The Generals Game Design Document

Niko Koivumäki
University of Helsinki
Helsinki, Finland
niko.koivumaki@gmail.com

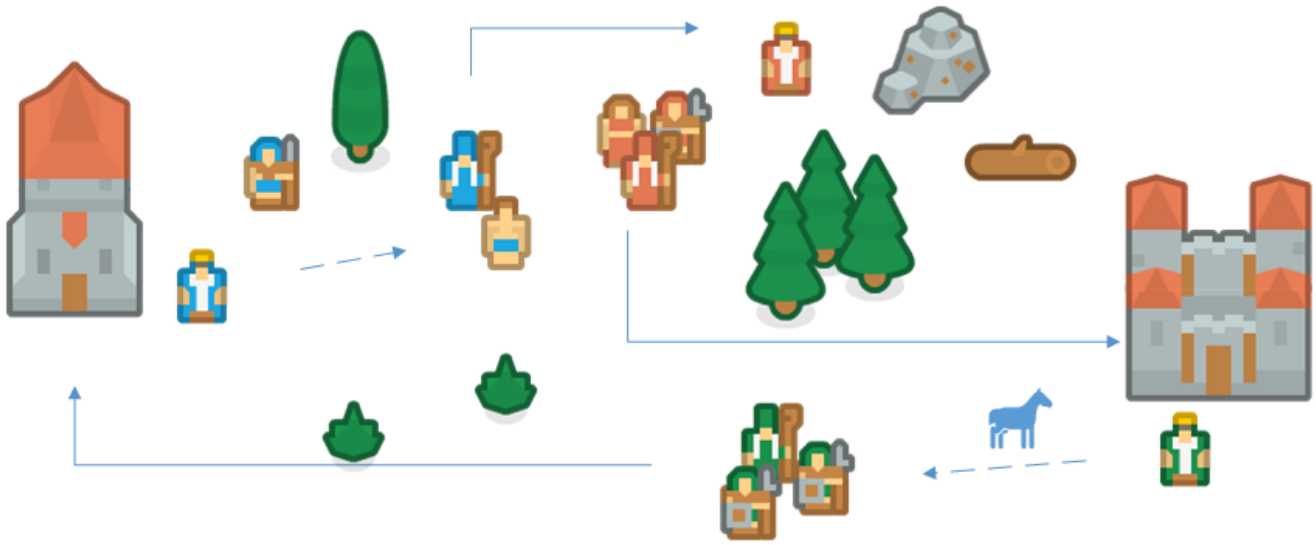

Figure 1: A mock up of gameplay

ABSTRACT

An assignment for Networked Systems and Services course at University of Helsinki. This document presents a multiplayer game with a reliable and scalable networking framework. While mimicing ACM style as an exercise, this document is in no way affiliated with ACM.

CCS CONCEPTS

• **Software and its engineering** → **Virtual worlds software**; **Peer-to-peer architectures**; • **Networks** → *Network reliability*.

KEYWORDS

game design, peer-to-peer networking, multiplayer games

Networked Systems and Services '19, October 03, 2019, Helsinki, FI

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Niko Koivumäki. 2019. The Generals Game Design Document. In *Proceedings of Networked Systems and Services '19: Game Design assignment (Networked Systems and Services '19).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Generals Game is a real time online multiplayer game, operating in either peer-to-peer or client-server fashion. It's designed to provide easy scaling both for individual game sessions and game ecosystems as a whole. Focus of the design is in providing service with low operating costs by offloading computing and traffic to clients where possible.

This design document gives a brief overlay on the key game concepts and mechanics, further exploring the networking solutions required to make the game work.

2 GAME OVERVIEW

Premise of the game is two have a number of generals (players) defend their base while simultaneously attacking opponents. Players can be assigned to teams for cooperative experience, but the game can also be played as a more chaotic free-for-all.

The game is divided into two separate modes: base building and combat. Base building is done without time limits, purely solo. A

player uses given building blocks to construct a base of their liking. After building a base, the player can queue up to fight other players.

The actual battle consists of players having their bases planted on a common, randomly generated map. Bases produce troops which the players can direct to attack other bases, and losing home base drops the player out of the game.

2.1 Gameplay loop

Gameplay loop inside the battle revolves around players not having global view of the game field. Players can give direct instant orders to troops residing in their base, but anything further than that requires messengers to be sent out. By default the messengers are reliable, but orders take time to deliver. Because troops are fully loyal and follow orders in a deterministic manner, it's possible to route a single messenger to several armies, but such maneuvers require careful planning. One cannot change the orders that are on route, except by sending another messenger with new, updated orders.

Same goes for receiving status of armies. While exceptions apply, the player is for most totally reliant on their messengers to bring back information on what is happening at the front lines.

3 GAME OBJECTS

Key components in the game are bases, the battlefield, armies (composed of troops) and messengers. Everything is located on the battlefield, with armies moving around the battlefield and messengers moving between armies and bases.

3.1 Battlefield

Each battlefield is constructed procedurally with deterministic methods from a given random seed. In practice this means that clients can do the battlefield constructing individually, while still ensuring that everyone plays on identical field. Locations for the players bases are also included in the battlefield, so that first, second and so on players are always placed in specific locations. Any battlefield can host anything between two and N players, where N is the number of slots generated, based on the size of the battlefield.

3.2 Bases

Bases are constructed locally by players in a separate editor and then serialized into an transmittable data format. Each player needs to receive base-information from every other player before the battle starts, because implementing the bases on a given battlefield might cause mutations in it.

3.3 Armies

Armies are composed of troops and they are built during the game. Troops are built at a players base with resources that accumulate over time. They are then formed into an army and sent out to the battlefield. Army formations and movement can be relayed to other players either by constructing the armies locally from transmitted player commands, or by providing server side book keeping of armies and their movements, which are then transmitted to clients. Both of these methods are further explained in the Service architecture chapter.

3.4 Messengers

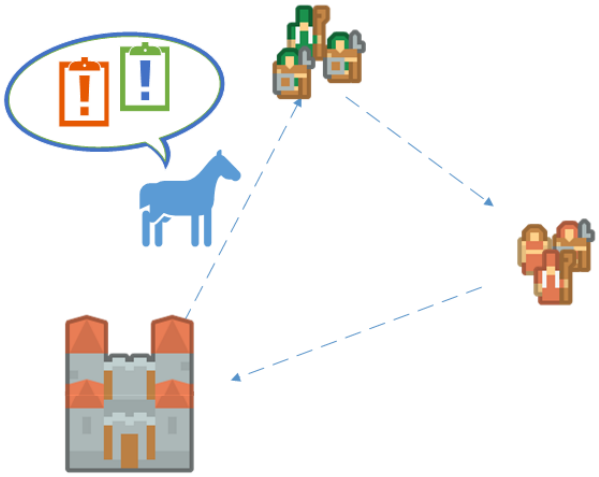

Figure 2: A messenger on a path to deliver new orders to two separate armies, before returning back to base

Messengers are a limited resource that can be upgraded and accumulated during the gameplay. They can also be lost, but losing a messenger only cuts the message delivery as the messenger itself respawns in players base after a short delay. This delay is based on the time it would've taken the messenger to complete the assigned route. Gameplay here is that the player can't be entirely sure the message got through until the time the messenger should've returned.

4 NETWORK DESIGN

4.1 Internal update rate

Generally server and client updates are measured in ticks. This means the frequency which with the game updates the gamestate. Client side things, such as unit animations or UI input can (and almost always do) run at a different pace from the internal tickrate.

In this game, the internal tickrate is 33.3 milliseconds - or 30 ticks per second. Player input however is only processed once per second, and queued for the next second. In practice this means that all player actions have an execution delay of up to two seconds. Reason for this lies in networking latency.

For example, if 0,2 seconds after game start a player activates a command, this command is marked as having been queue for the second (2) second of the game. Any timers related to the completion of the said action are based on the marked activation time. If the said action takes three (3) seconds to complete, every client would know it's ready at the fourth (4) second mark. This gives the game at least one full second to relay to command to all other clients, before anyone gets meaningfully out of sync.

From the perspective of the player that executes the command, it is triggered instantly. From the perspective of other players it's executed as soon as they receive the network package. If the action

was for example to give movement orders to an army, it might move slightly faster on the screen of a client that receives the command late, but in both views the action completes at the same exact time.

4.2 Data serialization

Each individual game command is serialized and recorded in a per-player fashion. A command object contains a player ID, per-player incremented serial number, timestamp and the actual action used. A game replay or spectating can be constructed from simply playing the commands in order in a client without player interaction.

Because the commands have an incremental serial number, it's easy to notice if one goes missing. In this case a client can ask for a repeat for any missing packages. The serial number increments both with time- and action component: 5 bits for actions and 16 bits for time (seconds). This means that up to 32 actions can be performed per player per second, for up to 18 hours of gametime. The action cap of 32 APS is chosen as it suits even the needs of most Starcraft professionals [2].

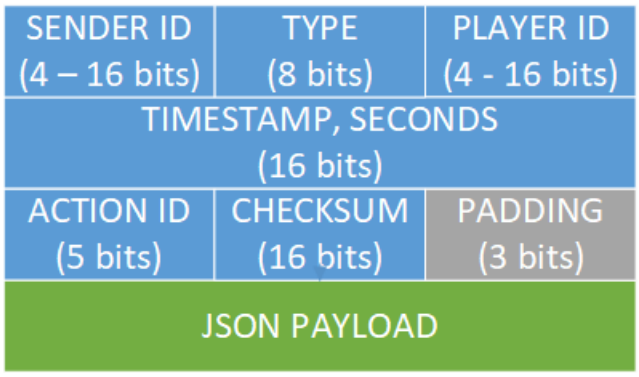

Figure 3: Anatomy of the game data packet, containing 10 bytes of header information and a JSON serialization of the action performed

The player ID is chosen per game session, and the number of players is locked at game start. Players can drop out of the game, but new players cannot enter a one that is in progress. The player ID can reserve 4, 8 or 16 bits, depending on the number of players entering the session.

Actual actions will be serialized into JSON, which is provides a suitably lightweight and easy to parse protocol for the payload.

Every client also transmits one keep-alive action per second. This action contains no JSON component, but has the action ID of last command from previous second. This information is used to both verify that the client is still online, and to ensure that all actions have been received.

4.3 Network structure

With peer-to-peer mode, each client needs to receive gameplay data from every other client. A fully connected mesh is not used, because in it the number of connections would grow quadratically with the number of nodes. Instead the game operates in a partial mesh topology, where clients forward actions from peer to their neighbours, and then relay the actions to other neighbourhoods.

A single neighbourhood is given a maximum capacity, and when the number of neighbourhoods reaches that capacity, a new tier of neighbourhoods is created. For example with a capacity of five, up to 25 players can play with a single neighbourhood tier, up to 125 with two tiers.

Every client maintains a list of all the other clients, their respective neighbourhoods and their connection addresses.

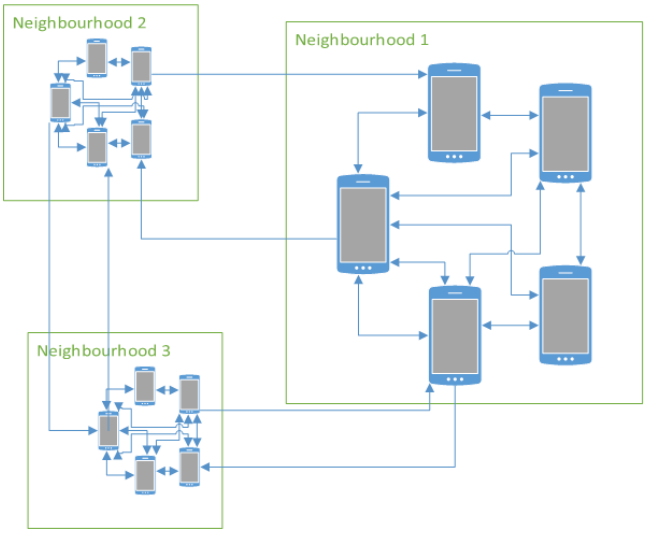

Figure 4: Three neighbourhoods of clients, connected together by selected clients

Because the real time gameplay is somewhat low paced, game can withstand reasonable latency without users noticing anything out of place.

Each client relays their actions to a neighbourhood with one to five peers, and because of the serialization and record keeping of player actions, the receiving end can simply discard duplicates. Whenever a client receives an action it already has a copy of, it replies with a message telling the data sender to stop providing that or those actions. While this means that every action is delivered at least twice, it also replaces the need for traditional ACK messages and sidesteps the retransmission ambiguity problem.

Each client also sends the actions of their neighbourhood to a peer of another neighbourhood, where applicable. The client receiving the actions of another neighbourhood handles distributing them locally. It is worth noting that one client can relay messages to and from several other neighbourhoods. This can happen if a neighbourhood has fewer clients than there are peer neighbourhoods.

5 JOINING AND LEAVING THE GAME

Each game starts with a lobby, where players register for a game. This is used as an opportunity to divide the peer clients into initial neighbourhoods and to synchronize client timestamps. While the game design doesn't support a player joining after the game has started, the technical implementation doesn't prevent it. If a game were to be modded to support players joining a game in progress,

the new player would simply need to receive the current action-stack from an existing player.

5.1 Disconnects

A player is assumed disconnected if no actions have been received for a set timeout period. This parameter can be modified, but should default to 5 - 10 seconds. A disconnect is broadcast forward to other peers, but it's initiated only by a client(s) that has been receiving direct messages from the disconnected peer.

A client that receives their own disconnect signal or otherwise detects disconnection can rejoin by a join action. The rejoining client provides the (game time) seconds from which forward they need the actions to catch up. If a crash happened, the client requests commands from time 0 onwards.

Once a client has been declared disconnected, the slot in the game is left open for anyone to join. In practice this lets the original player to join from another device if needed.

6 MODDING

Mods are player-generated content that alters gameplay in some fashion. A mod might for example change the way map is generated, introduce new unit types or change graphical layout of the game. Each mod is stored in an individual package and enabled separately for each game instance. A game can use multiple mods, although some mods might clash together with the mechanics they're modifying. Mods are built outside of the actual game engine.

Every client partaking in the same game session must have the same mods installed. This is done by hash validation, generated from each of the chosen mod packages.

6.1 Distributing mods

While mods can be downloaded and installed manually as individual files, they can also be distributed inside the game. Whenever a player or server starts a modded session, connecting clients can choose to receive the related mod packages.

The distribution is done via peer-to-peer transfer, using network coding for the distributed packages. While the network topology is largely known and stable, using network coding ensures faster package propagation when accounting for different clients having widely varying upload capabilities [1].

7 SUMMARY AND FURTHER DIRECTIONS

The provided game design enables scaling of players both inside a single game, as well as in the entire ecosystem. Because no centralized server structure is required, there's no limit to how many game sessions can run simultaneously. However while the amount of players in a single session theoretically limited only by the player ID (currently up to 16-bits), in practice the transmitting the synchronized game state via actions will eventually start to consume considerable bandwith. The peer-to-peer structure aims to alleviate this, but it will not provide infinite scaling.

Assuming 18 bytes of header (8 UDP 10 game action) and 50-100 bytes of JSON action data, a single clients actions will consume perhaps one or two KB/s bandwith. At maximum player count (65 000), this would mean up to a hundred megabytes per second of player action data. While compressing the JSON would help, the sheer amount of actions means just the packet headers will continue to be a problem after a certain point.

7.1 Further scaling

Extra scaling could be provided if clients were joined in neighbourhoods containing those peers they interact with directly. Having a single outside observer with full gamestate could direct the peer communications so that players only receive the actions of those peers that they have a chance of interacting with in near future. This would require certain amount of intelligence in predicting player actions and possible require the client to catch up on the actions of a newly encountered player every now and then.

Scheduling action data transfer to happen less often would cut the overhead from message headers, as well as provide a better opportunity to compress the action data. However the trade-off would mean slower detection of missed packets and disconnects.

Both of these options could be combined with good effects. Packing and transmitting "faraway" data less frequently, while being fast and precise with nearby players. Achieving this with decentralized network might prove challenging, but could be provided with an external coordination service, or a game server.

REFERENCES

[1] Christos Gkantsidis and Pablo Rodriguez Rodriguez. 2005. Network coding for large scale content distribution. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, Vol. 4. IEEE, 2235–2245.

[2] Andrew James Latham, Lucy LM Patston, and Lynette J Tippett. 2013. Just how expert are âĂIJexpertâĂİ video-game players? Assessing the experience and expertise of video-game players across âĂIJactionâĂİ video-game genres. *Frontiers in Psychology* 4 (2013), 941.