Olli Keski-Hynnilä, Katri Mattila & Jerry Tamlander

Distributed systems task 1

The purpose of this project was to design a simple three node application for this task in a way that it fulfills the requirements with the scope of the project being small enough. Our core functionality comes from clients and servers communicating locally with each other by sending messages via the server.

Project is a simple rock paper scissors game. Three nodes form a functional distributed application by working together. Participating client-nodes can exchange messages with each other via the server while playing the game. Application is based on a client-server model architecture. Server implements a specific service and clients request a service from the server by sending a request. Clients and server communicate via request-reply method.

Clients join to the game and send messages and plays to the server and server then delivers the messages to the other client. Client can send plays and messages to server and the server responds accordingly. The clients can only communicate with each other via the server and not directly. This design choice was intentional, since in the domain of multiplayer games clients never communicate with each other directly. Logging is session based so no logs are kept from other sessions.

TCP was used with the communication since the game is turn based instead of real time. With a real time game UDP messaging might've been used but we felt that TCP was the better choice for this kind of an application since the delay isn't as important as reliability. UDP messaging is used when sending messages from the server to clients and the client has a thread for listening UDP-messages from the server.

Problems we encountered were related to some difficulties with python and it's sockets. The debugging can sometimes be frustrating with ports still being used after closing the application. This lead to a more careful approach when compiling code and working with the project which sometimes can lead to a more slower progress, but with a good balance improves the development process. Also dividing up the project to do work without everyone being present felt hard, since the scope of the project isn't that large. We ended up doing most of the work with everyone together and some development so that we "took turns" coding. This method is slower than doing parallel work, but removes the danger of merge conflicts.

To run this project you need to have python3 installed. Installation instructions can be found from the official python site: https://www.python.org. To run the project you first need to start the server in a terminal with the command 'python3 server.py' after which clients can be started in a different terminal with `python3 client.py <port_number>` where <port_number> is any available port. When starting clients ports need to be different since they are reserved to the client. After starting both clients the client prints instructions which can be used to play the game.

Measuring the time it takes for messages to travel between nodes is not relevant in our solution since the project works entirely locally. Instead of measuring the amount of time it takes for messages to travel between nodes the thing that is measured is the speed of the computer's processor. If nodes would've been located in different machines and the messages were sent over the internet the measuring would've been more interesting and telling.

The average time for sending 50 messages between the client and server is around ~0,003 seconds.

With 25 messages of three different sizes the results are virtually identical to the 50 random messages. With 50 000 messages the duration is around ~1,3 seconds.

The inter arrival rate of messages is extremely small, around ~0,0001 seconds.

The architecture is extremely reliable due to it working in localhost. There are also some disadvantages from this architecture. While this architecture works great on local projects with two clients, if it would be hosted on a remote server that could support multiple clients it could cause problems for example bottlenecks. When there's a lot of traffic, server may get overloaded because server can only accommodate a limited number of requests at a time. Benefits from this kind of architecture are that data is in a single place (server) so it's easy to protect and data can be accessed efficiently. That's why applications that use authentication and authorisation could benefit from this type of architecture. Applications that need nodes to be easily replaced, upgraded or relocated could use this type or architecture because nodes are independent and request data only from the server.