TTM 4100 Communications – Services and Networks

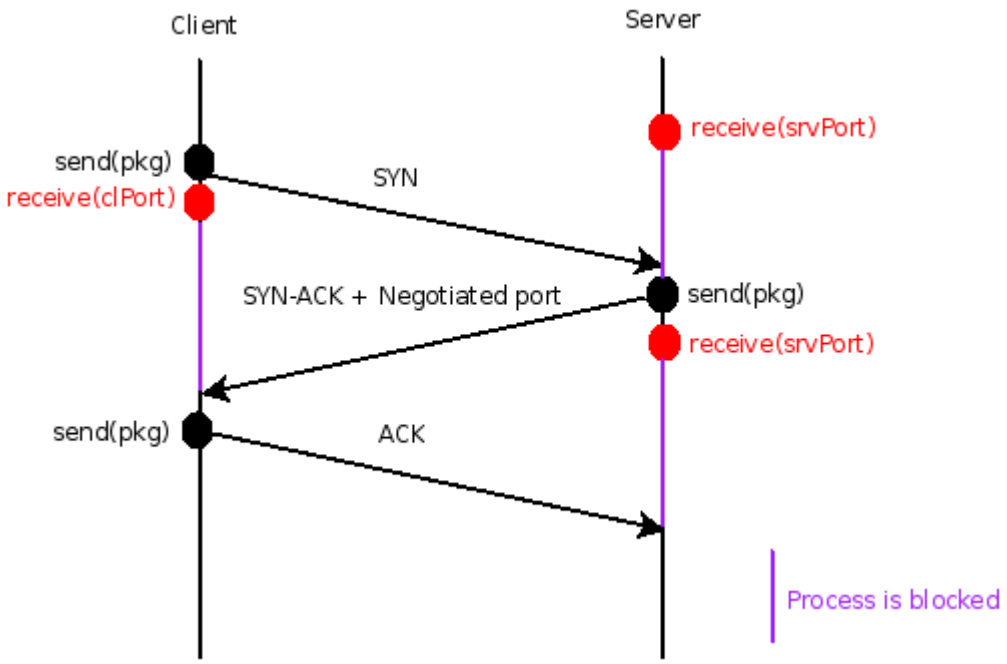
KT1

Tomáš Babiak
Ángela Ayet
María Fernández
Sveinung Kvilhaugsvik
David Rozas

1. Message sequence charts

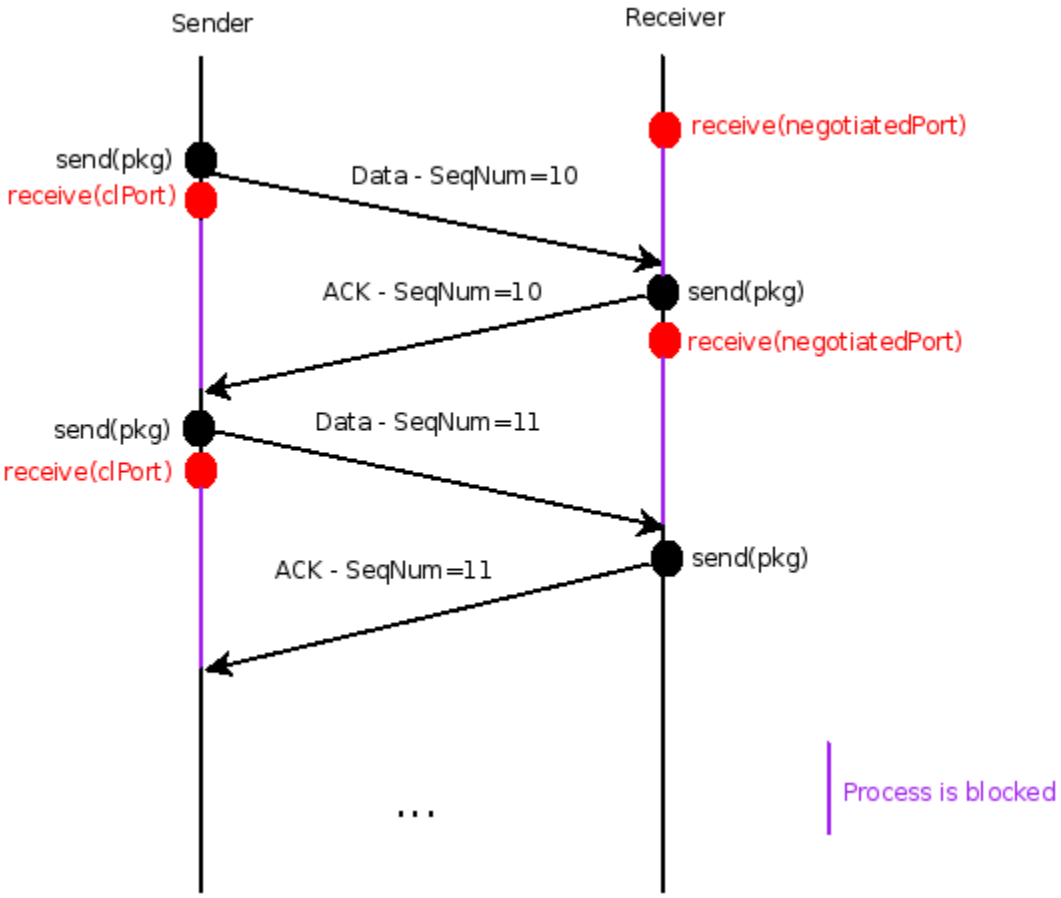
Connection

The connection process concerns to the methods accept() and connect() of A1, and it is implemented with a three-way handshake using the methods send() and receive() of A2.
The flags are enabled through the set methods provided by the class KtnDatagram.
Once the connection is established, the client will connect with the server using another port which has been negotiated during this process.



Sending and receiving

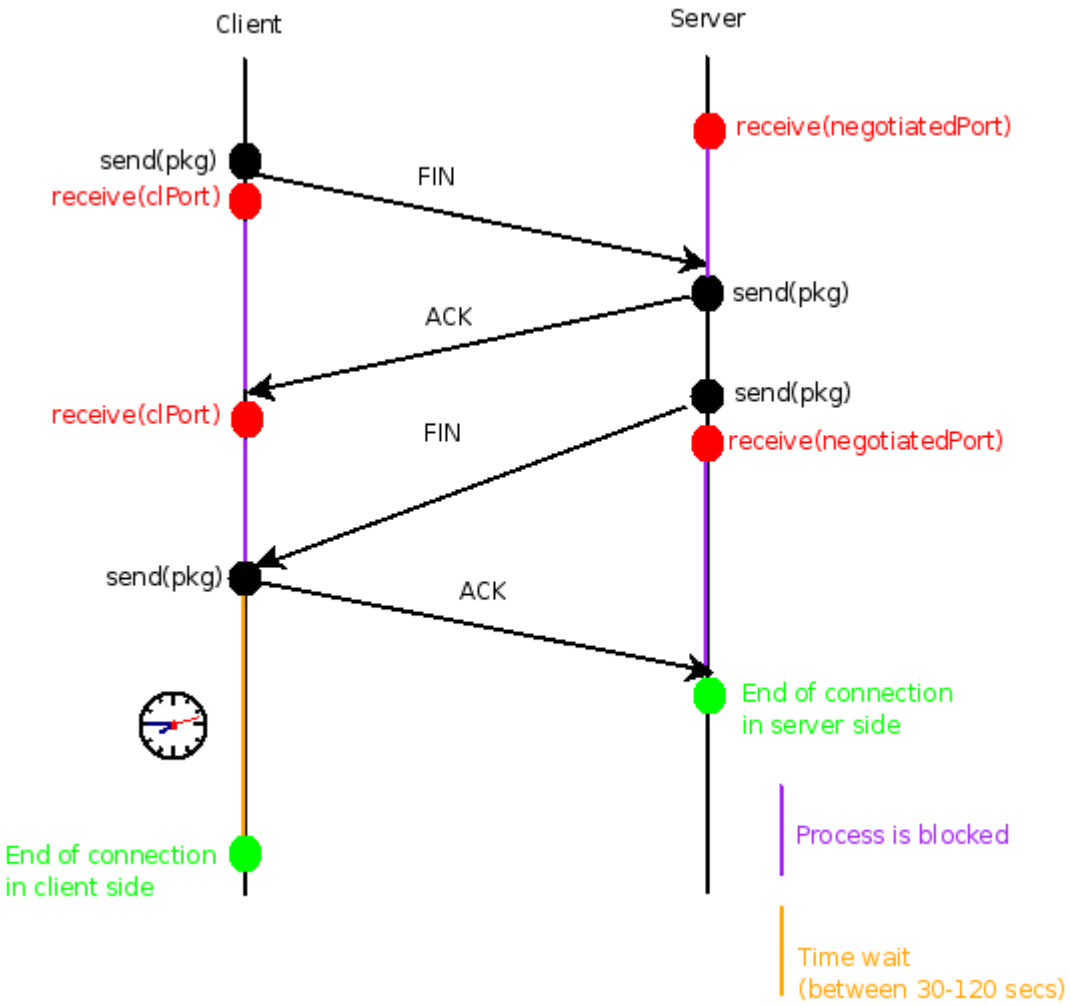
The methods send() and receive() of A1 are implemented by send() and receive() of A2 in an scenario without errors as is shown in this chart (errors handling is explained in subsequent sections).
As well as the flags, the access to the sequence number is provided by KtnDatagram methods.



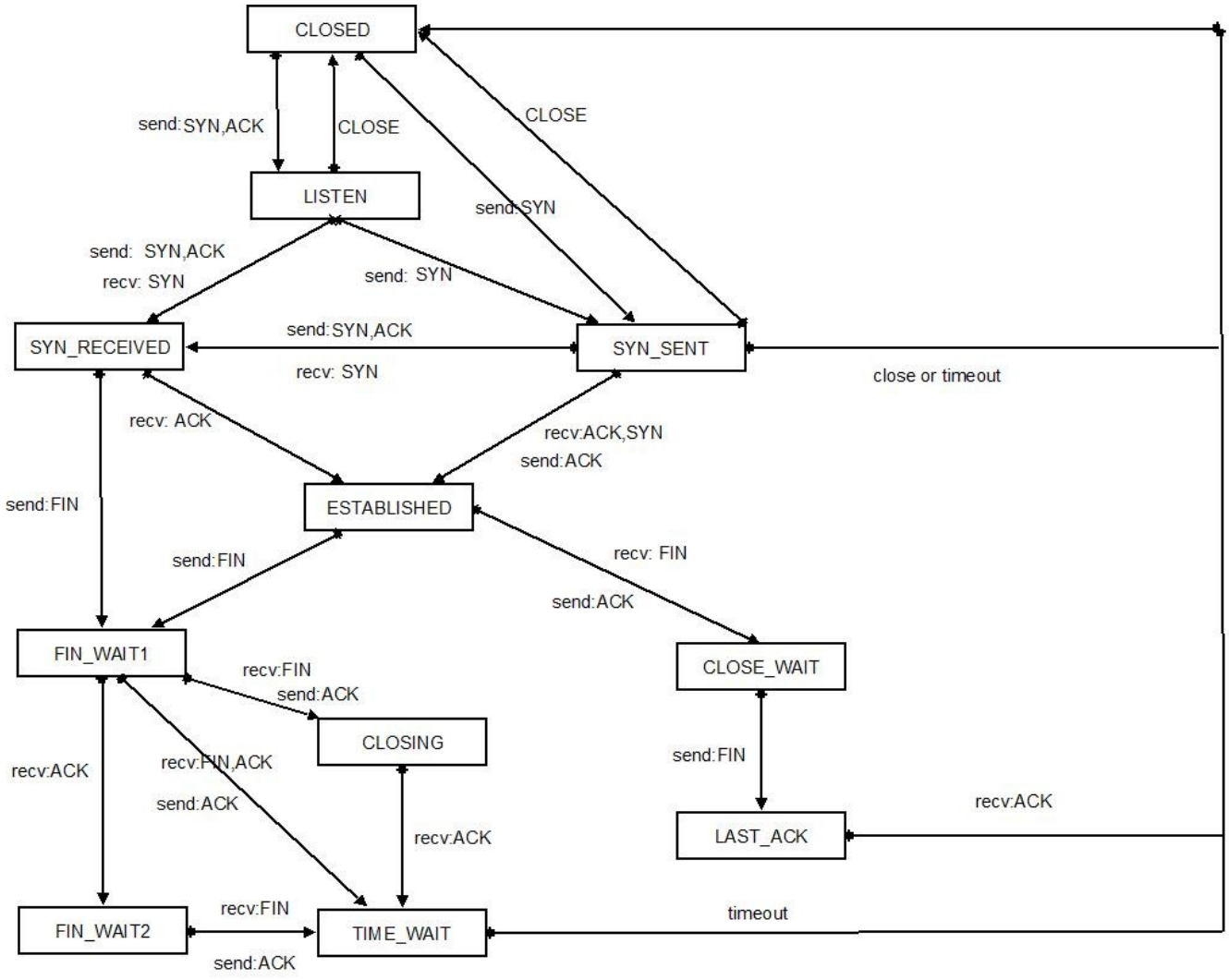
This example suppose the server as receiver to illustrate the use of the negotiated port in next stages, but the server could also be the sender.

Disconnection

As in the termination of the connection process in TCP, we are going to implement a four-way handshake, including a TIME_WAIT state where the client retransmits the final ACK in case the ACK is lost. This scenario refers to the method close() in A1.



2. State diagrams for A1



3. Description of the design and realization of A1

Regarding to layer A1, the following diagram shows how three of the main classes belonging to A1 are associated, concretely how AbstractConnection, Connection and ConnectionImpl classes are related:

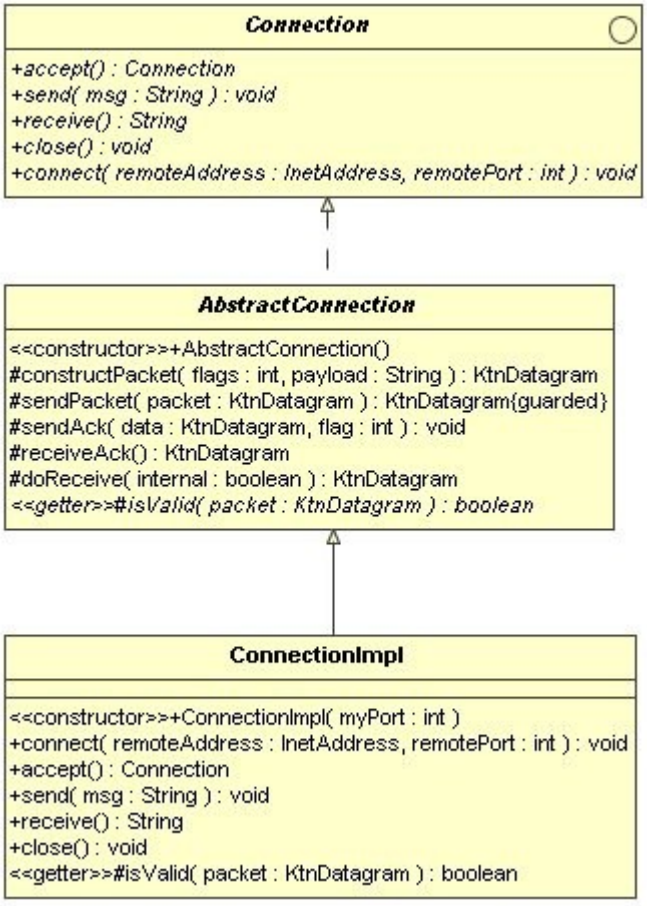


*Illustration 1: Associations between *Connection* classes*

Our main task resides in implementing methods from ConnectionImpl using methods provided by A2 connectionless communication service. As we can see in Illustration 2, the messages the Application, A1 and A2 can exchange are clearly defined.

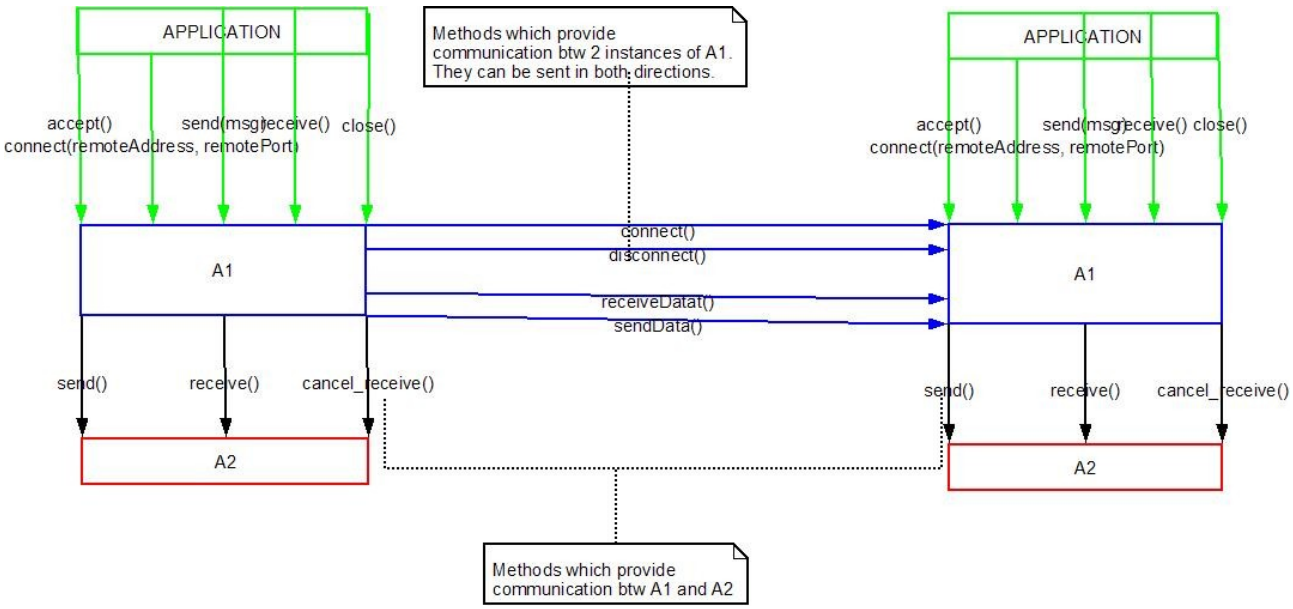


Illustration 2: Communication between application and service A1 and service A2.

Since interfaces between services are distinctly precised, that is to say, methods which communicate different sevices are already specified, our design must concentrate in defininig how methods from A1 use methods from service A2.

Let's exemplify this with an example: Imagine a client application wishes to establish a connection with a server application. The server application will create an instance of a class ConnectionImpl and repeatedly call accept().

How will A1.accept() method be translated in terms of A2 service?

```
     KtnDatagram datagram = A2.receive(port);
         int flag=datagram.getFlags();
     if(flag==KtnDatagram.SYN){
          KtnDatagram d =A1.constructPacket(KtnDatagram.SYN, null);
          d.setDest_port(newNegotiatedPort);
          // The new port must be done here since it's the only
     // message the server sends to client  in 3-way handshake
          A2.send();
     }
     datagram =  A2.receive(port);
     int flag=datagram.getFlags();
     if(flag==KtnDatagram.SYN_ACK){
          //create new connection at new negotiated port
          // ClSocket socket= new Clsocket();
}
```

4. Error handling

This table shows different types of error within communication and the proper response:

Type of error	**Type of package**	Response
Packege lost	Data package	Sender retransmits the package after the timeout expires.
	ACK package	Sender retransmits the package after the timeout expires. Receiver throws away the duplicate package and resents an appropriate acknowledgment.
Package delayed	Data package	Sender retransmits the package after the timeout expires. Receiver throws away the duplicate package and resents an appropriate acknowledgment. Sender do not process the duplicate acknowledgment.
	ACK package	
Package has errors	Data package	Receiver throws the package away and does nothing. Sender thinks the package was lost and retransmits it.
	ACK package	Sender can not be sure if it is the real acknowledgment but corrupted or if it is a ghost package. Therefore he retransmits the data package (same situations as if the acknowledgment has been lost).
Ghost package	Data package	Receiver throws the package away and communication continues the way if the package has been lost.

	ACK package	Sender throws the acknowledgment away and retransmits data (same situations as if the acknowledgment has been lost).

Now the proper standard behavior of sender and receiver can be derived.
Sender will send the data package and wait for acknowledgment or until the timeout expires. If that happens he retransmits the data. If the sender receives the same acknowledgment twice, he just ignores it.
Receiver receive the package. He checks the header if it contains proper sender and receiver address and port. Then whether the sequence number is correct (whether it continues) and at the end the data checksum. If the package is corrupted he just throws it way. If the sequence number is same as the previous one (thus it is a duplicate), he resents the appropriate acknowledgment and throws the package away (already has an original). If everything is correct, he just sends an acknowledgment.

Described errors relates only to communication (sending and receiving data) after the connection was established. Any other errors, such that the connection was broken (no successful delivery even after several retransmits of the data) or any lost message during the establishing of the connection will cause that the proper exception will be thrown and application notified about this problem.

5. Test plan

The plan for testing is to use JUnit. This allows the testing to be automated and therefore cheaper. It also makes it possible to have several small tests so we know exactly what is wrong. Code to configure the desired errors for the testing will be used. (It can be done even if we don't modify the code we are not supposed to modify) As long as the tests are properly split up there should not be to mutch extra code needed: To test a combination of a test configuration and a generic test only two lines of Java are needed: one to call the configuration and one to the generic test. The generic tests will also share code since some tests simply are an earlier stage in another. All combinations of Configurations x Generic test should be tried.

Generic tests: A first test is of course to see if establishing a connection works. The following tests should try to actually send data, one in each direction. One should try to connect and disconnect several times (5-10). The next should try to connect more connections to one server, then one should try more connection from one client to server. A test for a longer lasting connection that will connect and send several messages should follow. The tests that transfer data can easy be implemented by using assertEquals() in JUnit. The rest will have to trust other criteria.

Test configurations: The requirements wants us to test when there are no errors at all and the set {Package lost, Package delayed, Package has errors, Ghost package} x { 10% errors, 50% errors}. It also asked for tests that had some and all errors in various occurrences. One test where all errors have a probability of 80% should be added to have a worst case scenario in order to see if our implementation can take a beating. In order to test how it will run on a mobile phone behind a NAT (like most phones are) another scenario is: package loss 30%, package delayed 60%, error payload 40% and error header 28%.