CSCI 53700 : Fall 2022

Assignment Number 2

Abrar Jahin

Introduction

Distributed systems may have no physically synchronous global clock (for this implementation as there are some like NTP servers or TSA servers), so a logical clock allows global ordering on events from different processes in such systems [1]. Clocks need not be synchronized absolutely if two processes do not interact; it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable and thus not cause problems.

Lamport's bakery algorithm is a computer algorithm devised by computer scientist Leslie Lamport, which is intended to improve the safety in the usage of shared resources among multiple threads by means of mutual exclusion [2].

Simpler way to implement this use increments its counter before each event in that process where event could be either Send, Receive or Internal communication. When a **process object (PO)** sends a message to **Master object (MO)**, it includes its counter value with the message. Similar way on receiving message the counter of recipient is updated (if necessary) by the value send by MO, the counter is then incremented by I before the message is considered received. In our implementation, MO once receive such message from any PO it will calculate the average of all POs counter value including its counter value then calculate the offset for each and finally broadcast to all POs.

In this assignment, we supposed to implement a simple distributed system based on the principle of clock consistency, associated drifts, inter-process-communication, and the end-to-end argument. Here section 2 is briefly covered the Interaction model, Failure model and Data encryption. The implementation is covered in section 3. Finally, section 4 represents the analysis based on the simulation take place between 4 POS and MO with different event probabilities.

Interaction Model

- What are the main entities in the system?

4 POs which are Process Object, and a 1 MO is Master object. Each POs and MO will have its own logical clock. All POS and MO are allowed to perform three different type events, are: send, receive and internal communication with different probabilities.

- How to Interact?

By message passing in asynchronous fashion they interact with MO, where each PO maintains a message queue. POs of this system follow end-to-end principle; where one end is PO; is responsible to send its current counter value along with the message. Other end is MO which is responsible to maintain the clock consistency in the system by broadcasting the current clock value (which is basically request number) to all existing POs. Here inter-process messages get encrypted at the sending end and get decrypted at the receiving end that contains logical clock values from POS and offset value from MO.

- What are the characteristics that affect their individual and collective behavior?

By changing the probabilities of three different events inside POS and MO have the effect on both individual and collective behavior. More and More communication with MO will lead to less drift compare with others.

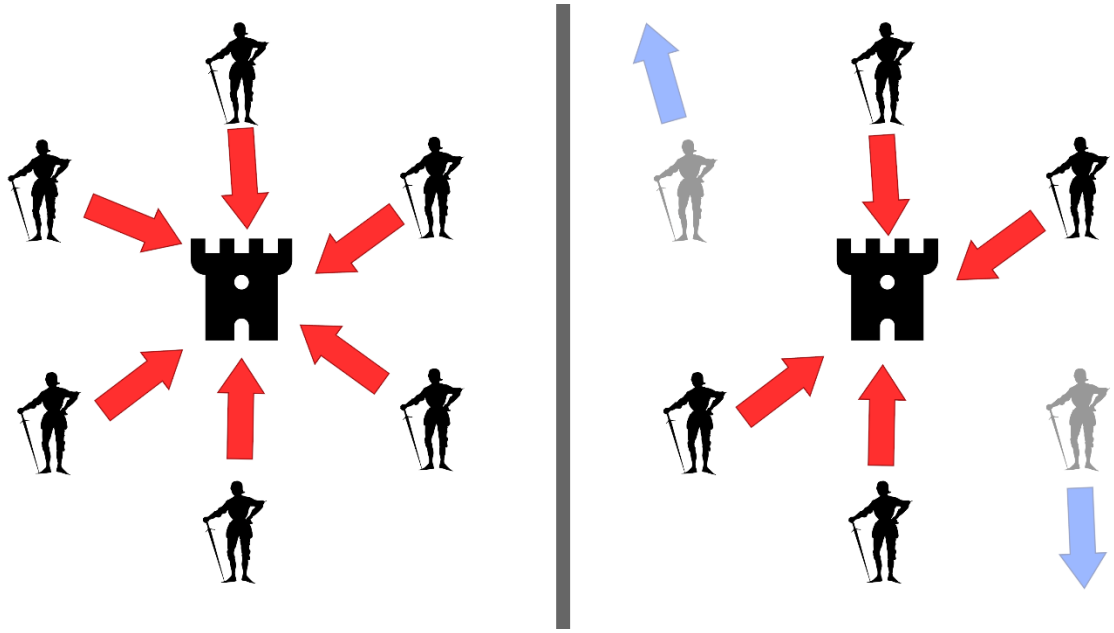

Figure 1: Byzantine fault : If all generals attack in coordination, the battle is won (left). If two generals falsely declare that they intend to attack, but instead retreat, the battle is lost (right)

Failure Model

- There will be arbitrary failure only during message receiving. The arbitrary behavior could be either not to update the logical clock value or to move forward the logical clock value by 100 with different user provided profanities for each event.
- Due to message buffer overflow may also result to omission failure. This situation may arise when POS send message very frequently and MO unable to handle that lead buffer become full and message can be lost.
- A Byzantine fault is any fault presenting different symptoms to different observers. A Byzantine failure is the loss of a system service due to a Byzantine fault in systems that require consensus. In a distributed system, this kind of faults can also happen.
- **Pros**: Due to asynchronous principle it is possible to achieve higher concurrency as after sending it don't need to wait for the response from MO.
- **Cons**: Here the message may get lost as there no checksum mechanism applies between two ends so no guarantee that message delivered correctly or not.

User Authentication:

We have a predetermined list of users and passwords that we utilize for authentication. Only after receiving a request containing a username from the list is the password verified, and if it is accurate, the file upload process begins. The file is not uploading if authentication is not feasible. Additionally, a response is returned with the upload status when the file has been uploaded. If it has been fully uploaded, the solution is to give that. It is also indicating whether the file has only been half uploaded. If authentication is unsuccessful, the sent file will not upload, and the client will receive a response stating that the file upload failed.

Already registered username and passwords are-

Username	Password
a	1
b	2
c	3
d	4
e	5
f	6
g	7
h	8
i	9
j	10

More authenticated users can be added from the "**UserLogin**" class at line no 10 where you need to add new dictionary/HashMap entry for a new user as all the user data are stored in RAM.

File Data Encryption:

Data encryption is important because it allows to securely protect data that you don't want third parties to have access to. As the network is an open platform so the data that are transmitted may contain sensitive information and can be exposed to anyone else. Therefore, data encryption mechanism is also introduced in our implementation. Here data get encrypted in the sending end point and get decrypted in the receiving end.

Figure 2: Data Encryption mechanism

The encryption and decryption technique are implemented in both POS and MO end point. So that when one PO send message to MO at first, it's encrypted the time stamp and send it to the MO. In the MO it has decrypt mechanism apply that on the received message. Later when MO needs to broadcast the offset to

others POS it again applies the encryption mechanism. For this assignment, we don't need to implement any encryption.

For our case, we are using AES encryption which is an asymmetric key encryption. We are not encrypting username and password. We are encrypting only file chanks before sending. We are dividing text files in different parts as file are big content and all the contents are sent into equal packets to send via TCP. And when the file is received in the server, it is being decrypted with the same key.

Implementation

I have implemented the model with a client-server model. All the functions are distributed in different classes. Let us discuss all in brief.
Client:
The client is the most straightforward part. For the client, there is mainly 2 class:

1. SocketClient.java (Which is **runnable** and runs the client)
2. ClientRequest.java (Handles the socket communication works for the client)

Server:
The server consists of mainly these classes:

1. SocketServer.java (Which is **runnable** and runs the client)
2. UserLogin.java (Responsible for user management)
3. ServerRequestHandler.java (This file handles the multi-threaded server and provides all the responses as well)

There are some standard classes as well which are used by both clients and servers. Let us have a look at the files in brief-

1. AES.java (Used for AES encryption)
2. Encryption.java (Used for encryption but was replaced by AES.java later time)
3. FailingModel.java (Providing another failing model like a server crash or network crush)
4. FileHandler.java (Responsible for file conversion to chanks and file merging to actual format from chanks)

For my implementation, I am passing username, password, and file simultaneously to the server, and the server is processing the request. If the bid is successful, the server is returning a success. If failed, it produces a failure message. If anything, terrible happens, the server or client will not crash; it will catch the exceptions and show the results for the exception. If the server or client is disconnected for any circumstances, they are also automatically connected if the network is fixed.

Running the code:

Ther are mainly 2 executable files. Lets discuss how we can run them with and without command.

1. SocketServer.java

If you run this without a parameter, then there will be default parameters which will be used. And if you need to run it with parameters, then the parameters are-

a. Port: this is the only parameter, and it is an integer. If you run it like-

Running command

```
javac .\javaSocket\SocketServer.java
cd .\javaSocket\
java SocketServer <optional_parameter_1>
```

If you don't provide any parameter, default port would be 8080. If you provide any, that would be the port for the server application.

2. SocketClient.java

There are some parameters for these applications. Let's see how we can run the programme first.

```
javac .\javaSocket\SocketClient.java
cd .\javaSocket\
java SocketClient <op_param_1> <op_param_2> <op_param_3> <op_param_4> <op_param_5>
```

Lets see what are the parameters-

Symbol	Parameter Name	Default Value
op_param_1	Server IP or url	Localhost
op_param_2	Port	8080
op_param_3	File Path	`"D://Education//Academic//Distributed Computing//Assignments//2//java Code//javaSocket//javaSocket//try.txt"`
op_param_4	Username	a
op_param_5	Password	1

I have added the makefile file for running the file in the same server. So, if you use the makefile, then a server and client will run in the same machine. If you need to run the client in different machine, you need to update the cmake file in that server. Only updating the localhost portion (server) and updating the port will do the trick.

Implementation Coverages:

I have implemented this thing as bonus implementation-

1. Multi-threaded Server
2. Fault tolerant server
3. Fault tolerant client
4. Automatic network communication establishment for server and client after failure
5. Data passing through serialization and de-serialization
6. Fault counting through exception
7. File transferring through chanks
8. Encrypted file transfer
9. Login check, if login failed, file is not uploaded or rejected

I have implemented counter for the exceptions so that this implementation can be extended easily. And all the

Result Analysis

To set equal Byzantine failure probabilities for the first experiment, we use the "FailingModel" class. Other failures could include network interruption, server crush, client crush, etc. Figures 3 and 4 depict the outcome after considering those failures.

For figure 4, X axis is showing total no of requests, success and failure counts in the server and Y axis is showing timeframe when the input was collected. Blue line is showing the total no of requests, ash line is showing total success transmission count and orange line is showing total no of failures by time.

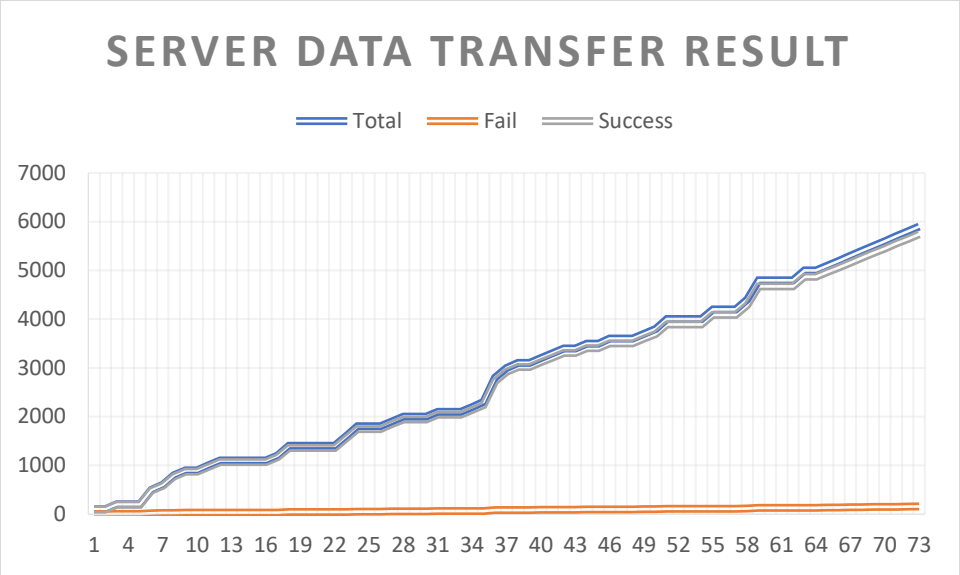

Figure 3: Server failure analysis: unequal probability of send and receive

By analyzing the server portion, we are seeing that our file upload failures are not more than 5%.

In figure 4, we are seeing that file sending error is almost consistent for different times. In the figure X axis is showing total no of requests and Y axis is showing the count. Blue and orange color is showing the success and failures in the requests. And after analyzing the result, we are seeing that, less thatn 5% request are failing if username and password is correct and file is provided.

Figure 4: Client Failure model 1 : relatively low probability of receive

For every request done, we are sending the current clock count from server to client to synchronize the client clocks as well.

Conclusion

When using a TSP connection, we must divide a large file into several equal pieces. If we don't do it, the likelihood that the data transmission will fail will grow, and if the data is too big, data transmission will fail. Furthermore, encryption is essential for data transfer; otherwise, data compromise is risky. Additionally, there might be various problems during data transfer across systems, which we should be aware of. Such network and system failures ought to be fault tolerant of our systems.

References:

1. Logical Clock: https://en.wikipedia.org/wiki/Logical_clock
2. Lamport's bakery algorithm: https://en.wikipedia.org/wiki/Lamport's_bakery_algorithm
3. Byzantine fault: https://en.wikipedia.org/wiki/Byzantine_fault
4. A Study on Byzantine Fault Tolerance Methods in Distributed Networks: https://www.sciencedirect.com/science/article/pii/S1877050916304641?ref=pdf_download&fr=RR-2&rr=755cdbf4da472bc4
5. Modes of Failure: https://medium.com/baseds/modes-of-failure-part-2-4d050794be2f#:~:text=Arbitrary%20failures%20are%20ones%20that,the%20system%20communicate%20with%20it.