

TTM 4100 Communications – Services and Networks

KTN2

Tomáš Babiak
Ángela Ayet
María Fernández
Sveinung Kvilhaugsvik
David Rozas

Introduction - about the implementation

The implementation of the project has followed the design almost to the letter, so not important changes in the documentation are needed.

Changes in the documentation are only in section 2 (States diagram), where the same diagram has been followed but due to the way in which some of the functions in `AbstractConnection` are implemented, and to the nature of our protocol itself we have decided to split the old diagram in two and eliminate unnecessary transitions and states.

An explanation about the changes and a new diagram can be found in section 2.

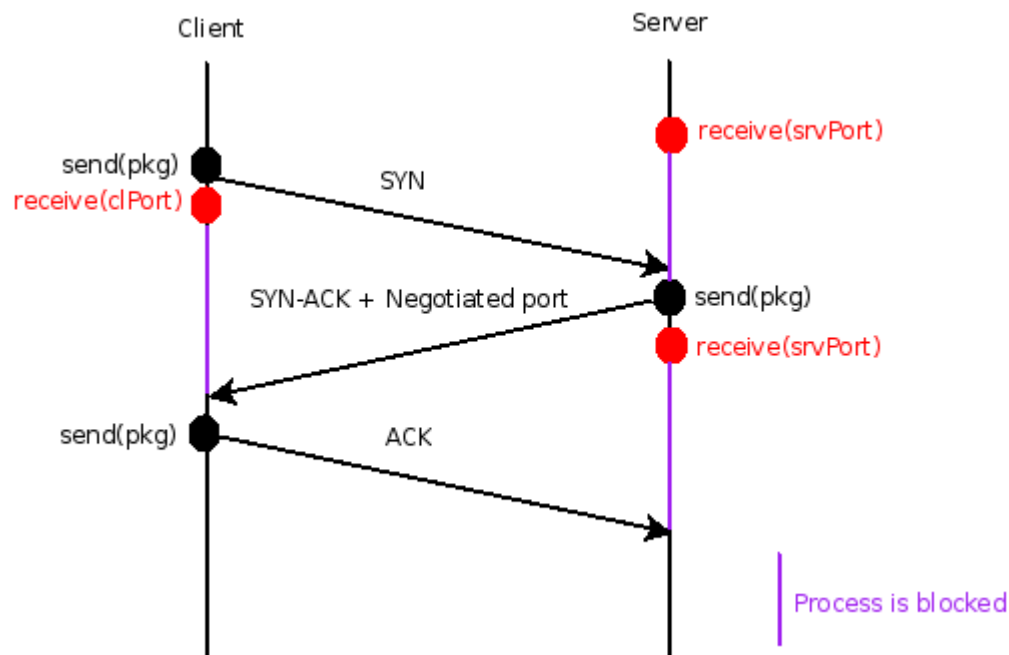
1. Message sequence charts

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`.

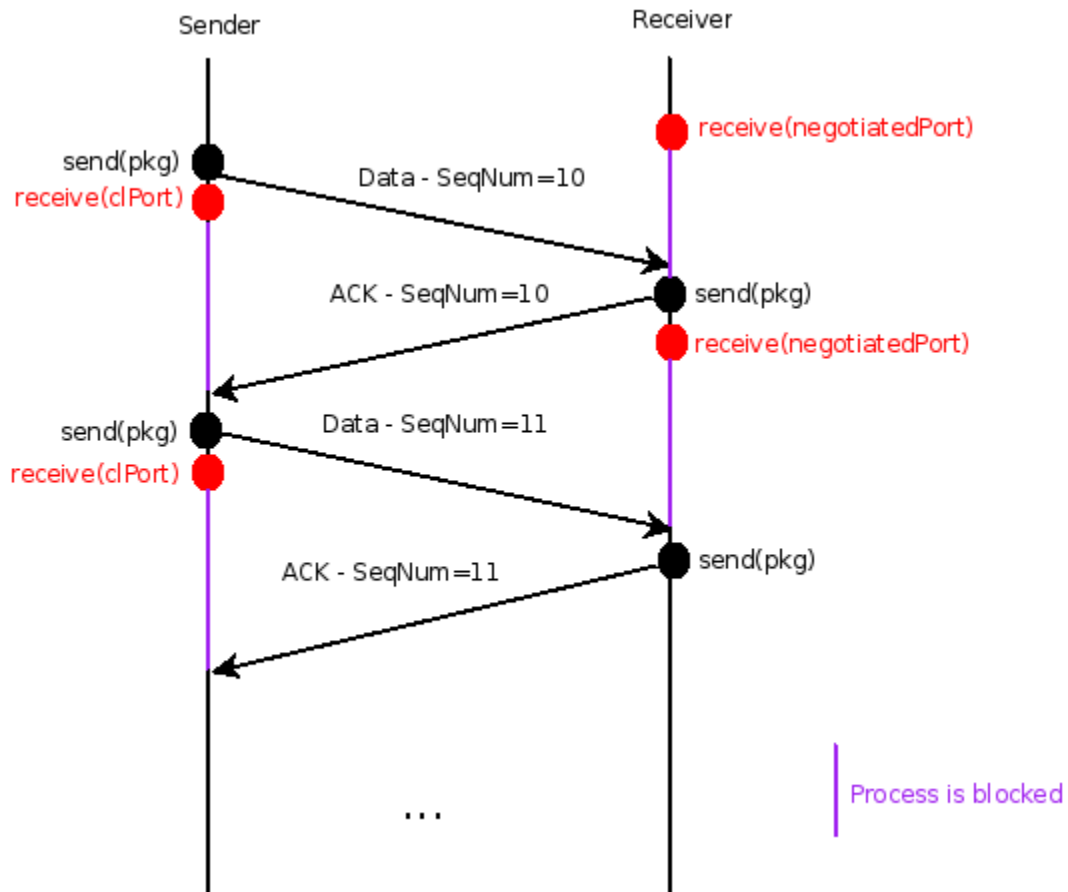
Once the connection is established, the client will connect with the server using another port which has been negotiated during this process.



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).

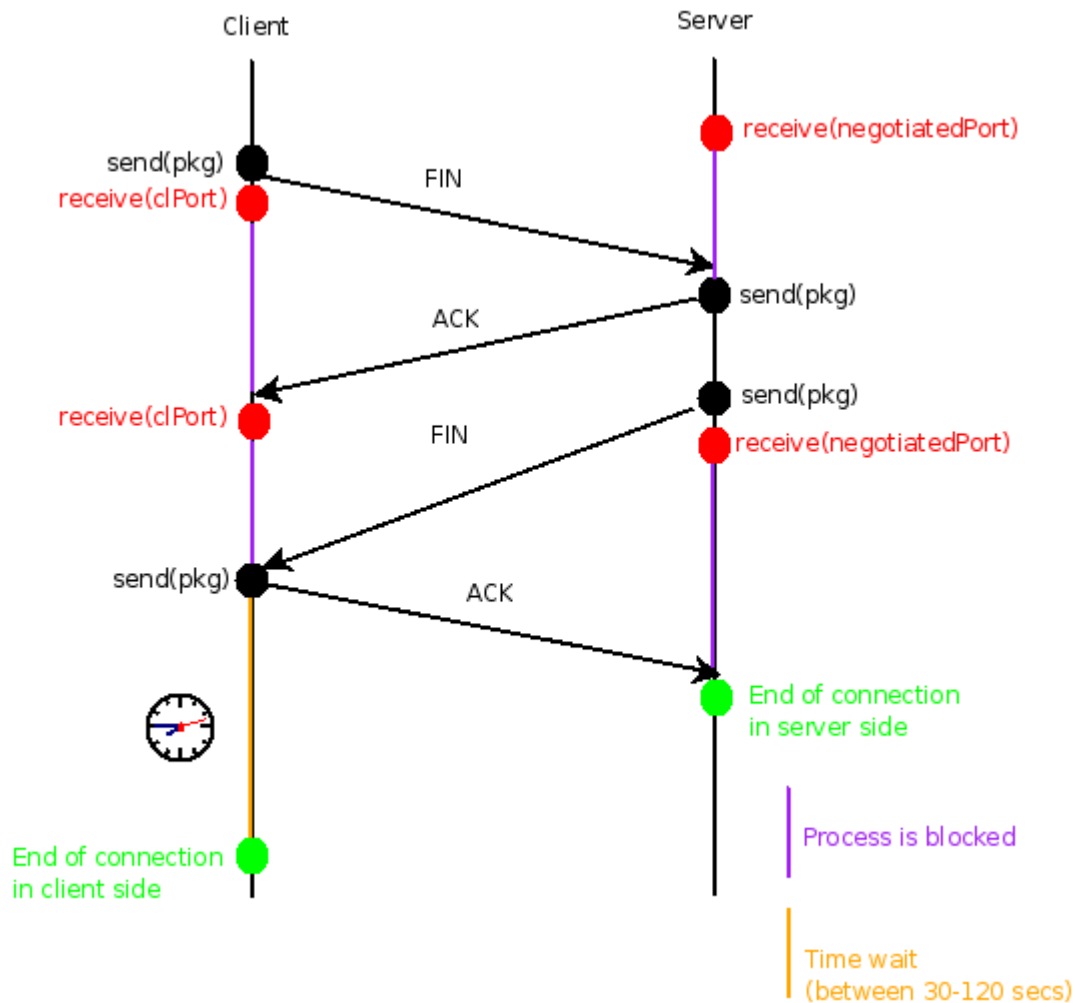
As well as the flags, the access to the sequence number is provided by `KtnDatagram` methods.



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.

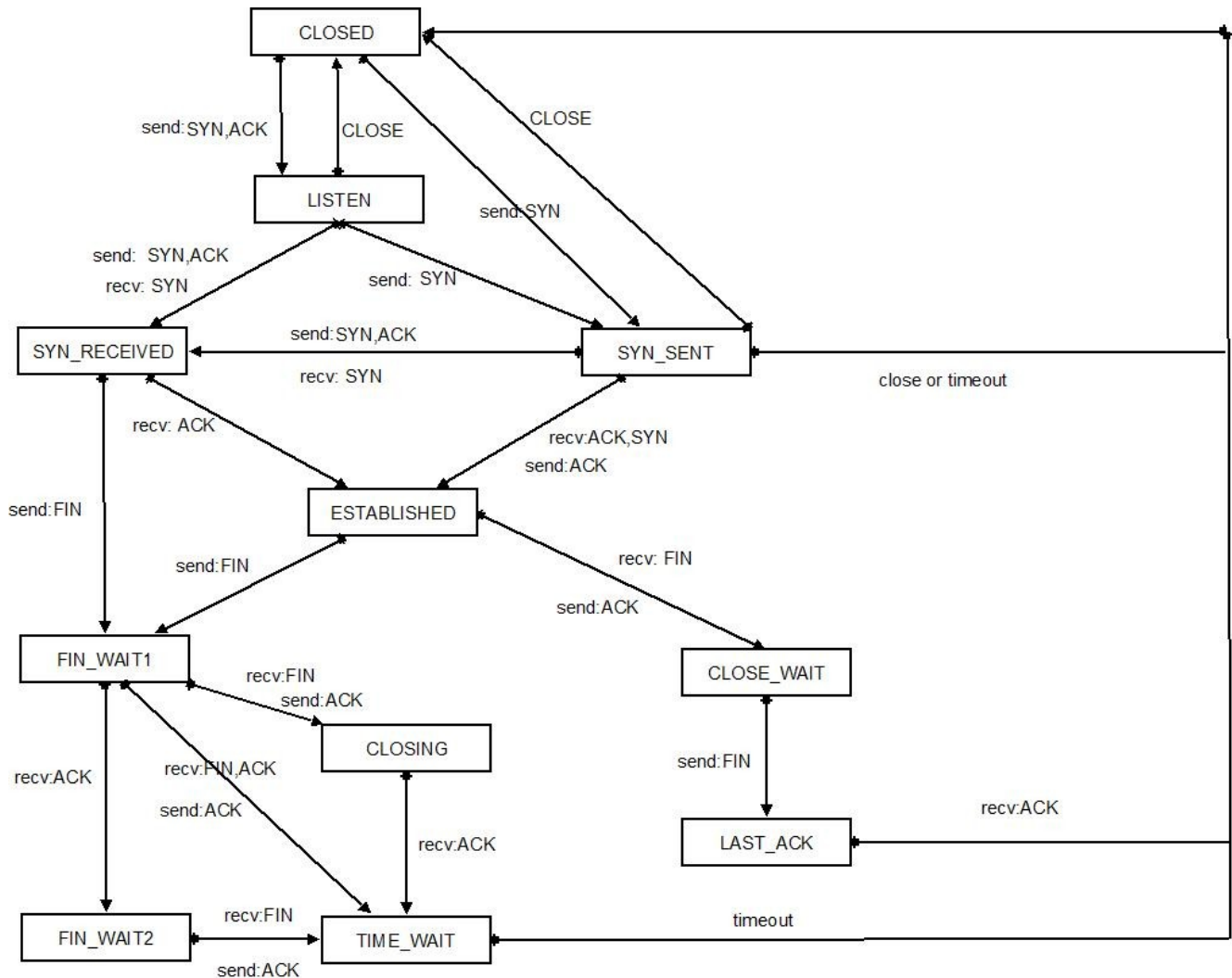
Disconnection

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. States diagram for A1

Old states diagram

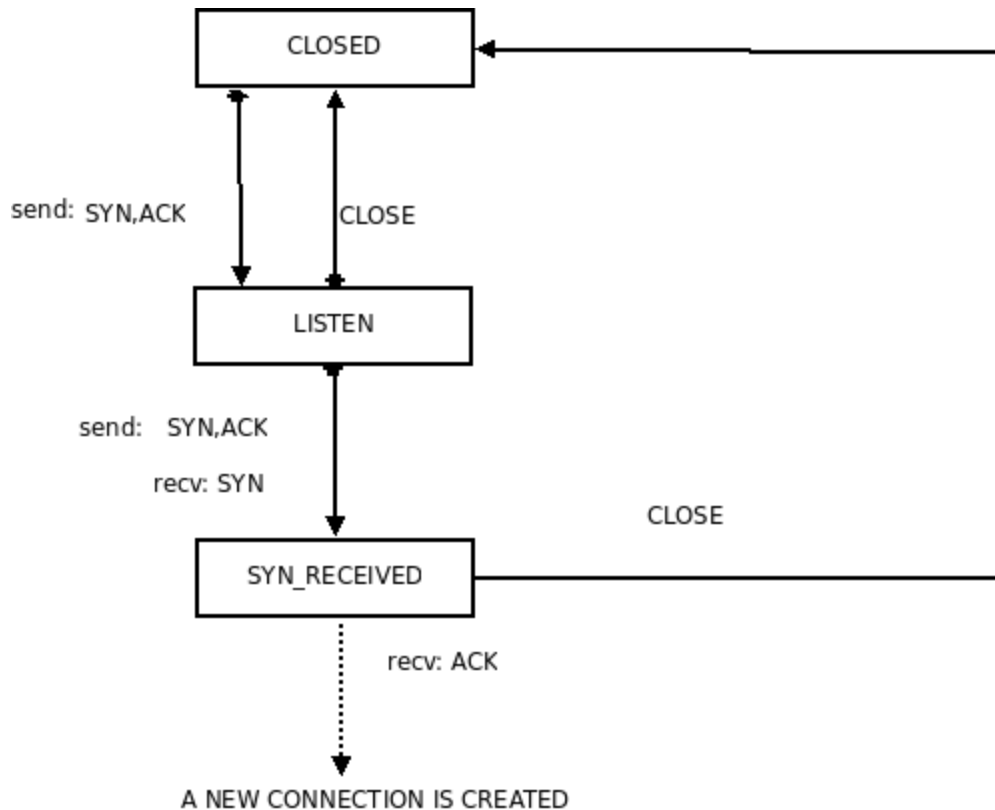


Explanation of the changes performed in the states diagram and new diagram

- The states diagram in which we are inspired is designed to have only one connection. In our case we have one connection which transits between CLOSED->LISTEN->SYN_RECEIVED creating then a new connection with state ESTABLISHED and returning to CLOSED. Therefore we have to split the scheme into two, one for the mother connection kind, which can be only one; and another one for the new connection type, which can be many.

State diagram for mother connection:

- The scheme for our *mother* connection transits to CLOSED from SYN_RECIEVED once the ACK was received as explained before. The resulting scheme is:

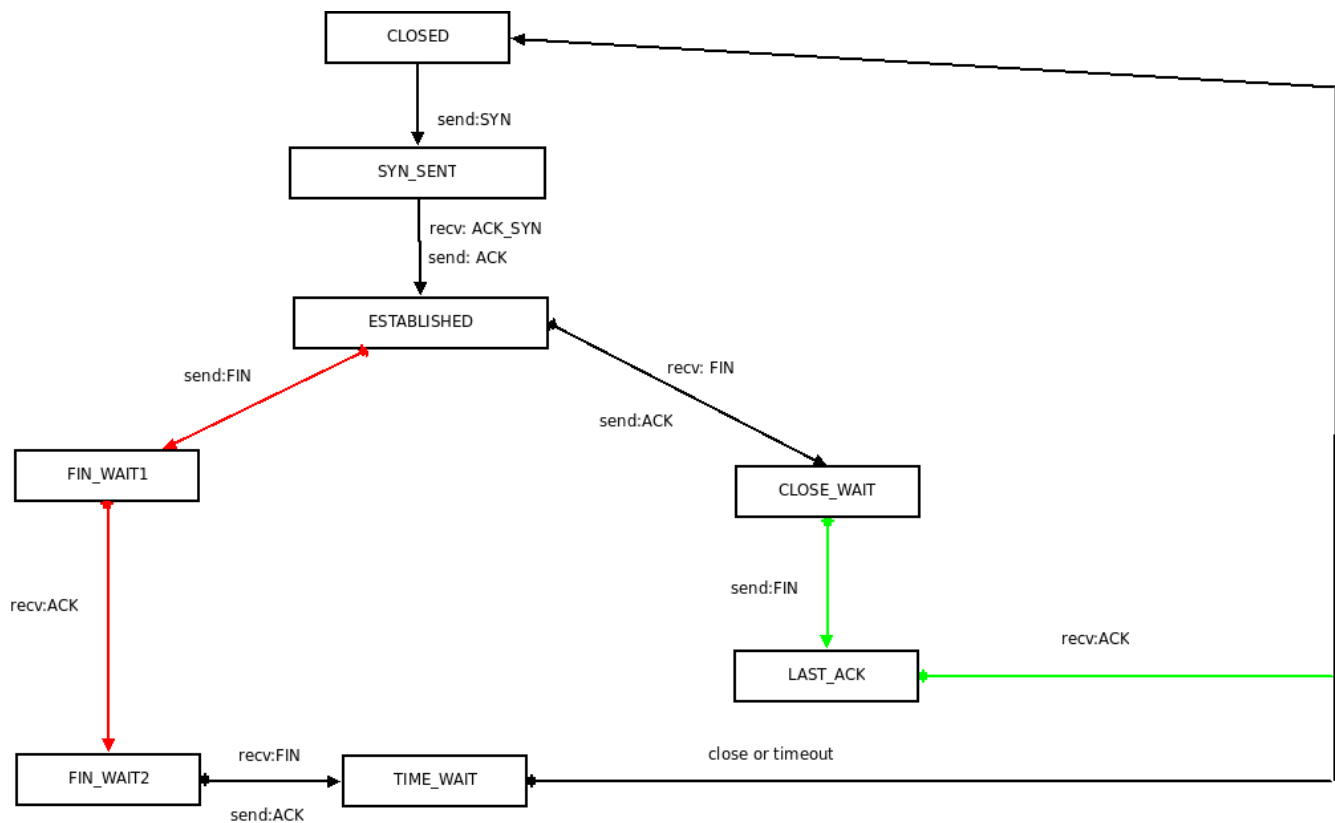


States diagram for new connection

- We do not implement simultaneous open, so the transition between SYN_SENT and SYN_RECEIVED is not necessary.
- Due to the implementation of the method sendPacket() in Abstract Connection, transitions which send a message and transit to the next step due to the reception of the ACK of this message are performed in one step. This occurs two times in our diagram:
 - **ACTIVE CLOSING**: Transition from ESTABLISHED to FIN_WAIT_1 once we have sent a FIN package, and from FIN_WAIT_1 to FIN_WAIT_2 once we have received the ACK to this package. This is performed in only one step in the method close(), as indicated in the diagram in red.
 - **PASSIVE CLOSING**: Same reasoning for transition CLOSE_WAIT-

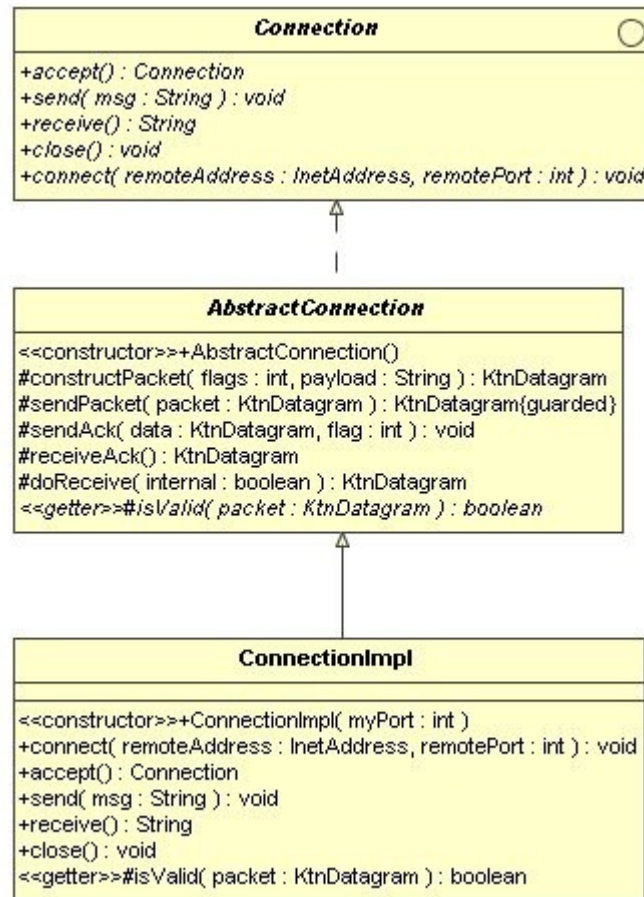
>LAST_ACK->CLOSED. We change the state from CLOSE_WAIT to LAST_ACK before sending FIN signal, just to follow the original scheme, but this state does not involve anything because sendPacket() send and receive ACK internally. This is indicated in the diagram in green.

- Our protocol does not implement simultaneous closing neither, so the state CLOSING, the transition from FIN_WAIT_1 to it and from FIN_WAIT_1 to TIME_WAIT are not necessary in our protocol.
- The resulting states diagram for the new connection is:



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.

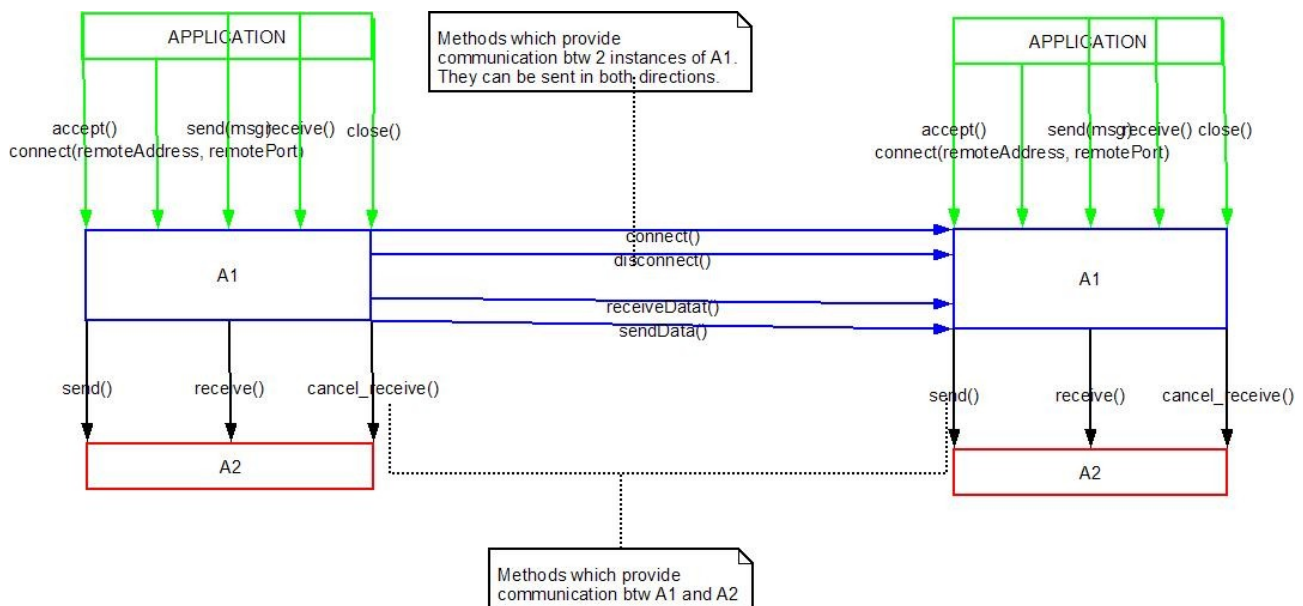


Illustration 2: Communication between application and service A1 and service A2.

Since interfaces between services are distinctly precised, that is to say, methods which communicate different services are already specified, our design must concentrate in defining 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);
    // The new port must be done here since it's the only
    // message the server sends to client in 3-way handshake
    d.setPayload(newNegotiatedPort);
    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();
    ConnectionImpl connection= new
        ConnectionImpl( newNegotiatedPort);
```

```
        return connection;  
    }
```

How to face the design and realization of A1?

It is required that five methods in A1 are implemented: `accept()`, `connect(InetAddress remoteAddress, int remotePort)`, `send(String msg)`, `receive()` and `close()`.

Methods `accept()` and `connect(InetAddress remoteAddress, int remotePort)` must implement the three-way handshake in a similar way as it was above presented.

Methods `send (msg)` and `receive()` will utilize `A2.send()` and `A2.receive()` respectively, whereas `close()` must implement carefully both active and passive closing connection, but the main idea was reflected in the previous example.

Finally, regarding to the use of Admin, we understand this package as a very useful tool to perform the system test.

We will record every attempt of connection as well as every `send()` and `receive()` in A2, so records entering in the log file allow us to analyze the behaviour of the connection and to verify if the percentage of errors is the selected in the error settings. Anyway, a more detailed description of how Admin will be used and its goals can be seen in the test plan.

4. Error handling

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

Type of error	Type of package	Response
Package 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 resends an appropriate acknowledgment.
Package delayed	Data package	Sender retransmits the package after the timeout expires. Receiver throws away the duplicate package and resends 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 away. If the sequence number is same as the previous one (thus it is a duplicate), he resends 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 too much 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 easily be implemented by using `assertEquals()` in JUnit. The rest will have to trust other criteria.

Test configurations: The requirements want 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%.