

# Application-Driven TCP Recovery and Non-Stop BGP

Robert Surton, Ken Birman, and Robbert van Renesse  
Department of Computer Science, Cornell University

**Abstract**—Some network protocols tie application state to underlying TCP connections, leading to unacceptable service outages when an endpoint loses TCP state during fail-over or migration. For example, BGP ties forwarding tables to its control plane connections so that the failure of a BGP endpoint can lead to widespread routing disruption, even if it recovers all of its state but what was encapsulated by its TCP implementation. Although techniques exist for recovering TCP state transparently, they make assumptions that do not hold for applications such as BGP. We introduce *application-driven TCP recovery*, a technique that separates application recovery from TCP recovery. We evaluate our prototype, TCP-R, and show that it outperforms existing BGP recovery techniques.

**Keywords**—TCP, fault tolerance, middleware, BGP, non-stop routing, Graceful Restart.

## I. INTRODUCTION

The state that defines a TCP connection is generally encapsulated within a network stack, while the usual socket interface provides no mechanism for an application to checkpoint, recover, or migrate that state. Unfortunately for an application such as HTTP or BGP, written standards and legacy peers prevent using a session layer to decouple application state from connection state, so the network stack is a point of vulnerability for the entire system.

In HTTP, for example, a client might submit a request to a distributed web application in the cloud. If the request modifies application state, but the connection is reset, the client can neither assume that the request was processed nor safely resubmit it.

BGP is the protocol used by gateway routers to stitch the Internet together out of smaller autonomous networks. Each pair of neighboring routers maintains a TCP connection, which they use to keep their routing tables in sync. between the BGP routers that interconnect networks. If a router detects that one of its connections has failed, it assumes that its link to that neighbor has also failed, so it immediately withdraws all of its affected routes—even if the neighbor immediately reconnects. And once it does so, the new TCP connection cannot be used to carry network updates or inform routing decisions until each endpoint has transmitted its entire routing table. Meanwhile, as network routing reconverges to accommodate the supposed failure, which can take minutes, packet loss can increase 30-fold due to transient routing loops and black holes [1], [2], [3], [4].

This paper introduces a new technique for tolerating connection failure, in the face of a network stack with no interface to enable it, and legacy peers that preclude graceful reconnection. We focus on BGP, because it challenges many

assumptions of prior work in the area, but our technique is applicable to any use of TCP.

To understand how prior work could be effective for HTTP but not for BGP, it is worth distinguishing between *fault masking* and *recovery* [5]. Consider a technique in which a backup network stack is kept synchronized with the primary, used by the application. When the primary fails, the backup replaces it. The fault has been masked, because the peer need not be aware of the failover from primary to backup. However, recovery has not taken place, because there is no longer a backup, and a future fault will cause a failure.

Prior work handles recovery by starting a fresh copy of the application, then replaying all of the peer's input since the connection was established. (Depending on how the particular system deals with the network stack, the replay might be at the application level or the packet level.) Under the assumption that the application is fault-tolerant, the result is a valid backup replica.

The benefit of the *replay-driven* approach to recovery is that the application can be treated as a black box, requiring no modifications. However, not only does the approach assume a deterministic application, it also assumes that replay is practical. The input to an HTTP server is usually a short request. On the other hand, BGP connections persist for the lifetime of a peering between two routers, and often carry thousands of update messages per minute. Replaying such a connection quickly becomes more burdensome than failure.

We introduce an *application-driven* approach. Our prototype, TCP-R, is network middleware not unlike a NAT box, which sends state gleaned from packets to the application, and obeys the application to manipulate packets for recovery. By accepting minor changes to the application, application-driven recovery avoids the much greater burden of accepting responsibility for the application's state. Furthermore, by giving the application responsibility for checkpointing its connection state along with its other state, we achieve a simple form of application-driven recovery that is both lightweight and easy to implement.

In addition to connection recovery, we also evaluate a standardized, BGP-specific approach to recovery called Graceful Restart (GR). Because there are many reasons a connection might close other than link failure, a router can enable GR for a connection, informing its peer not to trigger reconvergence immediately when the connection fails. Instead, the peer will wait until a new connection is resynchronized and the link is known to be down [6]. The time and update load to resynchronize the connection is still necessary, but the routing disruption is avoided.

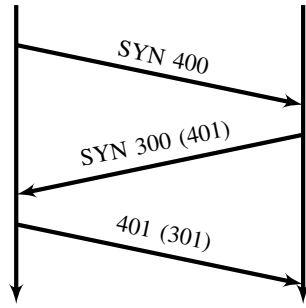


Fig. 1: A TCP connection begins with a SYN, SYN-ACK, ACK handshake that establishes each endpoint’s initial sequence number.

Unfortunately, the assumption that the link survives can be just as wrong as the assumption that the link fails. We introduce a new approach to BGP analysis and use it to demonstrate the inadequacy of GR, concluding that only true connection recovery is sufficient to protect the stability of the Internet.

## II. DESIGN

We assume that we are given a fault-tolerant application that can recover its own state, for example using checkpoints, but depends on TCP connections that have inaccessible state within a network stack. We will consistently refer to the fault-tolerant application as the *application*, and to the remote endpoints of its connections as its *peers*. We assume nothing of the network stack, the application’s interface to it, or the peers, other than what is specified by TCP [7].

A TCP connection consists of two independent streams of bytes, one from the application to the peer and one in the opposite direction. A TCP stream is reliable, in the sense that each byte is delivered exactly once, in order. To that end, each packet bears both a sequence number, indicating the position of its first data byte within its stream, and an acknowledgment, indicating the next sequence number expected in the opposite direction. An endpoint buffers each byte it sends, retransmitting it as necessary, until it receives an acknowledgment with a later sequence number.

When a connection is established, each endpoint chooses an initial sequence number at random, in order to avoid confusion with packets that might still be in the network from a previous connection between the same addresses. A packet with the SYN flag establishes a new stream and sets the initial sequence number; for example, after the handshake in Figure 1, the client’s first data byte will have sequence number 401 (a SYN counts as a byte sent), and the server’s first data byte will have sequence number 301. A packet too far in advance of the expected sequence, or which bears a SYN flag even though the connection has already been established, will be considered “unacceptable”.

TCP uses unacceptable packets to drive a non-transparent form of connection recovery. If an endpoint with an established connection receives an unacceptable packet, it replies with a control packet, indicating the sequence number and acknowledgment it thinks are current. However, if the connection is

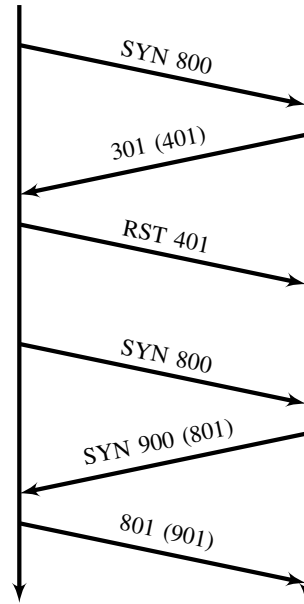


Fig. 2: Continuing after Figure 1, the client fails and reconnects. Its SYN is unacceptable, so the server replies with an empty packet. The reply is in turn unacceptable to the client, which does not yet have a connection, so the client sends a RST and the server deletes the state of the old connection. When the client retransmits its SYN, they establish a new connection.

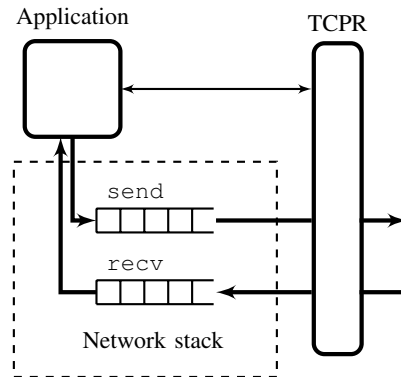


Fig. 3: TCPR is a packet filter interposed between the application’s network stack and peers, which allows the application to initiate connection recovery in a manner transparent to the remote end-point.

not established, the endpoint replies by acknowledging the unacceptable sequence number with the RST flag, notifying the remote endpoint and causing it to abort its connection. If a recovering client attempts to reconnect, the connection will recover as shown in Figure 2. The outcome is that the old connection is aborted, and the two endpoints establish a new one.

We build on TCP’s notion of recovery, and make it transparent by interposing middleware between the application’s network stack and peers, as shown in Figure 3. The middlebox, TCPR, communicates with the application both implicitly,

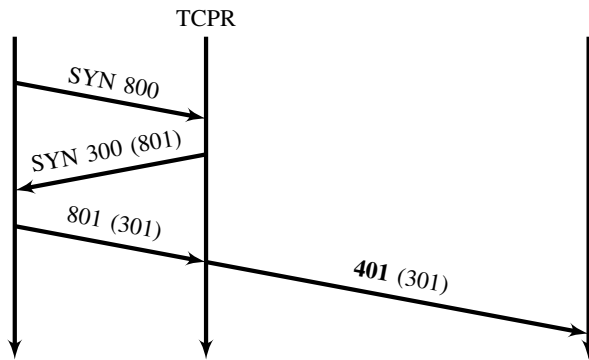


Fig. 4: TCPR tracks acknowledgments, so that it can immediately answer a recovery SYN. If the client from Figure 2 were an application using TCPR, TCPR would establish a new connection locally and splice it back to the original, so neither network stack is aware of the recovery.

through the behavior of the network stack, and directly, through a side channel. TCPR maintains state for each connection, of which the application also maintains a copy; the side channel exists to synchronize those copies of the connection state.

From TCPR’s perspective, an application is just a side channel and a set of connections whose packets can be manipulated; without loss of generality, we will discuss how TCPR enables a single process to protect a single connection. Our goal for TCPR is to enable recovery with the simplest possible middlebox and the least burden on the application, in terms of code modification, per-connection state, communication on the side channel, and overhead versus unprotected TCP.

#### A. Resynchronizing

As before, the application signals its desire to recover when its network stack sends a SYN in the middle of an established connection. Rather than revealing recovery to the peer, TCPR intercepts the SYN and establishes a new connection locally, as shown in Figure 4.

As with any new connection, the application’s network stack chooses an initial sequence number in a manner deliberately designed to be unacceptable to old peers. In order to splice the new and old connections back together, TCPR’s per-connection state includes the application’s and peer’s acknowledgments, *ack* and *peer\_ack* respectively.

By definition, the peer expects that *peer\_ack* will be the next sequence number it receives. Suppose the new connection begins from some other value, *seq*. TCPR computes  $\Delta = seq - peer\_ack$ , and for the life of the new connection, subtracts  $\Delta$  from the application’s sequence numbers, and adds  $\Delta$  to the peer’s acknowledgments. Thus, translation occurs through small header modifications on packets in flight, much as a network address translator remaps addresses and ports.

The opposite stream is simpler to resynchronize. TCPR chooses its initial sequence number as *ack* - 1 (recall that the SYN flag implicitly occupies a byte), so that the new connection expects the peer to continue from *ack* just as before.

However, TCP’s reliability depends not only on sequence numbers, but on the send and receive buffers at each endpoint, which enable data to be retransmitted until it is safe at the remote endpoint. If the application migrates to a new machine or its network stack loses its state, the lost buffers must be recovered.

#### B. Recovering the Send Buffer

When an application calls *send*, success only indicates that the argument has been copied into the send buffer in the network stack. Should the send buffer be lost, some of its contents might not yet have been sent, or might be dropped in the network. Outside the network stack, only the application itself knows what it intended to send, so TCPR depends on it to replay what might be lost.

The application can use TCPR to learn how much of its output is safe at any time, and this information is necessary after resynchronization in order to know what to re-*send*. To do so, the application requests the latest state from TCPR, which includes *peer\_ack*. To translate from sequence numbers to total bytes sent since an arbitrary checkpoint, the application can simply subtract the value of *peer\_ack* from that checkpoint.

Note that unacknowledged data is not necessarily lost. Some might have been delivered to the peer, although no acknowledgment had arrived by the moment at which the network stack lost its state. Thus, the application must repeat whatever it sent the first time. Generating all output deterministically is sufficient, but isn’t necessary. For example, it is also sufficient for the application to checkpoint any data it is preparing to send, so that it recovers not only the old value of *peer\_ack*, but at least enough buffered data to recover. Beyond that point, the application’s output is unconstrained. On the other hand, a deterministic application does not need a buffer. TCPR enables the most efficient choice based on specialized knowledge about each connection.

#### C. Recovering the Receive Buffer

When data arrives from the peer, the application’s network stack buffers it until the application consumes it using *recv*. The TCP standard considers such data safe to acknowledge immediately; once acknowledged, the remote peer will remove it from its send buffer. However, the application might not yet have invoked *recv* and obtained the data, let alone checkpointed it. Accordingly, TCPR intercepts and modifies TCP acknowledgments to ensure that unsafe data will not be acknowledged, relying on the recoverable application to tell TCPR when received data is safely checkpointed. Delayed acknowledgments were introduced with FT-TCP [8], which acknowledges packets only after it has checkpointed them into a “stable buffer”, all hidden from the application. Putting the application in charge enables more flexibility; for example, the application could checkpoint raw input immediately, or process whole application-layer messages and checkpoint the resulting state changes—if some input causes no significant state changes, the application could acknowledge it without waiting for a checkpoint at all.

Delaying an acknowledgment can inflate the peer’s estimate of the round-trip time of the connection. However most TCP implementations already delay acknowledgments by up to

500 ms to conserve bandwidth and prevent “silly window syndrome” [9]. Zagorodnov et al. [8] evaluated a variety of strategies for generating delayed acknowledgments from an advancing checkpoint; TCPR uses the strategy they call “Delayed”, which provides the best throughput for the fewest packets.

By putting the application itself in charge of its acknowledgments, TCPR lifts the end-to-end argument [10] for TCP’s reliability from the host level to the application level.

#### D. Handling Options

T CPR supports the most common TCP options. The TCP standard leaves up to 20 bytes in the TCP header for such options, and specifies three that all implementations must support (many more have subsequently been proposed). The three standard options are No-Operation, End of Option List, and Maximum Segment Size. The first two are used to manipulate padding, so they have no impact on a connection’s state. An endpoint may advertise Maximum Segment Size with its SYN packet to negotiate a packet size that avoids IP fragmentation. TCPR simply passes the value through, and records it to ensure that the same negotiation takes place during recovery.

RFC 1323 [11] describes the first additional options to be defined, Window Scaling and Timestamps, with the goal of supporting high-latency high-bandwidth networks. To support Window Scaling and Timestamps, TCPR passes the parameters through and saves them for recovery, just as it does with Maximum Segment Size. The authors of RFC 1323 noted that the only previously non-padding option, Maximum Segment Size, was only sent on SYN packets, so they worried that buggy TCP implementations might erroneously fail to handle unknown options on normal traffic. To address that concern, they established the convention that TCP options are negotiated in the handshake or else disabled. The result for TCPR is that suppressing unknown options on a handshake will generally avoid the need to suppress them any further.

T CPR also supports the Selective Acknowledgments [12] option, which enables an endpoint to acknowledge data that it receives out-of-order or with gaps. Advancing the actual acknowledgment would erroneously cover the gaps, but failing to acknowledge the received data might force the remote endpoint to wastefully retransmit data that wasn’t really lost. At first glance, Selective Acknowledgments might seem incompatible with TCPR’s delayed acknowledgments. However, the standard specifies that Selective Acknowledgments are purely advisory: although they serve as notification, the peer is still responsible for eventually retransmitting that data if the cumulative acknowledgment never catches up. Thus, it suffices for TCPR to apply  $\Delta$  to the peer’s selective acknowledgments as well as to its ACKs.

#### E. Masking Failure

T CPR cannot trust the network stack to accurately distinguish between the application closing a connection and failure. During failover or migration, the application has (conceptually if not in fact) two unsynchronized network stacks—the new one, in which the connection is not yet established, and the old one, in which the connection is no longer established.

Until the new network stack is resynchronized, any packet it receives will be unacceptable, and it will send a RST as discussed in Figure 2. An endpoint with an established connection never sends a RST, so TCPR drops it to prevent what we term a *Romeo and Juliet* scenario: if the peer received the notice that the application is dead, it would abort the connection just as the application came back to life.

Failure can also be revealed to the peer if the old network stack tries to clean up by closing the connection, such as when only the application process fails.

An endpoint closes its output by sending a packet with the FIN flag, which occupies a byte at the end of the stream and must be acknowledged by the remote endpoint, like the SYN at the beginning. At the packet level, there is no way to distinguish whether a FIN indicates failure or a deliberate call to `close`. Prior approaches have interposed on the network stack’s interface to learn when the application closes a connection deliberately. In TCPR we adopt a more explicit approach: the application sets a flag, *done\_writing*, just before it closes. TCPR treats a FIN as spurious if and only if that flag is clear.

T CPR responds to a spurious FIN with a RST, in order to enable the application to recover quickly in the case where the old and new network stacks are the same. The network stack will abort the old connection and be ready immediately to establish the new connection.

#### F. Closing Input

To deliberately close the stream in the other direction, the application sets another flag, *done\_reading*. When *done\_reading* is set, TCPR does not delay acknowledgments. The network stack is free to acknowledge any remaining data, notably including the peer’s FIN, so the peer can close gracefully.

If the FIN is the only byte remaining to be acknowledged, the application could instead advance *ack* by one byte. TCPR provides *done\_reading* to echo the behavior of `shutdown`, and it also enables us to perform experiments without delayed acknowledgments.

#### G. Recovering After Closing

As with `send`, a successful `close` indicates only that the FIN is in the send buffer. It might take some time for all of the data to be sent and acknowledged. Even once the final acknowledgment is sent, TCP implementations wait, usually 120 seconds, to be sure that the acknowledgment arrives and to handle any straggling packets. What if the application crashes after abdicating its socket?

T CPR sets a flag, *done*, when it thinks both flows are closed, and another, *failed*, when it has detected that the network stack has failed. The application can check at any time whether the connection is really closed on the wire. If necessary, the application can recover as normal; during recovery, TCPR always unsets *done\_writing* to give the application the chance to call `close` again.

Once *close* has been set for an appropriate duration (such as 120 seconds) the application explicitly instructs TCPR to

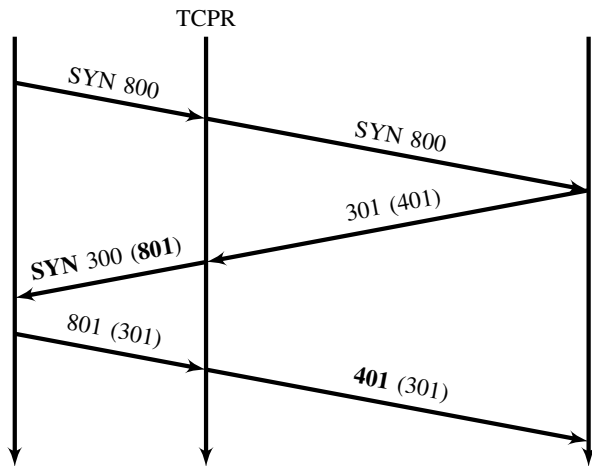


Fig. 5: When TCPR fails over, it depends on the application for some of its state, and can recover the rest from the peer itself. For example, if TCPR cannot immediately answer a recovery SYN as in Figure 4, sending the unacceptable packet to the peer prompts it to fill in the missing state.

delete its state. Of course, there is no reason the application has to wait, and being in control of when TCPR deletes its state also makes it easy to experimentally inject failures.

#### H. Recovering TCPR

Should TCPR itself fail, it can recover some of its state, such as *peer\_ack*, by observing packets on the wire. The remaining state, such as the latest *ack* for delaying acknowledgments, is provided by the application. TCPR avoids the need to replicate any of its own state because of our assumption that recovery is driven by a fault-tolerant application.

For example, if the application is trying to recover but the latest *peer\_ack* is missing, TCPR delivers the recovery SYN to the peer uncorrected; the peer’s answer reveals the desired value. See Figure 5 for an example. Resynchronization takes place as in Figure 4, but with an additional round-trip to recover soft state from the peer.

That default behavior also avoids the need for a special case to detect whether a connection is new or recovering. If *peer\_ack* is missing, either TCPR crashed and lost it, or the connection is new and it doesn’t exist yet; in the former case, the peer provides the missing value, and in the latter, it sends its own valid answer to the handshake.

On the other hand, fields such as *ack* and *done\_writing* cannot be inferred from packet-level observation, because they are inherently controlled by the application. Neither can fields such as the saved values of options, which are advertised only once. Thus the application is expected to reset these values through the side channel it has with TCPR.

The state that depends on the application changes much more slowly than the soft state recoverable from packets. Whereas *peer\_ack* is updated with every packet from the peer, all of the application-dependent state is either fixed at connection establishment (such as the peer’s TCP options),

```

struct tcpr_hard {
    uint16_t port;
    struct {
        uint16_t port;
        uint16_t mss;
        uint8_t ws;
        uint8_t sack_permitted;
    } peer;
    uint32_t ack;
    uint8_t done_reading;
    uint8_t done_writing;
};

struct tcpr {
    struct tcpr_hard hard;
    uint32_t delta;
    uint32_t ack;
    uint32_t fin;
    uint32_t seq;
    uint16_t win;
    uint16_t port;
    struct {
        uint32_t ack;
        uint32_t fin;
        uint16_t win;
        uint8_t have_fin;
        uint8_t have_ack;
    } peer;
    uint8_t have_fin;
    uint8_t done;
    uint8_t failed;
    uint8_t syn_sent;
};

struct tcpr_ip4 {
    uint32_t address;
    uint32_t peer_address;
    struct tcpr tcpr;
};

```

Fig. 6: TCPR state structures. The application keeps one struct *tcpr\_ip4* for each TCP/IP connection, but only the 14-byte portion defined by struct *tcpr\_hard* is necessary for recovery.

set occasionally by the application itself (*ack*), or set by the application once when the connection closes (*done\_reading* and *done\_writing*). Thus, both the extent of the modifications to the application code and the communication overhead of maintaining TCPR’s copy of the state are minimal.

### III. IMPLEMENTATION

The TCP-manipulating core of TCPR is implemented as a portable C library. The simplicity of application-driven recovery is reflected in the fact that the library’s single file contains only about 150 semicolons. We have experimented with TCPR using a variety of techniques to interpose on packets; the current prototype is a loadable module for the Linux kernel firewall, iptables. The system administrator writes rules to match packets to and from the application, delivering them to TCPR rather than dropping or forwarding them.

TCPR’s per-connection state appears in Figure 6. Notably, there is no buffered data. Both TCPR and the application keep exactly one struct *tcpr\_ip4* per connection. The network-independent state is in struct *tcpr*, while only

the subset of the state in `struct tcpr_hard` is crucial for recovery—if any field outside `struct tcpr_hard` is missing, TCPR will recover it on the fly.

The side channel is a UDP connection, and the protocol consists only of entire states sent back and forth. The state is small enough to avoid being a burden to send, and its constant size and the atomic delivery of UDP combine to make the update protocol easy to implement at both ends. For example, to acknowledge some data, the application locally sets `tcpr_hard.ack` and sends the entire state to TCPR.

Using UDP makes it possible for TCPR to be situated on a middlebox physically distinct from the one on which the application is running, but other options are also possible. In our experiments, the highest efficiency was achieved when running TCPR in a separate network namespace but on the same machine as the recoverable application. Network namespaces are a recent Linux kernel feature that enables an individual process to have an isolated routing table, network stack, and set of network interfaces, while sharing the host’s memory, filesystem, processors, and kernel. With the application in its own network namespace, TCPR can run on the host as a middlebox while enjoying loopback-interface throughput and latency.

The TCPR distribution includes a `netcat`-like program and a TCP proxy that provides TCPR-support for unmodified applications, along with a utility to craft UDP TCPR updates on the command line to query TCPR state. To measure recovery time, we have implemented a utility that opens hundreds of connections in parallel, injects failure on each of them, and then times its recovery using Linux’s high-resolution real-time clock. To measure throughput, we have also modified the venerable `ttcp` to support TCPR; including error handling and new command-line options for configuring TCPR, all that it required was the addition of 28 lines.

Modifying an application to use TCPR does not require any changes to existing socket system calls. Instead, one simply adds code to interact with TCPR during connection setup and teardown, and when input is to be checkpointed. A trivial example is shown in Figure 7.

A fault-tolerant application that uses TCPR makes the usual socket calls. After calling `connect` or `accept`—that is, once the connection has been established—the application retrieves the connection’s state from TCPR. As the application consumes its input, it updates `ack` locally, then updates TCPR. Similarly, when there is no more input, it sets `done_reading`, and when it is finished writing output, it sets `done_writing`.

If TCPR itself fails, the application need only send another update message. If the application fails, it need only `bind` and `connect` again to establish a new connection with the same endpoints. Optionally, TCPR can remap the source port to avoid collisions if the original port is already bound on the recovering machine (using `tcpr.port` different from `tcpr_hard.port`). Both TCPR and application recovery are included in the snippet in Figure 8.

#### IV. EVALUATION

We have evaluated application-driven TCP recovery, using BGP as our recoverable application, with the goal of validating

```
s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
bind(s, &addr, addrlen);
listen(s, backlog);
c = accept(s, &peeraddr, &peeraddrlen);
getsockname(c, &addr, &addrlen);

// request connection state
state.address = addr.sin_addr.s_addr;
state.peer_address = peeraddr.sin_addr.s_addr;
state.tcpr_hard.port = addr.sin_port;
state.tcpr_hard.peer.port = peeraddr.sin_port;
write(tcpr, &state, sizeof(state));

// receive TCPR's copy
read(tcpr, &state, sizeof(state));

bytes = read(c, readbuf, readbuflen);

// update delayed acknowledgment
state.tcpr_hard.ack =
    htonl(ntohl(state.tcpr_hard.ack) + bytes);
write(tcpr, &state, sizeof(state));

write(c, writebuf, writebuflen);

// close gracefully
state.tcpr_hard.done_reading = 1;
state.tcpr_hard.done_writing = 1;
write(tcpr, &state, sizeof(state));
close(c);
```

Fig. 7: A simple C server that uses TCPR.

```
// send TCPR the latest state
write(tcpr, &state, sizeof(state));
c = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
bind(c, &addr, addrlen);
connect(c, &peeraddr, peeraddrlen);
```

Fig. 8: A snippet by which the server in Figure 7 recovers from simultaneous TCPR and connection failure.

four key assertions:

- TCPR adds negligible overhead to throughput and latency.
- TCPR enables rapid recovery.
- By contrast, unprotected BGP recovery causes unnecessary and severe outages.
- Graceful Restart (GR) sometimes causes even more disruption than standard BGP recovery.

The TCPR microbenchmarks were conducted between two commodity Linux machines, each with two cores, connected by a 1 Gbps Ethernet link. The BGP fault injection benchmarks were conducted using an experimental framework we implemented, which connects actual software routers and network stacks in a virtual network. All of the nodes run in network namespaces on a single host machine, which we verified was not bottlenecked by CPU. Running on a single machine means that network latencies are negligible, and all of the nodes share a precise wall clock. In all experiments, TCP was implemented by an unmodified Linux kernel network stack.

	Mbps	% Raw
Unprotected	896.354 ± 0.331	100
TCPR	896.385 ± 0.280	100

Fig. 9: TCP throughput from the application to the peer.

	Mbps	% Raw
Unprotected	851.206 ± 4.652	100
Unsafe TCPR	837.275 ± 5.355	98
TCPR	838.699 ± 1.934	98

Fig. 10: TCP throughput from the peer to the application.

### A. Overhead

We evaluated throughput overhead using a version of `ttcp` modified to support TCPR. First, we measured send goodput from the application to the peer both with unprotected TCP and with TCPR, reporting the average and standard deviation of 10 runs in Figure 9. TCPR does not cause any measurable overhead.

Next, we measured in the opposite direction: receive goodput from the peer to the application. This is the flow that is subject to delayed acknowledgments, so in addition to the previous two cases, we measured “Unsafe TCPR”, in which delayed acknowledgments were disabled.

As shown in Figure 10, while there is slight overhead on the incoming packets, there is no measurable impact on throughput from delayed acknowledgments. That is reasonable, because TCP keeps a window of packets in flight in order to mask latency.

The latency impact of delayed acknowledgments can be seen in TCP round-trip times from packet traces. We used traces of throughput experiments like those described above. For each acknowledgment visible to the sender that covered new data, we computed the elapsed time since it had sent that data, reporting the average and standard deviation over all such packets (about 50 in each trace) in Figure 11 and Figure 12.

There is no significant difference in latency from the application to the peer. However, for input from the peer to the application, both setups of TCPR exhibit much higher variance than unprotected TCPR; delayed acknowledgments seem to add nearly 50 microseconds of latency, but the overhead is small and within the standard deviation.

Notice that since the recoverable application is responsible for the delays to its own acknowledgments, these numbers could be arbitrarily large. Our experiments are thus something of a best-case scenario, because we measured an application that always acknowledges its input as soon as possible after the `recv` operation.

### B. Recovery

The delay associated with application recovery obviously depends strongly on the nature of the application, and would often include the latency to detect failure and that associated with launching a replacement. To isolate the costs specifically associated with TCPR, we microbenchmarked its ability to

	Microseconds
Unprotected	318 ± 27
Unsafe TCPR	326 ± 16
TCPR	334 ± 24

Fig. 11: Latency from the application to the peer.

	Microseconds
Unprotected	550 ± 23
Unsafe TCPR	547 ± 94
TCPR	594 ± 85

Fig. 12: Latency from the peer to the application.

recover a connection, measuring the time from when a fresh TCP stack was created by the recovering application until the application is able to `send` and `recv` again.

We measured two cases, shown in Figure 13. In the “Application” case, TCPR retained its state and only the application failed and recovered, so that TCPR could establish a new connection immediately. In the “TCPR + Application” case, they failed and recovered together, so that soft state had to be restored from the peer’s packets during recovery. If the application had saved the soft state as well, recovery would proceed exactly as in the “Application” case.

To set these numbers into context, consider our recoverable BGP scenario: in both cases, TCPR-mediated connection recovery is faster than usual inter-arrival times of BGP updates even on a heavily-loaded core Internet router. Thus, if the BGP failure itself is handled quickly enough by the router, one can completely mask the event from remote BGP peers.

### C. BGP Failover

Our experimental evaluation of BGP failover was heavily influenced by Pei et al. [13], who simulated networks of BGP routers to measure the effectiveness of various proposals at limiting the disruption of link failover. As in this prior work, we evaluate CLIQUE and B-CLIQUE topologies, but rather than examining link failure, we inject router failure followed by immediate recovery. We simulate the network, but run a full BGP implementation on our software routers.

In our experiments, most of the routers run the Quagga routing software and the Quagga implementation of BGP. However, to the best of our knowledge no open source BGP router supports recovering with GR. Accordingly, we added one node running `exabgp`, an easily-configured route injector that supports GR. This node also has a packet filtering ability that we exploit to artificially inflate the path length of some routes, allowing us to explore scenarios with connected-but-undesirable backup paths.

We didn’t modify the routers in any way, except for enabling network namespace virtualization. Each router thus had total control over its own routing table. We recorded the experiments using `rtmon`, a utility that is included with the standard Linux tool `iproute2`. The result is a log of timestamped updates to each routing table. We also used `tcpdump` on the host bridge to record all of the BGP messages sent over the virtual network.

	Microseconds
Application	$39 \pm 8$
TCPR + Application	$167 \pm 25$

Fig. 13: Recovery time.

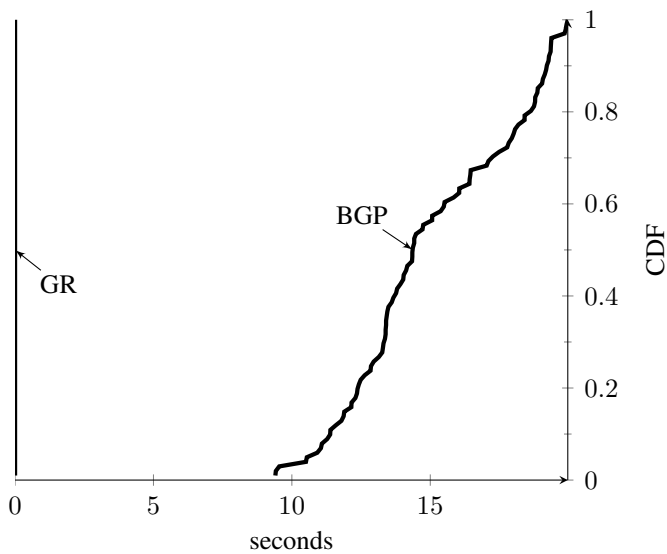


Fig. 14: Duration of convergence due to control failure.

1) *Control Element Failover*: Modern routers are often constructed from a collection of computing nodes. An internal node in a router that runs protocols such as BGP is called a control element, whereas a forwarding element runs the hardware engine and terminates physical links to peers. A common configuration for large routers is to have two redundant control elements, so that when one fails, the other can replace it. As discussed in the introduction, even if BGP failover is rapid, an extended period of disruption to network routing can still ensue. We set out to measure these effects and to see how TCPR can help.

Our experiments start by demonstrating that unmasked failover can cause severe routing disruptions. We experimented with a CLIQUE topology of 16 BGP routers. They were all identical and each peered with all others. One router had an additional peering relation with the route injector, and was used to model a router that must fail-over from a primary to a backup control element. Once the initial convergence completed, we killed the route injector and immediately restarted it, then observed the BGP network until it converged again. We collected data for more than 100 runs.

An important parameter in BGP convergence experiments is the Minimum Route Advertisement Interval (MRAI), which is a rate-limiting knob in BGP. It has been shown that up to a certain value, the MRAI improves convergence times, but past that, convergence times degrade. It is hard to determine the optimal MRAI for a particular topology, and unfeasible for the entire Internet; the original recommendation for MRAI in peerings between providers was 30 seconds, but newer experiments have indicated 5 seconds is better [14]. Our data was collected with an MRAI of 5 seconds. We also

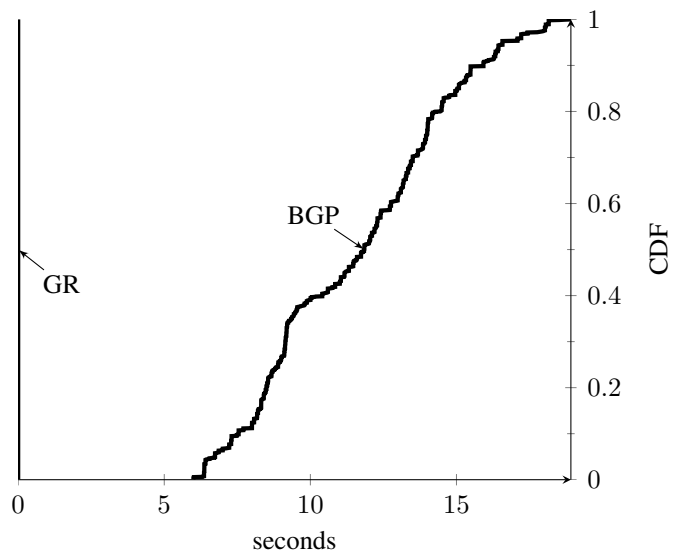


Fig. 15: Average duration a router is disconnected from the destination due to control failure.

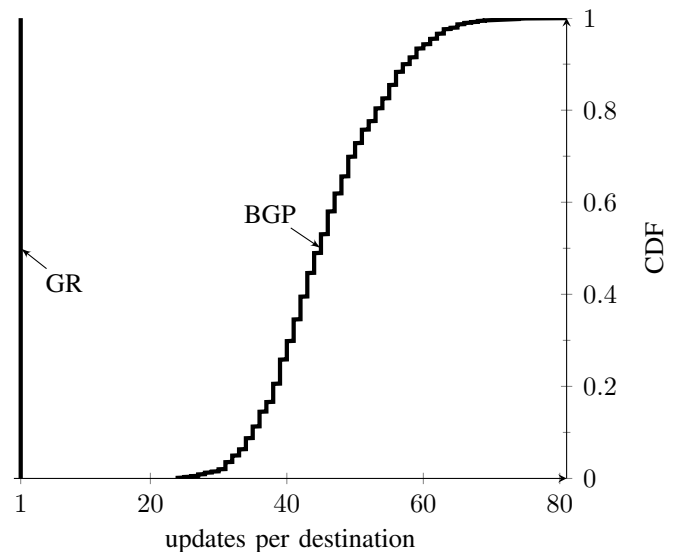


Fig. 16: Update load due to control failure.

experimented with an MRAI of 30 seconds, and observed even longer convergence times and greater disruptions to network connectivity.

Figure 14 shows a CDF of convergence times in each experiment. On average, it took more than 15 seconds to reconverge to the original route, even though the underlying network topology was unchanged. During this period, we observed many events in which some of the routers believed some destination to be unreachable, or reachable through a path that is actually a routing loop. We report these periods in Figure 15, showing the time interval during which each individual router was unable to reach one or more destinations, as determined by analysis of the global collection of the routing tables at each instant. On average, every router in the system



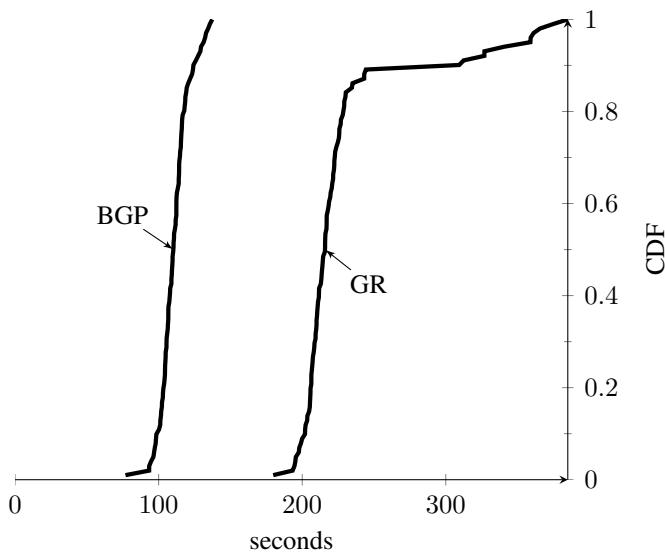


Fig. 17: Duration of convergence due to correlated control and forwarding failure.

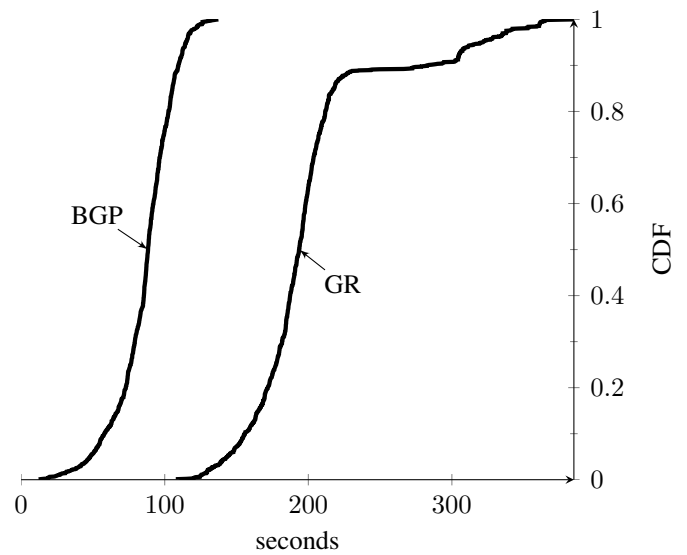


Fig. 18: Average duration a router is disconnected from the destination due to correlated control and forwarding failure.

experienced more than 11 seconds of connectivity loss for each fail-over event, even though BGP itself recovered immediately.

In addition to disrupting the forwarding plane, reconvergence taxes the control plane by consuming bandwidth and CPU to process updates. Figure 16 shows the number of updates sent for each router for the single destination being advertised. In a core Internet deployment, where a routing table might include tens of thousands of destinations, the more than 45 updates per router we see here would be multiplied by the number of destinations.

The vertical CDFs demonstrate the impact of using GR. With this feature enabled, the router announces that it will preserve its routing tables across restarts, and its peers continue to route traffic through it during restart. The routing flap seen with the basic BGP recovery is thus avoided and reconvergence is immediate. No network disconnections occur, and there is just a single redundant advertisement of each route. Thus, in these experiments, GR does nearly as well as the non-stop routing achievable with transparent connection recovery, except for sending a small number of unnecessary route advertisements.

2) *Forwarding Element Failover*: There are cases when GR can perform worse than totally unmasked restarts. GR is effective because it changes the way that peers interpret connectivity loss. In the default BGP behavior, connection loss is interpreted to indicate forwarding plane failure; with GR, the forwarding plane is assumed to continue functioning on “autopilot”, so that only the control plane requires resynchronization. When this assumption is valid, GR almost solves the failover problem. However, when the network topology does change while the control plane is still recovering, GR can instead delay the needed routing adaptation. A consequence is that routers may use bad paths based on the assumption that the recovering router has applied a routing update that it has not had time to receive, process, and install in its hardware-mediated forwarding plane.

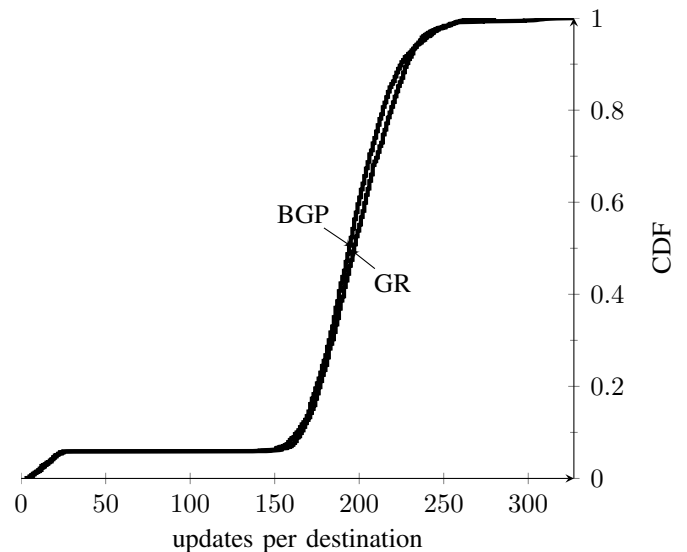


Fig. 19: Update load due to correlated control and forwarding failure.

To experiment with forwarding failover, we modified the control experiment to use a B-CLIQUE topology. A B-CLIQUE is the same as a CLIQUE, but with an additional backup path. A second router in the clique peers with the route injector, and receives a route to the destination that has an inflated path length, making it less desirable than any path through the primary link.

As shown in Figure 17, the convergence times for actual link loss are worse than for a pure control failover; on average, it takes more than 111 seconds for the network to completely switch over to the backup. The disconnectivity, shown in Figure 18, is similarly inflated, with an average of 86 seconds outage. Finally, as shown in Figure 19, an average of nearly

190 updates per router, per destination are required to reach convergence.

More important than the behavior of BGP under link failure, however, is the behavior of GR. In our experiments, the recovering connection with the primary link advertises many new routes, which get filtered out before reaching the main network but do serve to prolong resynchronization. During that time, GR prevents convergence from even beginning, and yet an underlying network link has failed. Thus, the GR data can be seen to be similar to the raw case, but with a convergence delay increased by more than 100 seconds. Worse still, our experiments used a fairly small network. These delays depend strongly on the size of the underlying network routing tables, which in the core of the Internet are far larger than in our experimental test case, and growing.

#### D. TCPR

Fail-over with TCPR offers the best results: the failure is masked, hence the control-plane recovers as soon as the failure has been sensed and the recoverable application restarted, with no overheads. For forwarding failover, TCPR is identical to unprotected BGP, and avoids the additional delays of GR.

### V. RELATED WORK

Pei et. al. [13] have studied the disruption caused during BGP convergence, explaining how topology and configuration affect the result, and evaluated a number of proposed strategies for making convergence faster and less disruptive. There has also been work that, rather than making convergence itself faster, ensures that routers can rapidly fail over to temporary routes until convergence is complete [15]. Both branches of work are valuable when convergence is unavoidable, such as when the topology genuinely changes. However, since they do not eliminate the disruption, it is preferable to mask failure and recover transparently when possible.

Prior work on transparent TCP masking and recovery has taken the approach of replicating the entire TCP stack and everything above it, the application included. For example, HydraNet-FT [16], CoRAL [17], [18], HotSwap [19], ST-TCP [20], [21], AR-TCP [22], TRODS [23], and others [24], [25] do so using primary-backup replication.

FATPETS [26] is a network stack written in Erlang, designed specifically for failover and migration. It can operate in an “active” mode, which essentially consists of primary-backup replication of the entire state including buffers, or a “passive” mode, which protects input with delayed acknowledgments like TCPR.

In order to avoid changing the network stack, new software can be interposed on its interfaces with the application and the network, recording and possibly modifying its incoming and outgoing events. Failover-TCP [27] and FT-TCP [28], [29], [8] use such a technique. Similar to our TCPR approach, acknowledgments are delayed until the data is safely handled and sequence numbers in the packets to and from a restarted network stack have to be rewritten.

All of these approaches suffer from being replay-driven. Often they maintain multiple active replicas by duplicating input to each of them, and using only the primary replica’s

output. Failover between active replicas is straightforward and fast. However, to avoid running out of replicas, new ones must be brought up to speed; the replay-driven approach assumes that the application is deterministic, and warms up new replicas by replaying all of a connection’s input, at either the packet or the socket level. For an HTTP request, that can be quite effective, but it quickly becomes impractical for connections that receive a lot of input, such as a BGP session.

Another approach is to use a proxy server between a server and its clients [30]. The client sets up a connection to the proxy server, and the proxy server handles failing over from a primary server to a backup server as necessary. However, without an approach for replicating the proxy [31], it also introduces a new single point of failure.

Virtualization offers an even lower-level approach to transparent migration. For example, Remus [32] replicates an entire virtual machine, including the application, the network stack, and the operating system. Virtual machine migration can also be exploited to protect particular applications [33], [34]. Using such technology, however, the failure or reconfiguration of the application itself within the virtual machine image still causes its TCP connections to break, thus providing only limited benefit for the heavyweight replication demands.

Backdoors [35] takes a unique approach based on replicating state *after* the application has already failed, by assuming that the network interfaces survive the failure and modifying them to support remote DMA.

Finally, when it is possible to modify all of the application’s peers, a good option is to introduce a session library or modified socket library [36], [37], [38], [39], [40]. Such an implementation could automatically set up a new connection to a new server in case the connection to the current one fails, or even maintain redundant connections.

### VI. CONCLUSIONS

Application-driven TCP recovery is a novel approach that enables a fault-tolerant application to protect connections in cases that were impossible with replay-driven recovery. Our prototype, TCPR, confirms the simplicity of application-driven recovery. Rather than replicating the TCP state and everything above it, TCPR is not even replicated itself, because the application assumes responsibility. TCPR is middleware, outside the network stack, enabling any application to use it with an unmodified, unwrapped network stack and whatever interface it provides. By controlling the acknowledgment of data, along with occasional input from the application layer, TCPR needs only a small, constant amount of state per connection, enabling low overhead in normal operation and sub-millisecond recovery.

BGP is an important application for TCP recovery, because broken connections trigger disproportionate amounts of disruption for core Internet forwarding. Furthermore, BGP stretches or breaks some crucial assumptions made by prior work, including determinism, whether the application can be a TCP client, and limitations on the length of its input.

Our evaluation of BGP uses a novel measurement framework to demonstrate that unmasked failover can result in tens or hundreds of seconds of outages, even when it is

not necessary to reconverge at all. Graceful Restart, although beneficial in some cases, can arbitrarily delay convergence when forwarding and control failures are correlated.

Ironically, a high-availability router might benefit from a design that *increases* the correlation between such failures. Forwarding elements in large routers usually have general-purpose CPUs and memory, in addition to the specialized forwarding hardware. The general-purpose hardware exists to handle slow-path forwarding decisions and configuration tasks, but is usually over-provisioned. Offloading control element responsibilities onto parts of forwarding hardware can dramatically increase the replication possibilities, from one or two dedicated control nodes to every component in the router. With more replicas come more frequent faults, but also additional fault-tolerance and greater parallelism for distributing workloads. Restructuring a router to exploit those resources would be unwise with Graceful Restart, because any forwarding element failure would be correlated with the failure of some control element functionality. However, failover using TCPR is lightweight and, more importantly, invisible to peers, so the system designer can be free to fully exploit the available resources. In fact, our work on TCPR originated in a project that prototyped just such an architecture [41].

The ability to easily migrate connections between replicas can also be useful when running multiple versions or configurations of an application. For example, a router could be made tolerant to implementation bugs by exploiting software diversity [42]. Using TCPR to protect the output of a master version or voting module can enable such an approach to tolerate hardware failure as well.

T CPR can offer benefit to applications other than BGP. For example, TCPR enables a lightweight approach to migration that can also tolerate unexpected failures, and can therefore be beneficial for load balancing long-running connections for streaming media within a CDN point-of-presence. We are also exploring applications of TCPR to a cloud-based smart power grid controller, where the middlebox helps make connections to data collectors fault-tolerant and enables elastic rebalancing of connections within the cloud, without complicating the implementation of deployed hardware.

High-availability, fault-tolerant applications already do a lot of work to maintain their own state. Application-driven TCP recovery can be seen as merely a way to given such an application access to its own state, which would otherwise be encapsulated in a network stack, without interfering with the implementation hidden behind that encapsulation.

#### ACKNOWLEDGMENTS

We thank Robert Broberg and the entire Ludd project at Cisco Systems, whose work on reliable routers inspired, supported, and tested TCPR, providing the uncommon opportunity to work on a real core Internet router architecture. We also thank Tudor Marian, who invested significant effort evaluating an early prototype of TCPR. Our work was supported, in part, by grants from the NSF, Cisco, DARPA, AFRL and DOE (ARPAe).

#### AVAILABILITY

T CPR is free software under the BSD license. Source code and documentation are available from:

<http://github.com/rahpaere/tcp/>

#### REFERENCES

- [1] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian, "Delayed Internet routing convergence," in *Proc. ACM SIGCOMM*, 2000, pp. 175–187.
- [2] D. Pei, B. Zhang, D. Massey, and L. Zhang, "An analysis of convergence delay in path vector routing protocols," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 50, no. 3, Feb. 2006.
- [3] X. Zhao, D. Pei, D. Massey, and L. Zhang, "A study on the routing convergence of Latin American networks," in *LANC 2003*, La Paz, Bolivia, Oct. 2003.
- [4] A. Sahoo, K. Kant, and P. Mohapatra, "Characterization of BGP recovery time under large-scale failures," in *IEEE International Conference on Communications (ICC'06)*, Jun. 2006, pp. 949–954.
- [5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. on Dependable and Secure Computing*, vol. 1, no. 1, 2004.
- [6] S. Sangli, E. Chen, R. Fernando, J. Scudder, and Y. Rekhter, "Graceful restart mechanism for BGP," RFC 4724, Jan. 2007.
- [7] Information Sciences Institute, "Transmission Control Protocol," RFC 793, Sep. 1981.
- [8] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. Bressoud, "Practical and low-overhead masking of failures of TCP-based servers," *Transactions on Computer Systems*, vol. 27, no. 2, 2009.
- [9] R. Braden, "Requirements for Internet hosts—communication layers," RFC 1122, Oct. 1989.
- [10] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," in *ICDCS*. IEEE Computer Society, 1981, pp. 509–512.
- [11] V. Jacobson, R. Braden, and D. Borman, "TCP extensions for high performance," RFC 1323, May 1992.
- [12] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP selective acknowledgment options," RFC 2018, Oct. 1996.
- [13] D. Pei, X. Zhao, D. Massey, and L. Zhang, "A study of BGP path vector route looping behavior," in *Proceedings of the 24th International Conference on Distributed Computing Systems*, 2004, pp. 720–729.
- [14] P. Jakama, "Revised default values for the BGP 'Minimum Route Advertisement Interval'," Internet-Draft draft-jakama-mrai-02, Nov. 2008.
- [15] O. Bonaventure, C. Filsfil, and P. Francois, "Achieving sub-50 milliseconds recovery upon BGP peering link failures," *IEEE/ACM Transactions on Networking*, vol. 15, no. 5, Oct. 2007.
- [16] G. Shenoy, S. K. Satapati, and R. Bettati, "HydraNet-FT: Network support for dependable services," in *Proc. of the International Conference on Distributed Computing Systems (ICDCS'00)*, 2000, pp. 699–706.
- [17] N. Aghdaie and Y. Tamir, "Client-transparent fault-tolerant web service," in *Proceedings of the 20th IEEE International Performance, Computing, and Communications Conference (IPCCC 2001)*, Phoenix, AZ, Apr. 2001.
- [18] —, "CoRAL: A transparent fault-tolerant web service," *Journal of Systems and Software*, vol. 82, Jan. 2009.
- [19] N. Burton-Krahn, "HotSwap – transparent server failover for Linux," in *Proc. of USENIX LISA 02: Sixteenth Systems Administration Conference*, 2002.
- [20] M. Marwah, S. Mishra, and C. Fetzer, "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: IEEE Computer Society, Jun. 2003.
- [21] —, "A system demonstration of ST-TCP," in *Proc. of the 2005 IEEE International Conference on Dependable Systems and Networks (DSN 2005)*. Yokohama, Japan: IEEE Computer Society, Jun. 2005.

- [22] Z. Shao, H. Jin, and J. Wu, "AR-TCP: Actively replicated TCP connections for cluster of workstations," in *Workshop on Frontier of Computer Science and Technology (FCST '06)*, Fukushima, Japan, Nov. 2006, pp. 3–10.
- [23] W. Lloyd and M. J. Freedman, "Coercing clients into facilitating failover for object delivery," in *DSN*, 2011, pp. 157–168.
- [24] M.-Y. Luo and C.-S. Yang, "Constructing zero-loss web services," in *Proceedings of IEEE INFOCOM*, Anchorage, AK, 2001, pp. 1781–1790.
- [25] R. Zhang, T. Abdelzaher, and J. Stankovic, "Efficient TCP connection failover in server clusters," in *Proceedings of IEEE INFOCOM*, Hong Kong, 2004, pp. 1219–1228.
- [26] J. Paris, A. Valderruten, and V. Gulias, "Developing a functional TCP/IP stack oriented towards TCP connection replication," in *Proceedings of the 3rd International IFIP/ACM Latin American conference on Networking (LANC'05)*, Cali, Columbia, Oct. 2005.
- [27] R. Koch, S. Hortikar, L. Moser, and P. Melliar-Smith, "Transparent TCP connection failover," in *Proc. of the International Conference on Dependable Systems and Networks (DSN'03)*. IEEE Computer Society, 2003, pp. 383–392.
- [28] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov, "Wrapping server-side TCP to mask connection failures," in *Proc. of Infocom 2001*, Anchorage, Alaska, Apr. 2001, pp. 329–338.
- [29] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. Bressoud, "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: IEEE Computer Society, 2003.
- [30] M. Marwah, S. Mishra, and C. Fetzer, "Fault-tolerant and scalable TCP splice and web server architecture," in *Proc. of the 25th Symposium on Reliability in Distributed Software (SRDS'06)*. IEEE Computer Society, 2006, pp. 301–310.
- [31] —, "Enhanced server fault-tolerance for improved user experience," in *Proc. of the International Conference on Dependable Systems and Networks (DSN'08)*. IEEE Computer Society, 2008, pp. 167–176.
- [32] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the USENIX Symposium on Networked Systems and Implementation*, San Francisco, CA, Apr. 2008.
- [33] E. Keller, J. Rexford, and J. van der Merwe, "Seamless BGP migration with router grafting," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI'10)*, San Jose, CA, Apr. 2010.
- [34] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the USENIX Symposium on Networked Systems and Implementation*, Boston, MA, May 2005.
- [35] F. Sultan, A. Bohra, S. Smaldone, Y. Pan, P. Gallard, I. Neamtiu, and L. Iftode, "Recovering Internet service sessions from operating system failures," *IEEE Internet Computing*, vol. 9, no. 2, pp. 17–27, 2005.
- [36] Y. Huang and C. M. R. Kintala, "Software Implemented Fault Tolerance: Technologies and experience," in *Proceedings of 23rd International Symposium on Fault Tolerant Computing (FTCS-23)*, Jun. 1993, pp. 2–9.
- [37] M. Orgiyan and C. Fetzer, "Tapping TCP streams," in *Proc. of the International Symposium on Network Computing and Applications (NCA'01)*. IEEE Computer Society, 2001, pp. 278–289.
- [38] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan, "Fine-grained failover using connection migration," in *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, CA, 2001, pp. 221–232.
- [39] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode, "Migratory TCP: Connection migration for service continuity over the Internet," in *Proc. of the 22nd International Conference on Distributed Computing Systems (ICDCS '02)*, Jul. 2002.
- [40] V. C. Zandy and B. P. Miller, "Reliable network connections," in *Proceedings of ACM MobiCom*, Atlanta, GA, 2002, pp. 95–106.
- [41] A. Agapi, K. Birman, R. M. Broberg, C. Cotton, T. Kielmann, M. Millnert, R. Payne, R. Surton, and R. van Renesse, "Routers for the cloud," *Internet Computing*, vol. 15, no. 5, Sep. 2011.
- [42] E. Keller, M. Yu, M. Caesar, and J. Rexford, "Virtually eliminating router bugs," in *CoNEXT*, 2009.