TTM - 4100

Communication

Services and Networks

KTN - PROJECT

March (2009)

Gruppe 9

alcoleaa	Alberto Fermín Alcolea Ayala
jornanl	Jørn André Larsen
vestergr	Knut André Karlsen Vestergren
trondsu	Trond Suleng
haiduyth	Hai Duy Thai Nguyen
castrosa	Miguel Castro Sánchez

KTN – FINAL CHANGES:

Changes in the test plan:

- Removed the point where we severed the connection, because we had to run the test on a single computer.

Changes in the connect & disconnect state diagram:

- Renamed all the states such that they match the premade states in the given code.
- Removed all corrupt test on SYNs, ACKs, and FINs, because we where made aware that it was not necessary.
- In the state SYN_RCVD and SYN_SENT, we changed the timeout event from resending the SYN_ACK and go back to waiting again, to change state to CLOSED and throw an exception. We also removed the timeout events from FIN_WAIT_1 & 2 and LAST_ACK. This was because in connect and disconnect there is no packet loss, and therefore no need to resend anything.
- In CONNECTED we removed the self-loop, since there no longer was any chance we would get a duplicate SYN_ACK (We removed the resending of it).
- We also added two timeout events, one out and one in, at FIN_WAIT_2 to make sure that we move on to the next state.
- Exiting the TIME_WAIT state, we changed timeout to .. because we did not deem it necessary to wait before closing the connection.
- In and out of CLOSE_WAIT we made some changes in the order things where done such that it fits better with our implementation.
- Change the trigger to leave LAST_ACK from receiving an ACK to nothing (Leave it at once).
- We also removed some unnecessary receive calls, since they did not have any function.

KTN - INDEX OF CONTENTS

1. – Message Sequence Charts

Message sequence charts for the interactions between the application, A1 and A2

1.1.- CONNECTION Sequence Chart

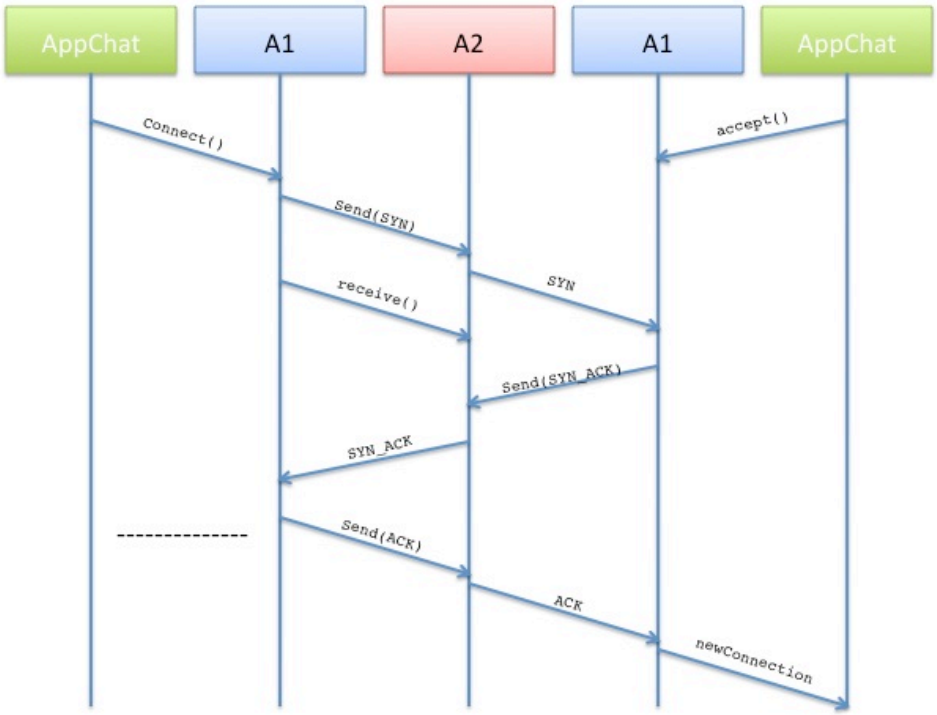

Figure 1.1.- CONNECTION Sequence Chart

CONNECT :: Brief Description

When A1 receives a connect() request from the chat application it will create a packet called SYN, containing destination IP-address, port number and so on, and send it out on to A2. When the SYN arrive at the server side A1, it will send a SYN_ACK packet back out on to A2. If the client side A1 does not receive the SYN_ACK within a certain time frame it will trigger a resend of the SYN. When the client side A1 receive the SYN_ACK, it will send its own ACK out on to A2 and tells the client side chat application that a connection has been made. When the server side A1 receive the last ACK it will tell the server side chat application that a connection has been made. If A1 does not receive the ACK within a certain time frame, a time-out will trigger a resend of the SYN_ACK.

1.2.- SEND & RECEIVE Sequence Chart

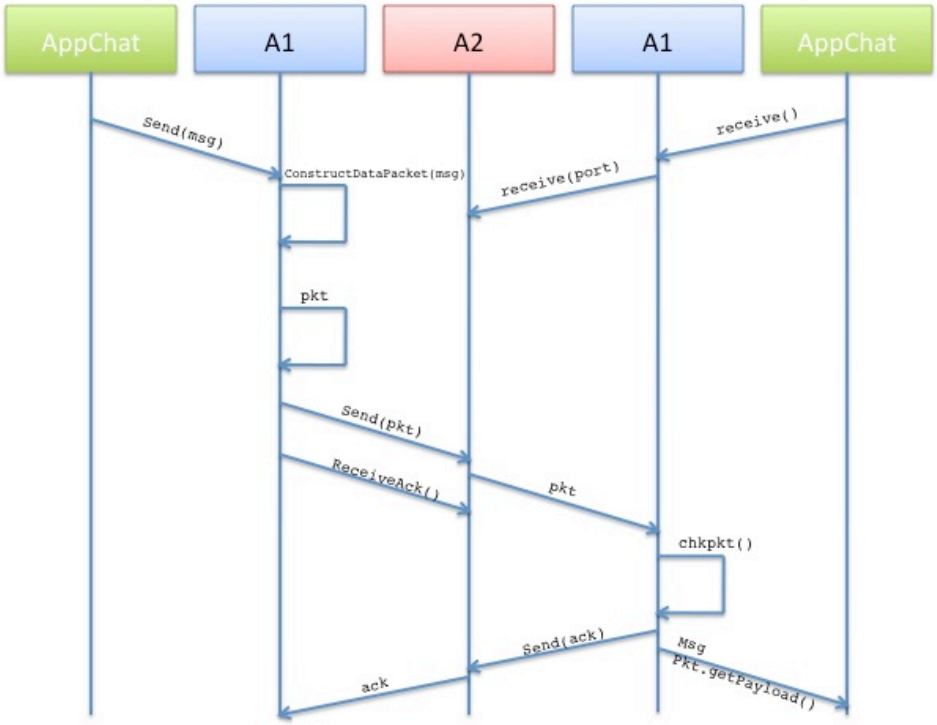

Figure 1.2.- SEND & RECEIVE Sequence Chart

Brief Description

1.2.1.- SEND

A1 will, upon receiving a send(string) call, construct a data packet (pkt) with the given text string, an increasing sequence number and the ip-address and port number of the receiver defined in the connection instance. This packet will be passed on towards the receiver, calling send(pkt) on A2. After sending the packet, A1 will start a timer and call receive() on A2, waiting for an acknowledgment from the other host confirming receiving the packet. Having not received the acknowledgment before the timer has elapsed, or receiving an old acknowledgment, A1 will resend the packet, reset the timer and repeat. If A1 does not receive an valid acknowledgment within a given time, or if it should at any time encounter a connection error, being unable to send the packet to A2, it will throw an exception.

1.2.2.- RECEIVE

A1 will, upon receiving a receive() call, call receive(port) on A2, where A2 is the connection port number stored in the connection instance. It will then wait until A2 returns a packet. Upon receiving the packet, A1 will check if the checksum matches the packet contents and that it's sequence number is the expected, and if so, it will construct an acknowledgment packet and send it to A2, extract the message from the packet and deliver it to the application. Should the packet be invalid, A1 will send an acknowledgment for the last valid packet received, and call receive(port) again awaiting the correct packet.

1.3.- DISCONNECTION Sequence Chart

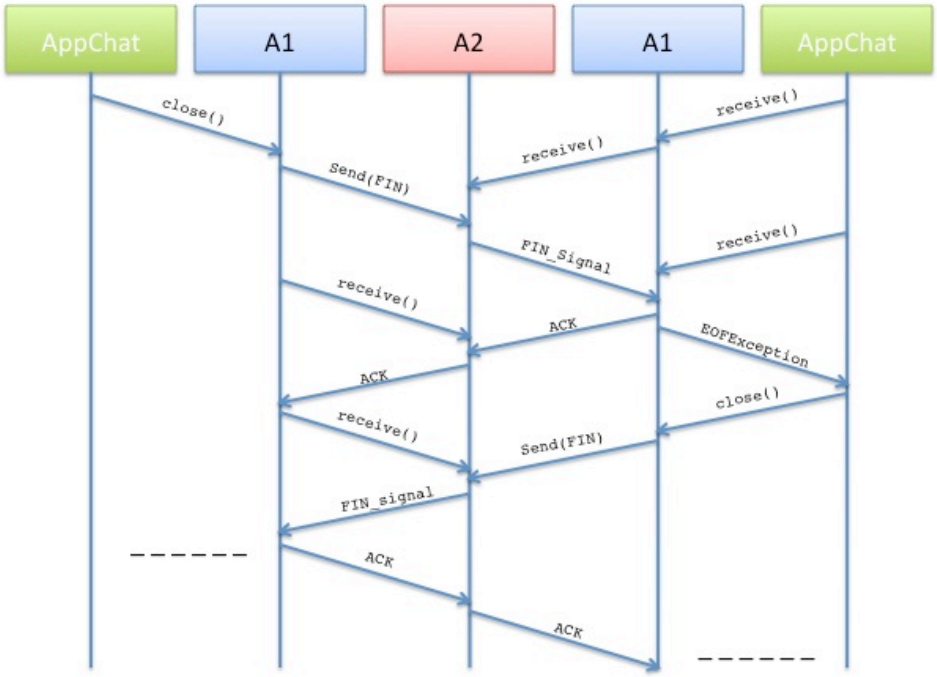

Figure 1.3.- DISCONNECTION Sequence Chart

DISCONNECT :: Brief Description

When the client side chat application wants to close the connection, it calls close() on A1. A1 then creates a FIN packet and sends it out on to A2. When the FIN has been received by the server side A1, it sends an ACK packet out on to A2, throws the EOF Exception to the server side chat application and sends the FIN packet out on to A2. If the client side A1 does not receive the ACK within a certain time frame, it will trigger a resend of the FIN. When the client side A1 gets the ACK and FIN, it will send another ACK confirmation out on to A2 and awaits for a while (typical values of 30 sec, 60 sec, 120 sec) before closing the connection.

2. – State Diagrams for A1

2.1.- CONNECTION & DISCONNECTION State Diagram

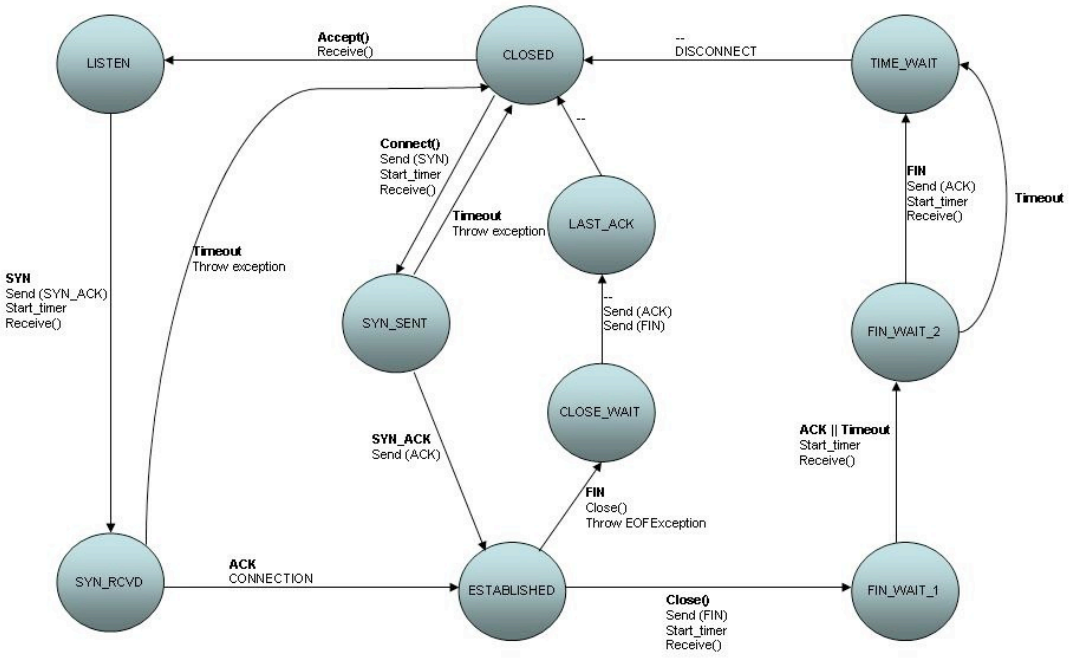

Figure 2.1.- CONNECTION & DISCONNECTION State Diagram

3. – Error Handling

3.1.- Description and Detection

This section offers a description of how to handle the different errors that can occur during the course of execution. The errors, causes of errors, and consequences are depicted in section 3.1 of the user documentation A2. The errors are:

- Package lost.
- Package delayed.
- Package has error in payload.
- Package has error in header.
- Ghost package.

Detection:

- **Loss:**
  If a packet is lost, either from sender or from receiver, it is easily detected as a timeout will occur. This works either way. The ACK was never received and a timeout makes sure the packet is resent.

- **Delay:**
  If a packet is delayed, a timeout will trigger if the delay is to long. The problem with this event is that a packet could appear two times, because the sender thought of it as lost (see above). This can be detected by checking the packets sequence number. If this number is not larger than the current sequence + 1, we know that the packet is the same as one we already have received.

- **Error in payload:**
  The data of the package is corrupted in some way. If this gets through to the receiver, it could result in messages being unreadable nonsense. The documentation A1-doc states a method for checking if a package has errors, though is has not been created yet, this can be used by the receiver to detect errors. It is desirable for us to only use this method to check for any type of error, so the isValid()-method will include checks to validate checksum field, address fields, ACK field and sequence field.

  For this type of error, we only need to check the checksum of the package, since payload errors is directly shown by a change in checksum, one can use calculateChecksum() to find the correct value, and use getChecksum() to find the current package checksum and then compare them to see if the package is corrupted in payload.

  The corrupted packets will be dropped and the error will be treated as a **loss** and hence solved by the *timeout/resend* methodology.

- **Error in header:**
  The user documentation states that the consequence of this error will result in the package ending up in the wrong place, in other words, it therefore becomes an outgoing ghost package, and is a lost package in the context of error handling (see above, and below). Other errors could also occur since both ACK, sequence and/or checksum also could be wrong. These errors can be detected by the use of the isValid() method described in A1-doc and in "Error in payload" above. An idea to solve this is to run the isValid() check on all packages and act upon the types of errors detected. If

checksum is wrong ,the package is treated as having error in payload. If the addresses are wrong, the packet is treated as a ghost. If the ACK is wrong, it can be treated as lost. If the sequence is wrong, it is treated as delayed.

- **Ghost:**
  Suddenly, from nowhere, a packet appears at the receiver which is not sent by the sender. This packet can be detected by checking the senders address in the header of the package. If this is different from the one set up in the connection, the package is considered to be a ghost-package

3.2.- Error Handling (Schemas)

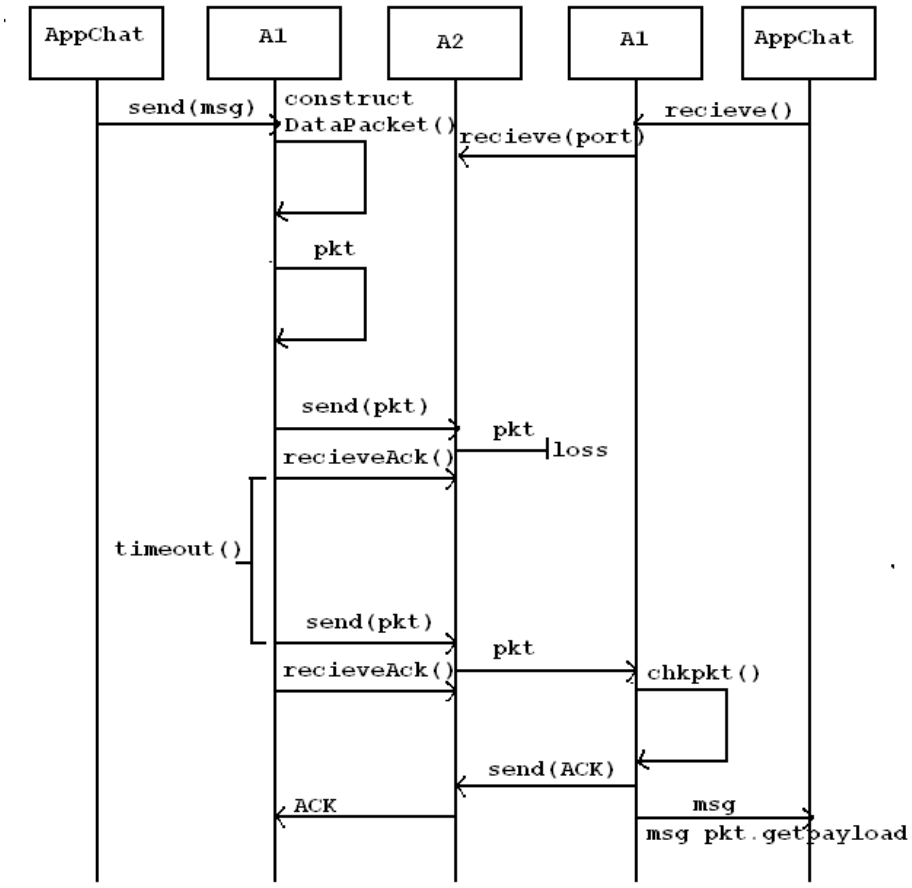

Figure 3.1.- Package Lost

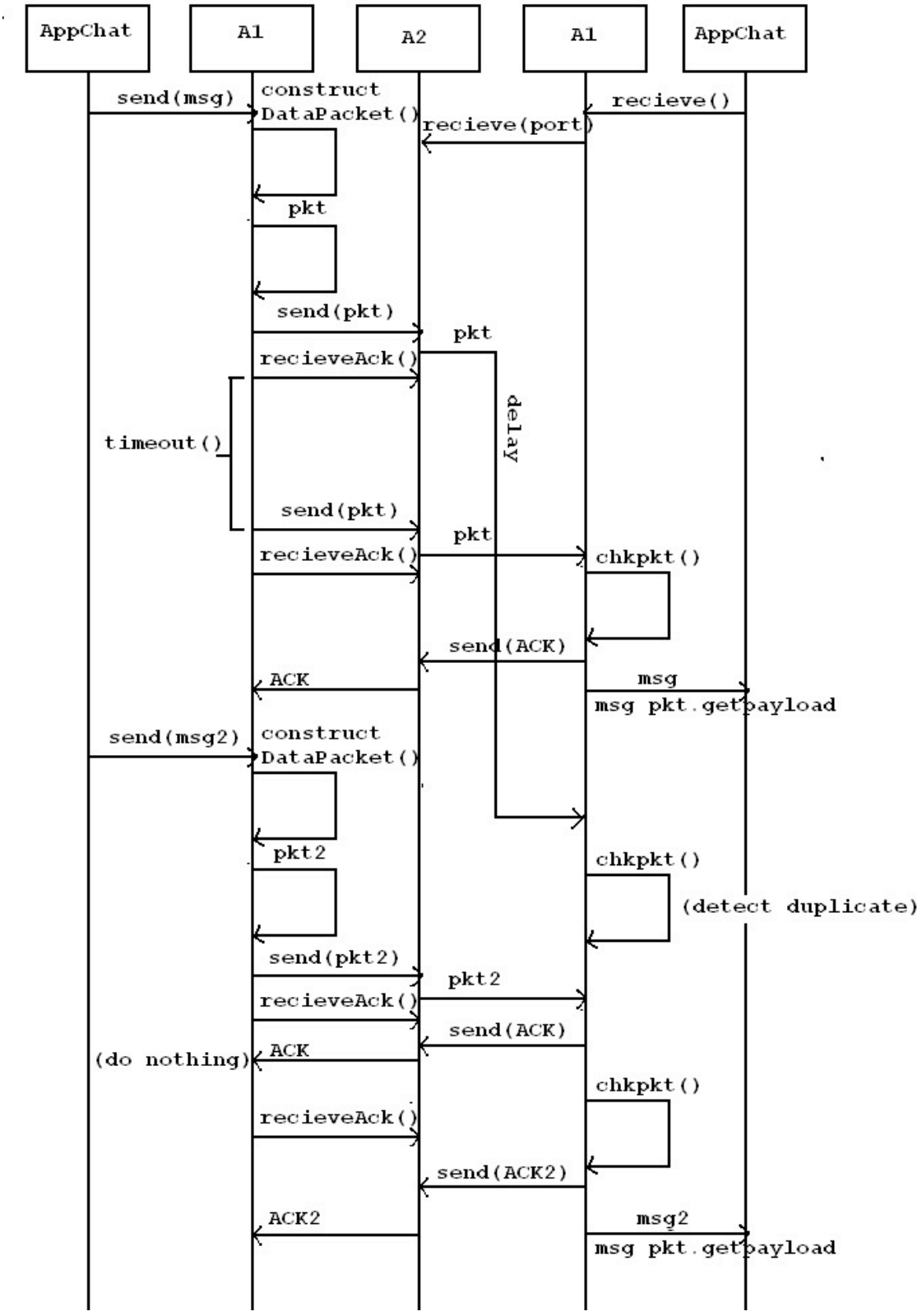

Figure 3.2.- Package Delay

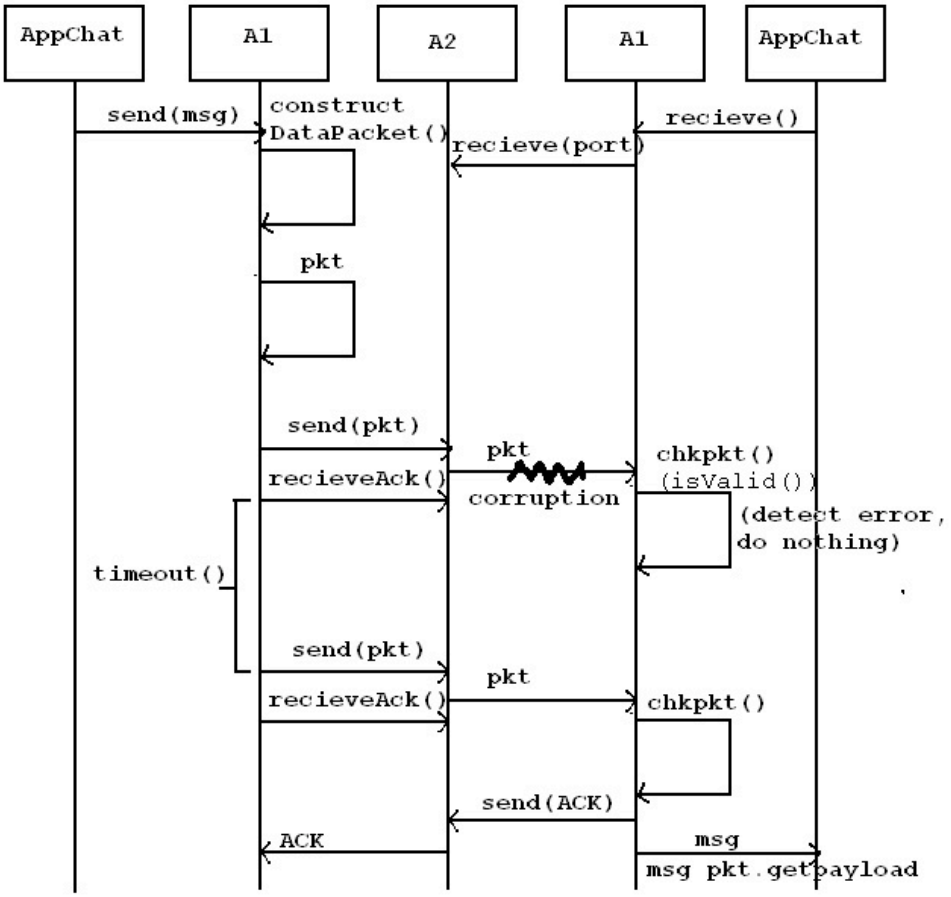

Figure 3.3.- Package Error

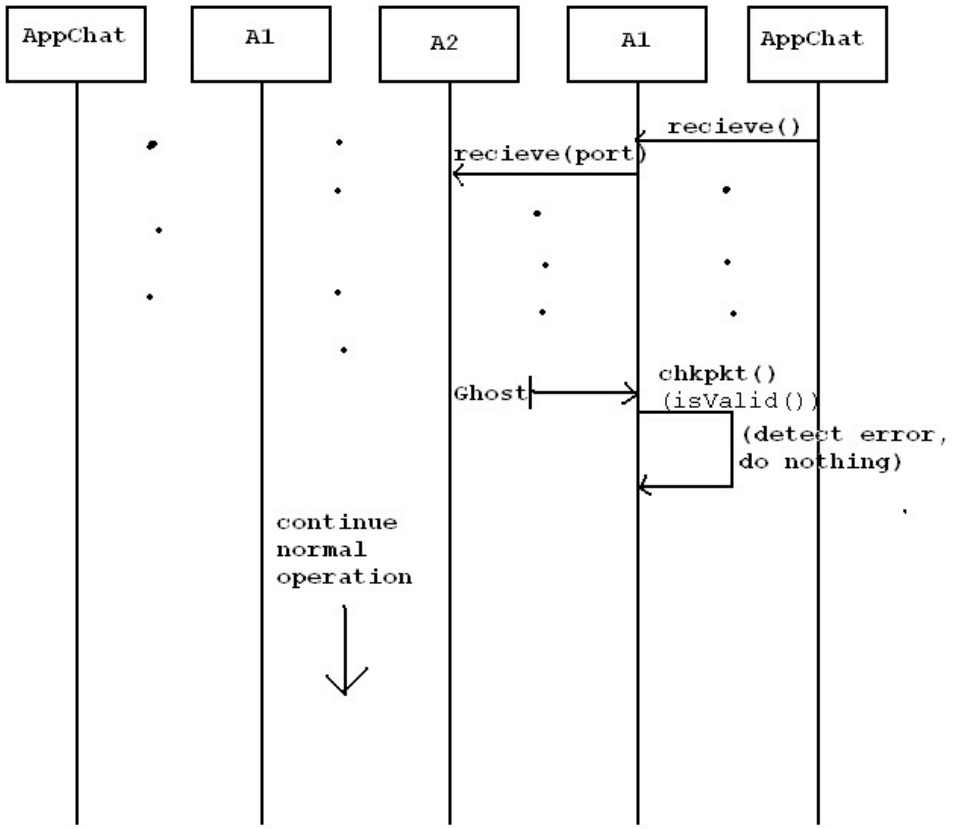

Figure 3.4.- Ghost

4. – Test Plan

4.1.- Actions and Expected behavior

The following actions will hereby be referred to as the basic test:

Action:
Open a connection from one host to an other

Expected:
If there exists a network connection, we will expect the return of a Connection instance within a limited time period. In the case of a severed network, we will expect a correct exception thrown.

Action:
Send a number (50) of messages from host to host

Expected:
All the send messages should be delivered intact in the right order without duplicates.

Action:
Perform a disconnect from host.

Expected:
The connection should shut down controlled on both sides

4.2.- Enviroments

The basic test should be performed in the following environments. If an environment produce errors, the whole test plan will be repeated from the top if the error result in a new version of the code. The code will be updated if an error is by close inspection to a packet trace of the sent communication and the built in logs of the Admin module found to be caused by it.

Environments:

Fault free:
Admin module is set to simulate loss free deliveries without delay.

Single fault:
In turn, corruption, loss, delay and miss deliveries (ghost) of packets will be enabled with 10% and 50% probability.

Multi fault:
Corruption, loss, delay and miss deliveries (ghost) of packets will be enabled all at once with 10% and 50% probability. Should this test produce faults after a successful completion of all the test in the single fault environment, new environments with faults activated in groups of two and three will be tested to isolate what faults will produce the errors when occurring together.

All the tests will be documented, together with any correction done to the code between each environment.