SACK TCP

(RFC 2018)

What's Wrong with Current TCP?

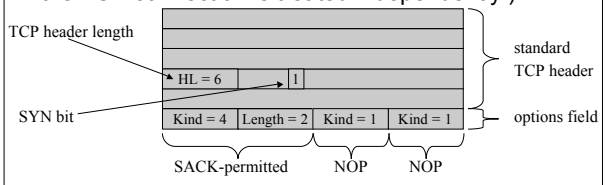
- TCP uses a cumulative acknowledgment scheme, in which the receiver identifies the last byte of data successfully received.
- Received segments that are not at the left window edge are not acknowledged.
- This scheme forces the sender to either wait a roundtrip time to find out a segment was lost, or unnecessarily retransmit segments which have been correctly received.
- Results in significantly reduced overall throughput.

Selective Acknowledgment TCP

- Selective Acknowledgment (SACK) allows the receiver to inform the sender about all segments that have been successfully received.
- Allows the sender to retransmit only those segments that have been lost.
- SACK is implemented using two different TCP options.

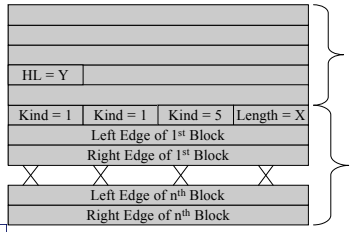
The SACK-Permitted Option

- The first TCP option is the enabling option, "SACK-permitted," allowed only in a SYN segment.
- This indicates that the sender can handle SACK data and the receiver should send it, if possible. (Both sides can enable SACK, but each direction of the TCP connection is treated independently.)



The SACK Option

- If the SACK-permitted option is received, the receiver may send the SACK option.



What is a simple formula for the SACK option length field (based on n, the number of blocks in the option)?

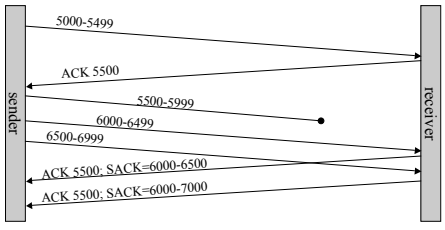
(2 + 8 * n) bytes

What is the maximum number of SACK blocks possible? Why?

The maximum size of the options field is 40 bytes, giving a maximum of **4** SACK blocks (barring no other TCP options).

The SACK Option

- Each block in a SACK represents bytes successfully received that are contiguous and isolated (the bytes immediately to the left and the right have not yet been received).



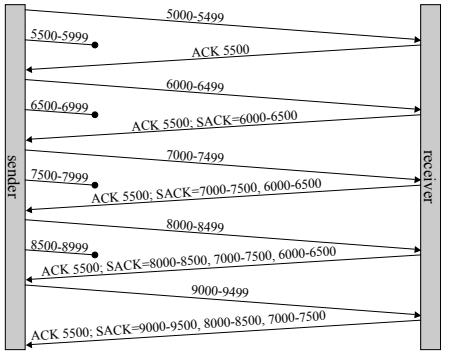
SACK TCP Rules

- A SACK cannot be sent unless the SACK-permitted option has been received (in the SYN).
- If a receiver has chosen to send SACKs, it must send them whenever it has data to SACK at the time of an ACK.
- The receiver should send an ACK for every valid segment it receives containing new data (standard TCP behavior), and each of these ACKs should contain a SACK, assuming there is data to SACK.

SACK TCP Rules

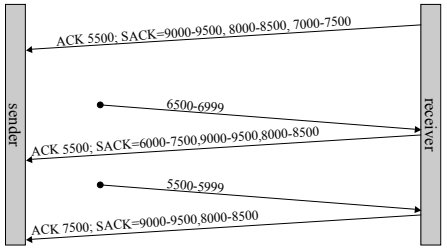
- The first SACK block must contain the most recently received segment that is to be SACKed.
- The second block must contain the second most recently received segment that is to be SACKed, and so forth.
- Notice this can result in some data in the receiver's buffers which should be SACKed but is not (if there are more segments to SACK than available space in the TCP header).

SACK TCP Example (assuming a maximum of 3 blocks)



SACK TCP Example (continued)

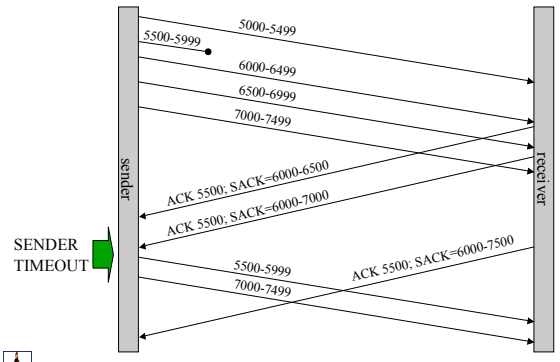
- At this point, the 4th segment (6500-6999) is received. After the receiver acknowledges this reception, the 2nd segment (5500-5999) is received.



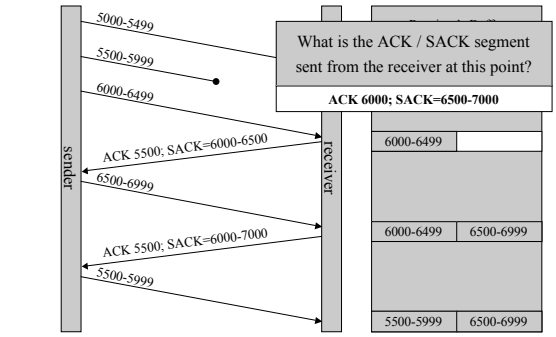
What Should the Sender do?

- The sender must keep a buffer of unacknowledged data. When it receives a SACK option, it should turn on a SACK-flag bit for all segments in the transmit buffer that are wholly contained within one of the SACK blocks.
- After this SACK flag bit has been turned on, the sender should skip that segment during any later retransmission.

SACK TCP at the Sender Example



Receiver Has A Two-Segment Buffer (A Problem?)



Reneging in SACK TCP

- It is possible for the receiver to SACK some data and then later discard it. This is referred to as reneging. This is discouraged, but permitted if the receiver runs out of buffer space.
- If this occurs,
  - The first SACK block must still reflect the newest segment, i.e. contain the left and right edges of the newest segment, even if that segment is going to be discarded.
  - Except for the newest segment, all SACK blocks must not report any old data that has been discarded.

Reneging in SACK TCP

- Therefore, the sender must maintain normal TCP timeouts. A segment cannot be considered received until an ACK is received for it. The sender must retransmit the segment at the left window edge after a retransmit timeout, even if the SACK bit is on for that segment.
- A segment cannot be removed from the transmit buffer until the left window edge is advanced over it, via the receiving of an ACK.

SACK TCP Observations

- SACK TCP follows standard TCP congestion control; it should not damage the network.
- SACK TCP has an advantage over other implementations (Reno, Tahoe, Vegas, and NewReno) as it has added information due to the SACK data.
- This information allows the sender to better decide what it needs to retransmit and what it does not. This can only serve to help the sender, and should not adversely affect other TCPs.

SACK TCP Observations

- While it is still possible for a SACK TCP to needlessly retransmit segments, the number of these retransmissions has been shown to be quite low in simulations, relative to Reno and Tahoe TCP.
- In any case, the number of needless retransmissions must be strictly less than Reno/Tahoe TCP. As the sender has additional information from which to devise its retransmission scheme, worse performance is not possible (barring a flawed implementation).

SACK TCP Implementation Progress

- Current SACK TCP implementations:
  - Windows 2000
  - Windows 98 / Windows ME
  - Solaris 7 and later
  - Linux kernel 2.1.90 and later
  - FreeBSD and NetBSD have optional modules
- ACIRI has measured the behavior of 2278 random web servers that claim to be SACK-enabled. Out of these, 2133 (93.6%) appeared to ignore SACK data and only 145 (6.4%) appeared to actually use the SACK data.

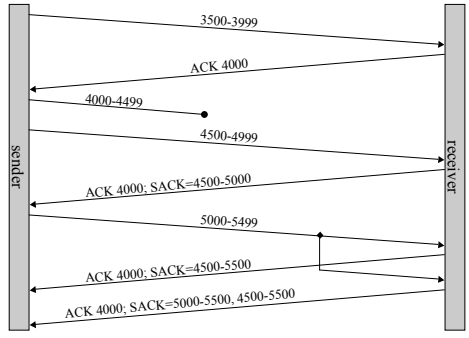
D-SACK TCP

(RFC 2883)

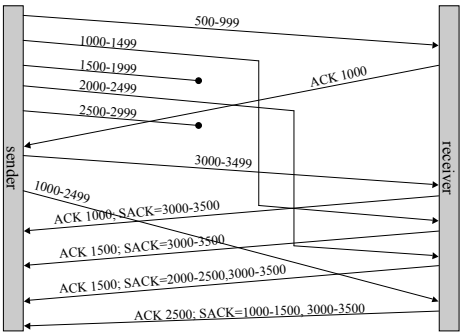
One Step Further: D-SACK TCP

- Duplicate-SACK, or D-SACK is an extension to SACK TCP which uses the first block of a SACK option is used to report duplicate segments that have been received.
- A D-SACK block is only used to report a duplicate contiguous sequence of data received by the receiver in the most recent segment.
- Each duplicate is reported at most once.
- This allows the sender TCP to determine when a retransmission was not necessary. It may not have been necessary due to the retransmit timer expiring prematurely or due to a false Fast Retransmit (3 duplicate ACKs received due to network reordering).

D-SACK Example (packet replicated by the network)



D-SACK Example (losses, and the sender changes the segment size)



D-SACK TCP Rules

- If the D-SACK block reports a duplicate sequence from a (possibly larger) block of data in the receiver buffer above the cumulative acknowledgement, the second SACK block (the first non D-SACK block) should specify this block.
- As only the first SACK block is considered to be a D-SACK block, if multiple sequences are duplicated, only the first is contained in the D-SACK block.

D-SACK TCP and Retransmissions

- D-SACK allows TCP to determine when a retransmission was not necessary (it receives a D-SACK after it retransmitted a segment). When this determination is made, the sender can "undo" the halving of the congestion window, as it will do when a segment is retransmitted (as it assumes net congestion).
- D-SACK also allows TCP to determine if the network is duplicating packets (it will receive a D-SACK for a segment it only sent once).
- D-SACK's weakness is that is does not allow a sender to determine if both the original and retransmitted segment are received, or the original is lost and the retransmitted segment is duplicated by the network.

SACK and D-SACK Interaction

- There is no difference between SACK and D-SACK, except that the first SACK block is used to report a duplicate segment in D-SACK.
- There is no separate negotiation/options for D-SACK.
- There are no inherit problems with having the receiver use D-SACK and having the sender use traditional SACK. As the duplicate that is being reported is still being SACKed (for the second or greater time), there is no problem with a SACK TCP using this extension with a D-SACK TCP (although the D-SACK specific data is not used).

Increasing the Maximum TCP Initial Window Size

(RFC 2414)

Increasing the Initial Window

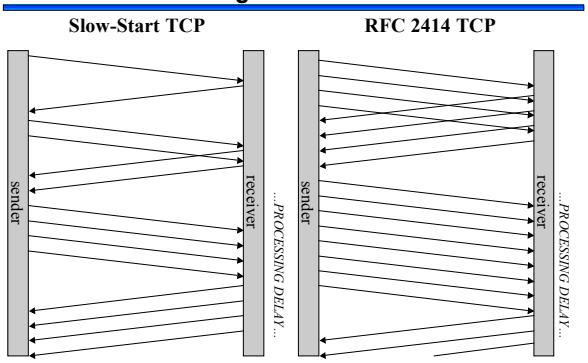
- RFC 2414 specifies an experimental change to TCP, the increasing of the maximum initial window size, from one segment to a larger value.
- This new larger value is given as:

$$\min (4*MSS, \max (2*MSS, 4380 \text{ bytes}))$$

- This translates to:

Maximum Segment Size (MSS)	Maximum Initial Window Size
<= 1095 bytes	<= 4 * MSS
1095 bytes < MSS < 2190 bytes	<= 4380 bytes
>= 2190 bytes	<= 2 * MSS

Increasing the Initial Window



Advantages of an Increased Initial Window Size

- This change is in contrast to the slow start mechanism, which initializes the initial window size to one segment. This mechanism is in place to implement sender-based congestion control (see RFC 2001 for a complete discussion).
- This new larger window offers three distinct advantages:
  - With slow start, a receiver which uses delayed ACKs is forced to wait for a timeout before generating an ACK. With an initial window of at least two segments, the receiver will generate an ACK after the second segment arrives, causing a speedup in data acknowledgement.

Advantages of an Increased Initial Window Size

- For TCP connections transferring a small amount of data (such as SMTP and HTTP requests), the larger initial window will reduce the transmission time, as more data can be outstanding at once.
- For TCP connections transferring a large amount of data with high propagation delays (long haul pipes; such as backbone connects and satellite links), this change eliminates up to three round-trip times (RTTs) and a delayed ACK timeout during the initial slow start.

Disadvantages of an Increased Initial Window Size

- This approach also has disadvantages:
  - This approach could cause increased congestion, as multiple segments are transmitted at once, at the beginning of the connection. As modern routers tend to not handle bursty traffic well (Drop Tail queue management), this could increase the drop rate.
- ACIRI research on this topic concludes that there is no more danger from increasing the initial TCP window size to a maximum of 4KB than the presence of UDP communications (that do not have end-to-end congestion control).

Increased Initial Window Size Implementation Progress

- Looking at ACIRI observations, current web servers use a wide range of initial TCP window sizes, ranging from one segment (slow start) to seventeen segments.
- This is a clear violation of RFC 2414, not to mention RFC 2001 (the currently approved IETF/ISOC standard).
- Such large initial window sizes seem to indicate a greedy TCP, not conforming to the required sender-side congestion control window (even if the experimental higher initial window is considered).

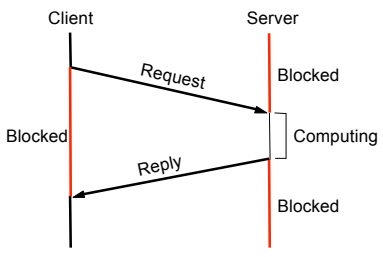
Summary

- SACK TCP provides additional information to the sender, allowing the reduction of needless retransmissions. There is no danger in providing this information, it simply serves to make a "smarter" TCP sender.
- D-SACK TCP allows the sender to determine when it has needlessly resent segments. This will allow the sender to continuously refine its retransmission strategy and undo unnecessary and incorrect congestion control mechanisms.
- Increasing the initial TCP window is a slight change that has advantages for both small and large data transfers, without significantly affecting the congestion control a smaller window provides.

Remote Procedure Call

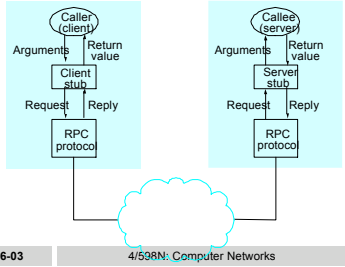
- Outline
  - Protocol Stack
  - Presentation Formatting

RPC Timeline



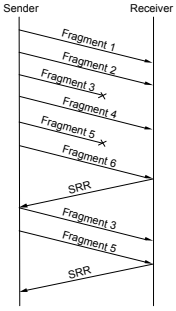
RCP Components

- Protocol Stack
  - BLAST: fragments and reassembles large messages
  - CHAN: synchronizes request and reply messages
  - SELECT: dispatches request to the correct process
- Stubs



Bulk Transfer (BLAST)

- Unlike AAL and IP, tries to recover from lost fragments
- Strategy
  - selective retransmission
  - aka partial acknowledgements



BLAST Details

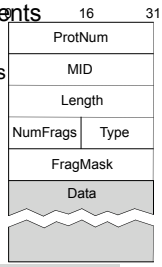
- Sender:
  - after sending all fragments, set timer DONE
  - if receive SRR, send missing fragments and reset DONE
  - if timer DONE expires, free fragments

BLAST Details (cont)

- Receiver:
  - when first fragments arrives, set timer LAST_FRAG
  - when all fragments present, reassemble and pass up
  - four exceptional conditions:
    - if last fragment arrives but message not complete
      - send SRR and set timer RETRY
    - if timer LAST_FRAG expires
      - send SRR and set timer RETRY
    - if timer RETRY expires for first or second time
      - send SRR and set timer RETRY
    - if timer RETRY expires a third time
      - give up and free partial message

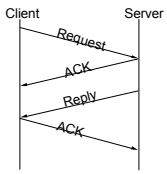
BLAST Header Format

- MID must protect against wrap around
- TYPE = DATA or SRR
- NumFrags indicates number of fragments
- FragMask distinguishes among fragments
  - if Type=DATA, identifies this fragment
  - if Type=SRR, identifies missing fragments

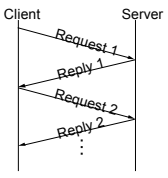


Request/Reply (CHAN)

- Guarantees message delivery
- Synchronizes client with server
- Supports at-most-once semantics
- Simple case



Implicit Acks



CHAN Details

- Lost message (request, reply, or ACK)
  - set RETRANSMIT timer
  - use message id (MID) field to distinguish
- Slow (long running) server
  - client periodically sends “are you alive” probe, or
  - server periodically sends “I’m alive” notice
- Want to support multiple outstanding calls
  - use channel id (CID) field to distinguish
- Machines crash and reboot
  - use boot id (BID) field to distinguish

CHAN Header Format

```
typedef struct {
    u_short  Type;    /* REQ, REP, ACK, PROBE */
    u_short  CID;     /* unique channel id */
    int      MID;     /* unique message id */
    int      BID;     /* unique boot id */
    int      Length;  /* length of message */
    int      ProtNum; /* high-level protocol */
} ChanHdr;

typedef struct {
    u_char    type;         /* CLIENT or SERVER */
    u_char    status;       /* BUSY or IDLE */
    int       retries;      /* number of retries */
    int       timeout;      /* timeout value */
    XkReturn  ret_val;      /* return value */
    Msg       *request;     /* request message */
    Msg       *reply;       /* reply message */
    Semaphore reply_sem;    /* client semaphore */
    int       mid;          /* message id */
    int       bid;          /* boot id */
} ChanState;
```


Synchronous vs Asynchronous Protocols

- Asynchronous interface

```
xPush(Sessn s, Msg *msg)
xPop(Sessn s, Msg *msg, void *hdr)
xDemux(Protl hlp, Sessn s, Msg *msg)
```

- Synchronous interface

```
xCall(Sessn s, Msg *req, Msg *rep)
xCallPop(Sessn s, Msg *req, Msg *rep, void *hdr)
xCallDemux(Protl hlp, Sessn s, Msg *req, Msg *rep)
```

- CHAN is a hybrid protocol
  - synchronous from above: **`xCall`**
  - asynchronous from below: **`xPop/xDemux`**


```
chanCall(Sessn self, Msg *msg, Msg *rmsg){
    ChanState   *state = (ChanState *)self->state;
    ChanHdr     *hdr;
    char        *buf;

    /* ensure only one transaction per channel */
    if ((state->status != IDLE))
       return XK_FAILURE;
    state->status = BUSY;

    /* save copy of req msg and ptr to rep msg*/
    msgConstructCopy(&state->request, msg);
    state->reply = rmsg;
    /* fill out header fields */
    hdr = state->hdr_template;
    hdr->Length = msgLen(msg);
    if (state->mid == MAX_MID)
       state->mid = 0;
       hdr->MID = ++state->mid;
```



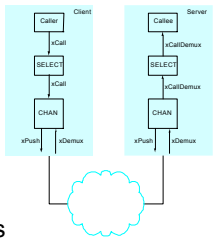
```
    /* attach header to msg and send it */
    buf = msgPush(msg, HDR_LEN);
    chan_hdr_store(hdr, buf, HDR_LEN);
    xPush(xGetDown(self, 0), msg);

    /* schedule first timeout event */
    state->retries = 1;
    state->event = evSchedule(retransmit, self, state->timeout);

    /* wait for the reply msg */
    semWait(&state->reply_sem);

    /* clean up state and return */
    flush_msg(state->request);
    state->status = IDLE;
    return state->ret_val;
}
```



```
retransmit(Event ev, int *arg){
    Sessn        s = (Sessn)arg;
    ChanState    *state = (ChanState *)s->state;
    Msg          tmp;

    /* see if event was cancelled */
    if ( evIsCancelled(ev) ) return;

    /* unblock client if we've retried 4 times */
    if (++state->retries > 4) {
       state->ret_val = XK_FAILURE;
       semSignal(state->rep_sem);
       return;
    }

    /* retransmit request message */
    msgConstructCopy(&tmp, &state->request);
    xPush(xGetDown(s, 0), &tmp);

    /* reschedule event with exponential backoff */
    evDetach(state->event);
    state->timeout = 2*state->timeout;
    state->event = evSchedule(retransmit, s,
    state->timeout);
}
```


Dispatcher (SELECT)

- Dispatch to appropriate procedure
- Synchronous counterpart to UDP



- Address Space for Procedures
  - flat: unique id for each possible procedure
  - hierarchical: program + procedure number

Example Code

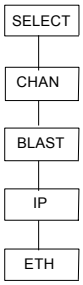
Client side

```
static XkReturn
selectCall(Sessn self, Msg *req, Msg *rep)
{
   SelectState *state=(SelectState *)self->state;
   char        *buf;

   buf = msgPush(req, HLEN);
   select_hdr_store(state->hdr, buf, HLEN);
   return xCall(xGetDown(self, 0), req, rep);
}
```

Server side

```
static XkReturn
selectCallPop(Sessn s, Sessn lls, Msg *req, Msg *rep, void *inHdr)
{
   return xCallDemux(xGetUp(s), s, req, rep);
}
```


Simple RPC Stack

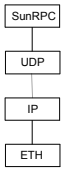


VCHAN: A Virtual Protocol

```
static XkReturn
vchanCall(Sessn s, Msg *req, Msg *rep)
{
   Sessn      chan;
   XkReturn   result;
   VchanState *state=(VchanState *)s->state;

   /* wait for an idle channel */
   semWait(&state->available);
   chan = state->stack[--state->tos];

   /* use the channel */
   result = xCall(chan, req, rep);

   /* free the channel */
   state->stack[state->tos++] = chan;
   semSignal(&state->available);
   return result;
}
```


SunRPC

- IP implements BLAST-equivalent
  - except no selective retransmit
- SunRPC implements CHAN-equivalent
  - except not at-most-once



- UDP + SunRPC implement SELECT-equivalent
  - UDP dispatches to program (ports bound to programs)
  - SunRPC dispatches to procedure within program

SunRPC Header Format

- XID (transaction id) is similar to CHAN's MID
- Server does not remember last XID it serviced
- Problem if client retransmits request while reply is in transit