CHATOS

TECHNICAL REPORT

CHARAMOND Lucien
PORTEFAIX Maxime
M1 Info - G1

Summary

Introduction

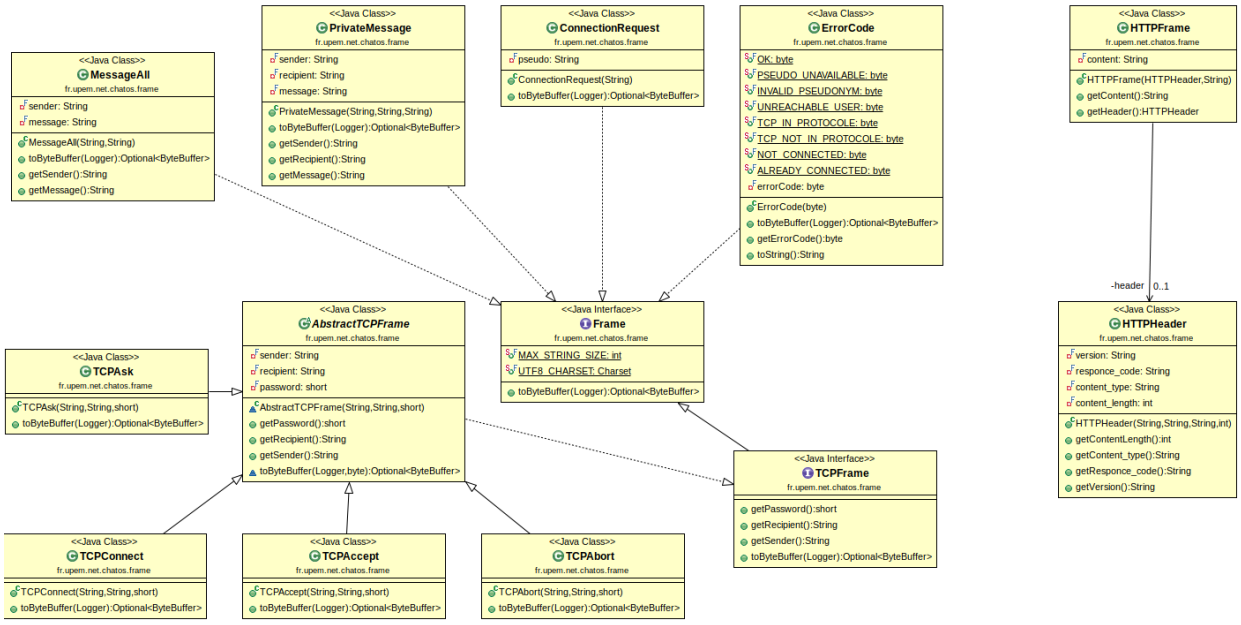
This document is the technical report of the project OSChat developped by CHARAMOND Lucien and PORTEFAIX Maxime. In the differents parts we will analyse the code and explain the context and the architecture.

Architecture

This part will explain the differents implemented architectures illustrated by UML graphes.

- **Frames**

The frame package represent the data of the different packet which could transit on the network. The UML graph is the following :



UMLFrame

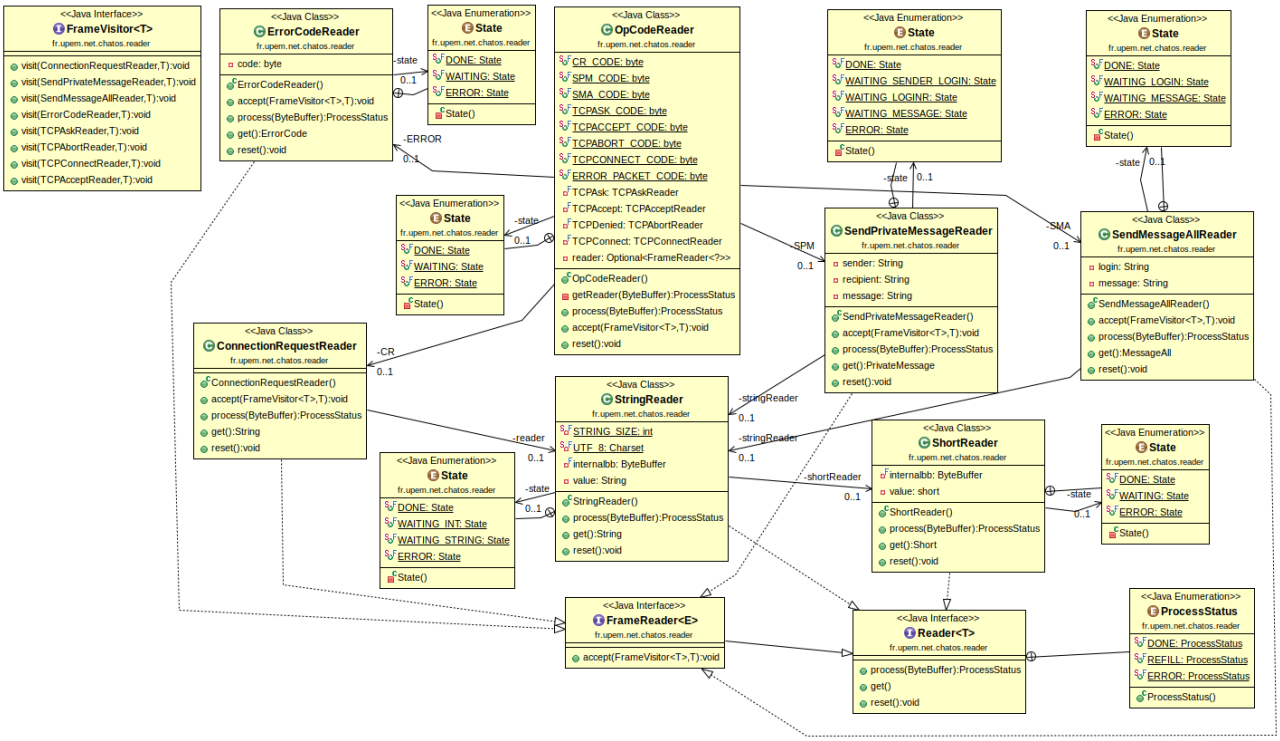
There are a lot of class but this part of the project is simple,
- the Frame interface assure that all frames can be transform as a ByteBuffer
- each class which implement the Frame interface represent a request (like MessageAll, PrivateMessage...)
- exeption for HTTP which is divided in the Frame and his header
- an other exeption for TCP which use some clean legacy with an interface and an abstract class. The TCPFrame interface assure that all TCP packet (frame) can get the recipient, the sender and the password of the connexion. So each class extending AbstractTCPFrame represent a request, a TCP packet (like TCPAsk, TCPConnect...).

This architecure is really simple, to add a new packet type in ChatOS, we just have to add a class which implement the Frame interface and write the methods.

- **Readers**

The reader package implements all non-blocking readers for all packets of ChatOS. There are too much class and dependancies to hold in a single UML graph then we divide in 3 parts : the ChatOSProtocol (different packets), the HTTP part and the TCP part.

The ChatOSProtocol is the following :



UMLReaderChatOsProtocol

The first thing to talk about is the interface FrameVisitor as you can see on the left of the UML graph. This interface is the visitor to accept (with the interface FrameReader) and visit the code with the right frame (packet). The visitor is really a pattern adapted for this situation.

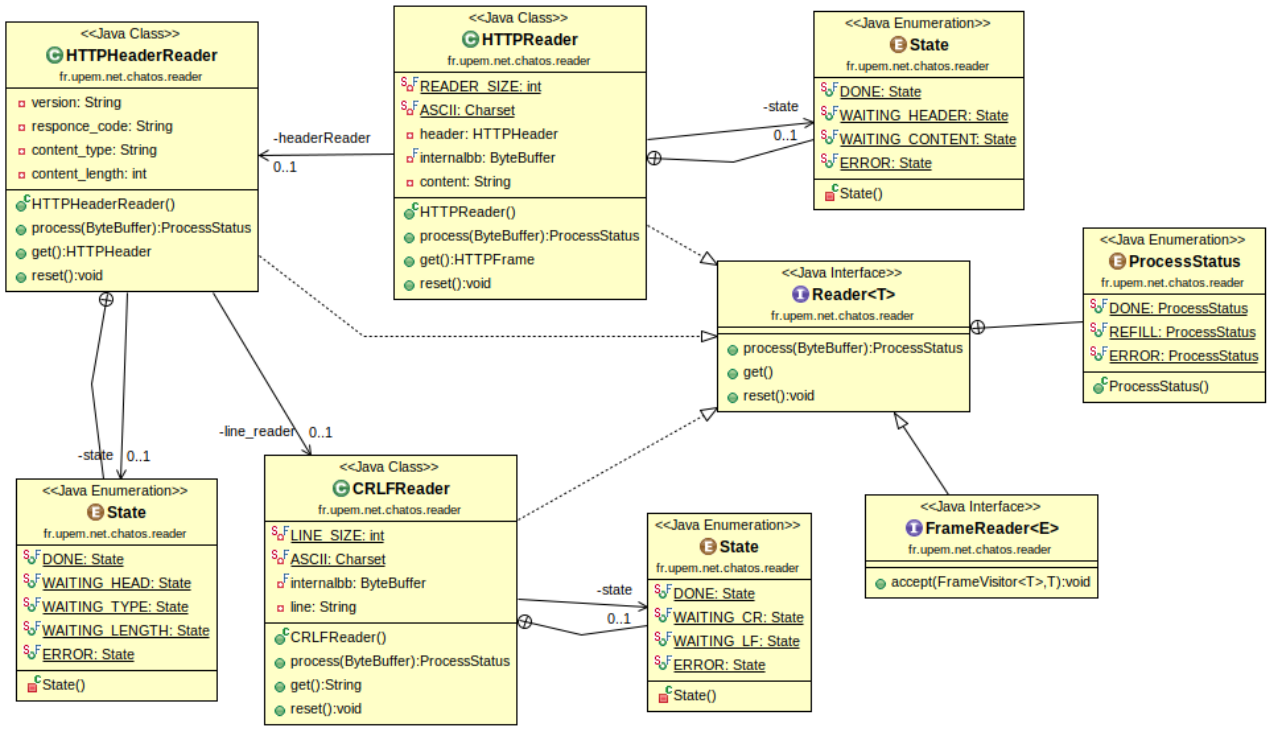
Then all other class turn around the interface Reader which assure that each class implementing it can process a ByteBuffer, get the result and reset the reader. This interface is accompanied with an enumeration ProcessStatus which represent the direct reading status, if it is running, over…

Each string transiting on the network is use with a short to control the string length (thats why we have a StringReader and ShortReader).

The class OpCodeReader represent the reading of a ne packet (we don’t know yet which packet type it is), so this class have as a field every packet Reader ready to be read.

Finally each ‘packet’ class are accompanied with an enumeration State which reprensent the next step to do for the concerned reader (is he done, waiting, on error?).

The UMLReaderHTTP is the following :



UMLReaderHTTP

This part is more simple, the Reader interface is the same that the previous part.

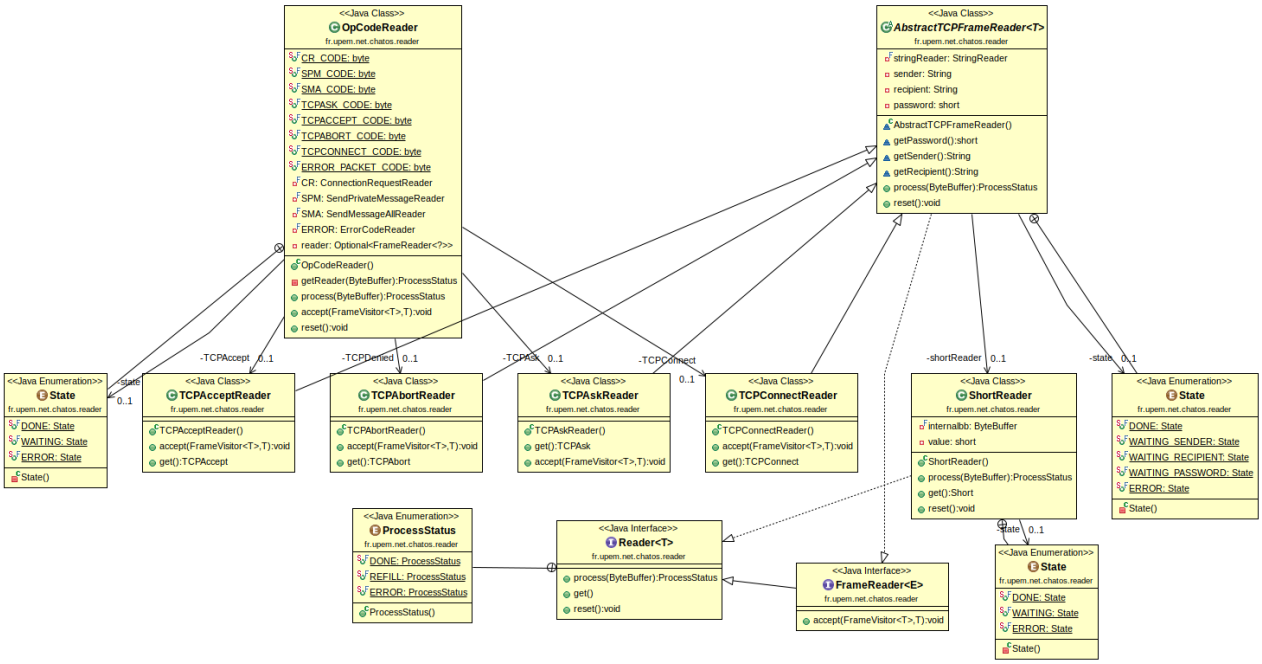
To read a HTTP message, we need to read each HTTP line, it's the job of CRLFReader (because a HTTP line end with \r\n). Then the class HTTPHeaderReader can read a simple HTTP header like this example :

« *HTTP/1.0 404 NotFound\r\n*
Content-Type: unknown\r\n
Content-Length: 0\r\n »

So this header is a HTTP answer in version HTTP/1.0, the answer is not found, the type is other than txt (in ChatOS HTTP content type is text or other), and the length is 0 (because not found).

Finally, the class HTTPReader can read the answer content (after the header) as a string and limited with the content-length token.

The UMLReaderTCP is the following :



UMLReaderTCP

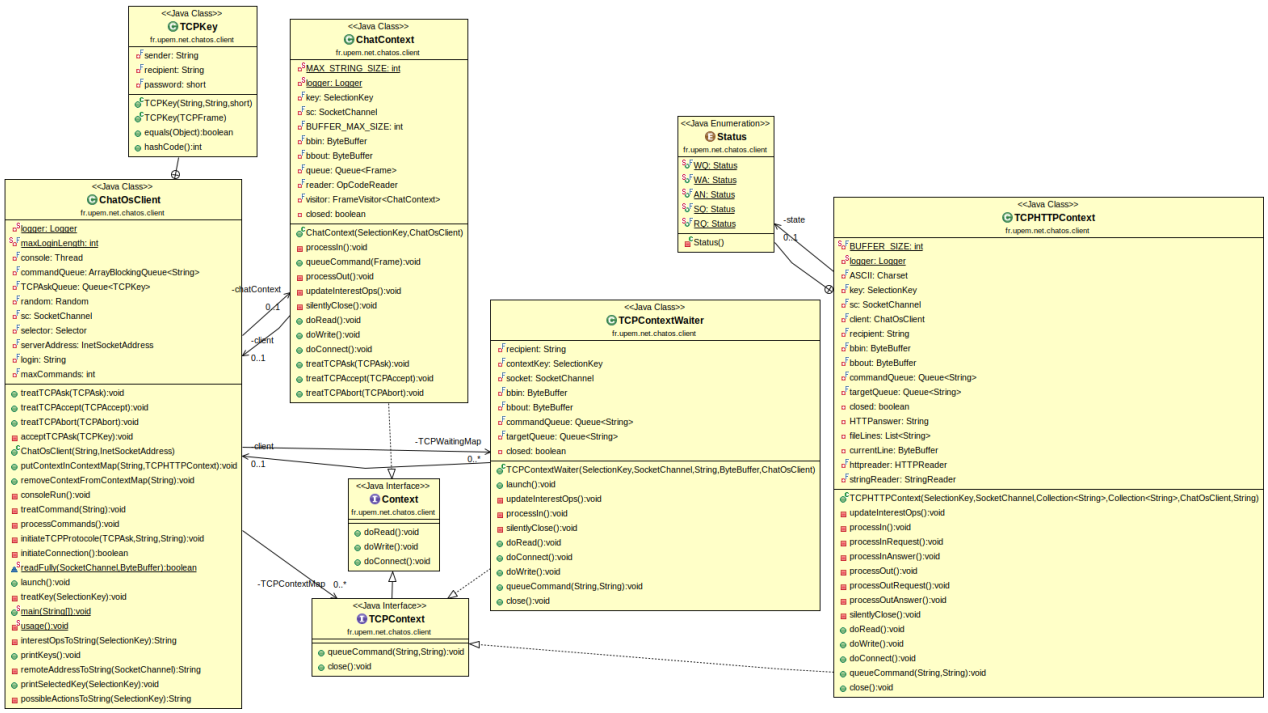
The Reader, the FrameReader and the OpCodeReader are the same that in the previous parts.

We put some clean legacy (implementing the interface FrameReader) to read TCP frames AbstractTCPFrameReader. This class assure to get on a TCP connexion it password, the sender or the recipient pseudonym.

With that each class concerning a TCP packet reader extends this abstract class and can accept from the frames visitor or get a new TCP packet the connexion identifier (pseudonyms and password).

- **Client**

The client package implement the ChatOS client which can connect to a ChatOSServer. The UML graph of the client part is the following :



UMLClient

Only the client is a public class, others are visibility package (not TCPKey), indeed the three contexts was previously internal class of the client and this last had 1000 lines, so we divide it properly.

First the class ChatOsClient can launch a client and connect it with a socket channel.

Then, the ChatContext is the initial classic chatting area (on the shell), this context can read and/or write from/to the server (connected by a socket channel). If this class is forced to treat a TCP packet the handling will be transferred firtly to the client and then added to the concerned TCP request container. The TCPKey class represent the key of a map of all TCP connexion, necessary to recover a specific connexion.

The TCPContextWaiter is just here to handle the connexion initiation. He can also read and/or write from/to the server, and when this connexion is completed or aborted, he transfer a new socket channel to the class TCPHTTPContext, a new because we still want be able to send other packet on the ChatContext even if the private TCP connexion is running. This context is accompagnied with an enumeration which is the different status of a HTTP packet transiting on the network (like : pending for a question, pending for an answer, running the answer, sending the question or reading the question), you can refer to the methods processOut and processIn to see how the automaton work.

The both interface Context and TCPContext are here to master context and ensure that each context can read, write and connect.

- **Server**

The server package implement the ChatOS server which can accept (or not) clients. The UML graph of the server part is the following :

UMLServer

Like the client part, only the class ChatOsServer is public, others are visibility package (not TCPLink and TCPKey) to relieve the server class.

First the class ChatOSServer can launch a server and initiate necessary field like the client pseudonym map (because we don’t want to have two client with the same pseudonym), or bind the the socket address.

The class TCPKey have the same goal as TCPKey from the client part and the class TCPLink is the class which link two client connected by a TCP private connexion by ettablish two TCPContext.

Finally the class ChatContext, WaitingContext and TCPContext are the server context to handle read/write from/to the differents client (same distribution as the client part).

Tests

Evolution Since Beta

Features

Uncounter difficulties