Sliding Windows Protocols

- A One-Bit Sliding Window Protocol
- A Protocol Using Go Back N
- A Protocol Using Selective Repeat

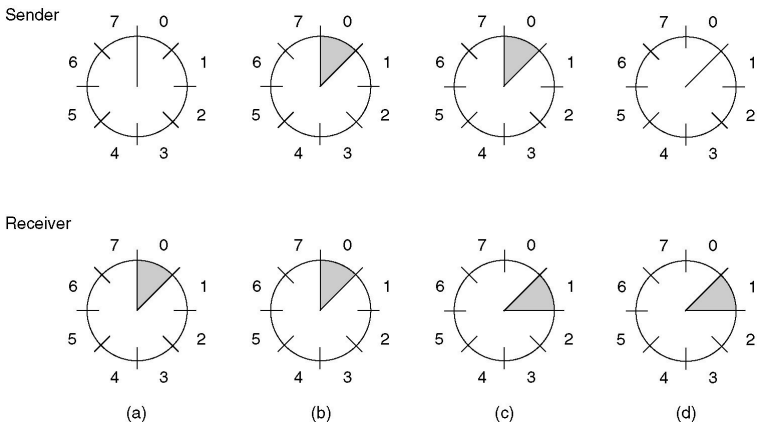
Introduction

- Full Duplex It is possible to use two physical connections for two-way communication, but it is more efficient to use a full duplex link,
- Possible to send acknowledgements by piggybacking: i.e., send the ack as part of data frames traveling in the opposite direction,
  - Advantages: Reduction in the number of frames transmitted, reduction in the number of interrupts, etc.
  - Problem: How much time do we wait for a frame going in the opposite direction?
  - Typically time picked will be somewhat less than the timeout period.

Introduction – Sliding Windows

- Each frame traveling in each direction is identified using **sequence numbers**
  - Range: 0 .. 2^n-1 where n is the number of bits,
  - Sending window: Includes the frames which have been sent but are not yet acknowledged,
    - Keep frames All unacknowledged frames must be kept in buffers in case they need to be retransmitted,
    - Refuse network layer request if the window (i.e. buffer space) is full,
  - Receiving window corresponds to frames that it may accept,
    - Pass to network layer frames that arrive at the leading edge of the window,

Example



One-Bit Protocol

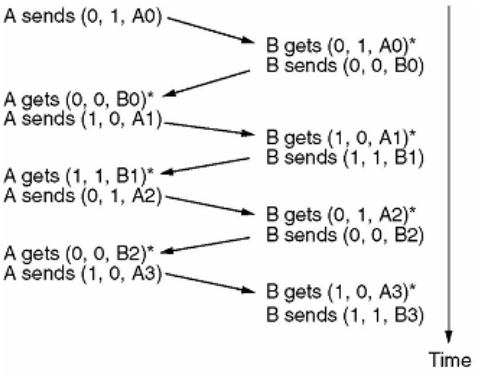

```
#define MAX_SEQ 1
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
  seq_nr next_frame_to_send = 0;
  seq_nr frame_expected = 0;
  frame r, s;
  packet buffer;
  event_type event;

  from_network_layer(&buffer);
  s.info = buffer;
  s.seq = next_frame_to_send;
  s.ack = 1 – frame_expected;
  to_physical_layer(&s);
  start_timer(s.seq);
```

Continued →

One-Bit Protocol


```
while (true) {
    wait_for_event(&event);
    if (event == frame_arrival) {
        from_physical_layer(&r);
        if (r.seq == frame_expected) {
            to_network_layer(&r.info);
            inc(frame_expected);
        }
        if (r.ack == next_frame_to_send) {
            stop_timer(r.ack);
            from_network_layer(&buffer);
            inc(next_frame_to_send);
        }
    }
    s.info = buffer;
    s.seq = next_frame_to_send;
    s.ack = 1 – frame_expected;
    to_physical_layer(&s);
    start_timer(s.seq);
  }
}
```

One-Bit Protocol



Results

Simulating Protocol 4

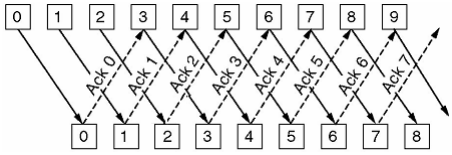
Events	100000
Timeout	30
pct_loss	20
pct_cksum	15

	Process 0	Process 1
Total data frames sent	10316	10289
Data frames lost	2073	2047
Data frames not lost	8243	8242
Frames retransmitted	2073	2045
Good ack frames rec'd	0	0
Bad ack frames rec'd	0	0
Good data frames rec'd	7008	7001
Bad data frames rec'd	1234	1242
Payloads accepted	4818	4818
Total ack frames sent	0	0
Ack frames lost	0	0
Ack frames not lost	0	0
Timeouts	2073	2045
Ack timeouts	0	0

Efficiency (payloads accepted/data pkts sent) = 46%
End of simulation. Time=100000

Pipelining

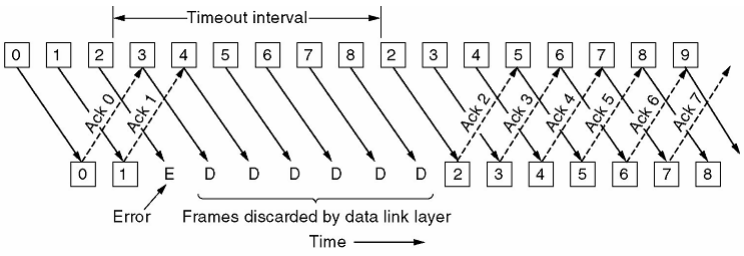
- We have been assuming that the Transmission time is negligible….
  - R is the round trip propagation delay, e.g. 500 msec,
  - Time to send a frame = L / B, frame size / bandwidth (1000 bits / 50000 bps)
  - Utilisation = L / (L + b.R) 1000 / (1000 + 50000*0.5)=0.03



- Pipelining
  - Multiple outstanding frames so that potentially usable transmission time is not wasted.
  - We need to consider Bandwidth * Propagation delay to decide how large the sending window needs to be.

Pipelining – Errors

- If a frame is damaged, we must wait for the timeout,
- Then retransmit, all subsequent frames,
- The timeout must be longer than the round trip propagation delay,



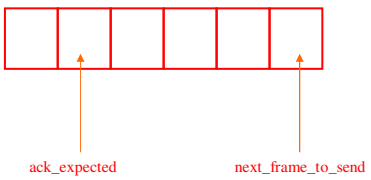
Go–Back–N Protocol (1)


```
#define MAX_SEQ 7
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready} event_type;
#include "protocol.h"

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
  if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
       return(true);
    else
       return(false);
}

static void send_data(seq_nr frame_nr, seq_nr frame_expected, packet buffer[ ])
{
  frame s;
  s.info = buffer[frame_nr];
  s.seq = frame_nr;
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
  to_physical_layer(&s);
  start_timer(frame_nr);
}
```

Continued →

Go–Back–N Protocol (2)


```
void protocol5(void)
{
  seq_nr next_frame_to_send = 0;
  seq_nr ack_expected = 0;
  seq_nr frame_expected = 0;
  frame r;
  packet buffer[MAX_SEQ + 1];
  seq_nr nbuffered = 0;
  seq_nr i;
  event_type event;

  enable_network_layer();
```



Continued →

Go–Back–N Protocol (3)


```
while (true) {
  wait_for_event(&event);

  switch(event) {
    case frame_arrival:
          from_physical_layer(&r);
          if (r.seq == frame_expected) {
                to_network_layer(&r.info);
                inc(frame_expected);
          }
          while (between(ack_expected, r.ack, next_frame_to_send)) {
                nbuffered = nbuffered −1;
                stop_timer(ack_expected);
                inc(ack_expected);
          }
          break;
    case cksum_err:
          break;
```

Continued →

Go–Back–N Protocol (4)


```
    case network_layer_ready:
          from_network_layer(&buffer[next_frame_to_send]);
          nbuffered = nbuffered + 1;
          send_data(next_frame_to_send, frame_expected, buffer);
          inc(next_frame_to_send);
          break;
    case timeout:
          next_frame_to_send = ack_expected;
          for (i = 1; i <= nbuffered; i++) {
                send_data(next_frame_to_send, frame_expected, buffer);
                inc(next_frame_to_send);
          }
  }
  if (nbuffered < MAX_SEQ)
          enable_network_layer();
  else
          disable_network_layer();
 }
}
```

Results

Simulating Protocol 5

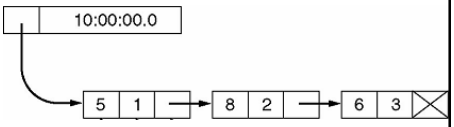
Events	100000
Timeout	30
pct_loss	20
pct_cksum	15

	Process 0	Process 1
Total data frames sent	22716	22364
Data frames lost	4570	4485
Data frames not lost	18146	17879
Frames retransmitted	17451	17143
Good ack frames rec'd	0	0
Bad ack frames rec'd	0	0
Good data frames rec'd	15178	15422
Bad data frames rec'd	2695	2724
Payloads accepted	5214	5260
Total ack frames sent	0	0
Ack frames lost	0	0
Ack frames not lost	0	0
Timeouts	2493	2449
Ack timeouts	0	0

Efficiency (payloads accepted/data pkts sent) = 23%
End of simulation. Time=100000

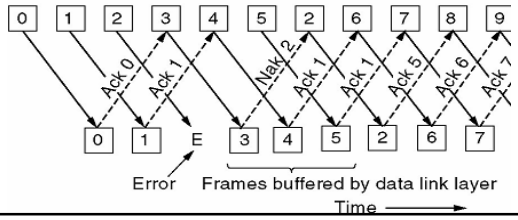
Comments

- Buffering
  - Required for all sent frames – up to maximum MAX_SEQ,
  - Not needed for received frames, as they are either transmitted to the network layer or are discarded.
- Assumes that acknowledgements are sent as soon as frames are successfully received,
- Multiple timers are needed to support multiple outstanding frames. This can be implemented in software using a linked-list like structure.



Selective Repeat

- In the case of an error, rather than wait for a timeout send a NAK (Negative Acknowledgement). Either:
  - Go Back N: Requires the sender to resend all frames since error,
  - Selective Repeat: Sender retransmits only the missing frame. More management by the receiver.
- Line utilization: 1-bit sliding window<Go-Back-N<Selective Repeat
- Buffering
  - Sent frames: Still need buffers for un-acknowledged sent frames,
  - Received frames: As many buffers as our receiving window, i.e. what happens when all frames are received minus the first?



Selective Repeat Protocol (1)


```
#define MAX_SEQ 7
#define NR_BUFS ((MAX_SEQ + 1)/2)
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout} event_type
#include "protocol.h"
boolean no_nak = true;
seq_nr oldest_frame = MAX- SEQ + 1;

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
  return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
}


static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[ ])
{
  frame s;
  s.kind = fk;
  if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
  s.seq = frame_nr;
  s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
  if (fk == nak) no_nak = false;
  to_physical_layer(&s);
  if (fk == data) start_timer(frame_nr % NR_BUFS);
  stop_ack_timer();
}
```

Continued →

Selective Repeat Protocol (2)


```
void protocol6(void)
{
  seq_nr ack_expected = 0;
  seq_nr next_frame_to_send = 0;
  seq_nr frame_expected = 0;
  seq_nr too_far = NR_BUFS;
  int i;
  frame r;
  packet out_buf[NR_BUFS];
  packet in_buf[NR_BUFS];
  boolean arrived[NR_BUFS];
  seq_nr nbuffered = 0;
  event_type event;

  enable_network_layer();
  for (i = 0; i < NR_BUFS; i++) arrived[i] = false;

  while (true) {
    wait_for_event(&event);
    switch(event) {
      case network_layer_ready:
            nbuffered = nbuffered + 1;
            from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]);
            send_frame(data, next_frame_to_send, frame_expected, out_buf);
            inc(next_frame_to_send);
            break;
```

Continued →

Selective Repeat Protocol (3)


```
case frame_arrival:
      from_physical_layer(&r);
      if (r.kind == data) {
            if ((r.seq != frame_expected) && no_nak)
              send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
            if (between(frame_expected, r.seq, too_far)
                              && (arrived[r.seq%NR_BUFS] == false)) {
                  arrived[r.seq % NR_BUFS] = true;
                  in_buf[r.seq % NR_BUFS] = r.info;
                  while (arrived[frame_expected % NR_BUFS]) {
                        to_network_layer(&in_buf[frame_expected % NR_BUFS]);
                        no_nak = true;
                        arrived[frame_expected % NR_BUFS] = false;
                        inc(frame_expected);
                        inc(too_far);
                        start_ack_timer();
                  }
            }
      }
      if((r.kind==nak) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next_frame_to_send))
            send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);
      while (between(ack_expected, r.ack, next_frame_to_send)) {
            nbuffered = nbuffered - 1;
            stop_timer(ack_expected % NR_BUFS);
            inc(ack_expected);
      }
      break;
```

Continued →

Selective Repeat Protocol (4)


```
        case cksum_err:
                if (no_nak) send_frame(nak, 0, frame_expected, out_buf);
                break;
        case timeout:
                send_frame(data, oldest_frame, frame_expected, out_buf);
                break;
        case ack_timeout:
                send_frame(ack,0,frame_expected, out_buf);
    }
    if (nbuffered < NR_BUFS)  enable_network_layer(); else disable_network_layer();
  }
}
```


Results

Simulating Protocol 6

Events	100000
Timout	30
pct_loss	20
pct_cksum	15

Process 0		
Total data frames sent	15489	15237
Data frames lost	3163	3133
Data frames not lost	12326	12104
Frames retransmitted	6080	5997
Good ack frames rec'd	73	61
Bad ack frames rec'd	15	18
Good data frames rec'd	10346	10473
Bad data frames rec'd	1756	1853
Payloads accepted	7981	8086
Total ack frames sent	100	101
Ack frames lost	21	13
Ack frames not lost	79	88
Timeouts	6080	5997
Ack timeouts	100	101

Efficiency (payloads accepted/data pkts sent) = 52%
End of simulation. Time=100000

Comments

- Problem:
  - Given a 3 bit sequence number
  - Frames 0,1,2,3,4,5,6 received
  - Send an acknowledgement (of all frames up to 6)
  - What happens if the acknowledgement is lost?

    Timeout, so sender retransmits same sequence, 0, then 1, etc. It is valid, but receiver does not know, since 0 is within his window.
- To overcome this we require that the window size for this protocol, is no bigger than half the range of sequence number,
- We also need an explicit ACK in case there is no traffic in the reverse direction…
  - The timer must be less than the timeout – round-trip propagation delay.