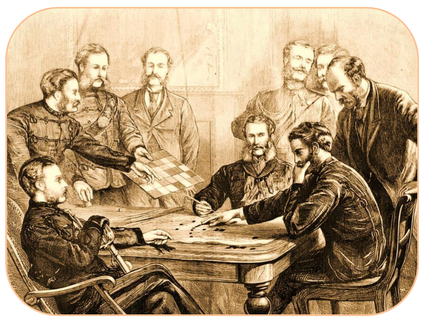
Design and Implementation of the JominiEngine:
Towards a historical Massively Multiplayer Online Role-Playing Game

Dave Bond (D.A.Bond@hw.ac.uk), Hans-Wolfgang Loidl (H.W.Loidl@hw.ac.uk) and Sandy Louchart (S.Louchart@gsa.ac.uk)
Heriot-Watt University — Glasgow School of Art

Code : http://www.macs.hw.ac.uk/~hwloidl/Projects/JominiEngine/

Context



The value of wargames and historical simulations as tools for the teaching of conflict simulation and decision-making has been recognised since Georg von Reisswitz first introduced his wargaming rules in the early 19th Century, and in the last half of the Twentieth Century their popularity as pastimes also grew markedly.

The arrival of computer gaming in the 1980's resulted in a significant increase in popularity for this type of game; in particular, with the advent of the Internet, the last two decades have seen a dramatic increase in the global player-base of MMORPGs, and the associated commercial rewards.

Challenges

The development of a historical MMORPG presents the developer with challenges in the areas of:

- ***Game design***, in which a balance has to be reached between historical accuracy and player enjoyment.
- ***System architecture***, which poses significant challenges with regard to system complexity and scalability.

Design Goals

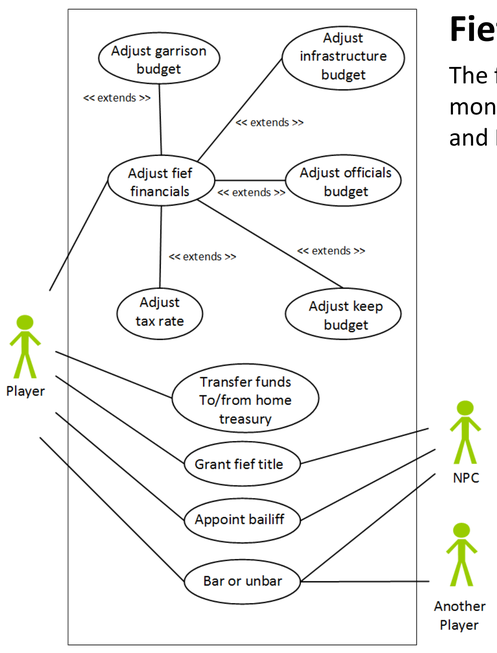
Goal: The JominiEngine is an emerging, distributed, scalable game engine for historical massive multiplayer online role-playing games (MMORPGs). Core game and system design principles of this engine are historical accuracy of the game model and scalability of the system to large numbers of players. The intended application domain is education in history, to provide an "*interactive history*" experience. Specifically, the engine has been instantiated to a concrete game, *Overlord: Age of Magna Carta*, a game set on mainland Britain in the time period of 1194-1225.

The implementation of the game engine focuses on modularity, extensibility and scalability, so that it can be instantiated for different time periods, and extended to also cover different application domains. We therefore view this game engine as a "*motherboard*" for developing educational tools with varying topic areas and learning objectives. Technically outstanding features of the implementation are the use of Riak as a non-SQL database and of C# as a programming language.

Key considerations of game and system design

1. Ensure ***historical accuracy***, to provide educational value and enhance player immersion.
2. Ensure ***modularity***, to allow for the future expansion of the *JominiEngine* in subsequent projects.
3. Address ***scalability***, ensuring the game supports high usage at an acceptable level of performance.

Game Model & System Architecture Model

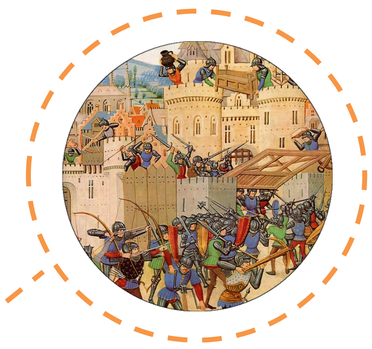


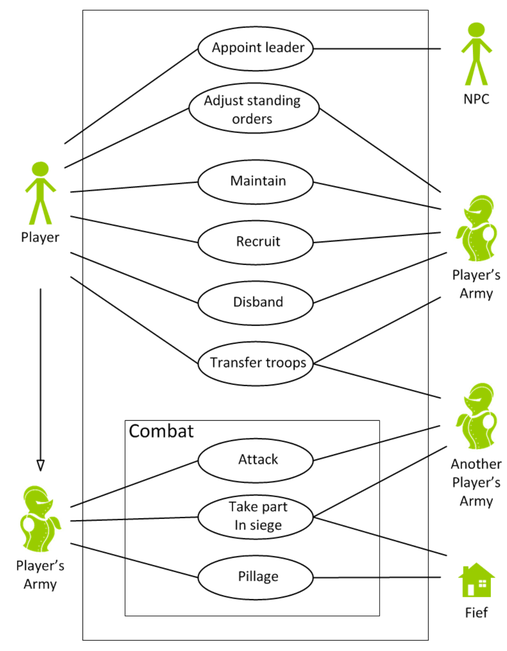
Fief management

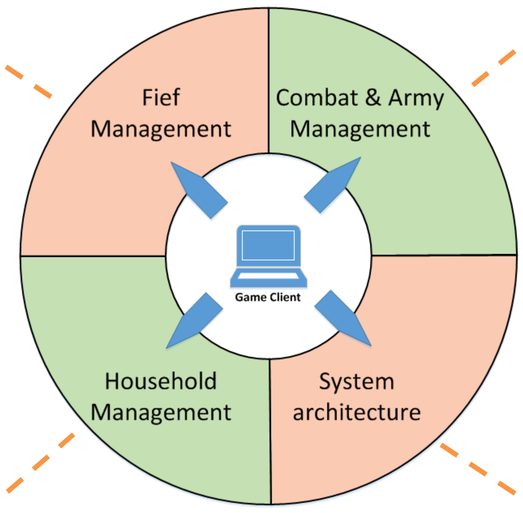
The fief is the historical unit of generating wealth, producing money and people, thus enabling the player to hire armies and NPCs.

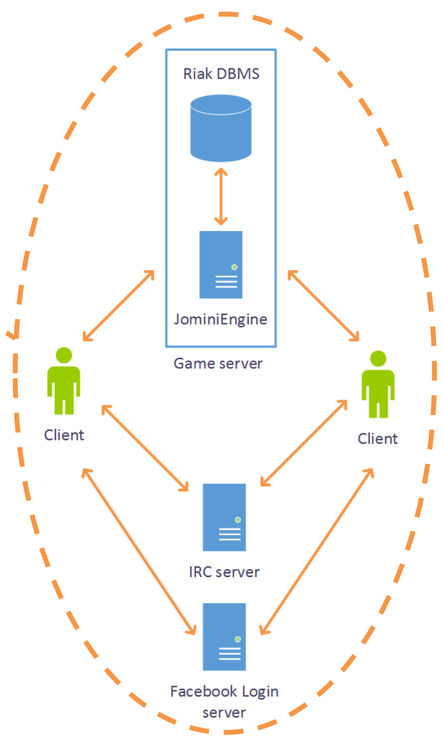
Army management

Armies provide both security for the player's fiefs and the means to 'acquire' new fiefs from his enemies.









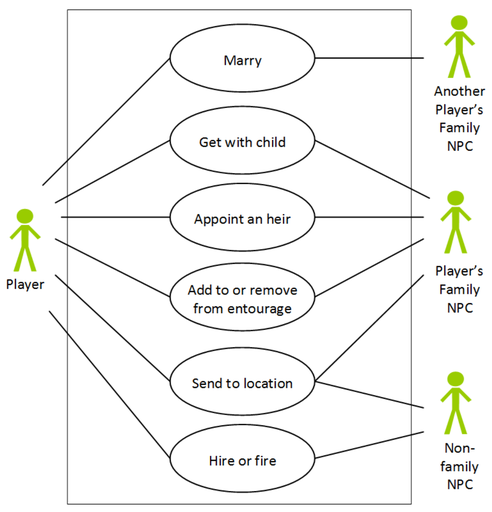
A basic ***client-server architecture*** was chosen due to: i) its simplicity to implement; ii) security (anti-cheating); iii) centralised control; iv) the ability to host the game on a dedicated server.

Riak was chosen as the ***backend database management system*** (details in next section).

It was decided to investigate the use of ***external services*** for operations such as inter-player communication and account functions, removing the need to embed them into the *JominiEngine*.

C# was chosen as the ***development language***. It boasts a well-developed catalogue of supporting libraries, and enables the integration of the game client into the *Unity* framework.

System architecture



To ensure his survival in the game, the player must marry and sire heirs. It is also advisable to obtain the services of talented NPCs.

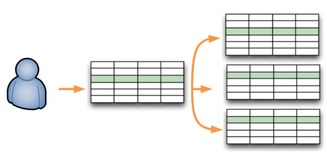
Household management

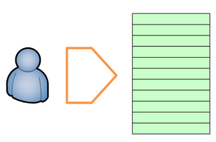
Backend DBMS: Why Riak?

Riak is a key-value, NoSQL (non-relational) DBMS. For an MMO game engine, its reduced reliability (NoSQL DBMS's are not ACID compliant) is more than compensated for by its inherent advantages:

Performance

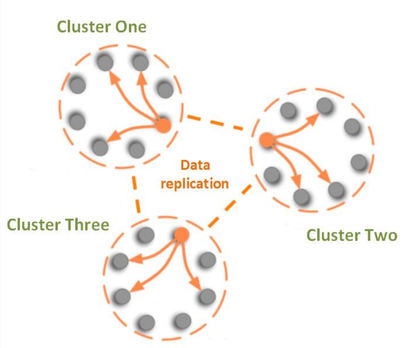
It is designed for the ***fast throughput*** of small operations, without the overheads associated with maintaining relational integrity. There is also less need for structural changes to the database due to modification of program code.





Distributed

It is distributed, which allows for ***scalability*** (by adding nodes and clusters), and ***reduced latency*** (by being located close to its user-base), whilst providing built-in ***redundancy*** in case of component failure.



Outcomes

State of the project

A ***game model*** was developed that stipulated basic class types, key game roles (king, herald, system administrator), component mechanics, rules and victory conditions. UML-style diagrams were used extensively, ensuring a well-documented basis for subsequent work.

A core ***game engine*** (the *JominiEngine*) was developed and implemented that allowed players to interact with the game world, as defined by the game model. ***Modularity*** was facilitated through the provision of thorough documentation and clear interfaces to game and system components, and existing protocols were used where possible. Formulas used in the game mechanics were, where possible, historically authentic and realistic, and ***historically accurate*** data was imported into the *JominiEngine* to allow *Overlord* to be fully instantiated.

A **Unity-embedded game client** has been produced as front-end, communicating with a game server on a separate machine.

A ***system architecture*** was developed, using the NoSQL, distributed DBMS, *Riak*, as the backend database. Such usage is relatively novel, and should help to address the issue of ***scalability*** through its ability to both sustain high performance, and to reduce latency through its distributed nature. The use of external services for operations such as inter-player communication and account functions, should also help with scalability.

Ongoing and future work

System architecture:

- The introduction of game content authoring, or 'modding' (for example, by using *Lua* scripting).
- Enhancements to the graphical game client to improve player experience.

Game model and engine:

- The expansion of existing mechanics; for example, the introduction of a more nuanced character model, expanding the role of women, or the introduction of sea movement.
- The introduction of new mechanics; for example, religion (a very important aspect of medieval life).
- The exploration of enhanced artificial intelligence for NPCs, increasing player immersion.