CSE 150 - Operating Systems
Documentation #3

Spring 2018 - Lab 04 - Group 1

Avery Berchek, Aleksandr Brodskiy, David Cabral, Christopher DeSoto,
Adiam G-Egzabher, Nanditha Embar, Christian Vernikoff

May 10, 2018

Outline

1. Documentation
2. Design Decisions
3. Testing/QA
4. Design Questions
5. Team Member Work-Log
6. Conclusion

Documentation

Task I

The devised approach to developing an efficient, sound, and complete transport control networking protocol was to dichotomize the implementation process into smaller abstractions such that the amalgamation of their respective functionalities would provide a successful transmission control paradigm. In this manner the abstractions used to establish TCP are as follows:

1. The implementation of handling ACCEPT and CONNECT syscalls.
2. The overriding of the READ function of OPENFILE for the new TRANSPORTFILE class.
3. The overriding of the WRITE function of OPENFILE for the new TRANSPORTFILE class.
4. The overriding MAILMESSAGE class to support TCP header fields.
5. The implementation of handling connection teardown with the CLOSE function.

With these abstractions it becomes much easier to implement the deliverables of a successful TCP. Therefore the ACCEPT and CONNECT syscalls were set−up with data structures associated with managing all the connections. Within this set−up the essence of the function ADDCONNECTION is in creating a new hash element to be inserted to the connections hash structure to represent a new connection that is being created. This is instantiated in the following data structure:

HASHSET STRUCTURE CONNECTIONS;

The manipulation of the CONNECTIONS data structure by the ACCEPT and CONNECT functions is modeled below in the following pseudo−code.

```
int handleAccept(int port)
{

  if (!pendingConnections[port].empty())
    TransportFile tcp = pendingConnections[port].poll();
    Send(SYN/ACK message to other host);
    TCP.acquireLock();
    TCP.state = ESTABLISHED;
    TCP.releaseLock();
    if (tcp!= null)
      for I to openFiles.length
        if (openFiles[i] == null)
          openFiles[i] = tcp;
          return I;
        end if
      end for
    end if
  end if
  return -1;

}
```

```
int handleConnect(int host, int port)
{
  int myPort = NetKernel.reservePort();
  TransportFile tcp = new TransportFile utilizing new myPort,host,port values
  Tcp.Send(SYN message to host, reliably(track the packet))
  Tcp.state = SYN_SENT;

  NetKernel.connectionsLock.acquire();

  while(tcp.state != ESTABLISHED)
        NetKernel.pendingConnectionsCond.sleep();
        NetKernel.connectionsLock.release();
        if (tcp!= null)
          for I to openFiles.length
            if (openFiles[i] == null)
                openFiles[i] = tcp;
                return I;
            end if
          end for
        end if
        return -1;
}
```

As is observable from the pseudo−code, the ACCEPT and CONNECT functions enable the connections to TCP ports on the operating system through the two−way

handshake SYN and ACK packet flag interchanges. However, it is also taken into consideration in the pseudo−code that there is an adjustment made to the packets being sent throughout the network that acknowledge what type of packet they are; for example, for an ACK packet flag the receive function would be modeled as follows. The CONNECT function, with respect to the accept This functionality is very similar to ACCEPT with the preliminary discrepancy that it puts the thread that calls it to sleep, in this manner waiting for the CONNECTion to be established. The method in which threads are put to sleep is by each respective thread acquiring a static lock that is within the NETKERNEL.***java*** file, then a check is performed whether or not the connection has been established and if it hasn't then the threads are put to sleep on a condition variable. This allows multiple threads to be put to sleep to wait for a connection establishment. Consequently, the functionality of the ACCEPT/CONNECT in RECEIVE is as follows:

```
void receiveTask()
{
  Socket s = connection that message is a associated with
  if (s = null && receivedMessage just has SYN flag)
    TransportFile connection = new connection trying to be made
    Connection.state = SYN_RCVD;
    NetKernel.pendingConnections[receivedMessage.destPort].push(connection);
  end if
  else if (s != null)
        if (receivedMessage has only SYN/ACK flags)
          s.tcp.state = SYN_RCVD;
          NetKernel.connectionsLock.acquire();
          NetKernel.pendingConnectionsCond.wakeAll();
          NetKernel.connectionsLock.release();
        end if
  end if
}
```

The overriding of the READ/WRITE functions for the NETKERNEL.*java* class was implemented with the following definitions for the HASHMAP data structure:

HASHMAP < STRING, SOCKET > CURRENTCONNECTIONS;

Where the STRING establishes both the local and remote ports as well as the remote machine address with the following variables:

LOCALPORT
REMOTEMACHINEADDRESS
REMOTEPORT

The SEMAPHORE is introduced in the HASHMAP instantiation to notify when a packet has arrived.

In this manner the functionality for NOTIFYing and CONNECTing are as follows:

```
boolean registerConnection(String identifier, Semaphore notify)

        called when a new TransportFile is initialized
        identifier: "<localPort>,<remoteMachineAddress>,<remotePort>"
        notify: Notify semaphore in TransportFile to notify it when a packet arrived
        returns false if there is already an entry matching that identifier, true otherwise


boolean unregisterConnection(String identifier)

        called when a new TransportFile is closed
        identifier: "<localPort>,<remoteMachineAddress>,<remotePort>"
        returns false if there is no entry matching that identifier, true otherwise


Semaphore getNotify(String identifier)

        called when reliablePostOffice.java receives a packet and wants to notify the dest
        identifier: "<destPort>,<srcMachineAddress>,<srcPort>"
        returns Notify of TransportFile if entry is present, null otherwise
```

The implementation of the desired functionality for the TRANSPORTFILE.*java* class was developed in the internal classes:

```
public class Event
{

        public static const int RECV = 0,
                SEND = 1,
                CLOSE = 2,
                TIMER = 3,
                SYN = 4,
                DATA = 5,
                ACK = 6,
                STP = 7,
                FIN = 8,
                FINACK = 9;
        public Event(int initEvent);

}


public class State
{

        public static const int CLOSED = 0,
                SYN_SENT = 1,
                SYN_RECVD = 2,
                ESTABLISHED = 3,
                STP_SENT = 4,
                STP_RCVD = 5,
                CLOSING = 6;
        private int currentState;
        public State() {currentState=CLOSED;}
        public int transition(Event e)
   // follow state change diagram and return resulting currentState
        public int getCurrentState() { return currentState;}

}
```

With the following data structures:

- TRANSPORTFILE.JAVA
  - HASHMAP<STRING,SOCKET> CURRENTCONNECTIONS
- NETKERNEL.JAVA
  - CONDITION PENDINGCONNECTIONSCOND
  - LOCK CONNECTIONSLOCK
  - LINKEDLIST<TRANSPORTFILE>[] PENDINGCONNECTIONS
  - ARRAYLIST<INTEGER> AVAILPORTS
- NETPROCESS.JAVA
  - OPENFILES[] OPENFILES

From the internal ***java*** class, the implementation of handleReceive, Read, and Write was predominantly involved with the manipulation of bytes[] array in which the packets addressed and received were successfully transmitted.

This is demonstrated in the following excerpt of code:

```
void handleRecv() {
        while true {
                Notify.P();
        }
}

int write(bytes[] toWrite) {
        bytesWritten = 0;
        while bytesWritten < toWrite.size() {
                structLock.acquire();
                if unAckEd.size() >= 16 then structLock.release() and return error code;
                create reliableMessage with chunk of data, offset at bytesWritten
                sendMessage and add seqNum,
                        reliableMessage to unAckEd,
                                and add currentTime() + 20000,
                                        seqNum to unAckEdResendTimes
                seqNum++,bytesWritten+=bytesPacked into reliableMessage
                structLock.release();
        }
        return bytesWritten;
}

int read(bytes[] bytesRead) {
        bytes[] toRet;

}


resendOperation() {
        Alarm thisResendTimer;
        while true {
                bool needToResend = false;
                structLock.acquire()
                needToResend = unAckEd.size() > 0;
                structLock.release()
                if(needToResend) {
                        send unAckEd.MinElem
                        structLock.acqiure()
                        unAckedResendTime.first = currentTime+20000 then rebalance
                        int sleepTime = max(0,unAckedResendTime.first-currentTime);
                        structLock.release()
                        thisResendTimer.waitFor(sleepTime);
                }
        }
}
```

Within the WHILE(***true***) loop of the HANDLERECV function, the NOTIFY.P(); line signifies that a packet addressed to this was received, read the RELIABLEMESSAGE from RELIABLEPOSTOFFICE, and follows the ***nachos*** transport protocol for control packets; Otherwise the STRUCTLOCK is acquired and if this is an ACK then the corresponding entry in UNACKED and removed and unAckEdResendTimes then releases the STRUCTLOCK lock. However, if this packet in transmission is categorized

as data, then it is added to the dataReceived buffer. Then, the structLock lock is released to send an ACK.

The overriding of the MailMessage.***java*** class to support the construction of transmission control packet header fields was done in the following manner with two constructor for the TCP.***java*** class:

TCP(***MailMessage***, ***seqNum***, ***ackNum***, ***controlSegment***);
TCP(***MailMessage***, ***seqNum***, ***ackNum***, SYN, ACK, FIN, STP);

```
class TCPMessage {

    constructor -> TCP(MailMessage, seqNum, ackNum, controlSegment)

    constructor -> TCP(MailMessage, seqNum, ackNum, SYN, ACK, FIN, STP)

            function get_value_from_control_block()->
        // gets the bit from the primitive (needs to be small to fit, so will use bit manipulation)

        Syn = controlSegment[12]
        ACK = controlSegment[13]
        FIN = controlSegment[14]
        STP = controlSegment[15]

            function create_control_block()->
        // generates control block from control bits (bit manipulation)

                controlSegment[12] = SYN

                controlSegment[13] = ACK

                controlSegment[14] = FIN

                controlSegment[15] = STP

        return (short) controlSegment
}
```

The overloading of the TCP constructor allowed for the simplified addition of TCP based headers for reliability.

Lastly, the implementation of the close function syscalls consummated the TransportFile.***java*** class as it facilitated a closing schematic for the connection and therefore provided the inverse two−handshake for the TCP teardown. Within this functionality when close is invoked, if the connection is still transmitting data then a STP packet will be sent to let the remote host know to stop sending data and just wait until the host that called close to stop sending data. Once the data transmission is complete, and no longer acknowledged, then a FIN packet will be sent out to

let the remote host know that the connection is closing. A FIN/ACK will be sent back from remote host so once received the connection will finally be CLOSEd and resources will be deallocated.

This is demonstrated in the CLOSE in receive function:

```
void receiveTask()
{
        Socket s = connection that message is associated with
        if (receivedMessage has just stp flag set)
                if (state == ESTABLISHED)
                else if (state == STP_SENT)
                else if (state == CLOSING)
        if (receivedMessage has just ack and not in state == ESTABLISHED)
                if (state == STP_SENT)
        if (receivedMessage has just fin flag set)
                if (state == ESTABLISHED)
                else If (state == STP_SENT || state == STP_RCVD)
                else If (state == CLOSING)
                else If (state == CLOSED)
        if (receivedMessage has just data flag and not in state == ESTABLISHED)
                if (state == STP_SENT)
                else if (state == CLOSING)
        if (receivedMessage has ack and fin flag and state == CLOSING)
                State = CLOSED
}
```

The pseudo−code for the TCP teardown CLOSE function is as follows:

Task II

The implementation of a relay chat client−server application atop the implemented transmission control protocol was developed with functionality similar to that of aforementioned syscalls.

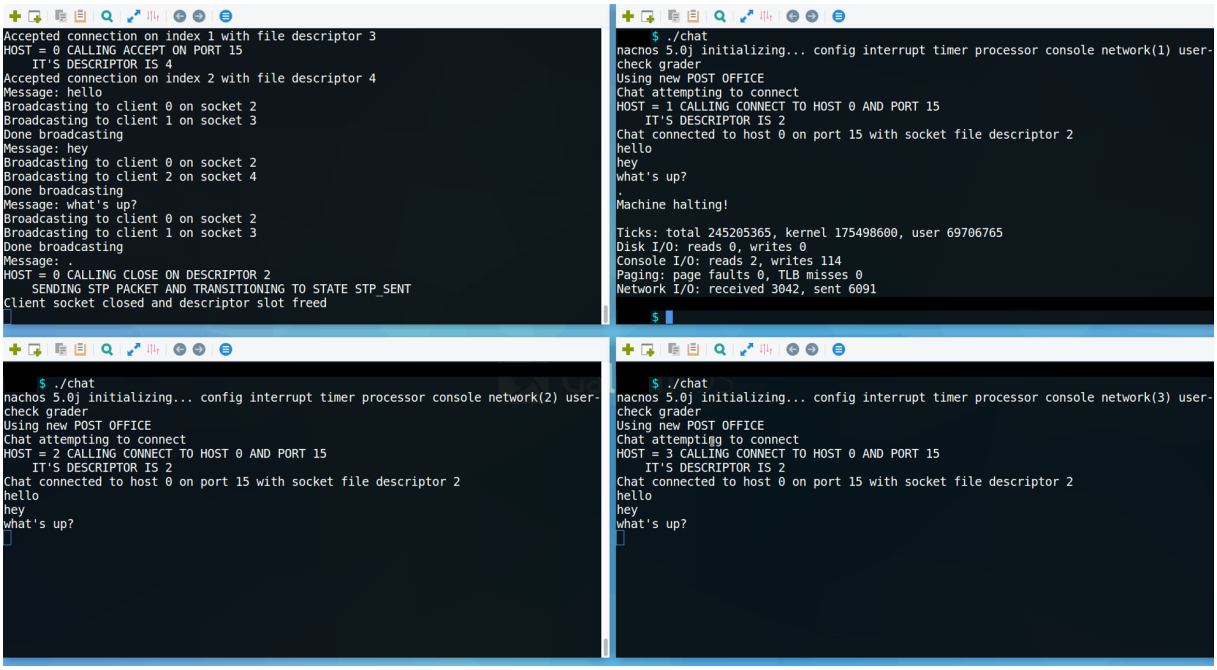
In this manner the primary objective of the chat application was to produce an N−way CHATSERVER/CHAT deliverable. Therefore the initial approach was to model functionality resembling that of socket interfaces.

It is observable from the pseudo−code that the CHATSERVER program attempts to forward all incoming connections and throws an exception error to catch should there be any malfunctions. Likewise, the functionality of the client node, is demonstrated in the pseudo−code excerpt below:

```
// Connect to host
    fd = connect(host, PORT_NUMBER);
    while(true)
    {
        // Buffers for user input and chat msg output
        curr_in_buffer = input_buffer;
        curr_out_buffer = output_buffer;

        // Check for msg from connection
        if(read(fd, curr_out_buffer, 1) > 0)
        {
            // Read in chat msg
            while(*curr_out_buffer != '\n') {
                if(read(fd, curr_out_buffer+1, 1) > 0)
                    curr_out_buffer++;
            }
            // Append a null char to end the string
            if(curr_out_buffer < output_buffer+BUFFER_SIZE) {
                *(curr_out_buffer+1) = 0;
            }
        }
        // Check for user input (in a non-blocking fashion)
        if(read(0, curr_in_buffer, 1) > 0) {
            // Read in all user input available
            while(*curr_in_buffer != '\n') {
                if(read(0, curr_in_buffer+1, 1) > 0)
                    curr_in_buffer++;
            }
            // Append a null char to end the string
            if(curr_in_buffer < input_buffer+BUFFER_SIZE) {
                *(curr_in_buffer+1) = 0;
            }
            // Send user msg to server
            curr_in_buffer = input_buffer;
            while(*curr_in_buffer != 0) {
                if(write(fd, curr_in_buffer, 1) > 0)
                    curr_in_buffer++;
            }
            // Disconnect msg received
            if(input_buffer[0] == '.' && input_buffer[1] == '\n') {
                break;
            }
        }
```

With this implementation a clients can connect to the server at any time and disconnect at any time while the server handle clients entering and leaving the chat room at any point during the conversation. Likewise, all users are able to view message contents in an identical format across all open connections. This feature is demonstrated in the following screenshot:



Design Decisions

* The chat client/server files are located within the ***test*** directory.

* Currently, 23/30 test cases pass. Redemption is established in the copious amounts of comments made in the submitted code.

* Explanations of all primary testing strategies are explained in the following section.

* Substantial time and resources were dedicated to debugging and quality control process.

* A ***byte***−length field header was added to the TCP MESSAGE constructor.

Testing/QA

Task I

The following code addresses test cases related to the aforementioned abstractions

```
public void selfTest()
{

    TransportFile a = new TransportFile(0,Machine.networkLink().getLinkAddress(),1,postOffice)
    TransportFile b = new TransportFile(1,Machine.networkLink().getLinkAddress(),0,postOffice)


    byte[] toSend = new byte[32]

    Lib.bytesFromInt(toSend,0,0x00010203)

    Lib.bytesFromInt(toSend,4,0x04050607)

    Lib.bytesFromInt(toSend,8,0x08090a0b)

    Lib.bytesFromInt(toSend,12,0x0c0d0e0f)

    Lib.bytesFromInt(toSend,16,0x10111213)

    Lib.bytesFromInt(toSend,20,0x14151617)

    Lib.bytesFromInt(toSend,24,0x18191a1b)

    Lib.bytesFromInt(toSend,28,0x1c1d1e1f)


    byte[] readData = new byte[toSend.length]
    int writeAmount = 0


    while (writeAmount < toSend.length)
        writeAmount += a.write(toSend, writeAmount, toSend.length-writeAmount)
        int readAmount = 0

    while (readAmount < toSend.length)
        readAmount += b.read(readData, readAmount, toSend.length-readAmount)
    for (int i = 0; i < toSend.length; ++i)
        print(readData[i])
        Lib.assertTrue(toSend[i] == readData[i])

}
```

The above function, selfTest, essentially creates a buffer of bytes. The creation of such a buffer enables the connection of two sockets across each other on a link on the same machine. The idea of invoking a testing strategy in this manner was to isolate the read and write syscalls and test their respective functionality with regard to reliability for the reading and writing of data. Therefore the invocation of internal calls to the respective TransportFile class functions, as aforementioned read and write, was able to isolate and determine their stochastic functionality

to be reliable within an error tolerance of only 0.01 through the retransmission of packets.

In order to model modern industrial programming paradigms, such as a minimally complete and verifiable example, reproductions of server errors were generated on local testing machines. This approached allowed for a more robust method of finding heisen−bugs and other difficult to contain, errors. Primarily, the CLOSE function which instantiated the TCP teardown.

Another testing strategy utilized for the verification of the CONNECT function revolved around the instantiation of two machines. One machine would invoke the CONNECT function and make sure that it got put to sleep by including a print statement at the end of the CONNECT function to see if it would wait for connection establishment. Then, once CONNECT was called, the ACCEPT function on the other machine would print to the console the statement included on the other machine. In this manner naively verifying the case that CONNECT would wait for connection establishment.

A strategy similar to this was utilized for testing reliability. The preliminary discrepancy in that case was to make sure that a SYN was transmitted multiple times until a SYN/ACK was received.

Task II

In order to test chat string length we inputted strings of increasing sizes from 1 to 300.

"***f***"
"***ffffffffff***"
"***ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff***"
etc...

In doing so we observed that the *printf* statement utilized from the given PRINTF.C file has a maximum buffer size of 256 bytes. Therefore, given our implementation, the maximum chat message size is 256 bytes.

The test for the server's client capacity was simple and involved opening eight chat clients communicating on a single server. Each client was tested and N-way communication was verified. This exceeded the project specification of at least 3 clients. Note: Our implemention supports up to 16 clients communicating on a single server.

Client argument testing was slightly unconventional. The project specification did not specify any command-line arguments for the chat client, however, the system call, ***connect(int host, int portNumber)***, required the inclusion of the host socket. Our implementation, therefore, accepts a single integer via the command-line and assigns it to be passed as the host socket when the connect system call is made. In order to test this, MACHINE.JAVA and USERKERNEL.JAVA were modified and functionality for command-line arguments was implemented. Within MACHINE.JAVA, a new method was created to return the process name and command-line arguments a string array:

```
public static String[] getShellProgramNameAndArgs() {
                if (shellProgramName == null)
                        shellProgramName = Config.getString("Kernel.shellProgram");

                Lib.assertTrue(shellProgramName != null);
                return shellProgramName.split(" ");
        }
```

Within UserKernel.java the run() method was modified as such:

```
    public void run() {
        super.run();
        UserProcess process = UserProcess.newUserProcess();
        String[] shellProgramAndArgs = Machine.getShellProgramNameAndArgs();
        String[] shellArgs = new String[]{};
        if(shellProgramAndArgs.length > 1) {
            shellArgs = Arrays.copyOfRange(shellProgramAndArgs, 1, shellProgramAndArgs.length);
        }
        Lib.assertTrue(process.execute(shellProgramAndArgs[0], shellArgs));
        KThread.currentThread().finish();
}
```

This allowed command-line arguments to be passed when running the chat client (this argument specifies a host socket number of 0):

JAVA NACHOS.MACHINE.MACHINE -X 'CHAT.COFF 0'

In doing so, we we're able to verify our implemention. Note: The above code is not included in the final submission code and was implemented temporarily, strictly for testing purposes.

Client disconnection was tested by passing the specified '.' message within the chat client. This was verified via printf statements from the server and clent. The clients were observed to send the disconnect message to the server and immediately exit. The server was observed to call the close() system call and assign -1 to the clients file descriptor, freeing the slot with in the array of clients to be used again should another client connect.

Team Member Work-Log

Avery Berchek
DevOps Engineer

- Designed and outlined the pseudo−code for TransportFile read/write and PostOffice manipulations.
- Implemented the algorithmic functionality for read/write.
- Primarily responsible for implementation of corner−case test cases.
- Conducted and performed code reviews.

Aleksandr Brodskiy
Project Manager

- Organized the *sprints* and weekly meetings in accordance with the *Agile/Scrum* project management methodology.
- Delegated tasks and assigned roles for the Engineering Team as well as conducted the code reviews.
- Outlined and participated in the design process associated with read/write.
- Created and formatted the Design Documentation.

- Managed all progress and operations of the Engineering Team in order to provide an efficient, robust, and optimal solution for a timely and submission.

David Cabral
Design Engineer

- Designed and outlined the pseudo−code for Accept & Connect.
- Implemented the algorithmic functionality for Accept & Connect.
- Implemented the algorithmic functionality for Close & TCP teardown.

Christopher DeSoto
Principal Engineer

- Designed and outlined the pseudo−code for the network chat application.
- Implemented the algorithmic functionality for the network chat application.

Adiam G-Egzabher
Systems Engineer

- Outlined and participated in the pseudo−code development associated with necessary adaptations to various file syscalls for TransportFile.
- Participated in the design process and Testing/QA phase for Task I.

Nanditha Embar
QA Engineer

- Outlined and participated in the pseudo−code development associated with necessary adaptations to various file syscalls for TransportFile.
- Participated in the design process and Testing/QA phase for Task II.

Christian Vernikoff
Software Engineer

- Designed and outlined the pseudo−code for theMailMessage inherited child class.
- Implemented the algorithmic functionality for the override MailMessage.java class to include TCP header fields.

Conclusion

In finishing this project algorithmic functionality of TCP (higher layer) syscalls was implemented along with a chat client/server application. Within this implementation, the consummation of a network stack as present in modern day operating systems was developed.