USENIX Association

Proceedings of USITS '03: 4th USENIX Symposium on Internet Technologies and Systems

Seattle, WA, USA
March 26–28, 2003

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

Jon Salz
MIT Laboratory for Computer Science
jsalz@lcs.mit.edu

Alex C. Snoeren
University of California, San Diego
snoeren@cs.ucsd.edu

Hari Balakrishnan
MIT Laboratory for Computer Science
hari@lcs.mit.edu

Session-layer services for enhancing functionality and improving network performance are gaining in importance in the Internet. Examples of such services include connection multiplexing, congestion state sharing, application-level routing, mobility/migration support, and encryption. This paper describes TESLA, a transparent and extensible framework allowing session-layer services to be developed using a high-level flow-based abstraction. TESLA services can be deployed transparently using dynamic library interposition and can be composed by chaining event handlers in a graph structure. We show how TESLA can be used to implement several session-layer services including encryption, SOCKS, application-controlled routing, flow migration, and traffic rate shaping, all with acceptably low performance degradation.

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, 15]).
- Sharing congestion information across connections sharing the same network path.
- Application-level routing, where applications route traffic in an overlay network to the final destination.
- End-to-end session migration for mobility across network disconnections.
- Encryption services for sealing or signing flows.
- General-purpose compression over low-bandwidth links.
- 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 [12]. 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 paper 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 own previous experience developing, debugging, and deploying a variety of Internet session-layer services. The earliest example is the Congestion Manager (CM) [4], which allows concurrent flows with a common source and destination to share congestion information, allocate available bandwidth, and adapt to changing network conditions. Other services include Resilient Overlay Networks (RON) [3], which provides application-layer routing in an overlay network, and the Migrate mobility architecture [23, 24], which preserves end-to-end communication across relocation and periods of disconnection.

Each these services was originally implemented at the kernel level, though it would be advantageous (for porta-

bility and ease of development and deployment) to have them available at the user level. Unfortunately this tends to be quite an intricate process. Not only must the implementation specify the internal logic, algorithms, and API, but considerable care must be taken handling the details of non-blocking and blocking sockets, interprocess communication, process management, and integrating the API with the application's event loop. The end result is that more programmer time and effort is spent setting up the session-layer plumbing than in the service's logic itself. Our frustrations with the development and implementation of these services at the user level were the prime motivation behind TESLA and led to three explicit design goals.

First, it became apparent to us that the standard BSD sockets API is not a convenient abstraction for programming session-layer services. Sockets can be duplicated and shared across processes; operations on them can be blocking or non-blocking on a descriptor-by-descriptor basis; and reads and writes can be multiplexed using several different APIs (e.g., select and poll). It is undesirable to require each service author to implement the entire sockets API. In response, TESLA exports a higher level of abstraction to session services, allowing them to operate on network flows (rather than simply socket descriptors) and treat flows as objects to be manipulated.

Second, there are many session services that are required *ex post facto*, often not originally thought of by the application developer but desired later by a user. 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 [9]—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 facilitate this, TESLA arranges for session services to be written as event handlers, with a callback-oriented interface between handlers that are arranged in a graph structure in the system.

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 paper 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 use by existing applications.

2 Related Work

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 [5] and the Exokernel [12], 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 [14] and ASHs [27]).

Operating systems such as Scout [21] and *x*-kernel [16] 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 [6]. 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.

In contrast to highly platform-dependent systems such as U-Net [26] and Alpine [11], TESLA does not attempt to allow users to replace or modify the system's network stack. Instead, it focuses on allowing 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, achieving greater platform independence and usability.

TESLA's modular structure, comprising directed graphs of processing nodes, shares commonalities with a number of previous systems such as *x*-kernel, the Click modular router [19], and UNIX System V streams [22]. Unlike the transport and network protocols typically considered in these systems, however, we view session-layer protocols as an extension of the application itself rather than a system-wide resource. This ensures that they are subject to the same scheduling, protection, and resource constraints as the application, minimizing the amount of effort (and privileges) required to deploy services.

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* [18]. It uses dynamic library interposition [9] to modify the interaction between the application and the system. This technique is popular with user-level file systems such as IFS [10] and Ufo [1], and several libraries that provide specific, transparent network services such as SOCKS [20], Reliable Sockets [29], and Migrate [23, 24]. Each of these systems provides only a specific service, however, not an architecture usable by third parties.

Conductor [28] traps application network operations and transparently layers composable "adaptors" on TCP connections, but its focus is on optimizing flows' performance characteristics, not on providing arbitrary additional services. Similarly, Protocol Boosters [13] proposes interposing transparent agents between communication endpoints to improve performance over particular links (e.g., compression or forward error correction). While Protocol Boosters were originally implemented in the FreeBSD and Linux kernel, they are an excellent example of a service that could be implemented in a generic fashion using TESLA. Thain and Livny recently proposed Bypass, a dynamic-library based interposition toolkit for building split-execution agents commonly found in distributed systems [25]. However, because of their generality, none of these systems provides any assistance in building modular network services, particularly at the session layer. To the best of our knowledge, TESLA is the first interposition toolkit to specifically support generic session-layer network services.

3 Architecture

As discussed previously, 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 introducing a *flow handler* as 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 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.

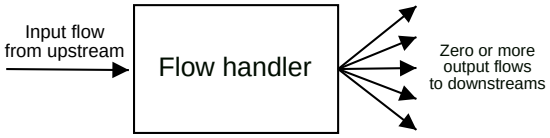

Figure 1: A flow handler takes as input one network flow and generates zero or more output flows.

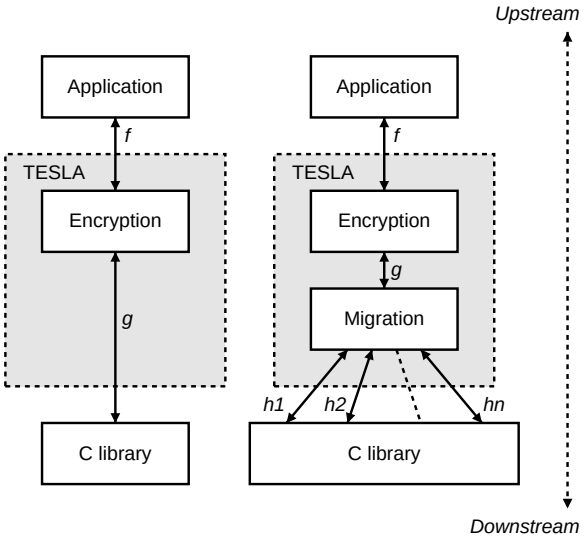

Figure 2: Two TESLA stacks. The encryption flow handler implements input flow f with output flow g. The migration flow handler implements input flow g with output flows $h_1 \ldots h_n$.

A flow handler, illustrated in Figure 1, is explicitly defined and constructed to operate on only one *input flow* from an *upstream* handler (or end application), and is hence devoid of any demultiplexing operations. Conceptually, therefore, one might think of a flow handler as dealing with traffic corresponding to a single socket only (as opposed to an interposition layer coded from scratch, which must potentially deal with operations on *all* open file descriptors). A flow handler generates zero or more *output flows*, which map one-to-one to *downstream* handlers (or the network send routine).[1] While a flow handler always has one input flow, multiple flow handlers may coexist in a single process, so they may easily share global state. (We will expound on this point as we further discuss the TESLA architecture.)

The left stack in Figure 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 Figure 2 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, the migration handler 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.

Each TESLA-enabled process has a particular *handler configuration*, an ordered list of handlers which will be used to handle flows. For instance, the configuration for the application to the right in Figure 2 is "encryption; migration." Flows through this application will be encryption and migration-enabled.

3.1 Interposition

To achieve the goal of application transparency, TESLA acts as an interposition agent at the C-library level. We provide a wrapper program, tesla, which sets up the environment (adding our libtesla.so shared library to LD_PRELOAD) and invokes an application, e.g.:

```
tesla +crypt -key=sec.gpg +migrate ftp mars
```

This would open an FTP connection to the host named mars, with encryption (using sec.gpg as the private key) and end-to-end migration enabled.

We refer to the libtesla.so library as the TESLA *stub*, since it contains the interposition agent but not any of the handlers themselves (as we will discuss later).

3.2 The flow_handler API

Every TESLA session service operates on flows and is implemented as a derived class of flow_handler, shown in Figure 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, namely the encryption handler; when its constructor in turn calls plumb, TESLA instantiates the next downstream handler, namely migration. Its plumb method creates a TESLA-internal handler to implement an actual TCP socket connection.

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

  public:
    // DOWNSTREAM methods
    virtual int connect(address);
    virtual int bind(address);
    virtual pair<flow_handler*, address> accept();
    virtual int close();
    virtual bool write(data);
    virtual int shutdown(bool r, bool w);
    virtual int listen(int backlog);
    virtual address getsockname();
    virtual address getpeername();
    virtual void may_avail(bool);
    virtual string getsockopt(...);
    virtual int setsockopt(...);
    virtual int ioctl(...);

    // UPSTREAM methods ('from' is a downstream flow)
    virtual void connected(flow_handler *from, bool success);
    virtual void accept_ready(flow_handler *from);
    virtual bool avail(flow_handler *from, data);
    virtual void may_write(flow_handler *from, bool may);
};
```

Figure 3: An excerpt from the flow_handler class definition. address and data are C++ wrapper classes for address and data buffers, respectively.

To send data to a downstream flow handler, a flow handler invokes the latter's write method, 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 to make events available to the upstream flow.

Flow handler methods are *asynchronous* and *event-driven*—each method call must return immediately, without blocking.

3.3 Handler method semantics

Many of the virtual flow_handler methods presented in Figure 3—write, connect, getpeername, etc.—have semantics similar to the corresponding non-blocking function calls in the C library. (A handler class may override any method to implement it specially, i.e., to change the behavior of the flow according to the session service the handler provides.) These methods are invoked by a handler's input flow. Since, in general, handlers will propagate these messages to their output flows (i.e., down-

stream), these methods are called *downstream methods* and can be thought of as providing an abstract flow service to the upstream handler.

(One downstream method has no C-library analogue: may_avail informs the handler whether or not it may make any more data available to its upstream handler. We further discuss this method in Section 3.5.)

In contrast the final four methods are invoked by the handler's *output flows*; each has a from argument identifying which downstream 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).
- may_write, analogous to may_avail, informs the handler whether or not it may write any more data downstream. This method is further discussed in Section 3.5.

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 downstream handler (in the case of downstream methods) or the upstream handler (in the case of upstream methods).

We further discuss the input and output primitives, timers, and blocking in the next few sections.

3.4 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 flow_handler semantics and the C library's I/O semantics is that, in flow handlers, writes and avails 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 simpler, 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, 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.

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.[2] 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 5.1), relieving handler authors of having to deal with multiplexing between different flows (e.g., with select).[3]

3.5 Timers and flow control

As we have mentioned before, TESLA handlers are event-driven, hence all flow_handler methods must return immediately. Clearly, however, some flow control mech-

anism 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 a potentially unbounded amount 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.

A flow handler may signal its input flow to stop sending it data, i.e., stop invoking its write method, 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). Flow handlers are *required* to respect may_avail and may_write requests from downstream and upstream handlers. This does not complicate 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 a timer facility allowing handlers to instruct the TESLA event loop to invoke a callback at a particular time.

3.6 Handler-specific services

Each flow handler exposes (by definition) the flow_handler interface, but some handlers may need to provide additional services to applications that wish to support them specifically (but still operate properly if TESLA or the particular handler is not available or not enabled). For instance, the end-to-end migration handler can "freeze" an application upon network disconnection, preserving the process's state and re-creating it upon reconnection. To enable this functionality we introduced the ioctl method to flow_handler. A "Migrate-aware" application (or, in the general case, a "TESLA-aware" application) may use the ts_ioctl macro to send a control message to a handler and receive a response:

```
struct freeze_params_t params = { ... };
struct freeze_return_t ret;
int ret = ts_ioctl(fd, "migrate", MIGRATE_FREEZE,
        &params, sizeof params, &ret, sizeof ret);
```

We also provide a mechanism for handlers to send events to the application asynchronously; e.g., the migration handler can be configured to notify the application when a flow has been migrated between IP addresses.

3.7 Process management

Flow handlers comprise executable code and runtime state, and in designing TESLA there were two obvious choices regarding the context within which the flow handler code would run. The simplest approach would place flow handlers directly within the address space of application processes; TESLA would delegate invocations of POSIX I/O routines to the appropriate flow_handler methods. Alternatively, flow handlers could execute in separate processes from the application; TESLA would manage the communication between application processes and flow handler processes.

The former, simpler approach would not require any interprocess communication or context switching, minimizing performance degradation. Furthermore, this approach would ensure that each flow handler has the same process priority as the application using it, and that there are no inter-handler protection problems to worry about.

Unfortunately, this approach proves problematic in several important cases. First, some session services (e.g., shared congestion management as in CM) 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 very difficult in this model.

For these reasons, we choose to separate the context in which TESLA flow handlers run from the application processes that generate the corresponding flows. When an application is invoked through the tesla wrapper (and hence linked against the stub), the stub library creates or connects to a *master process*, a process dedicated to executing handlers.

Each master process has its own handler configuration (determined by the user, as we shall describe later). 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*.

The application and master process communicate via

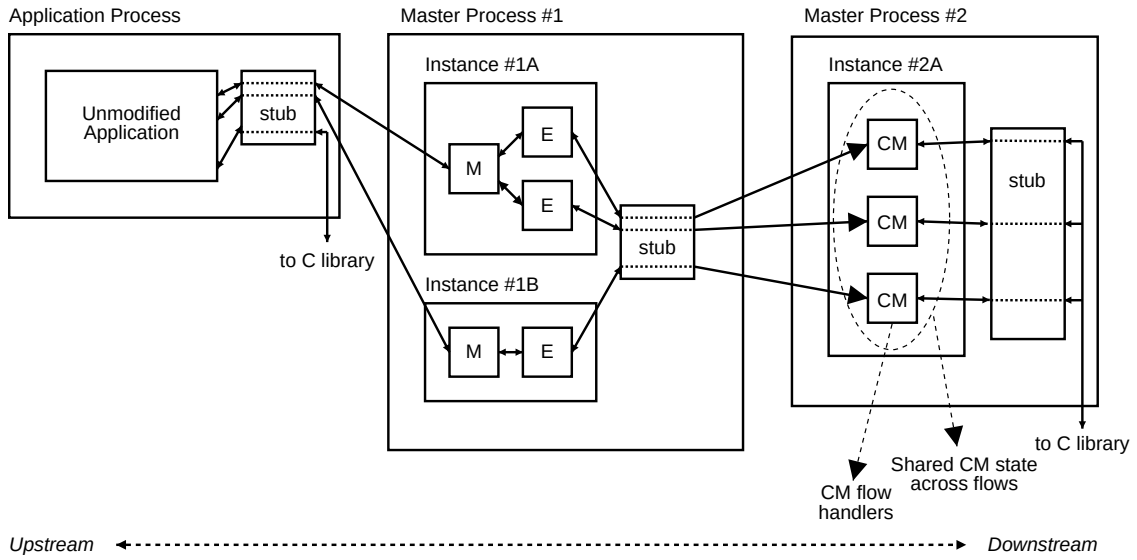

Figure 4: Possible interprocess data flow for an application running under TESLA. M is the migration handler, E is encryption, and CM is the congestion manager handler.

a TESLA-internal socket for each application-level flow, and all actual network operations are performed by the master process. When the application invokes a socket operation, TESLA traps it and converts it to a message which is transmitted to the master process via the master socket. The master process returns an immediate response (this is possible since all TESLA handler operations are non-blocking, i.e., return immediately).

Master processes are forked from TESLA-enabled applications and are therefore linked against the TESLA stub also. This allows TESLA instances to be chained, as in Figure 4: an application-level flow may be connected to an instance in a first master process, which is in turn connected via its stub to an instance in another master process. Chaining enables some handlers to run in a protected, user-specific context, whereas handlers that require a system-scope (like system-wide congestion management) can run in a privileged, system-wide context.

3.8 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.

In general, TESLA support for setuid and setgid applications, which assume the protection context of the binary iself rather than the user who invokes them, is a tricky affair: 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 user is a system administrator and, if so, allows the user to perform an administrative operation. If a malicious user provides flow handlers that spoof a positive response for the lookup, the binary may be tricked into granting him or her administrative privileges.

Nevertheless, many widely-used applications (e.g., ssh) 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.

4 Example handlers

We now describe 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.

4.1 Traffic shaping

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. TESLA allows us to provide a generic, user-level traffic-shaping service as an alternative to building shaping directly into an application (as in the rsync --bwlimit option) or using an OS-level packet-filtering service (which would require special setup and superuser privileges).

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.

We have also developed latency_handler, which delays each byte supplied to or by the network for a user-configurable amount of time (maintaining this data in an internal ring buffer). This handler is useful for simulating the effects of network latency on existing applications.

4.2 SOCKS and application-level routing

Our SOCKS handler is functionally similar to existing transparent SOCKS libraries [7, 17], although its implementation is significantly simpler. Our handler overrides the connect handler to establish a connection to a proxy server, rather than a 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 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 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. In the future, ron_handler may also utilize the end-to-end migration of migrate_handler (presented below in Section 4.4) 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.

```
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 5: A transparent encryption/decryption handler.

4.3 Encryption and compression

crypt_handler is a triple-DES encryption handler for TCP streams. We use OpenSSL [8] to provide the DES implementation. Figure 5 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).

Our compression handler is very similar: we merely wrote a wrapper class (analogous to des3_cfb64_stream) for the freely available zlib stream compression library.

4.4 Session migration

We have used TESLA to implement transparent support in the Migrate session-layer mobility service [23]. 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 endpoint 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 endpoint 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 endpoint 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 trickier: because received data may not have 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 (the "connection reset" messages typically experienced during long periods of disconnection) and prevent them from reaching the application, instead taking appropriate action to manage and conceal the changes in endpoints. In doing so, the handler also overrides the getsockname() and getpeername() calls to return the original endpoints, irrespective of the current location.

5 Implementation

The TESLA stub consists largely of *wrapper functions* for sockets API functions in the C library. When an application creates a socket, TESLA intercepts the socket library call and sends a message to the master process, inquiring whether the master has any handlers registered for the requested domain and type. If not, the master returns a negative acknowledgement and the application process simply uses the socket system call to create and return the request socket.

If, on the other hand, there is a handler 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 retained in the master and the other passed to the application 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.

Later, when the application invokes a socket API call, such as connect or getsockname, on a TESLA-wrapped filehandle, the wrapper function informs the master process, which invokes the corresponding method on the handler object for that flow. The master process returns this result to the stub, which returns the result to the application.

In general the TESLA stub does not need to specially handle reads, writes, or 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.

Implementing fork, dup, and dup2 is quite simple. Having a single 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.

5.1 top_handler **and** bottom_handler

We have 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 Figure 2) is a special flow handler called top_handler.

When a TESLA master'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.

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 system 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.

5.2 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 introduce overhead such as additional memory copies, interprocess communication, scheduling contention, etc., 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. Services implemented entirely within the operating system kernel are also likely to impose little in the way of architectural overhead. However, adding an interposition agent like TESLA outside the contexts of both the application and the operating system 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, and to an application-internal agent. Although we do not consider it here, we expect an in-kernel service implementation would perform similarly to the application-internal agent.

We provide two benchmarks. In bandwidth(s), a client establishes a TCP connection to a server and reads data into a s-byte buffer until the connection closes, measuring the number of bytes per second received. (The server writes data s bytes at a time as well.) In latency, a client connects to a server, sends it a single byte, waits for an 1-byte response, and so forth, measuring the number of round-trip volleys per second.

We run bandwidth and latency under several configurations:

1. The benchmark program running (a) unmodified and (b) with triple-DES encryption performed directly within the benchmark application.
2. The benchmark program, with each connection passing through TCP proxy servers both on the client and server hosts. In (a), the proxy servers perform no processing on the data; in (b) the proxy servers encrypt and decrypt traffic between the two hosts.
3. Same as 2(a) and 2(b), except with proxy servers listening on UNIX-domain sockets rather than TCP sockets. This is similar to the way TESLA works internally: the proxy server corresponds to the TESLA master process.
4. The benchmark program running under TESLA with (a) the dummy handler enabled or (b) the crypt handler enabled. (dummy is a simple no-op handler that simply passes data through.)

We can think of the (a) configurations as providing a completely trivial transparent service, an "identity service" with no algorithm (and hence no algorithmic overhead) at all. Comparing benchmark 1(a) with the other (a) configurations isolates the architectural overhead of a typical proxy server (i.e., an extra pair of socket streams through which all data must flow) and the architectural overhead of TESLA (i.e., an extra pair of socket streams plus any overhead incurred by the master processes). Comparing the (a) benchmarks, shown on the left of Figure 6, with the corresponding (b) benchmarks, shown on the right, isolates the algorithmic overhead of the service.

To eliminate the network as a bottleneck, we first ran the bandwidth test on a single host (a 500-MHz Pentium III running Linux 2.4.18) with the client and server connecting over the loopback interface. Figure 6 shows these results. Here the increased demand of interprocess communication is apparent. For large block sizes (which we would expect in bandwidth-intensive applications), introducing TCP proxies, and hence tripling the number

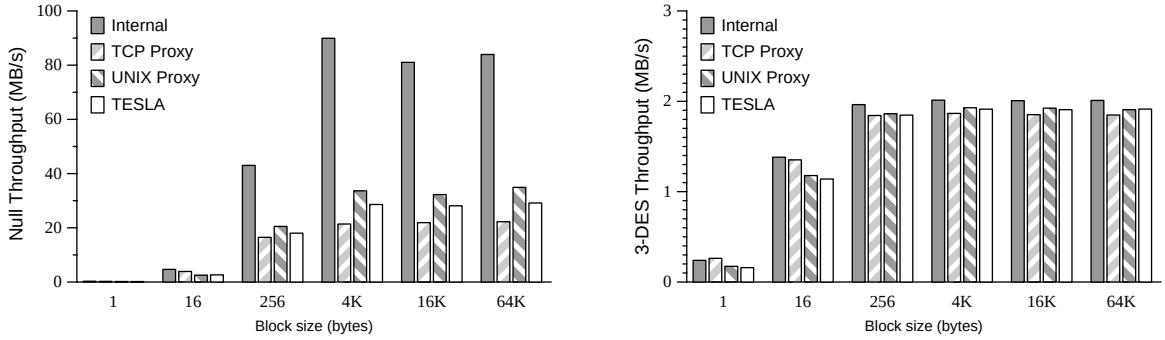

Figure 6: Results of the bandwidth test on a loopback interface.

of TCP socket streams involved, reduces the throughput by nearly two thirds. TESLA does a little better, since it uses UNIX-domain sockets instead of TCP sockets.

In contrast, Figure 7 shows the results of running bandwidth with the client and server (both 500-MHz Pentium IIIs) connected via a 100-BaseT network.[4] Here neither TESLA nor a proxy server incurs a significant throughput penalty. For reasonably large block sizes either the network (at about 89 Mb/s, or 11 MB/s) or the encryption algorithm (at about 3 MB/s) becomes the bottleneck; for small block sizes the high frequency of small application reads and writes is the bottleneck.

We conclude that on relatively slow machines and very fast networks, TESLA—or any other IPC-intensive interposition mechanism—may cause a decrease in peak throughput, but when used over typical networks, or when used to implement nontrivial network services, TESLA causes little or no throughput reduction.

The latency benchmark yields different results, as illustrated in Figure 8: since data rates never exceed a few kilobytes per second, the computational overhead of the algorithm itself is negligible and any slowdown is due exclusively to the architectural overhead. We see almost no difference between the speed of the trivial service and the nontrivial service. Introducing a TCP proxy server on each host incurs a noticeable performance penalty, since data must now flow over a total of three streams (rather than one) from client application to server application. TESLA does a little better, as it uses UNIX-domain sockets rather than TCP sockets to transport data between the applications and the TESLA masters; under Linux, UNIX-domain sockets appear to have a lower latency than the TCP sockets used by the proxy server. We consider this performance hit quite acceptable: TESLA increases the end-to-end latency by only tens of microseconds (4000 round-trips per second *versus* 7000), whereas network latencies are typically on the order of milliseconds or tens of milliseconds.

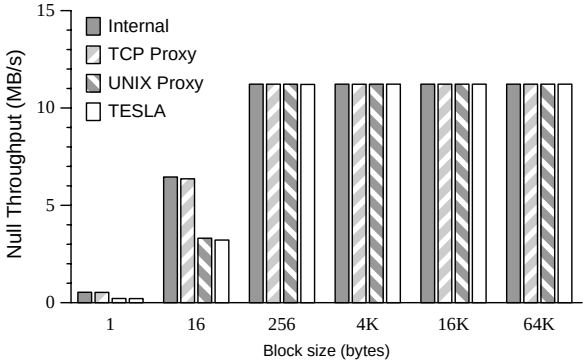

Figure 7: Results of the bandwidth test on a 100-BaseT network. Only the null test is shown; triple-DES performance was similar to performance on a loopback interface (the right graph of Figure 6).

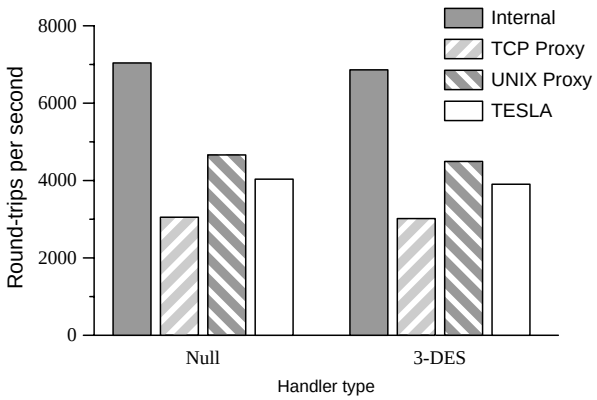

Figure 8: Results of the latency test over a 100-BaseT network. The block size is one byte.

6 Conclusion

This paper has outlined 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 to processes in a way that enables 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. We are pleased to report that other researchers have recently found TESLA useful in deploying their own session layer services, and even adopted it as a teaching tool.

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 configuration so that the user can specify which handlers to apply to flows based on flow attributes such as port and address. Finally, we plan to explore the possibility of moving some handler functionality directly into the application process or operating system kernel to minimize TESLA's architectural overhead.

TESLA is available for download at http://nms.lcs.mit.edu/software/tesla/.

Acknowledgments

This work was funded by NTT Inc. under the NTT-MIT research collaboration, by Acer Inc., Delta Electronics Inc., HP Corp., NTT Inc., Nokia Research Center, and Philips Research under the MIT Project Oxygen partnership, by Intel Corp., and by IBM Corp. under a university faculty award.

The authors would like to thank Dave Andersen, Stan Rost, and Michael Walfish of MIT for their comments.

References

[1] ALEXANDROV, A. D., IBEL, M., SCHAUSER, K. E., AND SCHEIMAN, C. J. Ufo: A personal global file system based on user-level extensions to the operating system. *ACM TOCS 16*, 3 (Aug. 1998), 207–233.

[2] ALLMAN, M., KRUSE, H., AND OSTERMANN, S. An application-level solution to TCP's inefficiencies. In *Proc. WOSBIS '96* (Nov. 1996).

[3] ANDERSEN, D. G., BALAKRISHNAN, H., KAASHOEK, M. F., AND MORRIS, R. T. Resilient overlay networks. In *Proc. ACM SOSP '01* (Oct. 2001), pp. 131–145.

[4] BALAKRISHNAN, H., RAHUL, H. S., AND SESHAN, S. An integrated congestion management architecture for Internet hosts. In *Proc. ACM SIGCOMM '99* (Sept. 1999), pp. 175–187.

[5] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., BECKER, D., FIUCZYNSKI, M., CHAMBERS, C., AND EGGERS, S. Extensibility, safety and performance in the SPIN operating system. In *Proc. ACM SOSP '95* (Dec. 1995), pp. 267–284.

[6] BHATTI, N. T., AND SCHLICHTING, R. D. A system for constructing configurable high-level protocols. In *Proc. ACM SIGCOMM '95* (Aug. 1995), pp. 138–150.

[7] CLOWES, S. tsocks: A transparent SOCKS proxying library. http://tsocks.sourceforge.net/.

[8] COX, M. J., ENGELSCHALL, R. S., HENSON, S., AND LAURIE, B. Openssl: The open source toolkit for SSL/TLS. http://www.openssl.org/.

[9] CURRY, T. W. Profiling and tracing dynamic library usage via interposition. In *Proc. Summer USENIX '94* (June 1994), pp. 267–278.

[10] EGGERT, P. R., AND PARKER, D. S. File systems in user space. In *Proc. Winter USENIX '93* (Jan. 1993), pp. 229–240.

[11] ELY, D., SAVAGE, S., AND WETHERALL, D. Alpine: A user-level infrastructure for network protocol development. In *Proc. 3rd USITS* (Mar. 2001), pp. 171–183.

[12] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE JR., J. Exokernel: An operating system architecture for application-level resource management. In *Proc. ACM SOSP '95* (Dec. 1995), pp. 251–266.

[13] FELDMEIER, D. C., MCAULEY, A. J., SMITH, J. M., BAKIN, D. S., MARCUS, W. S., AND RALEIGH, T. M. Protocol boosters. *IEEE JSAC 16*, 3 (Apr. 1998), 437–444.

[14] FIUCZYNSKI, M. E., AND BERSHAD, B. N. An extensible protocol architecture for application-specific networking. In *Proc. USENIX '96* (Jan. 1996), pp. 55–64.

[15] GEVROS, P., RISSO, F., AND KIRSTEIN, P. Analysis of a method for differential TCP service. In *Proc. IEEE GLOBECOM '99* (Dec. 1999), pp. 1699–1708.

[16] HUTCHINSON, N. C., AND PETERSON, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*, 1 (Jan. 1991), 64–76.

[17] INFERNO NETTVERK A/S. Dante: A free SOCKS implementation. http://www.inet.no/dante/.

[18] JONES, M. B. Interposition agents: Transparently interposing user code at the system interface. In *Proc. ACM SOSP '93* (Dec. 1993), pp. 80–93.

[19] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *ACM TOCS 18*, 3 (Aug. 2000), 263–297.

[20] LEECH, M., GANIS, M., LEE, Y., KURIS, R., KOBLAS, D., AND JONES, L. *SOCKS Protocol Version 5*. IETF, Mar. 1996. RFC 1928.

[21] MOSBERGER, D., AND PETERSON, L. L. Making paths explicit in the Scout operating system. In *Proc. OSDI '96* (Oct. 1996), pp. 153–167.

[22] RICHIE, D. M. A stream input-output system. *AT&T Bell Laboratories Technical Journal 63*, 8 (Oct. 1984), 1897–1910.

[23] SNOEREN, A. C. *A Session-Based Architecture for Internet Mobility*. PhD thesis, Massachusetts Institute of Technology, Dec. 2002.

[24] SNOEREN, A. C., BALAKRISHNAN, H., AND KAASHOEK, M. F. Reconsidering Internet Mobility. In *Proc. HotOS-VIII* (May 2001), pp. 41–46.

[25] THAIN, D., AND LIVNY, M. Multiple bypass: Interposition agents for distributed computing. *Cluster Computing 4*, 1 (Mar. 2001), 39–47.

[26] VON EICKEN, T., BASU, A., BUCH, V., AND VOGELS, W. U-Net: A user-level network interface for parallel and distributed computing. In *Proc. ACM SOSP '95* (Dec. 1995), pp. 40–53.

[27] WALLACH, D. A., ENGLER, D. R., AND KAASHOEK, M. F. ASHs: Application-specific handlers for high-performance messaging. In *Proc. ACM SIGCOMM '96* (Aug. 1996), pp. 40–52.

[28] YARVIS, M., REIHER, P., AND POPEK, G. J. Conductor: A framework for distributed adaptation. In *Proc. HotOS-VII* (Mar. 1999), pp. 44–51.

[29] ZANDY, V. C., AND MILLER, B. P. Reliable network connections. In *Proc. ACM/IEEE Mobicom '02* (Atlanta, Georgia, Sept. 2002), pp. 95–106.

Notes

[1]It is important not to take the terms upstream and downstream too literally in terms of the flow of actual bytes; rather, think of TESLA as the session layer on the canonical network stack, with upstream handlers placed closer to the presentation or application layer and downstream handlers closer to the transport layer.

[2]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.

[3]The guaranteed-completion semantics we discuss in this section apply to the TESLA flow_handler API only, *not* to standard socket I/O functions (read, write, etc.). TESLA makes sure that the semantics of POSIX I/O functions remain unchanged, for transparency's sake.

[4]Note the anomaly with small block sizes: on our test machines, using a UNIX-domain socket to connect the application to the intermediary process (whether a proxy server or the TESLA master) incurs a severe performance hit. Reconfiguring TESLA to use TCP sockets internally (rather than UNIX-domain sockets) increases TESLA's performance to that of the TCP proxy server. Since this anomaly does not seem specific to TESLA, and occurs only in a somewhat unrealistic situation—a high-bandwidth application using a 1- or 16-byte block size—we do not examine it further.