Tesla: A Transparent, Extensible Session-Layer Framework for End-to-end Network Services

by

Jonathan Michael Salz

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 24, 2002

Author . . . .
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by . . . .
Hari Balakrishnan
Assistant Professor in Computer Science
Thesis Supervisor

Certified by . . . .
Alex C. Snoeren
Candidate, Doctor of Philosophy in Computer Science
Thesis Supervisor

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

TESLA: A Transparent, Extensible Session-Layer Framework for End-to-end Network Services

by

Jonathan Michael Salz

Submitted to the
Department of Electrical Engineering and Computer Science

May 24, 2002

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis argues that session-layer services for enhancing functionality and improving network performance are gaining in importance in the Internet; examples include connection multiplexing, congestion state sharing, application-level routing, mobility/migration support, encryption, and so on. To facilitate the development of these services, we describe TESLA, a transparent and extensible framework that allows session-layer services to be developed using a high-level flow-based abstraction (rather than sockets), enables them to be deployed transparently using dynamic library interposition, and enables them to be composed by chaining event handlers in a graph structure. We show how TESLA can be used to design several interesting session-layer services including encryption, SOCKS and application-controlled routing, flow migration, and traffic rate shaping, all with acceptably low performance degradation.

Thesis Supervisor: Hari Balakrishnan
Title: Assistant Professor in Computer Science

Thesis Supervisor: Alex C. Snoeren
Title: Candidate, Doctor of Philosophy in Computer Science

Acknowledgments

The author wishes especially to thank Professor Hari Balakrishnan, Alex Snoeren, and Dave Andersen for their technical contributions to this thesis.

Thanks also to Michael Artz, Enid Choi, Doug Creager, and Derik Pridmore for their contributions, technical and otherwise.

Contents

List of Figures

Chapter 1

Introduction

Modern network applications must meet several increasing demands for performance and enhanced functionality. Much current research is devoted to augmenting the transport-level functionality implemented by standard protocols as TCP and UDP. Examples abound: Setting up multiple connections between a source and destination to improve the throughput of a single logical data transfer (e.g., file transfers over high-speed networks where a single TCP connection alone does not provide adequate utilization [2, 19]); sharing congestion information across connections sharing the same network path (e.g., the Congestion Manager [4, 6, 5]); application-level routing, where applications route traffic in an overlay network to the final destination (e.g., Resilient Overlay Networks [3]); end-to-end session migration for mobility across network disconnections [33]; encryption services for sealing or signing flows [10]; general-purpose compression over low-bandwidth links; and traffic shaping and policing functions. These examples illustrate the increasing importance of *session layer services* in the Internet—services that operate on groups of flows between a source and destination, and produce resulting groups of flows using shared code and sometimes shared state.

Authors of new services such as these often implement enhanced functionality by augmenting the link, network, and transport layers, all of which are typically implemented in the kernel or in a shared, trusted intermediary [14]. While this model has sufficed in the past, we believe that a generalized, high-level framework for session-layer services would greatly ease their development and deployment. This thesis argues that Internet end hosts can benefit from a *systematic* approach to developing session-layer services compared to the largely *ad-hoc* point approaches used today, and presents TESLA (a Transparent, Extensible Session Layer Architecture), a framework that facilitates the development of session-layer services like the ones mentioned above.

Our work with TESLA derives heavily from our colleagues' and our own previous experience developing, debugging, and deploying a variety of session-layer services for the Internet. The earliest example is the Congestion Manager (CM) [6], which allows concurrent flows with a common source and destination to share congestion information, allocate available bandwidth, and adapt to changing network conditions. CM is currently implemented in Linux kernel, in large part because CM needs to integrate with TCP's in-kernel congestion controller in order to support sharing across TCP flows [4]. There is no such requirement when CM controls only UDP flows, however—a capability that has proven quite useful [17]. In these cases, it would be advantageous (for portability and ease of deployment) to have a session-layer implementation of CM running at user level. Unfortunately, implementing a user-level CM is quite an intricate process [30]. Not only must the implementation specify the internal logic, algorithms, and API provided by CM, but considerable care must be taken handling the details of non-blocking and blocking sockets, inter-process communication between CM and applications, process management, and integrating the CM API with the application's event loop. The end result is that more programmer time and effort was spent setting up the session-layer plumbing than in the CM logic itself.

This is not an isolated example—we and our colleagues have had similar experiences in our work on Resilient Overlay Networking (RON) [3] and the Migrate mobility architecture [33]. Both can be viewed as session-layer services: RON provides customized application-layer routing in an overlay network, taking current network performance and outage conditions into account; Migrate preserves end-to-end communication across relocation and periods of disconnection. Our similar frustrations with the development and implementation of these services was the prime motivation behind TESLA, and lead to three explicit design goals.

First, it became apparent to us that the standard BSD sockets API [28] is not a convenient abstraction for programming session-layer services. In response, TESLA exports a higher level of abstraction to session services, allowing them to operate on network flows (rather than socket descriptors), treating flows as objects to be manipulated.

Second, there are many session services that are required *post facto*, often not originally thought of by the application developer but desired later by a system operator for flows being run by users. For example, the ability to shape or police flows to conform to a specified peak rate is often useful, and being able to do so without kernel modifications is a deployment advantage. This requires the ability to configure session services transparent to the application. To do this, TESLA uses an old idea—dynamic library interposition [11]—taking advantage of the fact that most applications today on modern operating systems use dynamically linked libraries to gain access to kernel services. This does not, however, mean that TESLA session-layer services must be transparent. On the contrary, TESLA allows services to define APIs to be exported to enhanced applications.

Third, unlike traditional transport and network layer services, there is a great diversity in session services as the examples earlier in this section show. This implies that application developers can benefit from composing different available services to provide interesting new functionality. To facililate this, TESLA arranges for session services to be written as event handers, with a callback-oriented interface between handlers that are arranged in a graph structure in the system.

Figure 1-1 presents a high-level illustration of the TESLA architecture.

1.1 Related Work

Functionally, TESLA combines concepts from three areas of research. At the highest level, TESLA brings much of the flexibility of extensible network protocol stacks

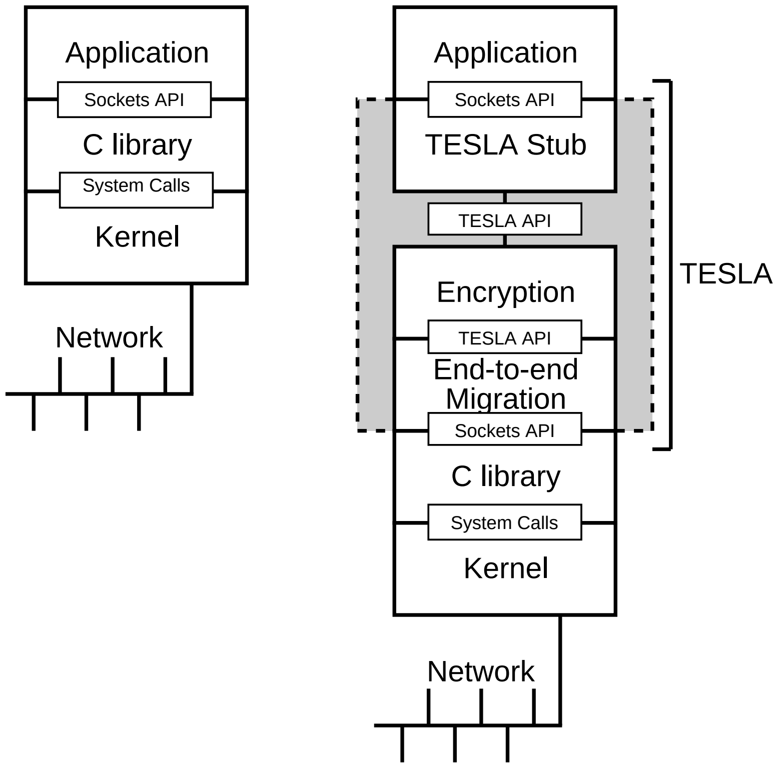


Figure 1-1: A typical networked system is at left. At right is a TESLA-enabled system with encryption and flow migration handlers enabled.

found in several research operating systems to commodity operating systems. TESLA's component-based modular structure is similar in spirit to a number of existing composable network protocol platforms, yet targeted specifically towards end-to-end session services with an eye to making these services easier to program. Unlike most previous systems, however, TESLA does not require modified applications, special operating system extensions, or super-user privileges. Instead, TESLA borrows well-known interposition techniques to provide an extensible protocol framework entirely at user level, allowing dynamic composition and transparent operation with legacy applications.

1.1.1 Extensible network stacks

Today's commodity operating systems commonly allow the dynamic installation of network protocols on a system-wide or per-interface basis (e.g., Linux kernel modules and FreeBSD's netgraph), but these extensions can only be accessed by the super-user. Some operating systems, such as SPIN [7] and the Exokernel [14], push many operating system features (like network and file system access) out from the kernel into application-specific, user-configurable libraries, allowing ordinary users fine-grained control. Alternatively, extensions were developed for both operating systems to allow applications to define application-specific handlers that may be installed directly into the kernel (Plexus [18] and ASHs [36]).

Operating systems such as Scout [29] and *x*-kernel [21] were designed explicitly to support sophisticated network-based applications. In these systems, users may even redefine network or transport layer protocol functions in an application-specific fashion [8]. With TESLA, our goal is to bring some of the power of these systems to commodity operating systems in the context of session-layer services.

Several other projects have explored user-level networking in the context of traditional operating systems. U-Net [35] provides direct, user-level access to the network interface and an infrastructure to implement user-level network stacks. Alpine [13] virtualizes much of the FreeBSD network stack, moving it to the user level, although it is highly platform-dependent and intended mostly for debugging networking modules which are to be moved to the kernel.

In contrast to these systems, TESLA does not attempt to allow users to replace or modify the system's network stack. Instead, it intends only to allow users to dynamically extend the protocol suite by dynamically composing additional end-to-end session-layer protocols on top of the existing transport- and network-layer protocols.

1.1.2 Composable network protocols

TESLA's modular structure shares commonalities with a number of previous systems, *x*-kernel in particular [21]. Like TESLA, the *x*-kernel builds a directed graph of processing nodes and passes network data between them. However, the models of control flow and protection differ in significant ways. Rather than create a distinct, flow-specific instance of each handler, *x*-kernel separates handlers into data and code components called protocols and sessions, and passes data alternatively between the two. Each protocol has one shared instance with multiple, separate session data structures. Since each incoming data packet is handled by its own thread of control, protocols must explicitly handle concurrency. Further, *x*-kernel's use of a single address-space model affords no inter-protocol protection.

Unlike the transport and network protocols typically considered in *x*-kernel, we view session-layer protocols as an extension of the application itself rather than a system-wide resource. Hence, TESLA ensures they are subject to the same scheduling, protection, and resource constraints as the application. This also contrasts with the scheduling and buffer model found in the Click modular router [24]. In particular, Click goes to great lengths to ensure that all queues are explicit and packet handling does not stall at arbitrary locations. In contrast, our queues are intentionally implicit. Since TESLA flow handlers are subject to context switches at any time, any attempt to specify specific points at which data should be queued would be futile. In many ways, TESLA is most similar to UNIX System V streams [32], although queues in TESLA can never block. TESLA achieves a model of programming akin to System V streams without requiring any kernel modifications and allows session applications to be written by manipulating flow handler objects.

1.1.3 Interposition agents

To avoid making changes to the operating system or to the application itself, TESLA transparently interposes itself between the application and the kernel, intercepting and modifying the interaction between the application and the system—acting as an *interposition agent* [23]. There are many viable interposition mechanisms, including ptrace-style system call interception, dynamic linking, and kernel hooks; the pros and cons of these techniques are well studied [1, 11] and implemented in a variety of systems.

A number of interposition techniques require assistance from the operating system. Jones developed an Interposition Agent toolkit [23] using the Mach 1 system-call redirection facility. SLIC [20] is a general interposition framework usable with certain production systems such as Solaris and Linux but requires patches to kernel structures. TESLA uses the more general technique of dynamic library interposition [11].

Some previously developed systems provide file system functionality by overloading dynamic library functions. This mechanism is especially popular for user-level file systems like IFS [12], Jade [31], and Ufo [1]. The 3D File System [25] is implemented using COLA [26], a generalized overlay mechanism for library interposition. Thain and Livny recently proposed Bypass, a similar dynamic-library based interposition toolkit for building split-execution agents commonly found in distributed systems [34]. Because of their generality, neither of these systems provides any assistance in building modular network services, particularly at the session layer.

Some existing libraries [9, 22] provide transparent access to a SOCKS [27] proxy using interposition. Reliable Sockets (Rocks) protects socket-based applications from network failures by wrapping socket-related system calls [38]; a similar method was used to implement the transparent connection migration functionality found in the latest iteration of Migrate [33][1]. Conductor [37] traps application network operations

[1]We borrowed heavily from Migrate's interposition layer in developing TESLA. Indeed, Migrate has since discarded its original *ad-hoc* interposition layer in favor of TESLA.

and transparently layers composable "adaptors" on TCP connections, but its focus is on optimizing flows' performance characteristics, not on providing arbitrary additional services. To the best of our knowledge, TESLA represents the first interposition toolkit to support generic session-layer network services.

1.2 Contributions

We argue that a generalized architecture for the development and deployment of session-layer functionality will significantly assist in the implementation and use of new network services. This thesis describes the design and implementation of TESLA, a generic framework for development and deployment of session-layer services. TESLA consists of a set of C++ application program interfaces (APIs) specifying how to write these services, and an interposition agent that can be used to instantiate these services for transparent use by existing applications.

We have analyzed the performance of our TESLA implementation and find that it runs with acceptable performance degradation. It imposes overhead comparable to that of another common interposition mechanism—using a proxy server—but is more powerful and transparent.

To demonstrate the feasibility of developing services with TESLA, and the power and ease of use of the interfaces it provides, we have implemented several TESLA handlers providing services such as compression, encryption, transparent use of a SOCKS [27] proxy, application-level routing [3], flow migration [33], and traffic shaping. We find services significantly easier to write with TESLA than when developing them from the ground up.

1.3 Organization

The next chapter describes the architecture of TESLA and how it meets the three goals of using a high-level flow-based abstraction, transparency, and composition. Chapter 3 shows how handlers are designed and chained together, and describes several implemented handlers. Chapter 4 discusses some implementation issues, and gives experimental results that demonstrate that Tesla does not incur significant performance degradation in practice. Chapter 5 summarizes our contributions.

Chapter 2

Architecture

This section discusses the architecture of the TESLA framework. We introduce the flow handler interface, the fundamental unit of abstraction which describes a particular session-layer service. We describe how flow handlers communicate with each other and discuss the mapping between flow handlers and UNIX processes. Finally, we describe dynamic library interposition, the mechanism that allows TESLA to act as an interposition agent between applications and system libraries, specially handling application networking calls. We identify some of this mechanism's security implications and possible solutions.

2.1 Flow handlers

As we discussed in Section 1.1.3, many tools exist that, like TESLA, provide an interposition (or "shim") layer between applications and operating system kernels or libraries. However, TESLA raises the level of abstraction of programming for session-layer services. It does so by making a *flow handler* the main object manipulated by session services. Each session service is implemented as an instantiation of a flow handler, and TESLA takes care of the plumbing required to allow session services to communicate with one another.

A network flow is a stream of bytes that all share the same logical source and destination (generally identified by source and destination IP addresses, source and

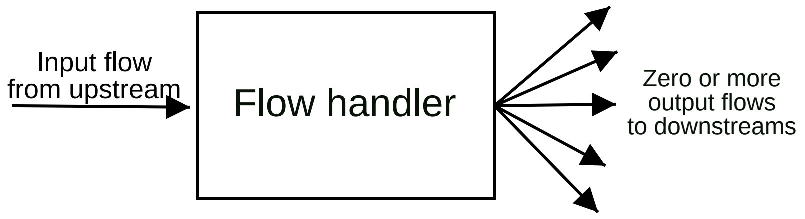


Figure 2-1: A flow handler takes as input one network flow and generates one or more output flows.

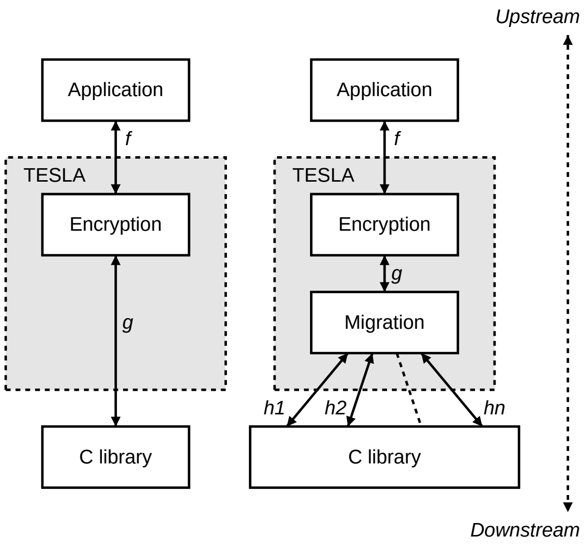


Figure 2-2: Two TESLA stacks. The migration flow handler implements input flow g with output flows $h_1 \ldots h_n$.

destination port numbers, and transport protocol). Each flow handler takes as input a single network flow, and produces zero or more network flows as output. Flow handlers perform some particular operations or transformations on the byte stream, such as transparent flow migration, encryption, compression, etc.

Figure 2-1 shows a generic flow handler. Because flow handlers are explicitly defined and constructed to operate on only one *input flow* from an *upstream* handler (or end application), they are devoid of any demultiplexing operations. Conceptually, therefore, one might think of a flow handler as dealing with traffic corresponding to a single socket (network flow), allowing the session service developer to focus primarily on the internal logic of the service. A flow handler generates zero or more *output flows*, which map one-to-one with *downstream* handlers (or the network send routine).

```
// address and data are C++ wrappers for sockaddr buffers and data
// buffers, respectively.  They are very similar to STL strings.
class address, data;

class flow_handler {
  protected:
    flow_handler *plumb(int domain, int type);
    handler* const upstream;
    vector<handler*> downstream;

    struct acceptret {
        flow_handler *const h;
        const address addr;
    };

  public:
    virtual int connect(address);
    virtual int bind(address);
    virtual acceptret accept();
    virtual int close();
    virtual bool write(data);
    virtual int shutdown(bool r, bool w);
    virtual int listen(int backlog);
    virtual address getsockname() const;
    virtual address getpeername() const;
    virtual void may_avail(bool) const;
```

Figure 2-3: Downstream methods in the flow_handler class. See Appendix B for the full interface declaration.

The left stack in Figure 2-2 illustrates an instance of TESLA where stream encryption is the only enabled handler. While the application's I/O calls *appear* to be reading and writing plaintext to some flow f, in reality TESLA intercepts these I/O calls and passes them to the encryption handler, which actually reads and writes ciphertext on some other flow g. The right stack in has more than two flows. f is the flow as viewed by the application, i.e., plaintext. g is the flow between the encryption handler and the migration handler, i.e., ciphertext. $h_1, h_2, \ldots, h_n$ are the n flows that a migration flow handler uses to implement flow g. (The migration handler initially opens flow h_1. When the host's network address changes and h_1 is disconnected, it opens flow h_2 to the peer, and so forth.) From the standpoint of the encryption handler, f is the input flow and g is the output flow. From the standpoint of the migration handler, g is the input flow and $h_1, h_2, \ldots, h_n$ are the output flows.

```
    // in each of these methods, <from> is a downstream flow.

    // Downstream flow <from> has bytes available (passed in <bytes>).
    virtual bool avail(flow_handler* from, data bytes);

    // Downstream flow <from> has a connection ready to be accepted,
    // i.e., from->accept() will succeed.
    virtual void accept_ready(flow_handler *from);

    // A connection attempt on <from> has
    // concluded, either successfully or not.
    virtual void connected(flow_handler *from, bool success);

    // <from> says: you may (or may not) write to me.
    virtual void may_write(flow_handler *from, bool may);
};
```

Figure 2-4: Upstream (callback) methods in the flow_handler class definition.

2.1.1 The flow_handler API

Every TESLA session service operates on flows and is implemented as a derived class of flow_handler, shown in Figure 2-3. To instantiate a downstream flow handler, a flow handler invokes its protected plumb method. TESLA handles plumb by instantiating handlers which appear after the current handler in the configuration.

For example, assume that TESLA is configured to perform compression, then encryption, then session migration on each flow. When the compression handler's constructor calls plumb, TESLA responds by instantiating the next downstream handler, the encryption handler; when its constructor in turn calls plumb, TESLA instantiates the next downstream handler, migration. Its plumb method creates a TESLA-internal handler to implement an actual TCP socket connection.

To send data to a downstream flow handler, a flow handler invokes the latter's write, which in turn typically performs some processing and invokes its own downstream handler's write method. Downstream handlers communicate with upstream ones via callbacks, which are invoked when data becomes available for reading by the upstream flow, for example. The parts of the flow handler class API are shown in Figures 2-3 and 2-4. (The complete API is presented in Appendix B.)

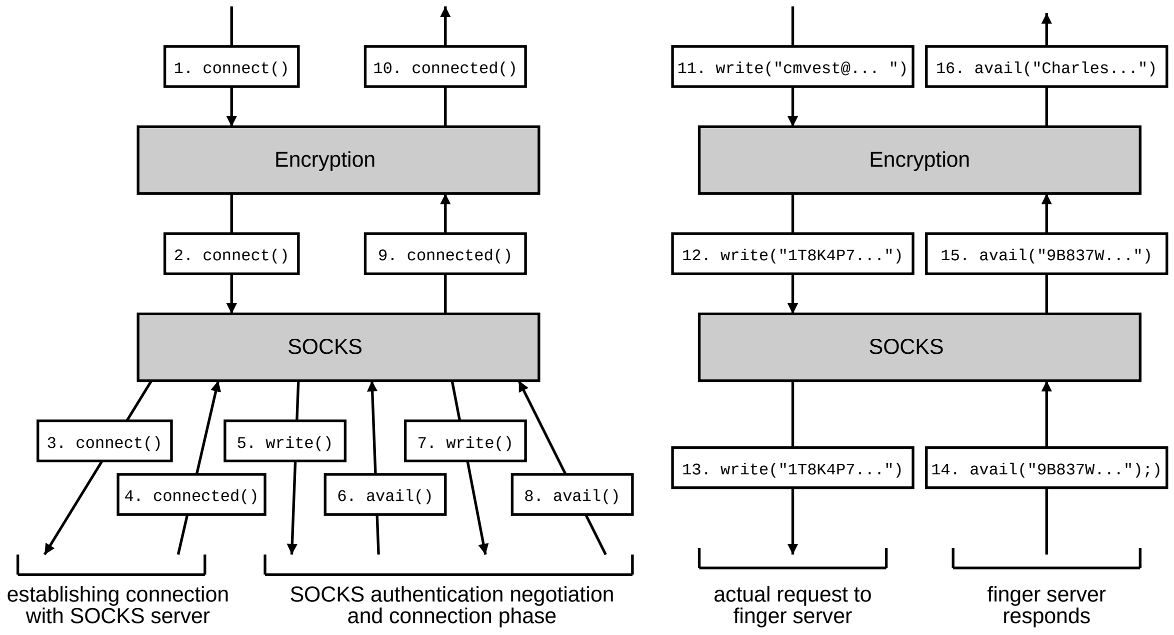


Figure 2-5: Methods invoked in the master process. TESLA invokes 1 and 11 in response to system calls by the application, and 4, 6, 8, and 14 in response to network events. TESLA handles 10 and 16 by delivering events to the application process, and handles 3, 5, 7, and 13 by actually communicating on the network.

2.1.2 Example

Figure 2-5 shows an example of how the API might be used by TESLA handlers in a stack consisting of encryption and a transparent SOCKS proxy handler (we describe both in more detail in Chapter 3). In response to a connection request by an application, TESLA invokes the connect method of the top-most handler (1), which handles it by delegating it (2) to its downstream handler, the SOCKS handler. It handles the encryption handler's connect request by trying to connect to a SOCKS server (3). Once the SOCKS server responds (4) the handler instructs the proxy server to connect to the remote host (5–7). When the server indicates (8) that the connection to the remote host has been established, SOCKS invokes its upstream connected method (9)—that of the encryption flow handler—which calls its own upstream connected method (10). At this point TESLA notifies the application that the connection has succeeded, i.e., by indicating success to the connect system call which initiated the process. The right diagram in Figure 2-5 illustrates the methods invoked when the application writes a message to the server (11–13) and receives a response (14–16).

2.1.3 Handler method semantics

Most of the virtual flow_handler methods presented in Figure 2-3 have semantics similar to the corresponding non-blocking function calls in the C library. A handler class may override these methods to implement it specially, i.e., to change the behavior of the flow according to the session service the handler provides.

Downstream methods

The following methods are invoked by the handler's input flow. Since in general handlers will propagate these messages to their output flows (i.e., downstream), these methods are called *downstream methods* and can be thought of as providing an abstract flow service to the upstream handler.

- connect initiates a connection to the specified remote host. It returns zero if the connection is in progress, or a nonzero value if the connection is known immediately to have failed.

- bind binds the socket to a local address. It returns zero on success.

- accept returns a newly created flow encapsulating a new connection; it also returns the address from which the connection was accepted. accept is generally invoked immediately in response to an accept_ready callback (see below).

  A handler overriding the accept method typically uses its downstream handler's accept method to actually accept a flow, then instantiates itself with the new downstream handler set to the accepted flow. This effectively makes a deep copy of the network stack from the bottom up.

  accept returns the newly created handler, and the peer address of the accepted connection. It returns a null handler/address pair if it fails.

- write outputs bytes to the connection, returning true on success; for precise semantics, see the next section.

- shutdown closes the socket for reading and/or writing. It returns zero on success.

- listen begins listening for up to backlog connections on the socket, returning zero on success.

- getsockname and getpeername return the local and remote addresses of the connection, respectively. Each returns a null address on failure. (A null address is false when coerced to a boolean; thus one can test for connection using a construct such as the following:

  ```
  address = h->getsockname();

  if (address) {
     // Connected
  } else {
     // Not connected
  }
  ```

- may_avail informs the handler whether or not it may make any more data available to its upstream handler. This method is further discussed in Section 2.1.5.

Upstream (callback) methods

The following methods are invoked by the handler's output flows; each has a from argument which specifies which handler is invoking the method. Since typically handlers will propagate these messages to their input flow (i.e., upstream), these methods are called *upstream methods* and can be thought of callbacks which are invoked by the upstream handler.

- connected is invoked when a connection request (i.e., a connect method invocation) on a downstream has completed. The argument is true if the request succeeded or false if not.

- accept_ready is invoked when listen has been called and a remote host attempts to establish a connection. Typically a flow handler will respond to this by calling the downstream flow's accept method to accept the connection.

- avail is invoked when a downstream handler has received bytes. It returns false if the upstream handler is no longer able to receive bytes (i.e., the connection has shut down for reading).

We further discuss the input and output primitives, timers, and blocking in the next few sections. See Appendix B for the complete flow_handler API.

2.1.4 Handler input/output semantics

Since data input and output are the two fundamental operations on a flow, we shall describe them in more detail. The write method call writes bytes to the flow. It returns a boolean indicating success, unlike the C library's write call which returns a number of bytes or an EAGAIN error message (we will explain this design decision shortly). Our interface lacks a read method; rather, we use a callback, avail, which is invoked by a downstream handler whenever bytes are available. The upstream handler handles the bytes immediately, generally by performing some processing on the bytes and passing them to its own upstream handler.

A key difference between TESLA flow handlers' semantics and the C library's I/O semantics is that, in TESLA, write*s and* avail*s are guaranteed to complete.* In particular, a write or avail call returns false, indicating failure, only when it will never again be possible to write to or receive bytes from the flow (similar to the C library returning 0 for write or read). Contrast this to the C library's write and read function calls which (in non-blocking mode) may return an EAGAIN error, requiring the caller to retry the operation later. Our semantics make handler implementation considerably easier, since handlers do not need to worry about handling the common but difficult situation where a downstream handler is unable to accept all the bytes it needs to write.

Consider the simple case where an encryption handler receives a write request from an upstream handler, performs stream encryption on the bytes (updating its state), and then attempts to write the encrypted data to the downstream handler. If the downstream handler could return EAGAIN, as in an earlier design of TESLA, the handler must buffer the unwritten encrypted bytes, since by updating its stream encryption state the handler has "committed" to accepting the bytes from the upstream handler. Thus the handler would have to maintain a ring buffer for the unwritten bytes and register a callback when the output flow is available for writing. This would make the handler significantly more complex.

Our approach (guaranteed completion) benefits from the observation that given upstream and downstream handlers that support guaranteed completion, it is easy to write a handler to support guaranteed completion.[1] This is an inductive argument, of course—there must be some blocking mechanism in the system, i.e., completion cannot be guaranteed everywhere. Our decision to impose guaranteed completion on handlers isolates the complexity of blocking in the implementation of TESLA itself (as described in Section 4.2), relieving handler authors of having to deal with I/O multiplexing.

TESLA handlers are asynchronous and event-driven—all method calls are expected to return immediately—but, in the next section, we also present mechanisms to register timers and simulate blocking.

2.1.5 Timers and blocking

So far we have not discussed any way for flow handlers to prevent data from flowing upstream or downstream through them, i.e., to block. Clearly some blocking mechanism is required in flow handlers—otherwise, in an application where the network is the bottleneck, the application could submit data to TESLA faster than TESLA could ship data off to the network, requiring TESLA to buffer potentially large amounts of data. Similarly, if the application were the bottleneck, TESLA would be required to buffer all the data flowing upstream from the network until the application was ready to process it.

To the list of virtual methods in flow_handler we now add may_write and may_avail.

[1]The inverse—"given upstream and downstream handlers that do not support guaranteed completion, it is easy to write a handler that does not support guaranteed completion"—is not true, for the reason described in the previous paragraph.

A flow handler may ask its input flow to stop sending it data by invoking may_write(false); it can later restart the handler by invoking may_write(true). Likewise, a handler may throttle an output flow, preventing it from calling avail, by invoking may_avail(false). This does not seriously interfere with our guaranteed-completion semantics, since a handler which (like most) lacks a buffer in which to store partially-processed data can simply propagate may_writes upstream and may_avails downstream to avoid receiving data that it cannot handle.

A handler may need to register a time-based callback, e.g., to re-enable data flow from its upstream or downstream handlers after a certain amount of time has passed. For this we provide the timer abstract class. A handler may define a subclass of timer that overrides the fire method, then instantiate the subclass and arm it to instruct the TESLA event loop to invoke it at a particular time. For convenience we provide a subclass of timer, method_timer, which invokes an object method (generally a handler method) rather than requiring the handler to create its own timer subclass. See Figure 2-6 for the timer API and Section 3.2 for an example of how timers and the may_write and may_avail methods may be used to implement traffic shaping.

2.1.6 Default method implementations

Many handlers, such as the encryption handler, perform only simple processing on the data flow, have exactly one output flow for one input flow, and do not modify the semantics of other methods such as connect or accept. For this reason we provide a default implementation of each method which simply propagates method calls to the upstream handler (in the case of upstream methods, listed in Figure 2-4) or the downstream handler (in the case of of downstream methods, listed in Figure 2-3).

The flow_handler class provides the protected upstream and downstream instance variables, presented in Figure 2-3, for use by the default implementations in flow_handler. Figure 2-7 illustrates the semantics of the default avail and write. These are representative of the semantics for the default implementations of the upstream and downstream methods, respectively. The default methods only work for handlers with exactly one output flow (hence the assert statements).

```
class timer {
  public:
    enum from { /* ... */ };
    static const from FROM_EPOCH, FROM_NOW;
    static const long long CLEAR;

    // Arm the timer for <when> microseconds from now or from the epoch.
    // Use CLEAR to reset the timer.
    void arm(long long when = CLEAR, from what_base = FROM_NOW);
    void arm(struct timeval when, from what_base = FROM_NOW);

  protected:
    // Called by the TESLA event loop when the timer expires.
    virtual void fire() = 0;
};

// A convenient subclass of timer which invokes
// obj->method(arg, *this) when it fires.
template <class H, class T = int>
class method_timer : public timer {
  public:
    typedef void(H::*M)(method_timer<H,T>&);

    method_timer(H* obj, M method, T arg = T());
    T& arg();
};


// An example use
class foo_handler {
  public:
    typedef method_timer<foo_handler, int> my_timer;
    my_timer timer1;

    foo_handler() : timer1(this, &foo_handler::go, 0) {
        // Call one second from now
        timer1.arm(1000000LL);
    }

    void go(my_timer& t) {
        // Increment counter t.arg()
        cerr << "Invocation #" << ++t.arg() << endl;

        // Call again five seconds from now
        t.arm(5000000LL);
    }
};
```

Figure 2-6: The API for timer and method_timer, and an example usage. TESLA will invoke go(timer1) on the foo_handler one second (1,000,000 μs) after the handler is created and every five seconds thereafter.

```
virtual bool flow_handler::write(data bytes)
{
    // Other downstream methods (listen, connect, getpeername, etc.)
    // look just like this.
    assert(downstream.size() == 1);
    return downstream[0].write(bytes);
}
virtual bool flow_handler::avail(data bytes)
{
    // Other upstream methods (accept_ready, etc.) look just like this.
    assert(upstream && downstream.size() == 1 && from == downstream[0]);
    return upstream.avail(bytes);
}
```

Figure 2-7: The semantics of default flow_handler methods.

2.2 Process management

The issue of how handlers map on to processes and whether they run in the same address space as the application using the session services involves certain trade-offs. If each flow handler (session service module) were run in the same address space as the application generating the corresponding flow, the resulting performance degradation would be rather small. Furthermore, this approach has the attractive side effects of ensuring that the flow handler has the same process priority as the application using it and there are no inter-handler protection problems to worry about.

Unfortunately, this approach proves problematic in two important cases. First, there are session services (e.g., shared congestion management as in CM) that require state to be shared across flows that may belong to different application processes, and making each flow handler run linked with the application would greatly complicate the ability to share session-layer state across them. Second, significant difficulties arise if the application process shares its flows (i.e., the file descriptors corresponding to the flows) with another process, either by creating a child process through fork or through file descriptor passing. Ensuring the proper sharing semantics becomes difficult and expensive. A fork results in two identical TESLA instances running in the parent and child, making it rather cumbersome to ensure that they coordinate correctly. Furthermore, maintaining the correct semantics of asynchronous I/O calls like select and poll is rather difficult in this model. We have implemented a version

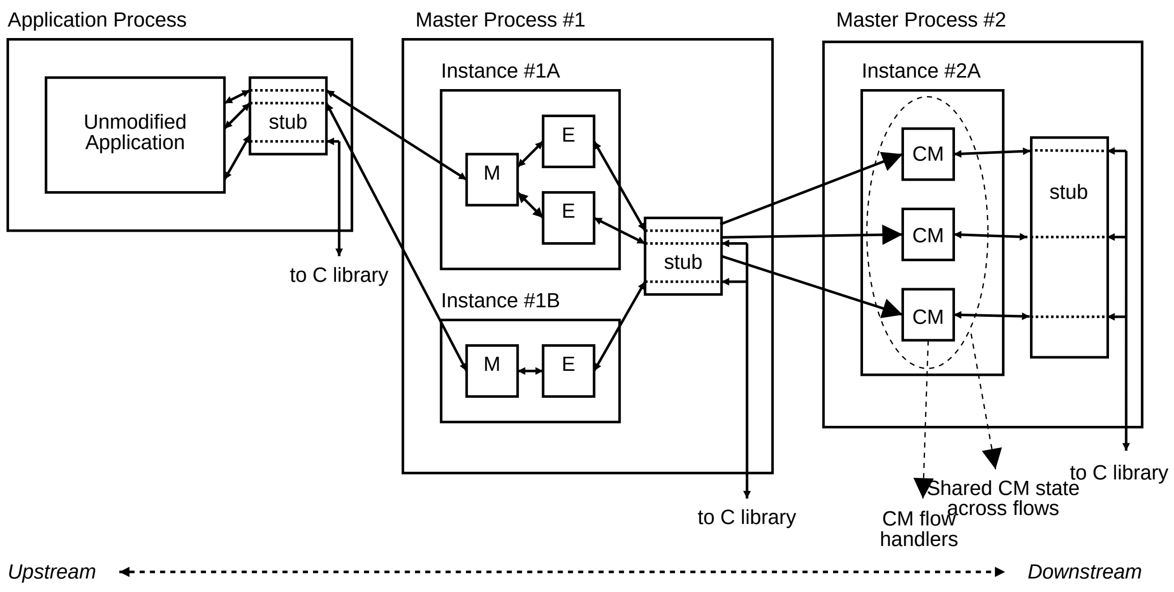


Figure 2-8: Possible inter-process data flow for an application running under TESLA. M is the migration handler, E is encryption, and CM is the congestion manager.

of TESLA that uses this architecture, which we describe in Appendix A.

A solution to this, of course, is to separate the context in which TESLA flow handlers run from the application processes that generate the corresponding flows. In this model, an application running through the tesla wrapper, and hence linked against the stub, is called a TESLA-*enabled application*. When such an application is invoked, the stub library creates or connects to a *master process*, a process dedicated to executing handlers.

Each master process has a particular *handler configuration*, an ordered list of handlers which will be used to handle flows. For instance, the configuration for master process #1 in Figure 2-8 is "TCP migration; TCP encryption." TCP flows through this particular master process will be migration-enabled and encrypted. Master process #2 is configured only for congestion management. When the application establishes a connection, the master process determines whether the configuration contains any applicable handlers. If so, the master instantiates the applicable handlers and links them together in what we call a TESLA *instance*.

Every master process is forked from a TESLA-enabled application, so master processes are also linked against the TESLA stub. Once a master process is forked, the application and master process communicate via a TESLA-internal socket for each application-level flow, and all actual network operations are performed by the dedicated TESLA process. When the application invokes one of the socket operations listed in Figure 4-1, TESLA traps it and converts it to a message which is transmitted to the master process via the master socket. The master process must return an immediate response (this is possible since all TESLA I/O is non-blocking).

In this fashion TESLA instances can be chained, as in Figure 2-8: an application-level flow may be connected to an instance (#1A) in a master process, which is in turn connected via its stub to an instance in another master process (#2A). Chaining enables some handlers to run in a protected, user-specific context, whereas handlers which require a system-scope (like system-wide congestion management) can run in a privileged, system-wide context.

In particular, UNIX semantics dictate that child processes inherit the same characteristics as thier parent, hence master processes receive equal privilege, scheduling priority, and protection as the initial application process. Further, when file descriptors (flows) are passed between TESLA-enabled applications, the handlers continue to run in the scope of their original master process. Hence, applications may interact with multiple master processes: their original master process and those associated with any file descriptors (flows) passed to it by other TESLA-enabled applications.

Ideally, TESLA could start in internal mode and move to master-process mode if flows ever become shared (e.g., because of a fork). While we have implemented an in-process version of TESLA (described in Appendix **??**), we have not yet implemented this dynamic optimization.

2.3 Interposition

To achieve the goal of application transparency, TESLA can be configured as an interposition agent at the C-library level. When a dynamically-linked application is run on an operating system like Linux, the dynamic linker is called to load the shared libraries which the program needs to function. Such libraries typically include implementations of system-wide services such as compression (e.g., zlib, or libz), graphic functions (e.g., libX11), user interface libraries (e.g., libgtk), and most notably for our purposes, the C library libc, which provides implementations of standard C functions (including interfaces to network socket and I/O functions).

However, before the linker loads libraries such as the C library, it checks the LD_PRELOAD environment variable, loading any shared libraries named therein. Function definitions in these preloaded libraries take first precedence. TESLA is activated by placing the *stub library*, libtesla.so, in LD_PRELOAD. A simple wrapper program, tesla, is used to set up the environment and invoke an application, e.g.:

```
tesla +encrypt -key secring.gpg +migrate telnet beacon
```

This would open a telnet connection to the host named beacon, with encryption (using secring.gpg as the private key) and end-to-end migration enabled.

We chose to keep TESLA invocation explicit, but another straightforward approach would be to put libtesla.so in one's LD_PRELOAD all the time (e.g., in system-wide or user-specific shell initialization scripts) and introducing a configuration file (perhaps /etc/tesla.cf, or .tesla.cf in the user's home directory) to control when TESLA is enabled.

2.4 Security Considerations

Like most software components, flow handlers can potentially wreak havoc within their protection context if they are malicious or buggy. Our chaining approach allows flow handlers to run within the protection context of the user to minimize the damage they can do; but in any case, flow handlers must be trusted within their protection contexts. Since a malicious flow handler could potentially sabotage all connections in the same master process, any flow handlers in master processes intended to have system-wide scope (such as a Congestion Manager handler) must be trusted.

Currently setuid and setgid applications, which assume the protection context of the binary iself rather than the user who invokes them, may not be TESLA-enabled:

one cannot trust a user-provided handler to behave properly within the binary's protection context.

Even if the master process (and hence the flow handlers) run within the protection context of the user, user-provided handlers might still modify network semantics to change the behavior of the application. Consider, for example, a setuid binary which assumes root privileges, performs a lookup over the network to determine whether the original user is a system administrator and. If a malicious user provides flow handlers that spoof a positive response for the lookup, the binary may be tricked into granting root privileges to the malicious user.

Nevertheless, many widely-used applications (e.g., ssh [15]) are setuid, so we do require some way to support them. We can make the tesla wrapper setuid root, so that it is allowed to add libtesla.so to the LD_PRELOAD environment variable even for setuid applications. The tesla wrapper then invokes the requested binary, linked against libtesla.so, with the appropriate privileges. libtesla.so creates a master process and begins instantiating handlers *only* once the process resets its effective user ID to the user's real user ID, as is the case with applications such as ssh. This ensures that only network connections within the user's protection context can be affected (maliciously or otherwise) by handler code.

Chapter 3

Example handlers

This chapter describes a few flow handlers we have implemented. Our main goal is to illustrate how non-trivial session services can be implemented easily with TESLA, showing how its flow-oriented API is both convenient and useful. We describe our handlers for transparent use of a SOCKS proxy, traffic shaping, encryption, compression, and end-to-end flow migration.

The compression, encryption, and flow migration handlers require a similarly configured TESLA installation on the peer endpoint, but the remaining handlers are useful even when communicating with a remote application that is not TESLA-enabled.

3.1 SOCKS and application-level routing

Our SOCKS handler is functionally similar to existing transparent SOCKS libraries [9, 22], although its implementation is significantly simpler. As we alluded in our description of Figure 2-5, it overrides the `connect` handler to establish a TCP connection to a proxy server, rather than a TCP or UDP connection directly to the requested host. When its `connected` method is invoked, it does not pass it upstream, but rather negotiates the authentication mechanism with the server and then passes it the actual destination address, as specified by the SOCKS protocol.

If the SOCKS server indicates that it was able to establish the connection with the remote host, then the SOCKS handler invokes `connected(true)` on its upstream handler; if not, it invokes connected(false). The upstream handler, of course, is never aware that the connection is physically to the SOCKS server (as opposed to the destination address originally provided to connect). We provide pseudocode for our SOCKS server (*sans* error handling) in Figure 3-1.

We plan to utilize the SOCKS server to provide transparent support for Resilient Overlay Networks [3], or RON, an architecture allowing a group of hosts to route around failures or inefficiencies in a network. We will provide a ron_handler to allow a user to connect to a remote host transparently through a RON. Each RON server exports a SOCKS interface, so ron_handler can use the same mechanism as socks_handler to connect to a RON host as a proxy and open an indirect connection to the remote host. ron_handler also utilizes the end-to-end migration of migrate_handler (presented below in Section 3.5) to enable RON to hand off the proxy connection to a different node if it discover a more efficient route from the client to the peer.

3.2 Traffic shaping: timeouts and throttling

It is often useful to be able to limit the maximum throughput of a TCP connection. For example, one might want prevent a background network operation such as mirroring a large software distribution to impact network performance. We had always had the goal of providing rate control as an interesting session service, so we took special interest in the following email message sent to our group by one of our colleagues:

> Subject: Slow file distribution?
> To: nms@lcs.mit.edu
> Date: Sun, 30 Sep 2001 13:47:46 -0400 (EDT)
>
> Before I start hacking 'sleep' statements into rsync, can anyone think of any very-network-friendly file distribution programs?
>
> I have to transfer some decently-sized logfiles from the RON boxes back to MIT, and I'd like to do so without impacting the network much at any given time, so that I don't have a synchronized slowing-down of the entire RON network that'll affect my measurements.

```
class socks_handler : public flow_handler {
    int state;
    address dest;
    method_timer<socks_handler> timer1;

  public:
    socks_handler(...) : timer1(this, &socks_handler::timeout, 0) {
        timer1.arm(10000000LL); // timeout in 10 seconds
    }

    int connect(address a) {
        dest = a;
        downstream[0]->connect(proxy server address);
    }
    void connected(flow_handler *from, bool success) {
        if (!success) {
            upstream->connected(this, false);
            return;
        }
        downstream[0]->write(supported authentication mechanisms);
        state = HELLO;
    }
    bool avail(flow_handler *from, string data) {
        if (state == ESTABLISHED) return upstream->avail(this, data);
        if (state == HELLO) {
            // Select an authentication mechanism based on data
            downstream[0]->write(...);
            state == AUTHENTICATING;
            return true;
        }
        if (state == AUTHENTICATING) {
            if (data indicates that authentication failed) {
                upstream->connected(this, false);
                return false; // I'm closed for reading!
            }
            downstream[0]->write(connect to <dest>);
            state = CONNECTING;
            return true;
        }
        if (state == CONNECTING) {
            if (data indicates that connecting to remote host succeeded) {
                state = ESTABLISHED;
                upstream->connected(this, true);
            } else
                upstream->connected(this, false);
        }
    }
    void timeout(method_timer<foo_handler>& t) {
        upstream->connected(this, false);
    }
    // no need to implement write; just pass it through to downstream[0]
};
```

Figure 3-1: Pseudocode for a SOCKS handler (lacking error-handling).

Anything come to mind?

This is an excellent example of functionality that can be provided using a session-layer service, but one that needs to be done transparent to the end application (in this case, RON clients). We developed a traffic shaping handler to provide this functionality.

The traffic shaper keeps count of the number of bytes it has read and written during the current 100-millisecond timeslice. If a write request would exceed the outbound limit for the timeslice, then the portion of the data which could not be written is saved, and the upstream handler is throttled via may_write. A timer is created to notify the shaper at the end of the current timeslice so it can continue writing and unthrottle the upstream handler when appropriate. Similarly, once the inbound bytes-per-timeslice limit is met, any remaining data provided by avail is buffered, downstream flows are throttled, and a timer is registered to continue reading later.

Figure 3-2 provides code for a shaper which restricts outbound traffic only. Here we introduce the flow_handler::init_context class, which encapsulates the initialization context of a flow handler, including user-provided parameters (in this case, timeslice length and bandwidth limit).

3.3 Triple-DES encryption/decryption

crypt_handler is a triple-DES encryption handler for TCP streams. We use OpenSSL [10] to provide the DES implementation. Figure 3-3 shows how crypt_handler uses the TESLA flow handler API. Note how simple the handler is to write: we merely provide alternative implementations for the write and avail methods, routing data first through a DES encryption or decryption step (des3_cfb64_stream, a C++ wrapper we have written for OpenSSL functionality).

```
class shape_handler : public flow_handler {
    int timeslice;           // 100000 = 100 ms
    int limit_per_timeslice; // 1024 = 10 KB/s

    method_timer<shape_handler> reset;
    int limit_this_timeslice; // bytes we can still write in this timeslice

    string buffer;

  public:
    shape_handler(init_context& ctxt) :
      flow_handler(ctxt), reset(this, &handle_reset)
    {
        // Read configuration
        timeslice = ctxt.int_arg("timeslice", 100000);
        limit_per_timeslice = ctxt.int_arg("limit_per_timeslice", 1024);
        // Initialize per-timeslice state and timer
        limit_this_timeslice = limit_per_timeslice;
        reset.arm(timeslice);
    }

    bool write(data d) {
        if (limit_this_timeslice >= d.length()) {
            // just write the whole thing
            limit_this_timeslice -= d.length();
            return downstream[0]->write(d);
        }

        // Write whatever we can...
        downstream[0]->write(data(d.bits(), limit_this_timeslice));

        // ...and buffer the rest.
        buffer.append(d.bits() + limit_this_timeslice,
                      d.length() - limit_this_timeslice);

        // Don't accept any more bytes!
        upstream->may_write(this, false);

        return true;
    }

    void reset(method_timer<shape_handler> t) {
        t.arm(timeslice);
        limit_this_timeslice = limit_per_timeslice;

        if (buffer.length() == 0) return;

        if (buffer.length() <= limit_this_timeslice) {
            // Clear the buffer!
            downstream[0]->write(data(buffer.data(), buffer.length()));
            buffer.resize(0);
            // Now accept more bytes for writing.
            upstream->may_write(this, true);
        } else {
            // Write whatever we can, and buffer the rest.
            downstream[0]->write(data(buffer.data(), limit_this_timeslice));
            buffer = string(buffer, limit_this_timeslice,
                            buffer.length() - limit_this_timeslice);
            limit_this_timeslice = 0;
        }
    }
};
```

Figure 3-2: Complete code for an outbound traffic shaper.

```
class crypt_handler : public flow_handler {
    des3_cfb64_stream in_stream, out_stream;

  public:
    crypt_handler(init_context& ctxt) : flow_handler(ctxt)
      in_stream(des3_cfb64_stream::ENCRYPT),
      out_stream(des3_cfb64_stream::DECRYPT)
    {}

    bool write(data d) {
        // encrypt and pass downstream
        return downstream[0].write(out_stream.process(d));
    }

    bool avail(flow_handler *from, data d) {
        // decrypt and pass upstream
        return upstream.avail(this, in_stream.process(d));
    }
};
```

Figure 3-3: Complete code for a transparent triple-DES CFB encryption/decryption layer.

3.4 Compression

Initially our compression handler was very similar to our encryption handler: we merely wrote a wrapper class (analogous to des3_cfb64_stream) for the freely available zlib stream compression library. Noting that many useful handlers can be implemented simply by plugging in an appropriate stream class, we decided to generalize crypt_handler into a templatized stream_handler. Figure 3-4 presents complete code for zlib_handler.

3.5 Session migration

We have used TESLA to implement transparent support in the Migrate session-layer mobility service [33]. In transparent mode, Migrate preserves open network connections across changes of address or periods of disconnection. Since TCP connections are bound to precisely one remote end point and do not survive periods of disconnection in general, Migrate must synthesize a logical flow out of possibly multiple physical connections (a new connection must be established each time either end-

```
template <class T>
class stream_handler : public flow_handler {
  protected:
    T input_stream;
    T output_stream;

    stream_handler(init_context& ctxt) : flow_handler(ctxt) {}

  public:
    bool write(data d) {
        data out = output_stream.process(d);
        bool success = downstream[0].write(out);

        // allow stream to free any internal buffers it may have
        // needed to allocate
        output_stream.release(d);

        return success;
    }

    bool avail(data d) {
        data in = input_stream.process(d);
        bool success = upstream.avail(this, in);
        input_stream.release(in);
        return success;
    }
};

class zlib_handler : public stream_handler<zlib_stream> {
  public:
    zlib_handler(init_context& ctxt) :
        stream_handler(ctxt),
        input_stream(zlib_stream::DEFLATE),
        output_stream(zlib_stream::INFLATE)
    {}
};
```

Figure 3-4: Complete code for stream_handler, a "convenience" handler simplying implementation of handlers which perform straightforward transformations on data. We use stream_handler to implement zlib_handler, our compression handler. transparent compression layer

point moves or reconnects). Further, since data may be lost upon connection failure, Migrate must double-buffer in-flight data for possible re-transmission.

This basic functionality is straightforward to provide in TESLA. We simply create a handler that splices its input flow to an output flow, and conceals any mobility events from the application by automatically initiating a new output flow using the new end-point locations. The handler stores a copy of all outgoing data locally in a ring-buffer and re-transmits any lost bytes after re-establishing connectivity on a new output flow. Incoming data is not so simple, however. Because it is possible that received data has not yet been read by the application before a mobility event occurs, Migrate must instantiate a new flow immediately after movement, but continue to supply the buffered data from the previous flow to the application until it is completely consumed. Only then will the handler begin delivering data received on the subsequent flow(s).

Note that because Migrate wishes to conceal mobility when operating in transparent mode, it is important that the handler be able to override normal signaling mechanisms. In particular, it must intercept connection failure messages and prevent them from reaching the application, instead taking appropriate action (e.g., the CONNECTION RESET messages typically experienced during long periods of disconnection) to manage and conceal the changes in end points. In doing so, the handler also overrides the getsockname() and getpeername() calls to return the original endpoints, irrespective of the current location.

Chapter 4

Implementation

This chapter discusses some of the important implementation choices made in TESLA. We describe how we implement the TESLA stub, which, when dynamically linked against an application, captures the application's network input and output events and sends them to the master process. We discuss two special handlers, top_handler and bottom_handler, present in all master processes. top_handler martials events between the application and applicable flow handlers, and bottom_handler martials events between the flow handlers and the underlying network streams. Finally, we describe how master processes' event loops are structured to handle blocking input and output flexibly, and examine the performance overhead of our implementation.

System call	Possible responses	
socket	$\langle$ "filehandle", $f\rangle$	$\langle$ "nak" $\rangle$
connect	$\langle$ "ack" $\rangle$	$\langle$ "nak" $\rangle$
bind	$\langle$ "ack" $\rangle$	$\langle$ "nak" $\rangle$
listen	$\langle$ "ack" $\rangle$	$\langle$ "nak" $\rangle$
accept	$\langle$ "filehandle", $f\rangle$	$\langle$ "nak" $\rangle$
getsockname	$\langle$ "address", $a\rangle$	$\langle$ "nak" $\rangle$
getpeername	$\langle$ "address", $a\rangle$	$\langle$ "nak" $\rangle$

Figure 4-1: Socket operations trapped by TESLA and converted to messages over the master socket.

4.1 Tesla stub implementation

When the application invokes the socket library call, the TESLA stub's wrapper for socket—in the application process—sends a message ⟨"socket", *domain*, *type*⟩ to the master process. (*domain* and *type* are the arguments to socket, e.g., AF_INET and SOCK_STREAM respectively.) If the master process has no handlers registered for the requested domain and type, it returns a negative acknowledgement and the application process uses the socket system call to create and return the request socket.

If, on the other hand, there is a handler class (a subclass of handler, declared in Figure 2-3) registered for the requested domain and type, the master process instantiates the handler class. Once the handler's constructor returns, the master process creates a pair of connected UNIX-domain sockets. One is retained in the master process, and one is passed to the application process and closed within the master process. The application process notes the received filehandle (remembering that it is a TESLA-wrapped filehandle) and returns it as result of the socket call.

The application process handles operation on wrapped filehandles specially. The TESLA library overloads connect, bind, listen, accept, getsockname, and getpeername with *wrapper functions*. When the application invokes any of these calls on a TESLA-wrapped filehandle, the wrapper function sends a message to the master process, which invokes the corresponding method on the handler object for that flow. Based on the return result of the method, the master process returns a message to the client as in Figure 4-1. The client process receives the message, returning the result to the application.

4.1.1 Reading, writing, and multiplexing

The TESLA stub does not need to specially handle reads, writes, and multiplexing on wrapped sockets. The master process and application process share a socket for each application-level flow, so to handle a read, write, or select, whether blocking or non-blocking, the application process merely uses the corresponding unwrapped C library call. Even if the operation blocks, the master process can continue handling

I/O while the application process waits.

4.1.2 Connecting and accepting

The semantics for connect and accept are necessarily different than for other socket operations, since unlike listen, bind, etc., they may not return immediately, but instead cause a potentially long-running operation to begin in the master process.

How can the application process be notified when the connect or accept is complete (whether successfully or not)? For blocking operations, the application process could simply wait for a message on the master socket indicating completion. However, if the connect or accept is non-blocking, the response could be received on the master socket at any time. This would complicate our simple synchronous request/response model on the master socket: we would have to be prepared at any time to receive the result of a connect or accept operation.

Furthermore, we would have to provide some sort of wrapper around select and poll. If the application attempts to block on write availability to wait for a connect operation to complete, select will return immediately (since our logical filehandle is always connected and immediately ready for writing). If it blocks on read availability to wait for an accept operation to complete, select will never return a read event, since no bytes are ever exchanged on a "server socket" being used for an accept.

We use the following trick to circumvent this problem. Within the master process, we modify the semantics of accept to write a single byte to the logical server socket whenever a connection is accepted. Hence when the application blocks on read availability on the server socket, select naturally returns a read event whenever a connection has been accepted. Our accept wrapper in the client process consumes the byte and synchronously requests and receives the newly accepted logical socket from the master process.

We use a similar trick for connect, which always returns a single byte on the logical filehandle when a connecting is accepted by the master process. Our connect wrapper consumes the byte and notes that the connection is ready for reading and writing.

Unfortunately, this doesn't *quite* work for selecting on connect, since to detect connection success one selects on *writing* rather than *reading*, but the master returns a byte to be *read*. (Since the connection between the application and the master process is a connected UNIX-domain socket, there is no way to force the application to block on a write to emulate the desired behavior.) Therefore, we must still wrap the select system call to handle this special case. In general, our approach is still beneficial, because handling this special case is far easier than switching to an asynchronous model for application/master communication.

4.1.3 Forking and filehandle duplication

Implementing fork, dup, and dup2 is quite simple. We wrap the fork library call so that the child process obtains a new connection to the master process. Having a single logical flow available to more than one application process, or as more than one filehandle, presents no problem: application processes can use each copy of the filehandle as usual. Once all copies of the filehandle are shut down, the master process can simply detect via a signal that the filehandle has closed and invokes the handler's close method.

4.2 top_handler and bottom_handler

In Chapter 2 we described at length the interface between handlers, but we have not yet discussed how precisely handlers receives data and events from the application. We have stated that every handler has an upstream flow which invokes its methods such as connect, write, etc.; the upstream flow of the *top-most* handler for each flow (e.g., the encryption handler in Figures 2-2 and 2-5) is a special flow handler called top_handler.

When TESLA's event loop detects bytes ready to deliver to a particular flow via the application/master-process socket pair which exists for that flow, it invokes the write method of top_handler, which passes the write call on to the first real flow handler (e.g., encryption). Similarly, when a connect, bind, listen, or accept message appears on the master socket, the TESLA event loop invokes the corresponding method of the top_handler for that flow.

The dual of this process occurs when the top-most handler invokes a callback method (e.g., avail) of *its* upstream handler, the top_handler for the flow. In the case of avail, top_handler writes the data to the application; in the case of connected or accept_ready, it writes a single byte to the application as described in Section 4.1.2.

Similarly, each flow handler at the bottom of the TESLA stack has a bottom_handler as its downstream. For each non-callback method—i.e., connect, write, etc.—bottom_handler actually performs the corresponding network operation.

top_handlers and bottom_handlers maintain buffers, in case a handler writes data to the application or network, but the underlying filesystem buffer is full (i.e., sending data asynchronously to the application or network results in an EAGAIN condition). If this occurs in a top_handler, the top_handler requests via may_avail that its downstream flow stop sending it data using avail; similarly, if this occurs in a bottom_handler, it requests via may_write that its upstream flow stop sending it data using write.

4.3 The event loop

As with most event-driven programs, at TESLA's core is a select system call. TESLA blocks on the following events:

1. Data available on the master socket, i.e., a control message from the application.

2. Data available on a top_handler file descriptor, i.e., a write from the application.

3. Write readiness on a top_handler (application) socket if the top_handler has bytes which still need to be sent to the application, i.e., the last attempt to write to the application resulted in an EAGAIN condition.

4. Data available on a bottom_handler file descriptor, i.e., bytes ready to be read from the network.

5. Write readiness on a bottom_handler (network) socket if the bottom_handler has bytes buffered, i.e., the last attempt to write to the network resulted in an

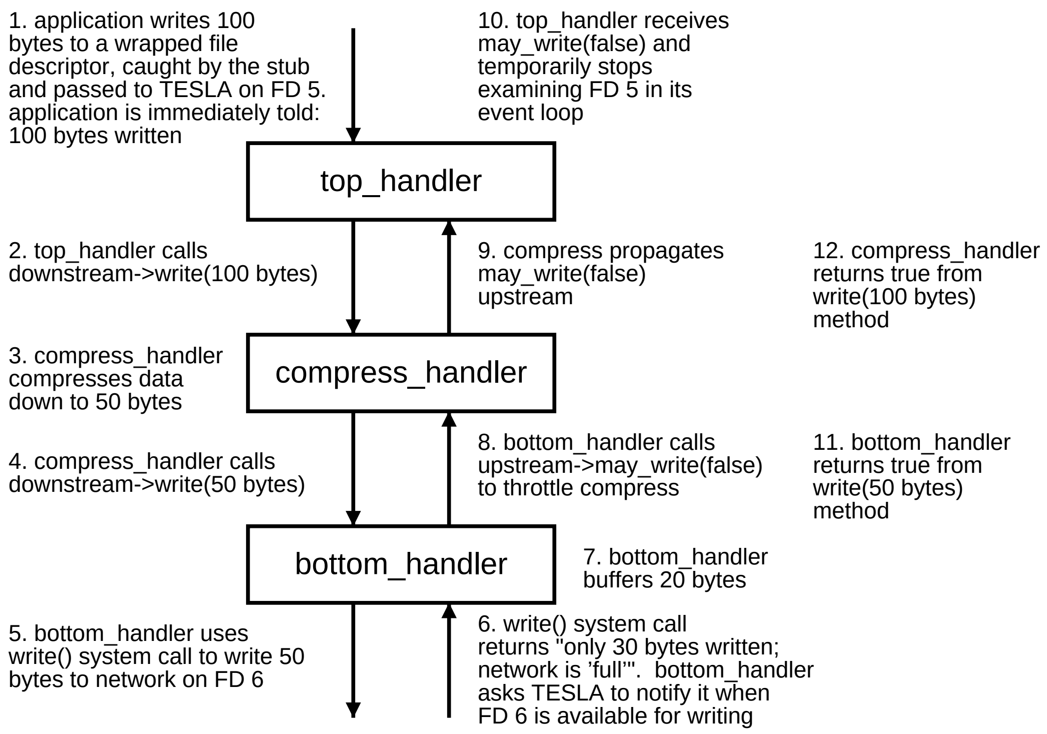
EAGAIN condition.

6. Expiration of timers (registered via a timer::arm call, as described in Section 2.1.5). TESLA maintains timers in a set structure sorted, decreasing with respect to time left until expiration, so it is a simple operation to set the select timeout to the time until the next timer expiration.

If a handler invokes may_write(h, false) on a top_handler which is its input flow, TESLA will suspend receiving data on this flow from the application, since there would be nowhere to pass along data written to it by the application. Eventually this may cause the application's writes to block. This behavior is desirable in, e.g., a traffic shaper (see Section 3.2), which calls may_write(*this*, false) to notify the upstream handler (and eventually TESLA) to stop writing, causing the application's flow to block. TESLA implements this behavior by simply removing the filehandle from the rfds input to the select call (case 2 above).

Likewise, if a handler invokes may_avail(false) on a bottom_handler which is its output flow, TESLA will suspend receiving data on this flow from the network, since there would be nowhere to pass along data received on the network. This feedback mechanism may cause the underlying transport mechanism (e.g., TCP) to slow down the connection from the peer. To implement this behavior, TESLA removes the filehandle from the rfds input to the select call (case 4 above).

Figure 4-2 demonstrates what may happen when an application performs a write of 100 bytes to a TESLA flow (compressed down to 50 bytes by the compression layer). TESLA receives the bytes from the application on file descriptor 3 (a UNIX-domain stream socket), but the underlying network connection (file descriptor 6, a TCP socket) can only accept 30 bytes to be sent. The bottom_handler buffers the remaining 20 bytes and requests to be notified by the TESLA event loop when it becomes possible to write to file descriptor 6. The bottom_handler then throttles the compression handler, which in turn throttles the top_handler, which causes TESLA to stop reading bytes on file descriptor 3. The bottom_handler still returns true (a successful write).



1. application writes 100 bytes to a wrapped file descriptor, caught by the stub and passed to TESLA on FD 5. application is immediately told: 100 bytes written
10. top_handler receives may_write(false) and temporarily stops examining FD 5 in its event loop
top_handler
2. top_handler calls downstream->write(100 bytes)
9. compress propagates may_write(false) upstream
12. compress_handler returns true from write(100 bytes) method
3. compress_handler compresses data down to 50 bytes
compress_handler
4. compress_handler calls downstream->write(50 bytes)
8. bottom_handler calls upstream->may_write(false) to throttle compress
11. bottom_handler returns true from write(50 bytes) method
bottom_handler
7. bottom_handler buffers 20 bytes
5. bottom_handler uses write() system call to write 50 bytes to network on FD 6
6. write() system call returns "only 30 bytes written; network is 'full'". bottom_handler asks TESLA to notify it when FD 6 is available for writing

Figure 4-2: An application writes to a network flow which cannot presently deliver all the bytes via the underlying transport protocol (a TCP stream on file descriptor 6).

4.4 Performance

When adding a network service to an application, we may expect to incur one or both of two kinds of performance penalties. First, the service's algorithm, e.g., encryption or compression, may require computational resources. Second, the interposition mechanism, if any, may involve overhead such as inter-process communication, depending on its implementation. We refer to these two kinds of performance penalty as *algorithmic* and *architectural*, respectively.

If the service is implemented directly within the application, e.g., via an application-specific input/output indirection layer, then the architectural overhead is probably minimal and most of the overhead is likely to be algorithmic. However, adding an interposition agent like TESLA outside the context of the application may add significant architectural overhead.

To analyze TESLA's architectural overhead, we constructed several simple tests to test how using TESLA affects latency and throughput. We compare TESLA's performance to that of a tunneling, or proxy, agent, where the application explicitly tunnels flows through a local proxy server.

We provide three benchmarks. rbandwidth(s) establishes a TCP connection to a server and reads data into a s-byte buffer until the connection closes. wbandwidth(s) is similar: it accepts a TCP connection and writes a particular s-byte string repeatedly until a fixed number of bytes have been sent. latency(s) connects to a server, sends it s bytes, waits for an s-byte response, and so forth, until a fixed number of exchanges have occurred.

The left graph in figure 4-3 shows the results from running rbandwidth under several configurations, both with and without a transparent network service (triple-DES stream encryption/decryption in CBC [chaining block cipher] mode, as implemented by the OpenSSL library [10]). We run all the benchmarks on a single machine to eliminate network performance as a variable. The first three configurations perform no encryption:

1. The benchmark program running unmodified.

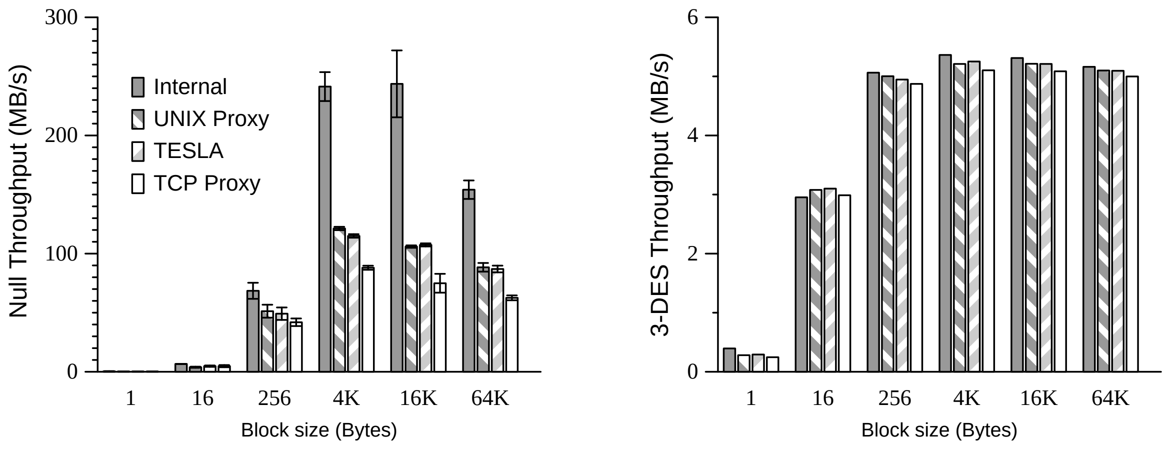


Figure 4-3: Results of the rbandwidth test. In the left graph, we show error bars at ±1 standard deviation; in the right graph, the variance was consistently very small. (wbandwidth yielded results similar to rbandwidth.)

2. The benchmark program, with each connection passing through a proxy server that performs no processing on the data. The benchmark program connects to the proxy server via a UNIX-domain socket, and the proxy contacts the server via TCP on behalf of the benchmark program.

3. The benchmark program running under TESLA with the dummy handler enabled. dummy is a trivial handler which performs no processing on the data.

4. The benchmark program, with each connection passing through a TCP proxy server that performs no processing on the data. The benchmark program connects to the proxy server via a TCP connection, and the proxy contacts the server via TCP on behalf of the benchmark program.

We can think of these configurations as providing a completely trivial transparent service, an "identity service" with no algorithm (and hence no algorithmic overhead) at all. Comparing the unmodified benchmark with the following three tests isolates, respectively, the architectural overhead of a UNIX-domain proxy server (i.e., an extra UNIX-domain socket stream through which all data must flow), TESLA (i.e., an extra UNIX-domain socket stream plus any overhead incurred by the master process) and a typical TCP proxy server (i.e., an extra TCP socket stream for all data), respectively.

The next four configurations, whose performance is illustrated in the right graph in Figure 4-3, provide a nontrivial service, triple-DES encryption:

1. A modified version of the benchmark, with triple-DES encryption and decryption performed directly within in the benchmark application.

2. The benchmark program, with each connection passing through a UNIX-domain socket stream proxy server. The proxy server performs encryption/decryption.

3. The benchmark program, running under TESLA, with the crypt (encryption/decryption) handler enabled.

4. The benchmark program, with each connection passing through a TCP proxy server. The proxy server performs encryption/decryption.

Comparing each of these tests with the corresponding benchmark above in the left graph illustrates the algorithmic overhead of the service.

The throughput graph shows that for trivial services (like the "dummy" service) using a proxy server or TESLA incurs a significant performance penalty compared to implementing the service directly within the application. This is due to the overhead of inter-process communication. In the first configuration without DES (#1), data flow directly from the server to the client, i.e., through a single socket. In the next three configurations (#2–3) data must flow through one UNIX-domain socket stream to the intermediary process, i.e., the TESLA master process or proxy server, and through another socket to the client. With two sockets instead of one, the kernel must handle twice as many system calls, and perform twice as many copy operations between address spaces. In the final configuration, data flow through two TCP streams; the extra overhead of IP/TCP processing (as opposed to a UNIX-domain socket stream, which requires none of the network-layer functionality of IP or transport-layer functionality of TCP) causes a small but noticable performance hit compared to configurations (#2 and 3).

In contrast, for nontrivial services such as encryption/decryption, the algorithmic overhead dominates the architectural overhead, and the performance differences

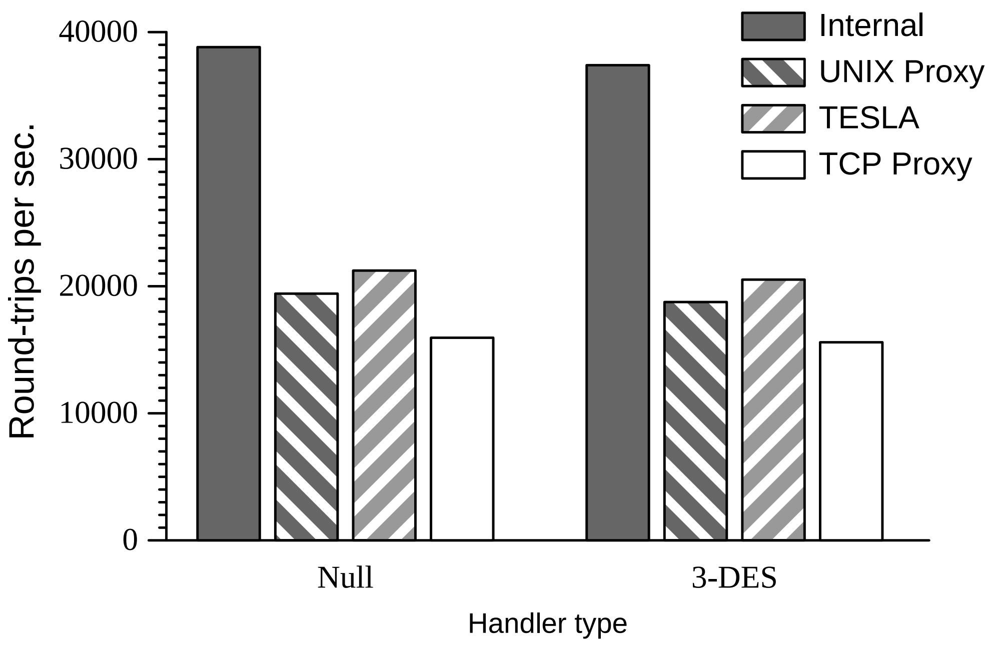


Figure 4-4: Results of the latency tests for a block size of one byte.

between the various configurations are barely noticeable. For large block sizes, the bottleneck is simply the speed at which the test computer can perform DES encryption. For small block sizes, the bottleneck is handling the 1-byte write system call in the application; the intermediary process (Tesla or the proxy) receives larger chunks of data at once, since bytes aggregate in the operating system's buffer between the application and the intermediary.

The latency benchmark yields different results: since data rates never exceed a few tens of kilobytes per second, the computational overhead of the algorithm itself is negligible and the slowdown is due exclusively to the architectural overhead. We see no difference at all between the speed of the trivial service and the nontrivial service. Again, the TCP proxy server is slower due to IP/TCP overhead in the kernel.

We conclude that when inter-process communication is a bottleneck, using a proxy server or Tesla as an interposition agent shows a marked slowdown. In the more typical case that the bottleneck lies elsewhere (in the transparent service's algorithm, or certainly in the network) then neither Tesla nor a proxy server incurs significant overhead.

Chapter 5

Conclusion

This thesis described the design and implementation of TESLA, a framework to implement session layer services in the Internet. These services are gaining in importance; examples include connection multiplexing, congestion state sharing, application-level routing, mobility/migration support, compression, and encryption.

TESLA incorporates three principles to provide a transparent and extensible framework: first, it exposes network flows as the object manipulated by session services, which are written as flow handlers without dealing with socket descriptors; second, it maps handlers on to processes in a way that provides for both sharing and protection; and third, it can be configured using dynamic library interposition, thereby being transparent to end applications.

We showed how TESLA can be used to design several interesting session layer services including encryption, SOCKS and application-controlled routing, flow migration, and traffic rate shaping, all with acceptably low performance degradation. In TESLA, we have provided a powerful, extensible, high-level C++ framework which makes it easy to develop, deploy, and transparently use session-layer services.

5.1 Future Work

Currently, to use TESLA services such as compression, encryption, and migration that require TESLA support on both endpoints, the client and server must use identical configurations of TESLA, i.e., invoke the tesla wrapper with the same handlers specified on the command line. We plan to add a negotiation mechanism so that once mutual TESLA support is detected, the endpoint can dynamically determine the configuration based on the intersection of the sets of supported handlers. We also plan to add per-flow and configuration so that the user can specify which handlers to apply to flows based on the flow attributes such as port and address.

We plan to continue developing TESLA flow handlers such as Congestion Manager. We are also developing a handler which has no output flows but rather connects to a network simulator, facilitating the use of real applications to test protocols and network topologies in simulators such as ns [16]. Finally, we plan to explore the possibility of moving some handler functionality directly into the application process to minimize TESLA's architectural overhead.

Appendix A

Alternate Design: In-process Flow Handlers

In our initial implementation, the TESLA library operated entirely within the context of the wrapped application's process. The major theoretical advantage of this approach is that data need not flow through an extra process intermediating between the application and the network. However, we find that this method makes it difficult to maintain the correct filehandle semantics for many input/output and process control system calls. Further, it compromises our goals of enabling information sharing between flows.

In this design, application-level I/O calls such as read, write, connect, and select are intercepted by TESLA and routed directly to the first applicable handler on the stack. For instance, in an encryption layer, a single application-level read call might be intercepted by TESLA and routed to the encryption handler. The encryption handler's implementation of read, shown in Figure A-1, reads a chunk of data from the output flow (using the TS_NEXT macro, which delegates control to the next handler on the stack or to the C library if there are no more handlers), and then decrypt the results in place. These steps are all performed within the context of the application process, so there is no additional latency penalty due to interprocess communication.

An in-process architecture has several significant drawbacks, however. First, when an application invokes the fork system call, each open file descriptor, as viewed by

```
int crypt_read(int fd, char *buffer, int length,
               ts_context ctxt)
{
    /* TS_DATA macro obtains connection state. */
    crypt_state *state = TS_DATA;

    /* TS_NEXT invokes the next handler. */
    int bytes = TS_NEXT(read, fd, buffer, length);
    if (bytes > 0)
        decrypt_data_inplace(buffer, bytes,
            &state->decryption_state);

    return bytes;
}
```

Figure A-1: The read method for a simple encryption/decryption handler using the in-process implementation of TESLA.

the application, becomes available to both the parent and child processes. Since the parent and child would share application-level flows, the TESLA handlers in the two processes would need to coordinate. We could have been supported this case by shuttling handler state between parent and child processes, or retaining a pipe between the parent and child so each would have access to the handler stack, but either method would have added significant complexity and eliminated any performance advantage.

Second, it can be difficult to share information between handlers running in different processes. While related handlers in different processes (e.g., multiple Congestion Manager handlers aggregating congestion handling for several flows) could certainly use a typical interprocess communication mechanism such as pipes, sharing state is far easier if the handlers coexist in the same address space—this way handlers can simply share access to high-level (C or C++) data structures and application logic.

Third, we found it extremely difficult to preserve certain semantics of file descriptors in an efficient way. To illustrate the problem, we will consider a simple application using asynchronous I/O and multiplexing to establish a connection to a server. It would typically:

1. Call socket, and use fcntl to make the socket non-blocking. Let $f =$ the file descriptor returned by socket.

2. Call connect to establish a connection to the server.

3. Enter a multiplexing loop containing a select call. Include f in writefds, the list of file descriptors on which select is waiting for a write to become available.

4. When select returns with f set in writefds, note that the connection has been established, and begin writing.

However, consider how TESLA must behave when the stack contains a handler to route traffic through a SOCKS proxy. SOCKS requires an initial setup phase where the client and server negotiate an authentication mechanism, if any. The client must read the list of supported mechanisms from the server; this implies that a TESLA SOCKS handler may be required to *read* from a output flow (the flow from the SOCKS handler to the proxy server, which we will refer to as g) even though the application has requested to be notified of *write* availability on the output flow (the flow from the application to the SOCKS handler, f above).

To emulate the select and poll C library calls in a fully general way, we split the functions into several steps:

1. For each input flow which the application has included in the lists of read, write, or error events it is interested in, call an rwselect ("rewrite select") handler to translate events on the input flow to the corresponding output flows. For instance, a SOCKS proxy handler in the authentication negotiation phase might convert a **input write event** on f into a **output read event** on g. If there is another handler beneath g on the network stack, this handler would have the opportunity to further convert the read event on g, which is a input write event from its standpoint, into one or more output events on *its* output flows. Select rewriting is illustrated in Figure A-2.

2. Call the select system call with the set of events obtained via this transformation.

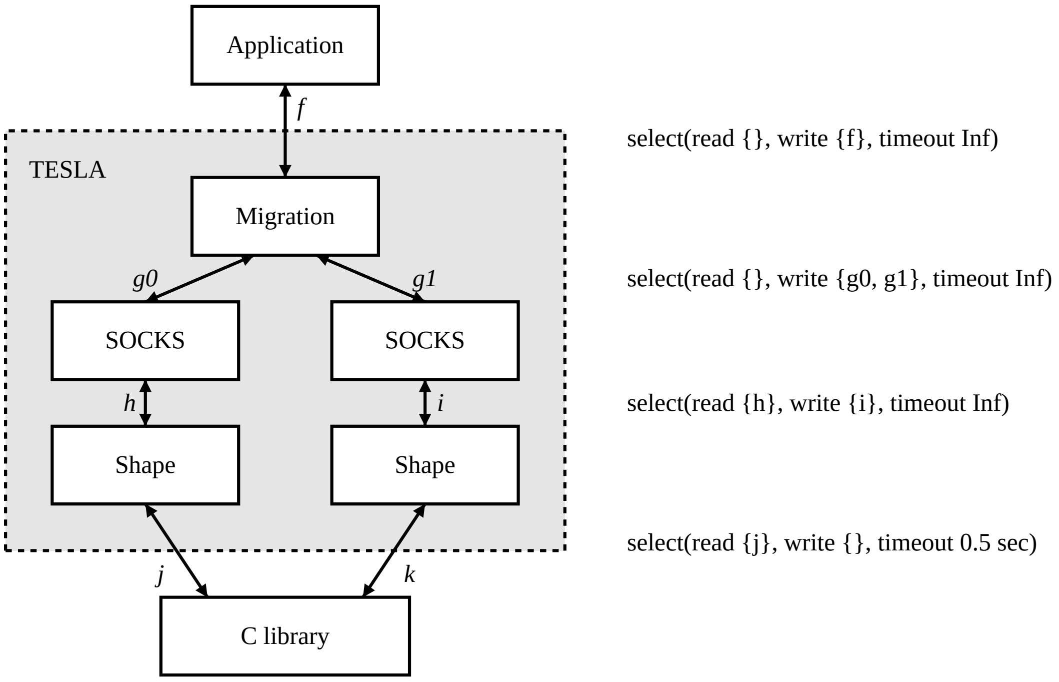

Application
f
TESLA
Migration
g0
g1
SOCKS
SOCKS
h
i
Shape
Shape
j
k
C library
select(read {}, write {f}, timeout Inf)
select(read {}, write {g0, g1}, timeout Inf)
select(read {h}, write {i}, timeout Inf)
select(read {j}, write {}, timeout 0.5 sec)

Figure A-2: An example of select rewriting. Flow h is in the authentication negotiation phase of the SOCKS protocol, so the input write event on g_0 is transformed into a read event on h. Flow k has used its bandwidth quota for the current timeslice, so the traffic shaper transforms the input write event on i into a half-second timeout.

3. Provide the results of the select system call to handlers via their ready methods, to translate any output events that have occurred back to input events.

4. If no output events have occurred that result in application-level events, i.e., from the application's standpoint no interesting events have occurred, repeat the process.

We consider a timeout to be an "event" as well — a traffic-shaping handler could, for instance, convert a input write event into a half-second timeout, in the event that the shaper wishes to prevent the application from writing for a second in order to impose a bandwidth limitation.

To ensure consistent behavior, all input/output operations within TESLA handlers are non-blocking. To provide blocking I/O to the application, TESLA provides wrappers which implement blocking operations in terms of the aforementioned multiplexing framework and non-blocking operations.

The process of selection rewriting, while providing fully correct semantics for blocking I/O, adds an extraordinary amount of overhead to select (and, in general, to all potentially blocking I/O operations) which eliminates any performance advantage of an in-process architecture.

Emulating the proper semantics of filehandle duplication proves a challenge as well. In particular, an application may use dup2 to duplicate a filehandle *on top of* some filehandle which a TESLA handler is using behind the scenes. We resolve this problem by adding a layer of indirection on top of filehandles whereby TESLA automatically renumbers filehandles which were about to be dup2ed out of existence, but this abstraction adds complexity to handler code.

Recall that one of TESLA's goals is to move certain network functionality from the kernel to the user level where it belongs. However, we found that the in-process approach requires us to move *too much* functionality away from the kernel — we have to do a lot of work to emulate filehandle semantics at the user level, within a single process. Essentially we are required to implement a user-level threads package, where each handler gets an opportunity to run whenever a potentially blocking system call is invoked by the application. This is clearly outside the scope of our intended usage. Rather, we should use the kernel's scheduling mechanisms to implement concurrency behavior, i.e., between the wrapped application and its handlers.

For these reasons we instead chose to implement TESLA with handlers running in TESLA-dedicated processes (master processes). This makes it much easier to maintain the proper semantics for filehandles, improves scheduling by relying on the kernel scheduler rather than an *ad hoc* user-level thread package, and facilitates sharing of data between handlers.

Appendix B

Application Program Interfaces

In this section we provide several APIs in their entirety—most importantly, the flow_handler class, which we presented piece by piece in Chapter 2. For convenience, rather than using pointers directly, the flow handler API relies on several nontrivial utility classes, which we document here as well. The address class encapsulates an address structure, i.e., a sockaddr buffer. The data class encapsulates a pointer to a data buffer and the length of the data.

For the timer API, which we have already presented in its entirety, see Section 2.1.5.

B.1 Flow Handler API

To create a session-layer service, one writes a subclass of flow_handler that overrides some of its methods (e.g., write and avail) to provide the desired functionality. flow_handler provides default implementations of all virtual methods that simply propagate messages upstream or downstream, as described in Section 2.1.6.

```
class flow_handler {
  protected:
    // Create a downstream flow of the given type.
    handler *plumb(int domain, int type);

    // The upstream and downstream flows.  Default method implementations
    // of the virtual functions below work when there is exactly one element
    // in downstream.
    flow_handler* const upstream;
    vector<flow_handler*> downstream;

  public:
    // A structure used only to return results from accept.
    struct acceptret {
        handler *const h;
        const address addr;

        acceptret(); // Null constructor (operator bool returns false)
        acceptret(handler *, address);

        operator bool() const;
    };

    // A class encapsulating handler initialization parameters.
    class init_context {
      public:
        // Domain and type that we handle
        int get_domain() const;
        int get_type() const;

        // Get user-provided configuration parameters
        string arg(string name, string default = string()) const;
        int int_arg(string name, int default = -1) const;
    };

    // Constructor sans init context.
    flow_handler(const handler& h);

    // Constructor with init context.  If plumb is true, the flow_handler
    // constructor will create downstream[0] automatically with plumb.
    flow_handler(const init_context& ctxt, bool plumb = true);

    /**
     *
     * Downstream methods; these messages pass downstream.
     *
     */

    // Initiate connection request.  Return zero on success.
    virtual int connect(address);

    // Bind to a local address.  Return zero on success.
    virtual int bind(address);

    // Accept a connection.  Return null acceptret on failure.
    virtual acceptret accept();

    // Close the connection.  Return zero on success.
    virtual int close();

    // Write data.  Return false only if the connection has shut down
    // for writing.
```

```
    virtual bool write(data);

    // Shut down the connection for reading and/or writing.
    virtual int shutdown(bool r, bool w);

    // Listen for connection requests.
    virtual int listen(int backlog);

    // Return the local/remote host address.
    virtual address getsockname() const;
    virtual address getpeername() const;

    // upstream says: "You {may|may not} call my avail method."
    virtual void may_avail(bool) const;

    /**
     *
     * Upstream methods; these messages pass upstream.
     *
     * In each of these methods, <from> is a downstream flow.
     *
     */

    // Downstream flow <from> has bytes available (passed in <bytes>).
    virtual bool avail(flow_handler *from, data bytes);

    // Downstream flow <from> has a connection ready to be accepted,
    // i.e., from->accept() will succeed.
    virtual void accept_ready(flow_handler *from);

    // A connection attempt on <from> has concluded, either successfully
    // or not.
    virtual void connected(flow_handler *from, bool success);

    // <from> says: "You {may|may not} write to me."
    virtual void may_write(flow_handler *from, bool may);
};
```

B.2 Addresses

An address is a data buffer encapsulating a sockaddr structure. Each copy may be freely modified—addresses have the same copy semantics as STL strings. addresses are used as arguments to connect and bind methods, and are returned by accept, getsockname, and getpeername methods.

```
class address {
  public:
    // Null address (operator bool returns false)
    address();

    // Make a copy of the sockaddr buffer in addr.
    address(const void *addr, socklen_t len);

    // Returns a pointer to the address.
    const sockaddr *addr() const;

    // Returns the length of the address.
    const socklen_t addrlen() const;

    // Returns true if the address is valid (i.e., was not created
    // with the null constructor)
    operator bool() const;

    // Returns a string representation of the address.
    operator string() const;
};
```

B.3 Data

A data object is precisely a tuple: a pointer to an immutable data buffer and its length. data do not have STL string-like copy semantics—copying a data only makes a copy of the pointer, not the buffer. We use data instead of string because creating a string generally incurs a copy operation.

```
class data {
  public:
    // Empty data buffer
    data() {}

    // Create from a null-terminated string (which must stay alive
    // as long as the data object.
    data(const char *cstr);
    // Create from a buffer and length.  The buffer must stay alive
    // as long as the data object.
    data(const char *bits, unsigned int length);

    // Create from a string.  Equivalent to data(s.data(), s.length()),
    // so the string must stay alive and unmodified as long as the
    // data object.
    data(const string& str);

    // Return a pointer to the data buffer.
    const char *bits() const;

    // Return the length of the buffer.
    unsigned int length() const;

    // Allocate a string which is a copy of the data buffer.
    operator string() const;

    // Return the indexth byte of the buffer.
    const char & operator[](unsigned int index) const;
};
```

Bibliography

[1] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Ufo: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, 16(3):207–233, August 1998.

[2] Mark Allman, Hans Kruse, and Shawn Ostermann. An application-level solution to TCP's inefficiencies. In *Proc. 1st International Workshop on Satellite-based Information Services*, November 1996.

[3] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert T. Morris. Resilient overlay networks. In *Proc. ACM SOSP '01*, pages 131–145, October 2001.

[4] David G. Andersen, Deepak Bansal, Doroty Curtis, Srinivasan Seshan, and Hari Balakrishnan. System support for bandwidth management and content adaptation in Internet applications. In *Proc. OSDI '00*, pages 213–225, October 2000.

[5] H. Balakrishnan and S. Seshan. *The Congestion Manager*. Internet Engineering Task Force, June 2001. RFC 3124.

[6] Hari Balakrishnan, Hariharan S. Rahul, and Srinivasan Seshan. An integrated congestion management architecture for Internet hosts. In *Proc. ACM SIGCOMM '99*, pages 175–187, September 1999.

[7] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility,

safety and performance in the SPIN operating system. In *Proc. ACM SOSP '95*, pages 267–284, December 1995.

[8] Nina T. Bhatti and Richard D. Schlichting. A system for constructing configurable high-level protocols. In *Proc. ACM SIGCOMM '95*, pages 138–150, August 1995.

[9] Shaun Clowes. tsocks: A transparent SOCKS proxying library. http://tsocks.sourceforge.net/.

[10] Mark J. Cox, Ralf S. Engelschall, Stephen Henson, and Ben Laurie. Openssl: The open source toolkit for SSL/TLS. http://www.openssl.org/.

[11] Timothy W. Curry. Profiling and tracing dynamic library usage via interposition. In *Proc. Summer USENIX '94*, pages 267–278, June 1994.

[12] Paul R. Eggert and D. Stott Parker. File systems in user space. In *Proc. Winter USENIX '93*, pages 229–240, January 1993.

[13] David Ely, Stefan Savage, and David Wetherall. Alpine: A user-level infrastructure for network protocol development. In *Proc. 3rd USENIX Symposium on Internet Technologies and Systems*, pages 171–183, March 2001.

[14] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. ACM SOSP '95*, pages 251–266, December 1995.

[15] Tatu Ylonen et al. OpenSSH. http://www.openssh.org/.

[16] Kevin Fall and Kannan Varadhan. `ns` network simulator. http://www.isi.edu/nsnam/ns/.

[17] Nick Feamster, Deepak Bansal, and Hari Balakrishnan. On the interactions between layered quality adaptation and congestion control for streaming video. In *Proc. 11th International Packet Video Workshop*, April 2001.

[18] Marc E. Fiuczynski and Brian N. Bershad. An extensible protocol architecture for application-specific networking. In *Proc. USENIX '96*, pages 55–64, January 1996.

[19] Panos Gevros, Fulvio Risso, and Peter Kirstein. Analysis of a method for differential TCP service. In *Proc. IEEE GLOBECOM '99, Symposium on Global Internet*, pages 1699–1708, December 1999.

[20] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. SLIC: An extensibility system for commodity operating systems. In *Proc. USENIX '98*, pages 39–52, June 1998.

[21] Norman C. Hutchinson and Larry L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[22] Inferno Nettverk A/S. Dante: A free SOCKS implementation. http://www.inet.no/dante/.

[23] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proc. ACM SOSP '93*, pages 80–93, December 1993.

[24] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[25] David G. Korn and Eduardo Krell. A new dimension for the Unix file system. *Software — Practice and Experience*, 20(S1):S1/19–S1/34, 1990.

[26] Eduardo Krell and Balachander Krishnamurthy. COLA: Customized overlaying. In *Proc. Winter USENIX '92*, pages 3–8, January 1992.

[27] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. *SOCKS Protocol Version 5.* Internet Engineering Task Force, March 1996. RFC 1928.

[28] Marshall K. McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley Publishing Company, April 1996.

[29] David Mosberger and Larry L. Peterson. Making paths explicit in the Scout operating system. In *Proc. OSDI '96*, pages 153–167, October 1996.

[30] Hariharan Rahul. Unified Congestion Control for Unreliable Protocols. Master's thesis, Massachusetts Institute of Technology, August 1999.

[31] Herman C. Rao and Larry L. Peterson. Accessing files in an Internet: The Jade file system. *Software Engineering*, 19(6):613–624, 1993.

[32] Dennis M. Richie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[33] Alex C. Snoeren, Hari Balakrishnan, and M. Frans Kaashoek. Reconsidering Internet Mobility. In *Proc. 8th Workshop on Hot Topics in Operation Systems*, pages 41–46, May 2001.

[34] Douglas Thain and Miron Livny. Multiple bypass: Interposition agents for distributed computing. *Cluster Computing*, 4(1):39–47, March 2001.

[35] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proc. ACM SOSP '95*, pages 40–53, December 1995.

[36] Deborah A. Wallach, Dawson R. Engler, and M. Frans Kaashoek. ASHs: Application-specific handlers for high-performance messaging. In *Proc. ACM SIGCOMM '96*, pages 40–52, August 1996.

[37] Mark Yarvis, Peter Reiher, and Gerald J. Popek. Conductor: A framework for distributed adaptation. In *Proc. 7th Workshop on Hot Topics in Operation Systems*, pages 44–51, March 1999.

[38] Victor C. Zandy and Barton P. Miller. Reliable sockets. ftp://ftp.cs.wisc.edu/paradyn/technical˙papers/rocks.pdf, June 2001.