Project #3 - Tips

Reliability

Every time a packet is sent out, you will need to keep track of the time it was sent out. The best way to do this is to keep a queue of values that includes packet information and the estimated timeout period. This estimated timeout period should be at least the Round Trip Time (RTT). If your application does not have an estimation of RTT then a very conservative RTT can be used instead. The timeout period should therefore be:

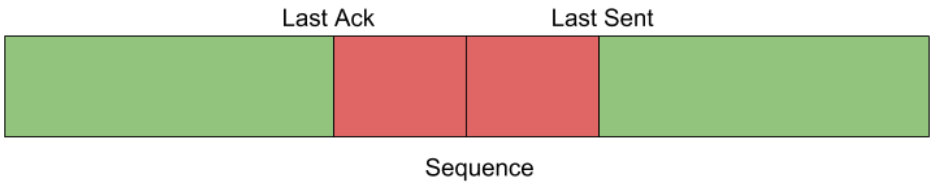
```
timeout = now() + 2*RTT
```

There is a `getNow()` command that is part of the standard Timer interface. After a packet is sent, start a oneshot timer with the delta in time. Once the event is fired, you can pop the head of the queue and check to see if this packet was received by comparing to the `lastByteAcked`. If the data has not been acked yet, immediately retransmit the data. At the end of the fired, restart the timer with the next value in the queue's delta time (use head instead of pop).

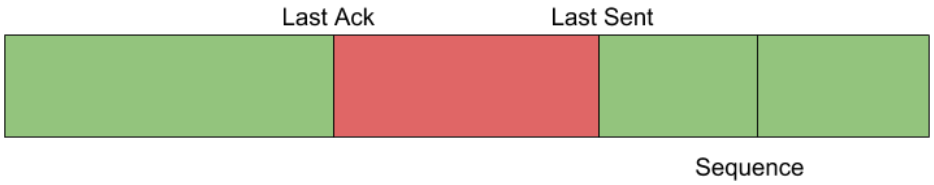
Checking Acknowledgments

Due to the nature of the limited space, there are several cases that can occur that is dependent on were pointers are located. This section depects both cases and how to see if a sequence number was acknowledged. In the figures below, green designates sections which have been acknowledged and the red are sections were data has not been acknowledge. If a sequence falls in a non-acknowledged region when your timers expire, then you will need to retransmit the data.

The Typical Case



The first case, and what is typically seen is where the last sent byte is larger then the last byte acked. The first instinct would be to check to see if the last byte ack is less then the sequence number. This however doesn't take into consideration the ack data after the last byte sent seen on the right hand side. Shown in the figure below.



If it falls in the right hand side of last sent, this case, then it has been acknowledged. The best way to tackle this problem is to stick with checking if it is inside the acknowledge range (green) or stick with checking if it is in the non-acknowledged range (red).

Wrap Around

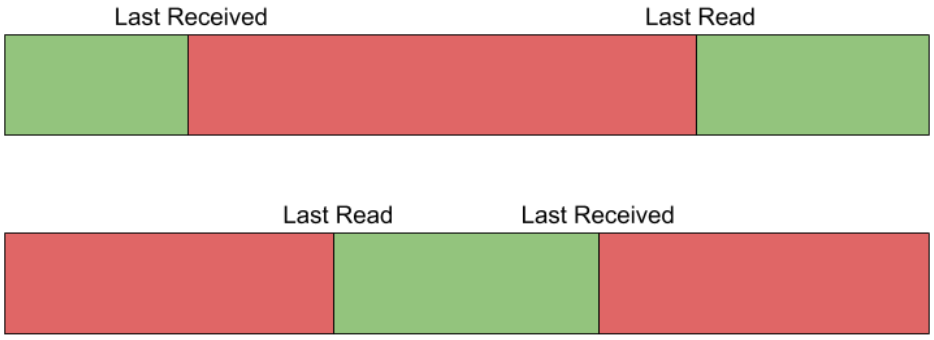
The second case is where the last sent begins to wrap around the buffer. This causes the last sent to be less then the last ack. The acknowledge portion is now located between the last ack and last sent. The two figures below show when a sequnce falls within the acknowledge range and the non-acknowledged range,
respectively.

To check if something is acknowledge you will want to check for

- Which type is it, Type: Typical or Type: Wrap.
- If it is Typical, check to see if it is less then the last ack or greater then the last sent
- If it is Wrap, check to see if the sequence is between the last ack and last sequence.

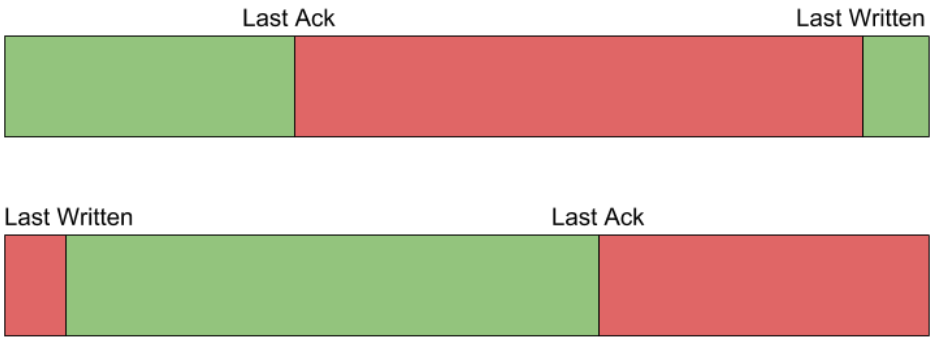
Read/Receive Buffer

Green is the data that can be read. Red is data that has already been read.



Write/Send Buffer

Green is where additional data can be written. Red is where data that has been sent but not yet acked.

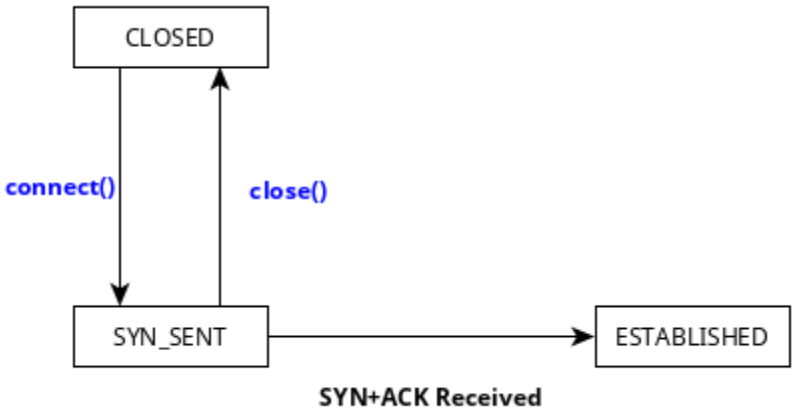


State Diagram

Below are a few simplified graphs based on what is written in *Computer Networks: A System Approach* to hopefully make it clearer what is happening. When a socket is first initated it is in the closed state. In the graphs, functions are in blue and receiving packets are in black.

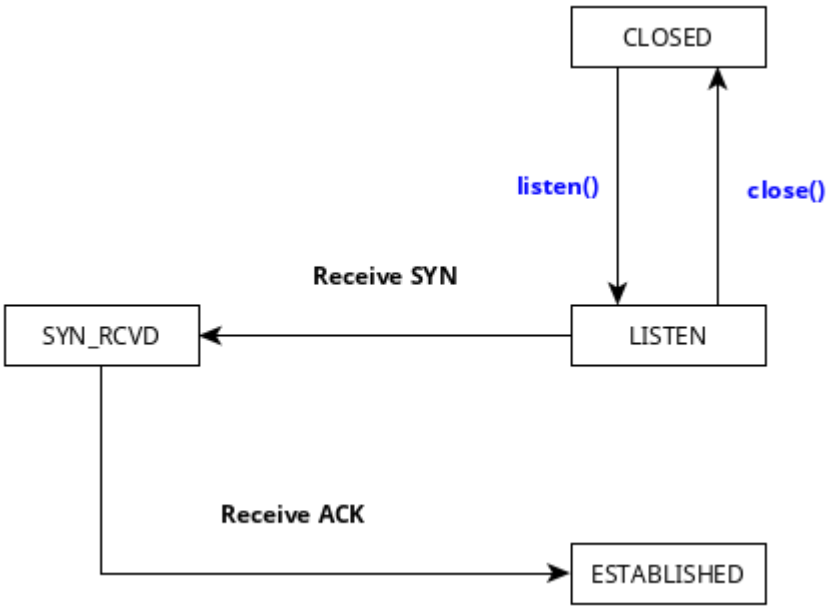
Connections

Client Connection



A connection is started in a client's side by calling the connect function. At this point it will send a **SYN** to the server and wait until it receives a **SYN+ACK** from the server at which point it will go into the **ESTABLISHED**state. This is part of the threeway handshake.

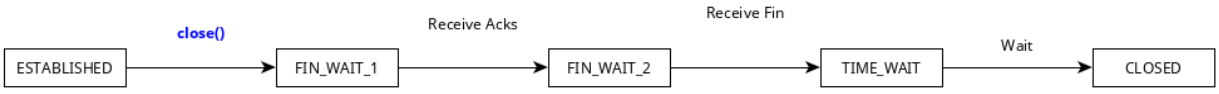
Server Connection



The server goes into the **LISTEN** state when the `listen` command is called. From here the connection waits until a client initiates a connect and sends a **SYN** to the server. Once it receives this **SYN** the server socket forks a new socket which goes to the **SYN_RCVD** state and sends a **SYN+ACK** packet to the client. Once this forked socket receives an **ACK** from the client it will go into the established state. See accept portion to see what happens next.

Teardown

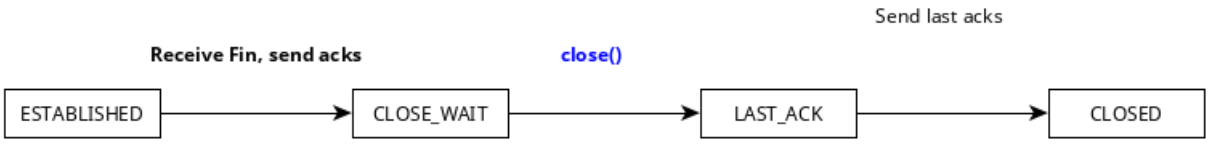
Node Initiating Teardown



A node initates a teardown when the `close` is called. At this point it will send out the remaining data and then a **FIN** packet. It will then wait until it receives the remaining **ACK** which will cause it to go into the **FIN_WAIT_2**state. Once it receives a **FIN** from the other node it will go into the **TIME_WAIT** state and

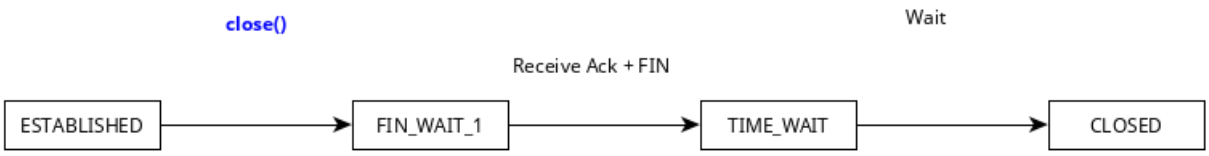
wait a “long” time until it goes to the **CLOSED** state. This wait period allows for a few missed **FIN** packets to be sent.

Recepient of Teardown



On the other hand, the recepient of the intial close will receive the **FIN** and go into the **CLOSE_WAIT** state. This state waits until the application also calls the `close` command. From here, transmit a **FIN** back to the other node and receive the final acks before closing.

Both Nodes Closing



If both nodes are closing then it is a bit simpilar. Once close is called, a **FIN** is sent. Once an **ACK+FIN** is received, it will behave similar to the *Node Initiating Teardown* section at the **TIME_WAIT** state.

Advertise Window

When you initiate a socket, the socket will have an advertised window for the connection set to 0 since it does not have any information about the receivers buffer. Once the connection is initiated in the three way handshake, both connections will exchange information about it’s advertised window. So the SYN will include the initial advertise window, which should be the full buffer size since no data has been transmitted yet and the SYN+ACK will include it’s advertised window. Each packet transmitted should include information about the advertised window through out the connection.

Accept/Connect

In standard implementations both accept and connect are blocking. This means that code execution will not be connected until they are completed. For a system like TinyOS, this is not ideal since other components needs to be called. Since TinyOS is single threaded a blocking system would prevent neighbor discovery, routing, and packet reception from running. To avoid blocking TinyOS programs use a concept of split-phase programming. On the server side, when the server accepts a connection a `accept event` should be `signal`ed. Similarly, the client can `signal` a `connectDone event` when it is done. The above two events should only be signaled when the connection is **ESTABLISHED**.

Application Layer

Once a connection is **ESTABLISHED**, the Transport layer will `signal accept`. On the server side you will keep track of multiple sockets that have been accepted. There are a few ways of handling the client portion of this project, one being the timer, the second being post/task.

Timer Method

In this method, your application layer will periodically write to the socket. For each connection keep track of a counter and write the entirity of the buffer.

Task Method

In the Task Method, the application will post a task every time it is successfully able to write data (i.e. write returns a value greater then 0). This allows for other components in the applications to run in between writes.

The above two methods can be used for the client layer but rather then writting the client should be reading data. So if you were to use the Timer

Method you would use the `read` function every few seconds and print it to the transport debug channel.

Notes on Testing

Make the assumption that we are contacting a server at a well known address (mote 1) and at a well known port (port 123). There will be multiple connections to this server.

For testing purposes you will be writing incrementing numbers from the server to the client or vice versa. The recipient of the data should immediately print the data.

Once you write all of the data from the client side, close the connection.

For a single connection, the clean output should be:

```
0:4:55.379883898 DEBUG (8): Reading Data:1,2,3,4,5,6,
0:4:57.333008898 DEBUG (8): Reading Data:7,8,9,10,11,12,
0:4:59.286133898 DEBUG (8): Reading
Data:13,14,15,16,17,18,
...
0:11:16.239258898 DEBUG (8): Reading
Data:955,956,957,958,959,960,
0:11:18.192383898 DEBUG (8): Reading
Data:961,962,963,964,965,966,
0:11:20.145508898 DEBUG (8): Reading
Data:967,968,969,970,971,972,
0:11:24.051758898 DEBUG (8): Reading
Data:973,974,975,976,977,978,
0:11:26.004883898 DEBUG (8): Reading
Data:979,980,981,982,983,984,
0:11:27.958008898 DEBUG (8): Reading
Data:985,986,987,988,989,990,
```

```
0:11:29.911133898 DEBUG (8): Reading
Data:991,992,993,994,995,996,
0:11:31.864258898 DEBUG (8): Reading
Data:997,998,999,1000
```

The above has been truncated. For two connections your output should be:

```
0:4:55.379883898 DEBUG (8): Reading Data:1,2,3,4,5,6,
0:4:57.333008898 DEBUG (8): Reading Data:7,8,9,10,11,12,
0:4:57.393555240 DEBUG (4): Reading Data:1,2,3,4,5,6,
0:4:59.286133898 DEBUG (8): Reading
Data:13,14,15,16,17,18,
0:4:59.346680240 DEBUG (4): Reading Data:7,8,9,10,11,12,
0:5:1.239258898 DEBUG (8): Reading
Data:19,20,21,22,23,24,
0:5:1.299805240 DEBUG (4): Reading
Data:13,14,15,16,17,18,
0:5:3.252930240 DEBUG (4): Reading
Data:19,20,21,22,23,24,
0:5:5.145508898 DEBUG (8): Reading
Data:25,26,27,28,29,30,
0:5:7.098633898 DEBUG (8): Reading
Data:31,32,33,34,35,36,
0:5:9.051758898 DEBUG (8): Reading
Data:37,38,39,40,41,42,
```

Note that although the output is intertwined, if they were separated, they both will print values that are in order. Print the above with `grep (4)` will give you:

```
0:4:57.393555240 DEBUG (4): Reading Data:1,2,3,4,5,6,
0:4:59.346680240 DEBUG (4): Reading Data:7,8,9,10,11,12,
0:5:1.299805240 DEBUG (4): Reading
Data:13,14,15,16,17,18,
```

```
0:5:3.252930240 DEBUG (4): Reading
Data:19,20,21,22,23,24,
```

Misc

dbg Tips

```
uin32_t a;
uin16_t b;
uin8_t c;

// 32 bit unsigned integer
dbg(CHANNEL, "%u", a);

// 16 bit unsigned integer
dbg(CHANNEL, "%hu", b);

// 8 bit unsigned integer
dbg(CHANNEL, "%hhu", c);
```