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. ChatOS is a chat service which can connect several clients to a server. Furthermore, ChatOS can establish a TCP private connection between two clients without disclusing the clients IP addresses.

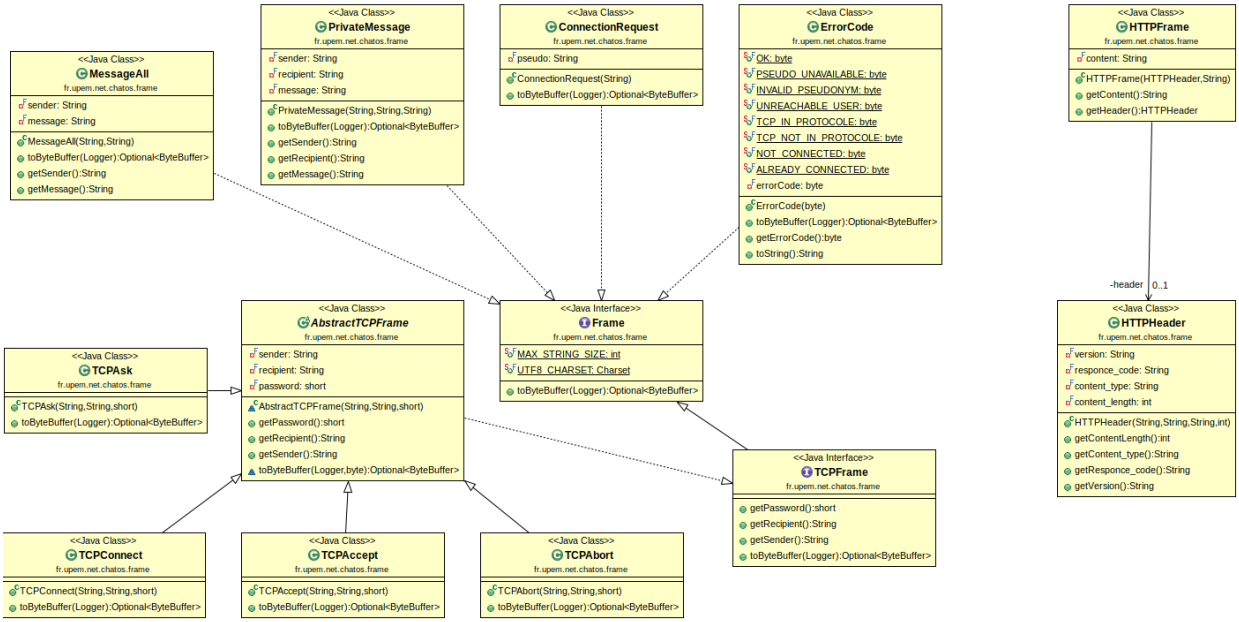
In the differents parts of this document, we will analyse the code and explain the context and the architecture.

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 :



Frame

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 is 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

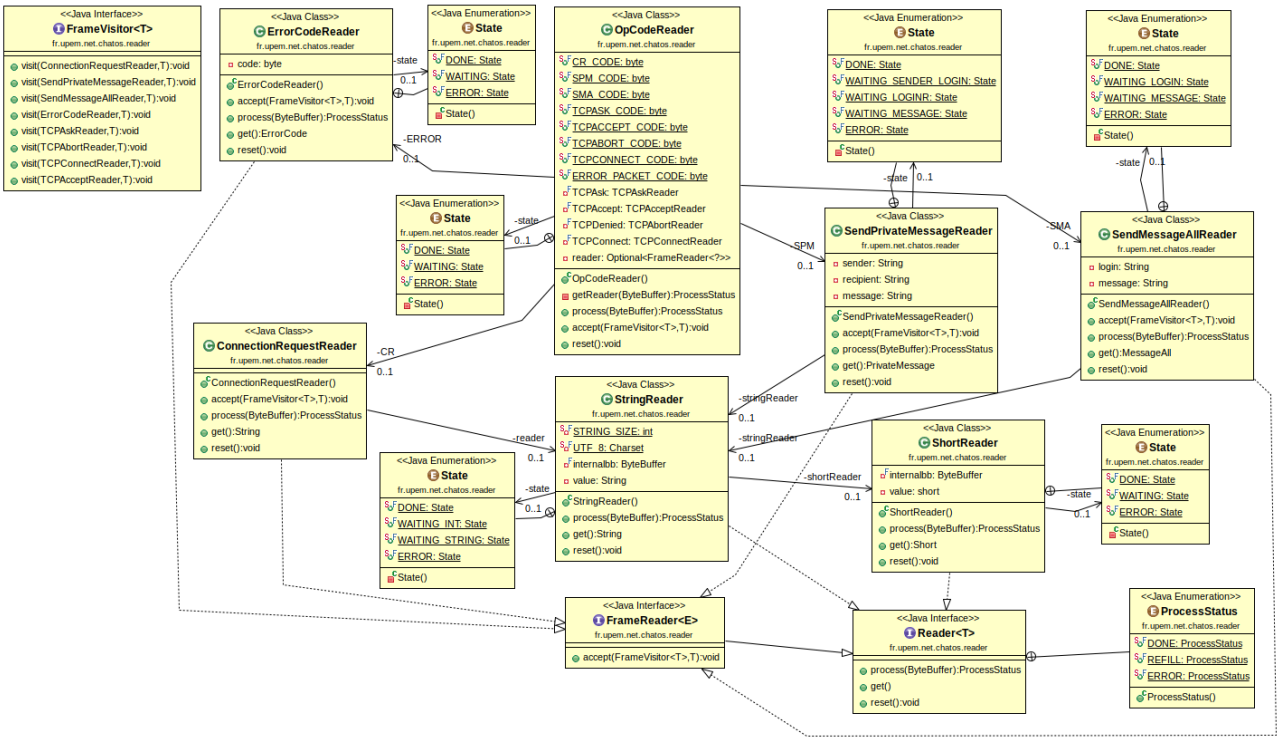
connexion. So each class extending AbstractTCPFrame represent a request, a TCP packet (like TCPAsk, TCPConnect...).

This architecture 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. The frame architecture allow a context to have only one message sending queue then a context don't have to know which specific packet he have to send or recieve.

- ## 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 :



ReaderChatOsProtocol

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 because we want differents behaviours for objects implementing the interface FrameReader.

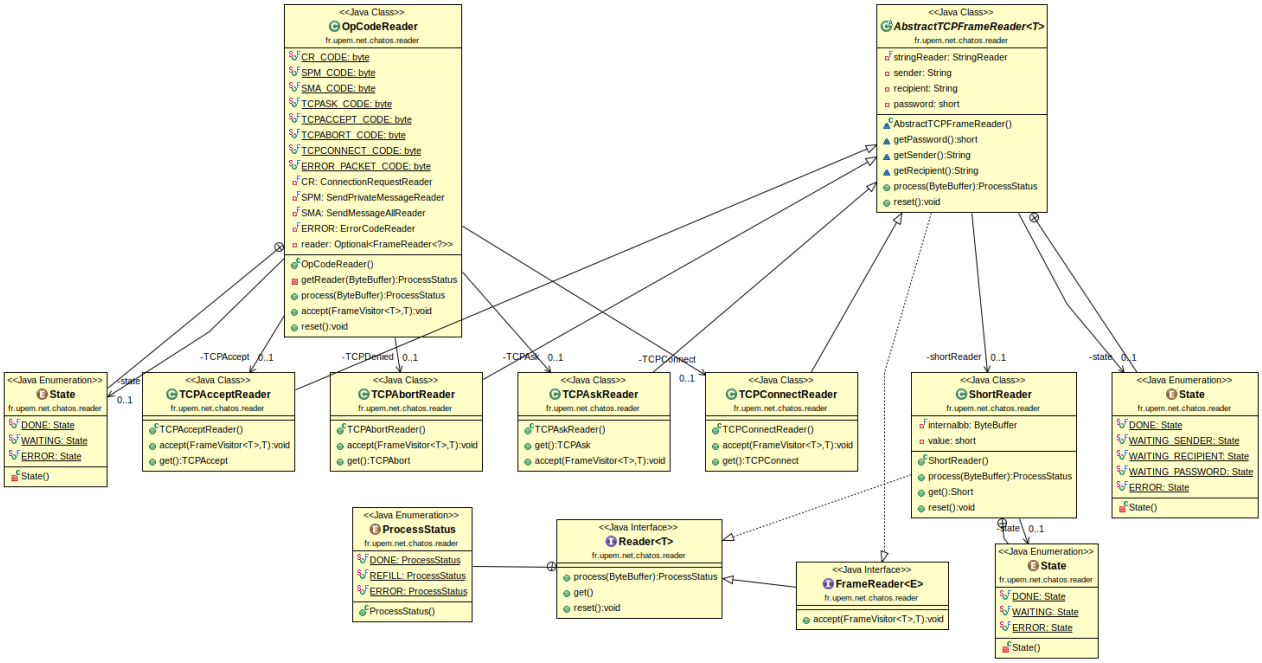
Then all other class turn around the interface Reader which assure that each class implementing it can process a ByteBuffer, get the result and reset the reader. This interface is accompanied with an enumeration ProcessStatus which represent the direct reading status, if it is running, over or in error.

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 new packet (we don’t know yet which packet type it is), so this class have a field for every packet Reader ready to 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 TCPReader UML graph is the following :



TCPReader

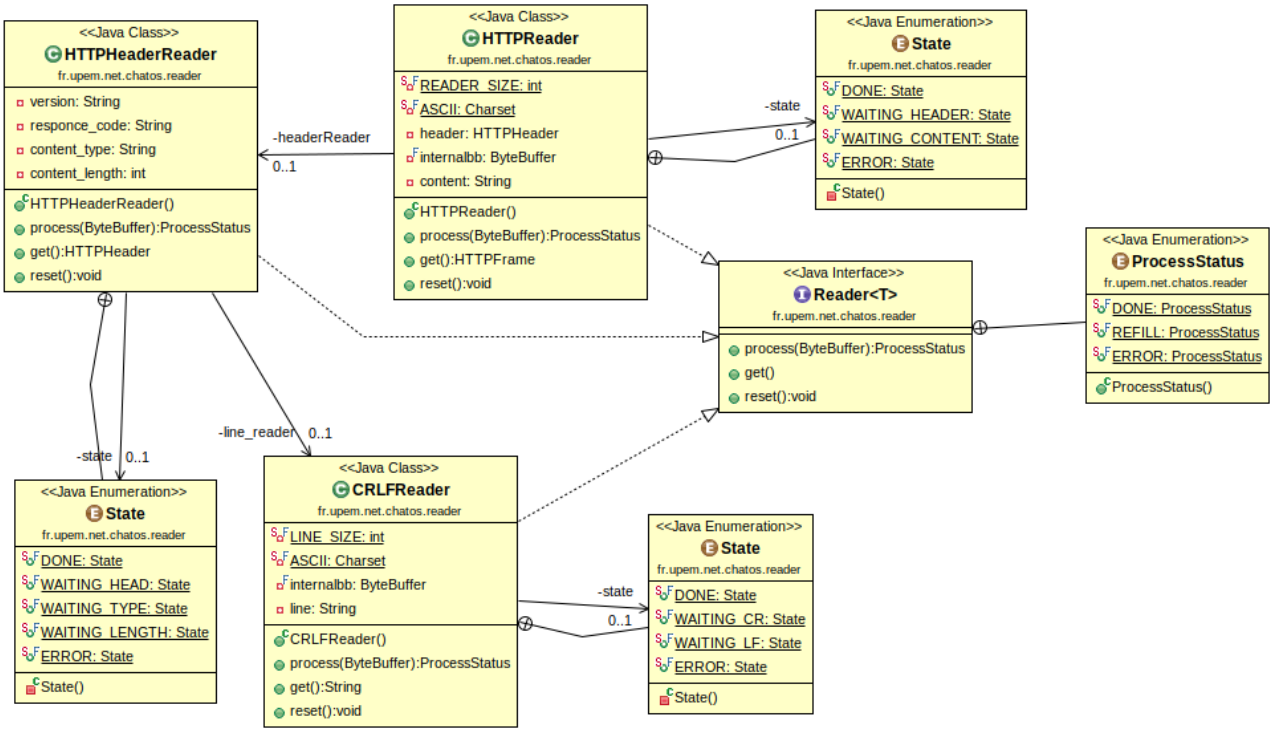
The Reader, the FrameReader and the OpCodeReader are the same that in the previous part.

We put some clean legacy (implementing the interface FrameReader to share the methods) to read TCP frames AbstractTCPFrameReader. This class assure to get on a TCP connexion its password, the sender and 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).

This architecture facilitate the reading of a frame by choosing the right one. The visitor architecture allow a particular and adaptative treatment for each frames and can be easily extended for a new packet type.

The HTTP UMLReader is the following :



HTTPReader

This part is more simple, the Reader interface is no longer a FrameReader as it read HTTP instead of frames.

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 »

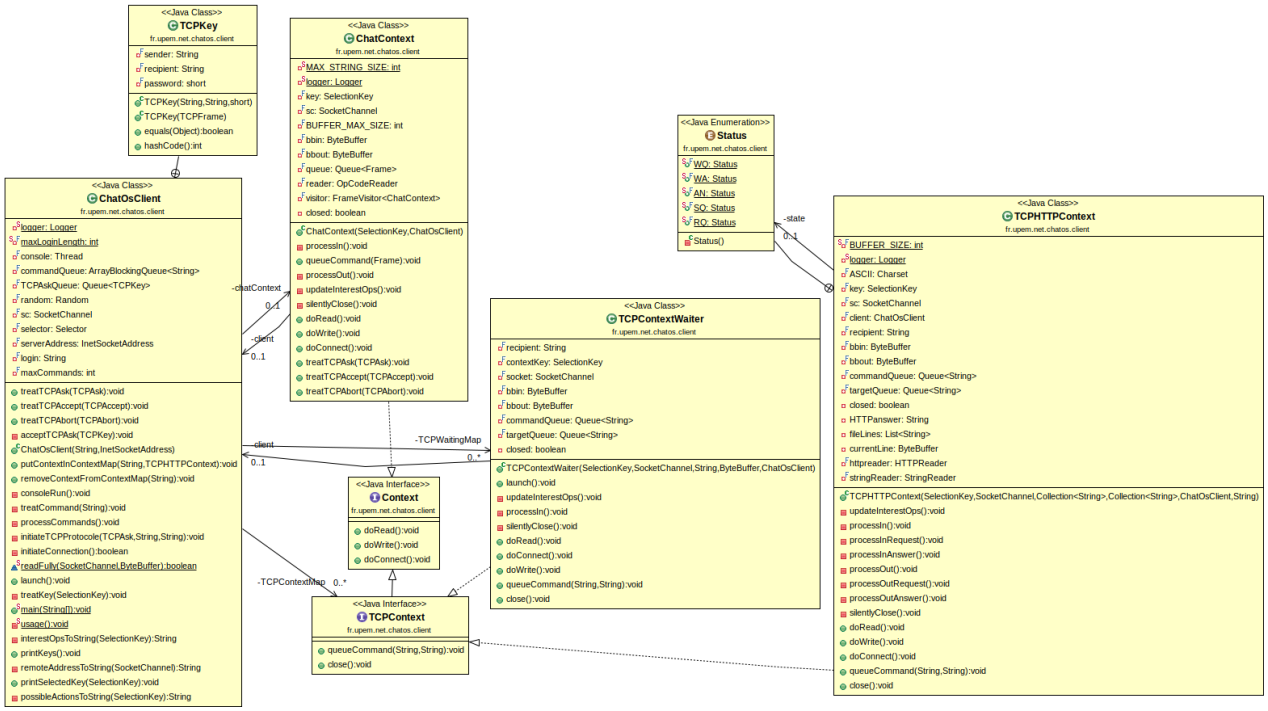
So this header is a HTTP answer in version HTTP/1.0, the answer is not found, the type is other than txt (in ChatOS HTTP content type is text or other), and the length is 0 (because not found).

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

This part is not used by the chat contexts but by the TCP contexts to exchange HTTP responces.

○ 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 which is private), indeed the three contexts was previously internal class of the client and this last had 1000 lines, so we divided it properly.

First the class ChatOsClient can launch a client and connect it with a socket channel by sending a connection request to the specified server address. If it is refused by the server, it close itself, otherwise, it goes in non blocking mode and become a chat client.

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). This class compute differents packet from the server by using a frame visitor like seen previously. To send a frame to the server, it uses a queue of frames.

TCP request are handling differently than classic chat request as it has side effects. When a client send a TCP request or accept a TCP request it create a new context (TCPContextWaiter).

When a client try to initiate a TCP private connection, it creates a new TCPContextWaiter and wait a responce from the server. If this answer is positiv the new Context connect itself to the server and wait for the server to acknowledge this connection. When it’s done, it became a TCPHTTPContext. If the answer is negativ, the context is deleted.

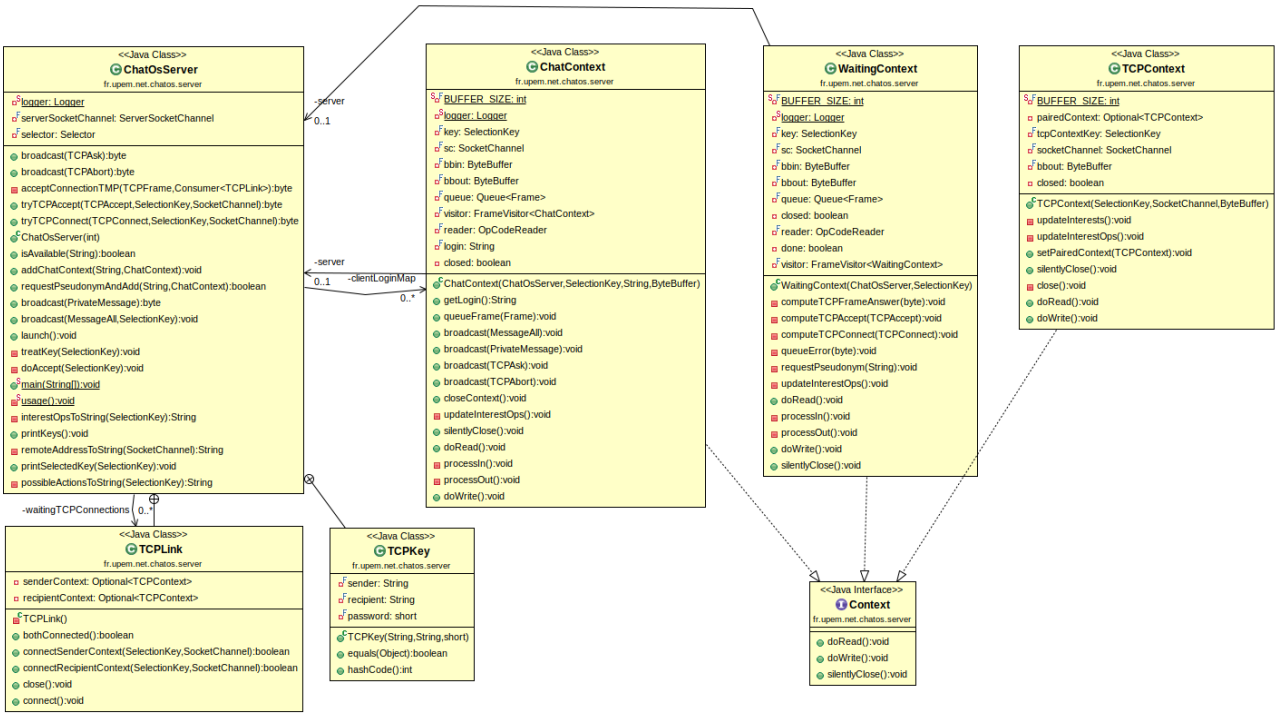
When a client receive a TCP private connection request, it can refuse or accept. If it accept, it create a new TCPContextWaiter and send to the server a TCPAccept packet. Then wait for the server to acknowledge this connection. When it's done, it became a TCPHTTPContext. If it refuse, it send a TCPAbort packet to the server.

The TCPHTTPContext goal is to ask and answer HTTP request using HTTP protocol. As ChatOS only uses GET request, the HTTP request are represented as string. 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.

When a client connect to the server, a WaitingContext is created. It wait a packet to determine the kind of the connection.

If a packet of type connection request is received, the connection is labeled ‘chat’, if this request satisfied conditions (pseudonym availlable) the WaitingContext become a ChatContext and chat operations can be executed, else the context return an error and wait for an other packet. When this context receive a packet, it is broadcasted to all correct recipient.

If a packet of type TCPConnect or TCPAccept is received and if the server is waiting for this connection, the context become a TCPContext. The server follow the RFC for the connection TCP, when both parts are connected the server send at each side that transactions can begin. If a TCPAbort packet of a waiting TCP connection is received then it is relay to each involved parts and the related TCP sockets are also disconnected.

The role of TCPLink is to ensure that both TCP sockets are connected before any transactions and disconnect them if the protocol is aborted.

All others types of packets will be followed by an error and the context will wait for a new packet.

Evolution Since Beta

Client can accept or refuse TCP private connection
HTTP packets and transmission
ContextWaiter for server
Documentation
Jar, Ant and Javadoc

Features

Client can connect to the server	OK
Server can reject client of the same name	OK
Server respect the ChatOS protocol	OK
Client can send public message	OK
Client can send private message	OK
Client can establish a TCP private connexion	OK
Client can refuse a TCP private connexion	OK
Client can send a file with the TCP connexion	OK

In this version of ChatOS, no bug has been encounter.

Improvement : use a better method to read the files for TCP request.

Uncounter difficulties

We struggled to create with ant the jar files and the javadoc, because of the junit path.

We had often bugs, needed a lot of thinking and time to implements features but no real blocking problems.