

SSLACC: A Clustered SSL Accelerator

Eric Rescorla
RTFM, Inc.
ekr@rtfm.com

Adam Cain
Nokia, Inc.
acain@cips.nokia.com

Brian Korver
Xythos Software
briank@xythos.com

Abstract

We describe a clustered SSL accelerator. Although current SSL acceleration solutions [1, 2] often employ multiple nodes in parallel (or in series [3]) for improved performance and resistance to single failures, the failure of any node results in all client connections to that node being torn down. Our implementation goes beyond this to provide robustness against node failures at the connection level—any proper subset of the nodes in the cluster can fail and no effect (other than possibly performance degradation) will be observed. This result is accomplished by a novel combination of tight control of TCP [4] behavior and state-sharing between cluster members. Unlike many high availability clustering systems, ours uses commodity hardware.

1 Introduction

Secure Sockets Layer (SSL) [5, 6] and its successor Transport Layer Security (TLS) [7] are the dominant approaches to web security. Both protocols provide a secure channel over which ordinary web (HTTP) [8] traffic can be transmitted. HTTP over SSL (HTTPS) [9] is widely used to protect confidential information in transit between the client and server.

However, SSL is dramatically more CPU intensive than ordinary TCP communication [10, 11, 12, 13] and the addition of SSL to unsecure web servers can create unacceptable performance consequences on the web server. The dominant performance cost is the RSA operation in the SSL handshake. One common approach to reducing this cost is to offload the RSA operations to a cryptographic coprocessor which is installed on the server machine.

Another approach has been to create standalone cryptographic accelerators. These accelerators are network devices that sit between the client and server. They accept HTTPS connections, decrypt them, and make HTTP connections to the backend web server. One key advantage of standalone accelerators is that scaling can be relatively simple: more than one box can be purchased, allowing the traffic to be load-balanced across the accelerators.

In conventional configurations, having multiple standalone accelerators provides improved performance and a form of high availability. If a given accelerator fails, other accelerators may be available to handle the load. However, these configurations only offer high availability in a bulk sense: every connection that is terminated on a node that fails is lost.

This paper describes the design and implementation of a clustered SSL accelerator which we have named SSLACC. The state of each SSL connection is shared across the entire cluster. When any node fails, the remaining nodes are able to take over all connections that terminated on that node with no interruption in service. We refer to this feature as *active session failover*.

2 SSL Accelerators

Essentially, an SSL accelerator is a proxy. It accepts SSL connections on the chosen port(s) and then forwards the decrypted traffic to the corresponding port(s) on the web server. The protocol being carried on top of SSL is generally HTTP, but could actually be anything—in practice, most accelerators do not examine the plaintext before forwarding it.

Conventionally the accelerator is placed in an inline configuration between the web client and web server. Connections are accepted on the red/exterior interface and the corresponding cleartext connections are made on the black/interior interface, as shown in Figure 1. Typically the accelerator behaves like a bridge, except for the connections that it is supposed to decrypt. It can also be configured as a router in which case the topology of the server's network will need to be modified to place the accelerator in the client-server route. In either case the accelerator impersonates the client to the server and the server to the client, so the client believes it's directly connected to the server and vice-versa. It is also possible to configure an accelerator in a "one-armed" configuration in which both client and server talk to the same interface on the device.

2.1 Multiple Accelerators

In large installations, it is common practice to use multiple accelerators in parallel or series. This provides two primary benefits. First, it allows the administrator to add acceleration capacity as needed without replacing

This document was originally published in the Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, August, 2001."

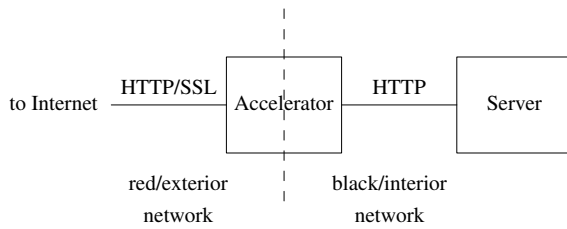


Figure 1 An inline accelerator

infrastructure components. Since HTTPS service is eminently scalable, one can simply add another box to the array and allow it to handle its share of the connections. The second benefit is reliability. Such arrays can be configured so that failures of a single unit are automatically detected and new connections are automatically allocated to the remaining nodes. However, when such unit failures occur, active connections are lost and the user sees an error.

When an error occurs in an HTTP transaction the user can simply resubmit the request. Although this is technically possible with HTTPS as well, the circumstances surrounding HTTPS transactions make HTTPS failures a more serious human factors problem. The most common use of HTTPS is for form submission of sensitive information, such as payment authorization. If an error occurs during such a transaction, the user doesn't know whether the transaction went through before the failure occurred, and thus may be unwilling to re-submit the transaction for fear of being billed twice. SSLACC addresses this problem by providing active session failover. When a node fails, all its state and connections are automatically assumed by an operational node. The transaction completes and the user is unaware that an error occurred at all.

3 AlchemyOS

Our implementation of SSL clustering was performed on top of AlchemyOS [14], a FreeBSD-based kernel specifically designed for clustering on commodity hardware. The original use of AlchemyOS was as a clustered VPN gateway. In AlchemyOS, unlike many clustering systems [15], each cluster member operates independently rather than being a clone slaved to another member, as in NCAPS [16]. Additionally, the only communication between cluster nodes is via the network—there is no shared memory bus. AlchemyOS provides support for cluster maintenance, member join/loss detection (via periodic keepalives), and clusterable TCP connections.

An AlchemyOS cluster is simply an appropriately configured set of machines on the same network. All

machines have their external interfaces on one wire and their internal interfaces on another wire. Each machine thus has a pair of IP addresses, one internal and one external. The cluster itself also has internal and external virtual IP and Ethernet MAC addresses.

Communication between the cluster members takes place via UDP packets on port 4320. These packets may be unicast, multicast, or broadcast, depending on the situation. Intra-cluster communication is protected via a *Message Authentication Code* (MAC) and optional encryption. For added security, state updates are performed only on the internal interface, which is assumed to be on the physically secure (black) network. Cluster keepalives, however, are transmitted on both interfaces in order to detect connectivity failures. Each cluster has one distinguished node called the *master*. The master's primary task is to assign workload to individual cluster members.

The SSL accelerator engine is implemented as an application on top of the AlchemyOS kernel. For minimal memory consumption, any given node runs only one instance of the application with a single thread of control, using callbacks to service I/O and cryptographic hardware. This architecture avoids having to consume stack space for each active connection.

Although AlchemyOS provides services that allow applications to cluster state, the applications are solely responsible for the contents of the clustering messages and sending them at the appropriate times. For instance, to move operation of a TCP connection to another member, the clustered application must first take a snapshot of the socket state, communicate that state to the other members of the cluster, and finally re-instantiate the state on the new cluster member. That state must be sufficient to allow the new cluster member to resume operation without service interruption.

3.1 Cluster State

It's most helpful to think in terms of two kinds of state: *working resources* and *mirrored state*. If a member is handling a given TCP connection it will necessarily have various working resources allocated to it, such as sockets, memory buffers, etc. However, since any other member must be prepared to take over for that member at any time, each of the other members must possess mirrored state—passive state sufficient to recreate the working resources if the mirror needs to take over.

A primary concern is to keep cluster updates as small as possible. Updates are transmitted over the same wire as the network traffic we're processing and therefore the simple strategy of multicasting the client data to every node in the cluster would reduce the

available network bandwidth by at least half, in addition to introducing latency. Instead we need to carefully send only the minimal amount of state to allow the other member to reproduce the original state upon failover.

3.2 Work Assignment

Each cluster member listens on the cluster IP address, thus each node sees every packet addressed to the cluster. However, AlchemyOS's load balancing scheme involves dividing the requisite packet handling among the different nodes in the cluster. AlchemyOS's IP stack automatically arranges to discard any packet which actually belongs to another member.

When a packet arrives destined for the cluster the stack automatically computes a hash function on the {source address, source port, destination address, destination port} 4-tuple. The function maps each packet into one of a small number (we use 1024) of *buckets*. If the resulting bucket is assigned to this member then the packet is handled. Otherwise it is discarded. Note that since only the address pair is used to compute the bucket, all packets corresponding to a given TCP connection fall into the same bucket.

The cluster master is responsible for ensuring that each bucket is assigned to some cluster member. This means assigning buckets when they first become active, moving buckets to rebalance cluster load and reassigning buckets owned by a dead member.

4 A Clustered TCP Relay

The goal of this project was to produce a clustered SSL accelerator providing active session failover. As often happens, new communications security features require modifications to the underlying communications stack. Currently available commodity-hardware clustering technologies such as Windows 2000 Clustering Technologies (Wolfpack) [17] and MOSIX [18] don't support active failover of TCP connections for any application protocol. Therefore, in order to build a clustered SSL relay we first had to figure out how to build a clustered TCP system. To illustrate these techniques we first present a somewhat simplified TCP relay. Such a relay is isomorphic to our SSL relay except that it simply copies the data between client and server without transforming it in any way. Once the basics of active session failover are clearly understood we can describe the real work of SSLACC, clustering SSL.

This work has a number of similar features to Fault-Tolerant TCP (FT-TCP) [19], which was devel-

oped independently at roughly the same time as SSLACC. However, there are a number of significant differences stemming from SSLACC's implementation as a relay as opposed to a server (as in FT-TCP). This difference allows us to exploit the inherent fault-tolerance of TCP to minimize clustering overhead in several ways that are not practical for FT-TCP. Moreover, FT-TCP requires that the application itself (as opposed to the TCP stack) be idempotent. When a failure occurs, FT-TCP restarts the application from scratch and replays all data from the client. Since SSL is typically used for e-commerce transactions, which of course have side effects, FT-TCP's strategy would frequently result in multiple orders and billing. Theoretically, it would be possible to make the Web server idempotent, but this would require rewriting all the back-end applications, which isn't feasible.

Additionally, SSL itself is hard to make idempotent: the server generates random numbers for each connection, so even a perfect replay of the client's data will generate different data from the server. An FT-TCP-like layer replaying the client data to the SSL server will result in the SSL implementations on client and server having different `ServerRandom` values and thus cause handshake failures. To avoid requiring idempotence, SSLACC employs application-level clustering of the SSL traffic, which also requires a generally different TCP clustering strategy from FT-TCP. In the interest of clarity, we provide a complete description of SSLACC's TCP clustering strategy here.

4.1 The Zeroth Law of Clustering

The most basic rule that a fully reliable clustered system must follow is that any peer with which it is communicating must not be able to tell if a failover occurs. Whenever a node takes over a connection from another node, it must generate traffic which could plausibly have been generated by the original handler. Thus, we have the Zeroth Law of Clustering: *all cluster nodes must generate the same data for any given connection*. Note that the Zeroth Law does *not* require that timing and TCP framing be the same, since such variability occurs during normal network behavior. During failovers it's quite common to see delays as well as shrinking windows and reframing. In fact, it's precisely such effects that lead to the Zeroth Law: a peer might receive part of a data record from one node and part from another. The record contents must be identical or decryption will fail. The requirement to obey the Zeroth Law is sometimes referred to as the *output commit problem* [20].

4.2 Server Accept

Figure 2 shows the TCP handshake and our first race condition. If a relay were to crash after sending the SYN/ACK, the backup relay would respond to the client's ACK with an RST, as shown in Figure 2. This is the behavior observed with redundant accelerators that lack active session failover.

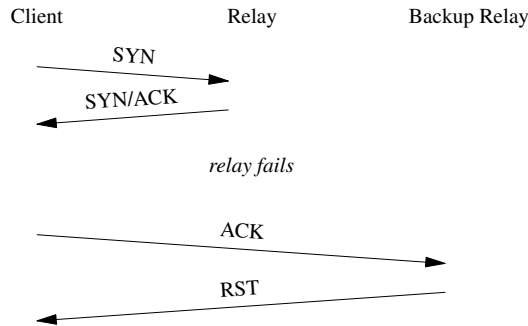


Figure 2 Failover during client connect

The naive solution to this problem is to cluster the receipt of the initial SYN, as is done in FT-TCP. However, this requires creating state on each cluster node upon receipt of any SYN, thus magnifying the effect of SYN-flood [21] denial of service attacks. Instead we use a *fingered ACK* technique: bits 3-20 of the TCP sequence number are replaced with a MAC of the address-port 4-tuple and a secret shared among the nodes. The client ACK echos back this sequence number, so when the client ACK segment comes in after a failover we can check if the fingerprint matches the ACK value in the packet and if so create the socket in ESTABLISHED state. This interaction is shown in Figure 3. Dashed lines indicate messages sent between the mirror and either client or server after a failure.

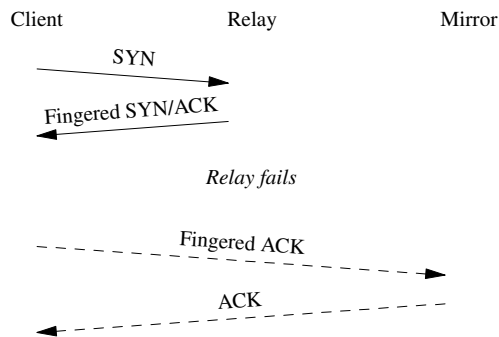


Figure 3 Failover with a fingered ACK

4.3 Client Connect

Once the connection from the client has been accepted, the relay must connect to the server. In order to minimize state sharing between cluster members, both the client and server sides of a given connection must be handled by the same relay. Specifically, if the client and server sides of a connection were handled by different relays, it would be necessary for those nodes to forward all content to each other.

If we complete the three-way handshake before checkpointing the socket, then if the relay crashes before clustering the update the connection will be left dangling on the server. Thus, we must cluster the state before transmitting the ACK to the server. The correct behavior is shown in Figure 4.

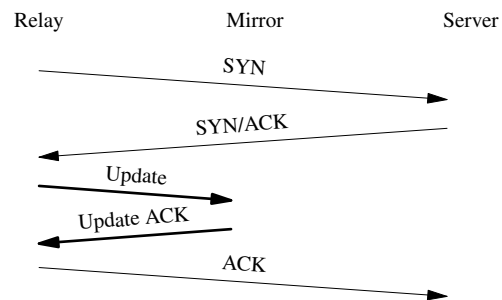


Figure 4 Clustered connect

There are two interesting locations where crashes might occur. (1) the crash occurs before the cluster update is received (2) the crash occurs after the cluster update has been received. Note that the case where it has been transmitted but not received is the same as the one where it hasn't been transmitted yet.

(1) In the case where the crash occurs before the update is received, the mirror will have no knowledge of the socket when it comes online. If it tries to initiate a new connection with the server the server will respond with an RST because the TCP *initial sequence number* (ISN) will be different. In order to fix this problem the mirror must use the same ISN as the relay did. To arrange this we use the same ISN as the client used. Thus, when the first packet of data from the client arrives with its fingered ACK, the `accept()` cycle will start over again cleanly as described above. (Note that we can derive the ISN from the sequence number of the first packet from the client. TCP slow start guarantees us that the window will only allow one outstanding packet at this point.) Figure 5 shows the sequence of events.

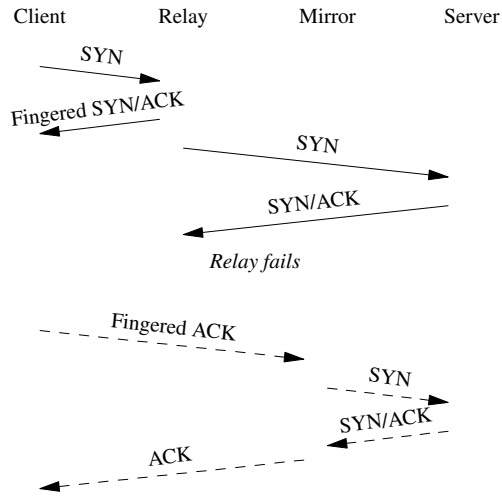


Figure 5 Server connect with fingered ACK

(2) If the crash occurs after the update is received then the mirror will come online with the appropriate mirrored state. It will resurrect the sockets connected to the client and the server. When the server retransmits its SYN/ACK the relay will transmit the ACK immediately. Since the state update has already occurred there is no need to do it again before transmitting the ACK to the server.

4.4 The First Law of Clustering

This example illustrates the first law of clustering: *Cluster then commit*. The relay must always cluster a state before it commits to it by sending network traffic. In the case of the relay connecting to the server, the relay needs to cluster the new ACKed state before committing to it by transmitting the ACK to the server. Consider what happens if we transmit the ACK first, as shown in Figure 6. If a failure then occurs before the new state is clustered the node which takes over will not know about the new socket. Any attempt by the server to transmit data on the newly established connection will produce an RST by the mirror.

4.5 Port number selection

Most TCP stacks select port numbers for active opens using a counter. This will not do for our environment. Because bucket assignment depends on the port number, this technique would often result in the relay-server connection falling into a different bucket from the client-relay connection. This would require us to

somehow handle client→server transmission on a separate node from server→client transmission, which would be extraordinarily difficult because it would require splitting the TCP stack. Instead, we arrange for the relay-server connection to use a port carefully chosen to ensure that both connections fall into the same bucket.

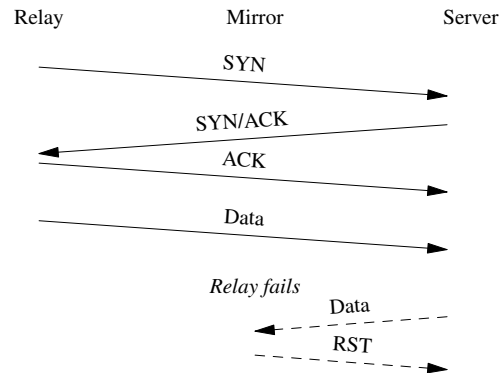


Figure 6 Failure when committing before clustering

4.6 ACK Handling

Note that withholding the ACK until the cluster update has been received requires modifications to the TCP stack. Ordinarily the ACK pointer is incremented as soon as data is received. In order to suppress ACKs we need to separate the ACK pointer from the `rcv_nxt` value (representing both the sequence number of next byte to be read and the ACK pointer). We add a new element in the protocol control block (PCB), `rcv_appack`. This value can be controlled by an API call. However, simply making this modification alone creates pathological behavior. When an ordinary TCP stack receives an in-order packet it generates an ACK (possibly delayed up to 200 ms by the delayed ACK timer). However, if we haven't incremented the ACK pointer value then generating an ACK may cause the sender to go into congestion control mode (upon receipt of 3 duplicate ACKs). In order to avoid this we need to not just tinker with the ACK pointer but actually suppress naked ACKs until the application allows them. This strategy resembles a combination of FT-TCP's Lazy and Eager ACK strategies.

Handling ACKs from the peer can also be tricky. Imagine that a relay transmits a given data segment and then crashes. The peer will transmit an ACK for that segment. If the failover happens at the wrong time the mirror will receive the ACK. There are two potential problems here. First, the ACK might arrive before the

socket is instantiated on the relay. Under normal circumstances the mirror would generate an RST when receiving an ACK for an unknown connection but this would terminate the connection. Therefore, this sort of RST needs to be suppressed for *twice the maximum segment lifetime* (2MSL) after a failover (or until all the relevant sockets have been restored).

The second problem is the receipt of a *forward ACK*—an ACK which is in advance of the `snd_nxt` value (containing the sequence number of the next byte to be written) on the restored mirror. Ordinarily, such ACKs are protocol errors and are ignored but that can create problems here. This can happen if a relay transmits data and then crashes. Consider the case where the ACK is for two segments. The restored mirror’s window may only allow one segment to be transmitted. In that case the system will deadlock. The mirror will retransmit the segment and then ignore the forward ACK sent by the peer. The correct behavior in this situation is to discard application writes until `snd_nxt` equals the new ACK value sent by the peer.

4.7 Contents of State Updates

AlchemyOS provides a `TCP_MOVABLE_STATE` argument to `getsockopt()` which permits the programmer to get a pickled version of the current TCP connection state. The resulting state can be transmitted across the wire to another node where `setsockopt()` is used to impose it on another socket. This procedure is all that is required to move a quiescent socket (one where no data or ACKs are outstanding) from node to node. However because of the First Law, we will often want to checkpoint sockets in non-quiescent states—indeed the state we’re about to enter. In such cases the `getsockopt()` will return the old state, which then must be modified to contain the new state.

In various cases (which we’ll see later in this paper) the application needs to manually control all four sequence numbers. Thus, we can specify any given update (containing both client-side and server-side connection states) by the 4-tuple $(client_rcv_nxt, client_snd_nxt, server_rcv_nxt, server_snd_nxt)$. We label the initial values for each member of the tuple—and hence the initial sequence numbers as: $(ISN_{cr}, ISN_{rc}, ISN_{sr}, ISN_{rs})$. As a notational convenience, we refer to the state at the beginning of any given transaction as $(S_{cr}, S_{rc}, S_{sr}, S_{rs})$ or simply S .

4.8 Data transmission

Once the connections have been established and clustered, we’re ready to transfer data. Since the TCP relay is inherently symmetrical we only need to cover the

case where data is being written in one direction. Without loss of generality we’ll illustrate client to server data transmission.

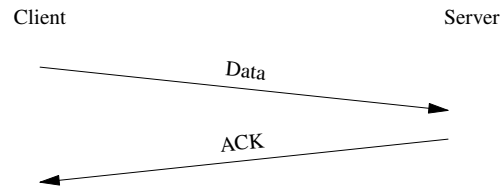


Figure 7 Normal client to server write

We can cluster this communication with a single cluster checkpoint after the server ACKs the data. Once the update has been ACKed the relay sends the ACK to the client, as shown in Figure 8. If the data is d bytes long the new state will be $S + (d, 0, 0, d)$. Knowing when to send the update requires the ability to determine when data has been ACKed by the peer. AlchemyOS provides a callback for this purpose. Note that this cluster checkpoint contains only sequence numbers, not data. Because we are implementing a relay, once the data has been acknowledged by the server we can simply discard it. The mirror never sees the application protocol data. This differs from FT-TCP where all the data transmitted by the client must be transmitted over the network and retained by the external data store.

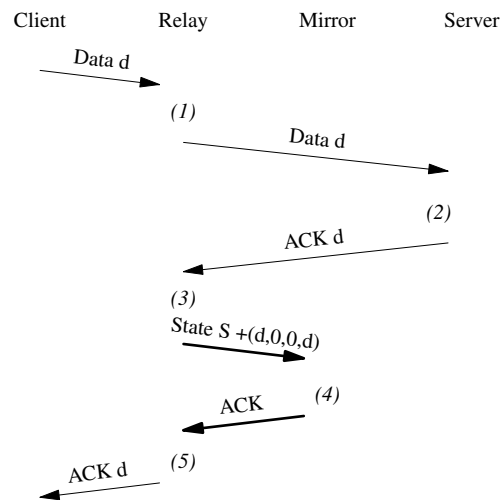


Figure 8 Clustered client to server write

Let’s examine what happens if the relay fails at various points in this interaction. There are five points where the relay may fail (numbered in italics in Figure 8).

1. Right after the relay has received the client data
2. Right after the server has received the client data but before the server has sent the ACK
3. Right after the relay has received the server ACK
4. Right after the relay has sent a cluster update to the mirrors but before this update has been ACKed
5. Right after the relay has received the ACK for the cluster update but before the relay has sent the ACK to the client

We'll take each of these cases in sequence:

(1) This case is indistinguishable from the situation in which the packet was lost on the wire. When the mirror comes online it is still in state S. The client retransmits the data and the mirror handles it as if it were being sent for the first time, as shown in Figure 9.

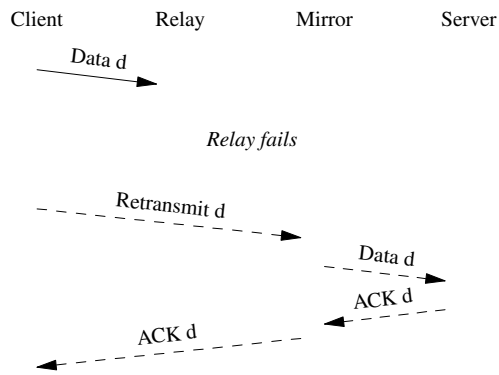


Figure 9 Failure after first data write

(2,3) From the client's perspective, cases (2) and (3) behave identically. We show case (2) in Figure 10. Since we still haven't hit the cluster checkpoint the client-relay interaction is the same as before; the client retransmits and the relay writes the data to the server. To the server, this merely looks like a retransmit, which might happen under normal circumstances if the server's ACK were lost. Case (3) is identical except that there is an ACK from the server.

(4,5) Finally, consider what happens if the relay fails after clustering the state. This can happen before the relay receives the update ACK (4) or after (5). The effect is the same. The simplest possibility, as shown in Figure 11 is that the client retransmits the data. However, this time the mirror's TCP state already has its ACK pointer at $S_{cr} + d$ and so it drops the client data on the floor and sends an immediate ACK for $S_{cr} + d$ bytes. Although this interaction works fine, it forces us to wait for the 500 ms TCP retransmit timer. In order to reduce latency a better approach is for the mirror to send a single ACK when it takes over the connection.

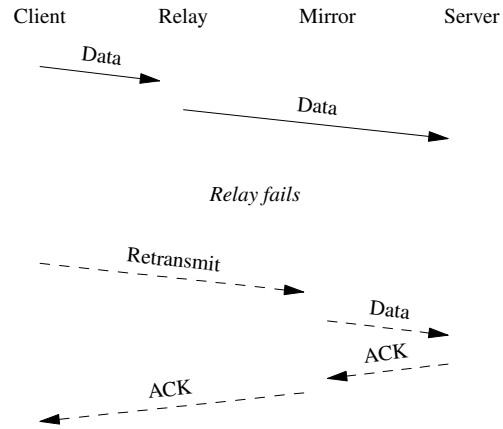


Figure 10 Failure after first write to server

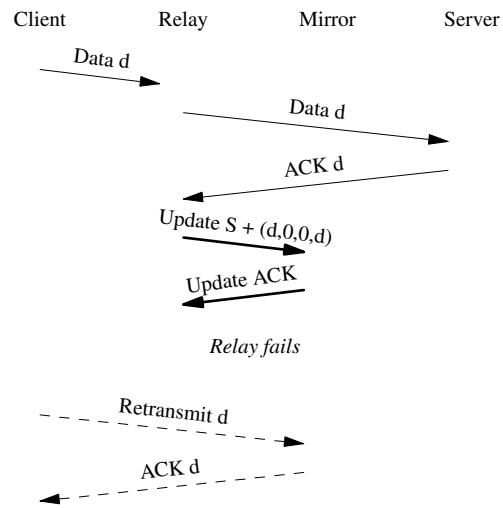


Figure 11 Failure after cluster update

There is a common error that people make—indeed that we made—when they attempt to cluster this interaction. In an effort to reduce the latency introduced by the cluster update, it initially seems reasonable to interleave the cluster update and the data write, as shown in Figure 12.

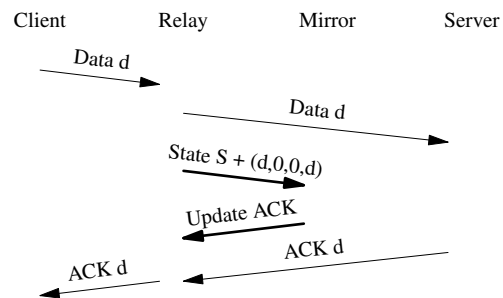


Figure 12 An incorrectly clustered client to server write

The problem with this interaction becomes apparent if you consider the possibility that the relay's transmission to the server gets dropped for some reason. If the relay fails after the cluster update has succeeded then the relay will have state $S + (d, 0, 0, d)$ but the server will have state S . At the best this will create deadlock and at the worst an ACK storm when the server sends duplicate ACKs to correct the relay's sequence number.

There are a number of subtle points worth considering before we leave the issue of data transmission. (1) TCP ACKs from the server are not 1 to 1 with reads from the client. Thus, it's necessary to keep track of the amount of data being ACKed, not just the fact that an ACK occurred, in order to know how much data to ACK to the client. (2) It's tempting to treat each ACK as if it is relative to the current ACK pointer state and therefore cluster the current ACK pointer plus the ACKed data. However, since the ACK pointer in the TCP state can only be incremented after an update is ACKed there is a race condition here as well. Consider the sequence of events shown in Figure 13 and described below.

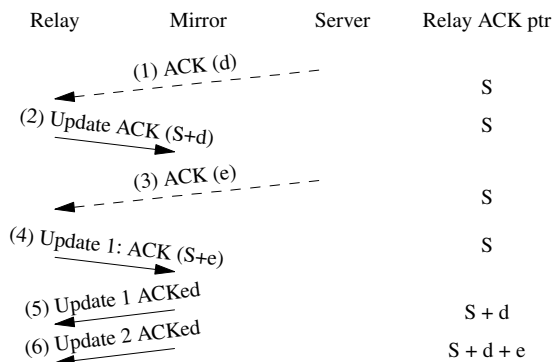


Figure 13 An ACK race condition

1. Receive Data ACK 1 from server for d bytes.
2. Cluster update 1 for $S + d$ bytes
3. Receive Data ACK 2 from server for e bytes.
4. Cluster update 2 for $S + e$ bytes
5. Receive ACK for update 1. Increment ACK pointer by d to $S + d$.
6. Receive ACK for update 2. Increment ACK pointer by e to $S + d + e$.

Once the updates are ACKed the actual ACK pointer will have been incremented by $d + e$ bytes but the mirrors will have incremented it only by e bytes, because the second cluster update was sent without taking into account the first write of d bytes. This happened because the first update had not yet been ACKed before

the second update was transmitted. If the ACK for Update 1 were received before Data ACK 1, then the result would be correct. In order to avoid this race condition it is best to work in absolute sequence numbers and maintain the value of the last *clustered* sequence number as well as the *current* sequence number. Thus, in step 4 we would cluster the update for $S + d + e$ bytes.

Another subtle aspect of clustering application data pertains to partial ACKs from the server. Consider what happens if the client transmits x bytes and the server ACKs $y < x$ bytes. When the relay clusters the ACK, $rcv_nxt_cr > rcv_appack_cr$ with the difference being $x - y$. If a failover occurs after this update, the client will retransmit starting from rcv_appack_cr . Since we do not cluster buffered data, the mirror needs to be able to read this transmission. Accordingly, when we perform cluster updates we set $rcv_nxt = rcv_appack$ in the update, allowing the mirror to process the retransmitted data.

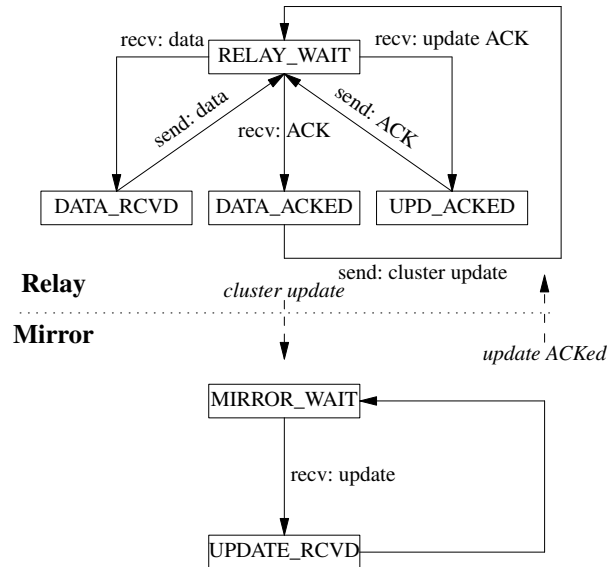


Figure 14 Data transmission state machine

Although we have presented the sequence of events in a linear fashion, it's important to remember that in practice, data transmission, cluster updates and ACK handling happen in parallel, and the relay needs to be able to handle all three kinds of events at once. Figure 14 shows the state machines for the active relay and the mirror. The top half of the figure shows the relay state machine and the bottom half the mirror state machine. Solid lines indicate state transitions and their associated messages and italics indicate messages between relay and mirror. The basic state transition path for the active

relay is from `RELAY_WAIT` through `DATA_RCVD` when the relay reads data and sends it to the peer; then `DATA_ACKED` when that transmission is ACKed and the ACK is clustered; then through `UPDATE_ACKED` when the update itself is ACKed and the relay finally ACKs the original data transmission to the client.

Each node only has one state in which it waits for network data (`RELAY_WAIT` on the relay and `MIRROR_WAIT` on the mirror). In every other state, the node simply transmits data and then returns to the read state. This allows the relay to process events of one type while waiting for events of another type, thus creating implicit wait states that are not shown in Figure 14. For instance, the relay might simultaneously be waiting for an ACK for one data packet and an ACK for a cluster update. This parallelism allows TCP to operate correctly in the face of clustering by continuing to allow data to flow even when previous segments are un-ACKed. Note, however, that the requirement to cluster update data ACKs before they are propagated to the sender gives rise to a new form of deadlock: if a mirror stops responding to cluster updates, the sender will eventually fill the entire TCP window and stop transmitting. This fact makes ACKing cluster updates a critical operating system function. A node which fails entirely does not bring down the cluster but one which appears to be functioning but does not ACK updates will deadlock every connection in the cluster.

4.9 The Second Law of Clustering

We now have our basic technique for reducing cluster update size. Rather than cluster the data itself we force the client to buffer it for us by withholding the ACK until the data has been acknowledged by the server. Failovers therefore result in TCP retransmits. In essence, device failures look like intermittent network lossage of the kind that TCP is already designed to be robust in the face of. Thus, it's safe to transmit the data to the server without clustering it, since the client will retransmit in the case of failure. This illustrates the Second Law of Clustering: *It's safe to transmit unclustered data as long as you can reproduce it.*

4.10 Closure

Consider the sequence of events required to close a connection. For reference, we show the ordinary TCP close sequence in Figure 15. Note that although TCP allows a half-close where one side sends the FIN and the other side transmits data, SSL forbids half-close and so we

decided to omit it from our TCP clustering implementation.

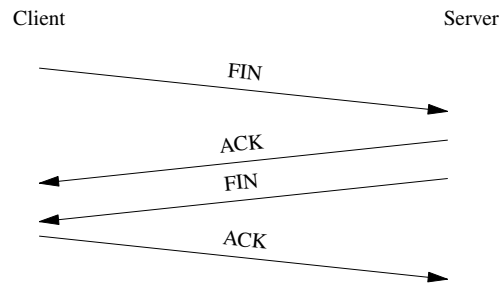


Figure 15 TCP close

The closure sequence has to accomplish two things:

1. Destroy the relay state on the mirrors
2. Close the connections on the relay

The simplest sequence that will accomplish this result is shown in Figure 16 (with the client and server FINs omitted for clarity). The relay sends an update indicating that it's about to close. Once that update has been ACKed the relay calls `shutdown()` on both sockets. Once the FINs have been ACKed the relay calls `close()`. This approach works reasonably well but is not ideal. One small problem is that if the relay handling the connection leaves the cluster or the bucket is reassigned before a peer FIN is received, the socket can get stuck in `FIN_WAIT_2` until the `FIN_WAIT_2` timer expires. To prevent this we turn on TCP keepalives with a timer of 1 minute before we call `close()`. When the keepalive timer fires and no response is received the socket is automatically discarded.

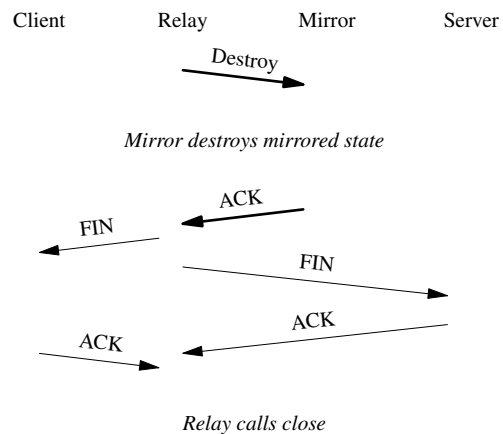


Figure 16 A clustered close

Another problem is that a failover after `shutdown()` has been called but before the TCP closure handshake has been completed results in RSTs when a peer sends its ACKs; since the mirrored state has been removed from the mirror, the mirror has no knowledge of the connection and simply sends an RST. Fixing this problem requires more cluster updates to indicate the initiation of the close and to indicate the success of the close. We judged that the performance consequences were not worth the modest improvement in TCP friendliness for what we considered an unlikely condition.

5 A Clustered SSL Relay

Clustering SSL encompasses roughly the same set of tasks as clustering TCP but is complicated by a number of factors:

- We first need to perform the SSL handshake. The handshake involves interaction with the client only but it all needs to be clustered.
- SSL data is structured in a record format whereas the data we were pushing over TCP is essentially freeform. This creates difficulty both sending and receiving.
- We need to cluster cryptographic keying material both on a global and a per-connection basis.
- The SSL session cache must be shared across the entire cluster.
- SSL has its own closure sequence on top of TCP.

5.1 Clustering the SSL Handshake

We assume that the reader is familiar with the SSL handshake. For reference, Figure 17 shows the basic SSL handshake, using static RSA.

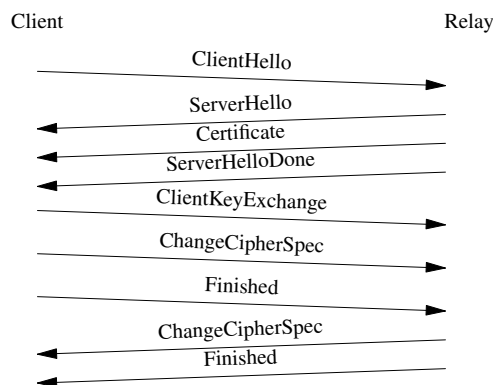


Figure 17 The SSL handshake

As Figure 17 shows, the first message we process is the ClientHello. The most obvious approach is to cluster the contents of the ClientHello and then ACK it to the client. This has two drawbacks: (1) Efficiency. We don't need to cluster the entire ClientHello, which contains all the ciphersuites that the client is willing to speak. We only need to cluster the ciphersuite that we've in fact chosen. (2) We'd have to introduce a second checkpoint for the ServerHello: the ServerHello contains a random value generated by the server. If the relay fails after sending the ServerHello then the mirror must be able to reproduce the ServerHello, including generating the same random numbers. One way to deal with this would be to synchronize the random number generators on all nodes but this is a difficult task. A better approach is to cluster a "pre-ServerHello" state. This state contains:

- client and server random values
- chosen cipher suite

Additionally, every handshake update contains

- the new TCP state
- the current value of the SSL handshake hashes
- the handshake state to enter upon failover. For the case where we have just read the ClientHello, this is "send ServerHello"

If a failover occurs when the mirror has received the pre-ServerHello update, it will generate a new ServerHello using the clustered random value and containing an ACK for the ClientHello.

In order to reduce the latency inherent in this operation we transmit the state update before we generate the messages (thus parallelizing the clustering latency and message generation). However, we can't actually transmit the messages until the update is ACKed because this would violate the First Law—we need to make sure that the random data in the ServerHello is clustered before we commit to it by transmitting the ServerHello to the client. We simply queue the messages and empty the queue when we receive the ACK. This isn't that important an optimization under normal circumstances since generating the messages is much faster than the cluster update round trip time so the RTT dominates the interaction. However, if we're using ephemeral RSA (which is only used in the increasingly uncommon export case) the time to generate the ServerKeyExchange is significant and therefore we get substantial parallelization.

Figure 18 shows the interaction in the case of static RSA. Because of TCP delayed ACKs the ACK of the ClientHello will typically appear on the first data segment (and of course on every subsequent data segment) rather than bare on the wire. Note that the number of

TCP segments used to transmit the records depends on the amount of buffering and whether the Nagle [22] algorithm is on, but this is irrelevant to the clustering logic.

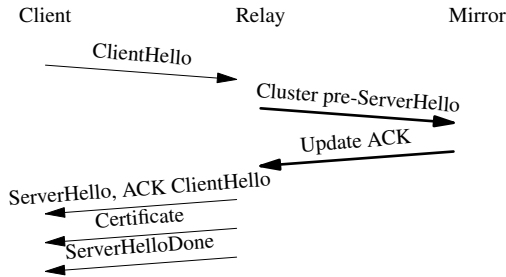


Figure 18 Clustering the ClientHello

Once the client receives the ServerHelloDone it sends the ClientKeyExchange followed by the ChangeCipherSpec and Finished. Optimally we'd like to wait for the Finished and ACK all three messages together. Unfortunately this creates problems if the client writes each message to the network separately. In that case the write of the ChangeCipherSpec will be blocked by the Nagle Algorithm until the ClientKeyExchange is ACKed. If all three messages fit into one TCP segment then the next two messages will be sent when the retransmit timer fires in 500 ms. If the messages don't fit, the transaction may be deadlocked while the client sits in congestion control mode waiting for the server's ACK.

This forces us into an approach where we individually ACK each message. The First Law requires that we first cluster them. Thus, the logic becomes that shown in Figure 19.

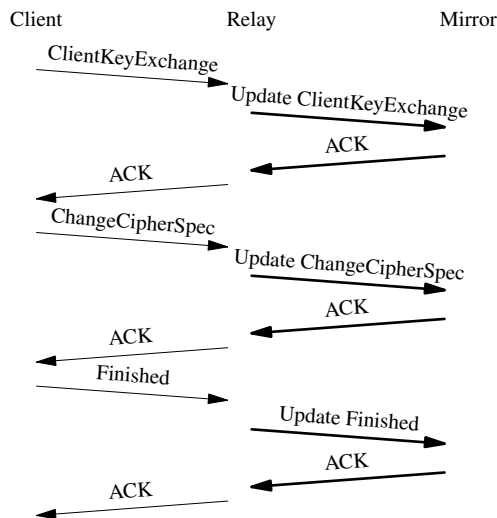


Figure 19 Clustered ClientKeyExchange, etc.

The first update contains the encrypted PreMaster secret. This reduces latency by allowing the relay to decrypt the PreMaster secret while waiting for the cluster update to complete. The second update contains the master secret and the pending cipher states in both directions. The third update contains the new read cipher state after having read the Finished message.

This logic creates quite a bit of cluster traffic and it's fairly likely that the client will send all three messages in one segment. One could detect the case where all three messages are present and if so issue one cluster update instead of three but we have not benchmarked this optimization.

Finally, the relay sends its own ChangeCipherSpec and Finished messages. When the server's Finished message is ACKed the handshake is over and the relay clusters the entry into the data state. This message also contains the new write cipher state after having written the Finished. Once that update is ACKed the relay enters the data state, as shown in Figure 20.

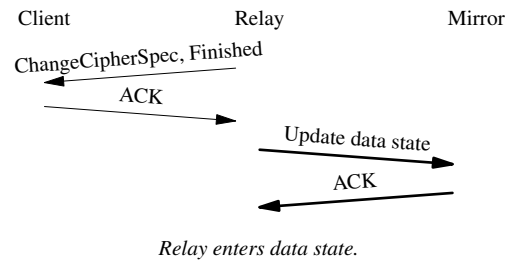


Figure 20 Clustered Finished

5.2 Resumed Handshakes

SSL includes a *session resumption* feature that allows the client and server to skip the RSA key exchange step. This is less important in the face of RSA acceleration on the server side but is still of considerable value. However, with a clustered system there is no guarantee that the same relay will handle the client when it reconnects again. Thus, we need to cluster the session cache.

Clustering the session cache is actually quite simple. When a mirror receives the cluster update indicating the end of the handshake it inserts the session information (derived from the cluster updates received thus far) into the local session cache. As is common practice, this is implemented as a hash table. When a client requests resumption, the relay handling that connection (including one which was a mirror for the original connection) can just consult its own hash table.

Clustering the session resumption handshake follows essentially the same pattern as the ordinary handshake. The most notable difference is that the steps for clustering the transmission and receipt of Finished are switched, as shown in Figure 21. Note that the last two cluster updates could be collapsed into one but this is not done in the current implementation.

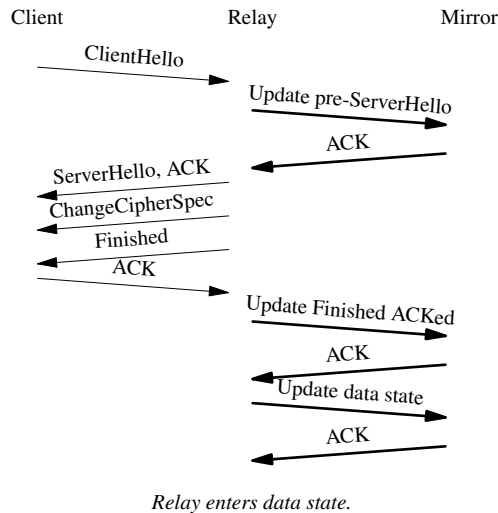


Figure 21 Clustering the resumed handshake

5.3 Connect to Server

After the client connects, two processes occur in parallel: the SSL handshake and the connect to the server. Either process may finish before the other. Only once both are completed do we enter the data state and prepare to read data from the client. However, because we need to checkpoint the SSL handshake state, some checkpoints may occur before the server connection has completed. This possibility, that there might be cluster checkpoints where only the client socket is valid, did not exist in the simple TCP case we discussed earlier.

Luckily, dealing with this case is relatively easy. When a failover occurs in such a state the relay simply restarts the appropriate connect call. As in section 4.3, we need to make our TCP ISN deterministic to avoid getting RSTs during the connect.

Unfortunately, since the SSL handshake is happening in parallel with the connect to the server we can no longer determine the client's choice of ISN from client retransmissions. The first message we receive after failover might not be the ClientHello, in which case it would have a sequence number that did not match the client's ISN. Thus we need to include the client's ISN in the first cluster checkpoint. This requirement means

that we cannot begin to connect to the server until after the first cluster checkpoint of the SSL handshake. This ensures that the mirror will have the client's ISN if it needs to reissue the `connect()` call. The relationship of the SSL handshake to the connect handshake is shown in Figure 22.

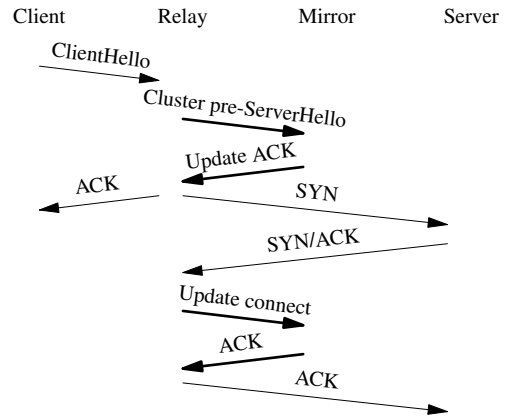


Figure 22 Parallel client and server handshakes

5.4 Cipher States

In order to encrypt or decrypt an SSL record the following pieces of state are required:

- the sequence number
- the encryption state
- the MAC key

Of these, only the encryption state and the sequence number vary during an SSL connection. Clustering the sequence number is straightforward but the encryption state is less obvious: when the cipher is DES, 3DES, or AES, we need to cluster only the key and the current CBC residue. With RC4 we have two options: cluster the current key schedule or cluster the key and an offset into the stream. The second option is more compact but can be excessively slow if failover occurs during a long transfer, requiring the mirror to generate and discard megabytes of keystream.

The compromise option used by SSLACC is to cluster the *base state* (the entire key schedule) every megabyte or so and in between transmit *deltas*. This actually presents an interesting implementation issue. The natural approach would be that the deltas each contain the number of bytes processed since the last base update. However, when a number of records are ACKed at once we only cluster the final record to reduce bandwidth consumption. This presents the possibility that one of the updates we skip might be a base update. In

this case we would advance the stream from the wrong base update and thus be encrypting using too early a section of keystream. To avoid this problem the deltas actually contain the offset from the beginning of the keystream. Thus, when we attempt to reconstitute the keystream we start from the last base update we receive (which also includes its position in the keystream) and then advance the keystream to the point indicated by the delta.

5.5 Client to Server Data Transmission

Clustering SSL client to server data transmission is analogous to clustering TCP communication but rather more complicated. The message diagram, shown in Figure 23, is essentially the same. The additional complication is introduced by the need to cluster the cipher state and the need to handle one record at a time. With TCP we could simply increment our own notion of how many bytes to ACK each time we saw an ACK. With SSL we can only generate updates whenever a full record is ACKed because only at that point can the connection state be simply summarized. Decrypted records cannot be transmitted until their MACs have been checked, which can only be done one record at a time. Without this restriction, an active attacker could inject a forged partial record which SSLACC would forward without performing an integrity check. Thus, checkpointing at any other location than a record boundary would require clustering the plaintext as well as the cipher state. To make things more complicated, the size of the SSL record is generally not the same as the size of the plaintext data. The addition of the headers, padding, and MAC increases the size of the plaintext data. Compression would reduce the size of the plaintext data but is generally unimplemented. As a result, the sizes would only match by accident and the number of bytes to ACK to the client will be different (generally greater) than the number of bytes that were ACKed by the server.

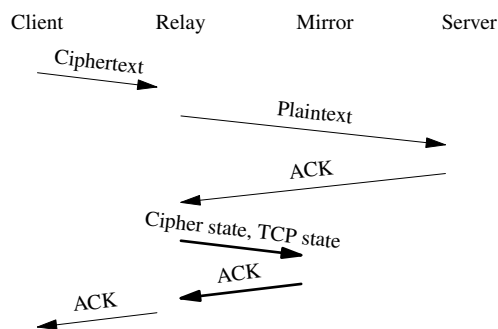


Figure 23 Clustering client to server writes

In order to map server ACKs to record state, SSLACC maintains a queue of the length and states for all records for which the plaintext has been written to the server but not yet ACKed. Whenever more data is ACKed by the server we move the ACK pointer forward in this queue the appropriate number of bytes. When a full record has been ACKed we remove it from the queue, cluster the new state, and ACK the client's data. Because multiple records might be decrypted before any of them are ACKed, each record in the queue has its associated cipher state attached to it at the time it is decrypted. Thus, when a record is ACKed the cipher state we cluster is the one attached to the record. If multiple records are ACKed by a given server ACK we simply cluster the state associated with the last one.

If a failover occurs, the mirror simply installs its mirrored TCP state and the appropriate cryptographic state and picks up from there.

Partial ACKs

SSL records can be up to 32K bytes long (the standard fixes them at 16K but some implementations violate this rule). This introduces a problem since it is quite possible that a record will be too large to fit in a single TCP segment. If the record is especially large or the connection is still in slow start, it's quite likely that the effective window will be too small to carry the entire record. This produces the possibility of deadlock. The relay cannot ACK any data until the entire record has been read but the client cannot transmit any more of the record until it receives an ACK.

The only way out is for the relay to ACK the data read so far. In order to do so it must first cluster that data. This is one of the few cases in which we actually cluster data rather than state, and since it's expensive to do, we do so only when necessary. SSLACC maintains an estimator of the packet interarrival time (IAT). When a partial record is read, we set a timer for $2 \cdot \text{IAT}$. If that timer fires before the rest of the record is read, we cluster the partially read record and then ACK that section of data. If a failover occurs during this period the mirror simply picks up with reading the rest of the record. Figure 24 shows a partial ACK.

In practice, partial ACKs occur infrequently. First, most HTTPS transactions involve a small client write (the HTTP request) followed by a lot of data from the server (the HTTP response). Thus, the records usually fit in the effective window. In the case where a lot of data is being written by the client the window will quickly open up to be larger than a record (although not larger than a pathological 32K record). Finally, we only allow partial ACKs to occur when there is no unACKed data written to the server, thus simplifying the logic

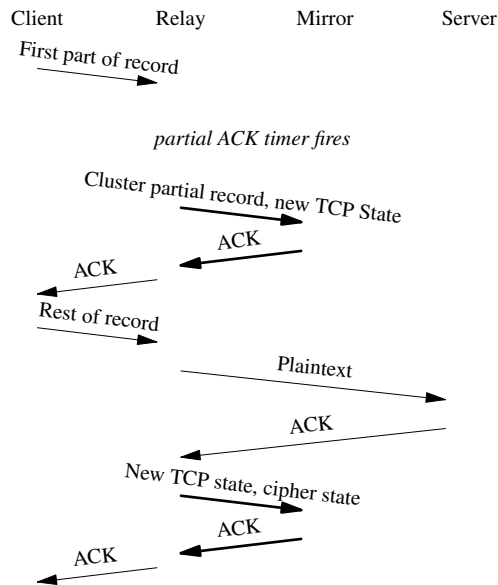


Figure 24 Partial ACK

considerably. The expectation is that when we ACK the previous record the client will transmit the rest of the record we're currently reading.

Unfortunately, this expectation is not always fulfilled. If the server is slow enough the client will be driven into congestion control mode. The congestion window will be one segment and thus we will never receive the rest of our record. Accordingly, whenever we ACK to the client *and* we have a partially read record we set a timer for the round trip time (RTT) + IAT. This allows the ACK to arrive at the client and the client to send the next segment if it is going to. If it doesn't and the timer fires we once again perform a partial ACK.

5.6 Server to Client Writes

Naively, one might think that we could use the same strategy for server to client writes. Unfortunately, this is not the case. The problem is that TCP has no concept of record boundary. In order to provide acceptable interactive performance we must be prepared to write data to the client as soon as we receive it, but this makes the sizes of the SSL records somewhat arbitrary. They are determined by some combination of the maximum read size, buffering TCP window, and relay loading. After failover the mirror is likely to read a different size chunk from the server than the original relay did. If nothing is done to ensure that the same record sizes are used, the stream of records won't match the original and MAC errors will occur.

Figure 25 shows a simple example of what can go wrong. The relay reads 4096 bytes from the server. This represents approximately 4 segments on an Ethernet. The relay packages this up in a single record (R1) and then sends it to the client. At this point the relay fails. When the mirror comes online only 1024 bytes are available from the server and so it transmits a 1024-byte record (R1', which has the same sequence number as R1). Its next read is 3072 bytes so it transmits a 3072-byte record (R2).

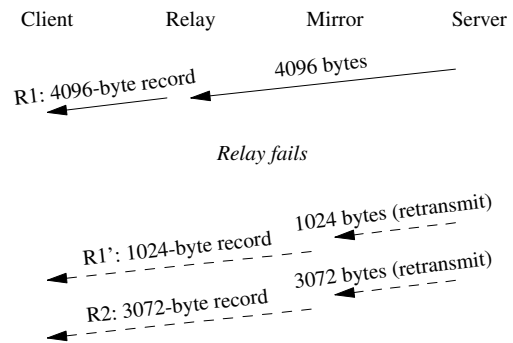


Figure 25 Record size problem

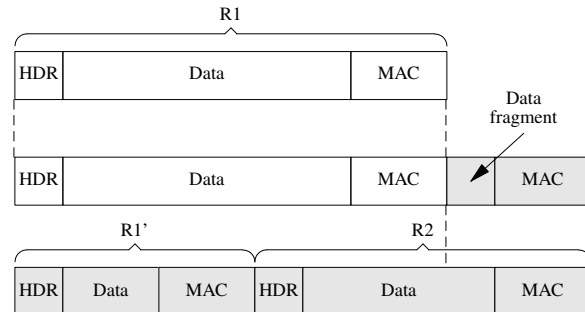


Figure 26 Record size misalignment

Figure 26 shows what happens when the client tries to read the records. On top we see the data stream as written by the original relay. At bottom we see the data stream as written by the mirror which takes over. In the middle is the inconsistent data stream as seen when the client gets part of its data from the relay and part from the mirror. In this case, we've assumed that the client has read the entire first record. Thus, it will attempt to start reading a new record but will actually start reading in the middle of the second record from the mirror (represented by the shaded section). This will create errors. Note that this particular error occurs because of the data expansion introduced by SSL record framing. Even though the same amount of plaintext data is transmitted

by both relay and mirror, the addition of headers, MACs and padding causes the cumulative size of the SSL traffic to be larger when transmitted as two records than as one. It is this size difference that causes the misalignment seen in Figure 26. Depending on the exact timing of events, a number of other merges of the data streams are possible but almost none of them are correct.

In order to avoid this situation we need to cluster the record *size* before we transmit the record. The mirror maintains a queue of the sizes of records which have been written but not ACKed. Thus, whenever it receives a *pre-write*—the size of a record about to be written—it adds it to the list. Whenever it receives an update that a record was ACKed it removes that record from the list. Upon failover the mirror uses this list to determine what size records to read. The sequence of events is shown in Figure 27. Note that because this technique commits us to reading certain record sizes after a restore, the relay can get blocked reading from the server in the same way as we saw in section 5.5. We use the same partial ACK technique to remove such deadlocks.

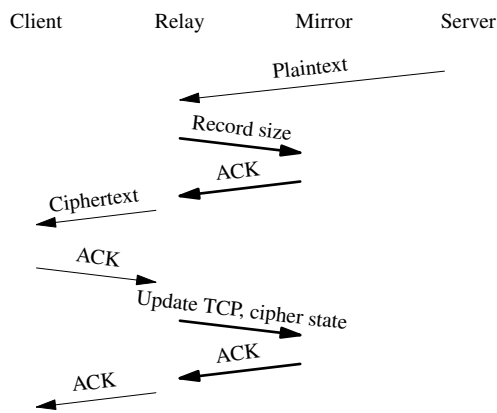


Figure 27 Pre-writing records

5.7 Closure

There are three conditions which might trigger closing the connection:

- a close from the client
- a close from the server
- an error

Rather than attempt to note each of these conditions and cluster them we adopt the simple expedient of withholding ACKs for the messages which generated them. After a failover, we expect the retransmits to generate the same condition on the mirror. However, if we are closing because of an error we do cluster that the

session is not to be resumed. When the mirror receives that update it removes the entry from the session cache.

However, all three of these conditions are likely to require us to transmit an SSL alert. We first disarm the read callbacks so that no further attempts will be made to read data. We then transmit the alert. Once the alert is ACKed we are ready to proceed with shutting down the socket, which we do in the same fashion as we described for TCP clustering. If an error occurred, we also cluster that the SSL session is not to be resumed, as required by the SSL specification. At this point the relay will ACK all received data along with its FIN.

5.8 Performance

Our primary focus with SSLACC has been on protocol and implementation correctness. However, we have done some simple performance tuning. We have benchmarked SSLACC in two configurations: "SSLACC 200" with a 200 RSA/sec coprocessor and "SSLACC 600" with a 600 RSA/sec coprocessor (both using Rainbow Cryptoswift cards). Both configurations are 846/MHz Pentium IIIs with 512 MB of RAM. Figure 28 shows the performance of SSLACC 200 clusters of sizes between one and seven (the number of machines we had available) under a pure handshake load: SSL handshake followed by a trivial HTTP fetch.

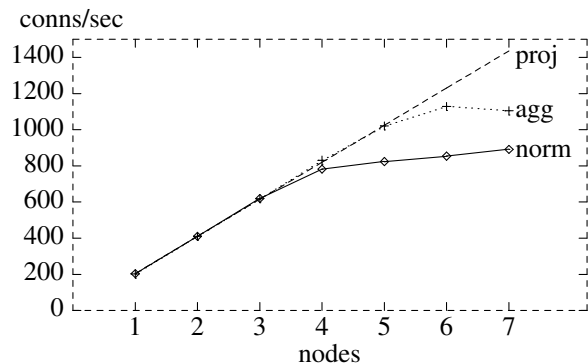


Figure 28 SSLACC 200 performance

The "norm" line represents the observed performance of a SSLACC 200 cluster. The "proj" line represents the theoretical performance of an unclustered system. Performance scales linearly up to about 3-4 nodes, at which point it starts to plateau off. The problem, at least in part, is that the nodes are being swamped by clustering overhead. Essentially every handshake message processed requires a checkpoint, so the number of messages scales with the number of nodes. Thus, even though we add more processing power, an increasing

amount of CPU time is spent handling messages from other nodes. As a result, the addition of a node doesn't increase the overall cluster capacity as much as desired. This provides strong motivation for reducing the amount of inter-cluster traffic. Some simple profiling determined that much of this overhead was spent in interrupt handlers for cluster message arrival. Aggregating multiple message sends into a single larger message reduces this overhead and gives us improved scalability, as shown by the "agg" line in Figure 28.

SSLACC 600 performance shows a similar pattern, but with the overload appearing far earlier, as shown in Figure 29. Note that the "norm" line shows plateaus at almost exactly the same place for SSLACC 600, as for SSLACC 200, indicating (as expected) that the bottleneck is the CPUs, not the acceleration hardware.

Other than the addition of aggregation, no significant performance tuning has been attempted on SSLACC, so SSLACC's performance behavior, especially for large clusters, is poorly understood. In particular, the causes of the drop at 7 nodes for SSLACC 200 and the plateau at 3-5 and jump at 6 units for SSLACC 600 (both with aggregation turned on) are unknown. It's also worth noting that the performance of SSLACC 600 at 6 and 7 nodes is superior to that of SSLACC 200, suggesting that even though the acceleration hardware is operating at less than rated capacity in both cases, the difference between the 200 and 600 cards is nevertheless affecting performance. Further performance tuning of SSLACC handshaking is a topic for future research.

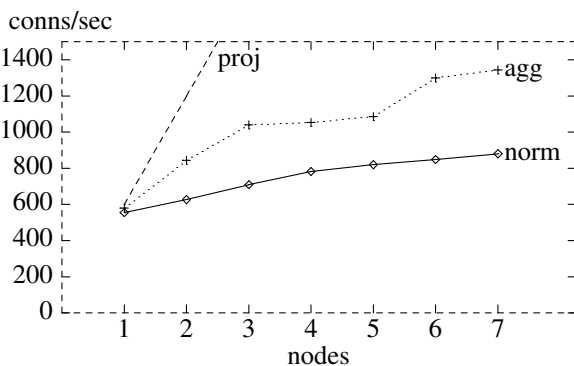


Figure 29 SSLACC 600 performance

Our second major performance metric is throughput. Our SSLACC test units are fitted with two 100 MHz Ethernet interfaces and we're capable of saturating the network with a single unit. We don't currently have SSLACC units with gigabit Ethernet interfaces so we have not yet determined the performance of multiple units with higher loads. Since cluster traffic is carried over the same wire as data traffic, the amount of capacity consumed by cluster traffic (in particular, update

ACKs) increases as the number of nodes increases. Thus, as we add cluster nodes, the overall throughput of the cluster decays, even though the entire network capacity is being used. Adding a third network interface for cluster traffic or converting to gigabit Ethernet would be obvious solutions to the problem of limited network capacity, but we have not benchmarked either.

5.9 Conclusions

SSLACC occupies a new point in the clustered accelerator design space. Three of the most desirable properties in a clustered accelerator are scalability, high availability, and the ability to run on cost-effective hardware. The current generation of SSL accelerators run on commodity hardware and performance scales more or less linearly with the number of nodes. Traditional clustering solutions can achieve high availability by using special hardware and at the expense of poor scalability. The IP clustering design used in SSLACC provides high availability and some scalability on commodity hardware. Although we never expect to achieve perfect linear scaling due to the demands of clustering, we anticipate substantial improvements as the result of further tuning.

The primary obstacle to using the IP clustering approach in general is that it comes at a substantial engineering cost. The primary engineering work on SSLACC consumed roughly 3 man years. Much of this time was spent in understanding the subtleties of clustering TCP applications and thus could be leveraged to reliably cluster other protocols.

A.1 Asynchronous I/O and Cryptography

SSLACC runs as a single process under AlchemyOS. In order to simultaneously serve multiple clients and servers, all I/O is done in non-blocking mode. Instead of using `select()`, AlchemyOS offers an asynchronous version of `select()` called `async_select()`. `async_select()` allows the programmer to register callbacks for the usual I/O events (read, write, exception) as well as an extra one (`writedrain`— indicating that data has been ACKed by a peer). Callbacks are fired synchronously: when an application enters the event loop by calling `Hibernate()` pending callbacks fire, in no particular order.

Cryptographic operations may also complete asynchronously, typically if they are being handled in hardware. Our cryptographic API allows the caller to provide a callback. If the operation is to be performed asynchronously the API point returns a result code indicating this. When the operation completes the callback

fires. SSLACC can then collect the output of the operation.

Finally, cluster messages are transmitted asynchronously. The programmer calls `IP_Cluster_Send()` and provides a callback which will be called when the message is acknowledged.

B.1 Destruction

Neither cryptographic operations nor cluster updates can be cancelled. This creates an interesting problem. Consider what happens if an attempt is made to destroy a connection which has a pending operation (perhaps due to errors or premature closure by a peer). If the connection context is destroyed immediately, the callback will be left holding a dangling pointer. Attempts to operate on such pointers generally result in segmentation faults.

SSLACC uses a complicated system of interlocks to prevent this case. If a request is received to destroy a quiescent connection, that connection can simply be destroyed immediately. If callbacks are pending then the connection is marked for destruction but not immediately destroyed. When a callback fires for such a connection, it checks to see if all callbacks have fired and the connection can be destroyed. The last callback destroys the connection.

If the interlocks are misconfigured, we either see panics (if we destroy too soon) or memory leaks (if we fail to destroy when the last callback fires). The complication arises because the interlocks are woven into the (rather complex) state transitions that a connection goes through as it closes. Interlock issues have been a substantial debugging problem, which might potentially be alleviated by a simpler design such as a reference count.

Acknowledgments

The authors would like to thank Jeremy Barrett, David Kashtan, Tom Kroeger, Stacey O'Rourke and Craig Watkins for their contributions to the development of SSLACC. Additionally, we would like to thank Derek Atkins, Steve Bellovin, Kevin Dick, and the anonymous USENIX reviewers for their comments on this paper.

References

[1] SonicWall, *High Availability Options for SonicWALL SSL Devices*.
http://www.sonicwall.com/products/↓documentation/High_Availability_SSL.pdf

[2] Schultz, K., "SSL In the Driver's Seat," *InternetWeek* (November, 2000).

[3] Intel, *Commerce Accelerator 1000 User Guide*.

[4] Postel, J., "Transmission Control Protocol," RFC 793 (September 1991).

[5] Hickman, K., *The SSL Protocol* (February 1995).
http://www.netscape.com/eng/security/↓SSL_2.html

[6] Freier, A.O., Karlton, P., and Kocher, P.C., *The SSL Protocol Version 3.0* (November 1996).

<http://home.netscape.com/eng/ssl3/↓draft302.txt>

[7] Dierks, T., and Allen, C., "The TLS Protocol Version 1.0," RFC 2246 (January 1999).

[8] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T., "Hypertext Transfer Protocol," RFC 2616 (June 1999).

[9] Rescorla, E., "HTTP over TLS," RFC 2818 (May 2000).

[10] Kant, K., Iyer, R., and Mohapatra, P., *Architectural Impact of Secure Socket Layer on Internet Servers*.

<http://www.cse.msu.edu/rgroups/isal/↓pubs/conf/iccd00.ps>

[11] Rescorla, E., *SSL and TLS: Designing and Building Secure Systems*, Addison-Wesley, New York, NY (2000).

[12] Abbott, S., and Keung, S., *CryptoSwift (ver:2) Performance on Netscape Enterprise Server* (April, 1988).

<http://isglabs.rainbow.com/isglabs/↓NS351-CSv2-NT-perf/NS351-CSv2.html>

[13] Keung, S., *Cryptoswift performance under SSL with file transfer* (1998).

<http://isglabs.rainbow.com/isglabs/↓SSLperformance/SSL+file%20performance.↓html>

[14] Nokia, *The IP Clustering Power of Nokia VPN* (April, 2001).

http://www.nokia.com/vpn/pdf/↓ip_clustering.pdf

[15] Nick, J.M., "S/390 cluster technology: Parallel Sysplex," *IBM Systems Research Journal*, 36, 2 (1997).

[16] Laranjeira, L., "NCAPS: Application High Availability in UNIX Computer Clusters," *Proc. 28th Intl. Symp. on Fault Tolerant Computing*, pp. 441-450 (June 1998).

[17] Microsoft, *Windows 2000 Clustering Technologies*.

<http://www.microsoft.com/windows2000/↓technologies/clustering/default.asp>

[18] Barak, A., La'adan, O., and Shiloh, A., *Scalable Cluster Computing with MOSIX for LINUX* (1999).

[19] Alvisi, L., Bressoud, T.C., El-Khashab, A., Marzullo, K., and Zagarodnov, D., "Wrapping Server-Side TCP to Mask Connection Failures," *IEEE INFOCOM 2001* (2001).

[20] Elnozahy, E., Alvisi, E., Wang, Y.M., and Johnson, D.B., "A Survey of Rollback-Recovery Protocols in Message Passing Systems," *CMU Technical Report CMU-99-148* (June 1999).

[21] CERT, "TCP SYN Flooding and IP Spoofing," CERT Advisory CA-96.21.

ftp://info.cert.org/pub/cert_advisories/CA-96.21.tcp_syn_flooding

[22] Nagle, J., "Congestion Control in IP/TCP Internetworks," RFC 0896 (Jan 1984).