Man-in-the-Middle TCP Recovery

Robert Surton
Cornell University

Ken Birman
Cornell University

Robert Broberg
Cisco Systems

Tudor Marian
Cornell University

Robbert van Renesse
Cornell University

Abstract

When an application connects with a remote peer using TCP, its network stack encapsulates the state of that connection. As a consequence, even if the application is capable of recovering its own state after failure (for example, by restarting at a backup location), the peer may experience a disruption of connectivity due to the loss of the network stack's state. We present RTCP, a new tool that enables fault-tolerant applications to recover connections in a manner hidden from their peers, without changes to the network stack, the operating system, or the remote endpoints. RTCP functions as a middle-man on the network, filtering packets in order to glean the internal state of the network stacks and manipulating them in a manner that masks failures and restarts. Unlike prior work on TCP recovery, RTCP is remarkably simple: it maintains only 32 bytes of state per connection, buffers nothing, does not assume determinism in the network stack, and has no notion of time. If the RTCP filter itself fails, once restarted it can to recover its state from the endpoints, even if the protected endpoint crashes simultaneously. We present the RTCP architecture and evaluate its impact on throughput and latency, comparing a protected TCP connection with an unprotected TCP connection that experiences no failures, benchmarking CPU and packet filtering overheads, and measuring scalability and speed of recovery when many protected connections are active.

1 Introduction

Fault-tolerant applications replicate their state so that even if one replica crashes, some other replica can transparently take over on its behalf. However, if an application connects with a remote peer using TCP, one critical part of the application is inaccessible, namely the connection state. Network stacks typically live within the kernel behind a socket interface, and the associated state is protected. As a result, even if the application recovers to its exact state before the crash, its connections will break and its remote peers will have to cope with "Connection reset by peer" errors.

Our paper reports on RTCP, a network service providing low-cost, transparent connection recovery, which we created as part of a joint project with Cisco Systems to develop a proof-of-concept next-generation backbone router. The routers used in this effort are designed as clusters, with large numbers of "line cards" for routing data-plane packets, and a few "route processors" providing control-plane functionality. For example, BGP traditionally runs on route processors, connecting with remote BGP peers on other routers over TCP. The goal of our effort was to enable the creation of fault-tolerant BGP and other routing services that can scale out over multiple cards to tolerate failures and gain performance by better utilizing the cluster's computing resources. With RTCP, such a BGP implementation can conceal the failure of the BGP instance currently responsible for a given remote peering relationship, shifting that role to some other BGP instance, which takes over the connection in a seamless, non-disruptive manner. That might occur because of a software failure, the loss of an entire line card, or because of a software upgrade or downgrade to a different version or implementation of BGP.

The importance of a solution such as RTCP reflects the poor handling of connection loss in many kinds of long-term peered services. For example, BGP cannot tolerate connection loss, as discussed in RFC 4724:

> Usually, when BGP on a router restarts, all the BGP peers detect that the session went down and then came up. This "down/up" transition results in a "routing flap" and causes BGP route re-computation, generation of BGP routing updates, and unnecessary churn to the forwarding tables. It could spread across multiple routing domains. Such routing flaps may create transient forwarding blackholes and/or transient forwarding loops. They also consume resources on the control plane of the routers affected by the flap. As such, they are detrimental to the overall network performance.

That RFC describes a mechanism for "Graceful Restart", but that feature is not widely supported in the Internet and, in any case, is specific to BGP. A modern router might host many kinds of distributed routing services and other applications, and quite a few of these

would potentially need an RTCP-like capability to conceal failures or reconfiguration from remote peers.

There has been significant prior work on TCP connection recovery [9, 7, 8, 11, 2, 5, 6, 3, 1, 12, 13], but existing solutions turn out to be too intrusive and costly for our target setting. For example, many prior solutions require source-level access to the network stack, involve changes to the kernel thread-scheduling layer, and insert some form of data checkpointing or even multicast operation in the critical path between the remote peer and the local TCP endpoint. A further consideration is that BGP itself uses TCP in idiosyncratic ways. For example, BGP is the motivation for and primary consumer of TCP's MD5 Signatures option. Many TCP implementations do not support MD5 signatures, including, until recently, the one in Linux, and the high-end BGP implementations on which we focus employ custom network stacks. More broadly, high-end routers often have custom, or at least customized, operating systems and network stacks.

This motivated us to ask a basic question: can connection availability be achieved without modifying the operating system or network stack, without making assumptions about their behavior other than those dictated by the TCP standard, and without modifying the behavior of the remote peer? This problem statement forces us to focus on the network path over which packets flow within the router, and to involve the application in its own protection.

RTCP makes three contributions to the state of the art. First, it provides applications with explicit control over TCP acknowledgment. As we will see below, this control gives the application an opportunity to replicate data received on a TCP session before that data is acknowledged, eliminating the need for any form of redundancy within the protected TCP connection itself. Prior work has employed similar techniques, but by exposing them to the protected application, it turns out to be possible to integrate the needed coordination with whatever replication mechanisms that application may be using. Second, RTCP is portable, protecting connections without needing access to (or cooperation from) the internal state of the operating system or network stack, and making no assumptions beyond conformance with the TCP standard. Third, RTCP is lightweight, keeping a constant amount of state per connection that is smaller than the connection state in the endpoints, and having an exceptionally simple implementation. The small, fixed-size state and simple processing translate into low latencies and unobtrusive protection.

2 Design

RTCP is designed as a man-in-the-middle network filter, observing and manipulating the packets transmitted on protected connections, and providing a side-channel by which applications can learn the necessary connection state so as to recover in a manner completely transparent to remote connected peers, and with no changes of any kind to the operating system or network stack on the local or remote hosts. Should RTCP itself fail, it recovers its state from the endpoints. The main requirement from the protected application is that it use any a fault-tolerance mechanism (for example, a checkpointing mechanism) capable of reporting the number of bytes received and safely stored or processed, and that it uses a TCP implementation that complies with the published standards. Unlike some prior work, RTCP does not require that the application be deterministic or that it implement fault-tolerance in any particular manner.

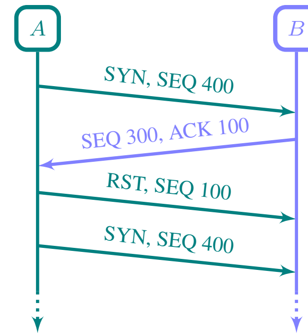

Figure 1: Reconnection. The TCP standard describes a non-transparent "recovery" procedure, in which A's TCP causes B's TCP to abort the half-open connection, and then reconnects from scratch.

2.1 Reconnection

The TCP standard describes the failure scenario and approach to recovery that also underlies RTCP:

> Assume that two user processes A and B are communicating with one another when a crash occurs causing loss of memory to A's TCP ... When the TCP is up again, A is likely to start again from the beginning or from a recovery point ... the user attempts to re-open the connection. TCP B, in the meantime, thinks the connection is open.

Non-transparent reconnection, shown in Figure 1, begins when A's TCP sends a packet with the SYN flag and a fresh, random initial sequence number. The response A's TCP expects is an acknowledgment from B's TCP that it received the SYN, carried in a packet with its own SYN flag and the B's new initial sequence number. However, because B's TCP still believes itself to be connected and synchronized, A's SYN is not acceptable, and the response to an unacceptable packet on a synchro-

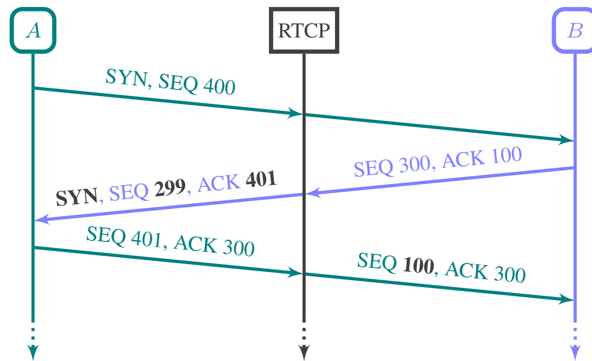

Figure 2: Resynchronization. A man-in-the-middle can rewrite the response from B's TCP to meet the expectations of A's TCP, resynchronizing the half-open connection without tearing it down or creating an entirely new connection.

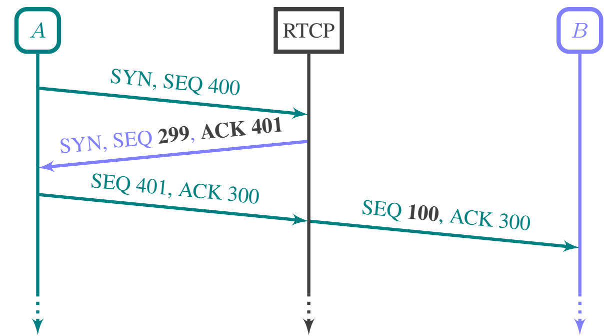

Figure 3: Optimized resynchronization. RTCP keeps enough state to eliminate a round trip on recovery, by rewriting the SYN from A's TCP into its own response, rather than waiting for B's TCP.

nized connection is an empty packet, bearing the current sequence number and acknowledgment. A's TCP, being unsynchronized but receiving what from its perspective is an incorrect acknowledgment, uses that information to craft a RST packet acceptable to B's TCP—which, upon receiving it, immediately aborts the old connection. A new connection can now be established by following the standard handshake protocol when A's TCP retransmits its SYN packet.

2.2 Resynchronization

A man-in-the-middle, such as RTCP, can rewrite packets to resynchronize A's TCP without causing B's TCP to abort, as shown in Figure 2. The reason the original connection aborted in the example was because the connection was revealed to be half-open when A's TCP received a non-SYN packet with the wrong acknowledgment. Rather than letting A's TCP see that packet, the illusion of a seamless connection can be maintained by setting the SYN flag and adjusting the acknowledgment. A's TCP can be synchronized to accept the sequence numbers sent by B's TCP, despite the fact that B's TCP believes itself to still be using the original connection, by decrementing the sequence number to account for the extra SYN injected in the stream.

A similar trick is needed for outgoing packets. Recall that A's TCP selected a new sequence number, so that and subsequent sequence numbers emitted by A's TCP will not be acceptable to B's TCP. To enable communication, the man-in-the-middle must remember the amount Δ by which it had to increase the acknowledgment from B's TCP, add Δ to the sequence numbers on future incoming packets originated by B's TCP, and subtract Δ from outgoing sequence numbers on packets produced by A's TCP. The modified TCP packets require new checksums, which can be computed from the original checksums using the same sort of incremental checksum computation employed when routers decrement IP time-to-live values. The result is to splice the two half-open connections together in a manner that costs constant time and requires only a few instructions.

RTCP further optimizes the recovery handshake, as shown in Figure 3: Rather than saving the sequence number from A's TCP, and waiting for B's TCP to provide its latest acknowledgment and a packet to rewrite, RTCP saves the acknowledgments it sees from B's TCP, so it can immediately rewrite the recovery SYN from A's TCP into its own response, thereby saving a round trip.

A single RTCP filter can handle multiple connections in parallel, for multiple protected applications. Using a constant-cost lookup table to access the state associated with each active connection, RTCP's actions have fixed cost regardless of the number of streams being protected. In our discussion, we consider a particular connection, referring to the protected application as A and the remote peer as B; if desired, a second RTCP filter could protect the remote side, as well. Thus, RTCP functions as a kind of gateway, mediating between the fault-tolerant applications within the router or local cluster and their peers without.

2.3 Retransmission

Once A has re-established the connection, it might be necessary for A and B to retransmit some data that was lost when A crashed.

2.3.1 Outgoing

When A calls `send`, data is buffered in its TCP, but might not actually be sent on the network until later. When A crashes, buffered but unsent data is lost; data that had been sent, but which the network drops, is also lost. Thus, A needs a mechanism to learn how many

bytes it sent B actually received, and should resume sending from that point. Because the standard socket abstraction provides no way to obtain that information, RTCP offers a UDP-based side channel whereby A can communicate directly with the man-in-the-middle. When A first establishes a new connection with B, it sends a UDP packet containing an `RTCP_TELL` command, which RTCP intercepts, rewrites to contain the latest acknowledgment from B (covering A's initial sequence number), and then returns, by swapping the source and destination. A persistently stores the acknowledgment, and, upon recovering, sends another `RTCP_TELL` command to learn the most recent one. Subtracting the latest acknowledgment from the first reveals how many bytes B has definitely received. Actually, it reveals the number of bytes modulo 2^{32}, because sequence numbers can wrap; A can disambiguate by remembering how much it believes it has sent, and using the `RTCP_TELL` information to determine within a window of the last four gigabytes how much was successfully delivered.

RTCP does not require that A be a deterministic program, but it does require a property that we'll refer to as *output determinism*, namely that any bytes actually written by A to its endpoint be reproduced in the event that A fails, recovers, and the acknowledgment from B is for a smaller sequence number. Suppose, for example, that A sends 1550 bytes to B, and the last 50 are in a packet that gets delayed in the network. Now A crashes and recovers, and the `RTCP_TELL` command returns 1500, because B has yet to receive those last 50 bytes. A race now ensues: the remaining 50 bytes could reach B, which will consume them, or A's next transmission could reach B, in which case the next 50 bytes will be those produced subsequent to A's restart. Clearly, those bytes must be identical. Thus A must be deterministic only with respect to retransmitting potentially lost portions of its output.

How can A achieve this property? If the application is deterministic, A just needs to resume from some checkpoint state and then accept any incoming bytes beyond the saved incoming sequence number. However, in modern settings, most programs are nondeterministic: an inescapable consequence of writing code that may use multiple threads, receive inputs, read clocks, accept timer interrupts, and have multiple input sockets. In that case, A should simply checkpoint its output prior to sending it. In that manner, the restarted A can resume by completing the interrupted output (if any), and then continue in a manner that might be very different from what the prior instance of A would have done.

Notice that this also suggests that the new instance of A might remain healthy even if the previous instance crashed while processing the next received segment from B. Nonetheless, applications using RTCP should be wary of poison pill situations that, with RTCP in the picture, could be transformed into infinite restart/crash loops due to a bug processing the continually pending input.

2.3.2 Incoming

When A's TCP receives a packet from B's TCP, it buffers the data until A consumes it with a call to `recv`; meanwhile, it might acknowledge the data at any time. If A's TCP does acknowledges data, but A crashes before receiving it at the application level and either checkpointing it or acting on it, it is lost. To avoid that risk, we make use of the fact that if no acknowledgment had been sent by A's TCP, B's TCP would have continued to retransmit the packet until successful and the data would not have been lost. Accordingly, RTCP delays acknowledgments until the application has confirmed that the incoming data is safely consumed. In effect, RTCP only allows A's TCP to send *checkpointed acknowledgments* to B's TCP.

The RTCP solution, which requires cooperation from the application, is not the only way of solving the problem. Some prior work involves replicating incoming packets within a group of agents, so that, in effect, A's TCP stack is replicated. We rejected such an approach because we felt that the needed mechanisms would unnecessarily complicate RTCP, forcing it to maintain checkpoint state and requiring a mechanism whereby those replicated endpoint stacks could coordinate timeout-based actions. Any timeouts or retransmissions would be redundant with what A's TCP already does, and, because we assume A is fault-tolerant, any checkpointing would be redundant with what the application already does. The only help RTCP requires for checkpointing acknowledgments is in the form of the four byte ACK field that the application persists along with the rest of its state.

Whenever A finishes storing or processing some of its input, it notifies RTCP by sending an `RTCP_ACKNOWLEDGE` command. When RTCP receives one of these `RTCP_ACKNOWLEDGE` commands, it notes the new sequence number that A's TCP is permitted to acknowledge. Each time A's TCP transmits a packet, RTCP rewrites the acknowledgment field using the current `RTCP_ACKNOWLEDGE` value. This is a sufficient mechanism to ensure safety, but it could create a different problem: A's TCP will believe that it already acknowledged the data, so it does not need to generate a new acknowledgment until B sends new data. Unless B has more data to send immediately, it just will wait for an acknowledgment that never comes, until it gives up and retransmits the data, which induces A's TCP to acknowledge it again successfully. To avoid such time-

outs, we take one additional step: RTCP also transforms the `RTCP_ACKNOWLEDGE` command packet into a fresh TCP acknowledgment, which it sends to B's TCP. The result is that B's TCP sees an acknowledgment as soon as the associated data is safely checkpointed.

One might worry that in delaying the TCP acknowledgment RTCP could cause the TCP stack on its remote peer to misbehave. Fortunately, many TCP stacks already delay acknowledgments, for an entirely different reason formalized in RFC 1122. That RFC describes optimizations that combine multiple acknowledgments associated with empty packets so that they can be piggybacked on some future packet containing outgoing data, in order to reduce wasted bandwidth.

Thus, RTCP is able to limit connection state to a small, fixed amount of data, avoid any need to multicast or otherwise replicate incoming data, and (in effect) achieve a powerful change in the semantics of TCP acknowledgments. TCP is touted as an example of the end-to-end principle [10], because it implements features such as reliability and flow control from end-host to end-host without network support; however, it ignores the end-to-end principle at work between application. RTCP corrects that oversight, and in place of the standard "received by A's TCP", an acknowledgment now means that the associated data has been "received and consumed by A (the application)".

2.4 Masking failure

Because TCP is designed to avoid half-open connections, if B's TCP detects that A has failed, it will abort. A would then recover only to find its peer dead, recalling the end of one of Shakespeare's most famous tragedies. This is a serious issue in RTCP, because one of the failure modes we need to contend with involves a crash of the application that might leave the operating system and TCP stack alive. In such a state, there are two main ways A's TCP might reveal A's demise to B's TCP: by sending a RST in response to incoming packets, or by injecting FINs to helpfully close A's connections on its behalf.

2.4.1 Spurious RSTs

If any packets from B's TCP arrive while A is crashed or not finished recovering, A's TCP (or perhaps its old TCP, if A has migrated), lacking a synchronized connection, will answer with a RST.

The simplest solution is to drop all outgoing RST packets. A correct TCP only sends RST in response to unacceptable packets received for unsynchronized connections; thus, whenever the connection should be synchronized, A's TCP should be prevented from erroneously resetting B's TCP and dropping such attempts is correct.

The connection is legitimately unsynchronized, however, while it is still first being established. RTCP thus prevents normal handling of errors in connection setup, such as the arrival of old duplicate SYN packets from B's TCP. As long as the new version of A recovers sufficiently quickly, dropping the RST packets will have no effect on the correctness of the protocol, since TCP is designed to handle packet loss. Obviously, the behavior of hiding A's death assumes that A will indeed eventually recover—if it is permanently dead, it must be cleaned up at some point in the manner described in subsection 2.5.

2.4.2 Spurious FINs

Depending on how A crashes, its TCP might send a FIN before A is done sending its data. Specifically, when a program terminates, the operating system closes all its file descriptors; on Linux, all its sockets are implicitly closed. Thus, the TCP, which resides in the kernel and survives any program's death, finishes retransmitting any outstanding data and tries to cleanly close the connection by sending a FIN.

RTCP requires a way of distinguishing between legitimate FIN packets and those that are spuriously injected by the TCP endpoint during recovery. To this end, RTCP has a side-channel command whereby A can share the needed information. By default, RTCP will assume any FIN from A's TCP is spurious. A sends `RTCP_SHUTDOWN` when it is ready to close the connection; upon receiving it, RTCP stops checkpointing acknowledgments and expects a FIN. In an earlier version of the RTCP system, we believed that it would be necessary to include an argument for the `RTCP_SHUTDOWN` command, specifying the sequence number of the permitted FIN. As it turns out, the extra information is unnecessary. Although the TCP might try to close a socket before A is ready, it first finishes sending enqueued data. Thus, as long as A's waits for its last `send` to succeed before sending the `RTCP_SHUTDOWN`, no issue arises. If the TCP itself crashes, the enqueued data might not be completely sent, requiring A to recover, but there will be no spurious FIN. Should A send `RTCP_SHUTDOWN` and then need to recover, RTCP, as part of the recovery, automatically returns to its default behavior before the `RTCP_SHUTDOWN`, in order to prevent a race condition should A crash again immediately before re-enqueueing its final output. When it is ready, A sends a fresh `RTCP_SHUTDOWN`.

When RTCP detects a spurious FIN, it could drop it, as though it were a RST (the problem is the same). However, that leaves A's TCP hanging in the state in which it waits for its FIN to be answered (FIN-WAIT-1), which ties up the connection for a timeout of minutes, during which time A cannot recover the hung TCP session. RTCP could instead rewrite the FIN into its own answer, letting A's TCP advance to FIN-WAIT-2, or even spoof

a FIN from B's TCP on top of that, taking it all the way to TIME-WAIT. However, even in the latter case, there is a timeout during which time the connection cannot be recovered. The `SO_REUSEADDR` socket option, where available, permits rebinding the connection while it is still in TIME-WAIT, but a better approach would be to remove the connection completely. Fortunately, there is a way to do just that: RTCP rewrites the spurious FIN into a RST that causes A's TCP to abort the connection immediately.

2.5 Cleaning up

After a connection closes, the TCP stack retains its information for a timeout of twice the maximum time a packet might spend in the network, in the TIME-WAIT state. After that timeout, the TCP stack assumes it is safe to forget the connection. However, for simplicity RTCP is designed as a purely reactive packet filter and has no built-in notion of time. As a result, although RTCP can detect when FINs have been sent both ways, it cannot implement TIME-WAIT.

Rather than introducing a notion of elapsed time, we solve this by adding a more general `RTCP_CLEAR` side channel command, which causes RTCP to zero the connection state immediately. The explicit command enables applications or an administrator to choose when it is safe to forget connections. By deleting the state for a connection in progress, it also provides a means to quickly and remotely simulate RTCP's failure. The application A should not sent `RTCP_CLEAR` until it is sure that no more packets will be sent, or RTCP will break the already-dead connection and try to recover.

In the RTCP implementation, which will be discussed at length in section 3, connection state is held in a lookup table by a packet filtering program, but actually manipulated in a portable RTCP library; the `RTCP_CLEAR` command causes the library to clear the state, but when does the filter program know it is safe to remove the state from its table? One choice is for the program to understand enough of RTCP to react to `RTCP_CLEAR` as well. Furthermore, because RTCP can recover from its own failure, even on a per-connection basis, the program always has the option to delete any connection state at any time, perhaps after an inactivity timeout or eviction from a constant-sized set of connection states. If it later turns out that the connection is still needed, RTCP will automatically recover that state when it sees traffic on the connection again.

2.6 Options

The TCP header size is 40 bytes, and the maximum data offset is 60 bytes; the difference is to accommodate up to 20 bytes of options. TCP implementations that conform to RFC 793 must support three such options: Maximum Segment Size, sent on SYN packets to enable endpoints to avoid fragmentation; No-Operation, a one-byte option that can be used for padding; and End of Option List, a one-byte option that may be used to indicate the end of the options if that would not coincide with the end of the header. RFC 1323 was the first to expand the possible options, with Window Scaling and Timestamps. The authors noted that the only previous meaningful option, Maximum Segment Size, could only appear in a SYN packet, and worried that buggy TCP implementations might erroneously fail to ignore unknown options on non-SYN packets. To address this concern, they established the continuing practice of negotiating TCP options in the handshake, thereafter using only those supported by the other side. Other RFCs later defined further options, notably including Selective Acknowledgment and MD5 Signatures.

The current implementation of RTCP suppresses unrecognized options by overwriting them with No-Operations. Due to the practice established by RFC 1323, by doing so on SYN packets, correct TCP implementations will not give RTCP any unwanted options to suppress on future packets. Currently, the supported options include those required by RFC 793 as well as those proposed in RFC 1323; it is worth describing some of the pitfalls and possibilities that other options will raise as they are supported as well, and mentioning some aspects of lower-level IP options.

2.6.1 Selective Acknowledgments

Selective Acknowledgments enable a TCP to acknowledge data that it receives out of order or with gaps. Advancing the cumulative acknowledgment in the ACK field to cover the received packet would erroneously include the gaps. A consequence is that the remote peer of an RTCP connection might wastefully retransmit data that hasn't really been lost.

At first glance, Selective Acknowledgments should wreak havoc on RTCP's checkpointed acknowledgments. However, according to Section 3 of RFC 2018:

> The SACK option is advisory, in that, while it notifies the data sender that the data receiver has received the indicated packet, the data receiver is permitted to later discard data which have been reported in a SACK option.

Thus, both ends are free to acknowledge what they will, and failing to recover such data will not impact correctness. RTCP's only involvement is to apply Δ to the selective as well as cumulative acknowledgments of B's TCP.

Not only do Selective Acknowledgments pose no problem, they can provide a way to safely and imme-

diately notify B's TCP of what A's TCP has buffered, before A's checkpointed acknowledgments catch up.

2.6.2 MD5 Signatures

The MD5 Signature option includes an MD5 checksum of the packet, with the same pseudo-header and so forth as for the standard TCP checksum, appended with a connection-specific password. Thus, it functions as a sort of HMAC, or symmetric signature. The option was proposed to protect BGP sessions, and we believe that BGP is still the main (perhaps only) application to use this option. To spoof or modify packets on a connection with MD5 signatures, an attacker would not only have to guess the relevant sequence numbers, but also the connection's password, which is configured out-of-band and never transmitted. However, because BGP is a primary target of our work, adding MD5 Signature support to RTCP is part of ongoing work.

Doing so is not as simple as one might wish. RTCP is a man-in-the-middle that just happens to be beneficial; the MD5 Signature option is designed to thwart men-in-the-middle attacks. Clearly, RTCP cannot protect a connection with MD5 Signatures without knowing the password. The solution is straightforward: we clearly need a side-channel command whereby the application can give RTCP the session password.

Transmitting a password to RTCP over the side channel violates the normal requirement that session passwords never be transmitted; however, we note that a malicious intruder with access to the side channel would also be able to query the current sequence numbers, arbitrarily change the ACK checkpoint, and kill the connection without possibility of recovery. For this reason, we have concluded that the path between applications and RTCP must be a secured one, and that RTCP should be limited to receive command packets only from trustworthy sources. There are many ways to provide these guarantees; the simplest option is to run RTCP on the same node that runs A's TCP, since the security material is physically on that node. A more careful study of security is beyond the scope of the present paper. In that case, of course, processor crashes would always kill RTCP as well as the application and its TCP, but the system can recover from simultaneous RTCP and application failure, and colocating A and RTCP could have performance benefits as well.

2.6.3 IP options

The IP header also supports options, including No-Operation and End of Options List (as for TCP), as well as timestamps, various kinds of source routing, and a means of specifying security compartments. Many IP options, notably source routing, are poorly supported and rarely used; the current RTCP implementation neither uses nor bothers to suppress them.

IPsec adds security to the IP layer; although implemented as protocols layered between IP and (in this case) TCP, it can also be seen as particularly heavyweight IP options. Fundamentally, RTCP could support IPsec in the same manner as MD5 Signatures, by adding a side-channel command whereby the endpoint could share the needed secrets with the RTCP filter. However, support for IPsec is significantly more complex than storing a password and recomputing an MD5 checksum, and although we do believe the problem to be solvable, we have no near-term plans to support this protocol feature.

One work-around would be to use IPsec in its Tunnel Mode, in which whole packets are encapsulated within the security layer. Tunnel Mode is widely used for such purposes as establishing Virtual Private Networks, and as long as RTCP sees the packets before or after the tunnel, there is no need for it to be aware that the encapsulated protocol is using IPsec.

2.7 Recovering RTCP

Should RTCP itself crash, it is able to recover its state from the endpoints, preventing it from becoming a single point of failure. Any packet from B's TCP, for example, reveals its latest sequence number and acknowledgment. Some of RTCP's state is already kept by A; for example, the next `RTCP_ACKNOWLEDGE` command RTCP receives from A implicitly recovers the ACK checkpoint. However, RTCP must be able to ensure that A will send such a command in a timely manner, because it cannot safely forward any packet from A's TCP without knowing if it acknowledges uncheckpointed data. Furthermore, some of RTCP's state, such as Δ, is internal and cannot be recovered directly from A.

All of the state RTCP needs to recover a connection, on the other hand, is implicitly recomputed during the recovery of A. So, if RTCP finds itself in a situation in which it cannot proceed—for example, when handling a packet from A but lacking the ACK checkpoint—it forces A to recover by rewriting the offending packet into a RST for A's TCP. A will receive an `ECONNRESET` ("Connection reset by peer") error or something similar, revealing that it must recover to continue.

As part of its recovery, the application will first send its latest `RTCP_ACKNOWLEDGMENT`, bringing RTCP up to date on the checkpointed acknowledgment. Then it will recover the connection as usual, which enables RTCP to recompute the offset Δ. Once it is reconnected, the application learns where to resume from, with its usual `RTCP_TELL`, and continues.

Indeed, that mechanism is the reason the system can handle simultaneous failure of RTCP and the application; in fact, it is simpler in that case than if just RTCP

were to fail, because there is no need to kill the application to trigger its recovery. The essential technique is to transform RTCP failure into application failure, allowing us to leverage the presumed application replication mechanism as a way to avoid state replication within RTCP itself.

3 Implementation

RTCP addresses five main concerns, some of which require state. Where state is required, the mechanism must work correctly when the connection is first established, while it is being established, when A has crashed, when A recovers, and when RTCP has crashed. When RTCP looks up the connection state for a particular packet it is processing and finds none, it creates a fresh state filled with default values: false for flags and zero for words. RTCP's interpretation of the default values accommodates the case where the state is new because the connection is new, as well as the case where the state is new because RTCP has crashed and restarted. The five concerns break down as follows:

Suppress options. Scan incoming and outgoing option lists to suppress unrecognized options. (No state is necessary.)

Suppress spurious reset. Drop outgoing RSTs. (No state is necessary.)

Reset spurious close. Keep a flag *fin_allowed*. On an `RTCP_SHUTDOWN` or `RTCP_SHUTDOWN_WRITING`, set *fin_allowed*. On an outgoing FIN, if $\neg$*fin_allowed*, rewrite the packet into an incoming RST.

Resynchronize. Keep two words, Δ and $ackB$, and a flag $recoverable$, which indicates whether $ackB$ is meaningful. On an incoming ACK, store it in $ackB$ and set $recoverable$. On an outgoing SYN with no ACK (A's first packet when it is a client): If $recoverable$, let $\Delta \leftarrow seq + 1 - ackB$, and rewrite the packet into its own handshake response; if $\neg recoverable$, forward the packet unchanged, to permit initial connection or elicit an empty packet from B (to reveal $ackB$ and set $recoverable$ for the retransmitted SYN, in case of recovery). Add Δ to all incoming acknowledgments and subtract it from all outgoing sequence numbers.

Checkpoint outgoing acknowledgment. Keep a word, $ackA$, storing the checkpointed acknowledgment, a flag $ack_allowed$, which indicates whether $ackA$ is meaningful, and a flag $ack_everything$, which indicates whether to acknowledge everything, regardless of the checkpointed acknowledgment. On an incoming SYN, let $ackA \leftarrow seq + 1$ and set $ack_allowed$. On an `RTCP_ACKNOWLEDGE`, store the command's argument in $ackA$ and set $ack_allowed$. On `RTCP_SHUTDOWN` or `RTCP_SHUTDOWN_READING`, set $ack_everything$. On an outgoing ACK: if $\neg ack_allowed$, rewrite the packet into an incoming RST; otherwise, if $\neg ack_everything$, $ack \leftarrow ackA$;

3.1 Algorithms

Reorganizing the separate concerns into the algorithms used to process packets will better elucidate RTCP's implementation:

Incoming. Process an incoming packet from B, updating the connection's state within RTCP, rewriting the packet as needed, and forwarding or dropping it.

I1. [Handle options.] Scan incoming and outgoing option lists to suppress unrecognized options by overwriting them with No-Operation bytes.

I2. [Initialize checkpointed acknowledgments.] If the SYN flag is set, that is, if SEQ is B's initial sequence number, let $ackA \leftarrow$ SEQ $+ 1$, and set $ack_allowed$.

I2. [Handle acknowledgment.] If the ACK flag is set, let $ackB \leftarrow$ ACK and ACK $\leftarrow$ ACK $+ \Delta$, and set $recoverable$.

I4. [Forward packet.] Update the checksum and forward the packet. ■

Outgoing. Process an outgoing packet from A, updating the connection's state within RTCP, rewriting the packet as needed, and forwarding or dropping it.

O1. [Suppress resets.] If the RST flag is set, drop the packet.

O2. [Handle options.] Scan incoming and outgoing option lists to suppress unrecognized options by overwriting them with No-Operation bytes.

O3. [Attempting recovery?] If the ACK flag is set, skip to step O6. (The ACK flag is set on every packet but the client's SYN, so its absence indicates A is attempting to connect.)

O4. [Recovery possible?] If $\neg recoverable$, either A is connecting for the first time, or RTCP has lost the connection state: Forward the packet unchanged. (In the former case, B will answer with a SYN-ACK; in the latter, B will answer with an empty ACK, which will enable recovery when A's TCP retransmits the SYN.)

O5. [Recover.] Recompute Δ. Unset *fin_allowed*. Rewrite the packet into the SYN-ACK response A's TCP expects, and forward it.

O6. [Reset?] If neither $ack_allowed$ nor $ack_everything$ is set, RTCP has lost the connection state and is unable to correctly update the ACK. If the FIN flag is set, but not *fin_allowed*, the FIN is spurious. In either case, rewrite the packet into an incoming RST and forward it.

O7 [Checkpoint ACK.] If $\neg ack_everything$, let ACK $\leftarrow ackA$.

O8. [Fix SEQ.] SEQ $\leftarrow$ SEQ $-$ Δ.

O9. [Forward packet.] Update the checksum and forward the packet. ■

Command. Process a UDP command packet, updating the connection's state within RTCP, rewriting the packet as needed, and forwarding or dropping it.

C1. [Tell.] On `RTCP_TELL`, fill in the command with $ackA$ and $ackB$.

C2. [Acknowledge.] On `RTCP_ACKNOWLEDGE`, copy $ackA$ from the command and set $ack_allowed$. Rewrite the packet into an empty outgoing ACK and forward it.

C3. [Shutdown writing.] On `RTCP_SHUTDOWN` or `RTCP_SHUTDOWN_WRITING`, set *fin_allowed*.

C4. [Shutdown reading.] On `RTCP_SHUTDOWN` or `RTCP_SHUTDOWN_READING`, set $ack_everything$.

C5. [Clear.] On `RTCP_CLEAR`, unset *recoverable*, *fin_allowed*, $ack_allowed$, and $ack_everything$.

C6. [Respond.] Change the command type to indicate a response, swap the source and destination, update the checksum, and forward the packet. ■

3.2 Packet Processing

Because RTCP is a packet processing filter, it requires a robust mechanism to capture packets from the wire, with the capacity to rewrite or drop them. Once it has the packets, RTCP must separate connection packets from its own commands by protocol, and distinguish incoming and outgoing packets by whether the destination or source is fault-tolerant. However, there is no best, portable way to filter packets, and identifying fault-tolerant applications is not a matter of technology but of policy. Therefore, a deployment of RTCP consists of two components: Portable logic and specialized packet filtering.

The logic of processing packets is encapsulated in the `librtcp` library, which exposes an interface of just three functions, one for each of the packet processing algorithms:

```
int rtcp_incoming(struct rtcp_connection *,
                  struct ip *, struct tcphdr *)
int rtcp_outgoing(struct rtcp_connection *,
                  struct ip *, struct tcphdr *)
int rtcp_command(struct rtcp_connection *,
                 struct ip *, struct udphdr *,
                 struct rtcp_command *)
```

For example, when the packet filtering program captures and identifies a TCP packet destined for a fault-tolerant application, it looks up or creates the relevant connection state and passes it to `rtcp_incoming` with pointers to the various parts of the packet in memory. The function updates the connection state and rewrites the packet, as necessary, in place, indicating by the return value whether the filter should forward or drop it. The three functions have identical calling conventions, except that in the case of `rtcp_command`, the packet is UDP, and the payload represents a `struct rtcp_command`.

The connection states are maintained in the calling program because it might know *a priori* that it will protect a particular connection, avoiding the need for lookup entirely, or a fixed set of connections, enabling a perfect hash function or just an array indexed by port number. If full generality is required, a deployment could employ a hash table on addresses and ports, such as one would find in a TCP stack.

For further optimization, if RTCP makes so many changes to a packet that incrementally updating the checksum would be more burdensome than recomputing it from scratch, rather than doing the recomputation, it sets the checksum field to zero. The packet filtering program can then recompute the checksum or take the opportunity to leverage checksum offloading if it is supported by the hardware.

RTCP is reactive, working with the packets that flow through it. It has no sense of time, and never copies packets or generates its own. To avoid issues of memory management, RTCP never grows packets, although it might shrink them; RTCP command packets have built-in padding to make that possible.

The `librtcp` code is about two hundred lines of C. Although it depends at compile time on standard headers that define various types, notably packet header structures, it has no runtime dependencies, not even the standard C library, and it should be possible to compile into a kernel.

We have implemented two packet filtering programs to drive `librtcp`. The first is based on the QUEUE target of the Linux kernel's Netfilter firewall; in addition to obvious targets such as ACCEPT and DROP, Netfilter can send packets to a queue that is accessible to userspace programs. The `rtcp-netfilter` program binds to the queue and filters the packets the kernel gives it; it identifies fault-tolerant applications according to port ranges specified as command-line arguments.

The second packet filtering program, the one evaluated in this paper, is based on NetSlices [4], a kernel modification that enables RTCP to take advantage of modern multiqueue NICs and optimized communication of packets from the device to userspace. Both programs are written in under three hundred lines of C, and are essentially glue code between the filtering mechanism and the portable RTCP logic, with a small amount of deployment logic to identify fault-tolerant applications.

It might be desirable to also have a more portable

packet filtering program; the most portable interface to packet capture is PCAP, which is supported on nearly all common operating systems. However, although PCAP provides a portable interface to capture and injection, it provides no means to drop packets or hold them to be modified before they are forwarded, so it is not directly useful for RTCP. Combined with clever configuration of the host-specific firewall, it might be possible to implement an RTCP packet filter on PCAP, but there is not yet a need for the increased portability in the filter program.

4 Writing a recoverable application

A normal socket application first acquires a file handle representing a socket with the `socket` function, then, if it is a client, connects it to a server with `connect`, which implicitly binds it to a local ephemeral port; if it is a server, it explicitly binds the socket with `bind`, does a passive open with `listen`, and loops on `accept`, accepting active opens from clients. Once the connection is established, client and server sockets are indistinguishable; both sides `send` and `recv` data according to whatever application protocol is in use, until finished, and clean up with `shutdown` or `close`.

As soon as a recoverable connection is established—immediately after `connect` or `accept`—the application initializes an RTCP command packet, using `getsockname` and `getpeername` to fill in the connection's endpoints. Before sending any data, it sets the command type to `RTCP_TELL`, sends it to RTCP's UDP address, and waits for the response with the latest acknowledgments in both directions, which, just after handshaking, reveals the initial sequence numbers. The command packet can be reused just by setting the command type to `RTCP_ACKNOWLEDGE` before proceeding; the checkpointed acknowledgment is already initialized thanks to the `RTCP_TELL`. Whenever the application saves or processed its input, it adds the number of bytes consumed to the checkpointed acknowledgment in the command and sends it to RTCP. Finally, just before calling `close`, the application sets the command type to `RTCP_SHUTDOWN` and sends the command to RTCP.

To recover, the application needs only a saved copy of its checkpointed acknowledgment, and B's acknowledgment of its initial sequence number from `RTCP_TELL`. Before reconnecting, the application sends the `RTCP_ACKNOWLEDGE` command, in order to recover the checkpointed acknowledgment if RTCP has restarted. Then, it reconnects as a normal client (even if it was originally the server). Before calling `connect`, it explicitly `binds` to its original port number, rather than letting the system allocate it an ephemeral port and thus a brand new connection. Once connected, the application sends a new `RTCP_TELL`, in order to compare the latest acknowledgment from B's TCP with its first and determine how many bytes have been successfully sent (modulo 2^{32}; if more than 2^{32} bytes might be sent, the application should also preserve how many bytes it believes it has sent, to disambiguate). The application can then resume sending its output from that point, and expect its input to resume from its checkpointed acknowledgment. When finished, it sends `RTCP_SHUTDOWN` and `closes` the socket as normal.

Like the `shutdown` socket function, with its separate `SHUT_RD` and `SHUT_WR`, RTCP has commands enabling the two halves of a connection to be closed separately: `RTCP_SHUTDOWN_READING` and `RTCP_SHUTDOWN_WRITING`. The former indicates that the application does not expect to read any more data, and tells RTCP to stop checkpointing acknowledgments (permitting A's TCP to acknowledge B's FIN). The latter indicates that the application does not expect to write any more data, and tells RTCP that its future FINs will be genuine. The `RTCP_SHUTDOWN` command is actually just a shortcut for accomplishing both commands simultaneously, analogous to `SHUT_RDWR` or `close`.

The changes to a fault-tolerant application are slight. For our evaluation, we modified `netperf` to support RTCP—the original, of course, was not fault-tolerant, and we did not add application-level fault tolerance. However, updating one of the tests to use the appropriate RTCP commands, including checkpointed acknowledgments, required about 13 lines of code plus error handling.

5 Evaluation

We evaluated the NetSlices implementation of RTCP, focusing on overhead with respect to unprotected TCP, scalability, and speed of connection recovery. We interposed RTCP on a 10 Gigabit Ethernet link, on a machine with a Myricom Network Interface Card and an 8-core (16-hyperthread) Intel Nehalem CPU. The NIC we used has the ability to map packets to virtualized queues based on port numbers, enabling us to pin connections to particular cores, achieving scalability by running an RTCP on each core to share the load. The actual path from NIC to RTCP is NetSlices. The nodes running applications and peers were less powerful, with only 1 Gigabit Ethernet cards, so several on each side were required to saturate the 10 Gigabit link. Ping latency from one of the A nodes to RTCP was about 119 microseconds, and ping latency through to one of the B nodes was about 299 microseconds.

We used `netperf`, a common network benchmarking tool, to measure three configurations: unprotected TCP, RTCP, and "dummy" RTCP. The dummy RTCP does all of the NetSlices filtering of packets, but never actually calls in to `librtcp`; thus, it enables us to

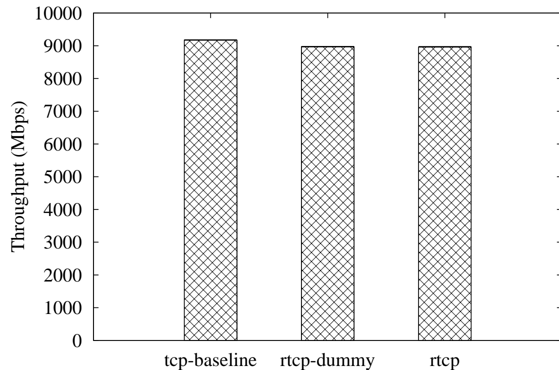

Figure 4: TCP_STREAM throughput. RTCP achieves 97.71% of the throughput of unprotected TCP saturating a 10 Gigabit Ethernet link.

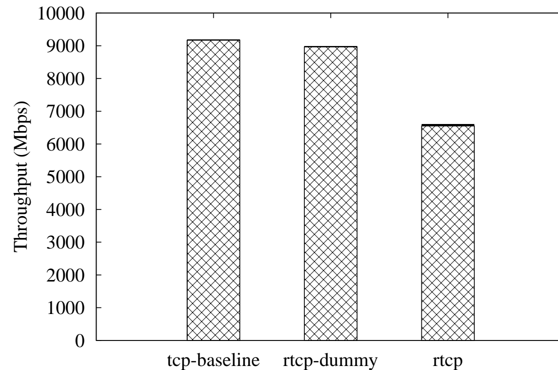

Figure 5: TCP_MAERTS throughput. With B sending data to A, thus stressing checkpointed acknowledgments, RTCP achieves 71.4% of the throughput of unprotected TCP. Careful choice of window size proves crucial in this case, because the effect of checkpointed acknowledgments is to artificially increase B's perceived round-trip time and thus the bandwidth–delay product of the connection.

separate overhead due to filtering from overhead due to RTCP processing. Under RTCP, `netperf` required minor additions to send appropriate RTCP control packets. We subjected the various configurations to three `netperf` benchmarks: TCP_STREAM, which measures throughput from A to B, TCP_MAERTS ("stream" spelled backwards), which measures throughput from B to A, and TCP_RR ("Request/Response"), which measures how quickly A and B can bounce a 1-byte message back and forth.

The results of the TCP_STREAM test are shown in Figure 4. Due to overheads at all layers from symbol encoding to TCP headers, it is not possible to achieve the full 10 Gigabits in application throughput; however, by carefully choosing TCP window parameters, unprotected TCP was able to saturate the link. The dummy RTCP achieved 97.77% of the throughput of unprotected TCP, whereas full RTCP achieved 97.71%. Thus, of the overhead introduced, most is due to packet filtering; even with packet filtering and full protection, RTCP is able to operate at line rate on a 10 Gigabit link.

Because of RTCP's excellent scalability in packet processing, it is not relevant to measure its scalability in terms of the number of protected connections that actively send data—in the absence of failure. The connections can not send faster than the link, so the only overhead related to their number is the cost of RTCP looking up state for each packet. In the experimental deployment, RTCP uses an array indexed by port number for quick, constant-time lookup. If a different deployment were to need a non-constant-time data structure for connection state, its evaluation should consider the overhead of such a data structure.

In the presence of failure, however, especially RTCP failure, the number of connections is quite relevant. Because RTCP failure induces failure in all the connections flowing through it, the extra control traffic generated by all recovering at once might be very different from their usual behavior; more precisely characterizing scalability in simultaneous failures is left for future work, because it is not immediately relevant to the relatively few applications RTCP will protect within the router project.

The TCP_STREAM test does not exercise checkpointed acknowledgments at all, because B never sends data back to A to be acknowledged. On the other hand, the TCP_MAERTS test—although *prima facie* identical to TCP_STREAM save for direction—primarily demonstrates the throughput impact of checkpointed acknowledgments. The results are shown in Figure 5. RTCP achieves 71.4% of the throughput of unprotected TCP. Our first numbers for this metric were quite bad, more like 20%; the reason for the discrepancy provides a key insight into the impact of checkpointed acknowledgments on throughput. Specifically, it artificially increases the round-trip-time B's TCP computes based on the acknowledgments, and thus the bandwidth–delay product of the connection from its perspective. Thus, correctly choosing a large enough TCP window is crucial. Additionally, changing the priority of the RTCP process on the filtering node dramatically increased performance, showing that even operating system scheduling enters the critical path in getting acknowledgments out. The longer it takes A to checkpoint its data, the more the connection appears (to B's TCP) to be high-bandwidth but high-latency.

Another performance impact of checkpointed acknowledgments is the bandwidth spent on sending the control messages themselves. We note that some prior

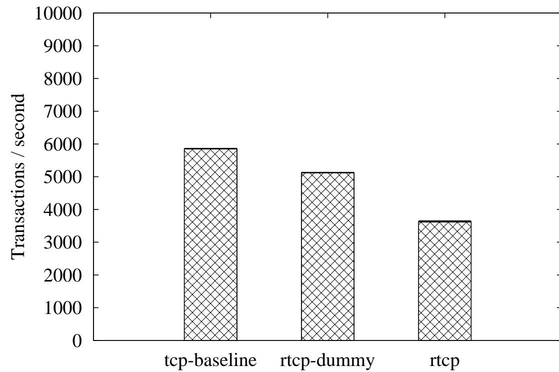

Figure 6: TCP_RR application round-trips per second. With A and B bouncing a 1-byte message back and forth, which both stresses checkpointed acknowledgment and strips away any pipelining to reveal all of RTCP's latency overhead—that is to say, in the most challenging common case test—RTCP achieves 61.9% of the application-level round trips per second of unprotected TCP.

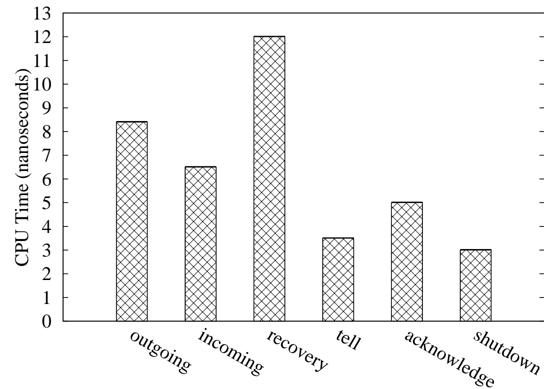

Figure 7: CPU microbenchmark. Microbenchmarking RTCP processing various kinds of packets, without any network or copying overhead, shows the cost of packet processing; in terms of CPU time, one instance of RTCP can handle about thirty million data packets per second.

work [13] specifically explored sending control information over dedicated links at various rates, with the result being greatly improved throughput. Another way to eliminate that cost would be to colocate RTCP and the fault-tolerant applications running on a particular node, as mentioned previously for security; thus, control messages could use the fast loopback link, avoiding congestion on the real link and also reducing latency in communication on the side channel.

When testing raw throughput, TCP can take advantage of large windows to pipeline a great deal of data and hide the actual latency of the link. The TCP_RR Request/Response test, on the other hand, ping-pongs a 1-byte message back and forth from A to B, and thus completely eliminates the effect of pipelining to reveal latency overheads. Additionally, unlike the other `netperf` tests we performed, TCP_RR is bidirectional, with both incoming and outgoing data; checkpointed acknowledgments are once again relevant. The result of TCP_RR, unlike the throughput measurements, is in operations (Request/Response application-level round trips) per second, as shown in Figure 6. The Request/Response test stresses RTCP the most of any common case test, because the alternating message eliminates TCP's usual advantage of pipelining the data in flight, stripping the result down to pure latency, and because half of the messages are subject to delayed acknowledgments. Nonetheless, RTCP achieves 61.9% of unprotected TCP's application-level round-trips per second.

To get a more detailed view of RTCP's performance and latency, we microbenchmarked the CPU time—independent of the network and filtering—that RTCP spends processing various kinds of packets. The microbenchmark program constructs the packets in memory, then calls the relevant `librtcp` processing function in a tight loop a large number of times, while measuring userspace CPU time (RTCP makes no library calls, let alone system calls, so system CPU time is always zero). All of the packets considered are idempotent, and repeated processing follows the same code path each time. In order to ensure that, even with rewriting, RTCP actually processes the same input each time around the loop, the microbenchmark makes a fresh copy each time; the time spent copying was microbenchmarked separately and subtracted out of the results. As a consequence of that and the cache-friendly behavior of processing the same buffer over and over, the microbenchmarks show an ideal picture of RTCP's performance from when it gets a packet in memory until it is ready to be dropped or forwarded.

The microbenchmark covers six kinds of packets:

- An empty incoming acknowledgment packet.
- An empty outgoing acknowledgment packet.
- A recovery packet, that is, an outgoing SYN on an already-synchronized connection.
- An `RTCP_TELL` command.
- An `RTCP_ACKNOWLEDGE` command.
- An `RTCP_SHUTDOWN` command.

The time spent on each is shown in Figure 7; all are on the order of nanoseconds. More processing is done on actual TCP packets than on control UDP packets. Within each, the most expensive kind of packet—recovery packets for TCP, and `RTCP_ACKNOWLEDGE` packets for UDP—are the ones that require the most

rewriting; recovery SYN packets are rewritten into answering SYN-ACKS, and RTCP_ACKNOWLEDGE packets are rewritten into TCP acknowledgments. Outgoing TCP packets require slightly more processing than incoming, because masking failure and enabling recovery are all aspects of processing A's output.

If we assume that failure is rare, the most important feature of RTCP is its unobtrusiveness in the common case, as measured above. However, if failure never occurred, RTCP would not need to exist; it is also crucial that when an application fails, it can pick up its connections quickly. Therefore, we concluded our experimentation with an application-level recovery benchmark. We ignore benchmarking recovery of the RTCP filter itself, because in the current implementation that cost is dominated by A waiting to retransmit its unacknowledged SYN, which has nothing to do with RTCP's performance. The connection recovery benchmark establishes a connection, then repeatedly "crashes" by simply calling `close` and letting RTCP turn the unexpected FIN into a RST. Then it recovers the connection immediately, going through all of the usual steps, including sending an RTCP_TELL and waiting for the response. Proceeding in that manner, the benchmark was able to repeatedly recover in an average of 378 microseconds per cycle, or about three round-trips (FIN into RST to fail, SYN into SYN-ACK to recover, and RTCP_TELL and response to learn new state) to RTCP.

6 Related Work and Discussion

A variety of projects have addressed the problem of a service recovering from a failure transparent to its TCP-based clients. One approach is to introduce a modified TCP socket library for clients [9]. Such a library could automatically set up a new connection to a new server in case the connection to the current one fails. However, modifying the clients is not usually an option, and would severely complicate its deployment.

Another possibility is to replicate the server-side TCP state onto multiple machines. The state of the TCP connection is replicated along with the state of the application. For example, HydraNet-FT [11], HotSwap [2], and ST-TCP [5, 6] use a primary-backup replication approach. Disadvantages of this approach include the high overhead of replication (incoming packets have to be delivered to all replicas) and, usually, the inconvenience of having to modify the server-side TCP stack in order to access its state.

An approach that leaves the TCP stack unchanged is to wrap the server-side TCP stack, recording and possibly modifying its incoming and outgoing events. Failover TCP [3] and FT-TCP [1, 12, 13] use this technique. Similar to our RTCP approach, acknowledgments are delayed until the data is safely handled and sequence numbers in the packets to and from a restarted TCP stack have to be rewritten. This approach suffers from high overhead due to the process of recording events, and although it leaves the TCP stack unchanged, it does require kernel modifications in the server operating systems.

To avoid such kernel modifications, yet another approach is to use a proxy server between the client and the server replicas [7]. The client sets up a connection to a proxy server, and the proxy server handles failing over from a primary server to a backup server as necessary. This approach has a single point of failure: if the proxy fails, the TCP connection to the client is interrupted. A solution is to replicate the proxy [8], but this suffers from many of the same disadvantages of TCP stack replication.

Our approach also uses a proxy technique, but our insight is that there is no need to replicate its state—all its state is "soft" and can be rebuilt upon recovery by simulating a server failure and recovery. Its use of checkpointed acknowledgment makes it unnecessary to log data, and consequently the RTCP state per connection is very small and the solution is highly scalable.

The decision to eliminate replication represents a tradeoff: Replication introduces additional network hops and processing costs in the common case, but causing every active connection to simultaneously fail introduces more processing and delay to recover in the case of failure. Thus, the choice depends on how often RTCP is expected to fail and the maximum delay that can be tolerated.

There is a third way, however, which is planned for future development. The reason RTCP failure induces connection failures is to reduce the problem to connection recovery, which we have already solved. Instead, the side channel could be extended to enable RTCP to make asynchronous requests back to the applications, indicating when recovery is needed, with explicit responses to rebuild the state without inducing failure. Adding that ability would complicate RTCP's configuration, in the form of `librtcp`'s API and its assumptions about the packet filtering driver, but would remove the need for inducing failure and enable faster RTCP recovery.

Finally, one of the costs of recovering connections, when RTCP recovers and induces failure, is waiting for A's TCP to retransmit its SYN. However, B's ACK response to the first gives RTCP enough information to respond immediately; rewriting it instead of waiting for retransmission would dramatically decrease recovery times without the need for a reverse side channel or internal replication.

7 Conclusion

RTCP is a new packet filter that enables the recovery of TCP connections that would otherwise break as the result of the failure of an endpoint, even a fault-tolerant endpoint, due to the information hidden in the network stack. Aimed at high availability implementations of services such as BGP on core Internet routers, RTCP is able to run at 97% of the throughput of unprotected TCP on a 10 Gigabit Ethernet link, thus achieving the desired protection with little impact on connection performance without failure, while also enabling connection recovery times on the order of 378 microseconds. Thus, the emphasis on simplicity shows in its performance.

Making RTCP a man-in-the-middle—thereby moving TCP recovery into the network—achieves independence from operating system and network stack support, which, in turn, significantly pares down the state required to track a connection. Employing checkpointed acknowledgments avoids all buffering within RTCP. Additionally, getting applications directly involved in protecting their own connections, aside from further simplifying RTCP, eliminates the need to make assumptions about application behavior such as determinism.

In future work, we will explore TCP security features such as the MD5 security header used as a TCP option by BGP, and will install and benchmark RTCP on a fully functional core Internet router; such an effort is part of a larger endeavor to dramatically improve the reliability of the Internet.

Availability

RTCP is free software under the New BSD License. Source code and information are available at

`http://rtcp.sourceforge.net/`

References

[1] ALVISI, L., BRESSOUD, T., EL-KHASHAB, A., MARZULLO, K., AND ZAGORODNOV, D. Wrapping server-side TCP to mask connection failures. In *Proc. of Infocom 2001* (Anchorage, Alaska, Apr. 2001), pp. 329–338.

[2] BURTON-KRAHN, N. HotSwap – transparent server failover for Linux. In *Proc. of USENIX LISA 02: Sixteenth Systems Administration Conference* (2002).

[3] KOCH, R., HORTIKAR, S., MOSER, L., AND MELLIAR-SMITH, P. Transparent TCP connection failover. In *Proc. of the International Conference on Dependable Systems and Networks (DSN'03)* (2003), IEEE Computer Society, pp. 383–392.

[4] MARIAN, T. *Operating Systems Abstractions for Software Packet Processing in Datacenters*. Phd dissertation, Cornell University, Department of Computer Science, August 2010.

[5] MARWAH, M., MISHRA, S., AND FETZER, C. TCP server fault tolerance using connection migration to a backup server. In *Proc. of the International Conference on Dependable Systems and Networks (DSN'03)* (San Francisco, CA, June 2003), IEEE Computer Society.

[6] MARWAH, M., MISHRA, S., AND FETZER, C. A system demonstration of ST-TCP. In *Proc. of the 2005 IEEE International Conference on Dependable Systems and Networks (DSN 2005)* (Yokohama, Japan, June 2005), IEEE Computer Society.

[7] MARWAH, M., MISHRA, S., AND FETZER, C. Fault-tolerant and scalable TCP splice and web server architecture. In *Proc. of the 25th Symposium on Reliability in Distributed Software (SRDS'06)* (2006), IEEE Computer Society, pp. 301–310.

[8] MARWAH, M., MISHRA, S., AND FETZER, C. Enhanced server fault-tolerance for improved user experience. In *Proc. of the International Conference on Dependable Systems and Networks (DSN'08)* (2008), IEEE Computer Society, pp. 167–176.

[9] ORGIYAN, M., AND FETZER, C. Tapping TCP streams. In *Proc. of the International Symposium on Network Computing and Applications (NCA'01)* (2001), IEEE Computer Society, pp. 278–289.

[10] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-end arguments in system design. In *ICDCS* (1981), IEEE Computer Society, pp. 509–512.

[11] SHENOY, G., SATAPATI, S. K., AND BETTATI, R. Hydranet-ft: Network support for dependable services. In *Proc. of the International Conference on Distributed Computing Systems (ICDCS'00)* (2000), pp. 699–706.

[12] ZAGORODNOV, D., MARZULLO, K., ALVISI, L., AND BRESSOUD, T. Engineering fault-tolerant TCP/IP servers using FT-TCP. In *Proc. of the International Conference on Dependable Systems and Networks (DSN'03)* (Los Alamitos, CA, USA, 2003), IEEE Computer Society.

[13] ZAGORODNOV, D., MARZULLO, K., ALVISI, L., AND BRESSOUD, T. Practical and low-overhead masking of failures of TCP-based servers. *Transactions on Computer Systems 27*, 2 (2009).