# T/TCP -- TCP Extension for Transactions

Bob Braden
USC Information Sciences Institute

LA IETF Meeting

March 7, 1996



## Transport Protocols

- IP was designed to support a variety of transport protocols, but we have only:
  - o UDP (unreliable datagrams)
  - o TCP (reliable streams).
- An important unserved need: transactions
  - -- Request, response message pairs
  - -- No distinct open/close phase
  - -- No replays *("At-Most-Once semantics")*
  - -- Minimum latency ( >= 1 RTT)
  - -- Minimum number of segments



## Transaction Transport Protocol

- Early attempts to define an Internet std transaction transport protocol went nowhere...

  Candidates included:

  Cheriton's VMTP, Stevens' TTP, Cole/Crowcroft's SEP, Cooper's BSD RPC.

- Another approach: extend TCP for transactions

  -- Original TCP spec almost did it...

  "Kamakazi packets" => (SYN, data, FIN)

  Challenge: support both transactions and streaming

  -- "Good old TCP..." -- help community acceptance?



## TCP for Transactions

In theory, can do a transaction in 2RTT (5 segments) using "legal" RFC-793 TCP.

BUT:

(a) User interface in RFC-793 is wrong

(b) User interface in BSD is even more wrong for transactions

Realistically, takes ~10 segments per transaction...



## RFC-793 Minimal Transaction

| | TCP A (Client) | | TCP B (Server) |
|---|---|---|---|
| 1. | → < SYN, data1, FIN > → | | (queue data1, FIN) |
| 2. | ← < SYN, ACK(SYN) > ← | | |
| 3. | → < ACK(SYN) > → | | (data1->server, process FIN) |
| 4. | ← < ACK(FIN), data2, FIN > ← | | |
| 5. | → < ACK(FIN) > → | | |



## TCP for Transactions

Using TCP for transactions: there are two problems:

A. 3-way Handshake

=> 1.5 RTT to deliver request to server host

B. Close sequence and TIME_WAIT state.

(This turns out to be the harder problem)

T/TCP [RFC-1644] solves both these problems to support both transactions and streaming.

## WHAT WE WANT...

**TCP A (Client)** **TCP B (Server)**

1. → < SYN, data1, FIN > → (data1->server, process FIN)

(Server computes reply)

2. ← < SYN, ACK(FIN), data2, FIN > ←

3. → < ACK(FIN) > →

**T/TCP does this.**



## Observations

- **T/TCP approach is not the only possible one**

  ***Some assumptions:***

  **-- Cached state can be discarded**

  **-- Minimal changes to TCP fundamentals**

  **-- Must be capable of high transaction rate**

- **T/TCP is only a transport protocol; supporting tranaction applications requires more machinery**

  **Synchronization (3-phase commits)**

  **Argument marshalling, de-marshalling**

  **Process binding**



## History

**The transaction transport problem has a long history. Some important papers are:**

**o Fletcher and Watson, "Mechanisms for a Reliable Timer-Based Protocol", Computer Networks v.2, #4-5, Sept,Oct 1978, pp 271-290.**

**o Watson, "The delta-t transport protocol: Features and experience", IEEE 4th Conf on Local Computer Networks, Oct 1989.**

**o Birrell and Nelson, "Implementing Remote Procedure Calls", ACM TOCS, v.2, #1, Feb 1984, pp 39-59.**

**o Liskov, Shrira, and Wroclawski, "Efficient At-Most-Once Messages based on Synchronized Clocks", ACM SIGCOMM 1990.**

**o Shankar and Lee, "Minimum-Latency Transport Protocols with Modulo-N Incarnation Numbers", T Ntwking, Jun 1995, 255-268.**



## Acknowledgments

**o Dave Clark proposed the idea of cacheing TCP state to bypass the 3-way handshake, during IAB discussions more than 7 years ago. He also gave highly useful feedback on an early version of T/TCP.**

**o Van Jacobson suggested in the End-to-End RG that his timestamp value (RFC-1323) would be the right value to cache. This inspired work on T/TCP.**

**o The National Science Foundation supported development of T/TCP under Grant NCR-8922231.**

**o Greg Minshall asked some tough questions.**

**o Sandy Murphy and Nancy Lynch have independently pointed out that T/TCP does not provide perfect at-most-once service.**



## Outline of Rest of Talk

**1. TCP Accelerated Open**

**2. Truncating TIME-WAIT state**

**3. User Interface**

**4. Conclusions**



## Connection Count

- **T/TCP Introduces new 32-bit sequence space:**
  ***Connection Count*** **(CC)**
  **(another name for *incarnation number*)**

- **For each connection, its TCPB contains:**
  ***CCsend*** **= CC of data sent**
  ***CCrecv*** **= CC of data received**

- **Each T/TCP segment carries *CCsend* value in new CC option in TCP header.**

- **T/TCP uses 32-bit counter to generate a new *CCsend* value for each connection.**

## TAO of T/TCP

- TCP Accelerated Open: bypass 3-way handshake

  -- Each T/TCP caches:
  o The largest valid CC value received from each client host A => *cache.CC[A]*

  o The largest valid CCvalue sent to each server host B => *cache.CCsent[B]*.

- If SYN segment's CC > *cache.CC[A]*, segment must be new and can be accepted immediately

  Else do normal 3-way handshake to validate SYN.



## Basic T/TCP Transaction

TCP A (Client) TCP B (Server)

cache.CC[A]
[x0]

CCsent = x
1. CCrecv = ?? < SYN, data1, FIN, CC=x > → (x > x0 => TAO OK;
data1 -> server;
cache.CC[A] = x;
CCrecv = x; CCsent = y)

[x0] [x]
2. ← < SYN, ACK(FIN), data2, FIN, CC=y, CC.ECHO=x >
(CC.ECHO == CCsent => OK;
data2 -> client;
CCrecv = y)

[x] [x]
3. < ACK(FIN), CC=x > →



## Basic T/TCP Transaction

TCP A (Client) TCP B (Server)

cache.CCsent[B] cache.CC[A]
[x0] [x0]

1. CCsent = x
CCrecv = ?? < SYN, data1, FIN, CC=x > → (x > x0 => TAO OK;
data1 -> server;
cache.CC[A] = CCrecv = x;
CCsent = y)

[x0] [x]
2. ← < SYN, ACK(FIN), data2, FIN, CC=y, CC.ECHO=x >
(CC.ECHO == CCsent => OK;
data2 -> client;
cache.CCsent[B] = x;
CCrecv = y)

[x] [x]
3. < ACK(FIN), CC=x > →



## Other Cases . . .

- Server is standard TCP

1. < SYN, data1, FIN, CC=x > → (Ignore CC option, queue data and FIN, and do 3-way handshake)
2. (No CC option => don't send more CC's) ← < SYN, ACK(SYN) >
3. < ACK(SYN), data1, FIN > →

- Old duplicate SYN (replay attempt)

1. . . . < SYN, data1, FIN, CC=x > → (x < x0 => test fails; queue data and FIN, and do 3-way handshake)
2. ← < SYN, ACK(SYN), CC=y, CC.ECHO=x >
!!??
3. < RST > →



## More Cases . . .

- Reordered SYNs

1. < SYN, data1, FIN, CC=x > (TAO OK => data2 -> server; cache.CC[A] = z)
2. < SYN, data2, FIN, CC=z > (TAO fails => do 3-way handshake, which will succeed)

Note: B's cache.CC[A] stays = z

- Long service time

1. < SYN, data1, FIN, CC=x > → (TAO OK => ...)

(Server T/TCP times out)

2. ← < SYN, ACK(FIN), CC=y, CC.ECHO=x >
3. < ACK(SYN), CC=x > →



## And Still More Cases ...

- Request data > 1 segment

1. < SYN, data1, CC=x > → (TAO OK => ...)
2. < data2, FIN, CC=x > →
3. ← < SYN, ACK(FIN), data2, FIN, CC=y, CC.ECHO=x >

If client sends burst, server may choose to ACK before server process delivers response. Makes transition to normal TCP behavior...

(Issue buried here ... how big is client's send window initially? RFC1644 says 4KB, but that may be a bad idea... better to cache VJ cwnd value)

## Monotonicity of CC

- TAO works because CC values are monotone increasing.

  But they can't increase forever...
  - o CC value will wrap around
  - o Client may crash and lose state

- Neither case can result in faulty data

  -- With reasonable engineering assumptions,
  CC cannot wrap in less than MSL

  -- Client cannot restart from crash in less than MSL
  (TCP 'Quiet Time')



But backwards jump of CCsend could force every transaction to do 3-way handshake, for a long time.

- Client uses cache.CCsent[] to detect wraparound for particular server. Sends CC.NEW instead of CC option in next request.

- After crash, cache.CCsent[*] is undefined; this also causes client to send CC.NEW instead of CC option.

- If server crashes, cache.CC[*] == ??, forces 3-way handshake for first transaction.



## T/TCP Connection States

- T/TCP introduces many new states to TCP state diagram (e.g., half-synchronized connections).

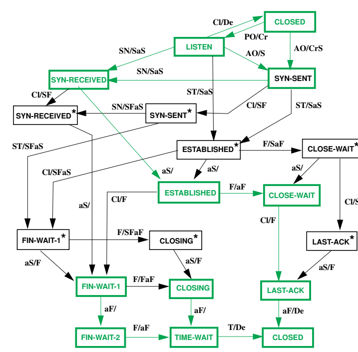
- Fortunately, can implement them using just two new state bits (see RFC1644)

...

So far, that was the EASY part of T/TCP...



## T/TCP Connection States




Green= RFC793



## 2. TIME-WAIT State

TIME-WAIT state ties up TCPB for 4 mins after
connection closes.
=> limited transaction rate

So, when new transaction comes along for TCB in TIME-WAIT state, just re-use it for new transaction.

Why is this hard? Because *TIME-WAIT state is part of fundamental TCP reliability machinery:*

T-W state allows time for old duplicate segments from earlier connection incarnations to expire.



## Two Classes of Connections in T/TCP

- "Long" TCP connections (D > MSL) --
  Same as standard TCP:

  -- T/TCP uses TIME-WAIT state for reliability

  -- Application cycles among ports

- "Short" TCP connections (transactions) (D < MSL)

  -- T/TCP *truncates* TIME-WAIT state, CC values provide reliability

  -- Application reuses same port pair.

## Truncating TIME-WAIT State

CLOSED — LISTEN

1. < SYN, data1, FIN, CC=x > →

SYN-SENT* — LAST-ACK*

2. ← < SYN, ACK(FIN), data2, FIN, CC=y, CC.ECHO=x >

TIME-WAIT — LAST-ACK*

3. < ACK(FIN), CC=x > → — CLOSED

TIME-WAIT

. . .

(new request) — LISTEN

4. < SYN, data3, FIN, CC=z > →

SYN-SENT* — LAST-ACK*

(etc.)



## SYN as Implicit ACK

SYN-SENT*

2. ← < SYN, ACK(FIN), data2, FIN, CC=y, CC.ECHO=x >

TIME-WAIT — LAST-ACK*

3. < ACK(FIN), CC=x > → X Lost

TIME-WAIT — LAST-ACK*

. . .

(new request) — (LISTEN)

4. < SYN, data3, FIN, CC=z> → (z > CCrecv => TAO OK;
ACK seg #2;
delete old TCB;
create new TCB;
data3 -> server;
cache.CC[A] = CCrecv = z;
CCsent = w)

SYN-SENT*

LAST-ACK*

5. ← < SYN, ACK(FIN), data4, FIN, CC=w, CC.ECHO=z>



## 3. Socket Interface to T/TCP

**Two changes in socket interface:**

- **sendto() call works for TCP as well as UDP**

  ***sendto(s, msg, len, flags, to, tolen), flags*** **includes EOF**

- **New setsockopt() call to turn off implied PUSH at server:**

  **setsockopt(s, IPPROTO_TCP, TCP_NOTPUSH, optval (0/1), optlen)**



## Client Code

```
sock_id = socket(AF_INET, SOCK_STREAM, 0);
sendto(sock_id, request_buffer, request_length,
                  0, foreign_socket, foreign_socket_len);

while( count = read(sock_id, reply_buffer, buffer_length) )
     { <process reply_buffer>; }
close(sock_id);
```



## Server Code

```
sock_id = socket(AF_INET, SOCK_STREAM, 0);
bind(sock_id, local_addr, local_addr_len);
setsockopt(sock_id, IPPROTO_TCP, TCP_NOTPUSH,
                              &pushval, sizeof(int));
listen(sock_id, n);
      new_sock = accept(sock_id, foreign_addr, foreign_addr_len);
      while(count = read(new_sock, request_buffer, request_length) )
            { <process request_buffer>; }
      <Compute reply and store into reply_buffer.>
      write(new_sock, reply_buffer, reply_length);
      close(new_sock);
```



## 4. Conclusions

**The good news -- T/TCP:**

- **Satisfies requirements for transaction transport**
  (except for at-most-once operation)
- **Is fully backwards-compatible with standard TCP**
- **Provides nearly seamless joining of transactions with streams.**
- **Preserves TCP semantics, syntax, and algoritms.**
- **Provides all VJ congestion-control machinery**

## Conclusions...

The bad news about T/TCP:

- o T/TCP does not support multicasting.

  Reliable multicast transport protocols live in a much larger and thicker jungle than transaction transport protocols...

- o T/TCP does not quite provide at-most-once service



## At-Most-Once Failure

TCP A (Client) TCP B (Server)

1. → < SYN, data1, FIN, CC=x > → (x > x0 => TAO OK; data1->server )

   (Server computes reply then crashes)

Client retransmits

2. → < SYN, data1, FIN, CC=x > → (Do 3-way handshake, which succeeds.)

   (Server re-computes reply and returns it.)



## Conclusions

More bad news --

- o Details are messy to get right
- o Long/short duration connections visible to applications.
- o CPU performance poor



## Open Issues in T/TCP

- Adapting VJ RTO measurement to transactions

  Each minimal transaction makes exactly ONE RTT measurement; must cache parameters and average across transactions. *Is that sufficient?*

- Initial send window

  Cache VJ cwnd; *is that sufficient?*

- CPU Performance improvement

  Complete redesign of implementation to favor transactions?