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 26, 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 $\times$ `clientId` + 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. In addition, between internal server components we found the need to append to the request message some metadata to help us in implementing some mandatory features. We then extended that message format and ended up with `request-<clientHost>-<clientPort>-<request>`, a self-explanatory format but the parameters will be detailed below.

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

- Heartbeat/HealthCheck: `healthcheck`
- New Client: `newClient-<clientHost>-<clientPort>`
- Client Down: `clientDown-<clientId>`
- New Server: `newServer-<serverHost>-<serverPort>`
- Client Down: `serverDown-<serverId>`
- New Request: `newRequest-<serverId>-<request>`
- Request Processed: `processedRequest-<serverId>-<request>`

As shown, some of these messages also need parameters such as: `clientHost`, a string containing the host name of the client machine; `clientPort`, a integer containing the port where the client process is running; `clientId`, an integer that uniquely identifies the client; `serverHost`, a string containing the host name of the server machine; `serverPort`, a integer containing the port where the server process is running; `serverId`, an integer that uniquely identifies the server and lastly `request`, a string containing the request message that triggered that event.

2 System Architecture

The architecture we were guided to was a junction of a client/server and a master/slave architecture. It is partly a client/server because in the first place, there is a client to our system and a server, that from the client's perspective seems to be only one entity but in fact it is composed of several, each with its specific responsibility. Those internal server entities are independent of one another, and in case they need to communicate they do so in a client/server manner, making the necessary requests and awaiting for the response of the entity in question.

The master/slave aspect of our architecture is due to the fact that the π calculation infrastructure is comprised of a main entity and a set of calculation servers; that main entity that in fact is divided in two(Load Balancer and Tactic Manager) is the one that interacts with the clients and then assigns work to the calculation servers that are awaiting.

All the communications in this system are done through WebSockets like the project guide proposed. A further description of each component is given below.

2.1 Components

2.1.1 Client

This component is the one that represents the client in our system. This entity is the one that makes the calculation requests by sending them to the Load Balancer and then awaits for the response to arrive. The client can make the number of request he wishes and he doesn't need to wait for the response of a calculation request to order another.

The client need to instantiate a WebSocket server so that the calculation request responses can be sent to him in a asynchronous fashion. After some deliberation our group concluded that the best approach was fro the client to choose the port it would use, since there could be constraints the machine the process is running and it wasn't viable for the client to receive the port indication from the server that doesn't know anything about the ports already in use on the client machine. For that reason the client, in initiating, as to send some control messages to notify the main server that it was created and to indicate it's websocket host ip and port, in response it will receive it's internal server id; when the client is stopped it will also notify the main server of that event.

2.1.2 Calculation Server

Responsible for making the π calculation themselves, this entity awaits for a calculation request to arrive, processes it and then responds directly to the client that triggered it. The direct response to the clients is possible because the request first goes through the Load Balancer, that

with the help of Tactic Manager adds the client metadata necessary for the calculation server to directly reply to the client. All the requests processing is parallelized.

Similarly to the clients, the servers also need a WebSocket server to receive the calculation requests, so they also need to choose the port it will be active in. The same physical constraints apply, so the servers when created need to identify the main server of that event and attach the necessary metadata, such as the host ip and port the websocket is running in; when a server is closed, if possible, it should also notify the main server of that event.

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 not 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 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 form 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

In addition to all those main component, there are some auxiliary ones that need to be mentioned since them role is quite important.

The first ones are the SocketServer and SocketClient, both are wrappers for a WebSocket Server and a WebSocket Client, respectively. This enabled us to use our wrappers when in need of a WebSocket Server or Client without worrying of the internal working, allowing us to focus on the logic of the problem. The only consideration we needed to make was that every entity tht used the SocketServer needed to pass a MessageProcessor interface instance, take would supply all the logic when the server received a message. Additionally, some refactoring was made to

these classes in relation to the one delivered in our first project, namely in the SocketServer. We added a AttendUser internal class to the SocketServer, which handled in a parallel way all the connections made to the WebSocket server.

Another important component is the LoadDistributer, a class internal to the LoadBalancer that is responsible for the actual distribution of the incoming requests. Every request is processed in a parallel way. Lastly, there's the HealthChecker, internal to the TacticManager and as the name implies, it is the class that as the duty to health check all registered servers and take action in case some of them are not alive.

2.3 User Interface

3 Architecture Constraints

3.1 Quality Attributes Assurance

3.1.1 Performance

As detailed by the project guidelines, this quality attribute focuses in main points: the computational replicas, meaning that there should exist more than one entity able to calculate π, translating to a necessity of distributing in a balanced fashion the request between the available computational nodes so that the response time can decrease; the other focus was on the mandatory concurrency, which by design would enable our solution to be more efficient by processing each request in its independent thread in a computational server, also specifying that in the case some synchronization needed to be made, it should use ReentrantLocks.

With that in mind, we developed several mechanisms to ensure that we were following the guidelines and the solution was the best we could produce. In relation to the aspect of computational servers replication, in the case there are more than one computational server active at a time, we made possible the balanced distribution across using the already mentioned Load Balancer, section 2.1.3. The flow of data was already explained in that section and the internal worker thread, Load Distributer, was also already presented in section 2.2. To illustrate the distribution itself, we present a code snippet of the Load Distributer normal distribution process:

```
//LoadDistributor
String[] server=myClient.send("leastOccupiedServer")
                    .split("-");
if(server[1].equals("none")){...}
else{
  SocketClient clientSocket=new SocketClient(...);
  try {
    for(String msg:messages){
      String[] tmp=msg.replaceAll("\\s+","").split("\\|");
      String[] processed=Arrays.copyOfRange(tmp,1,tmp.length);
      String[] client=myClient.send("clientInfo-"+
                              processed[0]).split("-");
      clientSocket.send("request-"+
                        client[2]+"-"+
                        client[3]+"-"+
                        msg);
    }
  } catch (IOException ex) {
    myClient.send("serverDown-"+server[1]);
  }
  clientSocket.close();
}
```

In relation to the concurrency aspect and the synchronization, in this project we made sure that all clients were handled by separate thread and every request made by each client also treated with its independent thread. The concurrent synchronization we found that it was only suitable and need in the Tactic Manager; all servers are independent of each other, the Load Balancer is a proxy for all the available servers, but the Tactic Manager need to receive and store crucial cluster information that needs to be consistent and reliable. For that reason, internally to the Tactic Manager, we implemented a monitor using the indicated *ReentrantLock* to assure a mutually exclusive access to that information between all the thread created by each message arriving at the Tactic Manager. To illustrate that we present a snippet of our SocketServer wrapper showing that each client is handled by an independent thread, a snippet of the computational server showing that each incoming request is handled by an independent thread and last a snippet of the Tactic Manager internal monitor.

```
//SocketServer
socket = new ServerSocket(this.port);
mp.setSocketStatus(1);
while(continueRunning){
  Socket inSocket = socket.accept();
  Thread t=new Thread(new AttendClient(inSocket));
  t.start();
}
socket.close();
```

```
//Server
String[] processedMessage = message.split("-");
switch(processedMessage[0]){
  ...
  case "request":
    SocketClient manager = new SocketClient(tmHost,tmPort);
    try {
      manager.send("newRequest-"+this.id+"-"+processedMessage[3]);
      manager.close();
      processingRequests.add(processedMessage[3]);updateProcessing();
      PiCalculation request = new PiCalculation(processedMessage[1],
                                Integer.valueOf(processedMessage[2]),
                                processedMessage[3]);
      Thread requestProcessing = new Thread(request);
      requestProcessing.start();
    } catch (IOException ex) {
      Logger.getLogger(Server.class.getName()).log(Level.SEVERE,
          null, ex);
    }
    break;
}
```

```
//ClusterInfo(Monitor)
private final ReentrantLock rl;
...
public void addServer(String host, int port){
  rl.lock();
  int id=0;
  while(serverInfo.keySet().contains(id)){id++;}
  ServerInfo si = new ServerInfo(id, host, port);
  serverInfo.put(id, si);
  SocketClient client=new SocketClient(host, port);
  try {client.send("serverId-"+id);}
  catch (IOException ex) {removeServer(id);}
  client.close();updateServers();
  rl.unlock();
}
```

3.1.2 Availability

On availability, the guidelines outlined once again two main objectives, the first was to provide redundancy, meaning that on the event of a server going down, the system should react and reallocate in a balanced way every request that was being processed by that server; the second objective was the implementation of a monitor entity, enabling a supervision of the entire cluster status, active servers, registered clients and requests status. To assure the availability of the calculation servers and also indicated by the guidelines, a healthcheck feature was added, verifying that every registered server is in fact alive in a scheduled fashion.

In our solution, the suggested redundancy was triggered by the unavailability of a server. This event could be assessed in two stages, either when the Load Distributer is sending the calculation request to one of the available servers and a acknowledge message is not sent as response; in this case we assume that that server is down and should be considered dead and removed from the internal registry. Another option, which takes advantage of a monitoring feature, is in the case of a healthcheck being negative, meaning that the server was unable to respond, most certainly being dead; in this case, similarly to the previous option, we consider the server dead, remove it from the internal registry and perform the reallocation of any request current being processed by that server. We will present a little code snippet for the first option but the healthcheck's one will be illustrated after describing the second objective. Referring to snippet 3.1.1, when a exception is raised means that an acknowledge wasn't received, so for that reason, the code for handling the exception is the one to reallocate the request(s).

Monitoring is the main purpose of the Tactic Manager as mentioned in section 2.1.4. Using all the management messages mentioned in section 1.2, both servers and clients send metadata information directly or indirectly to the Tactic Manager so that an updated and reliable state

of the cluster is always present. The usage of the already discussed monitor helps to keep this information consistent and error free. Another feature of the Tactic Manager that was also developed due to availability was the health checking process, which assure that all registered servers are alive and available to receive calculation requests. The flow of this process was already detailed in section 2.1.4, but a code snippet will be presented for a better understanding.

```
//HealthCheck
while(true){
  try {Thread.sleep(30000);}
  catch (InterruptedException ex) {
    Logger.getLogger(TacticManager.class.getName()).log(Level.SEVERE,
       null, ex);
  }
  for(ServerInfo si : serverInfo.values()){
    try {
      SocketClient client=new SocketClient(si.getHost(),
         si.getPort());
      client.send("healthcheck");client.close();
    } catch (IOException ex) {removeServer(si.getId());}
  }
}
```

3.1.3 Scalability

3.1.4 Usability

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 & Haenel, Christoph (2006). *Pi Unleashed*, in Springer-Verlag. ISBN 978-3-540-66572-4. Retrieved May 2020. English translation by Catriona and David Lischka.