# PSD 19/20 – Report

Alberto Faria (A79077), João Marques (A81826), Nuno Rei (A81918)

19 January 2020

## 1 INTRODUCTION

This document consists of the report for the practical assignment of the course entitled *Paradigmas de Sistemas Distribuídos*, part of the Integrated Master's Degree in Informatics Engineering at the University of Minho, lectured in the 2019/2020 school year. With this assignment, students were prompted to develop a prototype platform for negotiation between large-scale product manufacturers and importers.

This report is organized as follows. We begin in Section 2 by describing the objectives and system requirements put forward by the assignment. Section 3 then details the architecture of the developed system and justifies its major design decisions and describes several aspects of the system's implementation. Section 4 concludes the report.

## 2 PROBLEM STATEMENT

The platform that's being developed for negotiation, between product manufacturers and importers, must include clients, a front-end server, negotiators, a catalog of entities and active negotiations. The system should support a large number of clients connected in real time, and the clients will always be either product manufacturers or importers.

The manufacturers can manifest their availability to produce a given article while the importers show interest in their articles by making an offer with how much they are willing to pay for them. The front-end server authenticates clients and forwards their requests to one of the available negotiators. It should also inform the catalog of the clients registered in the system. Finally the catalog should provide a *RESTful* interface that includes all the information about the importers, manufacturers and active negotiations.

The assignment requires that the client, negotiator, and catalog programs be developed using the Java programming language, and the frontend server using Erlang. The developed system must leverage ZeroMQ [5] where appropriate and the catalog application must use Dropwizard [2] to expose its RESTful interface. Further, Protocol Buffers [4] shall be used for message encoding and decoding.

Both types of clients need to authenticate themselves on the frontend server with their name and password. Once they are authenticated, if the client is a manufacturer, it can state its availability to produce an article by stating its name, the maximum and minimum quantity, the negotiation period, i.e, the period of time that the importers can make offers. On the other hand, if the client is an importer, it can make offers for products posted by the manufacturers by stating the name of the manufacturer, the product name, quantity and the price he is willing to pay. Importers may also *subscribe* to a certain manufacturer to receive a *notification* every time he announces an article.

Negotiation works as follows: a **negotiator** receives, from the frontend server, the availability to produce an article from a manufacturer as well as offers for this article by the importers. After the period of time specified by the manufacturer the offer is canceled if the minimum quantity of offers is not met, otherwise the offers with the highest price are approved.

## 3 DESIGN AND IMPLEMENTATION

In this section, we describe the architecture and functioning of the developed system, identifying and justifying major design decisions, and also present several aspects of the system's implementation.

**System overview.** The system is composed of multiple clients, a set of negotiators, a catalog and a front-end server. Clients can exist in high numbers at any given time and they can be either importers or manufacturers. Negotiators exist as a fixed number and are pre-initialized and known to the front-end server. The front-end server is responsible for allowing communication between clients and an arbiter. Information about registered clients and active negotiations are stored and can be observed on the catalog, this information is obtained from the front-end server and from the arbiters.

Manufacturers announce, to the front-end server, their interest on producing an article and give a period of time to collect offers for that article. The front-end server then forwards that request to one of the available arbiters. Importers show their interest on the products announced by sending an offer, to the front-end server, and give a quantity and price to pay. The front-end server also forwards that request to the arbiter responsible for the negotiation of that product.

The arbiter will report to the front-end server the outcome of a negotiation after the period time specified expires, so that the front-end server can inform the manufacturer and all importers that made an offered of the negotiation outcome. After receiving a product announcement the arbiter also informs the catalog of the new active negotiation.

**Data serialization.** Clients, catalog and arbiters use *Java SE 11*, while the front-end server is developed using *Erlang/OTP 22.2*. This language heterogeneity raises the need for a consensus regarding the communication protocol. This is achieved with google's *Protocol Buffers* [4]. which is a binary serialization format thus more compact and computationally more efficient. Every message exchanged in the system are specified in the a *.proto* file shared by both languages.

**Communication between clients and the frontend server.** Communication between clients and the frontend server is implemented using bare TCP sockets. This is adequate as (i) clients do not need to communicate with each other, (2) always know which server to contact (there is only a single frontend server), and (3) follow a session-oriented protocol with the frontend server (clients must authenticate before performing any actions). For these same reasons, the use of messaing middleware with higher level primitives is not advantageous.

As protocol buffers do not delimit messages, however, plain TCP sockets require extra information to be transmitted alongside the messages. We simply prefix every message with a 4-byte, big-endian encoding of the number of bytes in the message. This has a straightforward Java implementation and can be handled transparently in Erlang using the `{packet, 4}` socket option.

**Communication between arbiters and the frontend server.** The frontend server delegates the responsability of managing each negotiation to a given arbiter. When a negotiation begins, it is *assigned to the arbiter with the least ongoing negotiations*. The frontend server must also forward offers made by clients to the appropriate arbiter. This is accomplished through a publish-subscribe messaing pattern using ZeroMQ, whereby the frontend server publishes notices of client offers (with a topic that identifies the manufacturer and product) and the interested arbiter receives those offers.

**Catalog state.** At any point, users of the catalog's interface may want to observe active negotiations, or the registered manufacturers and importers. The catalog must remain updated, and so the strategy is to use a *publish-subscribe* pattern with *ZeroMQ* [5] (*JeroMQ* [3] for the catalog and arbiters and *chumak* [1] for the frontend server).

Whenever a new client (importer or manufacturer) account is registered, the frontend server informs the catalog of this fact by publishing a message with the corresponding topic and the username of the new client. The catalog is a subscriber of messages with these topics, and so receives it and uses it to update its internal copy of the system state. Whenever a negotiation is created or ends, the respective arbiter also inform the catalog of the fact by publishing messages with the appropriate topic to the messaging middleware.

The catalog thus maintains an up-to-date copy of the relevant system state, and uses it to answer requests made to its RESTful interface. Note that this "push-based" design is most appropriate for workload in which the catalog is frequently queried for information, as answering these queries does not require any further communication with other system components.

## 4 CONCLUSION

In this document, we presented the development of a prototype platform for negotiation between large-scale product manufacturers and importers. The objectives and requirements of the assignment to which this report pertains were first identified, and the design of the developed system was then detailed along with several aspects of its implementation.

A production system should address several more issues, such as persistence of state and replication of services besides the already implemented support for several arbiters, which targets load-balancing. We leave these matters as future work.

## REFERENCES

[1] 2019. *chumak – Pure Erlang implementation of ZeroMQ Message Transport Protocol.* Retrieved 14 January 2020 from https://github.com/zeromq/chumak

[2] 2019. *Dropwizard.* Retrieved 14 January 2020 from https://www.dropwizard.io/

[3] 2019. *JeroMQ – Pure Java implementation of libzmq.* Retrieved 14 January 2020 from https://github.com/zeromq/jeromq

[4] 2020. *Protocol Buffers.* Retrieved 14 January 2020 from https://developers.google.com/protocol-buffers

[5] 2020. *ZeroMQ – An open-source universal messaging library.* Retrieved 14 January 2020 from https://zeromq.org/