Sliding Window Protocol (INCOMPLETE) + Homework 2

A. Udaya Shankar

1 Introduction

Consider a source system and a sink system connected by a lossy channel. A user at the source system sends data blocks into the source; it calls a function `tx(data)` implemented by the source to send in data block `data`. A user at the sink system receives (removes) data blocks from the sink; it calls a function `rx()` implemented by the sink to get a data block. The lossy channel provides its own send and receive functions, also named `tx` and `rx`, to the source and sink systems. The source (sink) system calls `tx(msg)` to give message `msg` to the lossy channel to be delivered to the sink (source) system. The source (sink) system calls `rx()` to get a message from the lossy channel sent by the sink (source) system.

The objective is to define programs to be executed by the source and sink systems such that the sink user receives data blocks in the order they were generated by the source user. The sliding window protocol achieves this in an elegant way.

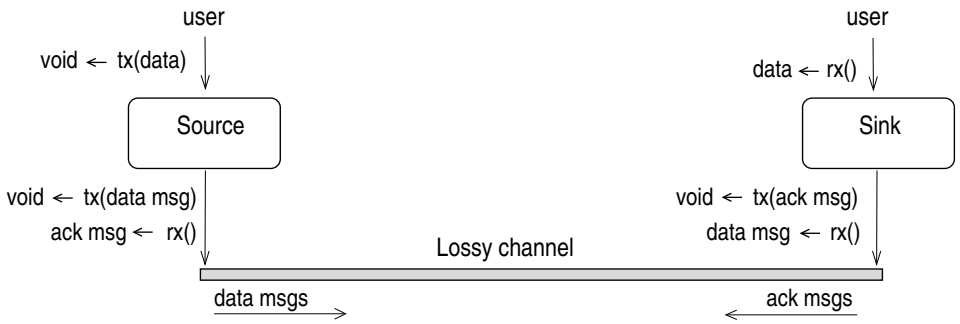

Figure 1: Source and sink systems attached to a lossy channel.

2 Solution using unbounded sequence numbers

Here is one possible solution. The source has a buffer, `sbuff`, for storing data blocks from the local user until they are acked by the sink. The source transfers a data block over the lossy channel to the sink by retransmitting it until an acknowledgment is received.

The source has a buffer, `rbuff`, for storing received data blocks until they are delivered to its local user. The sink delivers a data block from `rbuff` to its local user after determining that it is the next in fifo sequence to be delivered. An obvious way to achieve the latter is for the source to tag successive data blocks supplied by its user with increasing sequence numbers, and to include the sequence number in any message containing the data block.

The sequence numbers can also be used by the sink to acknowledge the reception of data blocks to the source. The accepted convention is for the ack to indicate the data block *next expected in sequence* rather than the received data block. (In networking jargon, the ack is "cumulative" and not "selective".) That is, suppose the sink receives data block `j` when it already has `0`, $\cdots$, `k-1` but not `k`; if $\texttt{j} \neq \texttt{k}$ the sink responds with ack `k`, and if $\texttt{j} = \texttt{k}$ the sink responds with ack `k+1`.

Finally, for good throughput, the source should be able to have several data blocks *outstanding*, i.e., sent but not yet acked, and the sink should be able to buffer data blocks received out of sequence. Note that if the sink has data blocks `0`, $\cdots$, `k-1`, `k+1`, $\cdots$, `j` and receives data block `k`, then it responds with ack `j+1`.

The only drawback of this solution is that the sequence numbers in the messages would grow without bound as more and more data blocks are transferred. This would greatly complicate fast (hardware) implementation of the component systems; it would also waste some channel bandwidth.

3 Solution using cyclic sequence numbers

The sliding window protocol fixes this problem by using *cyclic*, or modulo-`N`, sequence numbers instead of the *unbounded* sequence numbers. A **data message** has the form `[cn,data]`, where `cn` is a cyclic sequence number and `data` is a data block. When the source sends a data message with data block `j`, it sets the cyclic sequence number to `mod(j,N)`. An **ack message** has the form `[cn]`, where `cn` is a cyclic sequence number. When the sink sends an ack, it sets the cyclic sequence number to `mod(j,N)` where `j` is the next expected data block.

For this to work properly, when the sink (source) receives a data (ack) message, *it must correctly infer the unbounded sequence number corresponding to the message's cyclic sequence number*. For that to happen, the sequence numbers in transit must remain within appropriate bounds; for example, if the sink is expecting data block `j` and it receives data block `j+N`, the sink would incorrectly treat the received data block as data block `j`.

The sliding window protocol achieves this in an elegant way. The source maintains a **send window** corresponding to the sequence numbers of the data blocks that can be outstanding. The sink maintains a **receive window** corresponding to the sequence numbers of the data blocks that the sink is prepared to receive out of sequence. The parameters `SW` and `RW` indicate the sizes of the send and receive windows, respectively; obviously, each has to be less than `N`. Over time as data blocks are transferred, these two windows move towards increasing sequence numbers, always maintaining an overlap of at least one sequence number, and the sequence numbers in transit remain within a constant bound of the windows.

Figure 2 describes the source program. Figure 3 describes the sink program. Figure 4 illustrates the relationships between the variables of the source and the sink.

Homework 2

Due start of class Monday Oct 3. No late submission accepted.

Change the data message structure by adding an additional field containing the unbounded sequence number; i.e., let the source send `[mod(j,N), sbuff[j], j]` instead of `[mod(j,N), sbuff[j]]`.

(This is only for purposes of analysis only. The code executed by the sink remains the same, so the sink does not see the unbounded sequence number.)

When the sink receives a data message `[ck, data, k]`, what condition must `k` satisfy in order for the sink to correctly interpret the cyclic sequence number `ck`.

Source variables

- `ng`: initially `0`. Number of data blocks generated by local user.
- `ns`: initially `0`. Number of data blocks sent at least once.
- `na`: initially `0`. Number of data blocks acknowledged.
- `sbuff`: map initially empty. Buffers data blocks `na..ng-1`. Entry `sbuff[i]` stores data block `i`.

Source rules (atomically executed)

- When local user passes in data block `data`
  ```
  sbuff[ng] ← data;
  ng ← ng+1;
  ```
- Send `sbuff[ns]` only if `ns < min(na+SW, ng)`
  ```
  send [mod(ns,N), sbuff[ns]];
  ns ← ns+1;
  ```
- Resend outstanding data blocks every rtt:
  ```
  for (k in na..ns-1)
      send [mod(k,N), sbuff[k]];
  ```
- Receive `[cn]`:
  ```
  j ← na+mod(cn-na,N);      // j is first number on or after na that matches cn
  if (na < j ≤ ns) {
      for (k in na..j-1) sbuff.remove(k);
      na ← j;
  }
  ```

Figure 2: Variables and rules of source.

Sink variables

- `nd`: initially `0`. Number of data blocks delivered to local user.
- `nr`: initially `0`. Number of contiguous data blocks received.
- `rbuff`: map initially empty. Buffers received data blocks in `nd..nd+RW-1`. Entry `rbuff[i]`, if present, should store data block `i`.

Sink rules (atomically executed)

- When local user attempts to get a data block:
  ```
  block if nd < nr;
  deliver rbuff[nd] to user;
  rbuff.remove(nd);
  nd ← nd+1;
  ```
- Receive `[cn, data]`:
  ```
  j ← na+mod(cn-nr,N);    // j is first number on or after nr matching cn
  if ((nr ≤ j < nd+RW) and not (j in rbuff.keys)) {
     rbuff[j] ← data;
     while (nr in rbuff.keys)
           nr ← nr+1;
  }
  send [mod(nr,N)];
  ```

Figure 3: Variables and rules of sink.

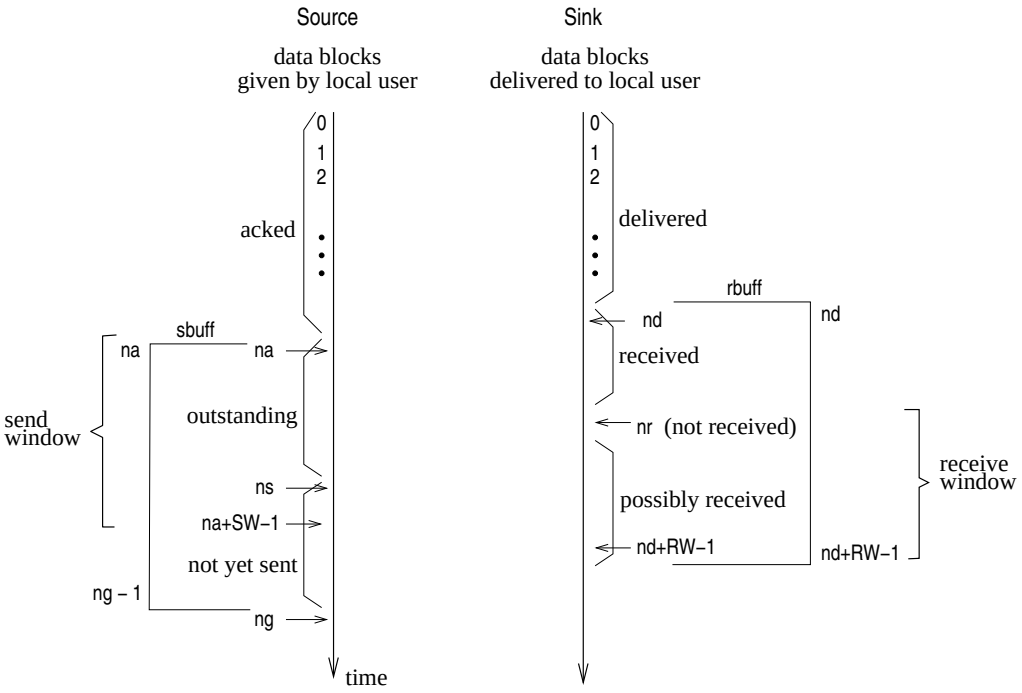

Figure 4: Variables of the sliding window protocol.