LECTURE 3
Coping with Best-Effort: Reliable Transport

This lecture discusses how end systems in the Internet cope with the best-effort properties of the Internet's network layer. We study the key reliability techniques in TCP, including TCP's cumulative acknowledgment feedback, and two forms of retransmission upon encountering a loss: *timer-driven* and *data-driven* retransmissions. The former relies on estimating the connection round-trip time (RTT) to set a timeout, whereas the latter relies on the successful receipt of later packets to initiate a packet recovery without waiting for a timeout. Examples of data-driven retransmissions include the *fast retransmission* mechanism and *selective acknowledgment (SACK)* mechanism. Because it is helpful to view the various techniques used in TCP through the lens of history, we structure our discussion in terms of various TCP variants.

We then discuss application-level framing (ALF) [3], an approach that helps achieve *selective reliability* by being more integrated with the application than TCP is with applications.

■ 3.1 The Problem: A Best-Effort Network

A best-effort network greatly simplifies the internal design of the network, but implies that there may be times when a packet sent from a sender does not reach the receiver, or does not reach the receiver in a timely manner, or is reordered behind packets sent later, or is duplicated. A transport protocol must deal with these problems:

1. *Losses*, which typically because of congestion, packet corruption due to noise or interference, routing anomalies, or link/path failures.

   Congestion losses are a consequence of using a packet-switched network for statistical multiplexing, because a burst of packets arriving into a queue may exhaust buffer space. It is usually a bad idea to maintain a really large queue buffer, because an excessively large buffer only causes packets to be delayed and doesn't make them move through the network any faster! We will discuss the factors governing queue buffer space during our discussions on router design and congestion control.

2. *Variable delays* for packet delivery.

3. Packet *reordering*, which typically arises because of multiple paths between end-points, or pathologies in router implementations.

4. *Duplicate* packets, which occasionally arrive at a receiver because of bugs in network implementations or lower-layer data retransmissions.

Many applications, including file transfers, interactive terminal sessions, etc. require reliable data delivery, with packets arriving *in-order*, in the same order in which the sender sent them. These applications will benefit from a *reliable transport protocol*, which is implemented in a layer below the application (typically in the kernel) and provides an in-order, reliable, byte-stream abstraction to higher-layer applications via a well-defined "socket" interface. TCP provides such semantics.

However, this layered in-order TCP model is not always the best way to transport application data. First, many applications can process data that arrives out-of-order. For example, consider the delivery of video or audio streams on the Internet. The loss of a packet in a video or audio stream may not be important, because the receiver is capable of "hiding" this loss by averaging neighboring samples, with only a small degradation in quality to the end user. However, it might be the case that some video packets are important, because they contain information useful both to the current frame and to subsequent frames. These lost packets may need to be recovered. Hence, *selective reliability* is a useful goal.

Second, if data is provided to the application *only* in-order as with TCP, a missing packet causes a stall in data delivery until it is retransmitted by the sender, after which the receiver application has to process a number of packets. If the receiver application cares about interactive response, or if receiver processing before the user looks at the presented information is a bottleneck, a TCP-like approach is sub-optimal.

The above observations motivate a different (non-layered) approach to transport protocol design, called *ALF, application-level framing*.

This lecture discusses both approaches to transport protocol design. Our goal, as mentioned before, is to cope with the vagaries of a best-effort network.

■ 3.2 Transmission Control Protocol (TCP)

The TCP service model is that of an in-order, reliable, duplex, byte-stream abstraction. It doesn't treat datagrams as atomic units, but instead treats bytes as the fundamental unit of reliability. The TCP abstraction is for *unicast* transport, between two network attachment points (IP addresses, not end hosts). "Duplex" refers to the fact that the same connection handles reliable data delivery in both directions.

In general, reliable transmission protocols can use one or both of two kinds of techniques to achieve reliability in the face of packet loss. The first, called *Forward Error Correction* (FEC), is to use redundancy in the packet stream to overcome the effects of some losses. The second is called *Automatic Repeat reQuest* (ARQ), and uses packet retransmissions. The idea is for the sender to infer the receiver's state using acknowledgments (ACKs) it gets, and determine that packets are lost if an ACK hasn't arrived for a while, and retransmit if necessary.

In TCP, the receiver periodically[1] informs the sender about what data it has received via *cumulative ACKs*. For example, the sequence of bytes:

```
1:1000 1001:1700 2501:3000 3001:4000 4001:4576
```

received at the receiver will cause the ACKs

```
1001 1701 1701 1701 1701
```

to be sent by the receiver after each segment arrival. Each ACK acknowledges all the bytes received in-sequence thus far, and tells the sender what the next expected in-sequence byte is.

Each TCP ACK includes in it a receiver-advertised window that tells the sender how much space is available in its socket buffer at any point in time. This window is used for end-to-end *flow control*, and should not be confused with *congestion control,* which is how resource contention for "inside-the-network" router resources (bandwidth, buffer space) is dealt with. Flow control only deals with making sure that the sender doesn't overrun the receiver at any stage (it's a lot easier than congestion control, as you can see). We will study congestion control in more detail in later lectures.

TCP has two forms of retransmission upon encountering a loss: *timer-driven* retransmissions and *data-driven* retransmissions. The former relies on estimating the connection round-trip time (RTT) to set a timeout value; if an ACK isn't received within the timeout duration of sending a segment, that segment is retransmitted. On the other hand, the latter relies on the successful receipt of later packets to initiate a packet recovery without waiting for a timeout.

Early versions of TCP used only timer-driven retransmissions. The sender would estimate the average round-trip time of the connection, and the absence of an ACK within that timeout duration would cause the sender to retransmit the entire window of segments starting from the last cumulative ACK. Such retransmissions are called *go-back-N* retransmissions, and are quite wasteful because they end up re-sending an entire window (but they are simple to implement). Early TCPs did not have any mechanism to cope with congestion, either, and ended up using fixed-size sliding windows throughout a data transfer.

The evolution of TCP went roughly along the following lines (what follows isn't a timeline, but a sequence of important events):

1. In the mid-1980s, portions of the Internet suffered various congestion-collapse episodes. In response, Van Jacobson and Mike Karels developed a set of techniques to mitigate congestion [10]. Among these were better round-trip time (RTT) estimators, and the "slow start" and "congestion avoidance" congestion control schemes (which we will study in detail in a later lecture).

2. Karn and Partridge's solution to the "retransmission ambiguity" problem, in which a TCP sender could not tell the difference between an ACK for an original transmission and a retransmission [12].

[1]Every time it receives a segment; modern TCP receivers implement a *delayed ACK* policy where they should ACK only on every other received segment as long as data arrives in sequence, and must acknowledge at least once every 500 ms if there is any unacknowledged data. BSD-derived implementations use a 200 ms "heartbeat" timer for delayed ACKs.

3. The TCP timestamp option, which provide more accurate RTT estimation for TCP and eliminates the retransmission ambiguity problem.

4. Coarse-grained timers: After various attempts, researchers realized that developing highly accurate timers for reliable transport protocols was a pipe-dream, and that timeouts should be conservative.

5. TCP Tahoe, which combined the Jacobson/Karels techniques mentioned above with the first data-driven retransmission scheme for TCP, called *fast retransmission*. The idea is to use duplicate ACKs as a signal that a data segment has been lost.

6. TCP Reno, which extended fast retransmissions with *fast recovery*, correcting some shortcomings in Tahoe. Reno also implemented a "header prediction" scheme for more efficient high-speed implementation (this mechanism has no bearing on TCP's reliability functions).

7. TCP NewReno, developed by Janey Hoe, an improvement to the fast recovery scheme, allowing multiple data losses in large-enough windows to be recovered without a timeout in many cases.

8. TCP Selective Acknowledgments (SACK), standaradized in RFC 2018, which extends TCP's cumulative ACKs to include information about segments received out-of-order by a TCP receiver [13].

There have been other enhancements proposed to TCP over the past few years, such as TCP Vegas (a congestion control method), various optimizations for wireless networks, optimizations for small windows (e.g., RFC 3042), etc. We don't discuss them here.

The result of all these proposals is that current TCPs rarely timeout if window sizes are large enough (more than 6 or 7 segments), unless most of the window is lost, or most of the returning ACKs are lost, or retransmissions for lost segments are also lost. Moreover, current TCP retransmissions adhere to two important principles, both of which are important for sound congestion control (which we will study in a later lecture):

P1 Don't retransmit early; avoid spurious retransmissions. The events of the mid-1980s showed that congestion collapse was caused by the retransmission of segments that weren't lost, but just delayed and stuck in queues. TCP attempts to avoid spurious retransmissions.

P2 Conserve packets. TCP attempts to "conserve" packets in its data-driven retransmissions. It uses returning duplicate ACKs as a cue signifying that some segments have been received, and carefully determines what segments should be sent, if any, in response to these duplicate ACKs. The idea is to avoid burdening the network with excessive load during these loss periods.

The rest of this lecture discusses the techniques mentioned before, roughly in the order given there. We start with TCP timers.

3.2.1 TCP timers

To perform retransmissions, the sender needs to know when packets are lost. If it doesn't receive an ACK within a certain amount of time, it assumes that the packet was lost and retransmits it. This is called a *timeout* or a *timeout-triggered* retransmission. The problem is to determine how long the timeout period should be.

What factors should the timeout depend on? Clearly, it should depend on the connection's round-trip time (RTT). To do this, the sender needs to estimate the RTT. It obtains samples by monitoring the time difference between sending a segment and receiving a positive ACK for it. It needs to do some averaging across all these samples to maintain a (running) average. There are many ways of doing this; TCP picks a simple approach called the Exponential Weighted Moving Average (EWMA), where $srtt = \alpha \times r + (1 - \alpha)srtt$. Here, r is the current sample and *srtt* the running estimate of the *smoothed* RTT. In practice, TCP implementations use $\alpha = 1/8$ (it turns out the precise value doesn't matter too much).

We now know how to get a running average of the RTT. How do we use this to set the retransmission timeout (RTO)? One option, an old one used in the original TCP specification (RFC 793), is to pick a multiple of the smoothed RTT and use it. For example, we might pick $RTO = \beta * srtt$, with β set to a constant like 2. Unfortunately, this simple choice doesn't work too well in preventing *spurious* retransmissions when slow start is used on certain network paths, leading to bad congestion effects, because the principle of "conservation of packets" will no longer hold true.

A nice and simple fix for this problem is to make the RTO a function of both the average and the standard deviation. In general, the tail probability of a spurious retransmission when the RTO is a few standard deviations away from the mean is rather small. So, TCP uses an RTO set according to: $RTO = srtt + 4 \times rttdev$, where *rttdev* is the mean linear deviation of the RTT from its mean. I.e., *rttdev* is calculated as $rttdev = \gamma \times dev + (1 - \gamma) \times rttdev$, where $dev = |r - srtt|$. In practice, TCP uses $\gamma = 1/4$.

These calculations aren't the end of the TCP-timer story. TCP also suffers from a significant *retransmission ambiguity* problem. When an ACK arrives for a segment the sender has retransmitted, how does the sender know whether the RTT to use is for the original transmission or for the retransmission? This question might seem like a trivial detail, but in fact is rather vexing because the RTT estimate can easily become meaningless and throughput can suffer. The solution to this problem is surprisingly simple—ignore samples that arrive when a retransmission is pending or in progress. I.e., don't consider samples for any segments that have been retransmitted [11].

The modern way of avoiding the retransmission ambiguity problem is to use the TCP *timestamp option* (RFC 1323), which most current (and good) TCP implementations adopt. Here, the sender uses 8 bytes (4 for seconds, 4 for microseconds) and stamps its current time on the segment. The receiver, in the cumulative ACK acknowledging the receipt of a segment, simply *echoes* the sender's stamped value. When the ACK arrives, the sender can do the calculation trivially by subtracting the echoed time in the ACK from the current time. Note that it now doesn't matter if this was a retransmission or an original transmission; the timestamp effectively serves as a "nonce" for the segment.

The other important issue is deciding what happens when a retransmission times out. Obviously, because TCP is a "fully reliable" end-host protocol, the sender must try again. But rather than try at the same frequency, it takes a leaf out of the contention resolution

protocol literature (e.g., Ethernet CSMA) and performs *exponential backoffs* of the retransmission timer.

A final point to note about timeouts is that they are extremely *conservative* in practice. TCP retransmission timers are usually (but not always) coarse, with a granularity of 500 or 200 ms. This is a big reason why spurious retransmissions are rare in modern TCPs, but also why timeouts during downloads are highly perceptible by human users.

■ 3.2.2 Fast retransmissions

Because timeouts are expensive (in terms of killing throughput for a connection, although they are a necessary evil from the standpoint of ensuring that senders back-off under extreme congestion), it makes sense to explore other retransmission strategies that are more responsive. Such strategies are also called *data-driven* (as opposed to *timer-driven*) retransmissions. Going back to the earlier example:

```
1:1000 1001:1700 2501:3000 3001:4000 4001:4576
```

with ACKs

```
1001      1701      1701      1701      1701
```

It's clear that a sequence of repeated ACKs are a sign that something strange is happening, because in normal operation cumulative ACKs should monotonically increase. Repeated ACKs in the middle of a TCP transfer can occur for three reasons:

1. Window updates. When the receiver finds more space in its socket buffer, because the application has read some more bytes, it can send a window update even if no new data has arrived.

2. Segment loss.

3. Segment reordering. This could have happened, for example, if datagram 1701-2500 had been reordered because of different routes for different segments.

Repeated ACKs that aren't window updates are called *duplicate* ACKs or *dupacks*. TCP uses a simple heuristic to distinguish losses from reordering: if the sender sees an ACK for a segment more than three segments larger than a missing one, it assumes that the earlier (unacknowledged) segment has been lost. Unfortunately, cumulative ACKs don't tell the sender which segments have reached; they only tell the sender what the last in-sequence byte was. So, the sender simply *counts* the number of dupacks and infers that if it see three or more dupacks, that the corresponding segment was lost. Various empirical studies have shown that this heuristic works pretty well, at least on today's Internet, where reordering TCP segments is discouraged and there isn't much "parallel" ("dispersity") routing.

■ 3.2.3 Fast Recovery

A TCP sender that performs a "fast retransmission" after receiving three duplicate ACKs must reduce its congestion window, because the loss was most likely due to congestion (we will study congestion control in depth in a later lecture). TCP Tahoe senders go into

"slow start", setting the window size to 1 segment, and sending segments from the next new ACK. (If no new ACK arrives, then the sender times out.)

The problem with this approach is that the sender may send segments that are already received at the other end.

A different approach, called *fast recovery*, cuts the congestion window by one-half on a fast retransmission. As duplicate ACKs beyond the third one arrive for the window in which the loss occurred, the sender waits until half the window has been ACKed, and then sends one *new* segment per subsequent duplicate ACK. Fast recovery has the property that at the end of the recovery, assuming that not too many segments are lost in the window, the congestion window will be one-half what it was when the loss occurred, and that many segments will be in flight.

TCP Reno uses this combination of fast retransmission and fast recovery. This TCP variant, which was the dominant version in the 1990s, can recover from a small number of losses in a large-enough window without a timeout (the exact details depend on the pattern of losses; for details, see [6]).

In 1996, Hoe proposed an enhancement to TCP Reno, which subsequently became known as *NewReno*. The main idea here is for a sender to remain in fast recovery until all the losses in a window are recovered.

■ 3.2.4 Selective acknowledgment (SACK)

When the bandwidth-delay product of a connection is large, e.g., on a high-bandwidth, high-delay link like a satellite link, windows can get pretty large. When multiple segments are lost in a single window, TCP usually times out. This causes abysmal performance for such connections, which are colloquially called "LFNs" (for "Long Fat Networks" and pronounced "elephants," of course). Motivated in part by LFNs, selective ACKs (SACKs) were proposed as an embellishment to standard cumulative ACKs. SACKs were standardized a few years ago by RFC 2018, after years of debate.

Using the SACK option, a receiver can inform the sender of up to three maximally contiguous blocks of data it has received. For example, for the data sequence:

```
1:1000 1001:1700  2501:3000  3001:4000 4577:5062 6000:7019
```

received, a receiver would send the following ACKs and SACKs (in brackets):

```
1001     1701       1701       1701      1701        1701
                 [2501-3000] [2501-4000] [4577-5062; [6000-7019;
                                          2501-4000]  4577-5062;
                                                      2501-4000]
```

SACKs allow LFN connections to recover many losses in the same window with a much smaller number of timeouts. While SACKs are in general a Good Thing, they don't always prevent timeouts, including some situations that are common in the Internet today. One reason for this is that on many paths, the TCP window is rather small, and multiple losses in these situations don't give an opportunity for a TCP sender to use data-driven retransmissions.

3.2.5 Some other issues

There are a few other issues that are important for TCP reliabilty.

1. **Connection setup/teardown.** At the beginning of a connection, a 3-way handshake synchronizes the two sides. At the end, a teardown occurs.

2. **Segment size.** How should TCP pick its segment size for datagrams? Each side picks a "default" MSS (maximum segment size) and exchanges them in the SYN exchange at connection startup; the smaller of the two is picked.

   The recommended way of doing this, to avoid potential network fragmentation, is via *path MTU discovery.* Here, the sender sends a segment of some (large) size that's smaller than or equal to its interface MTU (when the IP and link-layer headers are added) with the "DON'T FRAGMENT (DF)" flag in the IP header. If the receiver gets this segment, then clearly every link en route could support this datagram size because there was no fragmentation. If not, and an ICMP error message was received by the sender from a router saying that the packet was too big and would have been fragmented, the sender tries a smaller segment size until one works.

3. **Low-bandwidth links.** Several TCP segments are small in size and are mostly comprised of headers; examples include telnet packets and TCP ACKs[2]. To work well over low-bandwidth links, TCP header compression can be done (RFC 1144). This takes advantage of the fact that most of the fields in a TCP header are either the same or are predictably different from the previous one. This allows a 40-byte TCP+IP header to be reduced to as little as 3-6 bytes.

3.3 ALF

A good way to understand the ALF idea is by example. Consider the problem of designing a protocol to stream video on the Internet. In most (but not all) video compression formats including the MPEG variants, there are two kinds of compressed frames—*reference frames* and *difference frames*. A reference frame is compressed by itself, whereas a difference frame is compressed in terms of its difference from previous frames (in practice, some frames may be compressed as a difference not just from a previous frame but also a succeeding frame). Because video scenes typically don't change dramatically from frame to frame, a difference frame usually compresses a lot better than a reference frame. However, if packets are lost in the stream and not recovered, a stream that uses only one reference frame followed by only difference frames suffers from the *propagation of errors* problem, because the subsequent frames after a lost packet end up cascading the errors associated with missing data.

One approach to transporting streaming video is over TCP, but this has two problems. First, depending on the nature of the loss, recovery may take between one round-trip time and several seconds (if a long timeout occurs). Because not all lost packets need to be recovered, the receiver application might choose interactive presentation of whatever frames

[2]TCP does allow data to be piggybacked with ACKs and most data segments have valid ACK fields acknowledging data in the opposite direction of the duplex connection.

are available, over waiting for missing data that has only marginal value. However, some packets (e.g., in reference frames) may need to be *selectively* recovered. Second, when using TCP, a lost packet would cause all of the later packets in the stream to *wait* in the receiver's kernel buffers without being delivered to the application for processing. If receiver application processing is a bottleneck, a lost packet causes the application's interactive performance to degrade.

The above discussion suggests that what is needed is:

1. Out-of-order data delivery from the transport protocol to the receiver application, and

2. The ability for the receiver application to request the selective retransmission of specific data items.

One solution to this problem is to ask each application designer to design an application-specific transport protocol anew for each new application. Punting the problem like this doesn't seem like a good idea, because most such efforts will probably end up using very similar ideas.

It is hard to simply hack TCP to make the TCP receiver push out-of-order packets to the receiver application. The reason is that there is no shared vocabulary between the application and TCP to name data items! The application may have it's own naming method for data items (e.g., this packet is frame 93 and has to be displayed at coordinate (x, y) of the video screen), but the TCP knows nothing of this. TCP's name for the corresponding packet would be a sequence number and length. Unfortunately, the receiver application has no idea what the TCP name for the data item corresponds to in the application's vocabulary.

What is required is a *common vocabulary* between the application and the transport to name data items. A convenient way to do this is to use the application's own name in the transport protocol, which means that an *application data unit* (what the application thinks of as independenly processible data items) and the *protocol data unit* (what the transport thinks of as indepedently transmittable and receivable data items) are one and the same thing. By making these two data units equivalent, when an out-of-order data unit arrives at the receiver transport protocol, it can deliver it to the receiver application and have it be understood because the delivered data is named in a manner that the application understands.

Selective reliability is obtained easily with ALF, because a missing application data unit that is deemed important by the receiver can be requested by the receiver application to the receiver transport library. In turn, the transport library sends a retransmission request to the sender library, and the required data is re-sent to the receiver either from the sender library's buffer (if it has one) or more likely via a callback to the sender application itself. Again, our common naming between transport and applications of the data items is instrumental in making this approach work.

It is hard to design an ALF mechanism in a strictly layered fashion where the ALF transport is in the kernel and the application is across a strict protection boundary. Instead, the best way to realize ALF is to design the transport as a *library* that sender and receiver applications link to, and use the library to send and receive data items.

Several ALF applications have been implemented in practice, including for video transport, image transport, shared whiteboards and collaborative tools, and multicast conferencing. ALF is a clever and useful way achieving out-of-order data delivery to applications, and providing a way for applications to achieve selective reliability.

■ 3.4 Summary

TCP provides an in-order, reliable, duplex, byte-stream abstraction. It uses timer-driven and data-driven retransmissions; the latter provides better performance under many conditions, while the former ensures correctness.

References

[1] T. Bates, R. Chandra, and E. Chen. *BGP Route Reflection - An Alternative to Full Mesh IBGP*. Internet Engineering Task Force, Apr. 2000. RFC 2796. (Cited on page 11.)

[2] I. V. Beijnum. *BGP*. O'Reilly and Associates, Sept. 2002. (Cited on page 8.)

[3] D. Clark and D. Tennenhouse. Architectural Consideration for a New Generation of Protocols. In *Proc. ACM SIGCOMM*, pages 200–208, Philadelphia, PA, Sept. 1990. (Cited on page 1.)

[4] R. Dube. A Comparison of Scaling Techniques for BGP. *ACM Computer Communications Review*, 29(3):44–46, July 1999. (Cited on page 9.)

[5] Cisco IOS IP Command Reference, ebgp-multihop. `http://www.cisco.com/en/US/products/sw/iosswrel/ps1835/products_command_reference_chapter09186a00800ca79d.html`, 2005. (Cited on page 12.)

[6] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno, and Sack TCP. *ACM Computer Communications Review*, 26(3):5–21, July 1996. (Cited on page 7.)

[7] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *Proc. 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, pages 43–56, Boston, MA, May 2005. (Cited on page 12.)

[8] T. Griffin and G. Wilfong. On the Correctness of IBGP Configuration. In *Proc. ACM SIGCOMM*, pages 17–29, Pittsburgh, PA, Aug. 2002. (Cited on pages 9 and 12.)

[9] C. Hedrick. *Routing Information Protocol*. Internet Engineering Task Force, June 1988. RFC 1058. (Cited on page 2.)

[10] V. Jacobson. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM*, pages 314–329, Stanford, CA, Aug. 1988. (Cited on pages 3 and 19.)

[11] P. Karn. MACA – A New Channel Access Method for Packet Radio. In *Proc. 9th ARRL Computer Networking Conference*, 1990. (Cited on page 5.)

[12] P. Karn and C. Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *ACM Transactions on Computer Systems*, 9(4):364–373, Nov. 1991. (Cited on page 3.)

[13] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *TCP Selective Acknowledgment Options*. Internet Engineering Task Force, 1996. RFC 2018. (Cited on page 4.)

[14] J. Moy. *OSPF Version 2*, Mar. 1994. RFC 1583. (Cited on page 2.)

[15] D. Oran. *OSI IS-IS intra-domain routing protocol*. Internet Engineering Task Force, Feb. 1990. RFC 1142. (Cited on page 2.)

[16] Y. Rekhter and T. Li. *A Border Gateway Protocol 4 (BGP-4)*. Internet Engineering Task Force, Mar. 1995. RFC 1771. (Cited on pages 2 and 8.)

[17] Y. Rekhter, T. Li, and S. Hares. *A Border Gateway Protocol 4 (BGP-4)*. Internet Engineering Task Force, Oct. 2004. `http://www.ietf.org/internet-drafts/draft-ietf-idr-bgp4-26.txt` Work in progress, expired April 2005. (Cited on page 2.)

[18] P. Traina, D. McPherson, and J. Scudder. *Autonomous System Confederations for BGP*. Internet Engineering Task Force, Feb. 2001. RFC 3065. (Cited on page 11.)