Distributed Systems Project 01 – Multi-Server Chat System

K. I. Sewwandi Perera
login name: kalubovilage
student number: 725485

1. Heterogeneity

Distributed system should be capable of running its components on difference resources with different characteristics in terms of networks, computer hardware, operating systems, programming languages and implementations by different developers.

Internet consists of different types of networks like intranets and wireless networks. Therefore, distributed systems should be capable of communicating with distributed components which are in different networks. All communications among chat servers and chat client applications use Transmission Control Protocol (TCP). Therefore the network level heterogeneity is handled by TCP. So, the chat clients and servers can run on any type of network and communicate with each other regardless of the network they use.

Different computer hardware stores data in different formats, and different Operating Systems (OS) handle data and message communications in different ways. Therefore, distributed components should be capable of handling this heterogeneity. Chat server is developed using Java and it can be compiled and run on top of Java Virtual Machine (JVM). Therefore, heterogeneity of computer hardware and OS are handled by JVM. Here, the JVM acts as a middleware to handle computer hardware and OS level heterogeneity.

Finally, if the components of the application are written in different programming languages, or by different programmers, these components might not be able to communicate with each other because of the different data structures and message formats they have used. The chat server and client applications use agreed upon message formats for communications. Therefore, if someone wants to develop a new version of chat server using a different programming language, it can still work with the existing chat servers and clients because all communications use these agreed upon message formats.

2. Scalability

Capability of a distributed system to handle the growth of number of users is considered as the scalability of the distributed system.

The chat server application is designed as multiple servers that can coordinate and provide the functionality to end users. Therefore, scalability of the system can be easily achieved by increasing the number of servers as per the use of the application. But still there are some limitations and concerns of the scalability of this model.

1. The number of servers that are used to provide the service should be decided before starting the application and it is fixed throughout the life time of the application. Therefore, if we expect *n* users in the peek time, the number of servers to satisfy *n* users should be always in operation. That is an overhead because we have to bare the unwanted cost of physical resources. This issue can be handled by improving the chat server so it can dynamically add and remove coordinating servers to/from the system at any time. Then we need to have more servers only when it is needed.

2. Even if we have multiple servers, if all clients, or most of them, want to join the same chat room, they should be connected to the same chat server. In that case, the server which hosts the chat room will not be able to handle all clients without a performance loss. So, even if we have multiple servers running on the system, we might not be able to provide the scalability as needed.

But if all clients are spread across multiple chat servers and if we have initially assigned enough number of chat servers to handle all clients, the system provides the scalability without any issues because there are no bottlenecks in the system.

3. Failure Handling

Chat servers are capable of detecting and masking failures of chat clients. If a chat client fails, the server connected to the client detects the failure through the TCP connection and act as the client has left the application. The server releases the identifier of the client and any owned chat rooms of the client, so someone else can use those identifiers later. All other clients who were in the same chat room as the failed client are notified saying that the client has left. So, no one in the system aware that chat client has left because of a failure.

But chat servers are made with the assumption that they do not fail or crash. Therefore if a chat server fails, the failure will not be detected by other servers or even by the clients connected to it. In that case clients will throw exceptions. Even, the new clients who are trying to connect to the failed server will throw exceptions because a failure tolerance mechanism is not available. Servers communicate with each other using short-term TCP connections only when it is needed to coordinate something. Because servers are not capable of detecting failures of other servers, they will still try to communicate with failed servers. So, this will lead to many issues in the system. For an example, if a client in a server redirected to another server to join a chat room, but if the server has already failed, the client will get an exception. Therefore, the chat servers should be improved to detect failures of other servers, and either tolerate or mask them depending on the situation.

4. Concurrency

Chat servers contain many resources, data structures that are shared among all clients connected to it. Some examples of them are list of clients connected to the server, list of chat rooms owned by the server, identifiers of chat rooms owned by other servers and list of members in each chat room. Therefore, if concurrency is not handled properly, it is possible that these data structures become corrupted because of the concurrent updates by different users. For an example, let's assume one member in a chat room is leaving and at the same time another member is joining the chat room. If the members list of the chat room is not properly synchronised, it is possible that the both clients are listed in the chat room even if clients think that their operations are properly handled at the server side.

The easiest way to handle concurrency is making the access to shared resources sequential. But in the chat server application, we have multiple threads running for each client and if we make their resource access operations sequential, it will slow down the server very much. Therefore we need another approach to do this.

Concurrency in the chat server application is handled using synchronous methods and by using thread safe data structures provided by java.util.concurrent package. So, all clients connected to servers as well as other coordinating servers can concurrently access and update shared resources without any issues.

5. Transparency

Transparency is the aspect of hiding the components of a distributed system from its users and application developers. There are different types of transparencies.

The chat clients in our application should use same operations to access chat servers irrespective of whether the servers are locally or remotely hosted. Therefore chat servers provide access transparency.

Chat clients should know the IP address of the server to access it. Therefore chat servers are not location transparent. But we can use Domain Name System instead of the IP addresses to have the location transparency.

Each chat server use different threads to handle its clients and also it uses different threads to handle coordination massages from other servers. So, all clients and server coordination processes access shared resources of the chat server at the same time and clients do not have any idea about it. Therefore, chat servers provide concurrency transparency.

Chat servers are developed with the assumption that they will never crash or fail. Therefore failures of servers are not handled. So, if a server crashes, clients will get exceptions. Therefore the chat server does not provide failure transparency.

Since the chat client connects to the server using address of the server and because clients and servers communicate using TCP connections we cannot move server resources without affecting chat operation. Therefore chat server does not provide mobility transparency.

Chat servers do not reconfigure dynamically to improve performance as the load varies. Therefore they do not provide performance transparency.

Chat server can expand in scale by having multiple chat servers running in parallel. This feature does not need any changes in system structure or application algorithms. Therefore chat servers provide scaling transparency.