CSE 232: Assignment 2

Building a TCP receiver

Due date (Part I and II): Sep 12, 2023

Due date (Part III) : Oct 10, 2023

Total: 40 points

The overall objective of this assignment is to help you build and understand the working of a TCP receiver. The assignment is decomposed into three primary components:

1. Building the ByteStream class.
2. Building a reassembler to assemble back the segments received from the sender in the correct order.
3. Building the actual TCP receiver by stitching together the ByteStream and Reassembler.

Getting Starter code

1. Download and extract the zip file from google classroom.
2. **cd** into the project directory and create a new directory called build.
3. Now **cd** into the build directory and run **cmake ..** to configure the project.
4. Once the project is configured, run **make** in the same directory to build the project.
5. Once the project is built without any errors, you can run **ctest** to run all the tests for the TCP receiver.
6. As you can see, all the tests will fail. Once you have implemented all the classes correctly, all the tests must pass.

Please note that the provided test cases in the code are for your understanding and to help with the implementation. Additional tests will be used for evaluation.

Background on TCP

Generally when we want to retrieve a webpage or want to send an email to someone, the applications rely on a mechanism called *Reliable Byte Stream* to communicate between each other.

Byte Stream is an essential abstraction for applications to send data or communicate with each other. A Reliable Byte Stream ensures that the bytes received at the receiver side are exactly the same and in the same order as sent by the sender.

In reality however, the Internet doesn't provide a service of reliable byte-streams. Instead, the only thing the Internet really does is to give its “best effort” to deliver short pieces of data, called Internet datagrams, to their destination. Each datagram contains some metadata (headers) that specifies things like the source and destination addresses—what computer

system it came from, and what computer system it's headed towards—as well as some payload data (up to about 1,500 bytes) to be delivered to the destination computer.

The two systems have to cooperate with each other to make sure that the bytes in the stream eventually gets delivered, in the correct order to the correct destination on the other side. They also have to tell each other how much data they are prepared to accept from the other system and make sure not to send more than the other side is wdilling to accept. All this is done using an agreed-upon scheme set down in 1981 called the Transmission Control Protocol or TCP.

Part I: Building ByteStream [10 points]

A ByteStream class, as the name itself suggests, is basically a container that stores a collection of bytes from which bytes can be read or written to. In the first part of the assignment, your goal will be to build a ByteStream class that will be used to represent a reliable byte stream.

These are some of the properties of byte stream :

1. Bytes are written on the **input side** and read out from the **output side** (use a data structure that allows pushing the byte from one side and popping from the other side).
2. The byte stream is finite. The writer can end the input and no more bytes can be written.
3. When the reader has read to the end of the stream, it will reach **EOF** (End of File), that is no more available bytes to read
4. Your abstraction will also be initialized with a particular **capacity** which limits the total amount of bytes that can be held in memory at once (which are not read yet).
5. The writer would not be allowed to write into the byte stream if it **exceeds** the storage capacity.
6. As the reader reads bytes from the stream, the writer is allowed to write more.
7. This ByteStream will be used in a single threaded context and therefore you don't have to worry about readers, writers, locking or race conditions.

The interface for the ByteStream is available inside **src/byte_stream.hh**.

- You need to implement the methods of this interface inside **src/byte_stream.cc.** To test your implementation, run **make** inside the build directory to build the project and **ctest -R '^byte_stream'**, to run all the tests associated with byte_stream.

Part II: Building a reassembler [10 points]

In the second part of the assignment, you will be building a special data structure called **Reassembler,** which will be responsible for reassembling the string of bytes obtained from the sender and storing it in the ByteStream.

The TCP sender is dividing its byte stream up into short segments (substrings not more than 1460 bytes apiece) so that they can fit inside a datagram. But as we discussed before, the only thing the Internet does is to give its "best efforts" to deliver the datagram to the receiver. In reality, the network might reorder these datagrams, drop them or deliver them more than once. It

is the job of the receiver to reassemble the segments into a contiguous stream of bytes that the sender originally intended. This is where reassembler comes into play.

The Reassembler receives substrings from the sender which consists of a string of bytes along with an index of the first byte of the string that represents its position within the larger stream.

As a simple example, let's say the sender wants to send “abcdefgh” to the receiver and assume that the sender has divided the entire payload into two substrings **“abcd”** and **“efgh”**.

Now the indexes for these substream will be the following :

[0] : abcd
[4] : efgh

Now on the receiver end lets say the the datagrams are received in following order (efgh,4) -> (abcd,0)

The Reassembler using these unique indexes will paste the substrings into the byte stream in the correct order (**abcdefgh).**

The full (public) interface of the reassembler is described by the **Reassembler** class in the **src/reassembler.hh** header.

- Your task is to implement this class. You may add any private members and member functions you desire to the Reassembler class, but you cannot change its public interface.

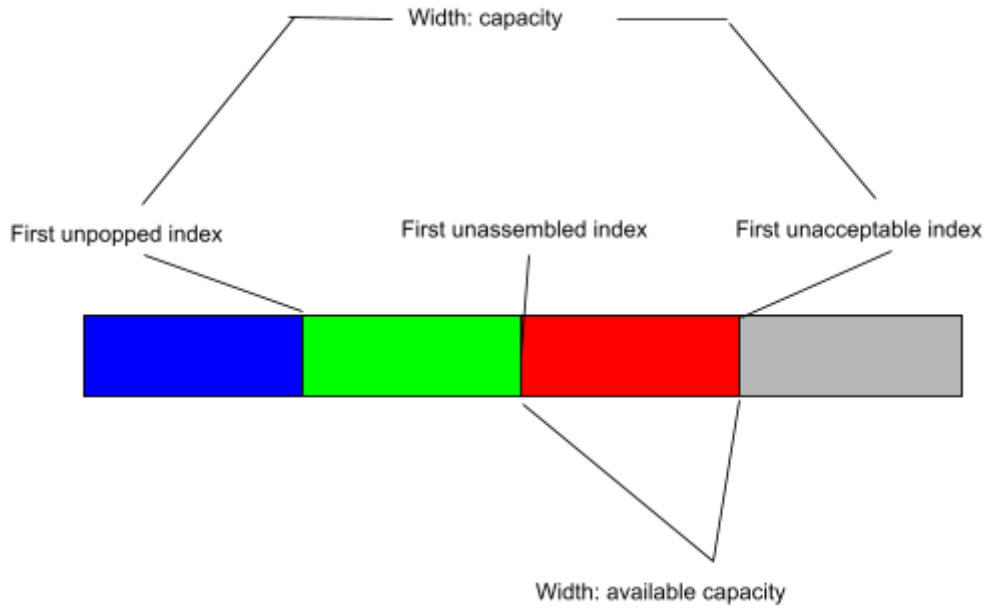
What should the Reassembler store internally ?
The insert method informs the Reassembler about a new excerpt of the ByteStream, and where it fits in the overall stream (the index of the beginning of the substring).

In principle, the Reassembler needs to handle three categories of knowledge :

1. Bytes that are the next bytes in the stream. The Reassembler should push these to the Writer as soon as they are known.
2. Bytes that fit within the stream’s available capacity but can’t yet be written, because earlier bytes remain unknown. These should be stored internally in the Reassembler (buffered basically).
3. Bytes that lie beyond the stream’s available capacity. These should be discarded. The Reassembler will not store any bytes that can’t be pushed to the ByteStream either immediately, or as soon as earlier bytes become known.

The goal of this behavior is to limit the amount of memory used by the Reassembler and ByteStream, no matter how the incoming substrings arrive. We’ve illustrated this in the picture below. The “capacity” is an upper bound on both:

1. The number of bytes buffered in the reassembled ByteStream (shown in green), and
2. The number of bytes that can be used by “unassembled” substrings (shown in red)



Blue: Bytes popped already
Green: Bytes Buffered in stream (in the correct order)
Red: Bytes buffered internally by the Reassembler (for the indexes that cant be sent to the byte stream because the previous indexes have not arrived)
Grey: Unacceptable indexes (More than the capacity of the buffer)

Part III: Building the actual TCP receiver by stitching together the ByteStream and Reassembler [20 points]

In the third part of the assignment, you will implement an actual TCP receiver that accepts a TCPSegment from the sender and utilizes the Reassembler from the previous part of the assignment to correctly write the bytes into the byte stream.

One thing to note is that apart from accepting the bytes, the TCPReciever also needs to generate messages back to the sender. These “receiver messages” are responsible for telling the sender:

1. Index of “**first unassembled**” byte called **acknowledgement number** or **ackno**.
2. The available capacity in the output ByteStream. This is called the “window size”.

Together the **ackno** and **window size** describe the receiver's window: a range of indexes that the TCP sender is allowed to send. Using the window, the receiver can control the flow of incoming data, making the sender limit how much it sends until the receiver is

ready for more.

The main task associated with this part of the assignment is to wire up the Reassembler and the ByteStream class and implement the TCP Receiver interface inside **tcp_reciever.hh**. The hardest part will involve thinking about how the TCP will represent each byte's place in the stream a.k.a **sequence number**.

Note : Placing a breakpoint inside the methods of this interface and debugging the relevant test cases can help you figure out the solution if you are stuck somewhere

Translating between 64-bit indexes and 32-bit seqnos

In the earlier part of the assignment, you built a Reassembler that reassembles substring where each substring has a 64bit stream index with the first byte in the stream always zero. In the TCP headers however, in order to conserve space, the indexes of substring are represented using a 32 bit sequence number. This adds three complexities :

1. **Your implementation needs to plan for 32-bit integers to wrap around**
   Streams in TCP can be arbitrarily long and 2^32 bytes is only 4GB which is not so big. So once the sequence number counts up to 2^32-1, the next byte will have the sequence number 0.

2. **TCP sequence numbers start at a random value**
   In order to improve robustness and avoid getting confused by old segments belonging to earlier connections between the same endpoints, TCP makes sure that sequence numbers can't be guessed and therefore don't start at 0. Instead the first sequence number in the stream is a random 32 bit integer called the **ISN (Iniital Sequence Number).** This is the number that represents the **zero point** or **SYN (beginning of stream).** Rest of the sequence numbers are simply **ISN+x (mod 2^32)** where **x** is the index of byte (1st byte, 2nd byte etc).

3. **The logical beginning and end, each occupy one sequence number**
   TCP also makes sure that the beginning and the end of the stream are received reliably. Thus, in TCP **SYN (beginning of stream)** and **FIN (end of stream)** control flags are assigned sequence numbers. Also **SYN=ISN**. Keep in mind that SYN and FIN aren't part of the stream itself and **aren't bytes.** They represent the beginning and ending of the byte stream itself.
   These sequence numbers (seqnos) are transmitted in the **header of each TCP segment.** It's also sometimes helpful to talk about the concept of an **"absolute sequence number"** (which always starts at zero and doesn't wrap), and about a **"stream index"** (what you've already been using with your Reassembler: an index for each byte in the stream, starting at zero).

To understand the distinction between **sequence number, absolute sequence number** and **stream index** refer to the example below.

Consider the byte stream contains “cat”. If the SYN happened to have seqno 2^32 − 2, then the seqnos, absolute seqnos, and stream indices of each byte are:

element	SYN	C	A	T	FIN
seq_no	2^32-2	2^32-1	0	1	2
abs_seq_no	0	1	2	3	4
stream_index		0	1	2	

Sequence Number :

- Start at ISN
- Include SYN/FIN
- 32 bits wrapping
- “Seq_no”

Absolute Sequence Number

- Start at 0
- Include SYN/FIN
- No wrapping 64bits
- “abs_seq_no”

Stream Indices

- Start at 0
- Omit SYN/FIN
- 64bit No wrap
- “Stream_index”

To perform conversion between **absolute sequence number** and **sequence number,** a new type name **WrappingInt32** contains a method called **wrap** and **unwrap** for performing the conversion.

You need to implement both wrap and unwrap methods inside wrapping_integers.hh and wrapping_integers.cc

You can test your implementation of wrapping integers by running **ctest -R '^wrapping_integers'** inside the build directory.

Finally the objectives of Part 3 of the assignment are :

1. Implement the conversion routines of WrappingInt32
2. Implement tcp_reciever.hh interface

To test the entire implementation of TCP receiver, run **ctest** inside the build folder.