

Simulation of a Scientific Computation Platform With a Focus on Quality Attributes

Filipe Pires [85122], João Alegria [85048]

Software Architecture

Department of Electronics, Telecommunications and Informatics

University of Aveiro

May 22, 2020

Introduction

This report aims to describe the work developed for the third and final assignment of the course of 'Software Architecture', focused on a platform that accepts and processes computational services requested by the scientific community.

The aim of the assignment was to design and develop a software architecture relying on four of the most relevant quality attributes: performance, availability, scalability and usability. Playing the role of software architects, we came up with the solution for an infrastructure for our stakeholder. The platform here presented is capable of deploying a cluster of servers monitored by a tactic entity and whose requests from clients are distributed by a load balancer. Although the cluster is locally simulated, the configuration is done so that it is possible to deploy in a distributed environment.

So in this report we present the architecture of our solution, justifying design decisions according to what we learned and found to be most suitable for our use case. We also mention how the work was distributed amongst the authors.

All code developed is publicly accessible in our GitHub repository:

<https://github.com/FilipePires98/AS/>

1 Scientific Computation and Custom Systems

Applied computer science and mathematics often use advanced computing capabilities to understand and solve complex problems. In practical use, computational science is typically the application of computer simulation and other forms of computation to tackle problems in various scientific disciplines. Scientists and engineers develop computer programs that model systems being studied and run these programs with various sets of input parameters. In some cases, these models require massive amounts of calculations and are often executed on supercomputers or distributed computing platforms.

This project does not describe one of these models, rather it is about the development of an infrastructure capable of supporting such computationally demanding tasks. The adopted strategy was of coordinating a cluster of processing servers. The following tactics were selected to be implemented:

- Computation Replicas - requests must be fairly distributed amongst the servers.
- Concurrency - each request runs on its own thread in a server.
- Redundancy - in case a server goes down, requests should be reallocated.
- Monitor - everything must be supervised, from the cluster's status, to the clients' identification and request treatment.
- Horizontal scalability - new servers can be deployed whenever necessary.

1.1 The Scenario

In order to test the application of the infrastructure on a scientific problem, an example scenario was adopted: the calculation of the mathematical constant π (pi). Being an irrational number, π cannot be expressed as a common fraction and its decimal representation never ends and never settles into a permanently repeating pattern. Nevertheless, in the 21st century, mathematicians and computer scientists have pursued new approaches that, when combined with increasing computational power, extended the decimal representation of π to many trillions of digits (1). The primary motivation for these computations is as a test case to develop efficient algorithms to calculate numeric series, as well as the quest to break records.

The most commonly adopted forms of calculating π are the iterative algorithms (2). In order to only focus on the infrastructure's implementation, while simulating the actual implementation of the iterative algorithms, a control variable corresponding to the number of iterations to be used on the calculation was introduced to regulate servers' response times.

1.2 The Messages

One of the fundamental constraints applied to our implementation was related to communications. Each individual server is launched as an independent process. This is also true for each individual client. The orchestrator runs in an independent process as well, serving as both a load balancer and a tactic monitor - although these two are built as separate entities. So in order for entities to communicate with each other, the TCP/IP socket technology was made a requirement.

For simplicity, only two service message types were defined:

- Request: | clientID | requestID | 01 | # iterations |
- Reply: | serverID | clientID | requestID | 02 | # iterations | pi |

In them are the following parameters: `clientID`, a positive integer that uniquely identifies a client machine; `serverID`, a positive integer that uniquely identifies each physical server in the cluster; `requestID`, a positive integer that uniquely identifies a pi calculation request (computed as $1000 \times \text{client id} + \text{increment}$); 01 and 02, the request and reply codes; # iterations, the chosen value for the previously mentioned control variable used to simulate the number of iterations for the computation of pi (each iteration / cycle corresponds to 1 second); `pi`, the computed value of π .

A few additional message types were added during development for management purposes:

- Heartbeat:
.....

2 System Architecture

2.1 Components

2.1.1 Client

2.1.2 Server

2.1.3 Load Balancer

This is one of the most important components of our architecture. It's existence started by being a project requisite, but we quickly understood and concluded that in fact it was a huge asset in our solution. This component proved to be quite paramount when handling the quality attributes we had to guarantee in this project. This component was mainly created to handle availability and scalability, reasons we will further detail in section 3. This section will describe how this component work.

It's main purpose, as the name implies, is to distribute in a balanced way the incoming π calculation requests across all registered servers. When one or more calculation requests are sent to the Load Balancer, for each request the least occupied server is requested from the Tactic Manager, enabling a balanced distribution; after knowing to which server the message should be redirected to, information such as the ip and port from the client requesting the service is also requested from the Tactic Manager, allowing the server to respond to the client directly, as stated by the project guide; finally, the message is redirected to the server with the client information added in a pre-defined fashion.

Additionally, this and only this component is the one responsible for interacting with the system clients. This means that not only the already mentioned π calculation requests but also some additional control messages that our team created need to be handled by the Load Balancer. These additional messages are related to the creation and destruction of a client. As already mentioned in section 2.1, we concluded and decided that the most logical approach was to enable each server to choose and define its port, due to physical restriction; this means that when a client is created, the server side of the system does not know its ip nor its port, information that needs to be registered. Following that thought process, we created both a creation message, containing the client's ip and port and a deletion message containing only the client's id that was given by the Tactic Manager. Those messages in the creation or deletion of the client are sent to the Load Balancer, the only entity the client knows about in the system's server side.

When receiving the client control messages, the Balancer acknowledges it as received it and redirects the message to the Tactic Manager, that will process and store the received information, and in the case it is a client creation, it will send to that client his just created internal id.

2.1.4 Tactic Manager

This component can be considered as the brain of the cluster, since it should have the knowledge about every registered server, every registered client, the state and who is processing every message at anytime. This is necessary, for example, as stated in section 2.1.3, to enable the Load Balancer to distribute in a balanced fashion the incoming calculation requests since the Manager knows the occupation of each server at anytime.

One of its responsibilities is to be the entity, and the only one, with whom the servers talk too. For that reason, similarly to the control messages developed for the clients refereed in section 2.1.3, we also needed to create control messages for the creation and destruction of each server, being sent in the appropriate time to the Task Manager, that will process the messages and store the information, also sending the created id in case of a server creation. Additionally, all servers when receiving or finishing to process a calculation request should notify this managing entity of its status change.

Another responsibility of this component is the executions and possible consequent actions related to the health-check of all registered servers. This functionally was concluded to be paramount when assuring our quality attributes, namely Scalability, Performance, Availability and Usability. The flow of this process goes as follows: when the Tactic Manager is created, the Health-checking subprocess is initiated in parallel, and from then on, it will send a health-check to all registered servers on a time interval previously established, in our scenario it was defined at 30 seconds. In the case one server doesn't respond to the health-check, it is considered dead, and for that reason it is deleted from the internal registries and any request assigned to that server (which was previously registered by the server when alive) will be equally distributed by all remaining servers by sending them to the Load Balancer which will execute that task.

2.2 Auxiliary Components

2.3 User Interface

3 Architecture Constraints

3.1 Quality Attributes Assurance

3.2 User Cases Compliance

4 Additional Remarks

4.1 Documentation

Our attitude towards the developed code was to ensure it could be applied to other similar scenarios and reused in systems intended to be deployed in real scenarios. With this in mind, we took great care with regards to code readability. By maintaining a code style equal throughout the project and defining intuitive and self-explaining variable and method names, we made the code easy to understand by someone already contextualized with Kafka.

Nevertheless, we wanted to make sure this was also true to someone looking at our project for the first time, so we resorted to the well-known Javadoc (?) tool to manage all code documentation. Comments were also added in key points throughout the code, including the scripts.

4.2 Assignment Contributions

As the entire development phase took place in a time where on-site cooperation was not possible, we resorted to online communication platforms to debate decisions and discuss difficulties. Team scheduling allowed us to work on the project simultaneously, so no member suffered from unbalanced workloads. The dimension of the project did not appeal to the usage of repository pull requests and other synchronization tools. However, each small solution was verified and agreed by both team members.

Having said this, it is difficult to isolate what each member actually implemented, as the influence of both is present in all components. Nevertheless, one might say that each had stronger responsibilities on a set of project aspects: Filipe took care of the execution of the individual Java processes and of the Shell scripts, while João developed the Kafka-related classes such as Consumer, Producer and EntityAction; Filipe developed the Python script for generation of `CAR.TXT`, while João developed the Shell scripts for Kafka initialization and deletion; each implemented 2 entities and each wrote a portion of this report; Filipe made sure everything was coherent throughout the report and the code documentation, while João solved the most critical issues regarding the configuration of the topics. In terms of work percentage, we believe it was about 50% for each student.

Conclusions

References

1. *Pi in the sky: Calculating a record-breaking 31.4 trillion digits of Archimedes' constant on Google Cloud*. Taken from web.archive.org. Retrieved May 2020.
2. Arndt, Jorg & Haanel, Christoph (2006). *Pi Unleashed*, in Springer-Verlag. ISBN 978-3-540-66572-4. Retrieved May 2020. English translation by Catriona and David Lischka.