Software Engineering – specification

Antoine GRIMOD, Arnaud DABY-SEESARAM, Gabriel JANTET and Zibo YANG

September 2021

1 Project introduction

The project is to build a online multiplayer Real Time Strategy game (RTS). In this project, we intend to write two main applications: a server and a client.

The RTS will take place in a fantasy world in which each player evolves. Their main goal is to build and continuously improve a village or small town, and raise and interact with other villages and towns. The players controls and influence the number of inhabitants, their professions and resources usage[1].

The players will be able to interact with one another through:

military campaigns a player can raise an army and use it to attack another player

trades two players can agree on a commercial trade

2 Project management and technical choices

2.1 Project management

In order to keep track of the project advancement, we use a plain bare repository (with no web-GUI) available at `git.sofamaniac.xyz:rts.git`. The repository is accessibe for reading to anyone (everyone can clone it using `git clone git://git.sofamaniac.xyz/rts`).

We will meet at least once a week during Software Engineering sessions and have a Discord channel in order to stay in touch during the week.

2.2 Programming languages and implementation tests

We will probably[2] use:

- Rust for the server-side
- Go for the client-side

[1] An inhabitant will cost resources. Thus, the player has to keep a balance between developing its place and increasing its size.

[2] We might have a change of heart after the first two weeks.

None of us have ever used these languages in whole projects in the past, but we are willing to discover them. This said, if the group decide that one of the above language no longer suits the project aim or that another known language would be a better solution, we would switch to the latest.

There exist static analysers for both these languages. We intend to try some of them on our code.

Each distinct implementation task will be developed in separate git-branches. These will be merged as soon as the features have been improved and stabilised. An implementation will be considered over if it matches its specification and passes the corresponding tests (which could be created for this implementation).

2.3 Networking part

2.3.1 Protocol

During the game, the server and the client might need to exchange data (map parts, location of the players, . . .).

Due to the large number of clients, the server might have several requests at the same time. The client might wait for several requests to complete as well, especially during the launch, as the client will have to load the player's location, their wealth, the map, . . .

The server and the client will share the following protocol:

- On the start: The client opens a TCP connection to the server.
- On the stop: The client sends an exit message and disconnects.

During the game, both the client and the server could use the following protocol to interact:

1. *A* sends "Q`{id}`.`{type}`:`query string`" to *B*, where `{id}` should be replaces by a query id, which is a string containing no dot'.', and the `type` is the query type that is a string containing so semicolon ":" .

   For example: "1634309700-ClientId" is a valid placeholder for `{id}` and "info" is a valid placeholder for `{type}`.

   The available queries are given below.
2. *B* sends "A`{id}`. . . " to *A*, where `{id}` refers to the same string as received in the "Q" string.
3. *A* treats the answer from the server and send back "S`{id}`. . . ", where `{id}` stands for the query id, and the . . . replace the status (either "ok" or "nok").

Where *A* represent the server (resp. the client), and *B* represent the client (resp. the server)

2.3.2 Tasks sharing between server and client

Running the client should not require heavy computational resources or use a large bandwidth. In order to respect that wish, this is the current repartition of the tasks between the client and the server:

client Stores:

- The state of the parts map it has already visited
- The position of some constructions

Computes:

- The local graphical effects
- The movement of the player as long as the map to print remains within the second boundary[3].

server Stores:

- The state of the map at all time
- The position of the players
- The possessions of the players

Computes:

- The remaining time for buildings
- Effects of map-wide actions

This part of the specification is likely to quickly evolve. This is a list of improvement and changes that may be implemented some day:

- Players have a cached version of what is on the map (for the parts they have already walked), but the map might have changed between two visits of a player.

  To manage this evolution, these three solutions might work:

  - Send the server some information it has on the map, to check its validity
  - The server stores the previous positions of the players and queries them to delete their cache of the updated parts of the map.
  - The client forgets the cached map of places that are outside the third boundary[4].

2.4 Protocol specification

The protocol specifies how the server can send queries to a client (or how a client can query the server).

[3]See the description of the map synchronisation between the client and the server.
[4]See the description of the map synchronisation between the client and the server.

2.4.1 Semantic of the messages

```
<message>  :=  <query> | <answer> | <status>
  <query>  :=  Q<id>.<keyword>:<option>
 <answer>  :=  A<id>.<option>
 <status>  :=  S<id>.ok
            |  S<id>.nok
     <id>  :=  [a-zA-Z0-9]+
<keyword>  :=  [a-zA-Z0-9]+
 <option>  :=  [a-zA-Z0-9]*
```

Figure 1: Semantic of the messages sent between the client and the server

The `<id>` element has to be unique for a given query. It will be used by the answer and status messages that follow to identifies the query that has been dealt with.

2.4.2 List of valid queries

Keyword	**Option**	Meaning
info	ϵ	Ask a client ID (if the server is the sender Ask the server ID (if a client is the sender
map	x,y,w,h	Client to server: Send the map centered on (x, y), of width w and height h
location	get	Server to Client: ask a player's location
location	set,<player's id>	Server to Client: Send a client another player's location.

2.5 Application packaging

The final work will be packaged:

- in an archive for a manual installation
- for several GNU/Linux-based OS (at least ArchLinux, Debian and GuixOs will be included)

3 Game description

3.1 Species

Each player will choose a species for its character at the beginning of the game. They will choose among:

elves Elves have access to magic. They will be able to develop new powers, nurture them and use them. These powers would help the player kick start the rebuilding of its village at first (if any damage was to be caused by attacks), and eventually help the player defend the village, using defensive spells.

An elf player will start its constructions slowly, but would store up a capacity to start better in case of destruction, as magic would then kick start the player.

humans The humans will have access to technology. This technology would bring them stability as they will be able to both defend themselves from enemies and computerise their blue collar jobs such as mining or manufacturing clothes.

As for the elfic magic, the human technology would take time to unlock completely, as the use of a piece of machinery for a certain amount of time might unlock a new system.

orcs The orcs will be more resilient than the above species, allowing them to build heavier structures and faster than the other players. This strength will allow a player to start the game faster.

Nonetheless, the orcs will not have tools to automatise the defence or the manufactures, which means that such tasks will always monopolise inhabitants.

3.2 Town specification

This section aims to give specifics on the different building types and the way a player can unlock new and better ones. It will be divided in, four sections, three ow which will cover the species specific structures a player can build and use.

3.2.1 Common structures

town hall to build any other structure and give order to the population (*i.e.* controlling how many citizen works in such and such area).

In case of an attack, if the town hall is destroyed, it needs to be rebuilt before any other building.

mines, forests, . . . in order to gain resources, which are required to build the structures and possibly trade with other players.

At the beginning of the game, each player will discover basic resources and inhabitants from and with which build a town hall and start expoiting those resources.

The forest will slowly grow as long as there exists a living tree. Thus, the player needs to be careful on how they use the resources available.

3.2.2 Elf village specific structures

Mana Tower Automagically gather mana

Academy of Magic Allocating inhabitants there allows the player to unlock new spells / abilities

3.2.3 Human village specific structures

Research centre By allocating inhabitants to the research centre, the player can collect techno-points which can be used to unlock new technologies

Automation centre ? same as research centre but used to unlock automation upgrades (could be a building unlocked by research)

3.2.4 Orc village specific structures

Orc breeding centre Orcs inhabitants and soldier should be fast to produce, maybe add a building that automatically create an orc every few seconds

3.3 Military specification

This section aims to give specifics on the different soldier types and the way a player can unlock new and better ones. It will be divided in, four sections, three ow which will cover the species specific military soldiers and structures a player can use.

3.3.1 Soldiers

We might specialize the units depending on their race, giving them bonuses and maluses For example for the first basic unit Men : att: +0; speed:+0; def:+0 Elves: att:+0; speed:+1; def:-1 Orcs: att:+1; speed: -1; def:+0

4 Other

The first tasks of the group is to define the elements of the game:

1. The different building types, professions, . . . for the every day virtual life,
2. The specification of the army and its evolution,
3. The protocol to use and the repartition of the information between the server and the client *i.e.* decide what will the server and the client each store.

Each of these three points lead to two implementations: one on the server-side and one on the client-side.

The group members are requested to write and complete the documentation related to their code, which is a way to split the work and ensures the correction of what is written.

Constraints:

- Provide a specification of the project: explain the main goal of the project, present the tasks distribution within the team, the temporary schedule, . . .
- Produce a documentation
- Provide testing tools
- Perform static analysis on the code
- Package the final work
- The controls should be fully customisable
- The client must be resource efficient

5 Planning

- October 1: Git tutorial
- October 5: Give our teachers an access to the git repository (which should contain the description / specification of the project: $\tilde{5}$ pages)
- October 8: Presentation of the project to the other groups