Protocol Implementation Lab Sessions.
Part 1. Testing the framework.
Laboratori d'Aplicacions i Serveis Telemàtics

Josep Cotrina, Marcel Fernandez, Jordi Forga, Juan Luis Gorricho, Francesc Oller

Contents

1 Introduction

The goal of this first set of lab exercises is to make you familiar with a framework for developing transport protocols. This framework will be used throughout the remaining lab sessions, therefore it is very important for you to fully understand its architecture and ways of operation.

1.1 Developing software within the framework

Throughout the lab sessions you will be asked to implement a sequence of transport protocols. Each implementation will add a new transport layer mechanism to its predecessor. As you will see, adding a new mechanism will generally mean a more complex implementation.

Since this increase in complexity is fairly quick, each lab session will introduce a new feature to a previously developed protocol. The introduction of new features will be done according to the following outline:

1. Interaction with network layer (IP) and port multiplexation, interaction with application layer and simulation startup.
2. Segmentation and transmission of data.
3. Simple flow control: stop and wait.
4. Flow control with better use of network capacity: sliding window.
5. Reliable communication with errors and/or losses in network layer.

1.2 Contents of distribution archive

The archive provided by the course professors contains all the components you will need for developing network transport protocols throughout the different lab sessions. These tools are:

- A `README` file.
- An archive `ast-protocols-1.3.0.jar` that contains the compiled framework and compiled solutions of the assignments.
- The framework javadoc documentation (directory `javadoc`).
- Code templates to guide you through the assignments (directories `src-prac*`).
- An archive `ast-protocols-1.3.0-src.zip` containing the source code of the framework.
- An example of logging configuration (file `logging.properties`).
- An example of test configuration (file `config.properties`).

2 Getting to know the framework

In order to understand the framework and address the implementation of different protocols, it is mandatory for you to get a deep understanding of the framework's different layers and various classes that have been defined at each layer.

To understand the simulation startup (comprising network, protocol and server and client creation) and its configuration you must see the classes:

- `ast.protocols.transportCO.test.Main`
- `ast.protocols.transportCO.test.SenderTask`
- `ast.protocols.transportCO.test.ReceiverTask`

To implement transport protocols you must known the following classes with detail:

- `ast.protocols.transportCO.TProtocol`

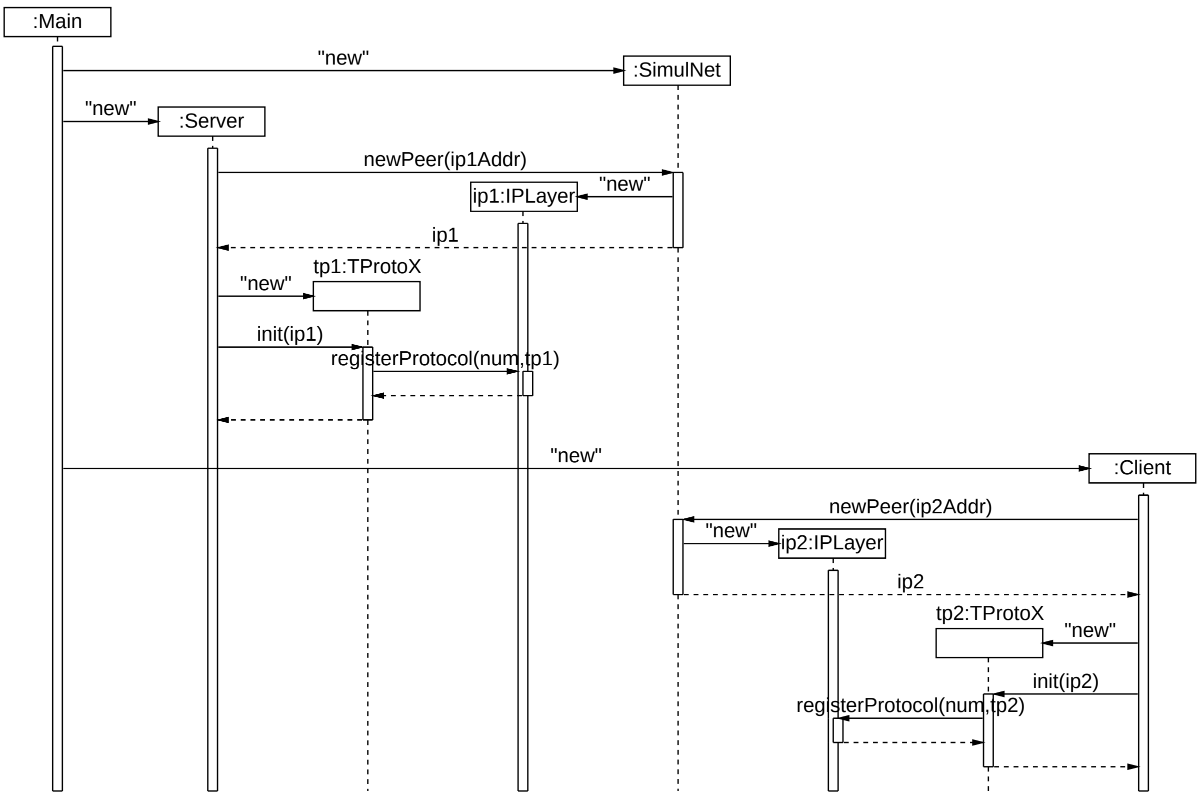

Figure 1: Network and host creation

- `ast.protocols.transportCO.TCBlock`
- `ast.protocols.tcp.TCPSegment`

The suffix `CO` conveys the meaning of Connection Oriented.
The nextwork (IP) layer, that interacts with the transport layer, is defined by the classes:

- `ast.protocols.ip.IPLayer`
- `ast.protocols.ip.IPPacket`

Figures are added to give a broader view of the framework.

2.1 Setting up the network

In figure 1 it is shown the creation of a simulated network with two nodes, with each node implementing IPLayer. A protocol is created in each node. Each protocol is an instance of some class `TProtoX` which must extend class `ast.protocols.transportCO.TProtocol`.

Sketch of code for host creation (see `ast.protocols.transportCO.test.Main` class for details):

```
SimulNet net;
// Create net
net = new SimulNet(config);

// For each host: create IP peer, create transport protocol and register in IP layer
```

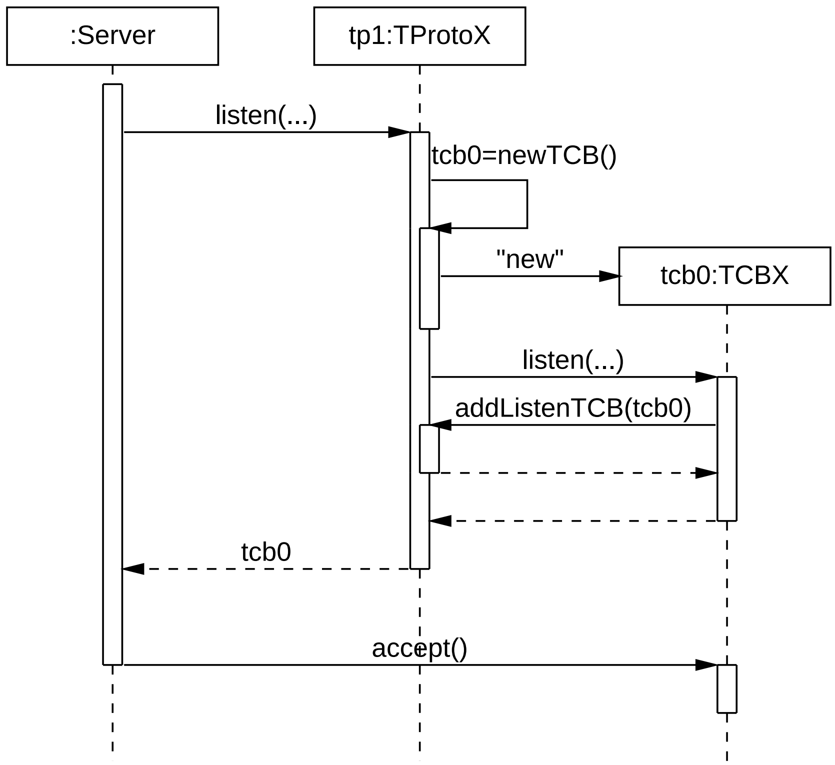

Figure 2: Connection passive open

```
IPLayer ip;      // One for each host
TProtocol tp;    // One for each host
ip = net.newPeer(ipAddr);
tp = new TProtoX();
tp.init(ip);
```

2.2 Connection establishment

The two endpoints of a connection will perform different roles. The endpoint waiting for a connection, usually called *passive endpoint*, will use the `listen()` and `accept()` methods. The endpoint initiating a connection will invoke the `connect()` method.

Each endpoint, also called *socket*, is represented by an instance of some class `TCBX` which must extend class `ast.protocols.transportCO.TCBlock`.

Figure 2 shows the passive open of an endpoint in a server node. The endpoint `tcb0` is created by the `listen()` method of the protocol. After the endpoint creation, the server calls TCBlock's `accept()` method to wait for incoming connection requests.

Sketch of code for server side (passive) open (see `ast.protocols.transportCO.test.ServerHost` class for details):

```
// Server side:
TCBlock tcb0;
tcb0 = tp1.listen(port, backlog);
```

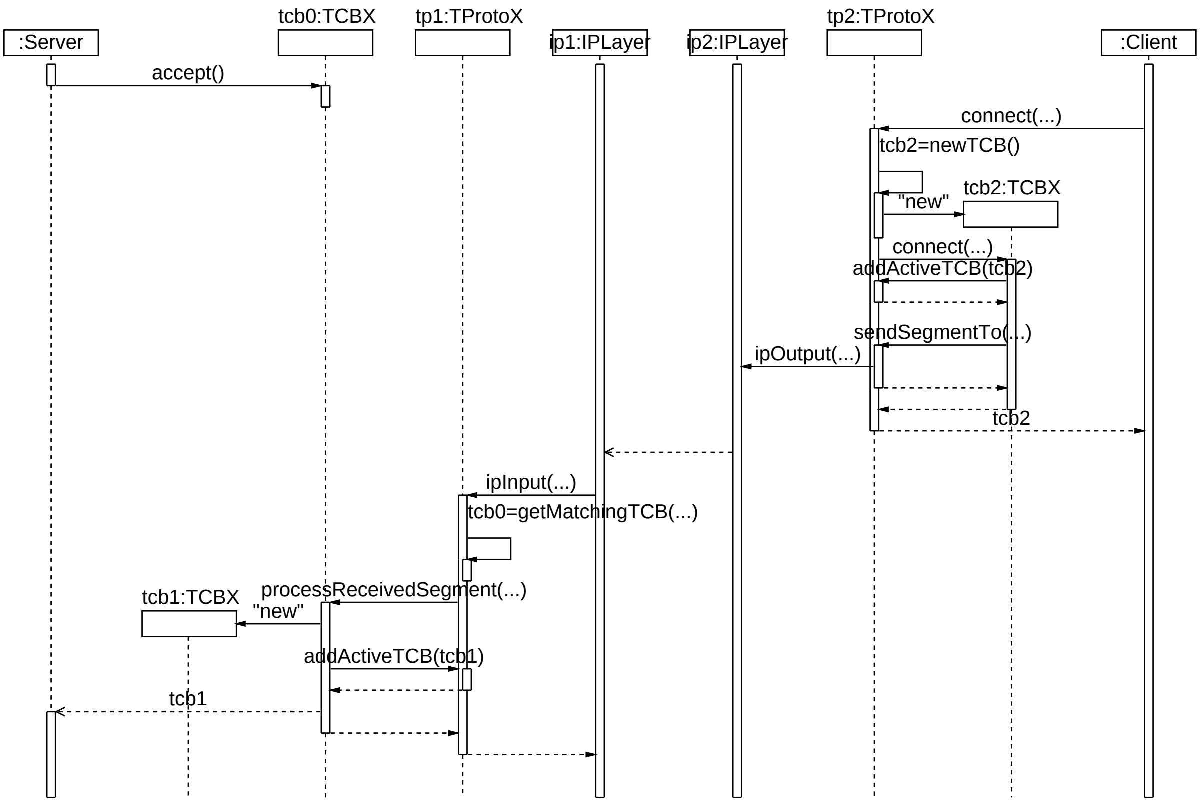

Figure 3: Connection establishment interactions

```
// For each incomming connection to wait:
TCBlock tcb1;
tcb1 = tcb0.accept();
```

Figure 3 shows how a connection is established: client node calls the protocol's `connect()` method, creating an active TCBlock `tcb2`. Client side TCBlock interacts with `tcb0` (server's passive TCBlock), creating a new active TCBlock `tcb1` in server side.

Sketch of code for client side (active) open (see `ast.protocols.transportCO.test.ClientHost` class for details):

```
// Client side:
TCBlock tcb2;
tcb2 = tp2.connect(remoteIP, remotePort);
```

2.3 Data transmission

After connection establishment, client TCBlock `tcb2` and newly created server TCBlock `tcb1` are connected and ready for data transmission. Figure 4 shows a sequence diagram that depicts how data is sent and received.

See `ast.protocols.transportCO.test.SenderTask` and `ast.protocols.transportCO.test.ReceiverTask` classes for details.

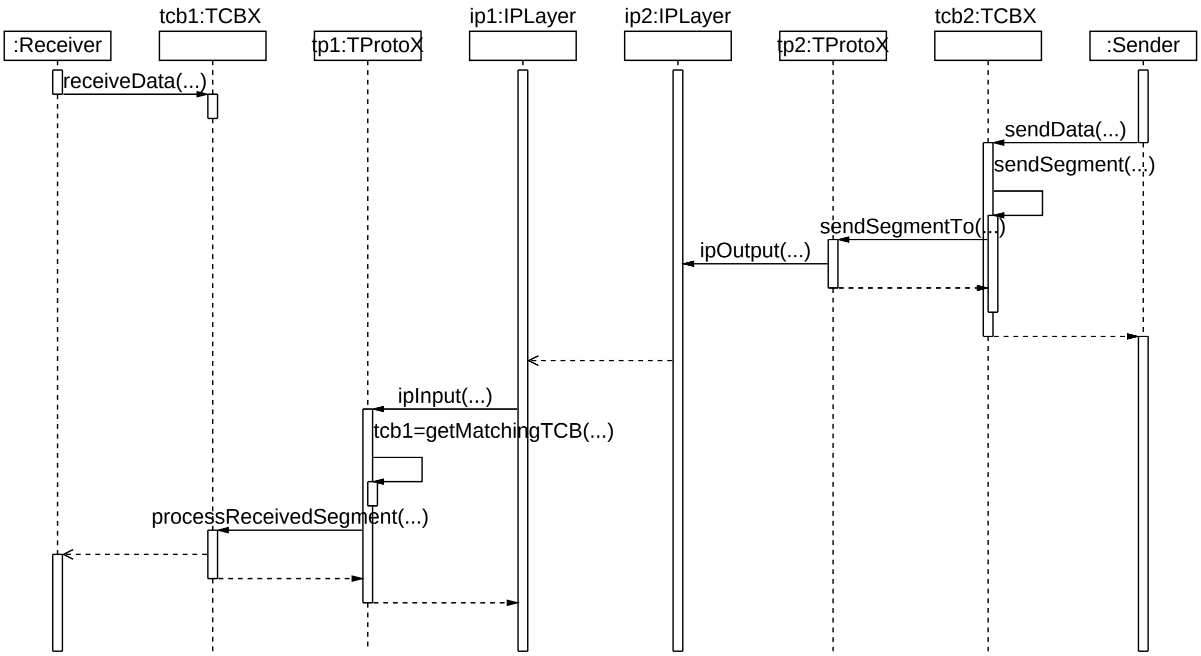

Figure 4: Data receiving/sending interactions

3 Connection Oriented Transport Protocols

Transport protocols lie on top of a network layer. It is needless to say that the most used network layer protocol is the Internet Protocol (IP). In the framework all transport protocols will implement the `ast.protocols.ip.Protocol` interface.

In the lab sessions we will focus on connection oriented protocols along the guidelines of the Transmission Control Protocol (TCP), as opposed to connectionless protocols such as the User Datagram Protocol (UDP). This connection oriented protocol paradigm is abstracted in the class `ast.protocols.transportCO.TProtocol` that implements the `Protocol` interface. A common feature of all network transports protocols is port multiplexing. A port identifies, within a host, a process accessing the network. So, a port helps to deliver an incoming segment to the appropriate process. This common mechanism is also implemented in the `TProtocol` class.

One immediate consequence of using connection oriented protocols, is that both endpoints will have to perform a *handshake* before any data is transmitted. The fact that there is a handshake means that the connection will go through different states. The most elementary states are, for instance, connection not established and connection established. In the different lab sessions we will see that actual protocols go through several more states. The state of a connection will be maintained by an object of a class subclassing the `ast.protocols.transportCO.TCBlock` class.

3.1 The `TProtocol` class

In this section we present several noteworthy elements of the `TProtocol` class. This abstract class will be the base class of all classes implementing a connection oriented protocol.

To begin with, we see that there are two lists of `TCBlock` objects. This is because, it is necessary to differentiate between `TCBlock` objects that will wait for incoming connection requests (`listenTCBs`) and `TCBlock` objects that are already connected to a remote endpoint (`activeTCBs`).

```
    protected ArrayList<TCBlock> listenTCBs;     /* All unbound TCBs (in state LISTEN) */
```

```
    protected ArrayList<TCBlock> activeTCBs;    /* All bound TCBs (in active states) */
```

Moreover, there are two methods for both listening to incoming connections and initiating connections to a remote endpoint: `listen(int port, int backlog)` and `connect(int addr, int port)`. The `backlog` parameter indicates the size of the queue of incoming connection requests that are not yet served.

This class also contains methods to interact with the network layer. The `ipInput` method is called from the network `IPLayer` whenever there is an IP packet to be delivered. The `sendSegmentTo` method is invoked in order to deliver a transport segment to the network `IPLayer`.

3.2 The `TCBlock` class

As we said above, objects of classes extending the `TCBlock` class maintain the state of a connection. The `TCBlock` class contains methods for transmitting and receiving data (`sendData`, `receiveData`) and processing incoming transport segments (`processReceivedSegment`).

Moreover, its subclasses will implements methods for establishing a connection either actively (`connect`) or passively (`listen`, `accept`). To release a connection both ends will invoke the `close()` method.

4 A First Transport Protocol Implementation

4.1 Unreliable Data Transmission

In the exercises of this part we address the following topics:

- Establishing and releasing transport layer connections.
- Interaction with network layer (IP) and port multiplexation.
- Segmentation, checksum computation and transmission of data.

In this first approach, we do not worry about any kind of reliability such as flow control or error control, and solely focus in understanding the basic operation. You will also acquire a deeper understanding of the framework for developing protocols that we are using. The flexible architecture of the framework minimizes the changes to be made in order to test a new transport protocol. Note that you only need to extend the `ast.protocols.transportCO.TProtocol` class, the `ast.protocols.transportCO.TCBlock` class and make the appropriate changes in the Properties file.

4.2 Description of the protocol

The diagram in Figure 5 shows our protocol for the establishing and releasing of a connection under the assumption of a totally reliable IP layer (no packet loss and no errors in packets). Throughout the exercises you will see that even under this assumption the problem is already non trivial.

4.3 Exercises

4.3.1 The `TProtocol` class implementation

Complete the following class. Observe that although the `ipInput` and `sendSegmentTo` methods are already implemented in the `TProtocol` class, you are asked to do your own implementation.

```
                             +---------+
                             |  CLOSED |-------------
                             +---------+             \
                          LISTEN  |                   \
                          ------  |                    | CONNECT
                                  V                    | -------
                             +---------+               | snd SYN
                             |  LISTEN |               |
                             +---------+          +----------+
                                  |               | SYN_SENT |
                                  |               +----------+
                        rcv SYN   |                    |
                        -------   |                    | rcv SYN
                        snd SYN   |                    | -------
                                  |                    |
                                  V                   /
                             +---------+             /
                             |  ESTAB  |<------------
                             +---------+
                      CLOSE    |     |    rcv FIN
                     -------   |     |    -------
+---------+          snd FIN  /       \                   +---------+
|  FIN    |<-----------------           ------------------>|  CLOSE  |
|  WAIT   |------------------           -------------------|  WAIT   |
+---------+          rcv FIN  \       /   CLOSE           +---------+
                     -------   |      |  -------
                               |      |  snd FIN
                               V      V
                             +----------+
                             |  CLOSED  |
                             +----------+
```

Figure 5: Connection State Diagram

```java
public class TProtocol_Unreliable1 extends TProtocol {
    @Override
    protected TCBlock newTCB(int port) {
        // Create an instance of TCBlock which implements desired protocol
        return new TCBlock_Unreliable1(this, port);
    }

    // We expect PROTO_NUM is not used by other registered protocols
    public static final int PROTO_NUM = 201;

    @Override
    public int getProtoNum() { return PROTO_NUM; }

    @Override
    public void ipInput(IPPacket packet) {
        // To be completed by the student:
        ...
        // Compute and verify TCP checksum
        ...
        // Search matching TCB
        ...
        // Process received segment
        ...
    }

    @Override
    protected void sendSegmentTo(TCPSegment segment, int localAddr, int remoteAddr) {
        // compute and set checksum
            segment.setChecksum(0);
            segment.computeChecksum(localAddr, remoteAddr, getProtoNum());
        // To be completed by the student:
        ...
    }
}
```

4.3.2 The `TCBlock` class implementation

Complete the following class that implements an unreliable protocol. Do not worry about segment size, flow control or error control. Use the `ast.util.ByteQueue` class to store the received data.

It is very important that you take into account mutual exclusion and condition waiting issues.

```java
class TCBlock_Unreliable1 extends TCBlock {
    protected int state;
    protected CircularQueue<TCBlock_Unreliable1> acceptQueue;
    protected ByteQueue rcvQueue;

    // States of FSM (see description in TProtocol_Unreliable1):
    protected final static int CLOSED = 0,
                               LISTEN = 1,
                               SYN_SENT = 2,
                               ESTABLISHED = 3,
                               FIN_WAIT = 4,
                               CLOSE_WAIT = 5;

    TCBlock_Unreliable1(TProtocol proto, int port) {
        super(proto, port);
    }

    // Passive open
    @Override
    public void listen(int backlog) throws IOException {
      lk.lock();
```

```java
    try {
      log.debug("%1$s->listen()", this);
      if (state != CLOSED) {
          throw new IOException("connection already exists");
      }
      acceptQueue = new CircularQueue<TCBlock_Unreliable1>(backlog);
      state = LISTEN;
      addListenTCB(this);
      logDebugState();
    } finally {
      lk.unlock();
    }
  }

  @Override
  public TCBlock accept() throws IOException {
    lk.lock();
    try {
      log.debug("%1$s->accept()", this);
      if (state != LISTEN) {
          throw new IOException("connection is not LISTEN");
      }
      try {
          // wait some client to connect to me
          while (acceptQueue.empty()) appCV.await();
      } catch (InterruptedException ie) {
          log.warn("Interrupted Exception at accept()");
      }
      TCBlock_Unreliable1 r = acceptQueue.get();
      log.debug("%1$s->accepted", this);
      return r;
    } finally {
      lk.unlock();
    }
  }

  // Active open
  @Override
  public void connect(int addr, int port) throws IOException {
    lk.lock();
    try {
      log.debug("%1$s->connect()", this);
      if (state != CLOSED) {
          throw new IOException("connection already exists");
      }
      initActive(addr, port & 0xffff, SYN_SENT);
      TCPSegment sseg = new TCPSegment();
      sseg.setSourcePort(localPort);
      sseg.setDestinationPort(remotePort);
      sseg.setFlags(TCPSegment.SYN);
      sendSegment(sseg);
      logDebugState();
      try {
          while (state != ESTABLISHED) appCV.await(); // wait SYN is received
      } catch (InterruptedException ie) {
          log.warn("Interrupted Exception in connect()");
      }
      log.debug("%1$s->connected", this);
    } finally {
      lk.unlock();
    }
  }

  // initialize for new connection
```

```java
protected void initActive(int remAddr, int remPort, int st) {
    remoteAddr = remAddr;
    remotePort = remPort;
    state = st;
    rcvQueue = new ByteQueue(1000);
    addActiveTCB(this);
}

@Override
public void close() throws IOException {
    lk.lock();
    try {
      log.debug("%1$s->close()", this);
      switch (state) {
        case LISTEN: {
            state = CLOSED;
            removeListenTCB(this);
            logDebugState();
            break;
        }
        case ESTABLISHED:
        case CLOSE_WAIT: {
            TCPSegment sseg = new TCPSegment();
            sseg.setSourcePort(localPort);
            sseg.setDestinationPort(remotePort);
            sseg.setFlags(TCPSegment.FIN);
            sendSegment(sseg);
            if (state == ESTABLISHED) {
                state = FIN_WAIT;
            } else {
                state = CLOSED;
                removeActiveTCB(this);
            }
            logDebugState();
            break;
        }
        default:
            throw new IOException("connection does not exist");
      }
    } finally {
        lk.unlock();
    }
}

@Override
public void sendData(byte[] data, int data_off, int data_len) throws IOException {
    // To be completed by the student:
    ...
}

@Override
protected void processReceivedSegment(int sourceAddr, TCPSegment rseg) {
    lk.lock();
    try{
        switch (state) {
        case LISTEN: {
            if (rseg.isSyn()) {
                if (acceptQueue.full()) {
                    log.warn(
                      "%1$s->processReceivedSegment: Backlog queue is full. SYN IS LOST !!!",
                      this);
                    return;
                }
                // create TCB for new connection
```

```java
                    TCBlock_Unreliable1 ntcb = (TCBlock_Unreliable1) newTCB();
                    ntcb.initActive(sourceAddr, rseg.getSourcePort(), ESTABLISHED);
                    // To be completed by the student:
                    ...
                    // prepare the created connection for accept
                    ...
                    // send SYN segment for new connection
                    ...
                    ntcb.logDebugState();
                }
                break;
            }
            case SYN_SENT: {
                if (rseg.isSyn()) {
                    // To be completed by the student:
                    ...
                    // Change state and wake up connect() thread
                    ...
                    logDebugState();
                }
                break;
            }
            case ESTABLISHED:
            case FIN_WAIT:
            case CLOSE_WAIT: {
                // Check SYN bit
                if (rseg.isSyn()) {
                    // A SYN is bad if it arrives on a connection in an active state. Ignore it
                    return;
                }
                // Process segment text
                if (rseg.getDataLength() > 0) {
                    if (state == ESTABLISHED || state == FIN_WAIT) {
                        // To be completed by the student:
                        ...
                        logDebugState();
                    } else {
                        // This should not occur, since a FIN has been received from the
                        // remote side. Ignore the segment text.
                    }
                }
                // Check FIN bit
                if (rseg.isFin()) {
                    if (state == ESTABLISHED) {
                        state = CLOSE_WAIT;
                    } else if (state == FIN_WAIT) {
                        state = CLOSED;
                        removeActiveTCB(this);
                    }
                    appCV.signalAll(); // wake up receiveData() thread
                    logDebugState();
                }
                break;
            }
            default:
                // Segment is ignored
            }
        } finally {
            lk.unlock();
        }
    }

    @Override
    public int receiveData(byte[] buf, int off, int len) throws IOException {
```

```
        lk.lock();
        try {
            if (state == ESTABLISHED || state == FIN_WAIT || state == CLOSE_WAIT) {
            } else {
                throw new IOException("connection does not exist");
            }
            // wait until receive buffer is not empty or FIN is received
            while (rcvQueue.empty() && !(state == CLOSE_WAIT || state == CLOSED)) {
                try { appCV.await(); } catch (InterruptedException e) {}
            }
            assert !rcvQueue.empty() || state == CLOSE_WAIT || state == CLOSED;
            if (rcvQueue.empty()) {
                // remote endpoint is closed (state is CLOSE_WAIT or CLOSED)
                return -1;
            } else {
                int r = rcvQueue.get(buf, off, len);
                log.debug("%1$s->receivedData: len=%2$d" , this , r);
                return r;
            }
        } finally {
            lk.unlock();
        }
    }
```

4.4 Testing

Test the classes above. Pay special attention to the connection establishment and releasement steps. Change the appropriate values in the `config.properties` file so data is almost surely lost. Change these values again to avoid the loss of data.

Note: A compiled solution is available in the class `ast.protocols.transportCO.impl.TProtocolUnreliable` with protocol number equal to 201. You can run tests with your implementation in one end against our compiled solution in the other end.

5 Maximum segment size

Now we focus in the maximum size allowed for transport segments. Upper bounding the size of transport segments improves efficiency because it prevents network packet fragmentation. Implement the following class, that extends the previous one, in order to take into account this issue.

```
public class TProtocol_Unreliable2 extends TProtocol_Unreliable1 {
    @Override
    protected TCBlock newTCB(int port) {
        return new TCBlock_Unreliable2(this, port);
    }
}

class TCBlock_Unreliable2 extends TCBlock_Unreliable1 {
    protected int sndMSS;    // Send maximum segment size

    TCBlock_Unreliable2(TProtocol proto, int port) {
        super(proto, port);
    }

    // initialize for new connection
    @Override
    protected void initActive(int remAddr, int remPort, int st) {
        // To be completed by the student:
        ...
    }
```

```
    @Override
    public void sendData(byte[] data, int data_off, int data_len) throws IOException {
        // To be completed by the student:
        ...
    }
}
```

5.1 Testing

Test the classes above. Check that some data might be lost. Change both the size of transport segments and the `rcvQueue` and comment on the outcome.

Note: A compiled solution is available in the class `ast.protocols.transportCO.impl.TProtocolUnreliable` with protocol number equal to 201. You can run tests with your implementation in one end against our compiled solution in the other end.

A Running the test `Main` class

The framework offers one entry point (static procedure `main`) in the `ast.protocols.transportCO.test.Main` class and is used to test connection oriented protocols (implementations of `ast.protocols.transportCO.TProtocol` class).

This entry point expect some specific virtual machine options and arguments. When you execute the virtual machine you must provide, among others, the following options:

- the classpath (option `-cp ...`) including the `ast-protocols-1.3.0.jar` archive.
- the system property `ast.simplelog.configFile` with value set to the name of the logging configuration file (for example, option `-Dast.simplelog.configFile=logging.properties`).

Moreover, you should give an argument to the `main` procedure that corresponds to the configuration file of some framework components: the simulated IP network and the different test tasks to be executed.

For example, the next command line

```
java -cp ast-protocols-1.3.0.jar -Dast.simplelog.configFile=logging.properties \
     ast.protocols.transportCO.test.Main config.properties
```

executes the test for connection oriented protocols, reading the logging configuration from file `logging.properties` and reading the IP net and test configurations from file `config.properties`.

A.1 Executing from `NetBeans`

To create a project you can proceed as follows:

1. Download the distribution archive and uncompress it.
2. Open *NetBeans* and select *File/New Project*.
   (a) Choose *Java Project with Existing Sources*.
   (b) Set the *Project Name* (for example, `ast-protocols`).
   (c) Select the downloaded and uncompressed directory as the *Project Folder*.
   (d) Select the subdirectory `src-prac1` and add it to *Source Package Folders*.
   (e) *Finish*.
3. Select *Project Properties*.
   (a) Go to *Libraries*.
      i. Click *Add Jar/Folder* and select the `ast-protocols-1.3.0.jar` archive.
      ii. Click *Edit* and attach the directory `javadoc` and the `ast-protocols-1.3.0-src.zip` archive as *Javadoc* and *Sources* references.
   (b) Go to *Run*.
      i. Set *Main Class* to `ast.protocols.transportCO.test.Main`.
      ii. Set *Arguments* to the name of the properties file `config.properties`. Note that this file should be in the *Project Folder*
      iii. Set *VM Options* to `-Dast.simplelog.configFile=logging.properties`.
      iv. Optionally, you can create new configurations for each assignment.
      v. *OK*.

Edit the `.properties` files to tune your executions.