TCP Implementation for SWANS

Overview

The goal of the project is to provide SWANS with TCP layer that can be used for TCP applications such as file transfer, web servers, etc. All of the coding is done using java, and the codes are compiled using java compiler and rewritten using JIST.

The TCP layer includes all of the basic functionalities such as connection establishment, data transfer, and connection tear-down. It also supports flow control, congestion control, and retransmission.

Implementation ideas are obtained from RFC 793 and RFC 2581. For more in-depth explanation of how TCP works, please refer to the book "TCP/IP Illustrated, Volume 1: The Protocols" by W. Richard Stevens chapters 17 – 23.

Components

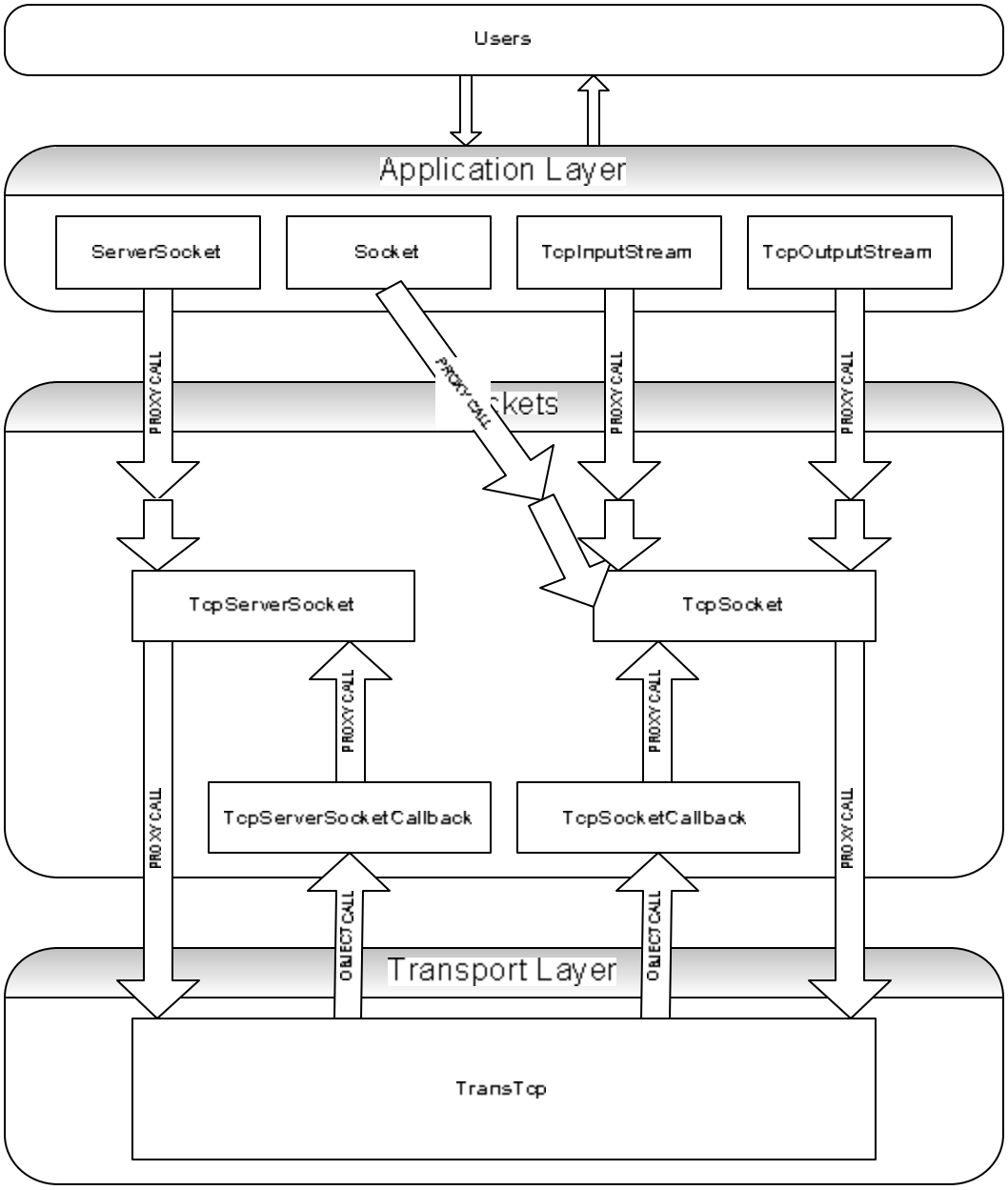
This project has 3 components or entities: the application layer, the transport layer, and the socket that connects these two layers. The application layer is built to handle requests from the application (or user), and then it will pass the requests to the transport layer. The request can be to send or to retrieve bytes. The application layer cannot send the requests directly to the transport layer, so a socket is needed to handle all of the low level implementation that are hidden from the users. After a socket receives a request from the application, it will do all of the necessary steps to ensure that the request is handled. The socket also handles requests from the transport layer. The requests are in the form of incoming packets. So, the socket will manage all of the incoming and outgoing packets and will inform the application layer (or the transport layer) when bytes are ready to be read (or packets are ready to be sent).

For each of these components, there are many objects that handle different things. In the application layer, ServerSocket and Socket are written to handle requests for establishing and terminating connections. InputStream and OutputStream classes are base classes for I/O operations to the socket. TcpInputStream and TcpOutStream derive from these base classes to handle I/O operations to our custom socket implementations.

Socket entities also have many objects. There are two different types of sockets, TcpServerSocket and TcpSocket. TcpServerSocket implements the logic for a server socket (server), while TcpSocket implements the logic for a socket (client). Two other classes are implemented to help the transport layer to communicate with these sockets, namely TcpServerSocketCallback and TcpSocketCallback.

The transport layer has two different entities, one for TCP and one for UDP. This project only concerns about the TCP implementation. The TCP layer, TransTcp, is used by the sockets to send and receive packets. TcpSocket will pass packets to TransTcp to be delivered to the other side, and TransTcp will call TcpSocketCallback to pass incoming packets to TcpSocket.

The following is a diagram of how they are connected:



Code

The codes for this project are in the following files:

- jist/swans/app/net/ServerSocket.java
  Implements application's server socket
- jist/swans/app/net/Socket.java
  Implements application's socket
- jist/swans/app/io/InputStream.java
  Implements class to replace java library's InputStream
- jist/swans/app/io/OutputStream.java
  Implements class to replace java library's OutputStream
- jist/swans/trans/TransInterface.java
  Interfaces to be implemented by transport layer
- jist/swans/trans/TransTcp.java
  Implementation of transport layer (TCP) entity
- jist/swans/trans/SocketInterface.java
  Interfaces of the socket and server socket entities
- jist/swans/trans/TcpServerSocket.java
  Implements server socket entity
- jist/swans/trans/TcpSocket.java
  Implements socket entity
- jist/swans/trans/TcpInputStream.java
  Implements socket's input stream
- jist/swans/trans/TcpOutputStream.java
  Implements socket's output stream
- jist/swans/trans/CircularBuffer.java
  Helper class that implements byte buffer
- jist/swans/trans/PriorityList.java
  Helper class that implements sorted list

The files in the jist/swans/app/ directory are created for the JiST Rewriter to replace the original java classes and therefore reside in the application layer. TcpInputStream and TcpOutputStream are derived from InputStream and OutputStream.

TransInterface and TransTcp are files that implement the transport layer. The interface is created because the transport layer is regarded as an entity in this implementation.

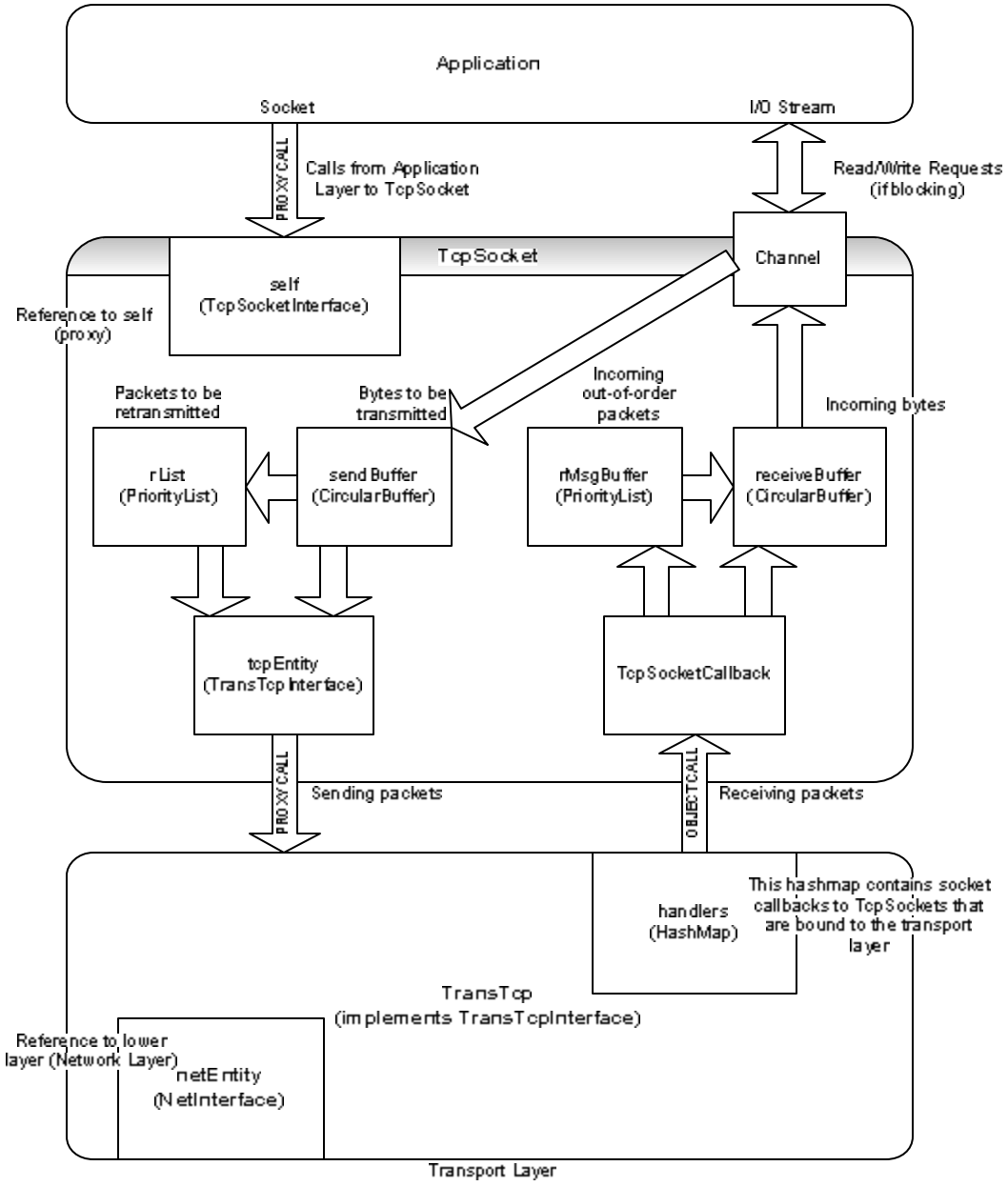
The rest of the files are responsible for the logic of TCP. TcpServerSocket and TcpSocket contain all of the important parts of TCP logic. TcpServerSocket implements the server side and therefore can only accept SYN packets to initiate connection. After receiving SYN packet, it will spawn a TcpSocket to take care of the rest. TcpServerSocket has an inner class called TcpServerSocketCallback, which is used by TransTcp to contact TcpServerSocket.

TcpSocket contains all of the TCP logic for client, which means it has all of the TCP functionalities such as flow control, retransmission, and congestion control. At the beginning of the file, there are some constants that are used by the TCP logic, for example, the maximum segment size and the initial window size. Like TcpServerSocket, TcpSocket also contains an inner class called TcpSocketCallback. Although this callback class is incorporated in TcpSocket, the objects of TcpSocketCallback are actually used by TransTcp, and method calls to TcpSocket from TcpSocketCallback are proxy calls, not direct object calls. When a TcpSocket object is bound to a port, it will register its socket callback (TcpSocketCallback object) with TransTcp so that the transport layer knows which TcpSocket is bound to which port and where to forward incoming packets.

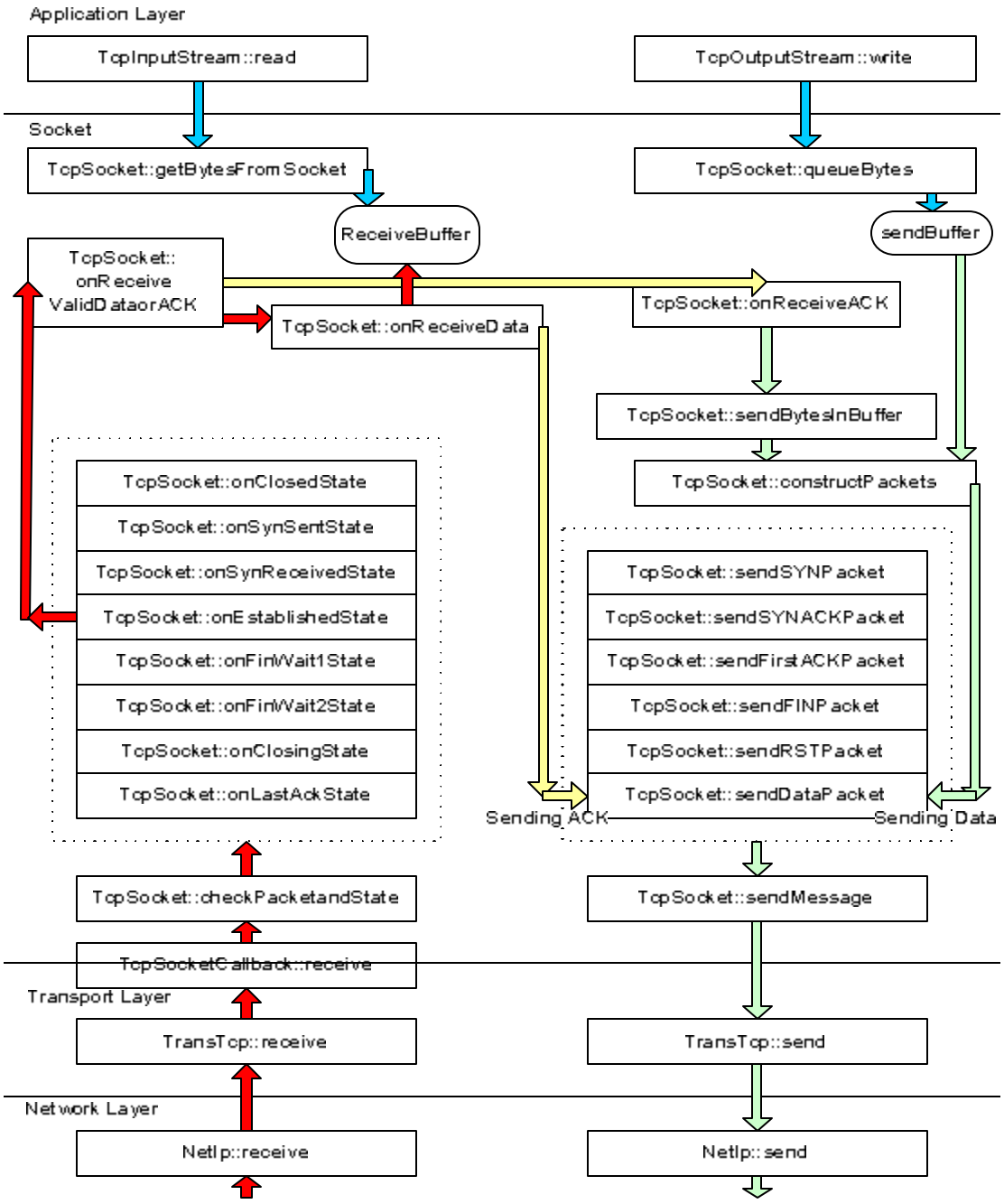
Two new classes, CircularBuffer and PriorityList, are created to help with the TCP logic implementation. CircularBuffer is used to store temporary bytes before being sent to the lower layer or being read by upper layer. This object has the capability to store and retrieve bytes, and also resize itself. PriorityList is used to store data packets that have been transmitted but not yet acknowledged, or to store out-of-order data packets. The priority is based on the sequence numbers of the stored packets.

Besides these classes, two more files are also changed. One is Rewriter.java, which be explained in more details in later section. The second one is Constants.java. This file stores all of the constants used for TCP states.

Below is a diagram of objects contained within a client socket (TcpSocket) and its connection with the application layer and the transport layer:



The next diagram explains the flow of the program during sending and receiving data and acknowledgement:



The arrows with red color indicates the execution flow when a packet is received. Then, in method onReceiveValidDataorACK in TcpSocket, it is divided into 2 paths. The first path is when data packet is received (still in red). The second path is when ACK is received, denoted by yellow arrows. From the diagram, it can be observed that after receiving data, the arrow becomes yellow since TCP will send acknowledgement packet after receiving data.

Data packet sending is induced by incoming acknowledgement packet, which means more data packets should be sent. The blue arrows represent executions caused by the application layer.

Here are some of the important methods:

- TcpSocket::checkPacketandState
  When a packet is received by the socket, it will be passed to this method. This method then decides, based on the current state of the socket, what to do with the packet. Depending on the current state, a particular method is called to process the incoming packet.

- TcpSocket::isPacketOutofWindow
  This method is used by a few other methods. Its function is to check if incoming packet is within the socket's receiving window.

- TcpSocket::isAcceptableACK
  This method is also used to evaluate incoming packets. It checks whether the acknowledgement packet received is acknowledging data in the socket's send window.

- TcpSocket::onReceiveData
  This method is called when a data packet is received. This method will check if the socket advertised zero window in the last sent acknowledgement packet. If it did, that means the received packet is a probe message from the other side. To avoid silly window syndrome, receiver window is not updated as soon as it is available, unless the available space is bigger than one maximum segment size or three probe messages have been received. This probe messages are monitored in this method. This method also sends acknowledgement packets back to the sender. Finally, this method also checks if the incoming packet fills in the blank space in the receiving window (when subsequent packets have been stored because they have arrived out-of-order).

- TcpSocket::onReceiveACK
  This method is called when an acknowledgement packet is received. This method checks if the socket receives ACK packet with zero advertised window, and will send back a probe message if it does. Otherwise, it will cancel the persist timer (used to send probe message) and check if the packet is a duplicate ACK. This is where the congestion control algorithm is executed.

Features

The implementation of TCP for SWANS includes the following features (listed in the order they were implemented):

1. *Connection Establishment*
   From the application layer, this is done when the connect method of Socket is called. The Socket object will call connect method of TcpSocket. The connect method will then set the remote address, send the SYN packet, change the current state, and block the application to wait for SYNACK packet.
   At the other end (server end), the application must have called listen method before the server can begin listening for incoming SYN packets. Calling listen method from ServerSocket will result in calling listen method of TcpServerSocket. Since listen method must block, a channel is used for this purpose. Upon receiving SYN packet, the server will create a new TcpSocket entity, bind the new TcpSocket object to a random port, send a SYNACK packet back, and block to wait for ACK packet. Since the new TcpSocket object is created by the transport layer (inside TcpServerSocket), a reference this new object has to be returned and this is done by sending the object to the application through the channel. These are all done in checkPacketandState method in TcpServerSocket.
   The client side blocks until SYNACK packet is received. When SYNACK packet arrives and the socket is in the correct state (SYN SENT state), the method onSynSentState is called to take care of sending ACK back and update the TCP variables, including sequence number, acknowledgement number, and receiver window. This method also unblocks the application.
   The new TcpSocket object (the one created by the server) will be waiting for the first ACK packet during the SYN RECEIVED state. Upon receiving the correct packet, onSynReceivedState method is called to unblock the application and update TCP variables. Since the first ACK packet can be piggybacked with data, this method must also check that.

2. *Connection Termination*
   Connection termination is initially initiated by the application by calling the close method. This method will check if the socket still has data pending to be sent. If it does, the socket will schedule for a FIN packet to be sent after all data are sent. Thisi done by setting the variable isClosing to true. Otherwise, it will send the FIN packet directly, done by calling method initiateClosingConnection.
   For the passive side, receiving FIN packet can happen in different states. The most common case is in method onEstablishedState. When the socket receives a FIN packet, it will update its TCP variables (including its state) and send an ACK back. Another case is when the socket actually has sent FIN packet and received ACK back, and is currently waiting for FIN packet from the other side. All of these cases can be looked up from RFC 793.

3. *Flow Control*
   The implementation of flow control is scattered in a lot of the methods, since this involves constantly updating TCP variables such as snd_una (unacknowledged send),

snd_nxt (next byte to be sent), and receiver window. The integral part of flow control is contained within method contructPackets, which is called by method sendBytesInBuffer. Basically, constructPackets method determines the size of the data packet to be sent. This is done by comparing the size of the socket's send window to the size of the receiver window. After constructing a packet, it will call itself to schedule another method call. This loop stops when the other side's receiver window is full.

4. *Persist Timer*
   Most of the implementation of persist timer are in methods startPersistTimer and persistTimerTimeout. Before the socket calls startPersistTimer, it creates a probe message and stores it in the retransmit queue. startPersistTimer method basically sleeps for a predetermined time and then calls persistTimerTimeout. In the timeout method, the probe message is retrieved from the queue and sent to the other side. After that, it schedules another timeout by calling itself. The probe message itself is created in method onReceiveACK, in particular when an acknowledgement with zero advertised receiver window is received. If the socket receives a nonzero window advertisement, the persist timer is cancelled by calling cancelPersistTimer method.

5. *Retransmission*
   Retransmission is similar to persist timer, although more complicated. It is also implemented in two methods, startRetransmitTimer and retransmitTimerTimeout. Since the wait interval before retransmission is different for each try, the wait time is part of the method parameter. startRetransmitTimer basically waits for the specified amount of time and then calls retransmitTimerTimeout. In the timeout method, it will call onRetransmit to retransmit a packet, and then schedule another retransmission for the same packet but with longer wait time. After the wait time reaches certain interval, the socket is closed (see TCP specification for details).
   A lot of methods schedule retransmissions. Basically, these methods are the ones sending data or special packets such as SYN and FIN packets. These methods are constructPackets, sendSYNPacket, and sendFINPacket. Retransmission is scheduled by putting the packet into the retransmit queue and then calling startRetransmitTimer and passing the sequence number to the method parameter. The sequence number is later used to retrieve the packet from the retransmit list.
   The retransmission timer is automatically cancelled by removing the message from the retransmit queue whenever the message is acknowledged by the other side. This is done by calling method removeMessages of PriorityList in method onReceiveACK.

6. *Congestion Control*
   The heart of congestion control is in the method onReceiveACK in TcpSocket. This method will check whether the socket has been receiving duplicate acknowledgements or just regular acknowledgements. If it receives duplicate acknowledgement, it will increment the duplicate ACK counter. This is needed to execute the congestion control algorithm. Depending on these cases, the appropriate methods for congestion control algorithm are called, namely SlowStart, CongestionAvoidance, and FastRetransmit (See TCP/IP Illustrated for details).

Testing

During implementation, each of the features has been individually tested before moving on to the next feature. After a feature is incorporated, the whole TCP logic is re-tested to make sure everything still works fine after integration.

How to Use

To use TCP implementation, a test program is provided. The test program is located in the driver directory. Number of bytes can be specified in the parameter to simulate how many bytes will be sent using TCP.

To run the test program, simply type:

jist –c off.properties jist.swans.Main driver.tcptest <n>

where n is the number of bytes. 250 bytes is set at the default for n (if n is not specified).

And also, currently, debug printouts are disabled. To enable it, go to TcpSocket.java and change the PRINTOUT static variable to the desired choice. There are 4 choices: OFF, INFO, TCP_DEBUG, and FULL_DEBUG. OFF turns off all printouts. INFO provides minimum output. TCP_DEBUG will give you printouts of sent and received packets. FULL_DEBUG will provide complete information of the execution flow.

Interesting Things

One of JiST features, channel, is heavily used for this implementation. This is because the application layer needs to block when it tries to read from the socket. A channel can be used to simulate blocking or non-blocking send and blocking receive. In the TCP implementation, each socket has its own channel since each is connected to different input and output streams. These streams can be used once the socket is connected to the other side.

When the application writes to the output stream, the socket must check its buffer to ensure that it does not receive more bytes than it can store. The channel is used to block the application until the socket receives all bytes. Similarly, when the application wants to read from socket through the input stream, the application is blocked whenever the read buffer is empty.

Another interesting thing is a new phase in the rewriter to support this TCP implementation. During the early stage of the implementation, all of the classes were derived from the classes of same functionalities from the java library. It turns out that this will not work since the original java classes are not blocking and therefore do not support any methods that have return values (any return values will be substituted with null

values when these classes are used under JiST). Therefore, these classes need to be substituted to custom classes. These classes include ServerSocket, Socket, InputStream, and OutputStream.

Since not all of java classes need to be substituted, the JiST rewriter is modified to handle these special cases. The change is to mark these classes in the rewriter so that the rewriter knows that these classes need be rewritten to use our custom implementation instead. This is done to preserve one of JiST properties that the programming in JiST can be done using java library. This also leads to the ability of running any ordinary network applications using SWANS without any modifications.

If, in future implementation, more java classes are used, these classes will also need to be rewritten. The following is the steps to mark a class in the rewriter:

1. open Rewriter.java in jist/swans directory.
2. go to the method: public JavaClass process(JavaClass jcl).
3. In this method, add a line for your new class.

Finally, _jistPostInit method in TcpServerSocket and TcpSocket is also worth mentioning. The sole use of this method is during creation of these objects when these objects are created by objects from different entities. Since constructors are also rewritten during the rewriting phase, constructors cannot have any calls to blocking methods. The use of _jistPostInit is to separate these calls to blocking methods from the constructor. However, note that _jistPostInit will not be called automatically when the objects are created in the same entity or created using direct object call.

Future Work

None of the TCP options are implemented. For example, the TCP implementation can be extended to support selective acknowledgement (SACK). Currently, the TCP implementation will buffer out of order packets and will send duplicate acknowledgement without giving extra information which packet was received. Another possibility is to implement window scale option, which is used to extend the window size so that the advertised window can be larger than 64 kilobytes.

References

RFC 793 “Transmission Control Protocol”
RFC 2581 “TCP Congestion Control”
“TCP/IP Illustrated, Volume 1” by W. Richard Stevens