Assignment 1

Software Architekturen

SS2011

Matthias Steinböck, 0527943, 937

Table of contents

Setting 1
UseCases 1
Components 2
Deployment 3
Database 4
Architectural Decisions 5

Setting

Most games are about killing other player's troops. „Korpuleum" is about having fun (http://www.youtube.com/watch?v=MZSKFW_bhrg): The target is to make your troops life as awesome as possible. You have to collect wheat to bake pies, plastic to produce Balloons and so on. Fights are decided on how much fun a group of troops has.

UseCases

The core UseCases are:

- Register
- Log in/out
- Delete account
- Choose map
- Navigate map
- Build base
- Build building
- Train troop
- Form group

- Move troop/group
- Upgrade building
- Send message to other players

Components

The main components are accessible using a RESTful API. The API-dispatcher routes the requests to the relevant component. Most requests to components are self-contained in favour for the principle of loose coupling.

Deployment

The client (Web browser) loads all required static resources from the cache servers and if any resource is not yet statically available, has been deleted due to being dirty or is a request to dynamic data, the request is forwarded to the application servers. The application servers share a replicated database.

The requested uptime of 99.99% is reached at 4 nodes assuming a .1 failure rate probability: exp(.1, 4) = .0001 probability of all 4 nodes failing = .01% which results in 99.99% uptime.

Database

The database does not store every game dynamic directly at the troop or any other resource having dynamics-influencing properties. It stores that data in the dynamics table.

Architectural Decisions

The following questions have to be answered/issues have to be resolved:

#	Issue
I01	The system has to have an uptime of 99.99%
I02	Strong security mechanisms have to be applied to prevent cheating.
I03	Game dynamics and mechanics should be easily extensible concerning minimum downtime while updating.
I04	Game balancing has to be easily changeable per map

Following the argument:

Issue	#I01 - The system has to have an uptime of 99.99%
Decision	The hardware is divided into three layers: Database Servers, Application Servers and Static Servers.
Assumptions	We assume that the database layer replicates correctly and that the application layer synchronizes same objects to each other.
Positions	- Setting up 4 monster computers - Divide hardware into layers
Argument	This does not only distribute load but also helps tweaking the hardware to the specific need (fast hard disks on static servers, lots of ram on application and database servers). On every layer we have to guarantee an uptime of 99.99% therefore 4 nodes on every layer.
Implications	More servers have to be maintained

Issue	#I02 - Strong security mechanisms have to be applied to prevent cheating.
Decision	- After logging in communication uses SSL. - Logged in users have a maximum number of actions per minute. - Measurement and Logging are set up to detect abnormal behaviour
Assumptions	We assume that the application is save to all “normal” kinds of attacks (SQL injection, …)
Positions	- No special security measures - ssl, apm-restriction, ids
Argument	Most cheating attacks happen by session hijacking or using bots. Both SSL and APM-restriction should prevent those. If a clever hacker still finds a possibility to circumvent the measures logging and ids should detect those.
Implications	Additional effort has to be done creating an IDS.

Issue	#I03 and #I04 – Maintenance and extensibility of game dynamics and balancing
Decision	- Game dynamics are not hard coded into the source code - Game resource properties that are the same for all instances of a resource (like sadness points of a “cake thrower”-troop) are not stored with the resource but in a special table
Assumptions	We assume that the application does not need to be super extensible but within explicit borders. Those are: - The game is a 2d tile based web game - Troops can be moved on this map - Troops are used to fight for bases
Positions	- Reflect all resource-type-specific properties within a specific type-class - Save those properties in the database - Create special configuration files
Argument	It is nearly impossible to correctly predict correct values for fair balancing. As it is necessary to minimize downtime all influencing values have to be changeable without restarting any server or even updating source code. Therefore those values are saved in a central table.
Implications	Properties can have different types. Casting has to be done.