A Distributed System Immune to Byzantine Failures
M. M. Fernandes
Instituto Superior Técnico

Abstract

Failures in data processing are impossible to predict, and many times to even detect, thus provoking errors in results, that apparently are correct. To skip these situations and guarantee that the data is not corrupted by software or hardware faults, or even by faults introduced on purpose, replication, comparison and encryption are very useful processes.

The immunity against errors can be very important in the remote execution situation. There are tools, like Condor, which manage the idle times of network machines processing, but they don’t guarantee that faults are not introduced in the communications or even in executions.

This work’s goal is to fill this gap in tools such as Condor when the occurrence of these errors is critical. To achieve this goal there was also implemented security procedures and a replication algorithm.

This work is composed by one process however this process has two distinct functions. The main function not only plays the role of Administrator of the system, but can also be a Client that executes or submits to execution, programs that require a guarantee of non corruption of data. The function is simpler and it’s responsible for the acquisition of files to send to remote locations.

The communications between processes and machines is relevant in this work, because the success of the proposed goals like the immunity to errors arises from the cooperation between them.

Keywords: Byzantine Failures, Distributed Systems, Voltan Machine, Cryptography, Replication, Transparency

1. Introduction

The goal of this work is the study and implementation of a fail safe distributed system that is immune to byzantine failures. This work is inspired by a tool developed by the Wisconsin-Madison University named Condor. This tool can send jobs to execute in remote machines, and can manage these communications and executions in a distributed and transparent way. However Condor does not guarantee that failures are not introduced in communications and in the jobs executions. This work’s goal is to fill this gap in this kind of tools when and when the occurrence of these faults is critical. In this work, the replication is used as a way of avoiding byzantine failures and therefore reduces the influence of this type of errors in the final results of the executions.

2. The Voltan Algorithm

The needed replication is achieved using the Voltan algorithm. The Voltan algorithm is also known as Leader-Follower algorithm.

The main base of this algorithm is that there are two types of processes, the Leader process, and the Follower process. Both receive the same messages to process, but they communicate between them, to guarantee that they really have the same messages. Then, they both process those messages. In the case of this work, this processing is the execution of a program.

When the processing is completed, they exchange messages with the processing results and compare them to see if there are any differences. Finally they both send the results forward, to the consumer process.

A more redundant system is the use of two pairs of Leader-Follower processes. Then these pairs communicate with each other in order to guarantee the consistency of the results. It is assumed in this algorithm that the communications between processes are safe, i.e. security mechanisms are used, like hash algorithms and cryptography.

Therefore some security criteria were implemented such as message digests that generates a short hash message, and digital signatures.
3. Byzantine Failures

Byzantine failures are arbitrary failures which can occur during the execution of an algorithm in a distributed system. The faults are also known as crash failures or send and omission failures. When a byzantine failure takes place, the response of the system can be unpredictable. In a worse case, it can happen that a server can be cooperating with other to make these failures take place. [1]

4. Fault tolerance in Distributed Systems

In a distributed system, the linked machines will cooperate in order to fill a common goal. One definition of distributed system is given by Tanenbaum, A.S. e Steen M.V. in the book "Distributed Systems, Principles and Paradigms", and it states that: A distributed system is a collection of independent machines that seems to be one simples and coherent machine to its users. In this definition are two aspects: one is the hardware; the machines are autonomous; and the other is the software; the users think that they are using only one single system. [1,7]

4.1 Failure Models

A system in failure is not providing properly, the services it is supposed to provide. If we consider that a distributed system is a collection of servers that communicate among them and with clients, if a system like this is failing it means that there are maybe errors in the server, or in the communication media or both. This means that there are multiple sources for errors. A crash failure happens if a server stops functioning but it was working correctly until it stopped. An omission failure happens when a server stops responding to the requests. This may happen due to lost of messages in the communication channel or maybe the server received the message correctly but it missed sending the response. Timing failures may occur if a given time frame for the response is not fulfilled. A response failure is a failure that happens when the server’s response is simply incorrect. The most serious failures are the arbitrary failures also known as Byzantine failures. When a server exhibits failures like these it is sending wrong responses but those responses are not recognized as such. Worst yet is that a server maybe maliciously working with others to give these wrong responses. There are some cases when the server knows that it's about to fail, so in this case it will warn the others of that situation, and the system will be prepared for this failure. Systems that have this property are called fail-silent. For the other hand there are cases of wrong responses that are detected as such by the other servers and therefore discarded. This means that a server may fail and the system will not be compromised by it. The systems with this property are called fail-safe. [1]

5. Implementation of the Byzantine Fault tolerant Distributed System

The structure of the developed system is divided into Administrator machines, and client machines. The Administrator is the manager of the network; it has a list of machines connected to the network as well as their information. The client can be a Submitting machine; a machine that submits jobs to the system; or an Execution Machine; this machine will execute the job, compare results and retrieve the comparing conclusions to the Submitting Machine. Also in this system is implemented a replication algorithm and security in the communications.

5.1 Voltan Algorithm Implementation

In this work, the original Voltan algorithm was modified in the following way: the submitting machine sends the files to execute to two different machines, called Leader 1 and Leader 2. In those machines, that we can call execution machines, for each Leader there is a follower. Only the two Leaders communicate with each other. They compare the receiving initial files and the final results, before they send it to the submitting machine. The communication between the Leaders and the respective Follower pair, is done in this way: The Leader sends the initial file to the follower, and executes the program, also, the follower executes it, then the Follower sends the results to
the Leader, and then the Leader compares the both results. In the end, the submitting machine also compares the two results that have received. The change in the algorithm was made to reduce the number of comparisons to enhance the performance of the system. Since that this system implements multi-threading in execution requests and that the communications are made using TCP, along with the security measures, this assures that the data is well transmitted. Therefore each job execution request is made in a separate thread, avoiding the message queues management.

Non deterministic system calls, like local clock calls, or random numbers generation, inside the executing code, could cause state difference, and hence error signaling. That could happen because calls like that can produce, and almost certainly will, different values to work with, and therefore different final results, when they are not wrong. Solutions to this problem were not discussed in this work, but one solution could be to submit source code to execution, forcing the code to be compiled later, and managing these system calls, like generating the necessary values outside the code and forcing them into it. Solutions for this problem have been considered and implemented before. Like its said in the paper by D. Black, C. Low and S.K. Shrivastava in [4]. There are two sources for non-determinisms; a synchronous and an asynchronous. “(...) for the previously mentioned synchronous result non-deterministic calculation, the only requirement is that the application replicas both receive the same value. The synchronous return nature of the call ensures that both replicas will receive that value at the same point within the execution of the application. (...)”. [4] Related to the asynchronous non-determinism problem “(...) the requirement here is that both the replicas must receive any asynchronous signal (e.g., a time-out exception) at exactly the same point within the execution of their respective computation. The approach taken for dealing with time-based non-determinism is to convert any asynchronous event into a message and to ensure that the replicas both select the same message at the same point during their execution.” [4]

6. System Implementation

The submission of jobs to the system proceeds as follows: the Submitting Machine asks for machines to execute to the Administrator; then the Administrator searches for two available machines and send to the Submitting Machine the information about these machines; then, the SM sends the jobs to the Execution Machines; the Execution Machines executes the jobs, and sends back to the Submitting Machine the results of the execution, finally, the Submitting Machine compares the results in order to find if there are any discrepancies.

6.1 Machine communications

In a distributed system, the communications between machines, and its protocols, have an important role. In fact, the characteristics of data transmission between machines can have different properties, like the time of transmission, or the guarantee of delivery. Also the remote procedure calls are often used in distributed systems, and it is used in this work in the form of remote method invocations.

6.1.1 UDP communications

The use of UDP communications is important to accelerate transmissions that don’t require the guarantee of delivery. Some of those communications are the registry of Clients in the Administrator. The protocol used for this registry is simple, the Client machine sends a “CON” message to the Administrator, and then the Administrator responds with an “OK” message. The receiving socket has a timeout, if it do not receive a message in 5 seconds, the timeout expires, and it re-sends the “CON” message 5 more times. If even then it cannot connect to the Administrator the Client gives up.

The turning off of a Client or the Administrator is also done by transmitting data over an UDP communication. If the machine that is turning off is a Client, it just sends a message “OFF” to the Administrator that receives that message and removes the machine that sent it, from the list of machines it has. If the machine quitting the system is the Administrator, then it sends a “ADMOFF” message to all machines telling them that there is no point in submitting jobs to the system, and also to its own thread Console to exit the execution.

There are also three more UDP communications. One is for retrieving a list of machines in the system
that can be a local communication, or remote. Another is to communicate the TCP ports where the Leader 1 is listening for the Leader 2 connect. And the last one is to grant a communication platform between the leader and the correspondent follower, in order to exchange data.

6.1.2 TCP communications

There are two main TCP communications in this system. The first one is the interaction between RecebeThread and Bizant. That is, when RecebeThread initiates its execution, it reads the data from the hard disk, and initiates a TCP connection with Bizant. Then it transfers data to Bizant, for the submission to remote machines. The threads involved in this communication are, for the RecebeThread side, the thread EnviaFilesThread; this thread sends the data to the Bizant side thread TrataRecebe, where the submission to remote machines is done.

Another TCP communication is done between Leader 1 and Leader 2 when they have to exchange data between them, for comparisons and therefore guarantee that they are executing the same files, and they’re results are the same.

6.1.3 RMI communications

RMI stands for Remote Method Invocation and is the Java implementation of Remote Procedure Calls. This feature enables the calls of methods in objects which are implemented in a remote Java virtual Machine.

The RMI communications are used mainly for the retrieving of machine information from the Administrator, in order to submit jobs to machines, and are also used for data transmission to remote machines.

Because this is a transparent communication, i.e. the communication is done by the calling of remote methods, like regular local methods, the protocols used in this type of communication are very simple, and similar to a regular local method call.

6.2 Hash and Cryptography

In order to guarantee that there are no communication errors introduced, one can implement secure communications. The kind of required security is achieved by using a hash algorithm to guarantee the consistency of the data transferred, and public key cryptography to guarantee that the data was not altered in the communication channel.

The hash of the transmitted message is digitally signed using a cryptography algorithm. The hash algorithm used is the SHA-1 and the digital signature algorithm is the DSA. The digitally signed message is sent to the receiver along with the original message.

The receiver, will confirm the signature using the public key of the sender, and will generate its own message hash and compare it with the received one in order to confirm the data integrity.

The generation of the public/private keys is done when a machine registers in the system. When a machine needs to access the public key of another machine, it will request it to the Administrator. Although this method of distributing the public key may be a target for a man in the middle attack, we can assume that the Administrator is a trusted key distribution center.

6.3 Important Classes

The application has two major functions; one is the main function that implements the communications between the machines to remotely execute the programs. The other function just sends files to the main one. This second function will be called RecebeThread and the main function will be called Bizant.

6.3.1 RecebeThread

The main class of this functionality is constituted by the thread RecebeThread and among the methods that are implemented here; there are two that can be brought up. One is called leFicheiro() and the other is leInput(). The first is responsible for reading the executable file form the disk, and the second one is responsible for the reading of the input file.

There is another important method, enviaVarios(), this method is responsible for sending multiple inputs for the same executable file.

There is another class that in fact is a thread, and is the responsible for sending the data to bizant.
Some methods are responsible for initializations, like `administrador()`, `cliente()`, `inicaRMI()` and `leConfis()`, which are responsible for initiate the administrator, the client, the RMI communications and for reading data in the configuration file, respectively.

The method called `submissionMachine()` is responsible for the processing of the communications between the SM, which is itself, and the EMs. It also launches a thread, called `TrataRecebe`. This thread is responsible for the communication with the thread `RecebeThread`, and as it receives files form `RecebeThread`, it communicates with the `Administrator` to retrieve the information of the EM machines. Then it starts a class called `EnviaFicheiros`, which transfers data with the EMs, in order to implement the Voltan algorithm, also this class instantiates another class whose function is to compare data retrieved form the EMs, and is called `CompraFiles`.

The class `Escuta` is also a thread, and it manages the requests for registration of machines in the system, and is responsible for deleting machines from the system. This class is only used in `Administrator` mode.

The class `ComunicaUDP`, is the Client side for the registration in the system, it has also the function of alerting the `Administrator` when this machine wants to get out of the system.

The fundamental class in this project is the remote class. It is called `PoolList` and it most important methods are those responsible for retrieving machine information, in the first step of the submission of jobs, and later the methods responsible for the execution of programs, as it is specified by the modified Voltan algorithm, that is implemented.

To make the first action, i.e. to retrieve machine information, there is the method `daPcs()`, this method randomly chooses two machines form the list, and retrieves them to the SM.

To implement the Voltan algorithm, there are the methods `EM()`, `leader1()`, `leader2()`, `Leader()`, and `Executa()`. `EM()` receives the data and calls `leader1()` or `leader2()` depending on the characteristic of the remote machine. Then, the two leaders communicate with each other, and call the `Leader()` method, this method launches the Follower thread, which is an inner thread. Both, Leader and Follower call the `Executa()` method. This is the method responsible for the execution of the program, with its input and retrieving the output and error. [5]

### 7. Tests

Some tests were made to verify the scalability of the system, and its capacity to determine byzantine failures in the execution.

The test files executed a factorial function, of a given number, passed by the `STDIN` and returned the value of that factorial to `STDOUT`. If there were any errors they were written to `STDERR`. The factorial is calculated only for values form 0 to 20, because negative values have no factorial, and factorial of numbers above 20 are very large.

To test the scalability some files with multiple inputs were submitted. This simulates the number of machines in the system that submits data for execution approximately at the same time, and in a worst case scenario. Note that the tests were made with only three computers, being this the minimum number of machines in the system.

There were input files with 25, 50 and 100 inputs, and there were inputs of 5, 10 and 15, to test various alternatives in order of charge, and computing complexity.

In the tables below, are the average execution times of a local execution (Table 1) and a remote execution (Table 2). As one can see, the remote execution spends more time, not too much in comparison, but relatively to the local execution time, it can, in some cases, be an important difference.

As one can see, in Table 3, the execution times, when the system has various charges, increment very quickly. In fact comparing these execution times with the ones in Table 1 (the local execution times) we can denote a very significant increase. Of course that in these tests, the files were small and light in processing, and this difference in execution is not important, but with other kind of files, this can be overwhelming.

Although there are some oscillations in the execution times, due to the execution of other process in the same machine, namely the garbage collector of Java, the main goal of these tests was to determine the magnitude of system charge, therefore these oscillations can be ignored.
Conclusions

The cornerstones of a distributed system immune to Byzantine failures are replication and safety in data transmission. To achieve these goals, the use of an algorithm such as Voltan, as well as the implementation of security measures in communication revealed as a very interesting approach.

Various types of communication protocols, like UDP or TCP are used to achieve conflicting goals such as performance versus delivery guarantee. On the other hand, RMI justified its use since it is a very interesting approach to guarantee transparency in distributed systems. However, although RMI is a good way to implement Remote Procedure Calls, Java has a garbage collector which uses some crucial resources and it may be a burden in terms of performance in a system like this.

So, one can conclude that a distributed system with these characteristics, with more efficiency in terms of remote execution may not imply a significant cost in time spending to execute. However the time spent in the remote execution is directly dependent of the communications made and the size of the data to transfer. So it is relevant the fact that the execution occurs two times in two machines and that there is various data exchanging between those machines and the submitting one to, therefore, the size of the files cannot be big either, or the benefits of using a system like this can be diluted unless it is imperious that the error avoidance is used.

References


<table>
<thead>
<tr>
<th></th>
<th>Factorial(5)</th>
<th>Factorial(10)</th>
<th>Factorial(15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>13442.84 ms</td>
<td>12523.52 ms</td>
<td>16230.96 ms</td>
</tr>
<tr>
<td>50</td>
<td>33558.3 ms</td>
<td>28327.02 ms</td>
<td>29484.82 ms</td>
</tr>
<tr>
<td>100</td>
<td>57824.68 ms</td>
<td>55908.34 ms</td>
<td>53089.36 ms</td>
</tr>
</tbody>
</table>

Table 1: Table with the execution times of the different system charges

<table>
<thead>
<tr>
<th></th>
<th>Factorial(5)</th>
<th>Factorial(10)</th>
<th>Factorial(15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>655.1 ms</td>
<td>871.2 ms</td>
<td>968.4 ms</td>
<td>Average</td>
</tr>
</tbody>
</table>

Table 2: Table with the local execution

<table>
<thead>
<tr>
<th></th>
<th>Factorial(5)</th>
<th>Factorial(10)</th>
<th>Factorial(15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1305.8 ms</td>
<td>1196.7 ms</td>
<td>1076.4 ms</td>
<td>Average</td>
</tr>
</tbody>
</table>

Table 3: times Table with the remote execution times