diameter of contract topology	17	10	8	7	6	5	5	5	4	4
max diameter of contract topology	22	14	9	7	6	6	5	5	5	5

Table 7.1: Properties of the ten simulated topologies.

7.1.1 Simulator Implementation

We use the CSIM discrete event simulator [113] as our simulation engine. CSIM enables us to model each node as an autonomous process and takes care of scheduling these simulated processes.

Each node implements the load shedding algorithm from Figure 6-4. We force nodes to sleep for a time-period, $D = 1$ simulated second, after every successful load movement or if no load movement is possible. Inserting a delay between load movements is necessary in practice to let load stabilize before computing the new load conditions. Each node also implements the algorithm for accepting load from Figure 6-5 with the difference that we implement each load offer and response with a method call. Therefore, a load offer, response, and possibly the resulting load movement occur instantaneously in our simulator. This idealization is not likely to affect our results because, in practice, the delay to exchange messages and move load are significantly shorter than the time-period D.

For price-range contracts, we also need to simulate price negotiations. Each node implements the price-negotiation algorithm from Section 6.6.2. Hence, when it receives a load offer, a node either accepts the load or subset of the load, rejects the offer, or responds with a counter-offer. Unless indicated otherwise, the counter-offer is equal to the valuation for one task. We simulate and discuss higher counter-offers at the end of Section 7.1.4. Because counter-offers are binding, when a node issues a counter-offer, it temporarily reserves resources for the expected load. Upon receiving a counter-offer, the buyer sends the original offer to another partner. If all partners respond with a counter-offer, the buyer moves load, if possible, to the partner with the lowest counter-offer, and tells other partners they are no longer bound by their counter-offers. We require buyers to wait for a short time-period, $d = 0.025D$, after each counter-offer, such that with 10 contracts, the total negotiation lasts at most a quarter of the interval. These short delays, d, enable us to better simulate interleaved price negotiations, where multiple nodes try to shed load to the same partners at the same time. If no load movement is possible, a node must wait for time D before retrying to offer load at the minimum price.

7.1.2 Topologies Simulated

We simulate a system of 995 participants that we connect with bilateral contracts to form various randomly-generated contract topologies. We simulate topologies with varying *minimum* number of contracts per node. To create a topology, each node first establishes a bilateral contract with an already connected node. Nodes that still have too-few contracts randomly select additional partners. With this algorithm, the difference between the minimum and maximum numbers of contracts that nodes have is small, as shown in Table 7.1. For instance, no node has more than 18 contracts when the minimum number of contracts per node is 10. The table also shows the diameters of the simulated topologies. For each

Contracts	Price Selection
Uniform Fixed	Node capacity 100 tasks. Contract price at 100 tasks.
Uniform Range	Node capacity 100 tasks. Price-range corresponds to [95,100] tasks.
Heterogeneous Fixed	Node capacities uniformly distributed in the range [80,120] tasks. Each contract at price corresponding to the lower of the two partner capacities.
Heterogeneous Range	Node capacities uniformly distributed in the range [80,120] tasks. Same contracts as in fixed-price heterogeneous variant but extended to a 5-task range: [Fixed-5,Fixed].

Table 7.2: Simulated variants of BPM.

minimum number of contracts, we generate 10 different topologies. The table shows the average, minimum, and maximum diameters across the 10 topologies. The first few contracts drastically shrink the diameter of the contract topology from 20 to 8. Additional contracts decrease the diameter much more slowly.

We simulate both uniform and heterogeneous systems. In a uniform system, all nodes have the same capacity. We set this capacity to be 100 tasks (or 100%). To create heterogeneity, we distribute node capacities uniformly in the range $[80, 120]$. This choice is arbitrary, but it enables the difference between node capacities to reach up to 50%. We enable load movements at the granularity of one task (or 1% of total capacity). In our simulator, we abstract away the exact cost functions and marginal costs. We express contract prices directly in terms of load levels. If a contract price is equal to 100, a load movement occurs as soon as the buyer's load is at or above 101 and the seller's load is at or below 99.

We study and compare the convergence properties of four variants of our mechanism. Table 7.2 summarizes the parameters of these four variants:

1. In the *Uniform Fixed* variant, all nodes have the same capacity and the same fixed-price contracts. We set contract prices to be equal to node capacities.
2. In the *Uniform Range* variant, all nodes have the same capacities again, but we extend their fixed-price contracts to cover a 5-task range (*i.e.*, contracts cover the range $[95, 100]$). We use 5-task ranges because such ranges correspond to the second smallest contract-topology diameter that we simulate.
3. In the *Heterogeneous Fixed* variant, node capacities and contract prices vary within $[80, 100]$. When two nodes with different capacities establish a bilateral contract, they use as price the smaller of their capacities. For example, if a node with capacity 84 establishes a contract with a node whose capacity is 110, the contract price is 84.
4. In the *Heterogeneous Range* variant, node capacities and contract prices also vary within $[80, 100]$. When nodes with different capacities establish a bilateral contract, they use as the higher bound for the price-range, the smaller of their capacities. The width of the range is always 5 tasks. For example, if nodes with capacity 84 and 110 establish a contract, the contract covers the range $[79, 84]$.

7.1.3 Convergence to Acceptable Allocations

We study load allocation properties for heterogeneous systems and compare the results to those achieved in a system of uniform nodes and contracts. We find that even a small

Configuration	Load level	k	T	α
Light load	50% total capacity	1	300	0.01
Heavy load	75% total capacity	1	300	-0.22
Light overload	125% total capacity	1	300	-0.70
Heavy overload	150% total capacity	1	300	-1.00

Table 7.3: Probability distributions used in initial load allocations. The distribution is: $F(x) = (1 - (\frac{k}{x})^{\alpha})(\frac{1}{1-(\frac{k}{T})^{\alpha}})$. $[k, T]$ is the range of initial load values. α is the shape parameter. Nodes start with at least one task and at most a load three times their capacity.

number of fixed-price contracts enables a potentially heterogeneous system to converge to nearly acceptable allocations for all load levels simulated. Small price-ranges help the system reach good load distributions with fewer contracts per node.

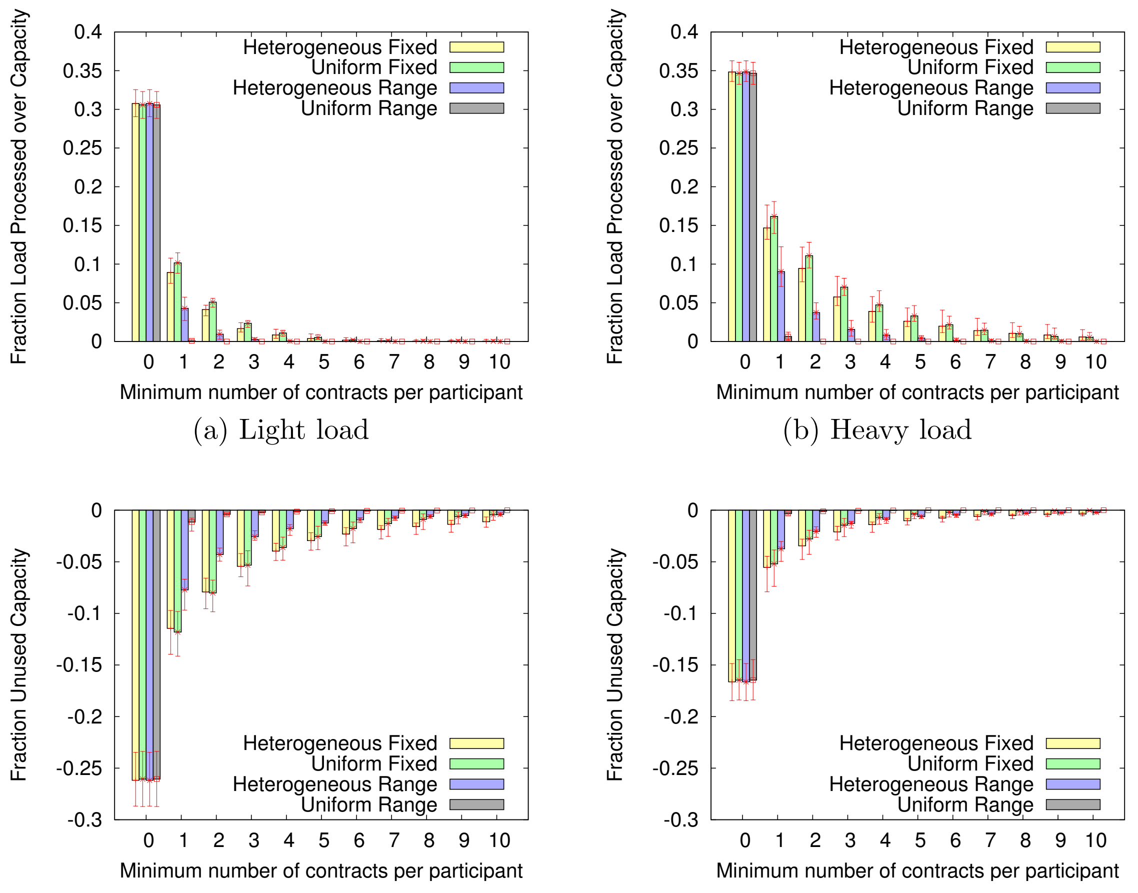
We simulate each of the four variants of BPM and each of the ten topologies under four different load conditions: 1) light load, where the overall load corresponds to 50% of system capacity, 2) heavy load, where the overall load corresponds to 75% of system capacity, 3) light overload, where the overall load corresponds to 125% of system capacity, and 4) heavy overload, where the overall load corresponds to 150% of system capacity. To create each load condition, we draw the load at each node from the same family of distributions as in the previous chapter. Table 7.3 summarizes the exact distributions used for each load level.

Starting from the skewed load assignments, we let nodes move load between each other. Once all load movements stop, we examine how far the final allocation is from being acceptable. Figure 7-1 shows the results. For the two underloaded configurations, the figure shows the fraction of all tasks that remain above the capacity of some nodes. These tasks should still be reallocated but the system is unable to do so. For the two overloaded configurations, the figure shows the fraction of total capacity that remains unused. This capacity should be used but the system is unable to use it. In all four graphs, the column with zero minimum number of contracts shows the initial conditions before any load movements take place. For example, in the light load scenario, the initial allocation is such that 30% of all tasks need to be moved for the system to reach an acceptable allocation. In the light overload scenario about 26% percent of the total available capacity is not being used initially.

Overall, as the number of contracts increases, the quality of the final allocation improves for all variants and all load configurations: nodes manage to reallocate a larger amount of excess load or exploit a larger amount of available capacity. The improvement is especially significant for the first two to four contracts that nodes establish. With approximately 10 contracts per node, the system converges to nearly acceptable allocations for all simulated configurations, *i.e.*, all types of contracts and all load levels.

With uniform price-range contracts, the system reaches an acceptable allocation with as few as two contracts per node for both underloaded configurations (Figure 7-1(a) and (b)). This result is significant because the theoretical analysis from Section 6.6 predicts that up to six contracts per node will be needed. With six contracts, the diameter of the topology is equal to the price-range, which is the necessary condition to ensure acceptable allocation. With random topologies, however, with a price range equal to only half the diameter, the system reaches an acceptable allocation, for all ten topologies simulated.

For overloaded systems (Figure 7-1(c) and (d)), with uniform price-range contracts, BPM only ensures nearly acceptable allocations, *i.e.*, some nodes may operate slightly below their capacities. The results show that with just two contracts per node, less than 0.5% of



(a) Light load (b) Heavy load

(c) Light overload (d) Heavy overload

Figure 7-1: Convergence to acceptable allocation for different total load levels and different numbers of contracts per node. Figures (a) and (b) show the fraction of total load that has failed to be reallocated in underloaded systems. Figures (c) and (d) show the fraction of total capacity that has failed to be used in overloaded systems. Histograms show average values. Error bars show minimum and maximum values from the 10 simulations.

total capacity remains unused. Hence, with just two contracts per node, our approach is able to exploit 99.5% of total system capacity. Interestingly, with a few additional contracts, the system even reaches acceptable allocation. For heavy overload, 8 contracts suffice to ensure acceptable allocation in all simulated topologies. For a smaller overload, a minimum of 10 contracts is necessary for some configurations to start reaching an acceptable allocation.

Interestingly, fixed-price contracts also lead to good load distributions. In a lightly loaded system, with only two contracts per node, the system reallocates on average all load except only 5% of tasks. As few as 5 contracts suffice to bring this value below 1% for *all simulated configurations*. This result shows that fixed-price contracts are highly efficient in handling load spikes in a generally lightly loaded system. Even when the load is high, fixed-price contracts perform quite well. At 75% total load, with only 8 contracts per node, all configurations reallocate all but 2% of excess tasks. For overloaded systems, with as few as 4 contracts per node, the system is always able to exploit over 95% of available capacity. Results are even better for heavily overloaded systems, where 5 contracts per node enable the system to exploit over 99% of total capacity. Hence fixed-price contracts are also effective at handling an overloaded system.

We find that our approach works well in heterogeneous systems as well (such configurations are likely to be more common in practice). As shown in Figure 7-1, although a heterogeneous system does not always achieve acceptable allocations or requires a larger number of contracts to do so, few contracts suffice to achieve nearly acceptable allocations.

More specifically, we find that heterogeneous fixed-price contracts lead to slightly better load distributions than uniform fixed-price contracts in underloaded systems with small numbers of contracts. For up to 7 contracts per node, the heterogeneous system reallocates up to 2% more tasks than the uniform one in both underloaded configurations. Indeed, heterogeneous contracts make it possible for load to move further away from its point of origin, by following a chain of contracts at decreasing prices, enabling the system to achieve better load distributions. Heterogeneity, however, causes inefficiency as load movements between partners are limited by the *lower* of their two capacities. This inefficiency becomes apparent with larger numbers of contracts, when uniform prices start to outperform heterogeneous ones. Inefficiency due to heterogeneity make it especially difficult for BPM to exploit all available capacity in an overloaded system. Uniform fixed-price contracts outperform heterogeneous ones with much fewer contracts in overloaded system. Overall, however, the performance of the uniform and heterogeneous settings is comparable in all configurations.

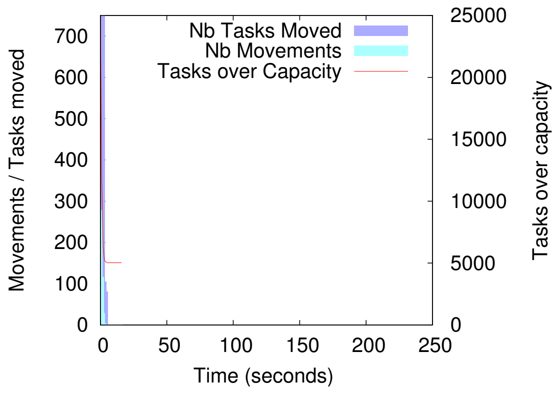
Similar dynamics control the performance of a system with heterogeneous range contracts. The performance is good in general. With as few as two contracts per node, the system reallocates all but 5% of load in the underloaded settings and use over 95% of available capacity in overloaded settings. However, it takes at least six contracts per node for the system to always achieve acceptable allocation when lightly loaded. The system achieves only nearly acceptable allocation under heavier load or overload.

7.1.4 Convergence Speed

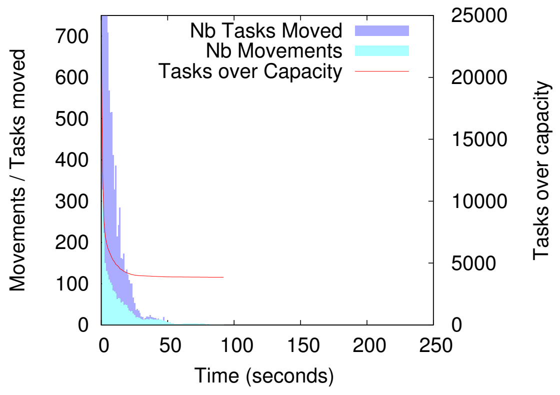
We now study the time it takes for convergence to occur in uniform and heterogeneous systems under different load conditions. We show that, as expected, with uniform fixed-price contracts all load movements occur almost simultaneously as soon as the simulation starts, leading to an extremely fast convergence. Convergence takes longer with price ranges because it takes time for load to propagate through chains of nodes. Most tasks, however, are reallocated rapidly as soon as the simulation starts. Subsequent movements provide decreasingly smaller benefits. In the uniform systems, speed also increases rapidly with the number of contracts per node because load can propagate in parallel in many directions. With price-range contracts, we find that counter-offers higher than valuations do not hurt convergence speed as soon as nodes have more than two contracts each.

Figure 7-2 shows concrete examples of convergence. These examples are from the 75% load level configuration and a minimum of three contracts per node. We chose a configuration with few contracts to show the effects of moving load through chains of nodes. Only uniform range contracts lead to acceptable allocation in this configuration. The results show the total number of tasks moved at every time period, the total number of load movements, and also the total number of tasks that remain to be reallocated. The exact values are not important as they depend on the system settings. The overall trends are the important result. Simulations stop 10 seconds after the last load movement.

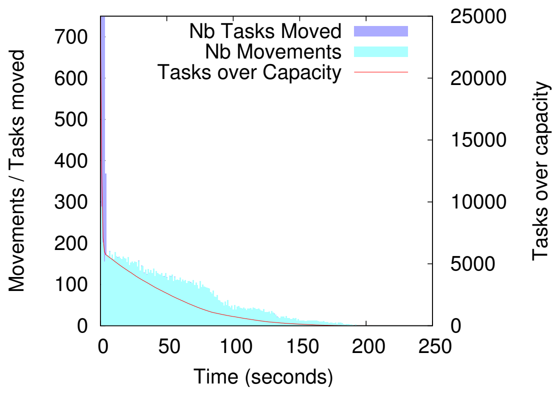
For all variants, most load movements occur within the first few seconds of the simulation



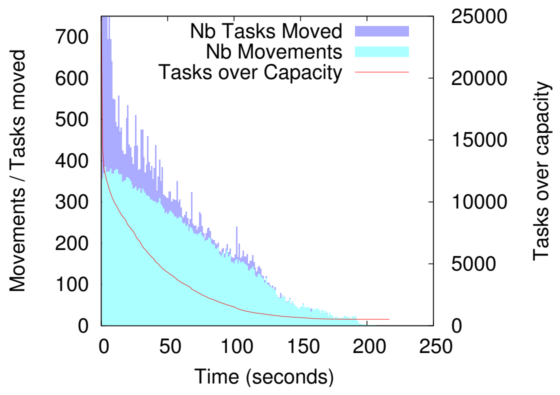
(a) Uniform Fixed



(b) Heterogeneous Fixed



(c) Uniform Range



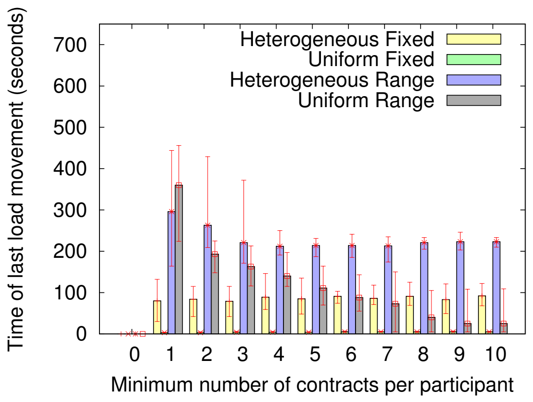
(d) Heterogeneous Range

Figure 7-2: Examples of convergence dynamics. The left y-axis is cut-off at 500. Initial load movement values are significantly greater than 500. In all cases, the first few movements provide the most benefits. Later movements push smaller amounts of load through chains of nodes.

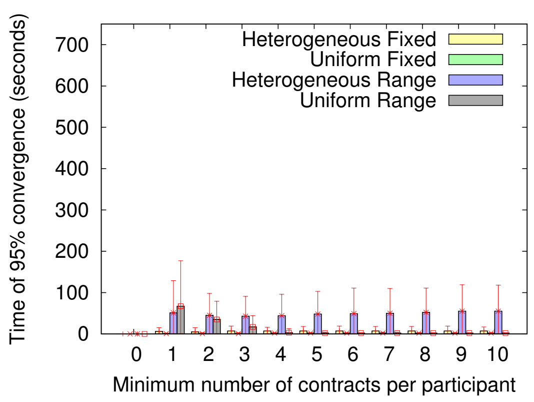
and each one of those early load movements reallocates a large number of tasks: *i.e.*, the number of tasks moved is significantly greater than the total number of load movements, indicating that each movement involves many tasks. Early load movements also provide most of the benefits of the overall convergence as shown by the sharp decrease in the total number of tasks that still need reallocation. This fast convergence is partly explained by the ability of BPM to balance load between any pair of nodes in a single iteration and partly by its ability to balance load simultaneously at many locations in the network.

With uniform fixed-price contracts, convergence quickly stops, because load can only move one hop away from its origin. All load movements thus occur within the first few seconds. Heterogeneous fixed prices allow some additional movements as tasks propagate through chains of contracts with decreasing prices. Nevertheless, load movements also quickly stop.

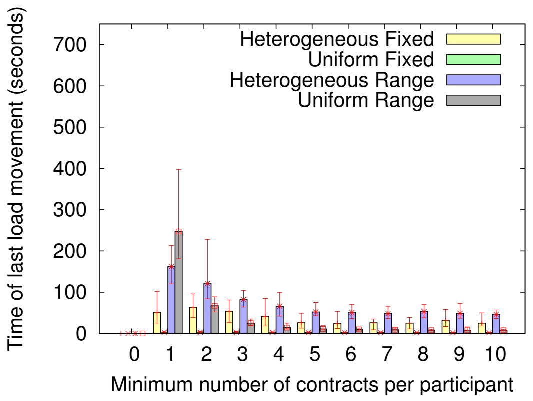
In the uniform configuration, with price-range contracts, convergence goes through two distinct phases. The first phase lasts only a few seconds. In these first few seconds, overloaded nodes move large groups of tasks to their direct partners. Many movements occur,



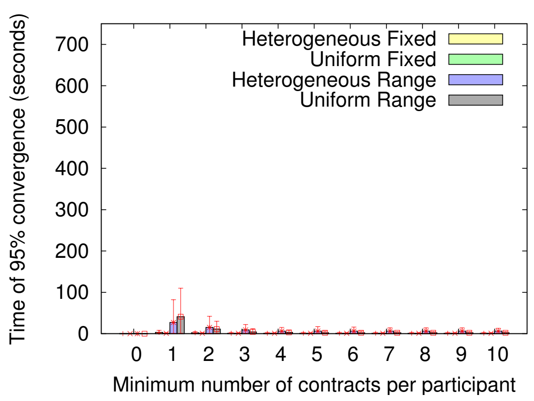
(a) Full convergence, light load



(b) 95% convergence, light load.



(a) Full convergence, heavy overload



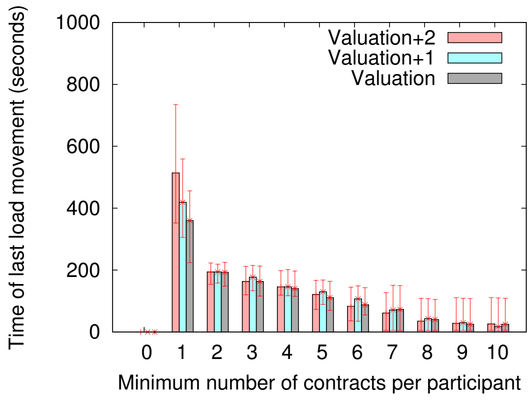
(b) 95% convergence, heavy overload

Figure 7-3: Convergence speed in lightly loaded and heavily overloaded systems. With fixed prices, convergence time is always short, even negligible for the uniform system. With price ranges, converges takes longer but 95% of its benefits occur within approximately the first 15% of convergence time or less.

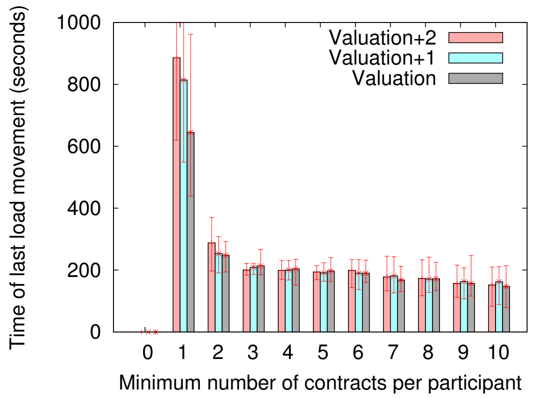
each involving many tasks. The number of tasks that needs reallocation decreases sharply. In the second phase, nodes slowly push the remaining excess load through chains of nodes. The graphs show a long tail of load movements that provide small incremental improvements to the total number of tasks to reallocate. The number of tasks moved at each time unit is equal to the number of load movements, indicating each movement reallocates a single task. This behavior is consistent with propagating load through chains of nodes.

Heterogeneity creates chains of contracts at decreasing prices. Heterogeneity thus reduces the number of tasks moved early on because contract prices are lower than in the uniform environment. Subsequent load movement, however, continue to move groups of tasks rather than one task at the time. This phenomenon occurs because the differences in contract prices are greater in the heterogeneous environment. In the uniform case, nodes can modify prices only within the small contracted range.

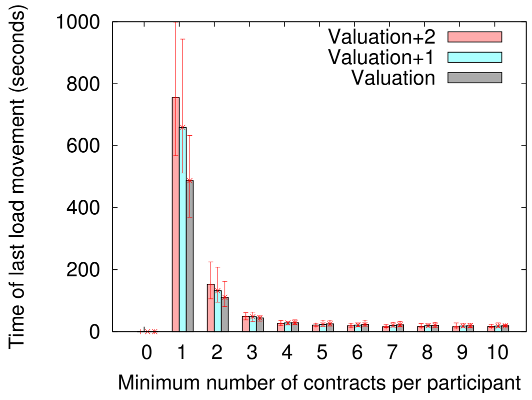
Figure 7-3 shows the final convergence time for multiple simulated configurations. The figure shows both the time of full convergence and the significantly shorter time to get within 5% of the final load distribution. With uniform fixed-price contracts, the conver-



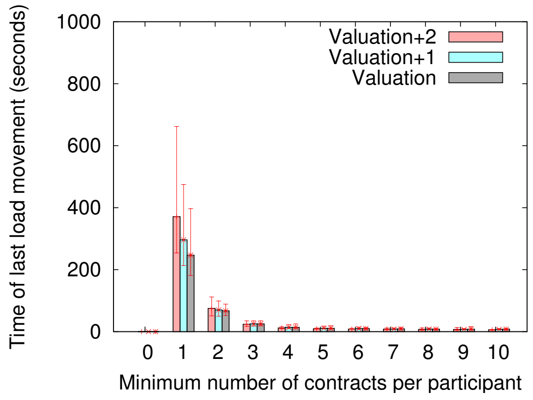
(a) Light load



(b) Heavy load



(c) Light overload



(d) Heavy overload

Figure 7-4: Effect on convergence time of counter-offers above valuation. There is almost no impact as soon as nodes have three or more contracts each.

gence is so short, it is barely visible on the graphs. Convergence always stops within five seconds of the beginning of the simulation. Price-range contracts lead to a longer convergence. In the uniform case, convergence time decreases quickly with each extra contract per node because load can propagate in parallel in many directions. Heterogeneous environments have a slower convergence than their uniform counterparts. Interestingly, when heterogeneous contracts are fixed, convergence time remains approximately constant with the number of contracts. Indeed, instead of simply speeding up convergence, each additional contract enables these system to converge to an allocation closer to acceptable. With price-range contracts, convergence time decreases quickly with the first few contracts per node. Additional contracts do not improve convergence speed much.

In the previous experiments, when negotiating the final price within the contracted range, nodes were always counter-offering their true valuation for one additional task. We now examine the effects of counter-offers above valuations. We modify our simulator as follows. When a node receives an offer while its load level is within the contracted range, it counter offers a price equal to its valuation plus a configurable number x of tasks. If a counter-offer does not result in a load movement, the node counter-offers its exact valuation on the second attempt. Counter-offers above valuations may thus affect convergence time but they have the same properties in terms of final load distribution. Except when the counter-offer equals the maximum price within the range, all counter-offers are for *a single task*. Because counter-offers are binding, when producing a counter-offer, a node *reserves* resources for the potential new task(s). Figure 7-4 shows the results for a uniform system, where nodes counter offer either their valuation or their valuation plus one or two tasks. One or two task differences are significant as the price range covers only a five task interval. Interestingly, except for configurations with one or two contracts per node, counter-offering a price higher than the valuation neither hurts nor helps convergence speed. Indeed, even though counter-offers are higher, nodes still move only one task at the time most of the time. A higher counter-offer causes a load movement to fail only when the buyer's valuation is within a task or two from that of the seller, which occurs rarely with multiple sellers.

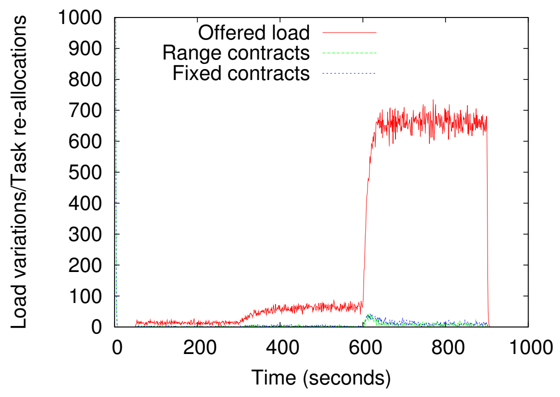
7.1.5 Stability under Changing Load

We now examine how BPM handles changing load conditions. Overall, we find that BPM efficiently absorbs a large fraction of load variations, even when nodes are allowed to cancel those load movements that are no longer profitable. Stability is similar with fixed prices and with small price ranges because the probability that the load of a node falls within range is small when the price-range is small.

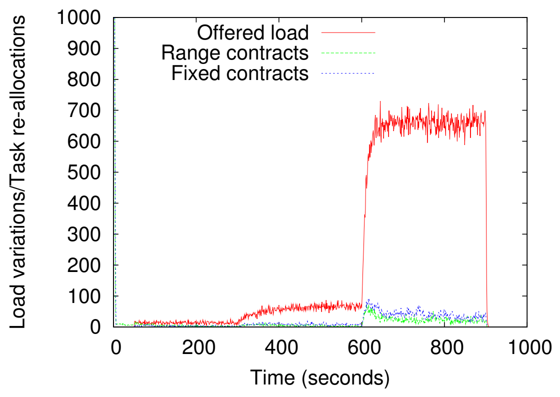
To simulate variable load, we add two processes to each simulated node: one process that periodically increases load and one process that periodically decreases load. The time intervals between adding or removing tasks follow exponential distributions. To simulate increasingly large load variations, we decrease the mean of the distributions. Because the simulator can only handle a maximum of 1000 processes, we decrease the size of the simulated system to 330 nodes. We simulate a topology where each node has at least 10 contracts because such a large number of contracts enables a larger numbers of load movements. The simulated topology has a diameter of four.

Starting from a skewed load assignment, we let the system converge to acceptable allocation, which occurs rapidly with 10 contracts per node. At time $t = 50$ seconds, we start to simulate small load variations using a mean of 50 seconds: every node receives a new task on average every 50 seconds. Because there are 330 nodes in the system, there are on average 6.6 new tasks in the system every second. We use the same distribution for removing load, so on average 6.6 tasks leave the system every second. The total load on the system thus remains approximately constant. At time $t = 300$ seconds, we increase load variations by reducing the mean to 10 seconds. Finally, at time $t = 600$ seconds, we further reduce the mean to 1 second. Figure 7-5 shows the results. Each graph has three curves. The top-most curve shows the variations in the overall offered load, which is the total number of tasks added or removed from the system. The bottom two curves show the total number of tasks reallocated when contracts are fixed or cover a price-range of five tasks. A bad property of the mechanism would be for small load variations to lead to excessive reallocations. Our simulations show that the system handles load variations well. For all load conditions, the system absorbs most load variations without reallocations. Interestingly, fixed and price-range contracts lead to a similar number of load reallocations, while we would expect fixed prices to cause fewer load movements. Because the price range is small, the probability that a node falls exactly within the range is small as well. The overall effect of price-range is thus negligible.

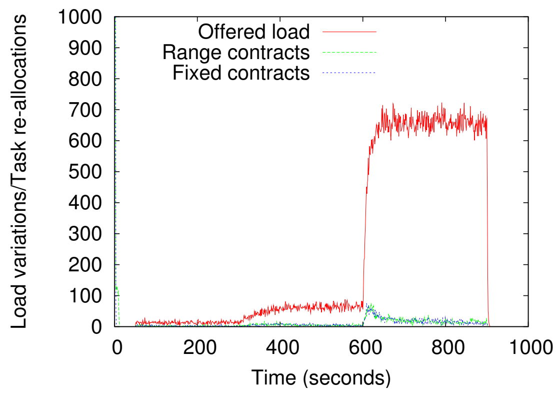
In the previous experiments, once a node moves some of its tasks to a partner, it moves these tasks forever. Load moves back only if it becomes profitable for both partners to use



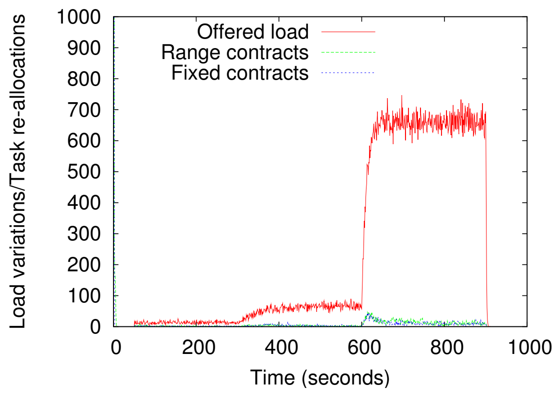
(a) Light load



(b) Heavy load



(c) Light overload

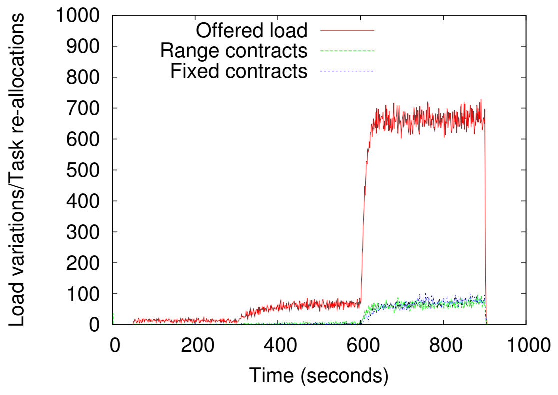


(d) Heavy overload

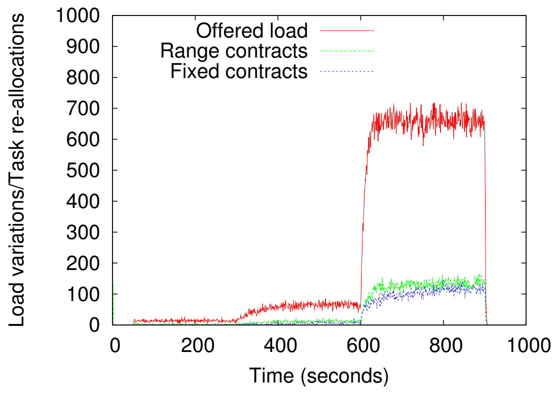
Figure 7-5: Stability under variable load. BPM and its fixed-price variant absorb most load variations without reallocations. The system size is 330 nodes.

their contracts and move load in the other direction. In our approach, however, load moves only for a time-period D. Once the time-period expires, either party can cancel a load movement if they no longer find it profitable. Moving load for a limited time-period can lead to greater instability. To measure the effect of moving load for a limited time-period, we start from an acceptable allocation (we bound the initial load to interval $[1, 100]$) and either a light load (50% load) or a heavy load (75% load). We then apply the same load variations as in the previous simulations. We set the minimum load movement duration, D, to 1 second. Figure 7-6 shows the results. As expected, limiting the minimum load movement duration increases the number of load reallocations. Nevertheless, BPM still successfully absorbs most load variations without reallocations.

In this section, we explored, through simulations, some properties of BPM. We showed that in randomly-generated topologies, a few price-range contracts per node suffice for the system to ensure convergence to acceptable allocations, independent of the overall load level. We also showed that fixed-price contracts and heterogeneous contracts lead to nearly acceptable allocations with only a handful of contracts per node. We showed that convergence to the final load distribution occurs rapidly, especially with fixed-price contracts, and that



(a) Light load



(b) Heavy load

Figure 7-6: Stability under variable load and limited load movement durations. Limiting load movement durations increases the number of load movements but most load variations are still absorbed by the system. The system size is 330 nodes.

in all cases most of the benefits of convergence occur quickly. Finally, we showed that BPM handles variable load conditions well, masking most load variations without causing load movements. In the next section, we focus on the actual implementation of the mechanism in Borealis.

7.2 Implementation

In this section, we describe the implementation of BPM in Borealis v0.2. This early version of Borealis is the Medusa system with just a few extra features.

We could implement our load management protocols at two different levels: between processing nodes or between groups of nodes belonging to the same participant. Our approach can work in both configurations. We decided to experiment with the simpler case, *i.e.*, load management at the granularity of individual nodes. We thus consider each node to be a separate participant, and present the implementation of our approach inside each Borealis node.

As discussed in Chapter 3, we add a Load Manager module to each Borealis node. This module encapsulates all the logic of BPM. For convenience, Figure 7-7 repeats the software architecture of a Borealis node from Figure 3-5. Load Managers on different nodes communicate with each other to make load movement decisions. To move load, the Load Manager at the source node instructs the Admin module of the local Query Processor to perform the load movement and update the Global Catalog. The local Query Processor gathers statistics about local load conditions. The Load Manager uses these statistics as input to BPM. We now present the Load Manager module in more detail, describing first the configuration information necessary for the Load Manager and then the actual protocol implementation. We conclude this section with a discussion of the limitations of our current implementation.

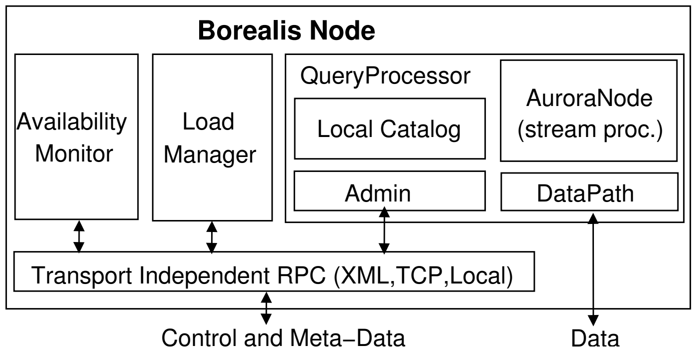


Figure 7-7: Software architecture of a Borealis node.

7.2.1 Static Configuration

The Load Manager needs three pieces of information defined offline: a set of contracts with other nodes, a cost function to compute the marginal costs for processing tasks locally, and a partition of the query diagram into fragments, which define the units of load movements.

Contracts between nodes are configured statically in a file. Each node reads a file called medusa_config.xml, where it finds information about its partners and the prices of their contracts. It would be an easy next step to pass this configuration information dynamically through the Global Catalog. Most importantly, contracts between nodes are defined offline, and each node must be given a set of contracts.

Each node uses the same cost function. Given ρ_{cpu}, the total CPU utilization, and ρ_{bw}, the current output bandwidth utilization, the total cost is computed as:

$$\frac{\rho_{cpu}}{1-\rho_{cpu}} + \frac{\rho_{bw}}{1-\rho_{bw}}. \tag{7.1}$$

This cost function is the total number of tuples being processed or awaiting processing using an M/M/1 approximation for the queuing model. We chose this function as a simple technique for computing load levels and marginal costs using only course grained information about CPU and bandwidth utilization. The cost function is currently hard-coded, but it would be interesting to allow the cost function to be more easily configurable.

In Borealis, when users add operators to the query diagram, they can group these operators into *queries*. A query is thus a fragment of the overall query diagram. Our load management mechanism uses queries as load movement units. With this approach, the user can decide on the most appropriate way to group operators for load movement purposes. The Load Manager does not make these decisions.

The above static information suffices for the Load Manager to run BPM. We now describe the Load Manager's runtime operations.

7.2.2 Load Management Protocol Implementation

The runtime operations consists of gathering and processing statistics about local load conditions, producing load offers when appropriate, and processing load offers received from partners.

To make load movement decisions, the Load Manager needs up-to-date information about the local load conditions. The Query Processor computes this information. More specifically, for each locally running query, the Query Processor measures the data rates on the query input and output streams. We compute average rates using an exponentially weighted moving average. Given an average input rate, the Query Processor computes the approximate CPU utilization by multiplying the input rate by a pre-defined constant. The Query Processor computes bandwidth utilization by dividing the aggregate output rate by a pre-defined constant, as well. These statistics are quite crude and work only for simple query diagrams where the CPU utilization grows with the input rate. New versions of Borealis have significantly improved statistics capabilities.

Given statistics about local load conditions, the load management protocol implementation is quite straightforward. The Load Manager periodically requests statistics from the Query Processor and recomputes the local load conditions. The period is a configurable parameter: longer periods help absorb short load spikes but delay load movements after load conditions change. Given the current load level, the Load Manager sends out load offers and examines the load offers received in the last interval.

To produce load offers, the Load Manager sorts contracts such that failed or unresponsive partners are sorted last, preceded by partners who rejected the last load offer, preceded themselves by all other partners sorted on contract price. Examining each contract in turn, the Load Manager computes the maximum number of queries that it should offer to its partner. As soon as it finds a profitable load offer, the Load Manager sends the offer and stops examining other contracts to ensure that the most profitable load movements are performed. Later, the Load Manager handles responses to load offers asynchronously, and moves operators as soon as they are accepted.

To process load offers received from other nodes, the Load Manager examines offers by decreasing price and decides for each one how many queries to accept, accepting as many queries as possible, and as many offers as possible. If no boxes can be accepted, an offer is explicitly rejected.

Currently, the Load Manager does not optimize the set of queries that it offers or accepts. These queries already correspond to pre-defined load fragments, and the goal is only to achieve an acceptable allocation. When offering or accepting tasks, the Load Manager compares their marginal costs to the contract price plus or minus a small value. We use this technique to avoid oscillations when the load level is close to the contract price. This safety is necessary because our statistics and load levels are not accurate and change with time.

The Load Manager thus basically follows the intended implementation for the fixed-price variant of BPM.

7.2.3 Load Movements

When two Load Managers agree to move load from one node to the other, the source Load Manager sends a load movement request to the local Admin of the Query Processor. The load movement then proceeds as follows:

1. Load movement preparation at the source node: The Query Processor at the source node suspends the operators that must be moved, optionally copies their states into temporary buffers, and starts buffering their inputs.
2. Remote instantiation:
   (a) The Query Processor at the source node sends the query diagram fragment to the Query Processor at the destination node, transmitting the list of boxes, optionally their states, and the list of downstream clients.

    (b) The destination node instantiates the new boxes locally and initializes their states from the copied versions.
    (c) The destination node updates its input stream subscriptions. It also updates its list of downstream clients, opens connections to these clients, and starts processing the new boxes.
    (d) The destination node notifies the global catalog about the new location of the boxes and streams.
3. Clean-up:
    (a) Once the boxes are safely running at the destination node, the source node deletes the boxes from its local processing engine and stops buffering their inputs.
    (b) The source node adjusts its own subscriptions and list of clients.

Newer versions of Borealis follow a slightly different load movement algorithm. For instance, the source node updates the global catalog and pre-computes all changes in connections between nodes. Also, in older versions of Borealis, the Admin module is separate from the Query Processor module.

7.2.4 Implementation Limitations

Our implementation has several limitations, which we now enumerate. First, the prototype still uses the old crude statistics for computing local load conditions. Second, the implementation supports only fixed-price contracts. Third, once load moves, it currently moves forever. Load Managers do not keep track of where there tasks are running and whether it may be profitable to move some load back. Finally, our implementation ignores the cost of moving load and moves load without state to ensure smoother and faster movements. Borealis has the capability to move load with state. It would be interesting to take that cost into consideration and keep the state as we move load.

7.3 Prototype Experiments

We demonstrate how our approach can work in practice by running a network monitoring query on real input data. We deploy the query in a simple federated system composed of three Borealis nodes, which have contracts with each other. We show that the bounded-price mechanism effectively reallocates the excess load that naturally occurs in this application. We also show that the system can scale incrementally. We overload the three node system then add a fourth node at runtime, showing that the system uses the extra capacity to reach an acceptable allocation once again.

The network monitoring query is that from Figure 1-3 but without the final Join operator. We run the query on network connection traces collected at MIT (1 hour trace from June 12, 2003) and at an ISP in Utah (1 day trace from April 4, 2003). To reduce the possible granularity of load movements, we partition the Utah log into four traces that are streamed in parallel, and the MIT log into three traces that are streamed in parallel. To increase the magnitude of the load, we play the Utah trace with a 20× speed-up and the MIT trace with an 8× speed-up.

Figure 7-8 illustrates our experimental setup. Node 0 initially processes all partitions of the Utah and MIT traces. Nodes 1 and 2 process 2/3 and 1/3 of the MIT trace, respectively. Node 0 runs on a desktop with a Pentium(R) 4, 1.5 GHz and 1 GB of memory. Nodes 1 and 2 run on a Pentium III TabletPC with 1.33 GHz and 1 GB of memory. The nodes communicate over a 100 Mbps Ethernet. All clients are initially on the same machines as

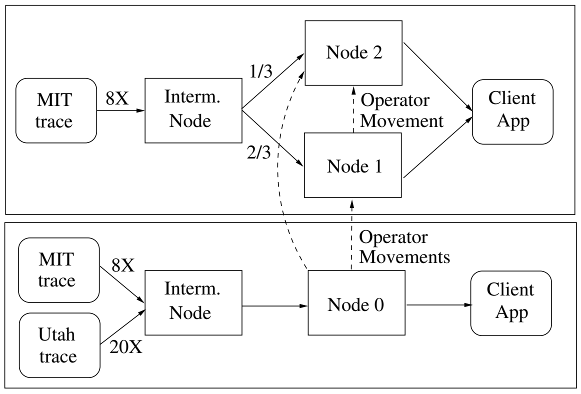


Figure 7-8: Experimental setup.

the nodes running their queries. All Borealis nodes have fixed-price contracts with each other and are configured to take or offer load every 10 seconds.

Figure 7-9 shows the results obtained. Initially, the load at each node is approximately constant. At approximately 650 seconds, the load on the Utah trace starts increasing and causes Node 0 to shed load to Node 1 twice. After the second movement, load increases slightly but Node 1 refuses additional load making Node 0 move some operators to Node 2. The resulting load allocation is not uniform but it is acceptable. At around 800 seconds, Node 1 experiences a load spike, caused by an increase in load on the MIT trace. The spike is long enough to cause a load movement from Node 1 to Node 2, making all nodes operate within capacity again. Interestingly, after the movement, the load on Node 1 decreases. This decrease does not cause further reallocations because the allocation remains acceptable.

Figure 7-10 shows screenshots from an application that periodically queries Borealis nodes to get information about their current load conditions and about the recent load movements they performed. The initial setup in this experiment is the same as above: three nodes with identical contracts with each other. Figure 7-10(a) shows a similar scenario to the first few movements from Figure 7-9. In Figure 7-10(b), however, we overload the system such that it operates in a nearly acceptable allocation. Node 0 operates at capacity. Node 1 is overloaded. Node 2 operates just below the contract price. We add a fourth node, Node 3, to the system. This node has a contract only with Node 2 but this contract is at a lower price than the other three contracts. The figure shows how Node 2 moves some load to Node 3 thus freeing capacity to accept the excess load from Node 1. Node 2 then moves this excess load to Node 3 because it is cheaper to pay Node 3 rather than process the load locally. The system reaches an acceptable allocation, once again. This second experiment shows that our approach can scale incrementally: nodes can be added to the system at runtime.

7.4 Limitations and Extensions

We now discuss the limitations of our approach and possible extensions to address these limitations.

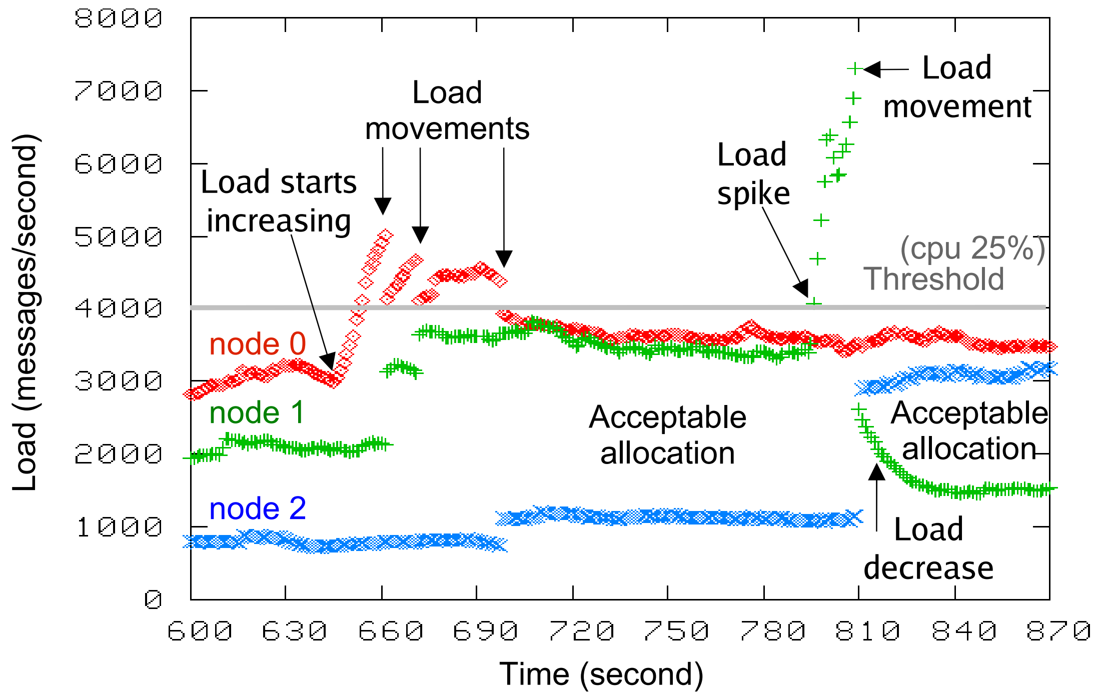


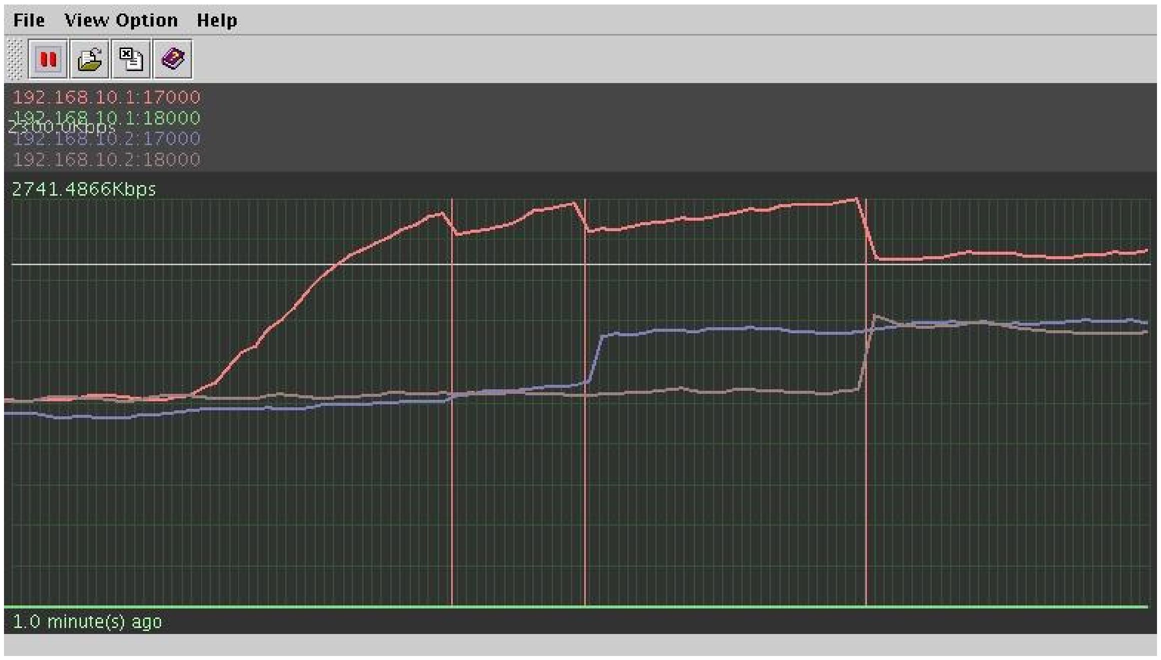
Figure 7-9: Load at three Borealis nodes running the network monitoring query over network connection traces.

7.4.1 No Resource Reservation

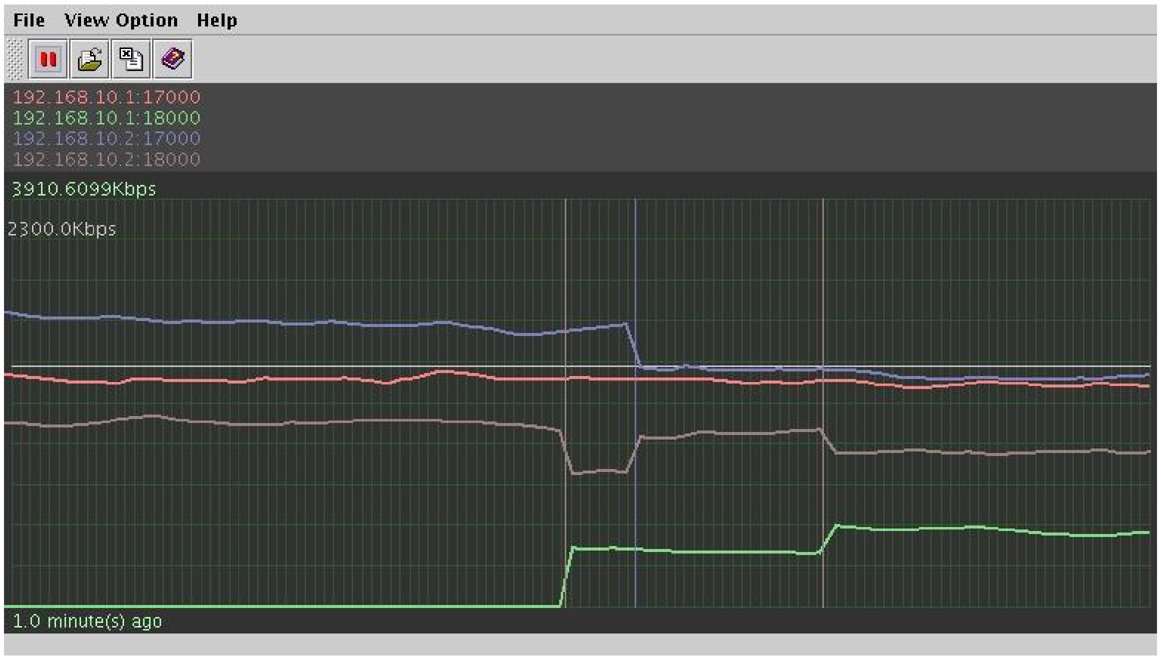
The main limitation of our approach is its focus on handling overload conditions once they occur. When a participant experiences overload, it can use its contracts to decrease its processing cost and pay its partner for the processing. This approach works well for situations when load spikes are unpredictable. However, if a participant plans to experience a large load in the future, it cannot reserve resources at its partners or ask them when they expect their load to be lowest. Other systems investigate the problem of resource reservation [62]. It could be interesting to analyze how to extend our contract-based mechanism to support resource reservation.

7.4.2 Different Types of Cost functions

A second important limitation of our approach is its restriction to systems where participants can model their processing costs with monotonically increasing convex functions. Such functions are appropriate for resources that queue requests as load increases, for example CPU and network bandwidth. Other resources, such as memory, do not follow a convex cost function. As long as the total load is below the memory size, all memory accesses are fast. Once pages need to be swapped to disk, memory access costs increase by a large, constant value. The cost follows a step-function. In this scenario, a fixed price contract can still be used. A participant that is swapping pages to disk can benefit from paying a partner rather than incurring the penalty of going to disk. Bounded-price contracts, however, do not improve load distribution with step cost functions.



(a) As load increases, nodes use their contracts to shed excess load and reach an acceptable allocation.



(b) As a new node joins an overloaded system, nodes use the new capacity to reach an acceptable allocation.

Figure 7-10: Screenshots from an application that monitors the state of a Borealis system. The application polls and displays the load at each node and the recent load movements they performed.

In general, it would be interesting to investigate various types of cost functions and determine the most appropriate types of contracts for these more diverse environments. For instance, we could model the energy utilization of a server as a concave cost functions. For every X clients, a participant needs to purchase one server at a fixed cost (step-function). The server then uses a given amount of power that increases only slightly with the workload (concave function).

7.4.3 System Provisioning

Discussing cost functions raises a second interesting limitation of our approach. We assume that each participant owns a fixed set of resources and that the average load at a participant is fixed. It would be interesting, however, to extend our model to include the cost of purchasing additional resources as the average load at a participant increases, because the participant is gaining clients. A participant could handle small load increases by setting up additional contracts. Eventually, it might be more cost-effective to purchase additional resources and cancel (or simply fail to renew) a fraction of the contracts.

7.4.4 Computing Marginal Costs

Another challenge with our approach is the requirement that participant compute, at least approximately, the marginal costs of processing tasks and convert these values into prices. To setup a contract, a participant must determine its desired load level. It must then compute the per-unit marginal cost at that load level. It must finally negotiate a contract at the given price. These computations can become complex when many resources come into play simultaneously. We can, of course, automate most of these computations but it would be interesting to develop a tool to help participants understand the values they select and the mapping from marginal costs to prices.

7.4.5 Optimizing Contract Prices

In our analysis, we provide heuristics for participants to establish a good set of contracts. Our analysis assumes that participant loads follow some well-known distributions. Additionally, once participants set-up contracts, we assume they keep these contracts, even though our approach does not require this. In future work, it would be interesting to develop a decision support system for contract re-negotiations. Starting from some set of contracts, the system could monitor the local load and the runtime contract utilization. After some time period, the system could indicate whether additional contracts may be profitable, whether reducing the price of a contract could result in greater savings or whether increasing a contract price might help purchase more resources. Such a system would reduce the importance of initial contract negotiations.

7.4.6 Complex Contracts

We focused our analysis on contract prices, assuming that the unit load-movement-duration is some fixed value, d. It would be interesting, however, to study the impact of different contract durations on runtime load movements and on the optimal set of contracts to negotiate. For example, a participant may want to establish contracts with different unit durations to absorb different load spikes.

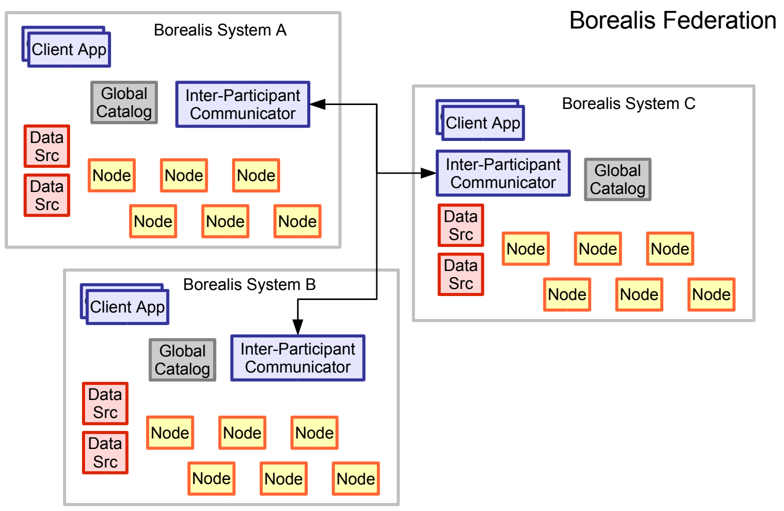


Figure 7-11: Borealis federation.

Similarly, in our analysis and experiments, we assume that all contracts are equivalent and that a participant can use any contract at runtime. Because contracts are set offline, they can include numerous additional clauses: security, availability, or performance guarantees, appropriate course of actions when failures occur, etc. A participant may also prefer to move certain tasks to specific partners. Such clauses and constraints make contracts different from one another and may even affect their values. Contract differences may in turn affect runtime load movements.

7.4.7 Enforcing Contracts

Our approach does not include any specific technique to enforce contracts at runtime. For example, we do not control that a participant provides sufficient resources for a partner's tasks. Such techniques exist for Web Services and data centers (see Section 2.8) but it would be interesting to investigate this problem in the area of stream processing, in particular. Indeed, in an SPE, a participant could sample input and output streams or submit dummy queries for control purposes.

7.4.8 Larger-Scale Federated-Operation Experiments

Finally, we have experimented with BPM implemented at the granularity of individual processing nodes. BPM can also work at the granularity of groups of nodes. We now describe how Borealis and BPM can be deployed in a federated environment.

We envision that in a federated deployment, each participant runs a complete Borealis system with its global catalog, processing nodes, clients, and data sources. Figure 7-11 illustrates a federated deployment. Because participants are autonomous entities, they may not want to give each other open access to their systems. Therefore, rather than communicating directly with each other's global catalogs or processing nodes, participants tunnel all communication through special components called *inter-participant communicators*. Within each domain, these components appear as ordinary clients. For participants to collaborate, their inter-participant communicators must know about each other.

The simplest type of collaboration is for one participant to provide a stream to a partner. Because the global catalog at a participant contains information only about streams that exist locally, for a stream to cross administrative boundaries, it must be defined in both domains. Once defined in both domains, a stream can cross administrative bounds by going through the inter-participant communicators. The inter-participant communicator in the source domain, P_S, subscribes to the stream and forward tuples to the inter-participant communicator in the sink domain, P_D. The latter renames the stream as necessary and takes the role of a data source for the stream within P_D. Both inter-participant communicators must know the mapping between streams in the two domains.

A more complex collaboration occurs when one participant temporarily performs some processing on behalf of a partner. To move load between administrative domains, participants use *remote definitions* in addition to the simpler remote instantiation used within a participant. A remote definition specifies how an operator in one domain maps on to an operator in another domain. Inter-participant communicators must know this mapping for each one of their partners. Remote definitions are performed offline. At runtime, when a path of operators in the query diagram needs to be moved to another domain, all that's required is for the local inter-participant communicator to request that its remote partner instantiate the corresponding operators in its domain. The inter-participant communicators then work together to divert the appropriately named inputs to the new domain. Remote definitions thus allow participants to move load with relatively low overhead compared to full-blown process migration.

Given the above techniques to facilitate collaborations, it would be interesting to experiment with BPM implemented within inter-participant communicators.

7.5 Summary

In this chapter, we presented the evaluation of BPM and its implementation in Borealis. We showed that BPM leads to good load distribution even in heterogeneous environments. In a 995-node network, with approximately 10 contracts per node, the system achieves allocations close to acceptable for all simulated configurations: overload, light load, fixed-price contracts, and heterogeneous contracts. In a uniform system, a price-range equal to only half the network diameter (and half the minimum theoretical value) suffices to ensure acceptable allocation in an underloaded system and a nearly acceptable allocation in an overloaded system. Fixed-price contracts also lead to good load distributions. They do not lead to acceptable allocation in all configurations, but with randomly generated topologies, just five fixed-price contracts per node suffice for a 995-node system to use over 95% of available capacity when overloaded, and, when underloaded, ensure that over 95% of task are properly allocated. In summary, uniform price-range contracts are highly effective at ensuring acceptable allocation. Fixed prices and heterogeneity lead to some wasted resources, making it more difficult for the system to reach acceptable allocations, but BPM easily ensures allocations within a few percent of acceptable. We also showed that convergence to acceptable or nearly acceptable allocation occurs quickly, especially with fixed-price contracts, and that BPM successfully masks a large fraction of load variations; even when load varies significantly (*e.g.*, on average one new tasks and one removed task per time-unit per node), load movements reallocate a number of tasks below 6% of the variation in offered load. Finally, we showed that BPM works well in sample applications processing real data.

Chapter 8

Conclusion

In this dissertation, we addressed the problems of fault-tolerance in a distributed SPE and load management in a distributed federated system. We summarize our contributions and outline topics for future work.

8.1 Fault-Tolerant Stream Processing

In a distributed SPE, especially one where data sources and processing nodes are spread across a wide-area network, several types of failures can occur: processing nodes (software or hardware) can crash, the communication between nodes can be interrupted, and network failures can even cause the system to partition. All these types of failures can disrupt stream processing, affecting the correctness of the output results and preventing the system from producing any results. Previous schemes for fault-tolerant stream processing either do not address network failures [83] or strictly favor consistency over availability, by requiring at least one fully connected copy of the query diagram to exist to continue processing at any time [146]. These schemes do not meet the needs of all stream processing applications because many applications can make significant progress with approximate results, and may even value availability more than consistency. Examples of such applications include network monitoring, network intrusion detection, some financial services applications, and sensor-based environment monitoring. These applications, however, can still benefit from knowing whether the most recent results are accurate or not, and from eventually seeing the correct, final values. We believe that it is important for a failure-handling scheme to meet the above requirements in a manner sufficiently flexible to support many types of applications in a single framework.

Main Contribution: We presented the **Delay, Process, and Correct (DPC)** scheme, a replication-based approach to fault-tolerant stream processing that meets the above goals. DPC supports applications with different required trade-offs between availability and consistency, by enabling these applications to specify the maximum incremental processing latency they can tolerate, and by processing any available input tuple within the required time-bound. Assuming all necessary tuples can be buffered, DPC also guarantees eventual consistency, *i.e.*, all replicas eventually process the same input tuples in the same order and client applications eventually see the complete and correct output streams.

Approach Overview: The main insight of DPC is to let each node (and each replica) manage its own availability and consistency by monitoring its input streams, suspending or delaying the processing of input tuples as appropriate when failures occur, and correcting earlier results after failures heal. DPC uses an enhanced stream data model that distinguishes between stable tuples and tentative tuples, which result from processing partial inputs and may later be corrected. To ensure consistency at runtime, DPC uses a data-serializing operator called SUnion. To regain consistency after failures heal, we investigated techniques based on checkpoint/redo and undo/redo.

Main Results: Through analysis and experiments, we showed that DPC handles both single failures and multiple concurrent failures. DPC ensures eventual consistency while maintaining, when possible, a required level of availability, both in a single-node and a distributed deployment.

We found that reconciling the state of an SPE using checkpoint/redo leads to a faster reconciliation at a lower cost compared with undo/redo. Furthermore, it is possible to avoid the overhead of periodic checkpoints and limit recovery to paths affected by failures, by having operators checkpoint and reinitialize their states in response to the first tentative tuple or undo tuple they process.

When buffers are bounded, DPC is particularly well suited for "convergent-capable" query diagrams (*i.e.*, boxes-and-arrows diagrams of operators interconnected with streams whose state always converges to the same state after processing sufficiently many tuples). With convergent-capable diagrams, DPC ensures that the system eventually converges back to a consistent state and the most recent tentative tuples are corrected, independent of failure duration.

As part of DPC, we investigated techniques to reduce the number of tentative tuples produced by an SPE without breaking a given availability requirement and ensuring eventual consistency. We found that the best strategy is for any SUnion that first detects a failure to block for the maximum incremental processing latency. If the failure persists, SUnions should process new tuples without delay because later delays are not helpful. With this approach, it is possible to mask failures up to the maximum incremental processing latency without introducing inconsistency, independent of the size of the distributed SPE. To maintain a required availability, when failures last a long time, we showed that it is necessary for nodes to process tentative tuples *both* during failures and stabilization, and we showed how DPC enables nodes to do so.

In summary, we showed that it is possible to build a single system for fault-tolerant distributed stream processing that can cope with a variety of system and network failures and support applications with different required trade-offs between availability and consistency.

8.2 Load Management in Federated Systems

In a distributed and federated system, individual participants own and administer subsets of resources. Federated SPEs are one example of such systems, but other federated systems exist, including Grid computing [3, 29, 61, 164], peer-to-peer systems [38, 45, 97, 120, 137, 153, 174], and systems based on Web services [48, 94]. In these environments, individual participants could acquire sufficient resources for peak operation and to handle flash-crowds. Alternatively, participants can collaborate to handle their excess load. The challenge in enabling such collaborations is that autonomous participants aim at maximizing their own utility rather than the utility of the whole system.

Main Contribution: Because previous approaches based on computational economies [3, 29, 154, 175] have failed to gain widespread adoption, while bilateral contracts commonly regulate collaborations between autonomous parties [53, 42, 94, 81, 138, 171, 176, 178], we proposed a contract-based approach to load management in federated environments. Our approach, called **Bounded-Price Mechanism (BPM)**, advocates the use of private pairwise contracts, negotiated offline between participants, to regulate and constrain runtime load movements. Unlike other work on load management with selfish participants, the goal of BPM is not to achieve optimal load balance but simply to ensure that participants operate within a pre-defined capacity.

Approach Overview: In BPM, participants interact with each other on two different timescales. Offline, participants negotiate private pairwise contracts. Each contract specifies a small range of prices that one participant will pay its partner for processing load at runtime. A contract also specifies a unit load-movement duration. At runtime, participants use their contracts to move load to their partners when doing so improves their utility. Partners dynamically negotiate the final price within the contracted range and the amount of load that should move.

Main Results: Through analysis and simulations, we showed that when contracts specify tightly bounded prices for load, and participants use their contracts in a way that maximizes their utility, the load allocation in the system always converges to an acceptable (or a nearly acceptable) allocation. Even in a large and heterogeneous system, a small number of contracts per participant suffices to ensure nearly acceptable allocations for both overloaded and underloaded systems. Interestingly, even though they do not always lead to an acceptable allocation, we find that fixed-price contracts often lead to good load distributions, where most excess load is reallocated and most capacity is used.

We also showed that, most of the time, participants agree on the final price without negotiating, and the bulk of convergence to a final load distribution occurs quickly, especially with fixed-price contracts.

Finally, because prices are almost fixed, BPM can effectively mask a large fraction of load variations as these variations frequently occur away from contract boundaries.

In summary, contracts enable participants to develop or leverage preferential, long-term relationships with each other. Such relationships, coupled with pre-negotiated bounded prices provide predictability and stability at runtime. They also facilitate service customization and price discrimination. Our approach is thus more practical and more lightweight than previous proposals, while still leading to good load distributions. We posit that it should also be more acceptable in practice.

8.3 Software Availability

Prototype implementations of DPC and BPM are part of the Borealis distributed stream processing engine. BPM is available on line as part of borealis-0.2. DPC will be available with the next Borealis release. All software is available on our project website: http://nms.lcs.mit.edu/projects/borealis/.

8.4 Open Issues

Our study of fault-tolerance with DPC and load management with BPM opens several areas of future work. In Sections 5.8 and 7.4, we presented several specific research directions for fault-tolerance and load management, respectively. We now summarize only a few of the most important open problems.

One of the main open issues with DPC is its lack of precision information. Currently, DPC only labels tuples as either tentative or stable. It would be interesting to enhance operators with the capability to read precision information off their input tuples and compute that same information for their output tuples, especially when some of their input streams are missing. Precision information may also help determine when failures affect only subsets of output results. Rather than labeling all outputs as tentative, some outputs may have a high precision while others may be less accurate. Second, it may also be interesting to enhance the system with support for integrity constraints on streams. Because different failures affect different parts of the system, it may sometimes be meaningless to correlate or merge certain tentative streams. It would be interesting to enable applications to define integrity constraints describing the conditions when streams still carry useful information and when they can be combined. The system could use these constraints to ensure the quality of the output data, but also, perhaps, as part of the upstream neighbor switching algorithm. A third interesting problem would be to extend DPC to support even more types of failures, such as Byzantine failures where a possibly malicious node produces erroneous results, or the simultaneous crash failure of all replicas of one or more processing nodes.

One of the main open issues with BPM is its focus on reacting to overload as it occurs rather than enabling participants to reserve resources ahead of time. Moving existing load makes it easier for participants to maximize resource utilization. This model, however, is not suitable to all types of applications. Some applications, such as running a large experiment on a shared distributed computing platform, would waste large amounts of resources if clients had to start these applications in order to check if sufficient resources were available for them. Extending contracts to enable some form of resource reservation would extend the scope of BPM. A second important problem is to explore the possibility for participants to use different types of cost functions that are not necessarily monotonically increasing and convex. For some types of resources step functions or concave functions may better model processing costs. Furthermore, as the overall offered load varies with time, participants may also acquire or dispose of resources causing their cost functions to change with time. It would be interesting to study how the contract-based approach could be extended for these different types of cost models.

Finally, in this dissertation, we investigated the problems of fault-tolerance and load management in isolation. These two problems, however, are tightly connected. Indeed, after a failure heals, a participant must reprocess earlier data to ensure eventual consistency. Extra state-reconciliation tasks can significantly increase a participant's load, and it may become profitable to offload a subset of these tasks to a trusted partner. Because processing restarts from a checkpoint, moving a state-reconciliation task may even be less costly than moving a running task. An interesting problem is how to integrate the existence of these *transient* replicas into the DPC protocol and how to extend BPM's contracts to support these new types of tasks with somewhat flexible resource requirements but strict availability requirements. In general, in a federated system, participants can use each other not only to handle excess load but also for fault-tolerance and replication. It would be interesting to investigate a structure of contracts that would promote these types of collaborations.

Appendix A

Computing Output Stream States

We present an algorithm to compute individual output stream states from input stream states in a query diagram. The Consistency Manager should run this algorithm. The states of input streams, stored in InState, come from monitoring all replicas of all upstream neighbors (Algorithm from Figure 4-10).

Figure A-1 shows the detailed algorithm. By default, all output streams are in the STABLE state. An output stream is in UP_FAILURE state as soon as one of its contributing input streams experiences a failure. The failure is total (FAILURE state) rather than partial (UP_FAILURE state) if the output stream is (1) downstream from a blocking operator with a missing input stream or (2) it is downstream from a non-blocking operator with all inputs missing. Finally, if a node is in the STABILIZATION state, all output streams that were previously experiencing a failure are in STABILIZATION state as well. With some state reconciliation techniques, all output streams are in the STABILIZATION state during state reconciliation.

The algorithm uses the states of streams produced by all replicas not just the current upstream neighbor. It assumes the node will switch between replicas as per algorithm in Table 4.3.

```
PROCEDURE COMPUTESTATE:
Inputs:
  QueryDiagram: description of the query diagram fragment running at the node.
  InputStreams: set of all input streams to the node.
  OutputStreams: set of all output streams produced by the node.
  Replicas: upstream neighbors and their replicas.
    $\forall s \in \texttt{InputStreams}, \texttt{Replicas}[s] = \{r_1, r_2, ...r_n\} \mid \forall i \in [1, n],\ r_i \in \texttt{Nodes}$ produces $s$.
  NodeState: state of the node. $\texttt{NodeState} \in \{\texttt{STABLE}, \texttt{UP\_FAILURE}, \texttt{STABILIZATION}\}$.
  InState: states of streams produced by different nodes.
    $\forall s \in \texttt{InputStreams}, \forall r \in \texttt{Replicas}[s]$,
    $\texttt{InState}[r][s] = x \mid x \in \texttt{States}$ is the state of stream $s$ produced by $r$.
Both Input and Output:
  OutState: states of output streams of this node
    $\forall o \in \texttt{OutputStreams}, \texttt{OutState}[o] = x \mid x \in \texttt{States}$ is the state of $o$.

  // First, identify streams in STABILIZATION state
01. foreach $o$ in OutputStreams
02.   if $\texttt{NodeState} = \texttt{STABILIZATION}$ and $\texttt{OutState}[o] \in \{\texttt{UP\_FAILURE}, \texttt{FAILURE}, \texttt{STABILIZATION}\}$
03.     $\texttt{OutState}[o] \leftarrow \texttt{STABILIZATION}$
04.   else // Reset the current state of the output stream before recomputing it
05.     $\texttt{OutState}[o] \leftarrow \texttt{STABLE}$

  // Second, identify streams affected by upstream failures
06. foreach $s$ in InputStreams
07.   if $(\forall r \in \texttt{Replicas}[s],\ \ \texttt{InState}[r][s] \neq \texttt{STABLE})$
08.     if there exists a path between $s$ and an output stream $o$ in QueryDiagram

        // First condition that may block the output stream: a blocking operator with
        // at least one missing input
09.       if path goes through a blocking operator and
            $(\forall r \in \texttt{Replicas}[s],\ \ \texttt{InState}[r][s] \notin \{\texttt{STABLE}, \texttt{UP\_FAILURE})\}$
10.         $\texttt{OutState}[o] \leftarrow \texttt{FAILURE}$

        // Second condition that may block the output stream: an operator with
        // all its inputs missing
11.       else if  path goes through operator downstream from a set $S$ of inputs such that
          $(S \subseteq \texttt{InputStreams})$ and $(\forall s' \in S, \forall r \in \texttt{Replicas}[s'], \texttt{InState}[r][s'] \notin \{\texttt{STABLE}, \texttt{UP\_FAILURE}\}$
12.         $\texttt{OutState}[o] \leftarrow \texttt{FAILURE}$

        // Otherwise, the output can produce tentative tuples
13.       else
14.         $\texttt{OutState}[o] \leftarrow \texttt{UP\_FAILURE}$
```

Figure A-1: Algorithm for computing the states of output streams. Nodes denotes the set of all processing nodes in the system. States = {STABLE, UP_FAILURE, FAILURE, STABILIZATION}. The states of output streams are stored in OutState. All data structures are local to each node.

Appendix B

Undo/Redo

To avoid the CPU overhead of checkpointing and to recover at a finer granularity by rolling back only the state on paths affected by the failure, another approach for an SPE to reconcile its state is to undo the processing of tentative tuples and redo that of their stable counterparts. In this appendix, we present the state-reconciliation technique based on undo and redo.

B.1 State Reconciliation

To support undo/redo reconciliation, all operators should implement an "undo" method, where they remove a tuple from their state and, if necessary, bring back into the state some previously evicted tuples. Supporting undo in operators may not be straightforward. For example, suppose an operator computes an average over a window of size 10 with advance 10, and receives the following input:

(**2**, 10.5) (**5**, 11.2) (**8**, 13.4) | (**11**, 9.7) (**14**, 9.2) (**17**, 10.8) | (**21**, 10.4), (**24**, 9.9), ...

Assuming the first attribute specifies the window, the aggregate will compute an average over windows [0,10), [10,20), [20,30), etc. We use the symbol "|" to indicate window boundaries on the input stream. After processing tuple (17, 10.8), the state of the aggregate holds tuples (11, 9.7), (14, 9.2), and (17, 10.8)— all tuples in the current window [10, 20). Upon receiving tuple (21, 10.4), the aggregate closes window [10, 20), outputs the result tuple (10, 9.9), and opens window [20, 30). The new state contains only tuple (21, 10.4). To undo tuple (21, 10.4), the aggregate must not only undo the output tuple (10, 9.9), but it must also reopen window [10,20), and bring back tuples (11, 9.7), (14, 9.2), and (17, 10.8). Supporting an "undo" method may thus be complex to implement and would require significant modifications to all operators.

Instead, we propose that operators undo by *rebuilding the state* that existed right before they processed the tuples that must now be undone. For example, to undo tuple (21, 10.4), the aggregate should clear its state and reprocess all tuples since (11, 9.7). To determine how far back in history to restart processing from, operators maintain a *set of stream markers* for each input tuple. The stream markers for a tuple p in operator u are the identifiers of the oldest tuples on each input stream that contribute to the operator's state *before* u processes p. In the above example, the stream marker for tuple (21, 10.4) is (11, 9.7), the oldest tuple in the operator's state before the operator processes (21, 10.4). To undo all tuples since (and

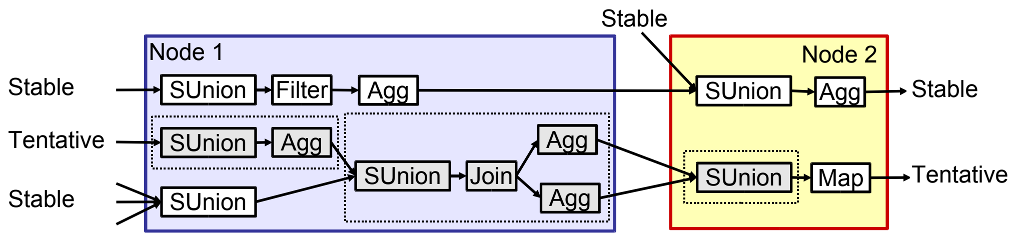


Figure B-1: Locations where tuples are buffered with undo/redo. All stateful operators (and some stateless operators) with at least one tentative input stream must buffer tuples that accumulate during a failure. Operators that buffer tuples are outlined.

including) p, u looks up the stream markers for p, scans its input buffers until it finds the marked tuples, and reprocesses its input buffers since then, stopping right before processing p again. In the example, to undo tuple (21, 10.4), the operator reprocesses all tuples since (11, 9.7). A stream marker is typically the beginning of a window of computation. Stream markers do not hold any state. They are pointers to some location in the input buffer. To produce the appropriate undo tuple, operators must also store, with each set of stream markers, the last tuple they produced

In the worst case, determining the stream markers may require a linear scan of all tuples in the operator's state. To reduce the runtime overhead, rather than compute stream markers for every tuple, operators can set stream markers periodically. Periodic stream markers increase reconciliation time, however, as reprocessing restarts from imprecise markers. Operators that keep their state in aggregate form must explicitly remember the first tuple on each input stream that begins the current aggregate computation(s).

Undo/redo also requires a new tuple type. To trigger undo/redo-based state reconciliation, the Consistency Manager injects a tuple of type UNDO_REDO_START on one input stream of each affected input SUnion operator.

B.2 Buffer Management

To support undo/redo, all stateful operators must store the input tuples they receive in an undo buffer. Since only tentative tuples can be undone, as long as an operator processes stable tuples it can truncate its undo buffer, keeping only enough tuples to rebuild its current state. During failures all stateful operators affected by the failure must accumulate input tuples in an undo buffer. As discussed in Section 5.5, many stateless operators (*e.g.*, Filter) also need an undo buffer because they must remember when they produced each output tuple in order to produce the appropriate UNDO tuple. Only operators that produce exactly one output tuple for each input tuple (*e.g.*, Map) need not keep an undo buffer, because they can process their input UNDO tuple to produce an output UNDO tuple. Figure B-1 illustrates which operators buffer data with undo/redo.

Forcing nodes to correct *all* tentative tuples enables great savings in buffer space. Indeed, if all tentative tuples are always corrected, only SUnions with at least one tentative input stream buffer tuples *on their stable inputs*. Stateful operators only buffer those tuples that will enable them to rebuild their pre-failure states. (Figure B-2).

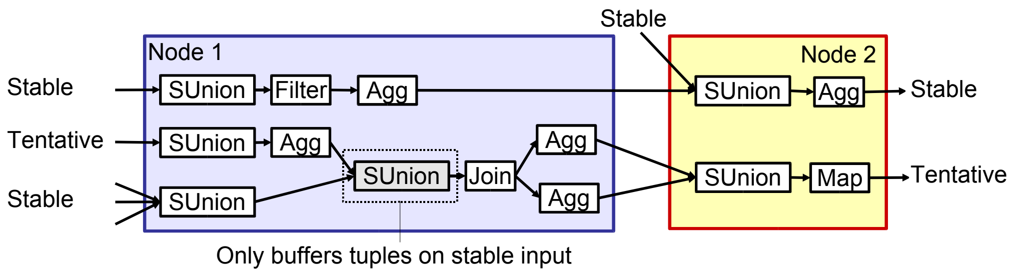


Figure B-2: Locations where tuples are buffered with undo/redo when all tentative tuples are always corrected. SUnions with at least one tentative input stream must buffer, during failures, tuples on their *stable inputs*. Other stateful operators keep just enough tuples to rebuild their pre-failure state (not represented). Operators that buffer tuples are outlined.

B.3 Failures during Recovery

We now discuss how undo/redo handles failures during recovery. We show that undo/redo can also meet the following property:

Property 6 Handle failures during failures and recovery*: Stabilization never causes stable output tuples to be dropped, duplicated, or undone.*

Condition*: Nodes buffer tuples and remove them from buffers as described in Section 4.9. All buffers are sufficiently large (respectively failures are sufficiently short) to ensure no tuples are dropped due to lack of space.*

First, we examine a simple failure scenario. When a failure occurs and tentative tuples propagate through a query diagram, as soon as an operator receives a tentative tuple, it starts labeling its output tuples as tentative. Therefore, undoing tentative tuples during reconciliation can never cause a stable output to be undone.

Second, we examine the case of undo tuples propagating simultaneously on multiple steams. If multiple streams failed, and multiple undo tuples propagate through the query diagram during stabilization, an operator with multiple inputs could receive the undo tuples in any order. However, operators with multiple inputs are always preceded by SUnions. SUnions must receive corrections on all their input streams before producing an undo and corrections in turn. The downstream operator thus receives a single undo followed by corrections.

Third, we examine the case of a stream that fails, recovers, and fails again. Because SUnions produce undo tuples followed by the stable versions of tuples processed during the first failure. Any tentative input tuples caused by a new upstream failure will accumulate in SUnions after the stable corrections. Therefore, the new tentative tuples will be pushed into the query diagram and processed by operators *after the undo and stable tuples.* Thus any new failure will follow the reconciliation, without affecting it.

Finally, we examine the case where a failure occurs during reconciliation on a previously stable input. While an undo tuple propagates on a stream, if a different input stream becomes tentative, and both streams merge at an SUnion (streams always merge at SUnions), the SUnion could see the new tentative tuples before the undo tuple. In this case, when the operator finally processes the undo tuple, it rebuilds the state it had *before the first failure* and processes all tuples that it processed during that failure *before* going back to processing the new tentative tuples. Basically, the undo and corrections will apply to *older buckets* than the new tentative tuples. The SUnions thus produces an undo tuple followed by stable tuples that correct the first failure, followed by the tentative tuples from the new failure. Once again, the new failure appears to occur after stabilization.

If a new failure occurs before the node had time to catch up and produce a REC_DONE tuple, SOutput forces a REC_DONE tuple between the last stable and first tentative tuples that it sees.

Hence, with undo/redo recovery, failures can occur during failures or reconciliation and the system still guarantees that stable tuples are not undone, dropped, nor duplicated.

Bibliography

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the Borealis stream processing engine. In *Proc. of the Second Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2005.

[2] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), September 2003.

[3] David Abramson, Rajkumar Buuya, and Jonathan Giddy. A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems*, 18(8), October 2002.

[4] Yanif Ahmad and Uğur Çetintemel. Network-aware query processing for stream-based applications. In *Proc. of the 30th International Conference on Very Large Data Bases (VLDB)*, September 2004.

[5] Gustavo Alonso, Roger Günthör, Mohan Kamath, Divyakant Agrawal, Amr El Abbadi, and C. Mohan. Exotica/FMDC: Handling disconnected clients in a workflow management system. In *Proc. of the 3rd International Conference on Cooperative Information Systems*, May 1995.

[6] Gustavo Alonso and C. Mohan. WFMS: The next generation of distributed processing tools. In Sushil Jajodia and Larry Kerschberg, editors, *Advanced Transaction Models and Architectures*. Kluwer, 1997.

[7] Gustavo Alonso, C. Mohan, Roger Günthör, Divyakant Agrawal, Amr El Abbadi, and Mohan Kamath. Exotica/FMQM: A persistent message-based architecture for distributed workflow management. In *Proc. of IFIP WG8.1 Working Conference on Information Systems for Decentralized Organizations*, August 1995.

[8] Gustavo Alonso, Berthold Reinwald, and C. Mohan. Distributed data management in workflow environments. In *Proc. of the 7th ACM RIDE*, April 1997.

[9] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. STREAM: The stanford data stream management system. To appear in a book on data stream management edited by Garofalakis, Gehrke, and Rastogi.

[10] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. Technical Report 2003-67, Stanford University, October 2003.

[11] Arvind Arasu, Shivnath Babu, and Jennifer Widom. An abstract semantics and concrete language for continuous queries over streams and relations. *VLDB Journal*, (to appear), 2005.

[12] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *Proc. of the 30th International Conference on Very Large Data Bases (VLDB)*, September 2004.

[13] Arvind Arasu and Gurmeet Manku. Approximate counts and quantiles over sliding windows. In *Proc. of the 23rd ACM Symposium on Principles of Database Systems (PODS)*, June 2004.

[14] The Aurora project. `http://www.cs.brown.edu/research/aurora/`.

[15] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data*, May 2000.

[16] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev Motwani. Chain : Operator scheduling for memory minimization in data stream systems. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*, June 2003.

[17] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proc. of the 21st ACM Symposium on Principles of Database Systems (PODS)*, June 2002.

[18] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *Proc. of the 20th International Conference on Data Engineering (ICDE)*, March 2004.

[19] Brian Babcock and Chris Olston. Distributed top-k monitoring. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*, June 2003.

[20] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri. Generic support for distributed applications. *IEEE Computer*, 33(3):68–76, March 2000.

[21] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbets, and Stan Zdonik. Retrospective on Aurora. *VLDB Journal*, 13(4), December 2004.

[22] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *Proc. of the 2005 ACM SIGMOD International Conference on Management of Data*, June 2005.

[23] Magdalena Balazinska, Hari Balakrishnan, and Mike Stonebraker. Contract-based load management in federated distributed systems. In *Proc. of the First Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[24] Guruduth Banavar, Tushar Deepak Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proc. of 19th International Conference on Distributed Computing Systems (ICDCS'99)*, June 1999.

[25] Joel Barlett, Jim Gray, and Bob Horst. Fault tolerance in Tandem computer systems. Technical Report 86.2, Tandem Computers, March 1986.

[26] Preeti Bhoj, Sharad Singhal, and Sailesh Chutani. SLA management in federated environments. Technical Report HPL-98-203, Hewlett-Packard Company, 1998.

[27] The Borealis project. `http://nms.lcs.mit.edu/projects/borealis/`.

[28] Eric A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[29] Rajkumar Buuya, Heinz Stockinger, Jonathan Giddy, and David Abramson. Economic models for management of resources in peer-to-peer and grid computing. In *Proc. of SPIE International Symposium on The Convergence of Information Technologies and Communications (ITCom 2001)*, August 2001.

[30] Don Carney, Uğur Çetintemel, Alexander Rasin, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator scheduling in a data stream manager. In *Proc. of the 29th International Conference on Very Large Data Bases (VLDB)*, September 2003.

[31] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. In *Proc. of the Nineteenth ACM Symposium on Principles of Distributed Computing (PODC 2000)*, July 2000.

[32] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, February 1999.

[33] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2003.

[34] Sirish Chandrasekaran and Michael J. Franklin. Streaming queries over streaming data. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB)*, August 2002.

[35] Sirish Chandrasekaran and Michael J. Franklin. Remembrance of streams past: Overload-sensitive management of archived streams. In *Proc. of the 30th International Conference on Very Large Data Bases (VLDB)*, September 2004.

[36] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data*, May 2000.

[37] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2003.

[38] Brent Chun, Yun Fu, and Amin Vahdat. Bootstrapping a distributed computational economy with peer-to-peer bartering. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[39] Brent Nee Chun. *Market-Based Cluster Resource Management.* PhD thesis, University of California at Berkeley, 2001.

[40] Enzo Cialini and John Macdonald. Creating hot snapshots and standby databases with IBM DB2 Universal Database$^{(TM)}$ V7.2 and EMC TimeFinder$^{(TM)}$. DB2 Information Management White Papers, September 2001.

[41] Graham Cormode, Theodore Johnson, Flip Korn, S. Muthukrishnan, Oliver Spatscheck, and Divesh Srivastava. Holistic UDAFs at streaming speeds. In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data*, June 2004.

[42] IBM Corporation. Web service level agreements (WSLA) project. `http://www.research.ibm.com/wsla/`, April 2003.

[43] Chuck Cranor, Theodore Johnson, Vladislav Shkapenyuk, and Oliver Spatscheck. Gigascope: A stream database for network applications. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*, June 2003.

[44] Chuck Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. The Gigascope stream database. *IEEE Data Engineering Bulletin*, 26(1), March 2003.

[45] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[46] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*, June 2003.

[47] Amol Deshpande and Joseph M. Hellerstein. Lifting the burden of history from adaptive query processing. In *Proc. of the 30th International Conference on Very Large Data Bases (VLDB)*, September 2004.

[48] W3C Architecture Domain. Web services activity. `http://www.w3.org/2002/ws/`.

[49] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.

[50] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Survey*, 34(3):375–408, 2002.

[51] Cristian Estan, Stefan Savage, and George Varghese. Automatically inferring patterns of resource consumption in network traffic. In *Proc. of the ACM SIGCOMM Conference*, August 2003.

[52] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Serveys*, 35(2), June 2003.

[53] George Fankhauser, David Schweikert, and Bernhard Plattner. Service level agreement trading for the differentiated services architecture. Technical Report 59, Swiss Federal Institute of Technology (ETH) Zurich, January 2000.

[54] Nick Feamster, David G. Andersen, Hari Balakrishnan, and Frans Kaashoek. Measuring the Effects of Internet Path Faults on Reactive Routing. In *ACM Sigmetrics - Performance 2003*, June 2003.

[55] Joan Feigenbaum, Christos Papadimitriou, Rahul Sami, and Scott Shenker. A BGP-based mechanism for lowest-cost routing. In *Proc. of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC 2002)*, July 2002.

[56] Joan Feigenbaum, Christos Papadimitriou, and Scott Shenker. Sharing the cost of multicast transmissions. *Journal of Computer and System Sciences*, 63:21–41, 2001.

[57] Joan Feigenbaum, Christos Papadimitriou, and Scott Shenker. Distributed algorithmic mechanism design: Recent results and future directions. In *Proc. of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, September 2002.

[58] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. In *Proc. of the Fifteenth ACM Symposium on Principles of Distributed Computing (PODC 1996)*, May 1996.

[59] Donald Ferguson, Christos Nikolaou, J. Sairamesh, and Yechiam Yemini. Economic models for allocating resources in computer systems. In S. H. Clearwater, editor, *Market based Control of Distributed Systems*. World Scientist, January 1996.

[60] Klaus Finkenzeller. *RFID Handbook—radio-frequency identification fundamentals and applications*. John Wiley & Sons, 1999.

[61] Ian T. Foster and Carl Kesselman. Computational grids. In *Proc. of the Vector and Parallel Processing (VECPAR)*, June 2001.

[62] Yun Fu, Jeffery Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. SHARP: An architecture for secure resource peering. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[63] Yun Fu and Amin Vahdat. Service level agreement based distributed resource allocation for streaming hosting systems. In *Proc. of 7th Int. Workshop on Web Content Caching and Distribution*, August 2002.

[64] Robert Gallager. A minimum delay routing algorithm using distributed computation. *IEEE Transactions on Communication*, COM-25(1), January 1977.

[65] Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841 – 860, October 1985.

[66] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, December 1992.

[67] David K. Gifford. Weighted voting for replicated data. In *Proc. of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, December 1979.

[68] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. *ACM SIGACT News*, 33(2), 2002.

[69] Lukasz Golab and M. Tamer Özsu. Update-pattern-aware modeling and processing of continuous queries. In *Proc. of the 2005 ACM SIGMOD International Conference on Management of Data*, June 2005.

[70] Jim Gray. Why do computers stop and what can be done about it? Technical Report 85.7, Tandem Computers, June 1985.

[71] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*, June 1996.

[72] Jim Gray and Andreas Reuters. *Transaction processing: concepts and techniques.* Morgan Kaufmann, 1993.

[73] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2), June 1995.

[74] Eric N. Hanson. Rule condition testing and action execution in Ariel. In *Proc. of the 1992 ACM SIGMOD International Conference on Management of Data*, June 1992.

[75] Eric N. Hanson, Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha, Sashi Parthasarathy, J. B. Park, and Albert Vernon. Scalable trigger processing. In *Proc. of the 15th International Conference on Data Engineering (ICDE)*, March 1999.

[76] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. Java Message Service. Sun Microsystems Inc., April 2002.

[77] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The anatomy of a context-aware application. In *Proc. of the 5th ACM Annual International Conference on Mobile Computing and Networking (MOBICOM)*, August 1999.

[78] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proc. of the 1997 ACM SIGMOD International Conference on Management of Data*, June 1997.

[79] B. Hoffmann-Wellenhof, H. Lichtenegger, and J. Collins. *Global Positioning System: Theory and Practice, Fourth Edition.* Springer-Verlag, 1997.

[80] David Hollingsworth. The workflow reference model. Technical Report WFMC-TC-1003, The Workflow Management Coallition, January 1995.

[81] Rackspace Managed Hosting. Managed SLA. `http://www.rackspace.com/solutions/managed_sla.php`, 2005.

[82] Meichun Hsu. Special issue on workflow systems. *IEEE Data Engineering Bulletin*, 18(1), March 1995.

[83] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Uğur Çetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. In *Proc. of the 21st International Conference on Data Engineering (ICDE)*, April 2005.

[84] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Y. Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *Proc. of the 1999 ACM SIGMOD International Conference on Management of Data*, June 1999.

[85] Zachary G. Ives, Alon Y. Levy, Daniel S. Weld, Daniela Florescu, and Marc Friedman. Adaptive query processing for Internet applications. *IEEE Data Engineering Bulletin*, 23(2), June 2000.

[86] Matthew Jackson. Mechanism theory. *Forthcoming in Encyclopedia of Life Support Stystems*, 2000.

[87] H. V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz. View maintenance issues for the chronicle data model. In *Proc. of the 14th ACM Symposium on Principles of Database Systems (PODS)*, May 1995.

[88] Yuhui Jin and Robert E. Strom. Relational subscription middleware for Internet-scale publish-subscribe. In *Proc of the 2nd International Workshop on Distributed Event-Based Systems (DEBS '03)*, June 2003.

[89] Theodore Johnson, S. Muthukrishnan, and Irina Rozenbaum. Sampling algorithms in a stream operator. In *Proc. of the 2005 ACM SIGMOD International Conference on Management of Data*, June 2005.

[90] Mohan Kamath, Gustavo Alonso, Roger Guenthor, and C. Mohan. Providing high availability in very large workflow management systems. In *Proc. of the 5th International Conference on Extending Database Technology*, March 1996.

[91] Ben Kao and Hector Garcia-Molina. An overview of real-time database systems. In *Advances in real-time systems*. Prentice-Hall, 1995.

[92] Sachin Katti, Balachander Krishnamurthy, and Dina Katabi. Collaborating against common enemies. In *Proc. of Internet Measurement Conference (IMC)*, October 2005.

[93] Leonard Kawell, Steven Beckhardt, Timothy Halvorsen, Raymond Ozzie, and Irene Greif. Replicated document management in a group communication system. In *Proc. of the 1988 ACM conference on computer-supported cooperative work (CSCW)*, September 1988.

[94] Alexander Keller and Heiko Ludwig. The WSLA framework: Specifying and monitoring service level agreements for Web services. Technical Report RC22456, IBM Corporation, May 2002.

[95] Krishna G. Kulkarni, Nelson Mendonça Mattos, and Roberta Cochrane. Active database features in SQL-3. In Norman W. Paton, editor, *Active Rules in Database Systems*. Springer-Verlag, 1999.

[96] James F. Kurose. A microeconomic approach to optimal resource allocation in distributed computer systems. *IEEE Transactions on Computers*, 38(5):705–717, 1989.

[97] Kevin Lai, Michal Feldman, Ion Stoica, and John Chuang. Incentives for cooperation in peer-to-peer networks. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[98] Kwok-Wa Lam, Sang H. Son, Sheung-Lun Hung, and Zhiwei Wang. Scheduling transactions with stringent real-time constraints. *Information Systems*, 25(6):431–452, September 2000.

[99] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[100] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Query languages and data models for database sequences and data streams. In *Proc. of the 30th International Conference on Very Large Data Bases (VLDB)*, September 2004.

[101] William Lehr and Lee W. McKnight. Show me the money: Contracts and agents in service level agreement markets. `http://itc.mit.edu/itel/docs/2002/show_me_the_money.pdf`, 2002.

[102] Alberto Lerner and Dennis Shasha. AQuery: query language for ordered data, optimization techniques, and experiments. In *Proc. of the 17th International Conference on Very Large Data Bases (VLDB)*, September 2003.

[103] Philip M. Lewis, Arthur Bernstein, and Michael Kifer. *Databases and transaction processing*. Addison-Wesley, 2001.

[104] David Lomet and Mark Tuttle. A theory of redo recovery. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*, June 2003.

[105] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[106] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The design of an acquisitional query processor for sensor networks. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*, June 2003.

[107] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122 – 173, 2005.

[108] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proc. of the 2002 ACM SIGMOD International Conference on Management of Data*, June 2002.

[109] Thomas W. Malone, Richard E. Fikes, Kenneth R. Grant, and Michael T. Howard. Enterprise: A market-like task scheduler for distributed computing environments. *The Ecology of Computation*, 1988.

[110] Amit Manjhi, Suman Nath, and Phillip Gibbons. Tributaries and deltas: Efficient and robust aggregation in sensor network streams. In *Proc. of the 2005 ACM SIGMOD International Conference on Management of Data*, June 2005.

[111] The Medusa project. `http://nms.lcs.mit.edu/projects/medusa/`.

[112] Jim Melton. *Advanced SQL:1999 – Understanding Object-Relational and Other Advanced Features.* Morgan Kaufmann Publishers, September 2002.

[113] Mesquite Software, Inc. CSIM 18 user guide. `http://www.mesquite.com`.

[114] Microsoft Corporation. Inside microsoft SQL server 2000. Microsoft TechNet, 2003.

[115] Mark S. Miller and K. Eric Drexler. Markets and computation: Agoric open systems. In Bernardo Huberman, editor, *The Ecology of Computation.* Science & Technology, 1988.

[116] C. Mohan, Gustavo Alonso, Roger Günthör, Mohan Kamath, and Berthold Reinwald. An overview of the Exotica research project on workflow management systems. In *Proc. of the 6th International Workshop on High Performance Transaction Systems*, September 1995.

[117] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2003.

[118] Chaki Ng, David C Parkes, and Margo Seltzer. Strategyproof computing: Systems infrastructures for self-interested parties. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[119] Chaki Ng, David C. Parkes, and Margo Seltzer. Virtual worlds: Fast and strategyproof auctions for dynamic resource allocation. `http://www.eecs.harvard.edu/~parkes/pubs/virtual.pdf`, June 2003.

[120] Tsuen-Wan "Johnny" Ngan, Dan S. Wallach, and Peter Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.

[121] Noam Nisan and Amir Ronen. Computationally feasible VCG mechanisms. In *Second ACM Conference on Electronic Commerce (EC00)*, October 2000.

[122] Noam Nisan and Amir Ronen. Algorithmic mechanism design. *Games and Economic Behavior*, 35, 2001.

[123] Chris Olston, Jing Jiang, and Jennifer Widom. Adaptive filters for continuous queries over distributed data streams. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*, June 2003.

[124] Christopher Olston. *Approximate Replication.* PhD thesis, Stanford University, 2003.

[125] Oracle Corporation. Oracle9i database release 2 product family. An Oracle white paper, June 2003.

[126] Gultekin Ozsoyoglu and Richard T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.

[127] David Parkes. *Iterative Combinatorial Auctions: Achieving Economic and Computational Efficiency (Chapter 2).* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 2001.

[128] David Parkes. Price-based information certificates for minimal-revelation combinatorial auctions. *Agent Mediated Electronic Commerce IV*, LNAI 2531:103–122, 2002.

[129] David C. Parkes and Jeffrey Shneidman. Distributed implementations of Vickrey-Clarke-Groves mechanisms. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multi Agent System*, 2004.

[130] Norman W. Paton and Oscar Diaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, 1999.

[131] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. of the First Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[132] Peter Pietzuch, Jeffrey Shneidman, Mema Roussopoulos, Margo Seltzer, and Matt Welsh. Path optimization in stream-based overlay networks. Technical Report TR-26-04, Harvard University, 2004.

[133] Nissanka B. Priyantha, Allen K. L. Miu, Hari Balakrishnan, and Seth Teller. The Cricket Compass for Context-Aware Mobile Applications. In *Proc. of the 7th ACM Annual International Conference on Mobile Computing and Networking (MOBICOM)*, July 2001.

[134] Vijayshankar Raman and Joseph M. Hellerstein. Partial results for online query processing. In *Proc. of the 2002 ACM SIGMOD International Conference on Management of Data*, June 2002.

[135] Ashish Ray. Oracle data guard: Ensuring disaster recovery for the enterprise. An Oracle white paper, March 2002.

[136] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proc. of the 13th Conference on Systems Administration (LISA-99)*, November 1999.

[137] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[138] Akhil Sahai, Anna Durante, and Vijay Machiraju. Towards automated SLA management for Web services. Technical Report HPL-2001-310R1, Hewlett-Packard Company, July 2001.

[139] Akhil Sahai, Sven Graupner, Vijay Machiraju, and Add van Moorsel. Specifying and monitoring guarantees in commercial Grids through SLA. Technical Report HPL-2002-324R1, Hewlett-Packard Company, November 2002.

[140] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[141] Tuomas Sandholm. An implementation of the contract net protocol based on marginal cost calculations. In *Proc. of the 12th International Workshop on Distributed Artificial Intelligence*, pages 295–308, 1993.

[142] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Serveys*, 22(4), December 1990.

[143] Fred B. Schneider. What good are models and what models are good? In *Distributed Systems*, pages 17–26. ACM Press/Addison-Wesley Publishing Co, 2nd edition, 1993.

[144] Ulf Schreier, Hamid Pirahesh, and Rakesh Agrawal an C. Mohan. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proc. of the 17th International Conference on Very Large Data Bases (VLDB)*, September 1991.

[145] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The design and implementation of a sequence database system. In *Proc. of the 22nd International Conference on Very Large Data Bases (VLDB)*, September 1996.

[146] Mehul Shah, Joseph Hellerstein, and Eric Brewer. Highly-available, fault-tolerant, parallel dataflows. In *Proc. of the 2004 ACM SIGMOD International Conference on Management of Data*, June 2004.

[147] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. of the 19th International Conference on Data Engineering (ICDE)*, March 2003.

[148] Tetsuya Shirai, John Barber, Mohan Saboji, Indran Naick, and Bill Wilkins. *DB2 Universal Database in Application Development Environments*. Prentice Hall, 2000.

[149] D. Skeen. Vitria's publish-subscribe architecture: Publish-subscribe overview. `http://www.vitria.com`, 1998.

[150] Richard Snodgrass and Ilsoo Ahn. A taxonomy of time in databases. In *Proc. of the 1985 ACM SIGMOD International Conference on Management of Data*, May 1985.

[151] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *Proc. of the 23rd ACM Symposium on Principles of Database Systems (PODS)*, June 2004.

[152] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins. In *Proc. of the 30th International Conference on Very Large Data Bases (VLDB)*, September 2004.

[153] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of the ACM SIGCOMM Conference*, August 2001.

[154] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: a wide-area distributed database system. *VLDB Journal*, 5(1), January 1996.

[155] Michael Stonebraker and Greg Kemnitz. The POSTGRES next generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

[156] Robert E. Strom. Fault-tolerance in the SMILE stateful publish-subscribe system. In *Proc of the 3rd International Workshop on Distributed Event-Based Systems (DEBS '04)*, May 2004.

[157] Mark Sullivan. Tribeca: A stream database manager for network traffic analysis. In *Proc. of the 22nd International Conference on Very Large Data Bases (VLDB)*, September 1996.

[158] Mark Sullivan and Andrew Heybey. Tribeca: A system for managing large databases of network traffic. In *Proc. of the 1998 USENIX Annual Technical Conference*, June 1998.

[159] Kian-Lee Tan, Cheng Hian Goh, and Beng Chin Ooi. Online feedback for nested aggregate queries with multi-threading. In *Proc. of the 25nd International Conference on Very Large Data Bases (VLDB)*, September 1999.

[160] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proc. of the 29th International Conference on Very Large Data Bases (VLDB)*, September 2003.

[161] Crossbow Technology. Products: Wireless sensor networks. `http://www.xbow.com/Products/Wireless_Sensor_Networks.htm`.

[162] Douglas B. Terry, David Goldberg, David Nichols, and Brian M. Oki. Continuous queries over append-only databases. In *Proc. of the 1992 ACM SIGMOD International Conference on Management of Data*, June 1992.

[163] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, December 1995.

[164] The Condor Project. Condor high throughput computing. `http://www.cs.wisc.edu/condor/`.

[165] The NTP Project. NTP: The Network Time Protocol. `http://www.ntp.org/`.

[166] Peter A. Tucker and David Maier. Dealing with disorder. In *Proc of the Workshop on Management and Processing of Data Streams (MPDS)*, June 2003.

[167] Peter A. Tucker, David Maier, and Tim Sheard. Applying punctuation schemes to queries over continuous data streams. *IEEE Data Engineering Bulletin*, 26(1), March 2003.

[168] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3), May 2003.

[169] Randy Urbano. *Oracle Streams Replication Administrator's Guide, 10g Release 1 (10.1).* Oracle Corporation, December 2003.

[170] Tolga Urhan and Michael J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *Proc. of the 27th International Conference on Very Large Data Bases (VLDB)*, September 2001.

[171] Dinesh C. Verma. Service level agreements on IP networks. *Proceedings of the IEEE*, 92(9):1382 – 1388, September 2004.

[172] William Vickery. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance*, 16(1):8 – 37, March 1961.

[173] Stratis D. Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *Proc. of the 2002 ACM SIGMOD International Conference on Management of Data*, June 2002.

[174] Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Gün Sirer. KARMA: A secure economic framework for peer-to-peer resource sharing. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[175] Carl A. Waldspurger, Tadd Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, SE-18(2):103–117, February 1992.

[176] Martin B. Weiss and Seung Jae Shin. Internet interconnection economic model and its analysis: Peering and settlement. *Netnomics*, 6(1):43–57, 2004.

[177] Jennifer Widom and Sheldon J. Finkelstein. Set-oriented production rules in relational database systems. In *Proc. of the 1990 ACM SIGMOD International Conference on Management of Data*, June 1990.

[178] Edward Wustenhoff. Service level agreement in the data center. In *SUN BluePrints Online*, April 2002.

[179] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the Borealis stream processor. In *Proc. of the 21st International Conference on Data Engineering (ICDE)*, April 2005.

[180] Yong Yao and Johannes Gehrke. The Cougar approach to in-network query processing in sensor networks. *Sigmod Record*, 31(3), September 2002.

[181] Yong Yao and Johannes Gehrke. Query processing in sensor networks. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2003.