Enhanced Transaction TCP—Design and Implementation

Ren Bin and Zhang Xiaolan

Department of Electronic Engineering

School of Electronics and Information Technology

Shanghai Jiaotong University

bren@sjtu.edu.cn, zhang_xiaolan@sjtu.edu.cn

Abstract

This paper describes the design and implementation of Enhanced Transaction TCP. ET/TCP, which is based on Transaction TCP defined in RFC1644, is designed to provide better security and back-compatibility with TCP. We start with an introduction to T/TCP, including its purpose, design and possible applications. Then, we present several widely held claims of T/TCP security problems. After analyzing those claims in details, we provide solutions. After that, we introduce methods to achieve better compatibility with TCP. Finally, we describe the implementation of ET/TCP in Linux Kernel 2.4.2.

1 Introduction

1.1 What is T/TCP

T/TCP is an extension for standard TCP. It uses a monotonically increasing variable CC (Connection Counts) to bypass 3-way handshake (called TAO, TCP Accelerated Open) and reduce TIME_WAIT period. Figure 1 depicts a standard T/TCP connection with only three datagrams exchanged. T/TCP greatly decreases the overhead standard TCP introduces when dealing with transaction-oriented connections.

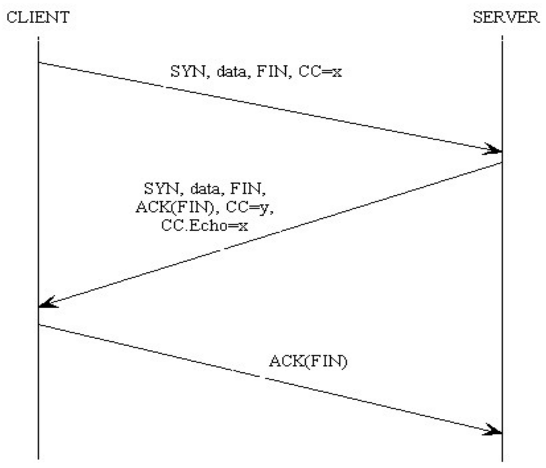


Figure 1: A typical T/TCP connection

1.2 Why T/TCP

The TCP protocol implements a virtual-circuit transport service that provides reliable and ordered data delivery over a full-duplex connection. Distributed applications, which are becoming increasingly numerous and sophisticated in the Internet nowadays, tend to use a transaction-oriented rather than a virtual circuit style of communication. Currently, a transaction-oriented Internet application must choose to suffer the overhead of opening and closing TCP connections or else build an application-specific transport mechanism on top of the connectionless transport protocol UDP.

The transaction service model has the following features:

- The fundamental interaction is a request followed by a response.
- An explicit open or close phase would impose excessive overhead.
- At-most-once semantics is required; that is, a transaction must not be "replayed" by a duplicate request packet.
- In favorable circumstances, a reliable request/response handshake can be performed with exactly one packet in each direction.
- The minimum transaction latency for a client is RTT + SPT, where RTT is the round-trip time and SPT is the server processing time.

In practice, however, most production systems implement only TCP and UDP at the transport layer. It has proven difficult to leverage a new transport protocol into place, to be widely enough available to be useful for application developers.

T/TCP is an alternative approach to providing a transaction transport protocol: extending TCP to implement the transaction service model, while continuing to support the virtual circuit model.

1.3 Feature Designs

1.3.1. TCP Accelerated Open (TAO)

T/TCP introduces a 32-bit incarnation number, called a "connection count" (CC), which is carried in a newly introduced TCP option (CC option) in each segment. A distinct CC value is assigned to each direction of an open connection. A T/TCP implementation assigns monotonically increasing CC values to successive connections that it opens actively or passively.

T/TCP uses the monotonic property of CC values in initial SYN segments to bypass the 3WHS, a mechanism that we call TCP Accelerated Open (TAO). Under TAO, a host caches a small amount of state per remote host. Specifically, a T/TCP host that is acting as a server keeps a cache containing the last valid CC value that it has received from each different client host. If an initial SYN segment (i.e., a segment with a SYN bit but no ACK bit) from a particular client host carries a CC value larger than the corresponding cached value, the monotonic property of CC ensures that the SYN segment must be new and can therefore be accepted immediately. Otherwise, the server host does not know whether the SYN segment is an old duplicate or is simply delivered out of order; it therefore executes a normal 3WHS to validate the SYN. Thus, the TAO mechanism provides an optimization, with the normal TCP mechanism as a fallback.

1.3.2. TIME-WAIT Truncation

In TCP, TIME-WAIT state is used for two purposes:

- Full-duplex connection close
- Protection against old duplicate segments

T/TCP introduces a 32-bit incarnation number, called a "connection count" (CC) that is carried in a TCP option in each segment. A distinct CC value is assigned to each direction of an open connection. A T/TCP implementation assigns monotonically increasing CC values to successive connections that it opens actively or passively.

The CC value carried in initial SYN segments will guarantee full-duplex connection close with TIME-WAIT truncated from 2*MSL (Maximum Segment Life) to 8*RTO (Retransmission Timeout).

Besides, the CC value carried in non-SYN segments is used to protect against old duplicate segments from earlier incarnations of the same connection (we call such segments 'antique duplicates' for short). In the case of short connections (e.g., transactions) that last shorter than MSL, these CC values allow TIME-WAIT state duration to be safely truncated.

1.4 Possible Applications

1.4.1. WWW

A typical HTTP client-server transaction is like this: The client does active open and sends a short request to the server. The server sends a response after receiving the request and processing the data. Then, the server does active close.

T/TCP is ideal for this transaction. The client can sends the request in its first SYN packet, thus reducing the overall time by one RTT and the number of packets exchanged. This way, both time and bandwidth are saved. The server can reduce the total number of TCBs by truncating TIME_WAIT length. This is especially useful for busy HTTP servers.

1.4.2. DNS

Nowadays, both DNS requests and responses are delivered with UDP. If a DNS response is longer than 512 bytes, only first 512 bytes are sent to the client with UDP, along with a "truncated" flag. Thereafter, the client retransmits the DNS request with TCP and the server transmits the entire DNS response with TCP.

The reason of this method is the possibility that certain hosts may not be able to reassemble IP datagrams longer than 576 bytes. (Actually, many UDP applications limit the length of user's data to 512 bytes.) Because TCP is stream-oriented, the same problem won't occur. During the 3-way handshake, both TCP sides get to know each other's MSS and agree to use the smaller one for the rest of the connection. Thereafter, the sender disassembles user's data into various packets shorter or equal to the negotiated MSS, thus avoiding IP fragmentations. The receiver simply reassembles all the received packets and delivers the received data to upper applications as required.

DNS clients and servers can make good use of T/TCP, which possesses both UDP's timesaving and TCP's reliability.

1.4.3. RPC

Nearly all the papers about applying transport protocols to transaction processing nominate RPC as a choice. With RPC, the client sends a request to the server that has the procedure to be invoked. The client includes arguments in its request while the server includes in its response the result obtained by running the specified procedure.

Usually RPC packets are large in size, resulting in the necessity of delivery reliability. T/TCP provides TCP's reliability and avoids TCP's huge overhead of 3-way handshake. All applications that reply on RPC, such as NFS, can use T/TCP for this reason.

1.4.4. Embedded Devices

In network-connected embedded devices, it is often a given that the Internet Protocol (IP) will serve as the network layer protocol. A crucial choice for transaction applications is which protocol to use at the transport layer. In light of memory use, network bandwidth, response time, reliability and interoperability, T/TCP fits these requirements better compared with standard TCP and UDP.

2 T/TCP Security

2.1 Claims of T/TCP Security Problems

2.1.1. Dominance of TAO

It is easy for an attacker to ensure the success or failure of the TAO test. There are two methods. The first relies on the second oldest cracking tool in history: sniffing; the second is more of a brutish technique, but is just as effective.

Packet Sniffing

If we are on the local network with one of the hosts, we can snoop the current CC value in use for a particular connection. Since the tcp_ccgen is incremented monotonically we can precisely spoof the next expected value by incrementing the snooped number. Not only will this ensure the success of our TAO test, but it will ensure the failure of the next TAO test for the client we are spoofing.

The Numbers Game

The other method of TAO dominance is a bit rougher, but works almost as well. The CC is an unsigned 32-bit number (ranging in value from 0 - 4,294,967,295). Under all observed implementations, the tcp_ccgen is initialized to 1. If we need to ensure the success of a TAO connection, but is not in a position where we can sniff on a local network, we should simply choose a large value for the spoofed CC. The chances that one given T/TCP host will burn through even half the tcp_ccgen space with another given host is highly unlikely. Simple statistics tell us that the larger the chosen tcp_ccgen is, the greater the odds that the TAO test will succeed. When in doubt, aim high.

2.1.2. SYN Flooding

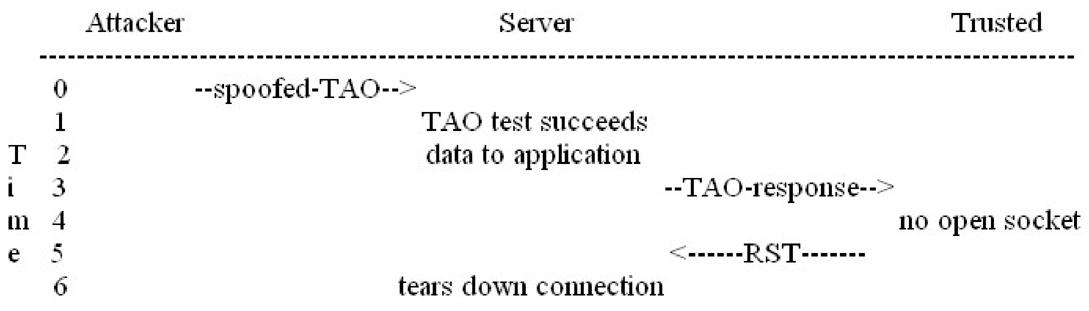
TCP SYN flooding hasn't changed much under T/TCP. The actual attack is the same: a series of TCP SYNs spoofed from unreachable IP addresses. However, there are 2 major considerations to keep in mind when the target host supports T/TCP:

SYN cookie invalidation: A host supporting T/TCP cannot, at the same time, implement SYN cookies. TCP SYN cookies are a SYN flood defense technique that works by sending a secure cookie as the sequence number in the second packet of the 3-way handshake, then discarding all state for that connection. Any TCP option sent would be lost. If the final ACK comes in, only then will the server host create the kernel socket data structures. TAO obviously cannot be used with SYN cookies.

Failed TAO processing results in queued data: If the TAO test fails, any data included with that packet will be queued pending the completion of the connection processing (the 3-way handshake). During a SYN flood, this can make the attack more severe as memory buffers holding these data fill up. In this case, the attacker would want to ensure the failure of the TAO test for each spoofed packet.

2.1.3. Trust Relationship

This is an old attack with a new twist. The attack paradigm is still the same; this time, however, it is easier to wage. Under T/TCP, there is no need to attempt to predict TCP sequence numbers. Previously, this attack required the attacker to predict the return sequence number in order to complete the connection establishment processing and move the connection into the established state. With T/TCP, a packet's data will be accepted by the application as soon as the TAO test succeeds. All the attacker needs to do is to ensure that the TAO test will succeed. Consider the Figure 2 below:



```
        Attacker                       Server                         Trusted
------------------------------------------------------------------------------
     0           --spoofed-TAO-->
     1                            TAO test succeeds
T    2                            data to application
i    3                                            --TAO-response-->
m    4                                                          no open socket
e    5                                            <------RST-------
     6                           tears down connection
```

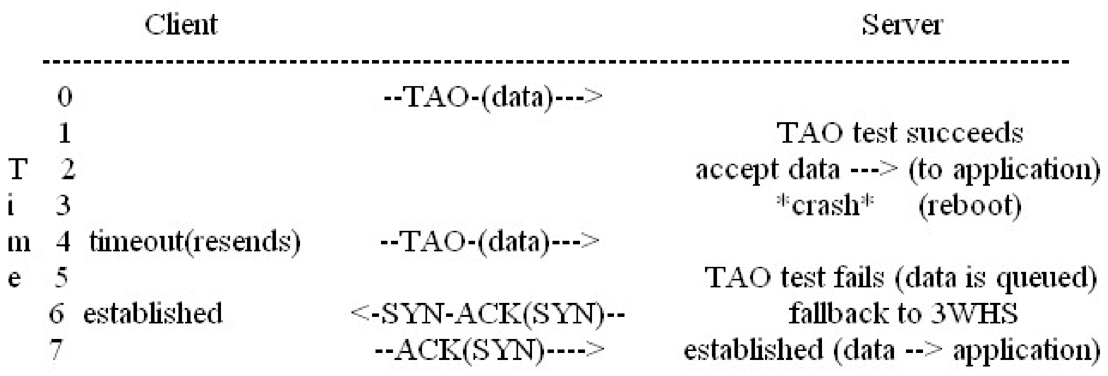
Figure 2: Trust Relationship

The attacker first sends a spoofed connection request TAO packet to the server. The data portion of this packet presumably contains the tried and true non-interactive backdoor command 'echo + + > .rhosts'. At (1) the TAO test succeeds and the data is accepted (2) and passed to application (where it is processed). The server then sends its T/TCP response to the trusted host (3). The trusted host, of course, has no open socket for this connection, and (4) responds with the expected RST segment (5). This RST will teardown the attacker's spoofed connection (6) on the server. If everything goes according to plan, and the process executing the command in question doesn't take too long to run, the attacker may now log directly into the server.

To deal with (5) the attacker can, of course, wage some sort of denial of service attack on the trusted host to keep it from responding to the unwarranted connection.

2.1.4. Duplicate Delivery

Ignoring all the other weaknesses of T/TCP, there is one major flaw that causes the T/TCP to degrade and behave decidedly non-TCP-like, therefore breaking the protocol entirely. The problem is within the TAO mechanism. Certain conditions can cause T/TCP to deliver duplicate data to the application layer. Consider the timeline in Figure 3 below:



```
          Client                                        Server
------------------------------------------------------------------------------
     0                        --TAO-(data)--->
     1                                              TAO test succeeds
T    2                                       accept data ---> (to application)
i    3                                             *crash*     (reboot)
m    4  timeout(resends)      --TAO-(data)--->
e    5                                        TAO test fails (data is queued)
     6  established        <-SYN-ACK(SYN)--          fallback to 3WHS
     7                        --ACK(SYN)---->  established (data --> application)
```

Figure 3: Duplicate Delivery

At time 0 the client sends its TAO encapsulated data to the server (for this example, consider that both hosts have had recent communication, and the server has defined CC values for the client). The TAO test succeeds (1) and the server passes the data to the application layer for processing (2). Before the server can send its response however (presumably an ACK) it crashes (3). The client receives no acknowledgement from the server, so it times out and resends its packet (4). After the server reboots it receives this retransmission, this time, however, the TAO test fails and the server queues the data (5). The failed TAO test forces a 3-way handshake (6) because the server's CC cache is invalidated when it reboots. After completing the 3-way handshake and establishing a connection, the server then passes the queued data to the application layer, for a second time. The server cannot tell that it has already accepted the data because it maintains no state after a reboot. This violates the basic premise of T/TCP that it must remain completely backward compatible with TCP.

2.2 Solutions to T/TCP Security Problems

2.2.1. Dominance of TAO

Packet Sniffing

T/TCP is not subjective to packet sniffing more easily than TCP. But after being sniffed, it's indeed much easier for a cracker to fake T/TCP connections than TCP ones, due to TAO mechanism. However, any blame of T/TCP based on this notion is obviously baseless, because it's just as ridiculous as blaming cars based on a notion that a thief can slip away faster after stealing your car instead of your bicycle.

The Numbers Game

Fairly speaking, current TAO mechanism is really too simple. There is only one requirement to pass TAO test: received CC value is larger than cached CC value on the server. Even worse, CC is an unsigned 32-bit integer which is, according to RFC1644, initialized to 1 when the machine starts up. As a result, a fabricated large value (such as 2^32 –1) has a nearly definite chance of passing TAO test. This can be a serious security hole. Therefore, TAO mechanism should be enhanced.

Solution

1. Make CC a socket variable instead of a global one

In RFC1644, CC is defined as a global variable. Whenever a new connection has been established, no matter through which network interface, CC is incremented by 1. However, as we find out, by making CC a socket variable instead of a global one, several benefits can be achieved:

In order to maintain T/TCP's correctness, CC values must advance at a rate slower than 2^32-1 counts per 2MSL. Originally, with CC a global variable, 2^32-1/2MSL is the maximum number of total transactions per second on a machine. Now, with CC a socket variable, 2^32-1/2MSL is the maximum number of total transactions per second on a socket, or a pair of machines. Machines with busier network interfaces will gain great benefits in this way.

As a socket variable, CC increases in an anticipated manner: CC value increments by 1 only when a new connection has been established between the specific pair of machines. Thus, the new requirement to pass TAO test is that received CC value is exactly one larger than cached CC value on the server side. As a result, the possibility of success for "The Numbers Game" is greatly reduced.

2. Make CC randomly initialized

As stated above, by making CC a socket variable, TAO mechanism is very much enhanced. However, there exists one fact: usually the transaction rate between a specific pair of machines is low, which means the CC value for a certain socket can be small (say under 100) in most cases. Accordingly, "The Numbers Game" can have an updated version: starting from 1, try different CC values one by one. After 100 times or so, a T/TCP connection may be successfully faked.

In order to make passing TAO test more expensive, we highly recommend making CC randomly initialized, which means CC can start from any place from 1 to 2^32-1. This way, crackers will really have a tough time.

2.2.2. SYN Flooding

SYN Cookie

SYN cookie cannot be used with T/TCP since no TCP options can be saved when SYN cookie are in use. However, it's an inherent problem in SYN cookie and not a fault at all of T/TCP. With SYN cookie, no TCP extensions with TCP options are possible, not only T/TCP.

Generally, there are two ways to defend against SYN flooding: SYN cache and SYN cookie. With SYN cache, the amount of memory for states is reduced greatly, thus making SYN flooding much more expensive but never eradicating the possibility. SYN cookie eliminates the need for storing states with the price of losing any TCP option.

Besides, SYN cookie has other 2 drawbacks:

1. TCP requires unacknowledged data to be retransmitted. The server is supposed to retransmit the SYN.ACK packet before giving up and dropping the connection. When SYN.ACK arrives at the client but the ACK gets lost, there is a disparity about the establishment state between the client and server. Ordinarily, this problem can be solved by server's retransmission. But in the case of SYN cookie, there is no state kept on the server and retransmission is impossible.
2. SYN cookie has the property that the entire connection establishment is performed by the ACK packet from the client, independent of the preceding SYN and SYN.ACK packets. This opens the possibility of ACK flooding, in the hope that one will have the correct value to establish a connection. This also provides an approach to bypass firewalls which normally only filter packets with SYN bit set.

Due to the limits of SYN cookie listed above, a typical implementation method is to use SYN cache first and falls back to SYN cookie when a certain amount of memory has been allocated for state preservation.

Due the existential limits of SYN cookie, T/TCP's incompatibility with it should not be a guilt. The way to get around can be simple: only use SYN cache with T/TCP and turn off T/TCP when SYN cookie is enabled.

Failed TAO resulting in queued data.

According to RFC1644, when TAO fails, the protocol falls back to TCP but the data sent in the SYN packet is queued to decrease overhead and increase throughput. However, deliberately failing TAO during SYN flooding can be quite easy but destructive. In order for T/TCP to be safer and completely back compatible with TCP, the data should be discarded and sent again after the 3-way handshake succeeds.

2.2.3. Trust Relationship

This attack is completely based upon "Dominance of TAO". If TAO can't be dominated as claimed above, "Trust Relationship" attack on T/TCP is, at least, as difficult as on standard TCP.

2.2.4. Duplicate Delivery

Duplicate delivery is an inherent problem in RPC, which is called "Server Crash". Fairly speaking, T/TCP shouldn't be blamed for it. Let's take a closer look at this problem.

In Remote Procedure Call (RPC), requests can be divided into two categories: idempotent or non-idempotent. Idempotent requests can be executed repeatedly without errors while non-idempotent ones can't. An example of non-idempotent request is withdrawing 1 million USD from your account. You surely don't want it to happen more times than you want.

"Server Crash" can happen in two ways:

- After receiving and before execution
- After execution and before responding

Unfortunately, the client can't tell any difference between those two kinds of server crashes. All it knows is response timeout.

Facing this problem, 3 choices are available:

1. The client waits until the server finishes reboot, after which the client resends the request. This technique is called "at least once semantics".
2. The client gives up and reports failure. This technique is called "at most once semantics".
3. No special action is taken. The client gets no help or guarantee. On the server side, RPC may be executed many times, or not at all. Obviously, the merit is ease of implementation.

The above 3 choices are not ideal. The ideal one is "exactly once semantics" which is very difficult to realize.

Possible solutions

1. The client assigns a sequence number to every packet it transmits. And the server stores the latest sequence number it receives from the client. Thus, the server can easily tell whether a packet from the client is a retransmitted one.
2. The client sets a certain bit when marking a packet retransmitted. That bit can work as a warning for the server: additional cares need to be taken for the packet.

In real-life implementations, it's the application layer that shoulders the responsibility. Generally speaking, idempotent requests are those inseparable from their contexts. If the application sees a response timeout after sending a request, it can take extra measures to check whether the request contexts have been changed and make a decision (e.g. retransmitting the request or sending a updated request) accordingly.

3 Methods to Achieve Better Compatibility with TCP

3.1 Discard data in SYN when TAO fails

When a T/TCP-enabled client sends initial SYN (with data) to a remote server, 3WHS can be carried out in two situations: the server doesn't support T/TCP, or the server supports T/TCP but TAO fails. Most current TCP implementations discard data sent with initial SYN so the data should be sent again after the 3-way handshake succeeds. T/TCP-enabled server should also discard data and behave exactly the same as a T/TCP-disabled server. Otherwise, the client will need to differentiate the above two situations and decide whether to retransmit the data. Besides, security can also be enhanced this way.

3.2 Turn on or off T/TCP globally or locally, at any time

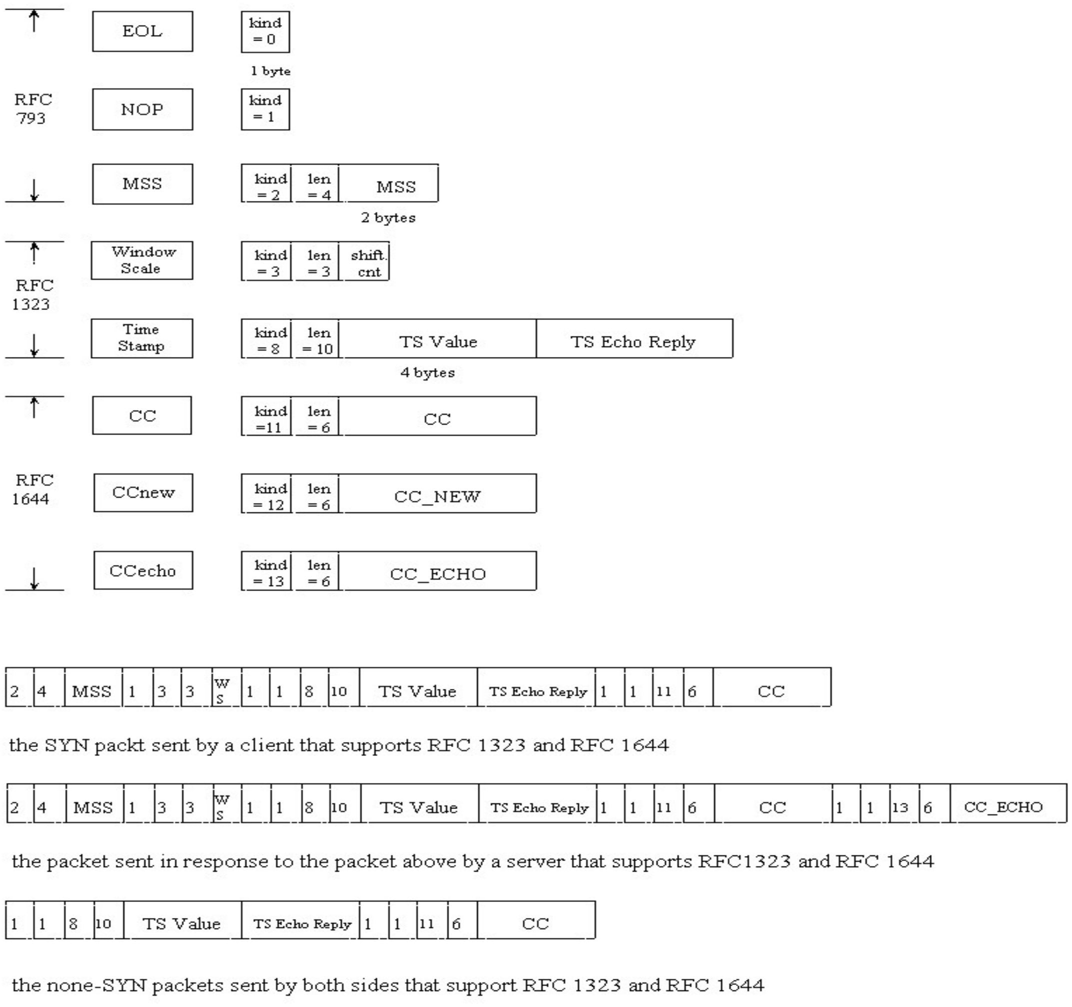
In our implementation, a global flag co-exists with socket-level flags. The global flag enables (when on) or disables (when off) T/TCP features for all the newly created sockets. In other words, the global flag acts as a default value. Currently, its state is a kernel compilation choice. By contrast, socket-level flags can override the global flag. They can be turned on or off by calling system function **setsockopt()** with option name "TTCP" and option value 1(on) or 0(off). Enabling or disabling T/TCP during run-time introduces greater flexibility and better compatibility with TCP.

4 Implementation of ET/TCP in Linux Kernel 2.4.2

4.1 New TCP Options

Notes

- CC_ECHO option is included only if CC option is included.
- CC_ECHO option is included only in SYN/ACK packets if both sides support T/TCP.



RFC 793
EOL
kind = 0
1 byte
NOP
kind = 1
MSS
kind = 2
len = 4
MSS
2 bytes
RFC 1323
Window Scale
kind = 3
len = 3
shift. cnt
Time Stamp
kind = 8
len = 10
TS Value
TS Echo Reply
4 bytes
RFC 1644
CC
kind =11
len = 6
CC
CCnew
kind = 12
len = 6
CC_NEW
CCecho
kind = 13
len = 6
CC_ECHO
2 4 MSS 1 3 3 WS 1 1 8 10 TS Value TS Echo Reply 1 1 11 6 CC
the SYN packt sent by a client that supports RFC 1323 and RFC 1644
2 4 MSS 1 3 3 WS 1 1 8 10 TS Value TS Echo Reply 1 1 11 6 CC 1 1 13 6 CC_ECHO
the packet sent in response to the packet above by a server that supports RFC1323 and RFC 1644
1 1 8 10 TS Value TS Echo Reply 1 1 11 6 CC
the none-SYN packets sent by both sides that support RFC 1323 and RFC 1644

Figure 4: New TCP Options

4.2 New Structures and Macros

in file "include/linux/tcp.h"

```
/* a macro used to increment ttcp_gen and
ensures ttcp_gen to be non-zero when it's
rounded-up */


#define CC_INC(a) (++(a) == 0 ? ++(a) : (a))


in file "include/net/tcp.h"
```

```
/* constants used in T/TCP options */


#define TCPOLEN_CC 6

#define TCPOLEN_CC_NEW 6

#define TCPOLEN_CC_ECHO 6

#define TCPOLEN_CC_ALIGNED 8
```

in struct dst_entry{} in file “**include/net/dst.h**”

```
/* The per-host cache tao_cc is the CC value
contained in the last SYN (no ACK) packet
received from the remote host; tao_ccsent is
the CC value contained in the last SYN (no ACK)
packet sent to the remote host; ttcp_gen is a
socket-level non-zero variable that’s
incremented everytime a new connection has been
started; */


struct rt_tao {

__u32 tao_cc;

__u32 tao_ccsent;

_u32 ttcp_gen;

} tao;
```

in struct tcp_opt{} in file “**include/net/sock.h**”

```
/* The per-connection cache t_duration shows
how long this connection has been in existence
since active open; cc_send is the CC value
contained in every packet sent to the remote
host on this connection; cc_recv is the CC
value contained in every packet received from
the remote host on this connection */


struct tcp_opt_cc {

unsigned long t_duration;

__u32 cc_send;

__u32 cc_recv;

} cc_cache;


/* two booleans for T/TCP states */


char TTCP_SENDSYN, TTCP_SENDFIN;


/* values of T/TCP options parsed from incoming
packets */


__u32 cc;

__u32 cc_new;

__u32 cc_echo;


/* a per-connection flag to enable or disable
T/TCP */


int do_rfc1644;
```

in file “**include/net/sock.h**”

```
/* macros to set two booleans for T/TCP states
*/

#define TTCP_SENDSYN_ON(sk)     ((sk)-
>tp_pinfo.af_tcp.TTCP_SENDSYN = 1)

#define TTCP_SENDFIN_ON(sk)     ((sk)-
>tp_pinfo.af_tcp.TTCP_SENDFIN = 1)

#define TTCP_SENDSYN_OFF(sk)    ((sk)-
>tp_pinfo.af_tcp.TTCP_SENDSYN = 0)

#define TTCP_SENDFIN_OFF(sk)    ((sk)-
>tp_pinfo.af_tcp.TTCP_SENDFIN = 0)


/* macros to test T/TCP states */


#define TTCP_SYN_SENT(sk)       ((sk)->state ==
TCP_SYN_SENT && (!(sk)-
>tp_pinfo.af_tcp.TTCP_SENDSYN) && (sk)-
>tp_pinfo.af_tcp.TTCP_SENDFIN)

#define TTCP_SYN_RECV(sk)       ((sk)->state ==
TCP_SYN_RECV && (!(sk)-
>tp_pinfo.af_tcp.TTCP_SENDSYN) && (sk)-
>tp_pinfo.af_tcp.TTCP_SENDFIN)

#define TTCP_ESTABLISHED(sk)    ((sk)->state ==
TCP_ESTABLISHED && (!(sk)-
>tp_pinfo.af_tcp.TTCP_SENDFIN) && (sk)-
>tp_pinfo.af_tcp.TTCP_SENDSYN)

#define TTCP_CLOSE_WAIT(sk)     ((sk)->state ==
TCP_CLOSE_WAIT && (!(sk)-
>tp_pinfo.af_tcp.TTCP_SENDFIN) && (sk)-
>tp_pinfo.af_tcp.TTCP_SENDSYN)

#define TTCP_LAST_ACK(sk) ((sk)->state ==
TCP_LAST_ACK && (!(sk)-
>tp_pinfo.af_tcp.TTCP_SENDFIN) && (sk)-
>tp_pinfo.af_tcp.TTCP_SENDSYN)

#define TTCP_FIN_WAIT1(sk)      ((sk)->state ==
TCP_FIN_WAIT1 && (!(sk)-
>tp_pinfo.af_tcp.TTCP_SENDFIN) && (sk)-
>tp_pinfo.af_tcp.TTCP_SENDSYN)

#define TTCP_CLOSING(sk) ((sk)->state ==
TCP_CLOSING && (!(sk)-
>tp_pinfo.af_tcp.TTCP_SENDFIN) && (sk)-
>tp_pinfo.af_tcp.TTCP_SENDSYN)
```

4.3 TIME-WAIT Truncation

Step One

In order to record the start point of a connection, we have added the following line:

```
tp->cc_cache.t_duration = jiffies;
```

to function `ttcp_connect` (**`net/ipv4/tcp_output.c`**)

Step Two

When a sock enters TIME_WAIT state, it calls function **tcp_time_wait()** in **net/ipv4/tcp_minisocks.c**. In that function, we check whether the connection has been in existence for a period shorter than TTCP_MSL. If so, TIME_WAIT truncation is valid. And we set the TIME_WAIT period to 12*TTCP_MSL (12s) instead of TCP_TIMEWAIT_LEN (60s).

Step Three

The client can actually call **bind()** to reuse one particular local port intentionally. In this process, **tcp_v4_get_port()** is called**.** We have modified the function to enable it to make a particular port available when the port is currently owned by one sock in TIME_WAIT state.

4.4 Sending Function Call Tree

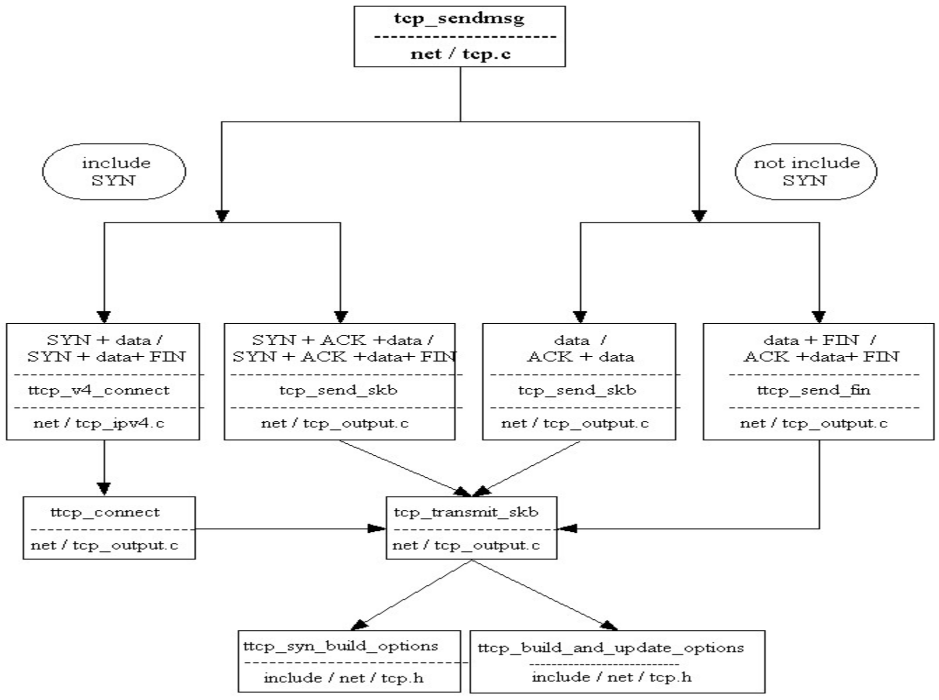


Figure 5: Sending Function Call Tree

4.5 Receiving Function Call Tree

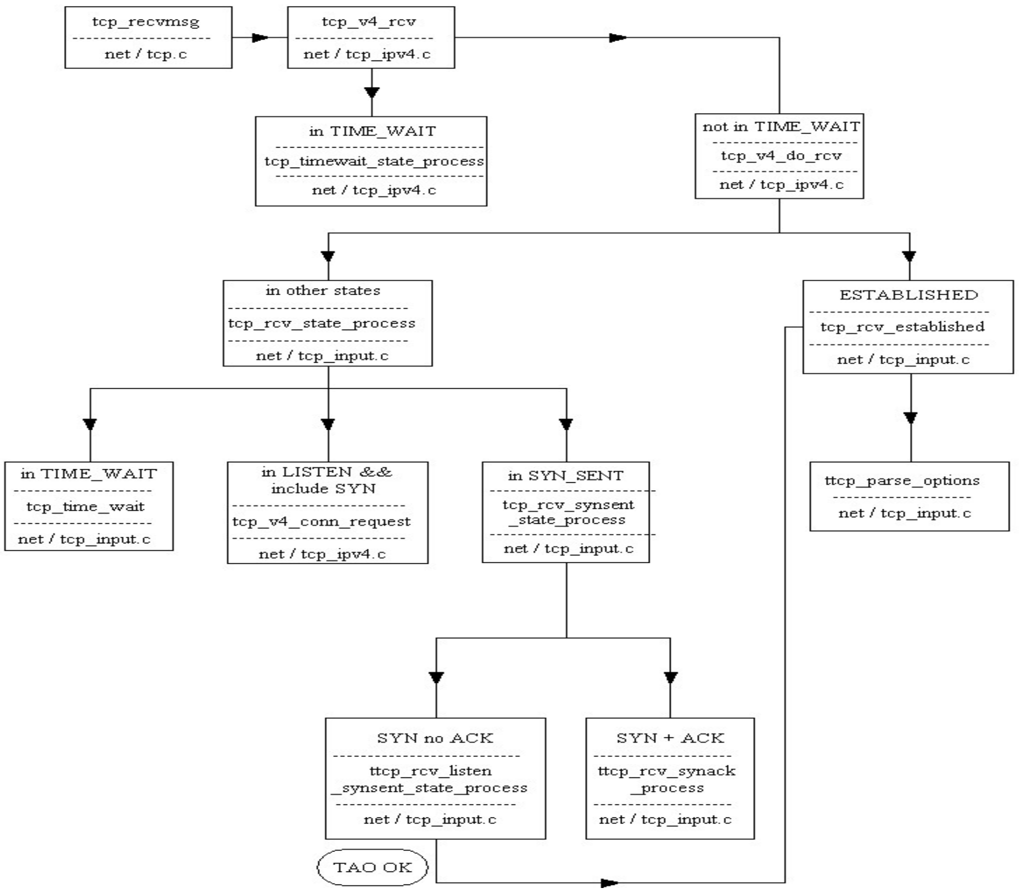


Figure 6: Receiving Function Call Tree

4.6 Write T/TCP-enabled Client and Server Codes

Client Codes

T/TCP client doesn't use "connect( )", but uses "sendto( )" directly. "sendto( )" connects to the server and sends a request simultaneously with only one packet. Pay attention: we choose a new flag "MSG_EOF" as the 4th argument of "sendto( )", which signals to the kernel the end of the user's data. As a result, this special "sendto( )" sends one packet with SYN + data + FIN. In other words, "sendto( )" performs the functions of "connect( )", "write( )" and "shutdown( )".

Server Codes

There is only one difference: T/TCP server uses "send( )" instead of "write( )". Thus, by setting "send( )" 4th argument to MSG_EOF, the server sends the response together with FIN flag in one packet.

References

[1] Richard W. Stevens. *TCP/IP Illustrated Volume 1: The Protocols*

[2] Richard W. Stevens. *TCP/IP Illustrated Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocol4*

[3] Bob Braden. *RFC1379: Extending TCP for Transactions -- Concepts*

[4] Bob Braden. *RFC1644: T/TCP -- TCP Extensions for Transactions Functional Specification*

[5] DARPA Internet Program. *RFC793: Transmission Control Protocol Specification*

[6] Jonathan Lemon. *Resisting SYN flood DoS attacks with a SYN cache*

[7] Bernstein D.J. *SYN cookies* *http://cr.yp.to/syncookies.html*

[8] Borman D. *TCP Implementation posting* *http://www.kohala.com/start/borman.97jun06.txt*

[9] Michael Mansberg. *TCP/IP for Transactions* *http://www.embedded.com*