Implementation of Transaction TCP in Linux Kernel 2.4.2

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

CONTENTS

FIGURE

1 Introduction

1.1 WHAT IS "T/TCP For Linux"

"T/TCP For Linux" is an open-source project launched at http://ttcplinux.sourceforge.net that focuses on the implementation and integration of T/TCP (Transaction TCP) into Linux kernels.

1.2 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 exchanging. T/TCP greatly decreases the overhead standard TCP introduces when dealing with transaction-oriented connections.

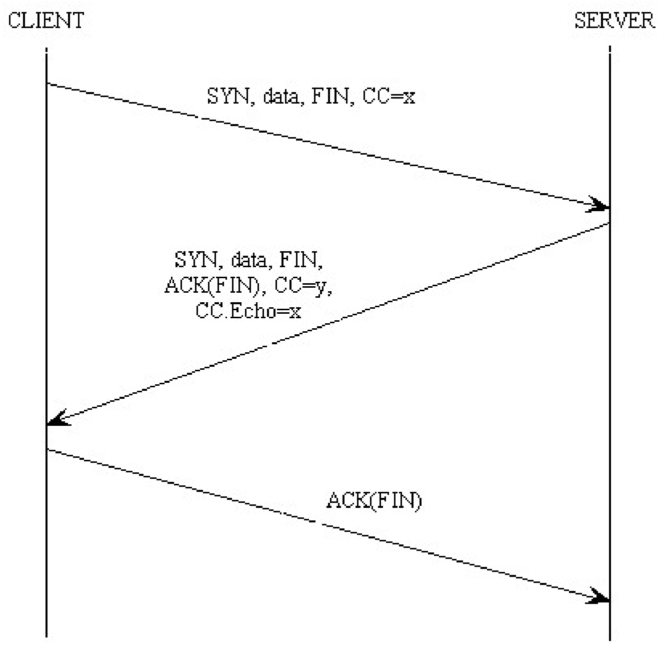


Figure 1: A typical T/TCP connection

1.3 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.

1.4 Possible Applications of T/TCP

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 reassembly 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 transmission 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 reliability in delivering. 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.

1.4.4.1 Memory Use

Because of T/TCP's TIME_WAIT truncation feature, an application can re-use a previous connection's local TCP port immediately, thus avoiding an additional TCP control block allocation necessary with standard TCP.

Even if an application does not re-use the same local TCP port, since T/TCP's TIME_WAIT state duration is usually very short compared to that of standard TCP, the constriction on transaction rate due to a limited simultaneous connection capacity is much less severe.

As for UDP, since it is connectionless and, therefore, without a connection state to be maintained, memory size/usage is not much of an issue.

1.4.4.2 Network Bandwidth

The initial T/TCP transaction between a pair of hosts requires six segments. Subsequent transactions between the same pair of hosts require just three segments.

By contrast, a minimal transaction using a standard TCP implementation usually requires nine TCP segments, even though only two of them carry application data. The seven extra segments are for the three-way handshake that opens and closes the connection and an acknowledgment, which is a lot of overhead.

As for UDP, a minimal transaction using UDP requires only two UDP datagrams, one in each direction. Thus, UDP minimizes the load on the network.

1.4.4.3 Response Time

The initial T/TCP transaction between a pair of hosts has the same response time as TCP (2*RTT + SPT). All subsequent transactions between the same pair of hosts have a response time that is similar to that of UDP (RTT + SPT).

1.4.4.4 Reliability

Both standard TCP and T/TCP are reliable transport layer protocols.

As for UDP, it is not a reliable transport protocol. If a UDP datagram is dropped due to congestion, the transaction does not complete. If it is necessary for the data transfer to be reliable, this reliability must be built into the application layer.

1.4.4.5 Interoperability

An application may be intended to communicate with peer applications running on various types of hardware and software platforms. This interoperability is easier to achieve with a protocol stack that conforms to widely accepted and implemented standards than by using a proprietary protocol stack, or a new one that has not yet become widespread.

Because of T/TCP’s backward compatibility with standard TCP, its use poses little or no obstacle to interoperability. An exception might be if a client-side T/TCP attempts to communicate with a defective standard TCP that cannot silently ignore the CC option. Normally, T/TCP automatically and silently falls back to standard TCP when encountering standard TCP counterpart. While the performance benefits of T/TCP could not be realized in this “interoperability” mode, at least the transactions themselves could still take place.

Obviously, standard TCP poses no obstacles to interoperability.

As for UDP, if reliability is required, a proprietary mechanism must be used on top of UDP, which could present an obstacle to interoperability.

2 T/TCP Design Analyses

2.1 TCP Accelerated Open (TAO)

2.1.1 Overview

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, using a mechanism that we call TCP Accelerated Open (TAO). Under the TAO mechanism, 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 values that it has received from each different client host. If an initial SYN segment (i.e., a segment containing 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's 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.

2.1.2 Protocol Correctness

CC values are 32-bit integers.

The essential requirement for correctness of T/TCP is this:

[R1] CC values must advance at a rate slower than 2^32-1 counts per 2MSL

[R2] where MSL denotes the maximum segment lifetime in the Internet.

The requirement [R1] is easily met with a 32-bit CC. For example, it will allow 10^7 transactions per second with the very liberal MSL of 1000 seconds [RFC-1379]. This is well in excess of the transaction rates achievable with current operating systems and network latency. Suppose x (n) is the most recent acceptable CC value carried in an initial SYN segment from a remote host and is cached by the server. If the server host receives a SYN segment containing a CC option with value y where y > x (n), that SYN must be newer; an antique duplicate SYN with CC value greater than x (n) must have exceeded its MSL and vanished. Hence, monotonic CC values and the TAO test prevent erroneous replay of antique SYNs.

There are two possible reasons for a client to generate non-monotonic CC values:

1. The client may have crashed and restarted, causing the generated CC values to jump backwards

2. The generated CC values may have wrapped around the finite space.

Wraparound may occur because CC generation is global to all connections. Suppose that host A sends a transaction to B, then sends more than 2^32-1 transactions to other hosts, and finally sends another transaction to B. From B's viewpoint, CC will have jumped backwards relative to its cached value. In either of these two cases, the server may see the CC value jump backwards only after an interval of at least MSL since the last SYN segment from the same client host. In case (a), client host restarts, this is because T/TCP retains TCP's explicit "Quiet Time" of an MSL interval. In case (b), wrap around, [R1] ensures that a time of at least MSL must have passed before the CC space wraps around. Hence, there is no possibility that a TAO test will succeed erroneously due to either cause of non-monotony; i.e., there is no chance of replays due to TAO.

However, although CC values jumping backwards will not cause an error, they may cause performance degradation due to unnecessary 3WHS's. This results from the generated CC values jumping backwards through averagely half their range, so that all succeeding TAO tests fail until the generated CC values catch up with the cached value. To avoid this degradation, a client host sends a CC.NEW option instead of a CC option in the case of either system restart or CC wraparound. Receiving CC.NEW forces a 3WHS, but when this 3WHS completes successfully the server cache is updated to the new CC value. To detect CC wraparound, the client must cache the last CC value it sent to each server. It therefore maintains cache.Ccsent[B] for each server B. If this cached value is undefined or if it is larger than the next CC value generated at the client, then the client sends a CC.NEW instead of a CC option in the next SYN segment.

2.2 TIME-WAIT Truncation

2.2.1 Why TIME-WAIT

TIME-WAIT is used for two reasons:

1. Full-duplex connection close

2. Protection against old duplicate segments

2.2.2 Overview

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 delay to be safely truncated.

2.2.3 Protocol Correctness

2.2.3.1 Full-duplex connection close

Suppose the client receives "...FIN, ack(FIN)..." from the server and therefore turns into TIME-WAIT state. Afterwards, it sends the last ACK packet, "ack(FIN)", to the server. This packet gets lost. Before the client receives server's retransmitted packet, it truncates TIME-WAIT state and starts another connection by sending to the server a new initial SYN with updated CC value. When the server receives this packet, TAO is successful. Obviously, this new SYN packet acts as an implicit last ACK packet from the client. As a result, on the server side, the old connection is closed and the new connection is established. Besides, when the client finally receives the server's retransmitted packet, it gets discarded because a new connection between the pair has already been started.

2.2.3.2 Protection against antique duplicates

For short connections (e.g., transactions), the CC values assigned to each direction of the connection can be used to protect against antique duplicate non-SYN segments. Here we define "short" as lasting for a period less than MSL.

Suppose that there is a connection that uses the CC values TCB.CCsend = x and TCB.CCrecv = y and is opened at time point "time_start". By the requirement [R1], neither x nor y can be reused for a new connection from the same remote host for a time at least 2*MSL; i.e., x or y can be reused for a new connection from the same remote host only after time point "time_start + 2MSL" .

If the connection has been in existence for a time (say, "time_length") less than MSL, then, all non-SYN segments with x or y will vanish before time point "time_start + time_length + MSL".

Since "time_length" < MSL, all antique duplicates with that CC value must vanish before it is reused. Thus, for "short" connections we can guard against antique non-SYN segments by simply checking the CC value in the segment against TCB.CCrecv. Note that this check does not use the monotonic property of the CC values, instead only the one that they do not cycle in less than 2*MSL. Again, the quiet time at system restart protects against errors due to crash with loss of state.

3 T/TCP Implementation in Linux Kernel 2.4.2

3.1 Linux Kernel TCP/IP Analyses

3.1.1 Stack Layers, Sending and Receiving Flow graphs

3.1.1.1 TCP Stack Layers

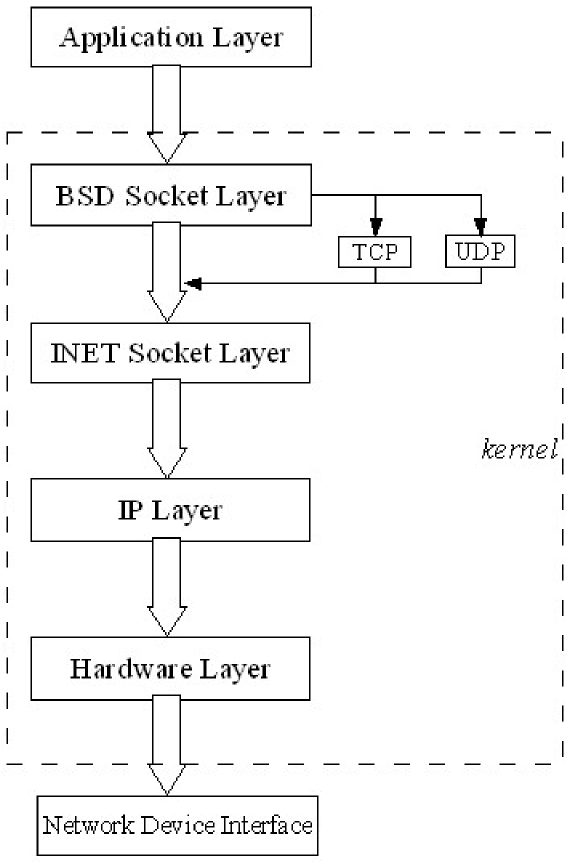


Figure 2: Stack layers

Application Layer

At application layer, what users directly manipulate is socket{} file descriptor. File system defines common interfaces for different types of files. Those interfaces in turn invoke certain system calls. Thus, any call to a particular interface of socket{} file descriptor is turned into a call to a particular function of BSD Socket, which lies at the BSD Socket layer.

BSD Socket Layer

At BSD Socket layer, the manipulation target is socket{}. Every created socket{} represents one network connection, belongs to one particular network address family and chooses operations accordingly, such as deciding whether to enter INET Socket layer or not. Data at this layer are stored in msghdr{}.

INET Socket Layer

At INET Socket layer, the manipulation target is sock{}. A sock{} is either connection-oriented (TCP) or connectionless (UDP). Data at this layer are stored in sk_buff{}.

IP Layer

From INET Socket layer to IP layer lies the routing process: during sending, deciding which network device interface to use and what's the address of the next hop; during receiving, deciding whether to deliver received packets to upper layers or to act as an intermediary.

Hardware Layer

This layer mainly consists of network device interface drivers.

Network Device Interface

Here is the hardware.

3.1.1.2 TCP Sending Flow Graph

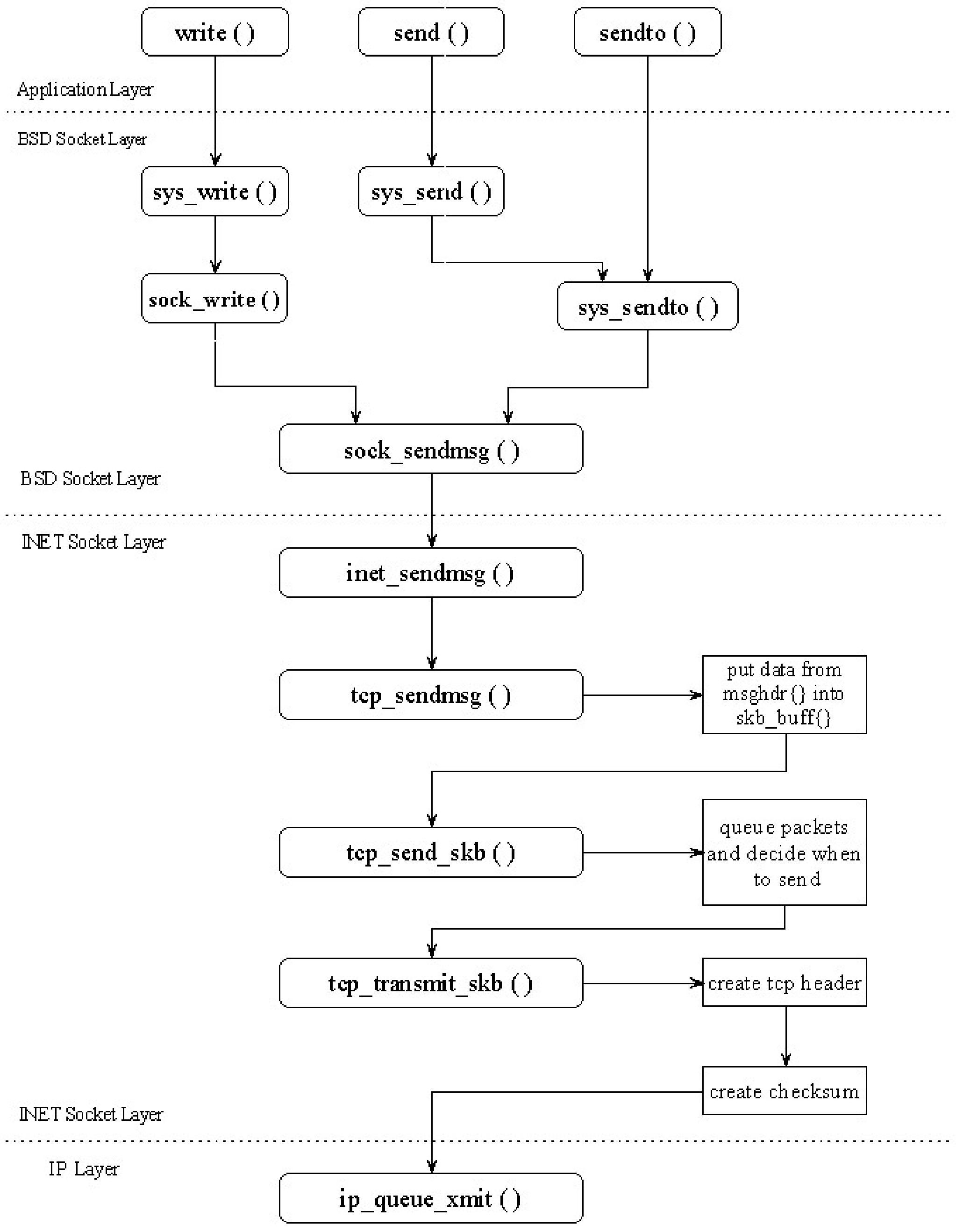


Figure 3: Sending Flow Graph

fs/read_write.c

asmlinkage ssize_t sys_write(unsigned int fd, const char *buf, size_t count)

net/socket.c

asmlinkage long sys_send(int fd, void *buff, size_t len, unsigned flags) asmlinkage long sys_sendto(int fd, void *buff, size_t len, unsigned flags, struct sockaddr *addr, int addr_len)

static ssize_t sock_write(struct file *file, const char *ubuf, size_t size, loff_t *ppos)

int sock_sendmsg(struct socket *sock, struct msghdr *msg, int size)

net/ipv4/af_inet.c

int inet_sendmsg(struct socket *sock, struct msghdr *msg, int size, struct scm_cookie *scm)

net/ipv4/tcp.c

int tcp_sendmsg(struct sock *sk, struct msghdr *msg, int size)

net/ipv4/tcp_output.c

void tcp_send_skb(struct sock *sk, struct sk_buff *skb, int force_queue, unsigned cur_mss)

int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb)

net/ipv4/ip_output.c

int ip_queue_xmit(struct sk_buff *skb)

3.1.1.3 Receiving Flow Graph

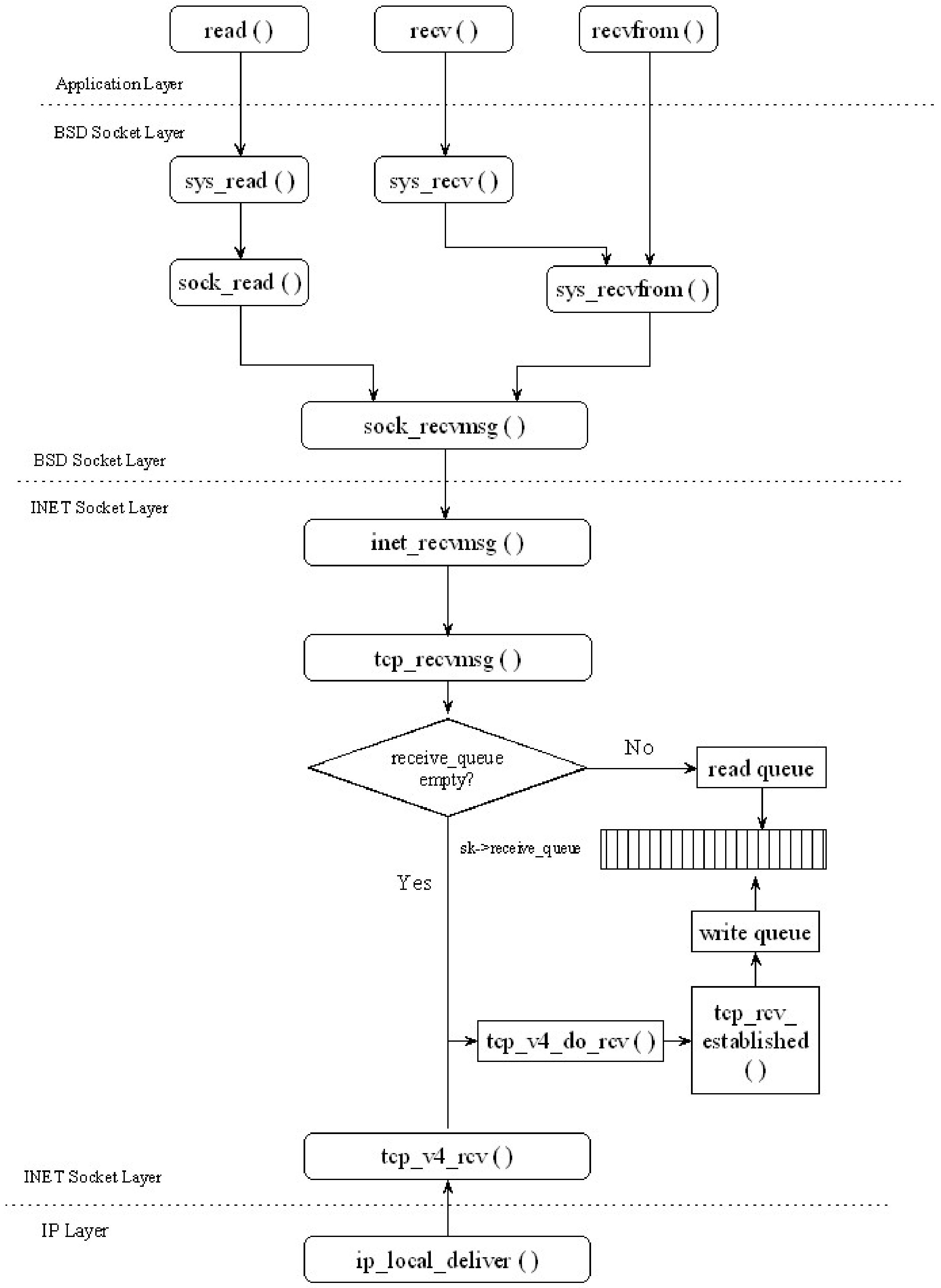


Figure 4: Receiving Flow Graph

fs/read_write.c

asmlinkage ssize_t sys_read(unsigned int fd, char *buf, size_t count)

net/socket.c

asmlinkage long sys_recv(int fd, void *ubuf, size_t size, unsigned flags)

asmlinkage long sys_recvfrom(int fd, void *ubuf, size_t size, unsigned flags, struct sockaddr *addr, int *addr_len)

static ssize_t sock_read(struct file *file, char *ubuf, size_t size, loff_t *ppos)

int sock_redvmsg(struct socket *sock, struct msghdr *msg, int size, int flags)

net/ipv4/af_inet.c

int inet_recvmsg(struct socket *sock, struct msghdr *msg, int size, int flags, struct scm_cookie *scm)

net/ipv4/tcp.c

int tcp_recvmsg(struct sock *sk, struct msghdr *msg, int len, int nonblock, int flags, int *addr_len)

net/ipv4/tcp_ipv4.c

int tcp_v4_rcv(struct sk_buff *skb, unsigned short len)

int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)

net/ipv4/ip_input.c

int tcp_rcv_established(struct sock *sk, struct sk_buff *skb, struct tcphdr *th, unsigned len)

net/ipv4/ip_input.c

int ip_local_deliver(struct sk_buff *skb)

3.1.2 State Transition Flow graph

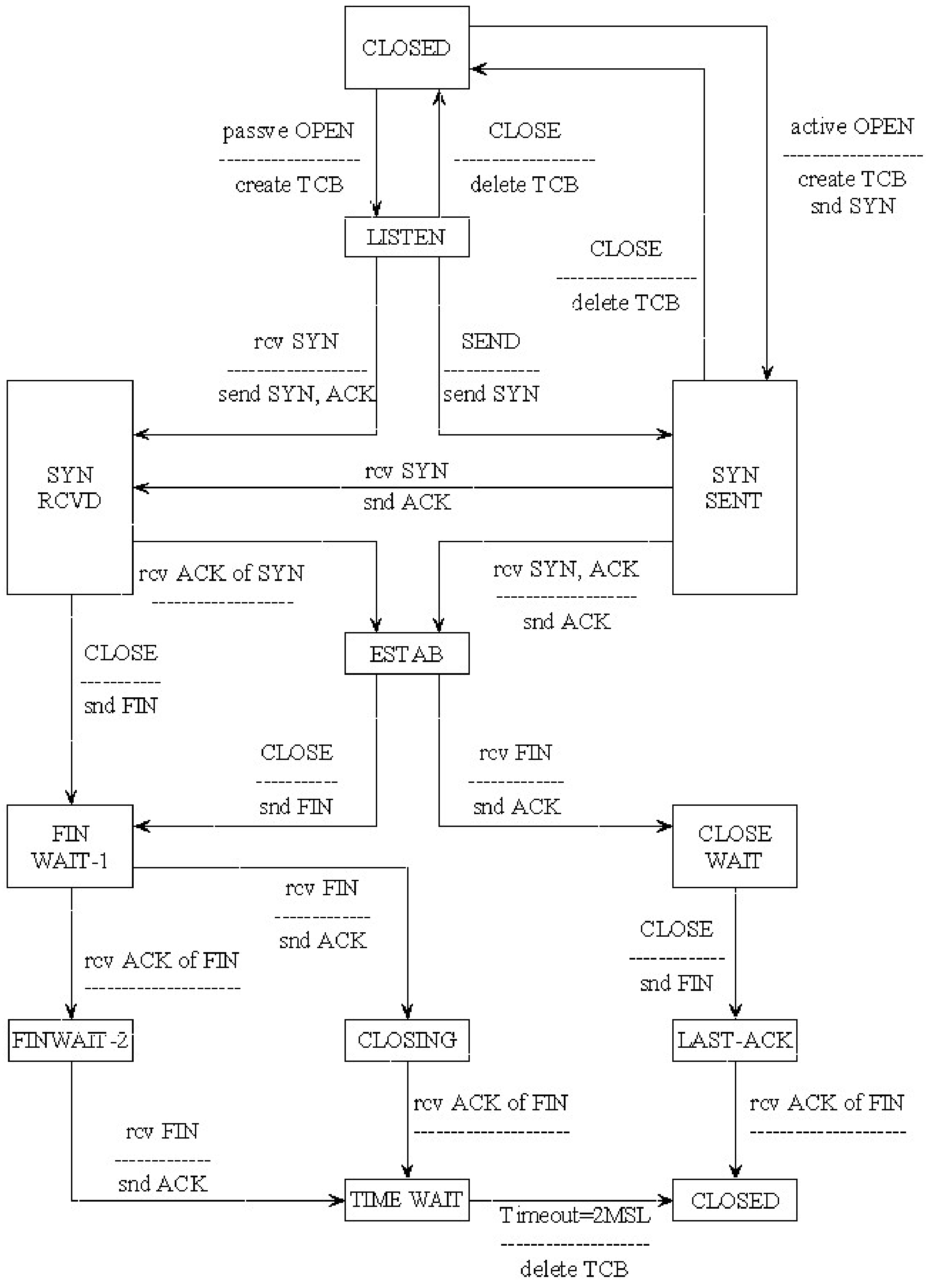


Figure 5: TCP State Transition

3.1.3 Important Structures

sock{}

This structure is the basis for implementation of the INET socket layer. It’s responsible for the sending and receiving of network packets.

sk_buff{}

This is the structure where data is saved in INET socket layer.

tcp_opt{}

This is the structure where lots of TCP options and TCP connection parameters are saved.

proto{}

This structure is the function set used in sock{}. According to different protocols, different proto{} function sets are used.

tcphdr{}

This structure is where the head buff (with no options) of a TCP package is specified.

tcp_skb_cb{}

This structure contains varieties of TCP parameters in TCP control block (TCB), which decides what should be put into tcphdr.

3.2 T/TCP Implementation

3.2.1 T/TCP State Transition

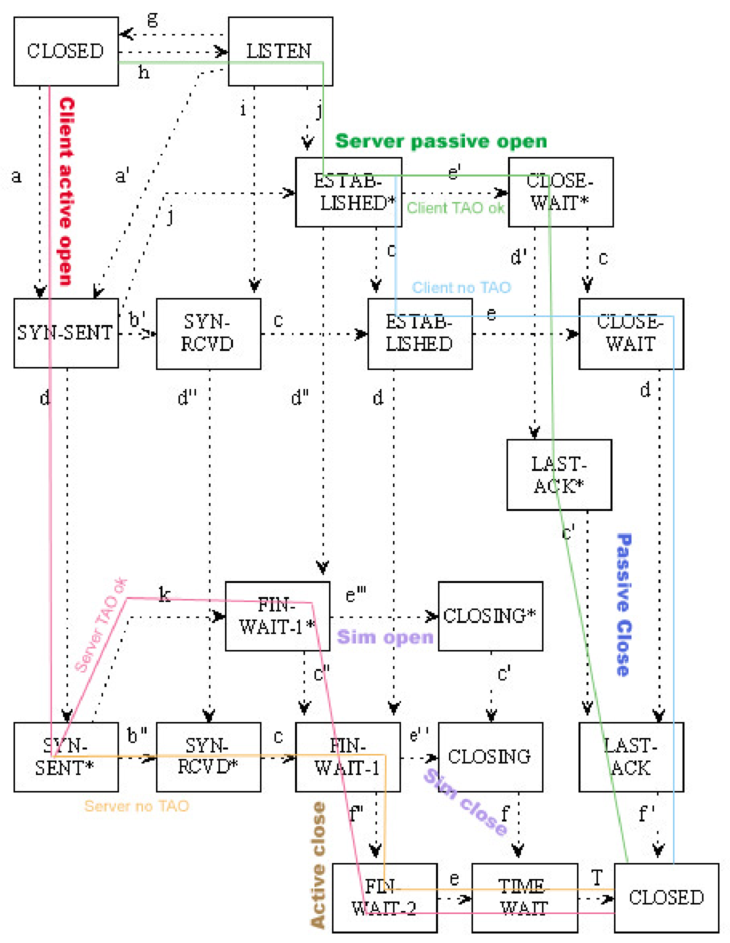


Figure 6: T/TCP State Transition

Definitions of State Transitions

Label	**Event / Action**	**Label**	**Event / Action**
a	Active OPEN / create TCB, snd SYN	e’	rcv FIN / snd SYN,ACK(FIN)
a’	Active OPEN / snd SYN	e”	rcv FIN / snd FIN,ACK(FIN)
b	rcv SYN [no TAO] / snd ACK(SYN)	e””	rcv FIN / snd SYN,FIN,ACK(FIN)
b’	rcv SYN [no TAO] / snd SYN,ACK(SYN)	f	rcv ACK(FIN) /

b”	rcv SYN [no TAO] / snd SYN,FIN,ACK(SYN)	f ‘	rcv ACK(FIN) / delete TCB
c	rcv ACK(SYN)	f’	rcv ACK(FIN) / ACK(SYN)
c’	rcv ACK(SYN) / snd FIN	g	CLOSE / delete TCB
c”	rcv ACK(SYN) / snd FIN,ACK(SYN)	h	passive OPEN / create TCB
d	CLOSE / snd FIN	i (=b’)	rcv SYN [no TAO] / snd SYN,ACK(SYN)
d’	CLOSE / snd SYN,FIN,ACK(FIN)	j	rcv SYN [TAO OK] / snd SYN,ACK(SYN)
d”	CLOSE / snd SYN,FIN,ACK(SYN)	k	rcv SYN [TAO OK] / snd SYN,FIN,ACK(SYN)
e	Rcv FIN / snd ACK(FIN)	T	timeout=2MSL / delete TCB

T/TCP requires new connection states and state transitions. Each of the new states in the figure above bears a starred name, created by suffixing a star onto a standard TCP state. Each “starred” state bears a simple relationship to the corresponding “unstarred” state:

- SYN-SENT* and SYN-RECEIVED* differ from the SYN-SENT and SYN-RECEIVED state, respectively, in recording the fact that a FIN needs to be sent.
- The other starred states indicate that the connection is half-synchronized (hence, a SYN bit needs to be sent).

This simple correspondence leads to an alternative state model, which makes it easy to incorporate the new states in an existing implementation. Each state in the extended FSM (Finite State Machine) is defined by the triplet (old_state, SENDSYN, SENDFIN) where ‘old_state’ is a standard TCP state and SENDFIN and SENDSYN are Boolean flags. The SENDFIN flag is turned on (on the client side) by a SEND (... EOF=YES) call, to indicate that a FIN should be sent in a state which would not otherwise send a FIN. The SENDSYN flag is turned on when the TAO test succeeds to indicate that the connection is only half synchronized; as a result, a SYN will be sent in a state that would not otherwise send a SYN.

Here is a more complete description of these Boolean variables:

- **SENDFIN**

SENDFIN is turned on by the SEND(...EOF=YES) call, and turned off when FIN-

WAIT-1 state is entered. It may only be on in SYN-SENT* and SYN-RECEIVED* states.

SENDFIN has two effects. First, it causes a FIN to be sent on the last segment of data from the user. Second, it causes the SYN-SENT* and SYN-RECEIVED* states to transition directly to FIN-WAIT-1, skipping ESTABLISHED state.

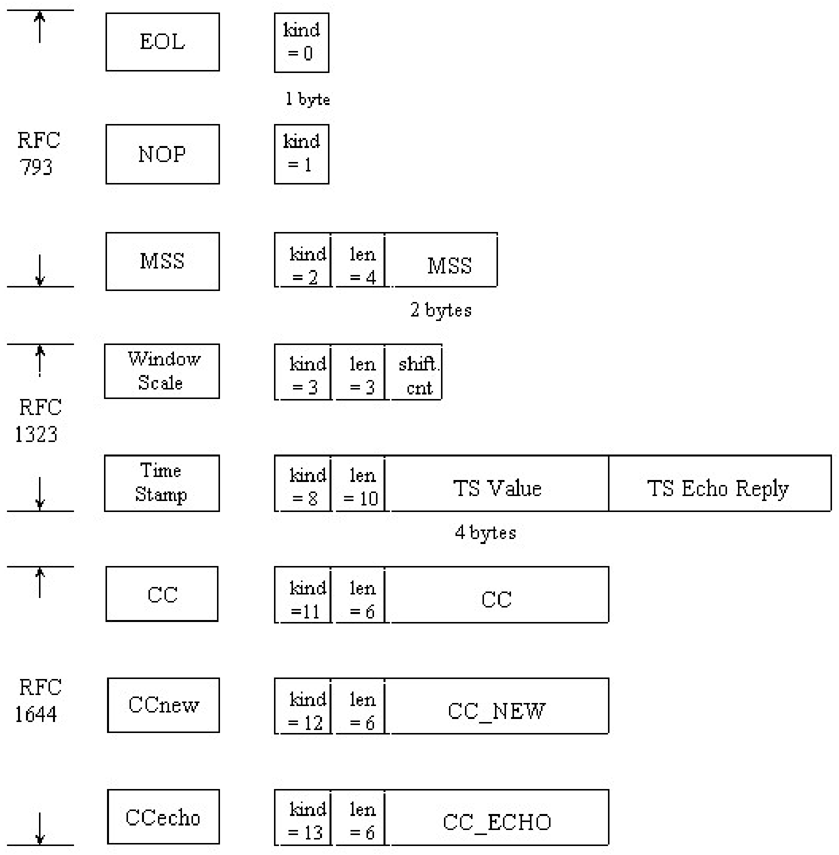
- **SENDSYN**

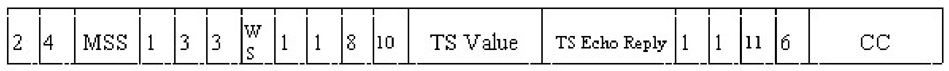
SENDSYN is turned on when an initial SYN segment is received and passes the TAO test. SENDSYN is turned off when the SYN is acknowledged (specifically, when there is no RST or SYN bit and SEG.UNA < SND.ACK).

SENDSYN has three effects. First, it causes the SYN bit to be set in segments sent with the initial sequence number (ISN). Second, it causes a transition directly from LISTEN state to ESTABLISHED*, if there is no FIN bit, or otherwise to CLOSE-WAIT*. Finally, it allows data to be received and processed (passed to the application) even if the segment does not contain an ACK bit.

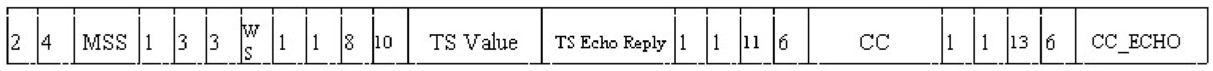
3.2.2 New TCP Options

- 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.

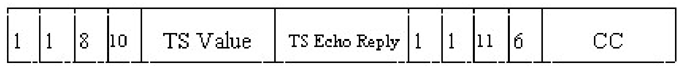



2 4 MSS 1 3 3 WS 1 1 8 10 TS Value TS Echo Reply 1 1 11 6 CC

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



the none-SYN packets sent by both sides that support RFC 1323 and RFC 1644

3.2.3 New Structures and Macros

include/linux/tcp.h

/* a global non-zero variable that's incremented everytime a new connection has been started */

_u32 ttcp_gen;

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

#define CC_INC© (++© == 0 ? ++© : ©)
```

include/net/tcp.h

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

#define TCPOLEN_CC 6

#define TCPOLEN_CC_NEW 6

#define TCPOLEN_CC_ECHO 6

#define TCPOLEN_CC_ALIGNED 8
```

include/net/dst.h in struct dst_entry{}

```
/* 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 */

struct rt_tao {

__u32 tao_cc;

__u32 tao_ccsent;

} tao;
```

include/net/sock.h in struct tcp_opt{}

```
/* 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;


/* 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)

3.2.4 Sock Initialization and Option Configuration

Initialization:

Sock is initialized in function **tcp_v4_init_sock() (net/ipv4/tcp_ipv4.c)**. T/TCP related initialization is as follows:

tp->do_rfc1644 = 1;

tp->cc_cache.cc_recv = 0;

tp->cc_cache.cc_send = ttcp_gen;

CC_INC(ttcp_gen);

Option Configuration:

In default, all T/TCP functions are turned on with **tp->do_rfc1644=1** (see above). However, you can turn it off by calling **setsockopt()** with option name "TTCP" and option value "0".

3.2.5 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() (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.

3.2.6 Sending Function Call Tree

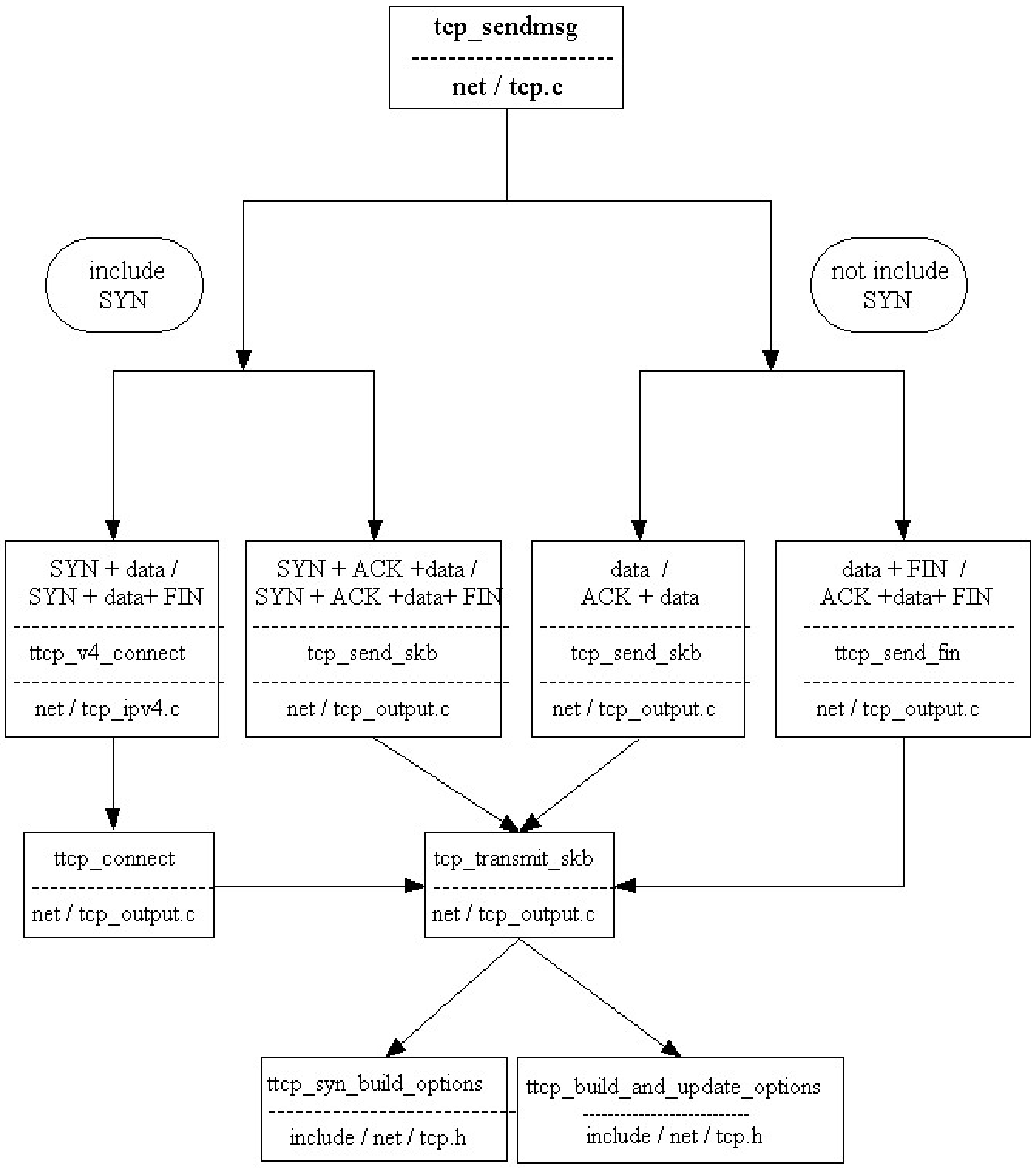


Figure 7: T/TCP Sending Flow Graph

3.2.7 Receiving Function Call Tree

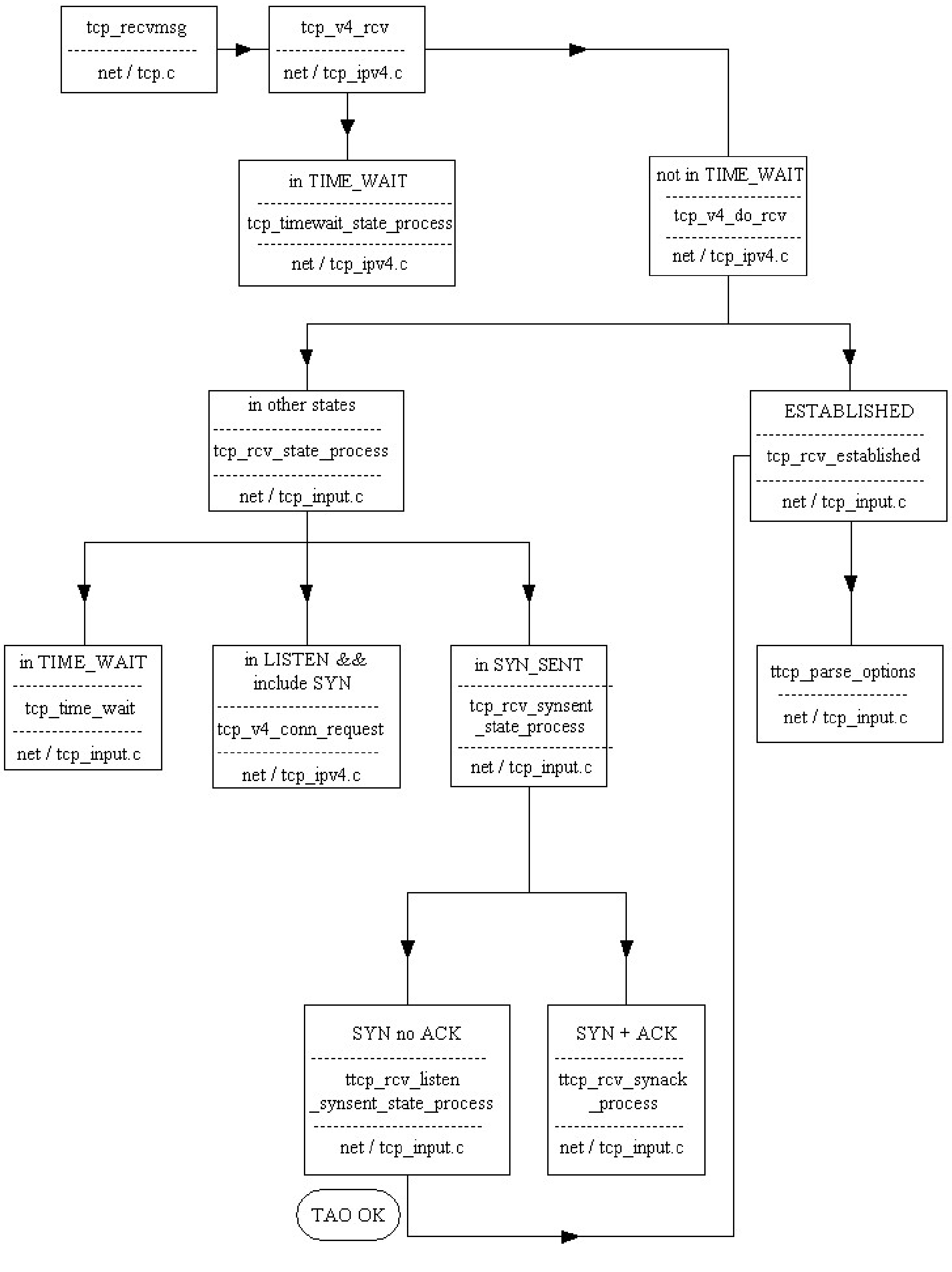


Figure 8: T/TCP receiving Flow Graph

3.2.8 T/TCP-enabled Client and Server

The T/TCP client and server codes were written by W. Richard Stevens.

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 kenel 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 funtions 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.

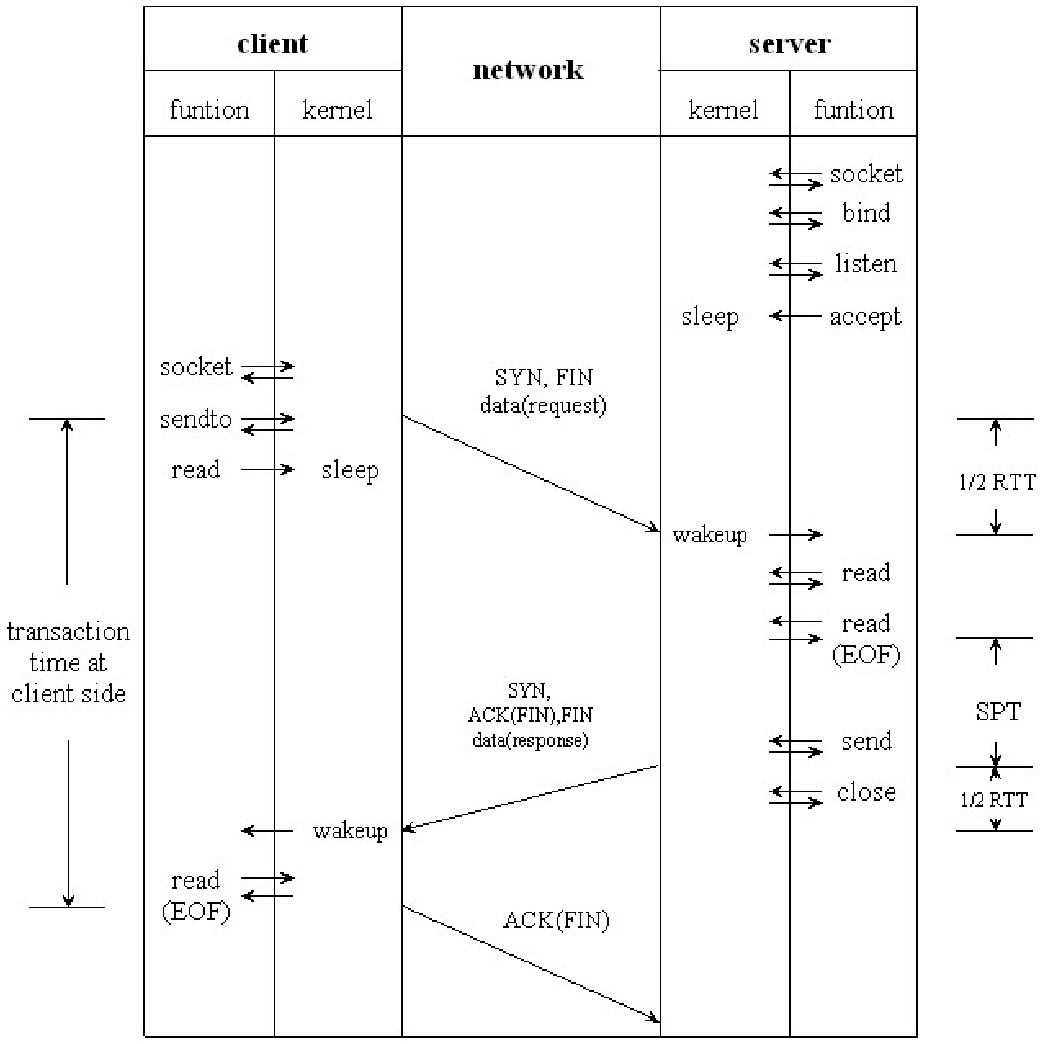


Figure 9: T/TCP-enabled Client and Server

4 Security Problems with T/TCP

4.1 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 hacking tool in history. 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 tells us that the larger the chosen tcp_ccgen is, the greater the odds that the TAO test will succeed. When in doubt, aim high.

4.2 SYN Flooding

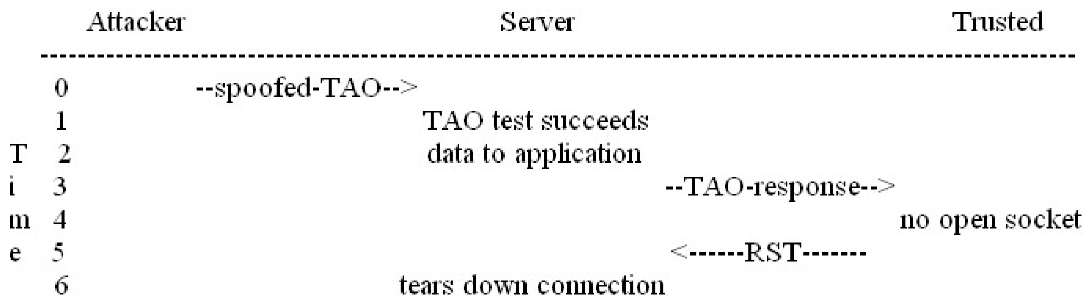
TCP SYN flooding hasn't changed much under T/TCP. The actual attack is the same: a series of TCP SYNs spoofed from unreachable IP addresses. However, there are 2 major considerations to keep in mind when the target host supports T/TCP:

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 options sent would be lost. If the final ACK comes in, only then will the host create the kernel socket data structures. TAO obviously cannot be used with SYN cookies.

Failed TAO processing result 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.

4.3 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 10 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
```

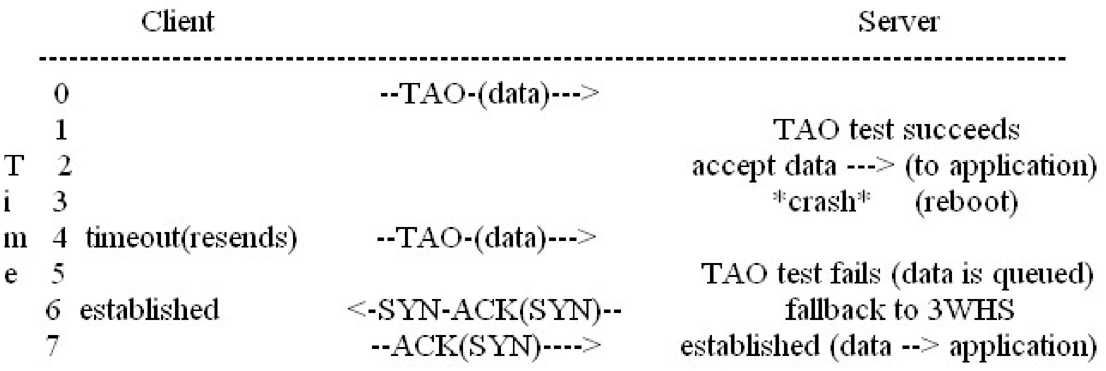
Figure 10: Security problem 1

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 went according to plan, and the process executing the command in question didn'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.

4.4 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 11 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 11: Security problem 2

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 TAO test failed and forced a 3-way handshake (6) because the servers CC cache was invalidated when it rebooted. 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 this 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.

4.5 Solutions to T/TCP Security Problems

4.5.1 Dominance of TAO

4.5.1.1 Packet Sniffing

T/TCP is not subjective to packet sniffing more easily than TCP. But after being sniffed, it's indeed much easier for you 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.

4.5.1.2 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 problem. 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 one. 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:

1) 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 see more obvious performance gains.

2) 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 one 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 can be 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 suggest making CC randomly initialized, which means CC can start from any place from 1 to 2^32-1. This way, crackers will really have some tough time.

4.5.2 SYN Flooding

4.5.2.1 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 eradicates the possibility. SYN cookie eliminates the need for storing states with the price of losing any TCP option.

Besides, SYN cookie has the 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. Again, it's groundless to blame T/TCP for its incompatibility with SYN cookie.

The solution can be simple: use SYN cache with T/TCP and turn off T/TCP when SYN cookie is enabled.

4.5.2.2 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, in order to be completely back compatible with TCP, the data should be discarded and sent again after the 3-way handshake succeeds.

4.5.3 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.

4.5.4 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 restart, 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 keeps 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.