Making Secure TCP Connections Resistant to Server Failures

Hailin Wu, Andrew Burt, Ramki Thurimella
Department of Computer Science
University of Denver
Denver, CO 80208, USA
{hwu, aburt, ramki}@cs.du.edu

Abstract

Methods are presented to increase resiliency to server failures by migrating long running, secure TCP-based connections to backup servers, thus mitigating damage from servers disabled by attacks or accidental failures. The failover mechanism described is completely transparent to the client. Using these techniques, simple, practical systems can be built that can be retrofitted into the existing infrastructure, i.e. without requiring changes either to the TCP/IP protocol, or to the client system. The end result is a drop-in method of adding significant robustness to secure network connections such as those using the secure shell protocol (SSH). As there is a large installed universe of TCP-based user agent software, it will be some time before widespread adoption takes place of other approaches designed to withstand these kind of service failures; our methods provide an immediate way to enhance reliability, and thus resistance to attack, without having to wait for clients to upgrade software at their end. The practical viability of our approach is demonstrated by providing details of a system we have built that satisfies these requirements.

1. Introduction

TCP is neither secure nor can withstand server failures due to malevolent intrusion, system crashes, or network card failures. Nonetheless, today's information assurance requirements demand building software, networks and servers that are resistant to attacks and failures. While individual connections can be made secure from eavesdropping or alteration by such protocols as the Secure Shell protocol (SSH), the server that provides these services continues to be a single point of failure. This is an artifact of TCP's original design, which assumed connections should be aborted if either endpoint is lost. That TCP also lacks any means of migrating connections implies that there is no inherent way to relocate connections to a backup server. Thus any secure software built on top of TCP inherits the vulnerability of the single server as a point of failure.Combining TCP with a mix of public key and symmetric key encryption such as SSH or SSL addresses the protocol's general security deficiency. In this paper we extend these methods to increase the resiliency of secure connections to tackle server failures. Specifically, we show practical ways to migrate active SSH connections to backup servers that do not require any alterations to client-side software, including their client application software, operating systems, or network stacks, thus making this solution immediately deployable. These techniques are general and can be employed for other forms of secure connections, such as SSL, which is our next research goal.

Recently, the authors [4] presented techniques to migrate open TCP connections in a client-transparent way using a system called Jeebs (Jeebs, from the film Men in Black, being the alien masquerading as a human who, when his head is blown off, grows a new head). Using this system, it is possible to make a range of TCP-based network services such as HTTP, SMTP, FTP, and Telnet fault tolerant. Jeebs has been demonstrated to recover TCP sessions from all combinations of Linux/Windows clients/servers.

The results in this paper are a natural extension of the recent results on TCP migration [4] to secure connections, with which the ordinary Jeebs implementation is unable to cope because of the very nature of their security. Our implementation for secure connections, SecureJeebs, consists of making simple, modular and secure extensions to the SSH software and placing a "black box" on the server's subnet to monitor all TCP connections for the specified server hosts and services, detect loss of service, and recover the TCP connections before the clients' TCP stacks are aware of any difficulty.

While great strides have been made in providing redundancy of network components such as load balancing switches and routers, and in proprietary applications such as used in database servers, a missing component in end-to-end fault tolerance has been the inability to migrate open TCP connections across server failures. Although neither these products nor SecureJeebs provide reliability if the whole cluster providing the service were to be involved in catastrophe such as an earthquake or fire, or if network components that are on the path of service were to fail, SecureJeebs eliminates servers as a single point of

failures. SecureJeebs is further distinguished from load balancing and other techniques in that it transparently and securely migrates secure connections that are in progress. This feature permits SecureJeebs to be used not only to enhance reliability of unreliable servers, but also to take production servers offline for scheduled maintenance without disrupting the existing connections.

Following an overview in Section 2 and discussion of related work in Section 3, we describe the necessary background in section 4 and present our techniques and the architecture of Jeebs in Section 5. We present a performance analysis in Section 6 and concluding remarks in Section 7.

2. Overview

2.1. Migration

Recovering TCP sessions that are about to abort due to loss of the server requires two components: (1) A monitor, to record pertinent information about existing connections and detect their imminent demise; and (2) a recovery system that can perform emergency reconnection to a new server that will take over the connection. Each is described briefly below.

unacknowledged data, and any prior data that may be required for recovery purposes (such as initial requests). Further, the monitor observes the health of each connection to detect imminent failure. Health monitoring and server crash detection use standard techniques as described elsewhere in the literature [3, 6, 12]. SecureJeebs is installed on the server's subnet to monitor and recover connections, thus is currently limited to recovering what appear to be local server crashes. Packets are logged at the TCP level by a sniffer, thus potentially suffering from missed packets, though mitigating this deficiency has been addressed in [4]. Recovery of TCP state is handled via a passive recovery daemon on a monitoring server, and application state is migrated using simple, per-protocol recovery modules described briefly here and fully in [4]. Connections are recovered to a backup server (which may co-exist with the recovery server or be a separate system on the subnet) as shown in the figures below.

When an IP number is deemed in need of migration, all connections to that server are restored by the recovery system. The recovery system takes over the IP number of the designated server and initiates recovery of each connection. Connection state is restored using simple per-service recovery procedures. There are three styles of

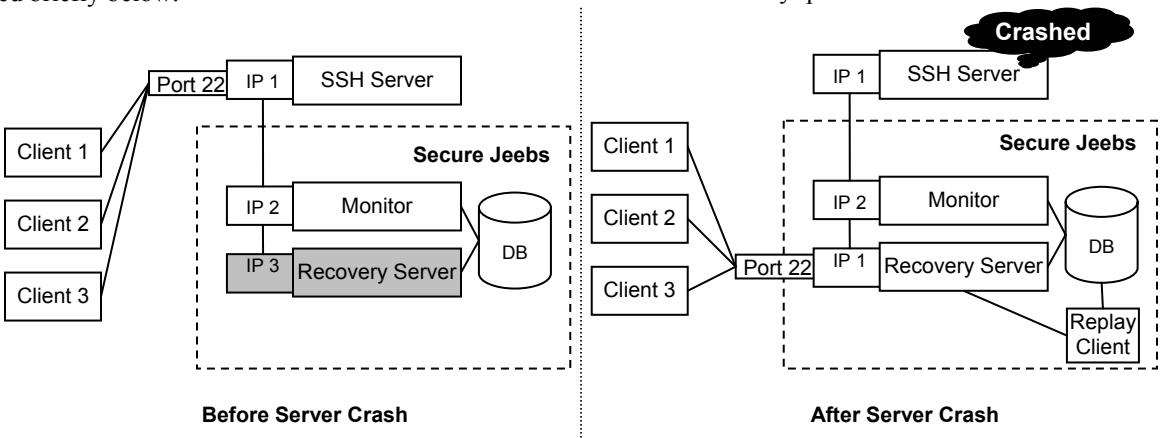

Figure 1. Illustration of Secure Jeebs. Monitor detects server crash and starts recovery server and replay client. After CPR ends, all the original sessions are recovered.

The monitor operates by logging traffic from the server host it is watching. The granularity of recovery is at the IP number level. The monitor can be further selected to only watch certain ports, but since the entire IP number is migrated to a new server, all ports on that IP number should be monitored in practice (However, since virtual IP numbers are used in practice, specific services can be isolated so that they are the only services using a given IP number. Thus individual services can be migrated if they are the only services using that virtual IP number). Logging includes the TCP state information,

recovery: Standalone, where a new piece of software is written specifically to handle connections in progress (with new connection requests being serviced by a copy of the original daemon for that service); Integrated, where the existing service daemon on the recovery system is modified to understand how to adopt stranded connections (in addition to handling new requests); and Proxy, where a small, programmable daemon interposes itself between the client and a backup copy of the original service daemon, such that it can replay the necessary parts of the original connection to bring the new server up to the point the original server failed, then acts in a pass-through mode

while the new server finishes the connection. Session keys and other sensitive data needed to ensure the integrity of secure connections are likewise migrated in a secure manner as described in detail in section 5.

The difficulties involved in migrating a secure connection such as SSH primarily arise from exporting and importing various session keys securely and efficiently, and making the state of the cipher consistent. In addition, such protocols are specifically designed to prevent various attacks such as man-in-the-middle or replay attacks. We have overcome these obstacles and devised several efficient, secure and reliable migration mechanisms which are successfully implemented in our testbed. Figure 1 illustrates one such approach: Controlled Partial Replay (CPR).

2.2. Preserving Security

It is always a legitimate concern whether a modification to a secure protocol such as SSH weakens the original security. We argue that the methods proposed here are sound from this perspective.

First of all, as explained in detail in section 5, the changes we make are all client-transparent protocol-level changes that are consistent with the regular operation of SSH. The main changes are to the key exchange phase on the server side: we export several entities so that if there were to be a failure, the recovery server can recreate the original session. The exported entities include client's payload of SSH_MSG_KEXINIT message, prime p, and generator for subgroup g, server's exchange value f and its host key. The export operation is independent of the regular behavior of SSH server, in other words, it does not interfere with the normal packet exchange between client and server at all, thus it does not open new holes within the transport layer or connection protocols.

Secondly, all the entities for export, including those mentioned above, the last block of cipher text (details in 5.3.1), and message sequence number (details in 5.3.2), are encrypted using the recovery server's public host key. In addition, a message digest is appended for integrity check, and we further provide non-repudiation by signing the message digest using the original server's private key. With these measures, only the recovery server can successfully decrypt these quantities with the assurance that they are from the original server and not tampered with during the export/import process.

Thirdly, access control is in place to make sure that after the original server exports those aforementioned quantities to the database, only the recovery server is allowed to access them. This is possible because to the original SSH server, the recovery server is a known identifiable entity, i.e., the database can authenticate the recovery server before granting access.

Finally, all these extra exporting and importing happen in a dedicated point-to-point physical channel and is totally transparent to the client or the third party. From the third party's point of view, the CPR is just like a regular SSH session, except that it is short and the recovery server promptly resumes connection to the original client at the end of it.

3. Related Work

Our primary motivation is to provide tools that enhance reliability, which can easily be attached to the existing infrastructure without making any modifications to the client. This contrasts with previous solutions whose purpose is to provide continuity of service for mobile clients [9,14,18,23], perform dynamic load balancing using content-aware request distribution [5,15], do socket migration as part of a more general process migration [7-8], or build network services that scale [13]. The difference in motivation between our work and the previous methods presents special challenges and has subtle effects on the proposed architecture.

Much of the previous work proposes modifications to TCP [1,2,16-17, 23-25] thus making client transparency difficult, if not impossible. One way to make these solutions work with legacy clients is by interposing a proxy: it uses the new protocol by default, but switches to TCP if that is the only protocol the client understands. This approach in general has a few drawbacks. First and foremost, instead of removing the original single-point of failure, it introduces another. These methods also create an additional point of indirection, potentially impacting performance of normal communication and potentially introducing an additional security vulnerability.

One way to achieve fault tolerance is to build recovery machinery into the server and develop clients to take advantage of this feature. The feature may be user controlled, such as the "REST" restart command in FTP, or it may be hidden from user control. An example of such a methodology is Netscape's SmartDownload that is currently gaining some popularity [10]. This approach requires modifying the clients and servers, and recoding of applications.

To the best of our knowledge, we are the first to describe a method to migrate a secure TCP connection in a client transparent way.

4. Background

SSH is a protocol for secure remote login and other secure network services over an insecure network. SSH encrypts all traffic to effectively eliminate eavesdropping, connection hijacking, and other network-level attacks. Additionally, it provides myriad secure tunneling

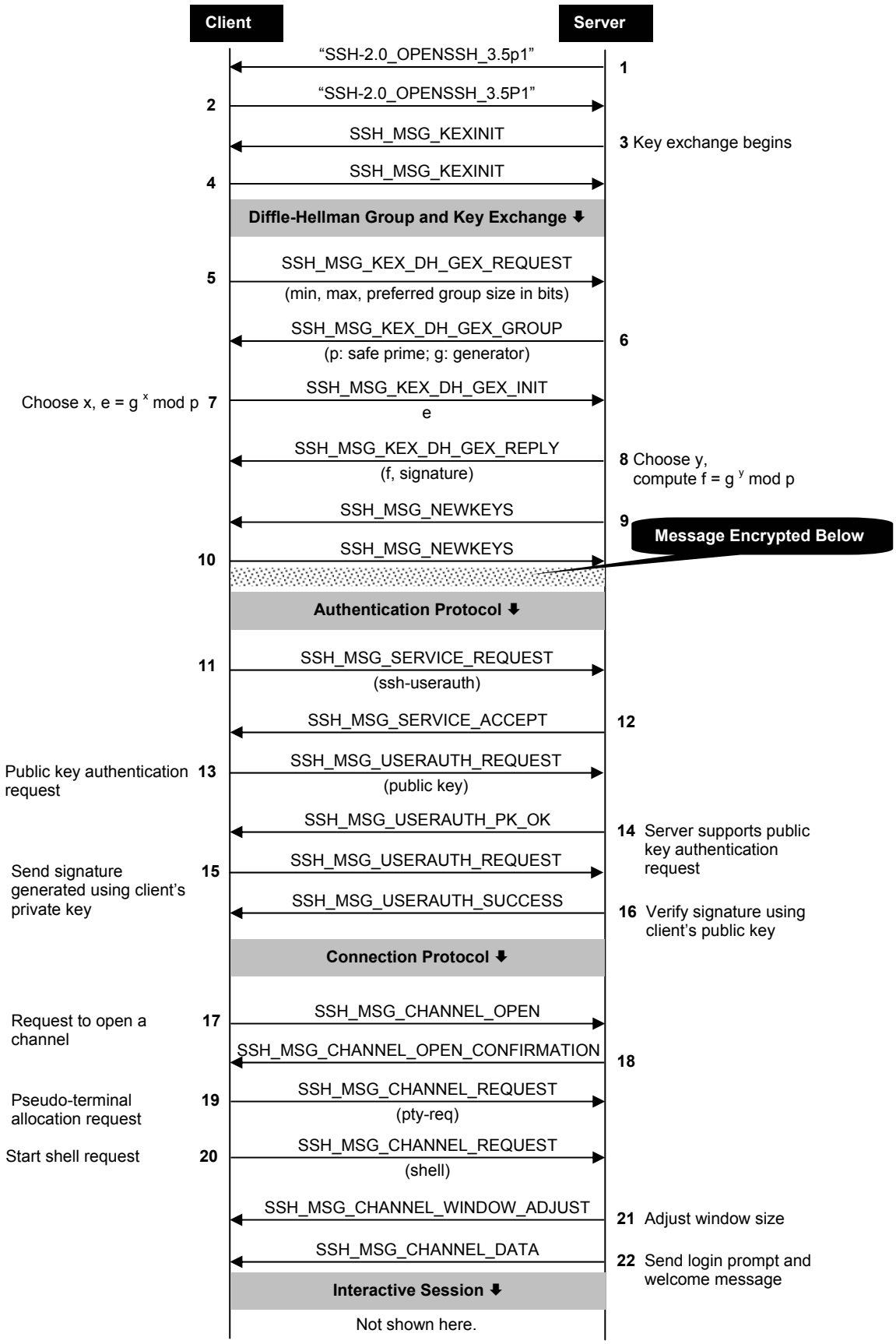

Figure 2. SSH protocol sample packet exchange

capabilities and authentication methods. With an installed base of several million systems, it is the de-facto standard for remote logins and a common conduit for other applications. Increasingly, many organizations are making SSH the only allowed form of general access to their network from the public Internet (i.e., other than more specialized access such as *via* HTTP/HTTPS).

SSH consists of three major components: The Transport Layer Protocol [19] provides server authentication, confidentiality, and integrity with perfect forward secrecy. The User Authentication Protocol [20] authenticates the client to the server. The Connection Protocol [21] multiplexes the encrypted tunnel into several logical channels. For further details refer to [19-22].

We will briefly show how SSH works by demonstrating protocol level packet exchange during a typical session in Figure 2 (previous page).

When the connection has been established, both sides send an identification string in steps 1 and 2. After exchanging the key exchange message (SSH_MSG_KEXINT) in steps 3 and 4, each side agrees on which encryption, Message Authentication Code (MAC) and compression algorithms to use. Steps 5 through 8 consist of Diffie-Hellman group and key exchange protocol which establishes various keys for use throughout the session. It is the focus of our recovery research and will be elaborated further in section 5.

Following the successful key setup phase, signaled by the exchange of new keys message (SSH_MSG_NEWKEYS) in steps 9 and 10, messages are encrypted throughout the rest of the session.

Steps 11 to 16 illustrate user authentication protocol, in particular, the public key authentication method. Steps 17 and above illustrate the SSH connection protocol, which provides interactive login sessions, remote execution of commands, and forwarded TCP/IP connections. Figure 2 also shows opening a remote channel (17, 18), and pseudo-terminal and shell start requests (19, 20). After the server sends the login prompt and greeting messages, the client begins transferring data, entering interactive session.

5. SSH Recovery

5.1. Overview

We have investigated two feasible approaches, a full replay "Proxy" based approach and Controlled Partial Replay approach (CPR). After a brief discussion of the Proxy approach, this paper will focus on the CPR approach because of its performance benefits.

A Proxy style recovery daemon is a standalone piece of software with some understanding of the protocol whose sessions are to be recovered. However, it does not listen on any original service port, only on a port dedicated to recovery requests. When a recovery request arrives, the Proxy opens a new connection to an existing service daemon on a designated recovery host and replays most of the entire initial part of the original conversation between the client and original server, a conversation it retrieves from the monitor's database. After replaying the connection up to the point it was (almost) disrupted, the Proxy simply acts as a two-way pipe between client and new server. In recovering an SSH daemon, the Proxy recovery daemon would invoke a new sshd process then replay the entire original conversation to the recovery SSH daemon (acting as if it were the client), so that the new sshd could advance the state of the encryption engine to match that of the original and now defunct sshd. (The new sshd would have itself been modified to use the same encryption data as the original, as is discussed below, in that this is a modification necessary to both approaches.)

In the CPR approach, once the monitor detects server failure, the CPR daemon starts an SSH recovery server—a modified copy of the regular SSH server—then performs a brief replay of the client process that mimics the original SSH client in that it sends and receives the same sequences of the same packets in the same order as the original client. (These are in no way sent to or seen by the original client.) The recovery server is modified to generate the same set of encryption/decryption/MAC keys as the original session, as described below. This replay proceeds until authentication and connection are successful and the recovery server arrives at the same connection state as the original server was. The recovery client then ends the partial replay process by sending to the recovery server a user-defined message "SSH_USEFUL_REPLAY_END" which contains TCP/IP kernel parameters (sequence numbers, port numbers, IP addresses, etc.). Upon receiving this message, the recovery server restores these TCP/IP kernel parameters via a small kernel module loaded on the recovery system, so that the sshd process invisibly resumes the connection to the original client, thus completing the recovery process. The recovery client terminates itself afterwards.

In order for our CPR to work, we need to guarantee that the SSH recovery server as well as the recovery client can derive the same set of keys as those of the original session, and in a secure manner. In addition, we need to address protocol specifics which normally are designed to prevent replay from happening in the first place. We will show that our recovery approach not only works, but also does not lessen the security of SSH.

Lastly, while the modifications needed for recovery must be made to the SSH software on the server side, the changes are not complex (in that they address the protocol and not the specific implementation), and can be easily expressed as simple patches for existing versions of SSH; ultimately these could be incorporated directly into future SSH revisions as standard functionality or optional modules.

5.2. Reproducing the keys

The first step for the recovery server and recovery client to reproduce the keys is to force the recovery server to send the same SSH_MSG_KEXINIT in step 3 in Figure 2. This is because SSH_MSG_KEXINIT contains 16-bytes of random data generated by the sender and used as an input to future key and session identifier generating functions.

Therefore, we need to make sure that the recovery server generates the same SSH_MSG_KEXINIT as that of the original server, so that both the recovery client and server can derive the same set of keys as those of the original session. This is accomplished in a straightforward manner: we first modify the original server so that it exports the 16-byte random number, after encrypting it using the recovery server's public key (and signing with the original server's private key); this is exported through secure channel to the recovery server for later use, should recovery be called for. During the CPR process, instead of generating the random numbers on the fly as is the normal mode of operation for SSH, the recovery server imports the saved value, decrypts it using its private key (validates the signature), and finally produces the same SSH_MSG_KEXINIT.

As stated earlier, the Diffie-Hellman group and key exchange is a secure key exchange method that produces a unique set of keys. The current SSH Transport Layer Protocol only designates Diffie-Hellman key exchange as the required method. However, the Diffie-Hellman *group and key* exchange method offers better security because it uses a different group for each session, and is the default key exchange method deployed in OpenSSH. Therefore, without loss of generality, it is assumed herein.

In Figure 3, we expand step 5-8 of Figure 2 to illustrate this key exchange method in detail. Note that || denotes string concatenation.

In step 5 of Figure 3, the client sends min, n, and max to the server, indicating the minimal acceptable group size, the preferred size of the group and the maximal group size in bits the client will accept. In step 6, the server finds a group that best matches the client's request, and sends p and g to the client. In step 7, client generates a random number x. It computes $e = g^x \bmod p$, and sends "e" to server. In step 8, server generates a random number y and computes $f = g^y \bmod p$. When the server receives "e" it computes $K = e^y \bmod p$, and $H = \text{hash}(V_c || V_s || I_c || I_s || K_s || \text{min} || n || \text{max} || p || g || e || f || K)$ where

V_c & V_s --client's & server's version strings, resp
K_s -- server host key
p -- safe prime
I_c & I_s -- the payload of the client & server's SSH_MSG_KEXINIT, resp
min & max --minimal & maximal size in bits of an acceptable group, resp
n -- preferred size in bits of the group the server should send
K -- the shared secret
g -- generator for subgroup
f -- exchange value sent by the server
K_S -- server certificate

Various encryption keys are computed as hash of a K and H and a known single character.

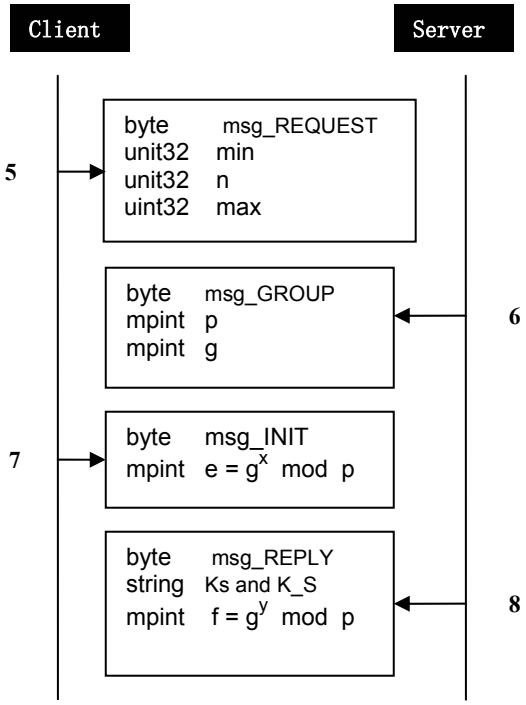

Figure 3. Diffie-Hellman Group and Key Exchange

Following the above description, we can conclude that the entities that are unique to each session that affect key generation are: V_c, V_s, I_c, I_s, K_s, min, n, max, p, g, e, f, K and H. In our CPR process, the recovery client replays the messages previously sent by the original client, thus V_c, I_c, min, n, max, e will be the same for the recovery session, but other items that are normally generated at run time by the server must be the restored as those originally used. Because the recovery server is only a slightly modified version of the original server, it will thus produce the same V_s. Therefore, the entities that we need to force the recovery server to duplicate in order to generate the same set of keys are: I_s, p, g, f, and K_s. We modify the original SSH server so that it encrypts these aforementioned entities using recovery server's public key, appends with message digest, signs and exports them to a secure network location. For the recovery server, instead of generating these host-specific entities dynamically, it reads

them in from the secure location, decrypts them using its private host key, verifies message digest and signature, and generates the same packets to be sent to client in step 6 and 8 in Figure 3. In doing so, the recovery server and the corresponding recovery client are guaranteed to produce the same set of initial IVs, as well as encryption and integrity keys, enabling our CPR to proceed.

5.3. SSH Transport Layer Protocol

Each SSH packet includes, respectively, 4 bytes in a packet length field, 1 byte in a padding length field, a payload field, and random padding. The encrypted packets have an additional MAC field at the end as described below. Packet format before and after encryption is depicted in Figure 4.

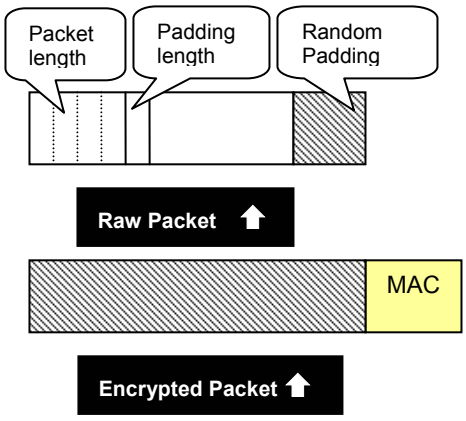

Figure 4. SSH packet format

5.3.1. Encryption

According to the export/import method we described in section 5.2.1, we can guarantee that the same encryption algorithms and identical set of keys will be used during CPR. However, for block ciphers, the previous block of cipher text, denote as C_i, is used as the random data that will be XOR'd into the next plaintext. This in essence, means that, though we start with the same sets of keys, because we are only doing a partial replay, we may still arrive at an inconsistent cipher context at the end of CPR. We have designed two approaches to solve this problem: (1) to modify the original SSH server to export the most recent C_i with every packet encryption and decryption, and to reset the cipher context of the SSH recovery server to C_i at the end of CPR; vs. (2) to modify the regular SSH server to securely export every raw packet, so that the cipher context can be advanced by applying encryption/decryption over all the saved raw packets. We implemented both of these two approaches and found that, as expected, the first approach is just as effective without the inefficiency of saving all the raw packets.

5.3.2. Data Integrity

As shown in Figure 4, each encrypted packet is appended with a Message Authentication Code (MAC) to achieve data integrity. MAC is produced according to the following formula:

MAC = mac_algorithm(key, sequence_number ||
unencrypted_packet)

where unencrypted_packet is the entire packet without MAC and sequence_number is an implicit 32 bit packet sequence number. The sequence number is initialized to zero for the first packet, and is incremented after every packet.

Of the three parameters to MAC, we observe here that the only entity that is unique to every packet in each session is the sequence_number. The regular SSH server is thus modified to securely export the latest sequence number after each packet send/receive operation. At the end of CPR, we reset the sequence_number of the recovery server as the latest one from the original session.

5.3.3. Random Padding

The random padding field consists of randomly generated bytes to ensure the total length of the raw packet is a multiple of the cipher block. Although the recovery server and original server generate different random padding for their packets, it is not necessary to alter the recovery server in order to reconcile this inconsistency. This is because both the recovery client and recovery server will derive the same encryption and MAC algorithms after the key exchange phase, as well as the same set of keys, which enables the recovery client to successfully decrypt any packet received from the recovery server and to proceed until CPR ends. The only ramification of different random padding is that the recovery server's cipher context, or the last block of cipher text (C_i), will be different from that of the original server. However, as explained in section 5.1, we will reset the cipher context of the recovery server at the end of CPR to make it consistent with the original server, thus making exporting and importing random padding field unnecessary.

5.3.4 Application State

Application state is recovered in a manner generally addressed in [4]. For a given application (such as a remote shell or a specific application invoked using SSH) a per-application recovery module is created. These are generally simple to create, and may be crafted from existing models. The primary issue in an application

recovery module is for it to monitor the original connection and extract relevant state from it. This can be restored by replay to an unmodified application daemon or by directly setting state into a daemon modified for that purpose. For example, highly non-deterministic applications like a shell session can display a list of previously executed commands for the user to choose to re-execute. More deterministic applications, such as FTP, can have their state replayed by a simple proxy client directly.

The only significant difference between recovery applications under Jeebs compared to SecureJeebs is that the recovery module must be connected into the SSH monitor, so it can decrypt the session's application communications to determine which are relevant state-setting messages, e.g., a CHDIR command in FTP, or gathering the list of commands executed for a login session. However, since the SSH software has been slightly modified for recovery purposes anyway, this is not a significant imposition.

6. Performance

We chose OpenSSH 3.5 and modified the source code to create both the regular and recovery SSH server. We conducted our experiments on several very modest machines (each an Intel Pentium 333 MHz with 128M memory and Intel Ethernet Pro 10/100B PCs running Red Hat 7.2 with a mySQL database).

The fundamental measure of success in this case is whether SSH connections can be restored before TCP's abort timer expires and the clients begin resetting connections. This value is established on the order of multiple minutes, with two minutes being the general minimum and nine minutes the common value. In our experiments, recovery even under load takes less than two minutes.

The following shows the time spent in a representative SSH recovery session:

Monitor alerts of server crash:	17:39:21
Recovery start:	17:39:26
IP take over and recovery server daemon started:	17:39:32
Recovery complete:	17:39:40

It takes approximately 11 seconds to discover a server crash, reset the virtual interface, and start a recovery daemon. The actual recovery process, which includes controlled partial replay, reading and decrypting the saved parameters, and resetting the recovery server's encryption cipher states, takes another 8 seconds. This is compared to observations that show a regular client login to server takes, on average, 3.2 seconds.

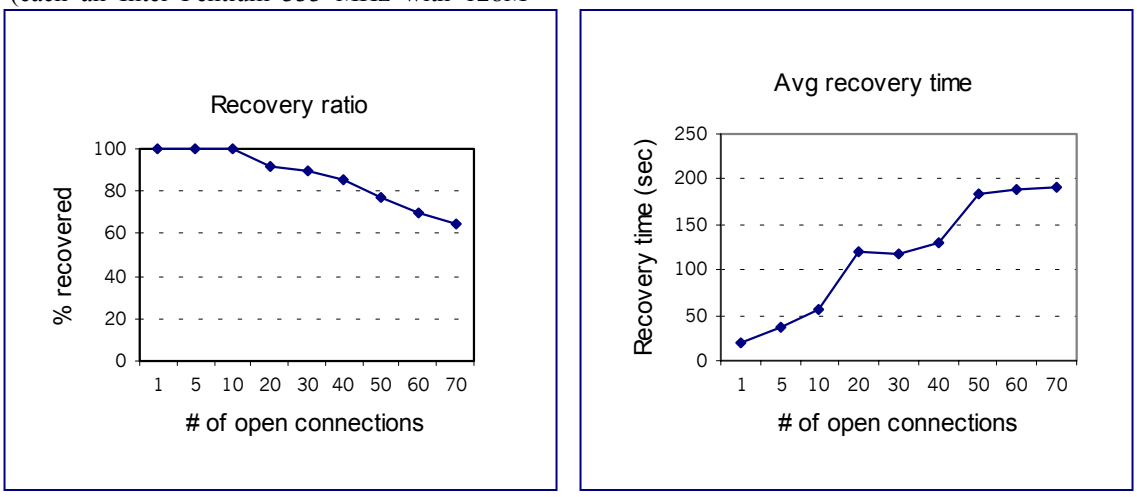

Figure 5. Recovery percentage and the average recovery time vs. # of open sessions.

Table 1. Recovery time for multiple concurrent sessions (time unit is second).

# of connections	1	5	10	20	30	40	50	60	70
Avg recovery time	19	37.4	57.2	120	118	129	184	189.6	191
standard deviation	0	0.89	2.68	8.4	13	17.3	54.8	53.49	55.61
median	19	38	58	124	123	134	171	173	169
shortest recovery time	19	36	51	101	88	97	54	99	91
longest recovery time	19	38	59	127	128	152	248	255	258

We observe that the recovery percentage drops from 100% with 20 or less simultaneous sessions, to around 60% recovered with 70 simultaneous sessions. This degradation is partially due the limitations imposed by a relatively obsolete hardware. Under 20~30 open sessions, we achieve the average recovery time within the two minute TCP timer expiration limit, again, determined by our inadequate hardware. It is obvious that more powerful systems can handle more demanding tasks such as larger number of concurrent logins, and thus likewise the recovery tasks presented in this paper. It is also possible that tremendously high network load could cause lost packets (as addressed more fully in [4], but this has not been found to be a limiting factor. Regardless of cause, recovery of most sessions, with some failures that would have failed anyway, may be preferable to losing all sessions.

7. Conclusions and Future Work

Until such time as secure TCP-based migration solutions are available on the hundreds of millions of existing systems, there will remain a need for client-transparent migration. The SecureJeebs system demonstrates how certain techniques can be deployed in a simple manner, without requiring changes to any clients. The simplicity and immediate applicability of the techniques demonstrated in this paper make SecureJeebs attractive for adoption in commercial product development.

We are currently extending this work in the following directions:

- Migrate HTTPS by proposing simple extensions to SSL
- Prove the applicability of the methods presented here to secure file transfer protocol (SFTP)
- Improve the recovery ratio under high load by employing more sophisticated recovery methods

Acknowledgements

We thank Sada Narayanappa for useful discussions on various modules, Profs. Leutenegger and Lopez for lending us the hardware to do experimentation, and Ocean Yang for her help with the figures.

References

[1] N. Aghdaie and Y. Tamir. Implementation and Evaluation of Transparent Fault-Tolerant Web Service with Kernel-Level Support. In Proc. of the 11th International Conference on Computer Communications and Networks (ICCCN 2002), Miami, Florida, October 14-16, 2002.

[2] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov. Wrapping Server-Side TCP to Mask Connection Failures. In Proc. of IEEE INFOCOM, Anchorage, Alaska, pp. 329-337 (April 2001).

[3] E. Amir, S. McCanne, and R. Katz. An Active Service Framework and its Application to Real-time Multimedia Transcoding. In Proc., ACM SIGCOMM '98, Sep. 1998.

[4] A. Burt, S. Narayanappa and R. Thurimella. Techniques for Client-Transparent TCP Migration. Submitted.

[5] Cisco Systems. Cisco Distributed Director. http://www.cisco.com/warp/public/cc/pd/cxsr/dd/tech/dd_wp.htm.

[6] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Cluster-based Scalable Network Services. In Proc. ACM SOSP '97, Oct. 1997.

[7] M. Haungs, R. Pandey, E. Barr, and J.F. Barnes. Migrating Sockets: Bridging the OS Primitive/Internet Application Gap. Manuscript. Available for download from http://www.cs.ucdavis.edu/~haungs/my_cv/

[8] B. Kuntz and K. Rajan. MIGSOCK: Migratable TCP Socket in Linux , M.S. Thesis, Information Networking Institute, Carnegie Mellon University, Feb. 2002.

[9] D. A. Maltz and P. Bhagwat. MSOCKS: An Architecture for Transport Layer Mobility. In Proc. IEEE INFOCOM, Mar. 1998.

[10] Netscape SmartDownload, http://wp.netscape.com/computing/download/smartdownload/ib/about.html.

[11] M. Orgiyan and C. Fetzer. Tapping TCP Streams. in Proc. of IEEE International Symposium on Network Computing and Applications (NCA2001), Boston, MA, USA, Feb. 2002.

[12] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware Request Distribution in Cluster-based Network Servers. In Proc. ASPLOS '98, Oct. 1998.

[13] A. E. Papathanasiou and E. V. Hensbergen. KNITS: Switch-based Connection Hand-off. In Proc. IEEE INFOCOM, Jun. 2002.

[14] X. Qu and J. Xu Yu and R.P. Brent. Implementation Of a Portable-IP System For Mobile TCP/IP. TR-CS-97-19, The Australian National University, Canberra, Australia, 1997.

[15] Scaling Next Generation Web Infrastructure with Content-Intelligent Switching, http://www.nortelnetworks.com/products/library/collateral/intel_int/l7_white_paper1.pdf

[16] C. Snoeren and H. Balakrishnan. An End-to-End Approach to Host Mobility. In Proc. 6th ACM MOBICOM, Aug. 2000.

[17] C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-Grained Failover Using Connection Migration. In Proc. 3rd USENIX Symp. on Internet Technologies and Systems (USITS), Mar. 2001.

[18] O. Spatscheck, J. S. Hansen, J. H. Hartman and L. L. Peterson. Optimizing TCP Forwarder Performance. IEEE/ACM Transactions on Networking, 8:2, pp. 146—157, 2000.

[19] SSH Transport Layer Protocol. http://www.ietf.org/internet-drafts/draft-ietf-secsh-transport-15.txt

[20] SSH Authentication Protocol. http://www.ietf.org/internet-drafts/draft-ietf-secsh-userauth-16.txt.

[21] SSH Connection Protocol. http://www.ietf.org/internet-drafts/draft-ietf-secsh-connect-16.txt
[22] SSH Protocol Architecture. http://www.ietf.org/internet-drafts/draft-ietf-secsh-architecture-13.txt
[23] R. R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. J. Schwarzberger, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. RFC 2960: Stream Control Transport Protocol, 2000.
[24] F. Sultan, K. Srinivasan, and L. Iftode. Transport Layer Support for Highly-Available Network Services. In Proc. HotOS-VIII, May 2001. Extended version: Technical Report DCS-TR-429, Rutgers University.
[25] F. Sultan, K. Srinivasan, D. Iyer, L. Iftode. Migratory TCP: Connection Migration for Service Continuity over the Internet. In Proc. of the 22nd International Conference on Distributed Computing Systems (ICDCS '02), July 2002.
[26] V.C. Zandy and B.P. Miller. Reliable Network Connections. In Proc. 8th Annual ACM/IEEE International Conference on Mobile Computing and Networking, pages 95–106, Atlanta, Georgia, September 2002.