Resilient Connections for SSH and TLS

Teemu Koponen
Helsinki Institute for Information Technology
teemu.koponen@hiit.fi

Pasi Eronen
Nokia Research Center
pasi.eronen@nokia.com

Mikko Särelä
Helsinki University of Technology
Laboratory for Theoretical Computer Science
id@tcs.hut.fi

Abstract

Disconnection of an SSH shell or a secure application session due to network outages or travel is a familiar problem to many Internet users today. In this paper, we extend the SSH and TLS protocols to support resilient connections that can span several sequential TCP connections. The extensions allow sessions to survive both changes in IP addresses and long periods of disconnection. Our design emphasizes deployability in real-world environments, and addresses many of the challenges identified in previous work, including assumptions made about network middleboxes such as firewalls and NATs. We have also implemented the extensions in the OpenSSH and PureTLS software packages and tested them in practice.

1 Introduction

An increasing number of Internet hosts are mobile and equipped with more than one network interface. Simultaneously, operation of mobile hosts has become more continuous: the hosts have long uptimes, and the applications do not need to be closed when the host enters a "suspended" state. However, in the today's Internet, applications experience this combination of improved connectivity and operation as a less stable networking environment. This is mainly because transport layer connections break more frequently due to changes in IP addresses, network failures, and timeouts during disconnected or suspended operation.

It is often desirable to hide these disruptions from the end user. For instance, a user should be able to suspend a laptop, move to a different location, bring up the laptop, and continue using the applications that were left open with minimal inconvenience. In other words, the system should provide *session continuity* over the disruptions in network connectivity (cf. Snoeren's analysis of the session abstraction [23]).

Traditionally, session continuity has been considered as a part of mobility, and has been handled in the data link layer (e.g., wireless LAN or GPRS handover mechanisms) or in the network layer (e.g., Mobile IP). However, there are a number of reasons why providing session continuity higher in the protocol stack is desirable:

Long disconnection periods: while network-layer mobility mechanisms can deal with changing IP addresses, they cannot help the transport layer to overcome likely timeouts during long disconnections. Moreover, how exactly should long disconnections be handled often depends on the application in question.

No network infrastructure: in today's Internet it is common that clients are mobile but servers are not. In this kind of environment, session continuity can be provided without requiring the deployment of additional fixed infrastructure (such as Mobile IP home agents).

Applications get upgraded: it is often claimed that mobility has to be low in the stack to enable it for a large number of different applications. However, we hypothesize that it is often actually easier to deploy resilient mechanisms built into applications. After all, the applications get upgraded all the time and processes for that exist; but installing and configuring a Mobile IP implementation is beyond capabilities of most users and system administrators.

Limited end-to-end connectivity: mobility mechanisms implemented in the network or transport layer may not work across various types of middleboxes that are present in the network. For instance, if a firewall near a client allows only outbound TCP connections, Mobile IP does not work. Session continuity mechanisms integrated into applications make the least number of assumptions about the network between the endpoints.

These arguments suggest that the session layer is the lowest layer to implement *resilient connections* that can span several sequential transport layer (TCP) connections, and thus, survive not only changes in IP addresses, but also relatively long periods of disconnection.

In this paper, we extend two common secure session layer protocols to support resilient connections: Secure SHell (SSH) Transport Layer Protocol [28, 29] and Transport Layer Security (TLS) [3].[1] We have implemented these extensions in two open-source software packages: OpenSSH, the most popular SSH implementation [16], and PureTLS, a Java TLS library [20].

Our main contributions are as follows. First, we have developed resiliency extensions for the common TLS and SSH protocols that largely avoid the deployability problems associated with previous proposals. Second, we have analyzed the challenges faced when implementing this kind of extensions to legacy software packages that were not designed with resiliency in mind. In particular, different styles of handling concurrency and I/O have large implications for the implementations: OpenSSH uses asynchronous (select-based) I/O with a process for each client, while PureTLS uses synchronous I/O with threads.

The rest of the paper is structured as follows. In Section 2, we introduce the SSH and TLS protocols and previous work on resilient connections. Our design principles, described in Section 3, attempt to address deployment challenges we have identified in the existing proposals. In Section 4, we introduce our extensions to the SSH and TLS protocols. Section 5 describes our prototype implementations, which are then evaluated in Section 6. Finally, Section 7 summarizes our conclusions and discusses remaining open issues.

2 Background and related work

The Secure Shell (SSH) is a protocol for secure login and other network services [28]. It consists of three main sub-protocols: the SSH transport layer protocol, user authentication protocol, and connection protocol. The SSH transport layer protocol is the lowest layer, and is responsible for authenticating the server and providing an encrypted and integrity-protected channel for the other sub-protocols. The user authentication protocol authenticates the client, while the connection protocol multiplexes several logical connections (such as interactive terminal sessions, X11 window system forwarding, and TCP/IP port forwarding) over a single transport layer connection.

Transport Layer Security (TLS) is a session layer protocol providing encrypted and authenticated communication session for communication between two applications [3]. It consists of two major parts: the TLS record protocol provides a secure communication channel to upper layers, and is responsible for encryption and integrity protection of data. The TLS handshake protocol provides the key material and authentication for the TLS record protocol; this usually involves X.509 certificates and a key exchange based on RSA encryption. The two remaining components of TLS, the alert and change cipher spec protocols, are beyond the scope of this paper.

[1] Note that despite their names, both protocols are strictly above the transport layer (TCP) in the protocol stack, and thus calling them session-layer protocols is more accurate.

The benefits of providing session continuity above the transport layer have been recognized before; for instance, Duchamp [4] and Snoeren [24] provide several arguments in its favor. There is a large number of proposals that provide resilient connections above the transport layer but below the application layer protocol: Persistent connections [32], Mobile TCP socket [18, 19], MobileSocket [15], SLM or Session Layer Mobility [11], Reliable sockets [30], Migrate [23], Robust TCP connections [5], NapletSocket [33], Channel-based connectivity management [26], and Dharma [13], to mention just a few examples.

The common part of most of these proposals is a library placed above the transport layer but below the sockets API used by the application. The library presents a single unbroken communication channel to the application, hiding transport layer disruptions from the applications. The library is responsible for the signaling required to manage the multiple TCP connections, and also buffers application data so it can be retransmitted over a new TCP connection if necessary (this is required since most operating systems do not allow applications to access the TCP buffers).

However, implementing resilient connections in the "sockets API" layer has a number of drawbacks.

- The proposals typically use out-of-band signaling: a separate TCP connection (or UDP-based "session") coordinates multiple TCP connections. This can lead to deployment problems if, e.g., a firewall allows the port used by the application itself, but not the port used for resiliency signaling. An important reason for out-of-band signaling is the lack of an extension negotiation mechanism in the sockets API layer; however, such a mechanism is essential for incremental deployment. While some proposals (such as Zandy's reliable sockets [30]) do actually implement the initial resiliency signaling in-band, they rely on obscure TCP semantics with questionable deployability properties. However, even these solutions change to out-of-band signaling after the connection setup (e.g., due to TCP's head-of-line blocking issues).

- A separate key exchange is required to protect the signaling messages (if the messages are protected at all). This introduces additional overhead.

- While a separately delivered dynamically linked library that "hijacks" the operations of the normal

socket calls is a good approach for research, it creates deployment problems if it is does not come bundled and tested with the software with which it is to be used. Deploying such separate component is likely to get less-than-enthusiastic response from, e.g., corporate IT departments who would have to deploy and manage this component in mission-critical environments.

Proposals to implement the session continuity even higher in the protocol stack than the session layer exist. If an application protocol connection setup is a light-weight operation (e.g., HTTP GET), it's not necessary to extend any protocol to implement reconnections. An application just reconnects and at the same time minimizes the visibility of the reconnection to its user. For example, most modern mail user agents operate in this manner.

An application protocol connection setup may consume an considerable amount of resources, however, and thus, several application protocols have been extended to provide session continuity. For instance, REX, an SSH-like remote execution utility [9], the XMOVE extension to the X Window System [25], the REST extension to FTP [6], and SIP [22] all allow continuing a session even if a transport layer connection is disrupted for some reason. These extensions are typically very specific to the application in question; in contrast, our TLS extensions would work with any application-layer protocol run over TLS.

Session continuity for interactive terminal sessions can also be provided by decoupling the terminal seen by applications from the remote terminal session, as done in, for instance, Screen [7] and Bellovin's Session Tty Manager [1]. However, these approaches still require the user to manually establish a new SSH connection and re-attach the terminal session.

Proposals that operate in the transport layer (e.g., Huitema's Multi-Homed TCP [8]) are beyond the scope of this paper. As they require modifications to the operating system's TCP/IP stack, and may not work with existing middleboxes, we do not consider them easily deployable.

3 Design principles

Based on the existing work, we have set our design principles to emphasize deployability.

No network changes: no extra requirements for the network or middleboxes between two communicating hosts should be set. As an example, the extensions must not require any additional configuration in firewalls.

Incremental deployment: the extensions should provide functionality once both connection end-points support it. The extensions should also interoperate with legacy end-points without the extensions.

Limited end-point changes: the extensions should require only modifications in TLS and SSH implementations, but no operating system changes or additional software components. The latter includes, e.g., dynamic libraries interposed between the application and the operating system.

In terms of functionality, our design principles were:

Disconnections may last long: the extension should deal gracefully with long periods of disconnection. The maximum supported disconnection period is a local policy issue, and not a protocol issue. Thus, the protocol extensions should not limit the duration of disconnections.

No handover optimization: the extensions are not optimized for fast handovers. This is mainly because we believe the default disconnection to be relatively long (from tens of seconds to hours).

In addition to the protocol extensions, there are certain implementation aspects to be considered. Server side concurrency is the most important one. The mechanisms used to implement concurrency often depend on the operating system and programming language used. Obviously, our extensions should not prevent typical server implementation strategies such as "a process for each client" (either forked on demand or beforehand), "a thread for each client", or select-style asynchronous I/O.

4 Protocol extensions

In this section we describe the extensions made to the SSH and TLS protocols. Since the extensions have much in common, we present the shared features first, followed by the SSH and TLS specific details.

4.1 Common features

In-band signaling: deployability concerns in practice mandate the use of in-band signaling. In other words, information required by the extensions is sent as part of normal SSH and TLS messages, and all TCP connections are initiated by the client. This ensures that resilient connections do not introduce any additional requirements for the network between the client and the server.

Extension negotiation: incremental deployment requires interoperability with endpoints that do not support these extensions, and thus, their use has to be negotiated. Fortunately, both SSH and TLS have mechanisms for negotiating protocol features when the connection is set up.

Securing signaling: when the client creates a new TCP connection to the server, it has to somehow indicate that it wants to continue a previous session, and prove that it is indeed the same client as previously. Thus, we need a way to identify an existing session (any public and unique information exchanged during the session setup

will do) and way to authenticate the signaling. Since both SSH and TLS establish session keys between the client and the server, the authentication is relatively easy to do.

Buffer management: both SSH and TLS operate over TCP, which provides a reliable lossless connection channel. However, when the TCP connection breaks, the TCP socket buffers may contain data that was not yet received by the other endpoint, and thus, the data has to be retransmitted when a new TCP connection is created. Since operating systems typically do not allow access to the TCP buffers, separate buffers have to be maintained in the application.

Previously, two different approaches have been used for managing these buffers: either data is removed from the buffer only when an explicit session layer acknowledgement is received (e.g., MobileSocket by Okoshi et al. [15]), or the buffer size is limited to the size of TCP buffers (e.g., Zandy's reliable sockets [31]). We chose the former approach: the endpoints send acknowledgements regularly (say, after receiving 64 kB from the peer). While the nodes may know their own TCP buffer sizes, the network may also contain transport layer proxies that buffer data: for instance, TLS is often run through web proxies using the "CONNECT" method [12]. Thus, while explicit acknowledgements add some overhead, only they ensure that the extensions work properly in existing network environments. This corresponds to the "end-to-end argument" by Saltzer et al. [21]: since parts of TCP may be implemented by the communication system itself, end-to-end reliability can be correctly implemented only above TCP.

Closing: SSH and TLS connections are both tightly bound to an underlying TCP connection. The resiliency extensions render the situation more complex: if the TCP connection breaks, the server should wait for the client to reconnect again. Thus, the protocol should have an explicit "close" message to be used when the endpoints actually want to close the session permanently. Fortunately, both the SSH transport layer protocol and TLS have this kind of messages. However, we discovered that OpenSSH did not actually send the close message, since previously there was no need to differentiate between a gracefully closed session and a broken TCP connection.

4.2 Extending SSH

Resilient connections for SSH could be implemented either in the SSH transport layer protocol or the connection protocol. In the end, we decided to implement our extension in the SSH transport layer protocol, since this seemed to be simpler, and had more in common with the TLS extensions described in the next section.

The SSH protocol suite is extensible: in the transport layer protocol, the client and the server negotiate the algorithms that will be used during the session. However, while the algorithms are negotiable, the negotiation does fix algorithm categories. Thus, we had to re-use an existing category to negotiate the resiliency support: the client and the server announce their support for this extension by including a Key EXchange (KEX) algorithm named "resilient" as the least-preferred algorithm. If both endpoints supports this extension, they enable buffering of data and sending of explicit acknowledgements. The acknowledgement is a new SSH message type that contains the sequence number of the next expected record.

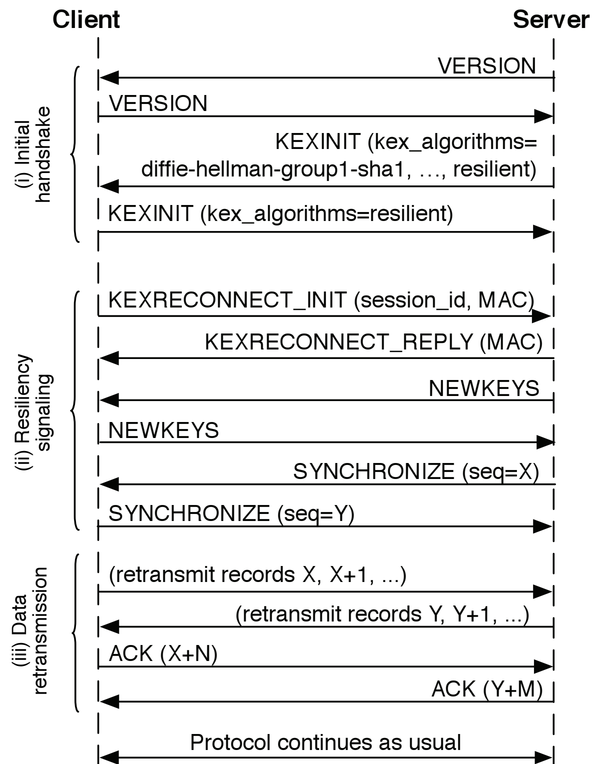

Figure 1: Reconnecting an existing SSH session.

Modeling the resiliency extension as a special key exchange algorithm also simplifies things when the client wants to reconnect; i.e., continue the same session over a different TCP connection. The exchange is shown in Figure 1. The client indicates that it wants to continue a session by listing "resilient" as the only supported key exchange algorithm. The client then sends a message containing a session identifier and a Message Authentication Code (MAC); the server responds with its own MAC. The MACs prove that the parties are still the same as in the original connection, and are calculated over the VERSION and KEXINIT messages (which include nonces to prevent replays). The MAC is calculated using a separate key used only for the KEXRECONNECT messages, and is derived during the initial handshake at the same time

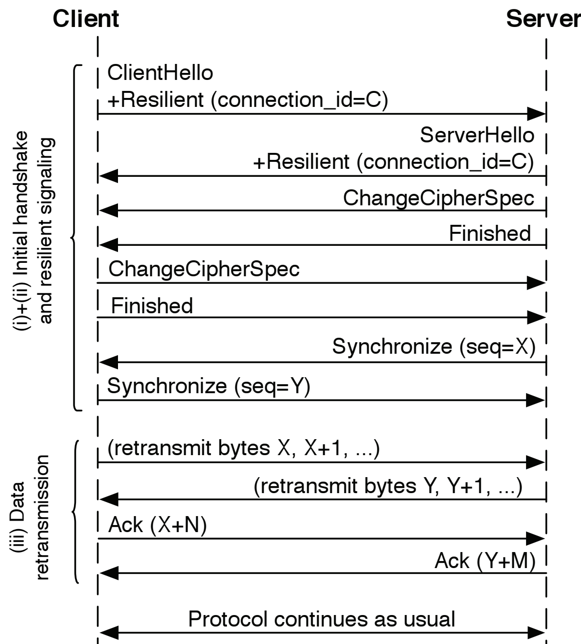

Figure 2: Reconnection procedure in TLS.

as the encryption and integrity protection keys.

After this, the endpoints take the cryptographic keys into use, send a “synchronize” message indicating where the previous connection was broken, and retransmit lost data from the buffers.

The SSH transport layer supports payload compression. While the transport layer protocol implements the compression, compression is done before encryption. Thus, the compression belongs to the topmost part of the transport layer. We decided to hide the connection disruptions from the compression engine to maintain the compression engine’s state intact. Re-establishing the compression state would only decrease the compression performance during reconnections.

4.3 Extending TLS

The resiliency extension to TLS is negotiated using the TLS extension mechanism [2] in ClientHello/ServerHello messages. Similarly as in the SSH case, if both endpoints support this extension, they start buffering data and sending acknowledgement messages. In the TLS case, the acknowledgement messages contain the number of application data bytes received instead of the TLS record sequence numbers. We chose this approach since the reconnection handshake is based on the abbreviated TLS handshake which resets the sequence numbers back to zero.

The reconnection exchange is shown in Figure 2. The client indicates that it wants to continue an existing connection by including a connection identifier in the ClientHello message. The ClientHello/ServerHello messages are followed by an abbreviated handshake (based on the normal TLS session resumption handshake) which verifies that the parties have remained the same and establishes fresh session keys.

After this, the endpoints tell how much data needs to be retransmitted, and retransmit the lost data, if any.

It is important to note that while the cryptographic handshake re-uses an existing TLS feature called “session resumption”, there is an important difference. TLS session resumption is a feature of the TLS handshake protocol which caches the results of expensive public-key operations. It is a performance optimization and is independent of the actual data transfer (the TLS record protocol). Thus, it does not enable a client to continue an existing connection that was for some reason broken.

4.4 Security analysis

Making SSH and TLS sessions resilient to disconnections could introduce new security vulnerabilities. However, we believe that the extensions presented in this paper provide the same level of security as the situation when new SSH and TLS sessions are initiated to handle disconnections. In this section, we provide a high-level analysis of our protocol extensions. A complete security analysis of our protocol is beyond the scope of this paper.

In our extensions, all messages are authenticated using shared keys created during the initial SSH or TLS protocol exchange. Thus, an attacker cannot spoof or modify the reconnect messages. Replay attacks are not possible, since the first SSH and TLS key exchange messages include fresh nonces that are covered by a MAC later during the handshake.

Since the extensions require the endpoints to buffer data that has not been acknowledged, the amount of resources needed by a single SSH or TLS session is increased. Thus, the work required for a denial of service attack against a server (by creating a large number of sessions) may be less than in normal SSH or TLS. However, in most cases the buffers are likely to represent only a small share of the resources, and thus, denial of service resistance is not significantly changed.

5 Implementation considerations

In this section, we analyze the implications of resilient connections for SSH/TLS client and server side implementations.

5.1 When to reconnect and which interface to use?

Resiliency against connection disruptions brings a new challenge to client and server side implementations of both protocols. On the client side, the challenge is to determine when to start the reconnection procedure. Some options include the following:

1. A manual request; e.g., a user could click a "reconnect now" button in the application user interface.
2. Automatically when the device is brought up from a "suspended" power management state.
3. Whenever the current TCP connection is broken.
4. Whenever a more preferred network interface is available.
5. Probably several more options exist.

In addition to deciding when to reconnect, there may be multiple interfaces available: which of these should be used to establish the connection?

To ensure easy deployability, the solution should depend only on tools and APIs commonly available on the deployment environment and not require any additional software on the client machine.

Therefore, we decided to simply rely on the operating system's source address selection. In other words, we leave it to the operating system to decide which local interface should be used when a TCP connection is established, and initiate reconnection when the operating system's decision changes, or the current TCP connection is broken.

This raises the question of how to notice that the OS's source address selection policy has changed. In Windows, the Winsock API has a feature ("SIO_ROUTING_INTERFACE_CHANGE" socket option; see [14]) that allows the application to be notified of changes. BSD-based Unixes have "PF_ROUTE" routing sockets [27] and Linux has Netlink sockets [10] that also allow monitoring of routing table changes.

In the end, we implemented two different approaches. For OpenSSH, we used a routing socket to monitor routing table changes. In PureTLS, we settled for polling the OS in regular intervals to see if the preferred interface has changed. The polling can be done, for instance, by creating a "connection-mode" UDP socket, and reading the local address using the getsockname() API call (note that no UDP packets are actually sent). The approach was preferable in Java, since it avoided the need to have native and platform-specific code.

On the server side, a certain level of uncertainty is imminent too. For a server, the challenge is to determine the time to discard a session that waits for its client to reconnect. The difference to the client side is that the server must make the decision completely without the help of a user. Our vision is that the time a server is willing to keep resources allocated for a session without a connected client is a local policy issue. Different users may have different timeouts as well as servers with different loads may have different timeouts. For instance, one could assume a shared server is willing to maintain sessions shorter period of time than a server solely used by a single user. For the prototype implementations, we implemented a configurable server-wide timeout.

5.2 Server side concurrency

A common server design strategy is to create a new process or thread for each new client connection. While this often simplifies the server design, in this context concurrency becomes a complicating factor, since it results in a situation where the client's original session and reconnection request are handled by two different processes or threads.

A server designer has two options to choose from: either the new process finds the corresponding old process and passes the new TCP connection to the old process, or the other way around. Regardless of the choice, the server must maintain a table mapping sessions to processes for inter-process (or inter-thread) communication. In our implementations, the new process passes the new connection to the old process. Before passing the connection, the new process validates reconnection attempts. The validation requires contacting the old process, as the new process has no other access to the session keys. Once the new process has passed the connection to the old process, it exits.

Two reasons made us to choose the new process/thread to pass its state to the old process/thread. First, the new process has simply less state to pass: in practice, passing a file descriptor of a transport connection and sequence numbers to synchronize is sufficient. Second, besides the amount of state, the new process has state information that is easier to transfer. The old process can have such state that is impossible to pass across process boundaries. As an example, consider a TLS server process that creates child processes. In majority of platforms, it is impossible to pass child processes from a process to another—which would be a requirement if the old process passed its state to the new process.

In a multi-threaded server, implementing state passing is straightforward. However, if a server is implemented using concurrent processes, the above indicates that the server requires certain Inter-Process Communication (IPC) facilities:

1. An inter-process message channel to validate recon-

nection requests using keys stored in the old process.

2. An inter-process message channel to pass sequence numbers for synchronization.

3. A file descriptor passing mechanism to transfer a transport connection (socket) from the new process to the old process.

The required IPC facilities are realistic on most modern platforms, but they have often platform-specific features. Therefore, while the resiliency extensions are unlikely to prevent porting a server implementation to another platform, IPC mechanisms may add an extra twist to the porting process.

5.3 Atomic reconnections

Reconnection attempts must be atomic: the protocol state machine of an old connection must not become corrupted if an attempt fails. As discussed above, we designed the server implementations in a way that the new process transfers its state to the old process only after a reconnection request is determined to be valid. The approach has a positive side effect: the server side reconnection handling becomes atomic from the old process' point of view. If a reconnection request is invalid, the old process sees nothing.

Client implementations required similar atomic reconnection attempts: either a reconnection attempt succeeds or no state is affected. Unfortunately, implementing this in a general case can be challenging, as we learned in a hard way. A normal client implementation can modify global variables and data structures while connecting, and if connecting fails, it simply exits. Being a perfectly valid approach without resiliency extensions, this becomes challenging when the client implementation should behave in a deterministic manner in the case of connection failures.

Our observation was that it is tempting to modify a client implementation to behave as a multi-process or multi-threaded server: a fresh client process or thread attempts to reconnect and only once it succeeds, it passes its state to the old process or thread. In this way, the client implementation may dirty the new process or thread state but the attempt still remains atomic from the old process' point of view. As we implemented our clients in this way, we found out an unfortunate side effect: clients implemented in aprocess model require similar IPC facilities as the servers do. Our OpenSSH client was implemented as a multi-process and PureTLS client as multi-threaded client.

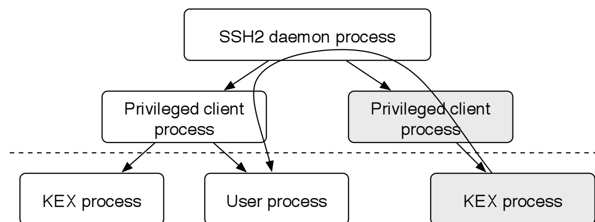

Figure 3: OpenSSH separates privileged processes (above dashed line) and less privileged processes (below dashed line). Grey boxes depict the processes processing a reconnection request.

5.4 Interface to higher layers

SSH transport layer protocol and TLS are not useful alone; they are always used together with some higher layer protocol. TLS is used with many different applications, while in the SSH case, the higher layer protocols are the SSH connection and user authentication protocols.

In general, we would like to change the interface offered by TLS and SSH transport layer protocol as little as possible. However, some changes and/or enhancements may be desirable. For instance, in the TLS case, some applications may be interested in knowing when a connection is no longer working, when a reconnection has happened, or even initiating reconnection.

Another set of issues arises from the fact that the IP addresses and port numbers used by the TCP connections may change when reconnecting. If the application uses these values for some other purpose than just sending packets, it may want to know when they change. For instance, OpenSSH can be configured to allow connections only from certain IP addresses. Similarly, a Java application can retrieve the addresses using Socket object methods such as getInetAddr(), and use them for, e.g., access control. Thus, it would be useful to have callbacks that allow the application logic to be notified when the addresses change.

These changes in the higher layer interfaces may require small modifications to OpenSSH and applications that use PureTLS. However, we have not yet implemented or further explored these modifications in the current versions of our prototypes.

5.5 OpenSSH

OpenSSH implements privilege separation to limit the effects of possible programming errors [17]. In the privilege separation, a privileged server daemon process uses less privileged processes to interface with clients. Less privileged processes then communicate with the privileged process through a monitor that protects the priv-

ileged process. Figure 3 depicts how OpenSSH forks (straight arrows) a separate process to do the key exchange and user authentication. Once the KEX process is done, the OpenSSH server forks yet another process to actually serve the client. Only the last process runs under the user's identity.

The OpenSSH privilege separation requires state serialization and passing across process boundaries: different processes perform the key exchange and the actual connection serving. After the key exchange, the KEX process serializes its key material together with information about the agreed algorithms and passes the state to its privileged parent process. The parent process then forks the actual connection serving process and passes the state further there.

It turned out that for both the OpenSSH server and client implementations, privilege separation facilities based on Unix socket pairs were enough to provide the required IPC facilities once they were extended to pass file descriptors. No additional authentication between processes was necessary either; the Unix socket pairs are invisible beyond a process and its child processes. On the server side, facilities provide atomic reconnections and the transfer of a new connection to an old process. On the client side, they only guarantee atomic reconnections as discussed earlier.

On the server side, we decided to transform the main daemon process into a message broker as it was the only common factor between all processes. The curved arrow in Figure 3 depicts how a connection together with synchronization information actually travels through several processes via the main daemon process, from the new process eventually to the old process.

5.6 PureTLS

In PureTLS, most of the implementation complexity comes from the requirement to keep the objects visible to the application (such as Socket, InputStream and OutputStream instances) unchanged over reconnections.

For Socket, this required creating an additional layer of indirection: a new Socket instance that forwards the method calls to the "real" underlying socket. Fortunately, PureTLS already contained this kind of indirection layer, and only small modifications were needed to allow changing of the underlying socket on-the-fly.

6 Evaluation

In this section we present measurement results of reconnection transactions for both protocols and discuss the complexity of implementations. In this paper, we did not focus on the performance optimizations. Instead, the main purpose of our evaluation is to show that our prototypes work and their performance is adequate for the intended use (relatively long disconnections).

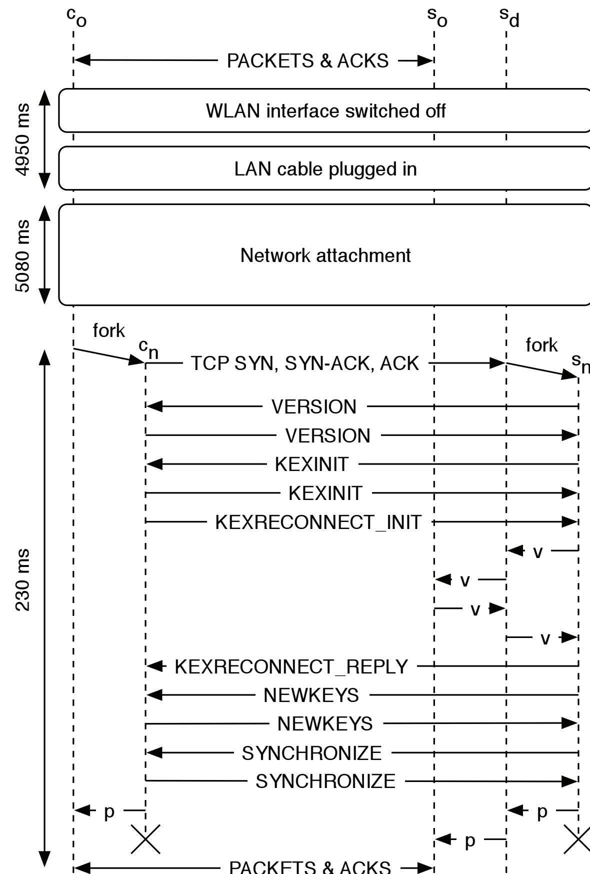

Figure 4: SSH processes involved in reconnecting a session.

6.1 Measurements

One of our main assumptions behind the design principles was that typical connection disruptions last a relatively long time. Therefore, we constructed one such scenario: a user manually switches from Wireless LAN to wired Ethernet. While the access to Internet from both networks goes through different NAT boxes, the user still expects his connections to survive from a changing IP address and NAT box. We conducted a set of measurements to validate the hypothesis and measure the actual expected length of typical reconnections.

In our scenario, the user downloads a large file from a remote server, either over SFTP or TLS. First, a user's laptop is attached to a wireless access point, but then the user decides to connect it to a fixed LAN to access remote services not available through the restricted public WLAN. Switching the access point requires, besides

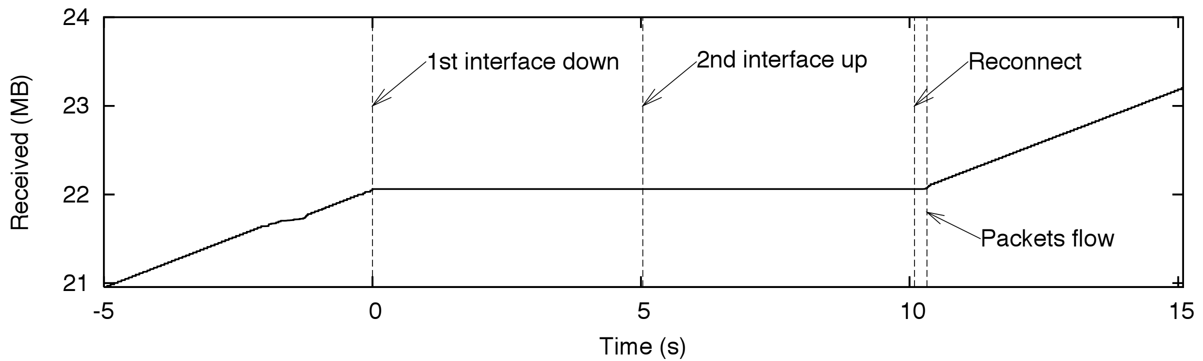

Figure 5: Progress of an SFTP transfer before, during, and after reconnection.

plugging an Ethernet cable, also turning off the WLAN interface; otherwise the laptop operating system keeps the WLAN interface as its primary interface.

We expect that typical disconnections would last significantly longer than the couple of seconds in these measurements. For example, we have used the extensions to keep SSH sessions alive while a suspended laptop is carried from the office to home.

OpenSSH measurements

In the OpenSSH tests, the remote download server was on the Internet and the round-trip time to the server was 10 ms through both WLAN and LAN. The SFTP client run on Mac OS X 10.4, while the SSH server was run on Linux.

As described in Section 5.1, the client can start reconnection not only when the TCP connection breaks, but also when the preferred source address has changed. Our OpenSSH extension uses a separate process to monitor the routing tables of the operating system. Once this process realizes that the route to the server has changed, it sends a signal to other processes that handle the actual reconnection.

Figure 4 represents the reconnection from a process viewpoint. On the client and server sides, temporary processes (c_n and s_n, respectively) handle the reconnection and old processes (c_o and s_o) receive the new transport connection only when it is time to resend lost packets. The main daemon process (s_d) only brokers messages between processes. In the figure, arrows titled as 'v' depict the inter-process validation messaging, while 'p' arrows depict the actual state passing.

Figure 5 shows the number of bytes an SFTP client has received as a function of time. The test user turned off the WLAN interface at time zero. The user quickly plugged a wired LAN cable in; finding the cable and inserting it to the laptop took less than three seconds. While it did not take that long to request an IP address (in the figure interface is up once it has an IP address), the graph illustrates how long it actually took before the network attachment was completely over from the SFTP point of view. Before the SFTP client receives a signal from the routing table monitoring daemon, 5 seconds has passed since the wired LAN interface came up. The actual reconnection then takes only about 200 ms before the file download continues.

PureTLS measurements

In the PureTLS tests, the client run on Linux and the server on Windows XP; the round-trip time to the server was around 1 ms.

Figure 6 shows the number of bytes received as a function of time. In this case, the network disruption lasted 5.5 seconds, and recovering from it took about 0.5 seconds. The differences compared to the OpenSSH case are explained mainly by how the reconnection is triggered (see Section 5.1).

Acknowledgment overhead

Figures 5 and 6 show only the downlink traffic and do not contain the additional network traffic caused by the session layer acknowledgements. However, this traffic is tolerable, and does not necessarily generate additional IP packets since the SSH/TLS ACKs can fit in the same packets as TCP ACKs.

Our OpenSSH acknowledgment implementation was suboptimal, since it acknowledges every received SSH transport layer message. While a single SSH ACK payload consumed only 5 bytes of space (packet type and 32-bit sequence number), the minimum cipher block sizes and MAC together increased the total size of ACK messages; a single ACK, with default OpenSSH configuration, consumed 32 bytes in total. Despite this suboptimal implementation, the extra traffic caused by the ACKs was more than acceptable, since the SSH transport layer messages can be up to 32 kilobytes: the ACK traffic amounted to less than 0.6% of the whole bandwidth. The PureTLS implementation does not acknowledge all records, but instead attempts to send ACKs at the same time as application data; the overhead figures were comparable to the OpenSSH case.

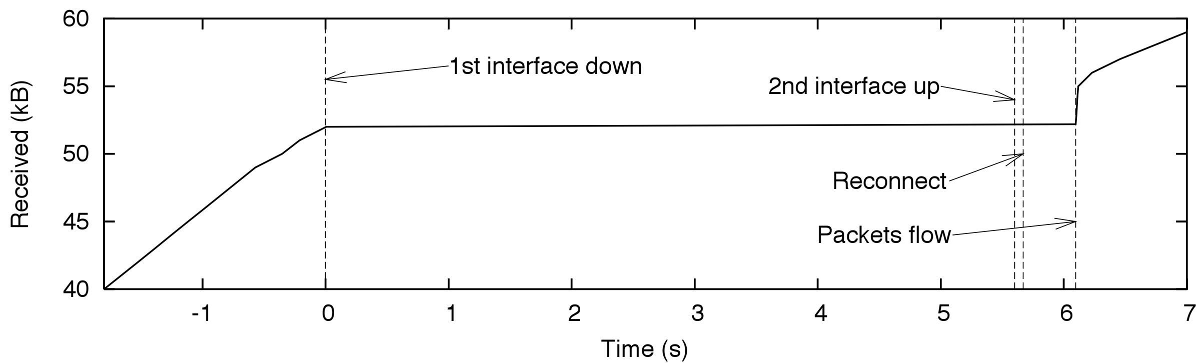

Figure 6: Progress of an TLS transfer before, during, and after reconnection. The temporary rapid speed-up after reconnection is caused by an implementation anomaly: the file transfer application uses a significantly smaller block size than the retransmissions done by PureTLS.

6.2 Implementation complexity

In-band resiliency signaling helps deployment, but it has an obvious extra cost: server and client applications must be modified. Next, we will briefly discuss the required implementation effort in the light of experiences from OpenSSH and PureTLS.

Our OpenSSH extension required about 2,200 lines of code. Over half of this code is related to passing state information and socket handles between the different processes. On the other hand, implementing explicit acknowledgments and buffering required relatively little effort, since much of existing OpenSSH functions were reusable as such. The OpenSSH implementation required roughly a man month of efficient work time, which included one complete refactoring.

The PureTLS extension was slightly simpler (about 1,000 lines of code) since Java's inter-thread communication facilities were much easier to use.

7 Conclusions

True *session continuity* in today's Internet requires not only handovers, but also gracefully handling long periods of disconnected operation. The contribution of our paper is three-fold. First, we have identified design principles that emphasize *deployability* and address some of the challeges in previous work. Second, following these principles, we have extended the SSH and TLS protocols to support resilient connections. Third, we have analyzed implementation issues faced when adding the functionality into two existing software packages.

Our three design principles are as follows: mechanisms for providing session continuity (a) should not place additional requirements for the network, (b) must allow incremental deployment, both providing benefits to early adopters and interoperating with legacy endpoints, and (c) should not require changes in operating system or third party libraries.

Our experience with the SSH and TLS extensions indicates that these design principles mandate certain protocol features, the most important one being in-band signaling. Furthermore, the protocol needs to support extension negotiation and explicit close messages, and has to be extended with explicit acknowledgements for transferred data.

The required extensions to the TLS and SSH protocols were relatively simple. In our case, we embedded the resiliency negotiation into the initial connection setup messages in a backwards compatible manner. In addition, both protocols execute mutual authentication while reconnecting simply by proving the possession of the shared secret of a suspended session.

In the implementations, handling the server side concurrency was clearly the most challenging part. The process (or thread) that is handling the reconnection request must find the corresponding old process, since only the old process can validate the request. After a successful validation, the new process must pass the connection state and the TCP socket to the old process. This translates into inter-process or inter-thread communication mechanisms. Moreover, while dividing functionalities between the new and old processes, we found out that a new process should prepare a reconnection attempt and only alter the state of the old process after the reconnection attempt has succeeded. This simplified the implementations considerably.

While our paper has focused on addressing deployment challenges, deployability remains a difficult concept. Much of existing work on mobility has focused on issues easy to measure and compare, such as handover performance. Deployability in general, as well as approaches to compare it, have received less attention, and clearly, more work is needed to better understand how

different protocol design choices affect deployability.

Acknowledgments

The authors would like to thank N. Asokan, Dan Forsberg, Andrei Gurtov, Tobias Heer, Janne Lindqvist, and Pekka Nikander for their comments and suggestions. Helpful comments were also provided by Tero Kivinen who said he had planned a similar extension to SSH several years ago (however, no additional information is available about this work). Finally, we thank the anonymous reviewers and our shepherd, Stefan Saroiu, for their insightful comments.

References

[1] Steven M. Bellovin. The "Session Tty" Manager. In *Proceedings of the Summer 1988 USENIX Conference*, pages 339–354, San Francisco, June 1988.

[2] Simon Blake-Wilson, Magnus Nystrom, David Hopwood, Jan Mikkelsen, and Tim Wright. Transport Layer Security (TLS) Extensions. RFC 3546, IETF, June 2003.

[3] Tim Dierks and Christopher Allen. The TLS protocol version 1.0. RFC 2246, IETF, January 1999.

[4] Daniel Duchamp. The discrete Internet and what to do about it. In *2nd New York Metro Area Networking Workshop*, New York, NY, September 2002.

[5] Richard Ekwall, Pétér Urbán, and André Schiper. Robust TCP connections for fault tolerant computing. *Journal of Information Science and Engineering*, 19(3):503–516, May 2003.

[6] Robert Elz and Paul Hethmon. Extensions to FTP. Work in progress (IETF Internet-Draft, draft-ietf-ftpext-mlst-16), September 2002.

[7] GNU Screen. http://www.gnu.org/software/screen/, 2006.

[8] Christian Huitema. Multi-homed TCP. Work in progress (draft-huitema-multi-homed-01), May 1995.

[9] Michael Kaminsky, Eric Peterson, Daniel B. Giffin, Kevin Fu, David Mazières, and M. Frans Kaashoek. REX: Secure, Extensible Remote Execution. In *Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, June–July 2004.

[10] Andi Kleen et al. rtnetlink, NETLINK_ROUTE – Linux IPv4 routing socket. Linux Programmer's Manual, man page rtnetlink(7), 2000.

[11] Björn Landfeldt, Tomas Larsson, Yuri Ismailov, and Aruna Seneviratne. SLM, a framework for session layer mobility management. In *Proceedings of the 8th International Conference on Computer Communications and Networks (ICCCN '99)*, Boston, MA, October 1999.

[12] Ari Luotonen. Tunneling TCP based protocols through Web proxy servers. Work in progress (draft-luotonen-web-proxy-tunneling-01), August 1998.

[13] Yun Mao, Björn Knutsson, Honghui Lu, and Jonathan M. Smith. Dharma: Distributed home agent for robust mobile access. In *Proceedings of IEEE INFOCOM 2005*, Miami, FL, March 2005.

[14] Microsoft Corporation. Windows sockets 2. MSDN Library, http://msdn.microsoft.com/library/en-us/winsock/winsock/windows_sockets_start_page_2.asp, 2006.

[15] Tadashi Okoshi, Masahiro Mochizuki, Yoshito Tobe, and Hideyuki Tokuda. MobileSocket: Toward continuous operation for Java applications. In *Proceedings of the 8th International Conference on Computer Communications and Networks (ICCCN '99)*, Boston, MA, October 1999.

[16] OpenBSD project. OpenSSH. http://www.openssh.org/, 2006.

[17] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, August 2003.

[18] Xun Qu, Jeffrey Xu Yu, and Richard P. Brent. A mobile TCP socket. Technical Report TR-CS-97-08, Computer Sciences Laboratory, RSISE, The Australian National University, 1997.

[19] Xun Qu, Jeffrey Xu Yu, and Richard P. Brent. A mobile TCP socket. In *Proceedings of the IASTED International Conference on Software Engineering*, San Francisco, CA, November 1997.

[20] Eric Rescorla. Claymore PureTLS. http://www.rtfm.com/puretls/, 2006.

[21] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[22] Henning Schulzrinne and Elin Wedlund. Application-layer mobility using SIP. *ACM SIGMOBILE Mobile Computing and Communications Review*, 4(3):47–57, July 2000.

[23] Alex Snoeren. *A Session-Based Approach to Internet Mobility*. PhD thesis, Massachusetts Institute of Technology, 2002.

[24] Alex C. Snoeren, Hari Balakrishnan, and M. Frans Kaashoek. Reconsidering Internet mobility. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany, May 2001.

[25] Ethan Solomita, James Kempf, and Dan Duchamp. XMOVE: a pseudoserver for X window movement. *The X Resource*, 11, July 1994.

[26] Jun-Zhao Sun, Jukka Riekki, Marko Jurmu, and Jaakko Sauvola. Channel-based connectivity management middleware for seamless integration of heterogeneous wireless networks. In *Proceedings of the the 2005 Symposium on Applications and the Internet (SAINT '05)*, Trento, Italy, January–February 2005.

[27] Sun Microsystems. route – kernel packet forwarding database. Solaris 10 Reference Manual Collection, man page route(7P), 2003.

[28] Tatu Ylönen and Chris Lonvick. The Secure Shell (SSH) protocol architecture. RFC 4251, IETF, January 2006.

[29] Tatu Ylönen and Chris Lonvick. The Secure Shell (SSH) transport layer protocol. RFC 4253, IETF, January 2006.

[30] Victor C. Zandy and Barton P. Miller. Reliable network connections. In *Proceedings of the 8th annual International Conference on Mobile Computing and Networking (MobiCom '02)*, Atlanta, GA, September 2002.

[31] Victor Charles Zandy. *Application Mobility*. PhD thesis, University of Wisconsin-Madison, 2004.

[32] Yongguang Zhang and Son Dao. A "persistent connection" model for mobile and distributed systems. In *Proceedings of the 4th International Conference on Computer Communications and Networks (IC-CCN '95)*, Las Vegas, NV, September 1995.

[33] Xiliang Zhong, Cheng-Zhong Xu, and Haiying Shen. A reliable and secure connection migration mechanism for mobile agents. In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops (ICDCSW '04)*, Tokyo, Japan, March 2004.