Reprinted from the

Proceedings of the Linux Symposium

Volume One

July 21th–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

TCP Connection Passing

Werner Almesberger
werner@almesberger.net

Abstract

tcpcp is an experimental mechanism that allows cooperating applications to pass ownership of TCP connection endpoints from one Linux host to another one. tcpcp can be used between hosts using different architectures and does not need the other endpoint of the connection to cooperate (or even to know what's going on).

1 Introduction

When designing systems for load-balancing, process migration, or fail-over, there is eventually the point where one would like to be able to "move" a socket from one machine to another one, without losing the connection on that socket, similar to file descriptor passing on a single host. Such a move operation usually involves at least three elements:

1. Moving any application space state related to the connection to the new owner. E.g. in the case of a Web server serving large static files, the application state could simply be the file name and the current position in the file.

2. Making sure that packets belonging to the connection are sent to the new owner of the socket. Normally this also means that the previous owner should no longer receive them.

3. Last but not least, creating compatible network state in the kernel of the new connection owner, such that it can resume the communication where the previous owner left off.

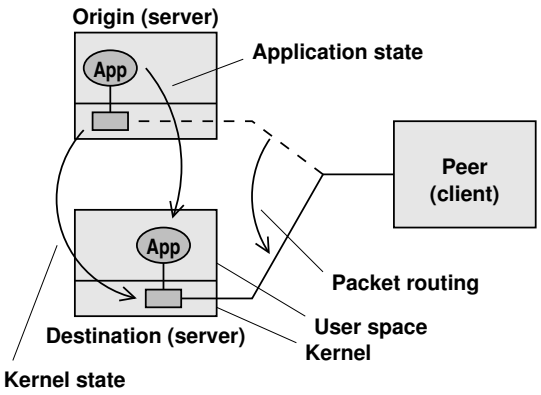

Figure 1: Passing one end of a TCP connection from one host to another.

Figure 1 illustrates this for the case of a client-server application, where one server passes ownership of a connection to another server. We shall call the host from which ownership of the connection endpoint is taken the *origin*, the host to which it is transferred the *destination*, and the host on the other end of the connection (which does not change) the *peer*.

Details of moving the application state are beyond the scope of this paper, and we will only sketch relatively simple examples. Similarly, we will mention a few ways for how the redirection in the network can be accomplished, but without going into too much detail.

The complexity of the kernel state of a network connection, and the difficulty of moving this state from one host to another, varies greatly with the transport protocol being used. Among the two major transport protocols of the Internet, UDP [1] and TCP [2], the latter clearly presents more of a challenge in this regard. Nevertheless, some issues also apply to UDP.

tcpcp (TCP Connection Passing) is a proof of concept implementation of a mechanism that allows applications to transport the kernel state of a TCP endpoint from one host to another, while the connection is established, and without requiring the peer to cooperate in any way. tcpcp is not a complete process migration or load-balancing solution, but rather a building block that can be integrated into such systems. tcpcp consists of a kernel patch (at the time of writing for version 2.6.4 of the Linux kernel) that implements the operations for dumping and restoring the TCP connection endpoint, a library with wrapper functions (see Section 3), and a few applications for debugging and demonstration.

The project's home page is at `http://tcpcp.sourceforge.net/`

The remainder of this paper is organized as follows: this section continues with a description of the context in which connection passing exists. Section 2 explains the connection passing operation in detail. Section 3 introduces the APIs tcpcp provides. The information that defines a TCP connection and its state is described in Section 4. Sections 5 and 6 discuss congestion control and the limitations TCP imposes on checkpointing. Security implications of the availability and use of tcpcp are examined in Section 7. We conclude with an outlook on future direction the work on tcpcp will take in Section 8, and the conclusions in Section 9.

The excellent "TCP/IP Illustrated" [3] is recommended for readers who wish to refresh their memory of TCP/IP concepts and terminology.

1.1 There is more than one way to do it

tcpcp is only one of several possible methods for passing TCP connections among hosts. Here are some alternatives:

In some cases, the solution is to avoid passing the "live" TCP connection, but to terminate the connection between the origin and the peer, and rely on higher protocol layers to re-establish a new connection between the destination and the peer. Drawbacks of this approach include that those higher layers need to know that they have to re-establish the connection, and that they need to do this within an acceptable amount of time. Furthermore, they may only be able to do this at a few specific points during a communication.

The use of HTTP redirection [4] is a simple example of connection passing above the transport layer.

Another approach is to introduce an intermediate layer between the application and the kernel, for the purpose of handling such redirection. This approach is fairly common in process migration solutions, such as Mosix [5], MIGSOCK [6], etc. It requires that the peer be equipped with the same intermediate layer.

1.2 Transparency

The key feature of tcpcp is that the peer can be left completely unaware that the connection is passed from one host to another. In detail, this means:

- The peer's networking stack can be used "as is," without modification and without requiring non-standard functionality
- The connection is not interrupted

- The peer does not have to stop sending
- No contradictory information is sent to the peer
- These properties apply to all protocol layers visible to the peer

Furthermore, tcpcp allows the connection to be passed at any time, without needing to synchronize the data stream with the peer.

The kernels of the hosts between which the connection is passed both need to support tcpcp, and the application(s) on these hosts will typically have to be modified to perform the connection passing.

1.3 Various uses

Application scenarios in which the functionality provided by tcpcp could be useful include load balancing, process migration, and fail-over.

In the case of load balancing, an application can send connections (and whatever processing is associated with them) to another host if the local one gets overloaded. Or, one could have a host acting as a dispatcher that may perform an initial dialog and then assigns the connection to a machine in a farm.

For process migration, tcpcp would be invoked when moving a file descriptor linked to a socket. If process migration is implemented in the kernel, an interface would have to be added to tcpcp to allow calling it in this way.

Fail-over is tricker, because there is normally no prior indication when the origin will become unavailable. We discuss the issues arising from this in more detail in Section 6.

2 Passing the connection

Figure 2 illustrates the connection passing procedure in detail.

1. The application at the origin initiates the procedure by requesting retrieval of what we call the *Internal Connection Information* (ICI) of a socket. The ICI contains all the information the kernel needs to recreate a TCP connection endpoint
2. As a side-effect of retrieving the ICI, tcpcp *isolates* the connection: all incoming packets are silently discarded, and no packets are sent. This is accomplished by setting up a per-socket filter, and by changing the output function. Isolating the socket ensures that the state of the connection being passed remains stable at either end.
3. The kernel copies all relevant variables, plus the contents of the out-of-order and send/retransmit buffers to the ICI. The out-of-order buffer contains TCP segments that have not been acknowledged yet, because an earlier segment is still missing.
4. After retrieving the ICI, the application empties the receive buffer. It can either process this data directly, or send it along with the other information, for the destination to process.
5. The origin sends the ICI and any relevant application state to the destination. The application at the origin keeps the socket open, to ensure that it stays isolated.
6. The destination opens a new socket. It may then bind it to a new port (there are other choices, described below).

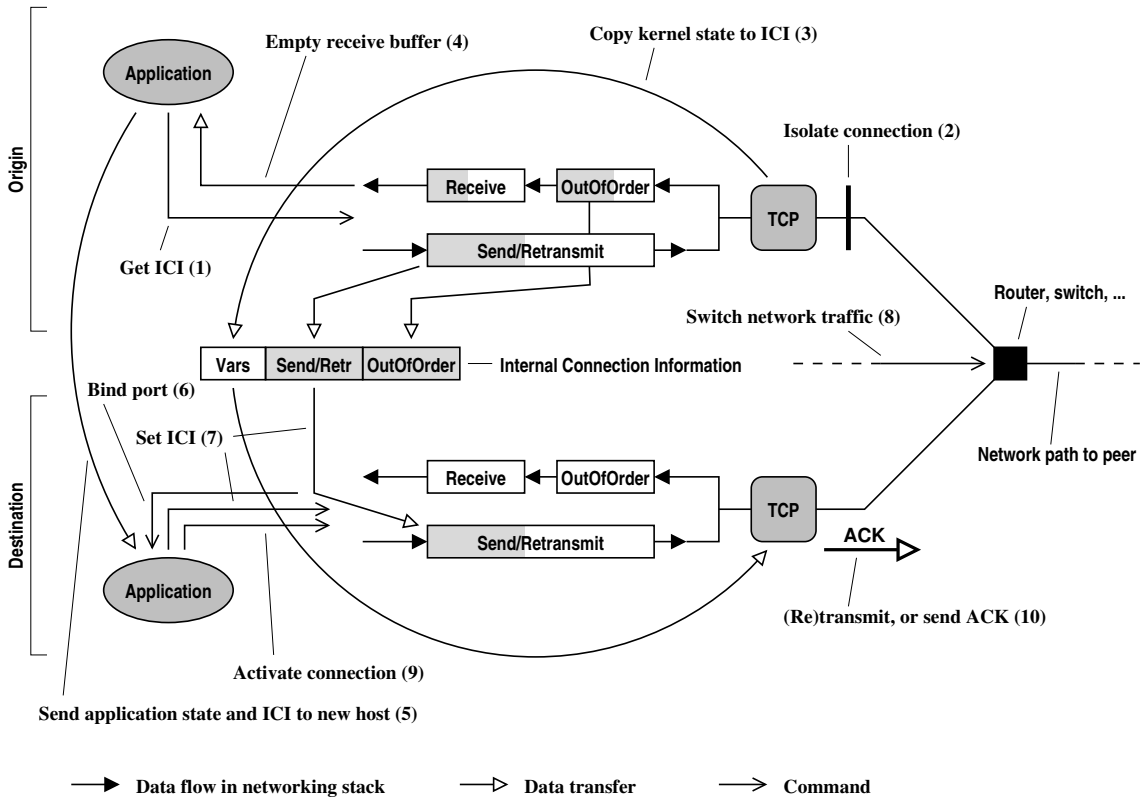

Figure 2: Passing a TCP connection endpoint in ten easy steps.

7. The application at the destination now sets the ICI on the socket. The kernel creates and populates the necessary data structures, but does not send any data yet. The current implementation makes no use of the out-of-order data.

8. Network traffic belonging to the connection is redirected from the origin to the destination host. Scenarios for this are described in more detail below. The application at the origin can now close the socket.

9. The application at the destination makes a call to *activate* the connection.

10. If there is data to transmit, the kernel will do so. If there is no data, an otherwise empty ACK segment (like a window probe) is sent to wake up the peer.

Note that, at the end of this procedure, the socket at the destination is a perfectly normal TCP endpoint. In particular, this endpoint can be passed to another host (or back to the original one) with tcpcp.

2.1 Local port selection

The local port at the destination can be selected in three ways:

- The destination can simply try to use the same port as the origin. This is necessary if no address translation is performed on the connection.

- The application can bind the socket before setting the ICI. In this case, the port in the ICI is ignored.

- The application can also clear the port information in the ICI, which will cause the socket to be bound to any available port. Compared to binding the socket before setting the ICI, this approach has the advantage of using the local port number space much more efficiently.

The choice of the port selection method depends on how the environment in which tcpcp operates is structured. Normally, either the first or the last method would be used.

2.2 Switching network traffic

There are countless ways for redirecting IP packets from one host to another, without help from the transport layer protocol. They include redirecting part of the link layer, ingenious modifications of how link and network layer interact [7], all kinds of tunnels, network address translation (NAT), etc.

Since many of the techniques are similar to network-based load balancing, the Linux Virtual Server Project [8] is a good starting point for exploring these issues.

While a comprehensive study of this topic if beyond the scope of this paper, we will briefly sketch an approach using a static route, because this is conceptually straightforward and relatively easy to implement.

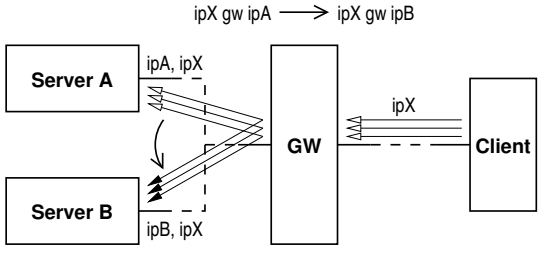

Figure 3: Redirecting network traffic using a static route.

The scenario shown in Figure 3 consists of two servers **A** and **B**, with interfaces with the IP addresses **ipA** and **ipB**, respectively. Each server also has a virtual interface with the address **ipX**. **ipA**, **ipB**, and **ipX** are on the same subnet, and also the gateway machine has an interface on this subnet.

At the gateway, we create a static route as follows:

`route add` *ipX* `gw` *ipA*

When the client connects to the address **ipX**, it reaches host **A**. We can now pass the connection to host **B**, as outlined in Section 2. In Step 8, we change the static route on the gateway as follows:

```
route del ipX
route add ipX gw ipB
```

One major limitation of this approach is of course that this routing change affects all connections to **ipX**, which is usually undesirable. Nevertheless, this simple setup can be used to demonstrate the operation of tcpcp.

3 APIs

The API for tcpcp consists of a low-level part that is based on getting and setting socket options, and a high-level library that provides convenient wrappers for the low-level API.

We mention only the most important aspects of both APIs here. They are described in more detail in the documentation that is included with tcpcp.

3.1 Low-level API

The ICI is retrieved by getting the `TCP_ICI` socket option. As a side-effect, the connection is isolated, as described in Section 2. The application can determine the maximum ICI size

for the connection in question by getting the `TCP_MAXICISIZE` socket option.

Example:

```
void *buf;
int ici_size;
size_t size = sizeof(int);

getsockopt(s,SOL_TCP,TCP_MAXICISIZE,
    &ici_size,&size);
buf = malloc(ici_size);
size = ici_size;
getsockopt(s,SOL_TCP,TCP_ICI,
    buf,&size);
```

The connection endpoint at the destination is created by setting the `TCP_ICI` socket option, and the connection is activated by "setting" the `TCP_CP_FN` socket option to the value `TCPCP_ACTIVATE`.[1]

Example:

```
int sub_function = TCPCP_ACTIVATE;

setsockopt(s,SOL_TCP,TCP_ICI,
    buf,size);
/* ... */
setsockopt(s,SOL_TCP,TCP_CP_FN,
    &sub_function,
    sizeof(sub_function));
```

3.2 High-level API

These are the most important functions provided by the high-level API:

```
void *tcpcp_get(int s);
int tcpcp_size(const void *ici);
int tcpcp_create(const void *ici);
int tcpcp_activate(int s);
```

[1] The use of a multiplexed socket option is admittedly ugly, although convenient during development.

`tcpcp_get` allocates a buffer for the ICI, and retrieves that ICI (isolating the connection as a side-effect). The amount of data in the ICI can be queried by calling `tcpcp_size` on it.

`tcpcp_create` sets an ICI on a socket, and `tcpcp_activate` activates the connection.

4 Describing a TCP endpoint

In this section, we describe the parameters that define a TCP connection and its state. tcpcp collects all the information it needs to re-create a TCP connection endpoint in a data structure we call *Internal Connection Information* (ICI).

The ICI is portable among systems supporting tcpcp, irrespective of their CPU architecture.

Besides this data, the kernel maintains a large number of additional variables that can either be reset to default values at the destination (such as congestion control state), or that are only rarely used and not essential for correct operation of TCP (such as statistics).

4.1 Connection identifier

Each TCP connection in the global Internet or any private internet [9] is uniquely identified by the IP addresses of the source and destination host, and the port numbers used at both ends.

tcpcp currently only supports IPv4, but can be extended to support IPv6, should the need arise.

4.2 Fixed data

A few parameters of a TCP connection are negotiated during the initial handshake, and remain unchanged during the life time of the connection. These parameters include whether window scaling, timestamps, or selective acknowledgments are used, the number of bits by

Connection identifier	
`ip.v4.ip_src`	IPv4 address of the host on which the ICI was recorded (source)
`ip.v4.ip_dst`	IPv4 address of the peer (destination)
`tcp_sport`	Port at the source host
`tcp_dport`	Port at the destination host
Fixed at connection setup	
`tcp_flags`	TCP flags (window scale, SACK, ECN, etc.)
`snd_wscale`	Send window scale
`rcv_wscale`	Receive window scale
`snd_mss`	Maximum Segment Size at the source host
`rcv_mss`	MSS at the destination host
Connection state	
`state`	TCP connection state (e.g. ESTABLISHED)
Sequence numbers	
`snd_nxt`	Sequence number of next new byte to send
`rcv_nxt`	Sequence number of next new byte expected to receive
Windows (flow-control)	
`snd_wnd`	Window received from peer
`rcv_wnd`	Window advertised to peer
Timestamps	
`ts_gen`	Current value of the timestamp generator
`ts_recent`	Most recently received timestamp

Table 1: TCP variables recorded in tcpcp's Internal Connection Information (ICI) structure.

which the window is shifted, and the maximum segment sizes (MSS).

These parameters are used mainly for sanity checks, and to determine whether the destination host is able to handle the connection. The received MSS continues of course to limit the segment size.

4.3 Sequence numbers

The sequence numbers are used to synchronize all aspects of a TCP connection.

Only the sequence numbers we expect to see in the network, in either direction, are needed when re-creating the endpoint. The kernel uses several variables that are derived from these sequence numbers. The values of these variables either coincide with `snd_nxt` and `rcv_nxt` in the state we set up, or they can be calculated by examining the send buffer.

4.4 Windows (flow-control)

The (flow-control) window determines how much more data can be sent or received without overrunning the receiver's buffer.

The window the origin received from the peer is also the window we can use after re-creating the endpoint.

The window the origin advertised to the peer defines the minimum receive buffer size at the destination.

4.5 Timestamps

TCP can use timestamps to detect old segments with wrapped sequence numbers [10]. This mechanism is called *Protect Against Wrapped Sequence numbers* (PAWS).

Linux uses a global counter (`tcp_time_stamp`) to generate local timestamps. If a moved connection were to use the counter at the new host, local round-trip-time calculation may be confused when receiving timestamp replies from the previous connection, and the peer's PAWS algorithm will discard segments if timestamps appear to have jumped back in time.

Just turning off timestamps when moving the connection is not an acceptable solution, even though [10] seems to allow TCP to just stop sending timestamps, because doing so would bring back the problem PAWS tries to solve in the first place, and it would also reduce the accuracy of round-trip-time estimates, possibly degrading the throughput of the connection.

A more satisfying solution is to synchronization the local timestamp generator. This is accomplished by introducing a per-connection timestamp offset that is added to the value of `tcp_time_stamp`. This calculation is hidden in the macro `tp_time_stamp(tp)`, which just becomes `tcp_time_stamp` if the kernel is configured without tcpcp.

The addition of the timestamp offset is the only major change tcpcp requires in the existing TCP/IP stack.

4.6 Receive buffers

There are two buffers at the receiving side: the buffer containing segments received out-of-order (see Section 2), and the buffer with data that is ready for retrieval by the application. tcpcp currently ignores both buffers: the out-of-order buffer is copied into the ICI, but not used when setting up the new socket. Any data in the receive buffer is left for the application to read and process.

4.7 Send buffer

The send and retransmit buffer contains data that is no longer accessible through the socket API, and that cannot be discarded. It is therefore placed in the ICI, and used to populate the send buffer at the destination.

4.8 Selective acknowledgments

In Section 5 of [11], the use of inbound SACK information is left optional. tcpcp takes advantage of this, and neither preserves SACK information collected from inbound segments, nor the history of SACK information sent to the peer.

Outbound SACKs convey information about the receiver's out-of-order queue. Fortunately, [11] declares this information as purely advisory. In particular, if reception of data has been acknowledged with a SACK, this does not imply that the receiver has to remember having done so. First, it can request retransmission of this data, and second, when constructing new SACKs, the receiver is encouraged to include information from previous SACKs, but is under no obligation to do so.

Therefore, while [11] discourages losing SACK information, doing so does not violate its requirements.

Losing SACK information may temporarily degrade the throughput of the TCP connection. This is currently of little concern, because tcpcp forces the connection into slow start, which has even more drastic performance implications.

SACK recovery may need to be reconsidered once tcpcp implements more sophisticated congestion control.

4.9 Other data

The TCP connection state is currently always ESTABLISHED. It may be useful to also allow passing connections in earlier states, e.g. SYN_RCVD. This is for further study.

Congestion control data and statistics are currently omitted. The new connection starts with slow-start, to allow TCP to discover the characteristics of the new path to the peer.

5 Congestion control

Most of the complexity of TCP is in its congestion control. tcpcp currently avoids touching congestion control almost entirely, by setting the destination to slow start.

This is a highly conservative approach that is appropriate if knowing the characteristics of the path between the origin and the peer does not give us any information on the characteristics of the path between the destination and the peer, as shown in the lower part of Figure 4.

However, if the characteristics of the two paths can be expected to be very similar, e.g. if the hosts passing the connection are on the same LAN, better performance could be achieved by allowing tcpcp to resume the connection at or nearly at full speed.

Re-establishing congestion control state is for further study. To avoid abuse, such an operation can be made available only to sufficiently trusted applications.

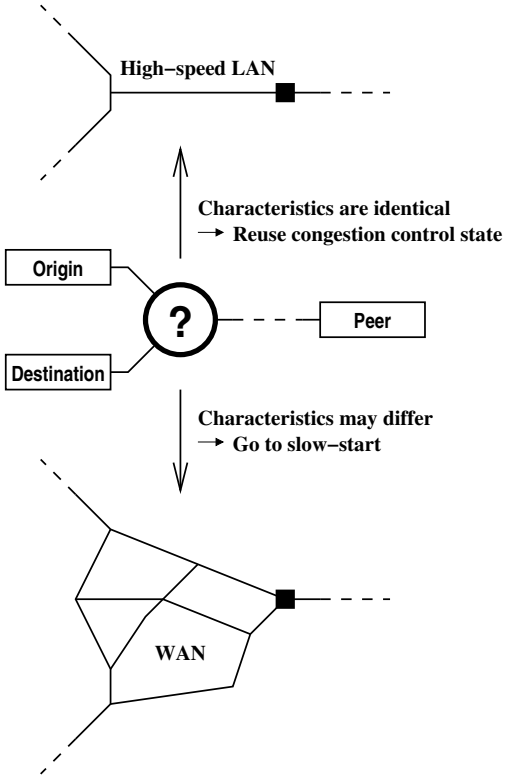

Figure 4: Depending on the structure of the network, the congestion control state of the original connection may or may not be reused.

6 Checkpointing

tcpcp is primarily designed for scenarios, where the old and the new connection owner are both functional during the process of connection passing.

A similar usage scenario would if the node owning the connection occasionally retrieves ("checkpoints") the momentary state of the connection, and after failure of the connection owner, another node would then use the checkpoint data to resurrect the connection.

While apparently similar to connection passing, checkpointing presents several problems which we discuss in this section. Note that this is speculative and that the current implementation of tcpcp does not support any of the exten-

sions discussed here.

We consider the send and receive flow of the TCP connection separately, and we assume that sequence numbers can be directly translated to application state (e.g. when transferring a file, application state consists only of the actual file position, which can be trivially mapped to and from TCP sequence numbers). Furthermore, we assume the connection to be in ESTABLISHED state at both ends.

6.1 Outbound data

One or more of the following events may occur between the last checkpoint and the moment the connection is resurrected:

- the sender may have enqueued more data
- the receiver may have acknowledged more data
- the receiver may have retrieved more data, thereby growing its window

Assuming that no additional data has been received from the peer, the new sender can simply re-transmit the last segment. (Alternatively, `tcp_xmit_probe_skb` might be useful for the same purpose.) In this case, the following protocol violations can occur:

- The sequence number may have wrapped. This can be avoided by making sure that a checkpoint is never older than the Maximum Segment Lifetime (MSL)[2], and that less than 2^{31} bytes are sent between checkpoints.
- If using PAWS, the timestamp may be below the last timestamp sent by the old sender. The best solution for avoiding this is probably to tightly synchronize clock on the old and the new connection owner, and to make a conservative estimate of the number of ticks of the local timestamp clock that have passed since taking the checkpoint. This assumes that the timestamp clock ticks roughly in real time.

Since new data in the segment sent after resurrecting the connection cannot exceed the receiver's window, the only possible outcomes are that the segment contains either new data, or only old data. In either case, the receiver will acknowledge the segment.

Upon reception of an acknowledgment, either in response to the retransmitted segment, or from a packet in flight at the time when the connection was resurrected, the sender knows how far the connection state has advanced since the checkpoint was taken.

If the sequence number from the acknowledgment is below `snd_nxt`, no special action is necessary. If the sequence number is above `snd_nxt`, the sender would exceptionally treat this as a valid acknowledgment.[3]

As a possible performance improvement, the sender may notify the application once a new sequence number has been received, and the application could then skip over unnecessary data.

6.2 Inbound data

The main problem with checkpointing of incoming data is that TCP will acknowledge data that has not yet been retrieved by the application. Therefore, checkpointing would have to delay outbound acknowledgments until the application has actually retrieved them, and has

[2] [2] specifies a MSL of two minutes.

[3] Note that this exceptional condition does not necessarily have to occur with the first acknowledgment received.

checkpointed the resulting state change.

To intercept all types of ACKs, `tcp_transmit_skb` would have to be changed to send `tp->copied_seq` instead of `tp->rcv_nxt`. Furthermore, a new API function would be needed to trigger an explicit acknowledgment after the data has been stored or processed.

Putting acknowledges under application control would change their timing. This may upset the round-trip time estimation of the peer, and it may also cause it to falsely assume changes in the congestion level along the path.

7 Security

tcpcp bypasses various sets of access and consistency checks normally performed when setting up TCP connections. This section analyzes the overall security impact of tcpcp.

7.1 Two lines of defense

When setting TCP_ICI, the kernel has no means of verifying that the connection information actually originates from a compatible system. Users may therefore manipulate connection state, copy connection state from arbitrary other systems, or even synthesize connection state according to their wishes. tcpcp provides two mechanisms to protect against intentional or accidental mis-uses:

1. tcpcp only takes as little information as possible from the user, and re-generates as much of the state related to the TCP connection (such as neighbour and destination data) as possible from local information. Furthermore, it performs a number of sanity checks on the ICI, to ensure its integrity, and compatibility with constraints of the local system (such as buffer size limits and kernel capabilities).

2. Many manipulations possible through tcpcp can be shown to be available through other means if the application has the `CAP_NET_RAW` capability. Therefore, establishing a new TCP connection with tcpcp also requires this capability. This can be relaxed on a host-wide basis.

7.2 Retrieval of sensitive kernel data

Getting `TCP_ICI` may retrieve information from the kernel that one would like to hide from unprivileged applications, e.g. details about the state of the TCP ISN generator. Since the equally unprivileged `TCP_INFO` already gives access to most TCP connection metadata, tcpcp does not create any new vulnerabilities.

7.3 Local denial of service

Setting `TCP_ICI` could be used to introduce inconsistent data in the TCP stack, or the kernel in general. Preventing this relies on the correctness and completeness of the sanity checks mentioned before.

tcpcp can be used to accumulate stale data in the kernel. However, this is not very different from e.g. creating a large number of unused sockets, or letting buffers fill up in TCP connections, and therefore poses no new security threat.

tcpcp can be used to shutdown connections belonging to third party applications, provided that the usual access restrictions grant access to copies of their socket descriptors. This is similar to executing `shutdown` on such sockets, and is therefore believed to pose no new threat.

7.4 Restricted state transitions

tcpcp could be used to advance TCP connection state past boundaries imposed by internal or external control mechanisms. In particular, conspiring applications may create TCP connections without ever exchanging SYN packets, bypassing SYN-filtering firewalls. Since SYN-filtering firewalls can already be avoided by privileged applications, sites depending on SYN-filtering firewalls should therefore use the default setting of tcpcp, which makes its use also a privileged operation.

7.5 Attacks on remote hosts

The ability to set `TCP_ICI` makes it easy to commit all kinds of of protocol violations. While tcpcp may simplify implementing such attacks, this type of abuses has always been possible for privileged users, and therefore, tcpcp poses no new security threat to systems properly resistant against network attacks.

However, if a site allows systems where only trusted users may be able to communicate with otherwise shielded systems with known remote TCP vulnerabilities, tcpcp could be used for attacks. Such sites should use the default setting, which makes setting `TCP_ICI` a privileged operation.

7.6 Security summary

To summarize, the author believes that the design of tcpcp does not open any new exploits if tcpcp is used in its default configuration.

Obviously, some subtleties have probably been overlooked, and there may be bugs inadvertently leading to vulnerabilities. Therefore, tcpcp should receive public scrutiny before being considered fit for regular use.

8 Future work

To allow faster connection passing among hosts that share the same, or a very similar path to the peer, tcpcp should try to avoid going to slow start. To do so, it will have to pass more congestion control information, and integrate it properly at the destination.

Although not strictly part of tcpcp, the redirection apparatus for the network should be further extended, in particular to allow individual connections to be redirected at that point too, and to include some middleware that coordinates the redirecting with the changes at the hosts passing the connection.

It would be very interesting if connection passing could also be used for checkpointing. The analysis in Section 6 suggests that at least limited checkpointing capabilities should be feasible without interfering with regular TCP operation.

The inner workings of TCP are complex and easily disturbed. It is therefore important to subject tcpcp to thorough testing, in particular in transient states, such as during recovery from lost segments. The umlsim simulator [12] allows to generate such conditions in a deterministic way, and will be used for these tests.

9 Conclusion

tcpcp is a proof of concept implementation that successfully demonstrates that an endpoint of a TCP connection can be passed from one host to another without involving the host at the opposite end of the TCP connection. tcpcp also shows that this can be accomplished with a relatively small amount of kernel changes.

tcpcp in its present form is suitable for experimental use as a building block for load balancing and process migration solutions. Future

work will focus on improving the performance of tcpcp, on validating its correctness, and on exploring checkpointing capabilities.

References

[1] RFC768; Postel, Jon. *User Datagram Protocol*, IETF, August 1980.

[2] RFC793; Postel, Jon. *Transmission Control Protocol*, IETF, September 1981.

[3] Stevens, W. Richard. *TCP/IP Illustrated, Volume 1 – The Protocols*, Addison-Wesley, 1994.

[4] RFC2616; Fielding, Roy T.; Gettys, James; Mogul, Jeffrey C.; Frystyk Nielsen, Henrik; Masinter, Larry; Leach, Paul J.; Berners-Lee, Tim. *Hypertext Transfer Protocol – HTTP/1.1*, IETF, June 1999.

[5] Bar, Moshe. *OpenMosix*, Proceedings of the 10th International Linux System Technology Conference (Linux-Kongress 2003), pp. 94–102, October 2003.

[6] Kuntz, Bryan; Rajan, Karthik. *MIGSOCK – Migratable TCP Socket in Linux*, CMU, M.Sc. Thesis, February 2002. `http://www-2.cs.cmu.edu/~softagents/migsock/MIGSOCK.pdf`

[7] Leite, Fábio Olivé. *Load-Balancing HA Clusters with No Single Point of Failure*, Proceedings of the 9th International Linux System Technology Conference (Linux-Kongress 2002), pp. 122–131, September 2002. `http://www.linux-kongress.org/2002/papers/lk2002-leite.html`

[8] *Linux Virtual Server Project*, `http://www.linuxvirtualserver.org/`

[9] RFC1918; Rekhter, Yakov; Moskowitz, Robert G.; Karrenberg, Daniel; de Groot, Geert Jan; Lear, Eliot. *Address Allocation for Private Internets*, IETF, February 1996.

[10] RFC1323; Jacobson, Van; Braden, Bob; Borman, Dave. *TCP Extensions for High Performance*, IETF, May 1992.

[11] RFC2018; Mathis, Matt; Mahdavi, Jamshid; Floyd, Sally; Romanow, Allyn. *TCP Selective Acknowledgement Options*, IETF, October 1996.

[12] Almesberger, Werner. *UML Simulator*, Proceedings of the Ottawa Linux Symposium 2003, July 2003. `http://archive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Almesberger-OLS2003.pdf`