# Architecture

We came up with this specific architecture in correlation with our scenarios. We will implement a listener to receive messages and this will use an event-loop. This event-loop is necessary because we don’t know in what order and fashion the messages will arrive. So we need to be listening all the time and prepare to handle them. To help this process, we need to make sure that the event-loop thread is never too busy. That is why per each connection we create a new thread. This means that we are going to be using a multi-threaded architecture. A shift towards multi-process architecture would be a bad decision, mainly because of two reasons:

1. Inter-thread communication (sharing data etc.) is significantly simpler to program than inter-process communication.
2. Context switches between threads are faster than between processes. That is, it's quicker for the OS to stop one thread and start running another than do the same with two processes.

The listener will then pass on the message to the validator. The validator checks the format of the message and checks to see if there is a mistake with the message according to specification. If there is a mistake with the message then the valuator logs the message in a logger, but regardless passes the message on to the sender, which takes the message and puts it back on track to its original destination. If the message has no mistakes, according to the specification, then the validator passes the message on but instead of going straight to the sender it could get passed through a quality control checkpoint. This checkpoint is responsible for peer-to-peer quality checks. For example it could participate in a scenario designed to measure the end-to-end delay. I.e. To see the time lapse on when the message was sent as to when it was received.

It has to be noted that each scenario (quality check or validation) is simply a function that starts certain module(s).

# Quality Control

We planned to derive the quality measurements from the user requirements. These requirements include Bandwidth, Delay and Availability. These requirements may not ensure the exact satisfaction of the user. For example these requirements do not make sure if the user is hearing a pleasant voice over the phone. But these concerns are usually dealt with after development and during a “beta test” if applicable. Anyway, while we are open to changes, and to enhance our quality tests and measurements in near future, we also claim that mentioned quality measurements (Bandwidth, Delay, Availability), could ensure a satisfactory user experience.

Multiple test scenarios will be developed, each to measure a specific parameter. Quality controller is a module that is responsible for these tests. It is a Module that uses KX module as means for an end-to-end connection. This means that KX will believes it is delivering a call, while it is actually exploited to deliver test messages.

## Measuring Delay

The scenario is initiated when the peer that wants to measure the delay sends a message to another peer, telling it to respond with a message. Then the delay is calculated.

1. DELAY

|  |  |  |
| --- | --- | --- |
| <quality test identifier> | DELAY | <timestamp> |

1. The <quality test identifier> is a Hard-Coded Random String that makes our Validator aware that this message is to be forwarded to the Quality Controller module. This is necessary because we are sending our test message as if it is a regular message from VOIP. There should be some method of distinguishing it. Yes there is a possibility that in some cases, the data sequence is by chance identical to our identifier. But the longer the identifier is, the less chance that this happens. We would pick a 256 bit hard coded identifier for all of test messages. And the chances that a conflict happens are (0.5)­­­256. This seems enough for a testing module.
2. DELAY is the type of this test message. One byte length is sufficient for such a use. The receiver, upon getting such a message, should response with a DELAY\_RESPONSE message.
3. DELAY\_RESPONSE

From here it is pretty much straightforward. The node, receiving the DELAY message responds with the same timestamp that it had received. When this timestamp is delivered to the initial node, then it could measure the time needed to a message to go and come back divided by two: i.e., Delay.

|  |  |  |
| --- | --- | --- |
| <quality test identifier> | DELAY\_RESPONSE | <sender timestamp> |

## Measuring Bandwidth

As a pragmatic approach, we designed a scenario, where peer Bob sends Alice a reasonable sized file and measures the sending time. File size divided by time, results into bandwidth. To make sure that the correct data is transferred, Alice and bob should communicate the Hash of the mentioned file.

1. BANDWIDTH

|  |  |  |  |
| --- | --- | --- | --- |
| <quality test identifier> | BANDWIDTH | <last message> | <data> |

The <last message> is a Boolean phenomenon and using this field Bob indicates when the sending of data is over and Alice should respond accordingly.

1. BANDWIDTH\_RESPONSE

|  |  |  |  |
| --- | --- | --- | --- |
| <quality test identifier> | BANDWIDTH\_RESPONSE | <hash of data> | <timestamp> |

The <timestamp> shows when Alice has received the last message. Bob receives BANDWIDTH\_RESPONSE and computes the hash of data himself and compares the two hashes. If they are the same, we have executed a valid test and the bandwidth could be easily calculated. If we have not received the right hash, simply the test has failed with no result.

## Availability

In this scenario we would measure the up-time of the system. In other terms we would measure the percentage of time that the system is liable to deliver a call. To reach this goal, we pick a certain amount of time, and periodically send messages to a certain peer during this time. It could include a one-hour test, a one day test, or a permanent testing strategy.

1. AVAILABILITY

|  |  |  |
| --- | --- | --- |
| <quality test identifier> | AVAILABILITY | <message number> |

Bob sends messages each N seconds/minutes, and Alice, the receiver, has to provide a response to every message she receives. This way, Bob will know which messages had been received and then answered and so the effective up-time of the system is measured by Bob.

1. AVAILABILITY\_RESPONSE

|  |  |  |
| --- | --- | --- |
| <quality test identifier> | AVAILABILITY\_RESPONSE | <message number> |

It has to be noted that some universal time out should be used for every message.

# Exceptions

We will handle exceptions for our scenarios in a couple different ways, depending of course on the type of exception that we are dealt. For instance, if the data that was sent through was corrupted, we would still do the normal procedure of putting it through the proxy, and pass it along to the intended receiver, but as soon as our proxy found that the data was corrupted it would make a log of what was wrong with it. We also might even drop some corrupted data into the thread to see how the DHT, KX, or VIOP would respond to it, and then the same process would be followed of making a log of the corrupted data and storing it away. In the instance of a peer crashing, it is not our responsibility as a testing group to respond to a peer if it crashed, we do our job of passing the message along from the sender to the receiver, and if the message is not received because of a crashed peer than we would make a note of it in the log and drop the message.