University of Groningen

Distributed Systems

Author:
Euaggelos Karountzos
Bart Marinissen
Joris Schaefers

Student Number:
s2705532

October 27, 2015

Contents

1 Itroduction

Nowadays distributes systems are everywhere and they are widely used for sharing information and data all over the globe. A very good example of such a system is the torrent systems which is a peer-to-peer distribution of data. Our idea was to create such a system that allows users to stream data (e.g. video streaming). We decided to go with this idea since video streaming has been acquiring more user in the past years and creating such a distributed system would be a challenge on its own.

2 State of the art

2.1 Examples of Distributed Systems

A very good example of video streaming application is FreeCast[1] build in Java programming language and it was released in 2006. Since then more video streaming applications have been created (some of them have been even integrated on torrent systems such as the μ-torrent [2]). A relatively new application as such is the CoolStreaming[3] which allows users to view streams of sports. More examples of distributed systems is the banking system or the Bit Coin system.

2.2 Trend and State of the art

Due to the large scale of such systems the industry as well as the researches move towards standardized methods. A good example of middle-ware used in the industry is the DCE (an early one), COBRA(a modern one) and Jini. Nowadays COBRA (Common Object Request Broker Architecture) framework is widely used on the distributed systems domain. Very similar to object-oriented programming (encapsulation and reuse) it allows communication between applications written in different programming languages.

[1] http://www.freecast.org/
[2] http://www.bittorrent.com/
[3] http://www.coolstreaming.us/blog/diretta-tv/

3 Problem Statement

A distributed system is highly complex as it requires the proper cooperation as well as synchronization of many different nodes/clients. The domain of these problems is also quite extensive since it could refer to software failure (crash failure), malicious behavior and performance optimization.

3.1 Implementation and Design

To create our application we used Java. We used Maven to build and maintain the dependencies and Docker to test the behavior of the application. The source code is split between 5 packages.

DTO (Data Transfer Object) package: This package contains all our messages that we use and distribute for the managing part of the system and to distribute information between the nodes. This contains election messages, join and leave messages.

Algorithms package: This Package contains all the algorithms used to handle the packages that either are being received or send by a node. Examples of that are the election algorithm used to handle the elections or the Casual and Basic multi-cast algorithms.

Main package: This package contains the essential information about the nodes such as names, roles, IP addresses etc.

Networking package: This package contains classes that are used for sending and receiving the streaming packets.

Tree package: This package contains the tree structure.

Root package: On the top (the root package) we have the main class of our application.

We decided that it was necessary to use both the basic as well as the casual multi-cast. Their use is described below;

Basic multi-cast: The basic multi-cast is used for the elections but it is also used by the Casual ordered multi-cast since the latter is based on the former there is no way around that. A second important functionality of the Casual order multi-cast is that by using it we ensure that the network will learn about the

existence of a child (a joining node) only after it learns the existence of its parent.

Casual order multi-cast: The casual order multi-cast is used by the nodes to announce joins and leaves from/to the tree.

The participants of the network are all nodes, which we can have 3 different roles;

Leader: The leader is elected by the rest of the nodes and its role is to log the nodes that join and leave.

Streamer: A streamer is a node that produces a streams and then passes it on to the rest of the nodes.

Client: A client is every node on the network that is neither a streamer or a leader and they can either listen to the stream and pass it on or simply act as nodes and simply propagate it through the network.

3.2 TCP/IP protocols

For the essential managing messages we used TCP since it guaranties that the message will reach its destination. For the streaming part however, TCP would not work properly so we used UDP for the streaming part of the application.

3.3 The structure of the network

The first problem we encountered was the structure of the system. As we want the stream to start from the streaming nodes and then to propagate through adjacent nodes to everyone that wants to receive it. To solve this problem we created a tree structure. To avoid overload of some nodes (such as the root node) we decided to put a limit on how many children a node can have. We set this limit to a maximum of three nodes.

3.4 Dynamic joining

Initially we wanted to make the system easily accessible by allowing everyone dynamically joining the network. This however would not be able to work over multiple networks due to the broadcast limitations. So we made the limitation of having the

whole network working only behind a router. For the new host to join in the network he is required to make an IP broadcast. Multiple nodes that can accept more connection will reply and the new host can decide which one he would like to join to.

3.5 Election System

The next problem on the list was how the leader will be elected. The initial plan was to try the bully algorithm, but this idea was quickly discarded because elections had to take place each time a new node was added to the domain of connected nodes. As one can understand that would be extremely inefficient and it would not scale well since a large enough system would be in a constant state of elections. Because of the above reasons we decided to use the HS algorithm. This does not come without any optimization issues since it would not scale properly as well. A way to optimize that, and avoid flooding the system with election messages was to either use a random time between elections (i.e. elections would take place in a random time between 5 and 10 minutes OR in cane the current leader fails) or elections would take place once the size of the system is deviated by a certain amount of nodes (i.e. elections will take place once the size of the system has been increased or decreased by 10% OR in case the current leader fails). The circle used in the HS elections is created dynamically when new node joins the network.

3.6 Propagate the Changes

The next problem in the list was to figure out how the changes will propagate through the network. For example, how does a node knows where to stream to. This was solved by using and sharing a list with all the nodes in the system. Once a node has joined the network its parent node, register the new children node into a list and then it propagates it through the entire network. This would work as long as two nodes do not connect at the same time. To avoid this we decided that each node that wants to join the network should first contact with the leader of the group which is responsible for managing the list himself.

3.7 Crash Detection and Recovery

Crash detection and recovery are important aspects of distributes systems. As they consist of many nodes, it cannot be the case that one node's failure can make the rest of the nodes unusable. For regular crash detection we use regular heart beats that are sent every now and then from to and from connected nodes. This ensures that if one node fails than the neighbor nodes will be able detect. In order to avoid unnecessary traffic as much as possible we wanted to combine heartbeat messages with other kind of messages but the lack of time did not allow us to do so.

We did not have the time to create the necessary functionality to deal with crash recovery. The plan however was to deal with it with - in our opinion - the most elegant was possible. One way to do that would be to recreate or shift the entire tree below the crashed node and reconnect it to the initial structure. This however has two main restrictions. First of all, if we choose to shift the whole tree upwards once a node fails that means that one node will get three children which violates our restrictions about the distribution of the load between nodes. An alternative to that would be to dissolve the whole tree connected to that node and then have the nodes rejoin the network. This means that in case of a node failure the systems would have to accept a huge amount of node simultaneously trying to reconnect to it, and thus we discarded this idea as well. Our solution to that problem was to replace the crashed node with one of the nodes on the lowest (leaf) nodes. This way the whole structure remains unaffected and only one node has to be relocated.

3.8 Synchronization

Our system is synchronous since we use heartbeats between the nodes in the system. We use sequence numbers for the Basic multi-cast and vector clocks for the Casual ordered multi-cast.

3.9 Usage of External Frameworks

We did not have the chance to integrate or use any external frameworks mostly because as far as our application is concerned they are not essential and Java is multi-platform language which makes the application mobile.

3.10 Defense against Malicious Behavior

We did not really concerned about implementing functionalities to defend against malicious behavior in our system. However if our software was to be used in industrial or commercial level many security measurements should be taken into account such as DOS (Denial of Service) or MitM (Man in the Middle) attacks.

4 Relation to Distributed Systems

We can define distributed a distributed system as a collection of independent computers that appears to the host user as a single system. We believe that our application meets the above requirements since the host user only sees the streaming that receives while in the background a huge number of different and probably distance machines cooperate to achieve that. Our system is also open, meaning that everyone can join and can immediately start streaming or listen to a stream. Scalability is also an important aspect of a distributed systems. Since such a system is not build around a physical location, such as a server it can scale up efficiently as more and more nodes are added to it. An important benefit of having such a system is that there is no a single point of failure and no bottle-necks are formed between nodes.

4.1 Conclusion

Overall the design, creation and deployment of a distributed application in such a short amount of time was a challenge by itself. Getting an insight on the logic behind a distributed system was enlightening and beneficial and it helped us to grab the overall concept of the course.

References

[1] Valeria Cardellini, Emiliano Casalicchio, Michele Cola janni, Philip S. Yu *IBM Research Report - The State of the Art in Locally Distributed Web-server Systems* 2001.

[2] Friedemann Mattern *State of the Art and Future Trends in Distributed Systems and Ubiquitous Computing* 2000.

References