Habbo architecture - a brief history

Aapo Kyrölä

November 2008

Contents

1 Introduction

1.1 Overview of this Article

The purpose of this article is to concisely tell the technical history of the back-end of Habbo virtual world. It should not be regarded as a complete introduction to the system. I will outline the three major architectural generations of the server software and will also explain some of the technological challenges we faced and how we solved them. As the history spans from year 2000, the general development of computing and especially Java technology play a major role. As a personal note, I was only 21 years old when the first version of Habbo was launched, and had very limited experience (and no education!) of building multiuser systems. Thus, this is also a personal learning story.

Because of the historical nature of the article, I have decided to use the original diagrams instead of recreating them.

I assume the readers of this article have a good knowledge of internet and Java technologies.

1.2 About Habbo

Habbo (or Habbo Hotel) is one of the biggest virtual worlds in the market. Its main target group is teenagers. The setting is a virtual ”hotel”, which users enter with their *avatar* using a client software running inside a web browser (*Shockwave plugin* by Adobe is required). In the hotel, users can chat with other players in the public rooms or create a room of their own. They can buy virtual furniture - with real money - for the room and use it to socialize with their friends, play games etc. In addition, the system incorporates an instant messenger, own customizable web page, multiplayer games and more. As of November 2008, the service operates in 32 countries and receives about 10 million unique visitors per month. The service is profitable and funded mostly by end-user revenue.

The service was opened in August 2000, first in Finland and shortly in the UK and after 2002 rapidly expanded internationally. The company behind the service, Sulake Corporation was founded in May 2000 by the author and Sampo Karjalainen. The service can be accessed at: `http://www.habbo.com`.

1.3 My role

I was the leading developer of Habbo during years 2000-2004. For years 2004-2006 I had still a strong influence on the development, but did not work full-time in the project any more. Since 2006, I have been mostly focused on other projects than Habbo. As this article is part of my Ph.D. student applications, I will focus only in the developments of Habbo that I have played a major role in. The current state of the Habbo platform is not covered in the article, but major re-architecting has not happened since year 2006, so the picture has not totally changed. Currently I am a board member of the company and a leading developer of our new - soon to be launched - product.

2 General architecture: introduction

2.1 Client, Client-server communication

Users connect to Habbo Hotel using a client that runs in their web-browser with a popular plugin Macromedia Shockwave. The client connects to the server using a TCP/IP socket and communicates using a proprietary protocol. The clients connect only to a server, never directly to each other. There is separate server cluster for each of the country site.

2.2 Server platform

The server software is created with Sun's Java technology. For the virtual world, we do not use a third party application server. The accompanying website runs on top of a web application server and our payment processing server is based on Enterprise Java Beans (EJB).

In general, we use extensively third-party libraries, of which most are open source. The usage of external libraries and frameworks has significantly increased during the evolution of Habbo, because nowadays the quality and variety of them is much greater than back in year 2000.

Most important open source frameworks in use currently:

- Spring Framework for dependency management and injection
- Hibernate object-relational mapping framework for data persistence
- Apache ActiveMQ Java Messaging Service (JMS) system
- several Apache Commons libraries.

2.3 Database

Since the beginning, we have used MySQL (www.mysql.com), a free and powerful database engine. For the first few years, MySQL did not yet offer transaction management and only used table-level locking. While it was fast for many purposes, the scalability was poor. Nowadays, MySQL provides an efficient modern InnoDB table engine that provides row-level locking and transactions. As with many large systems, the database has always been the biggest bottleneck for us, because it is inherently difficult to distribute to smaller units. Application level caching has thus been one of the biggest development issues in Habbo. Caching has has been required for adequate performance but it has also made maintaining data consistency in a distributed system a major challenge. More about these topics below.

2.4 Hosting

Habbo servers are hosted from three hosting centers around the globe: Canada, Germany and Singapore. Good user experience requires low latency, which is the main reason the servers are located in distinct locations. In addition, we use a Content Delivery Network (CDN) to provide the client code and graphics efficiently to customers.

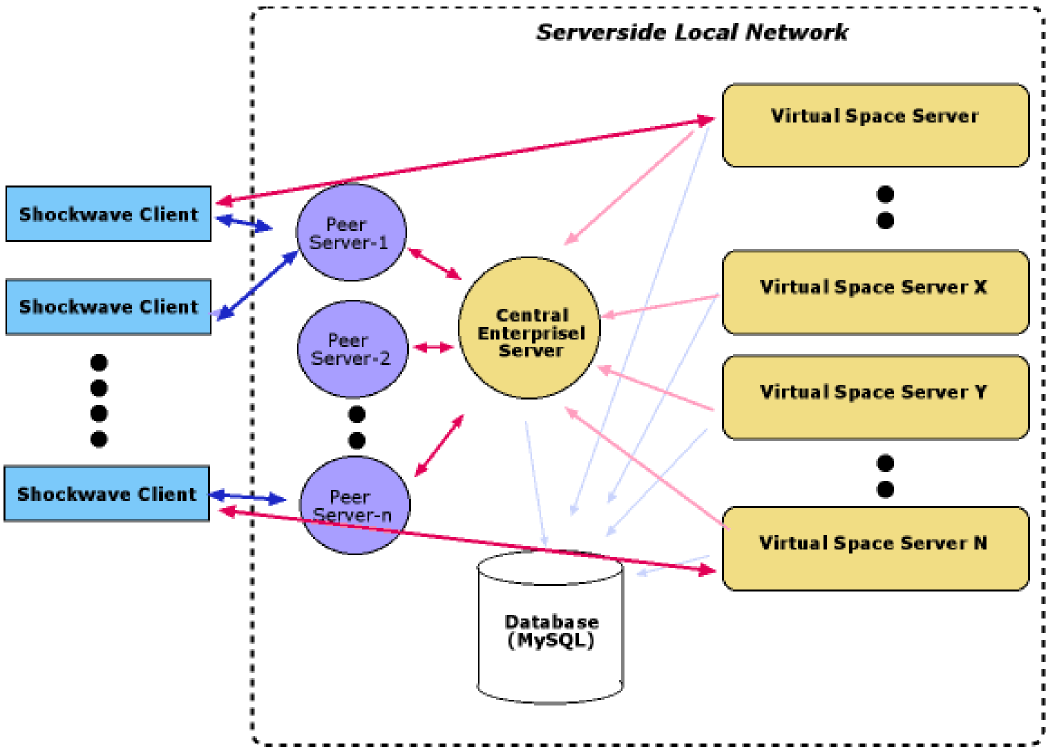
The server software is hosted on Linux and Solaris servers. One hosting center usually contains some tens of server machines of relatively affordable hardware. More expensive hardware is used for the database as it requires efficient disk arrays and backup systems.

2.5 Payment processing

Paying for content in Habbo is possible with over hundred different payment providers around the world. Most popular payment methods are premium mobile phone text messages, premium phone lines and credit cards. In addition we have redeem-code based scratch cards and bank transfers.

Payments require more reliability than the actual game. On the other hand, latency is not a critical issue in payment processing. For these reasons, we use Enterprise Java Beans -technology and JBoss application server for payment processing. Details are out of the scope of this article.

Figure 1: Habbo architecture: version 1 (11/2001)



3 First generation: 2000-2003

The first generation of Habbo architecture is shown in figure 1. Each of the server component is running its own Java Virtual Machine (JVM). As the naming of the components might sound a bit strange to a native English speaker, an explanation follows. *Central Enterprise Server* (CES) manages the shared state of the system. For example, Virtual Space Servers send constantly information about how many users are in the virtual spaces they manage. In addition, CES took care of instant messaging, friend list presence management and issuing of moderation commands. The *Peer servers* acted as front-ends to the CES. They handled requests from clients that were not related to the action in the virtual space. The messages of peer servers where request-response type, while the virtual space servers mediated messages between clients (users in same rooms) and managed shared game state.

3.1 Client connection

In this model, the client had always one or two socket connections to the server cluster open. Connection to a *peer server* was open during the whole session, and an active connection to a peer server meant that user was logged into the service. In addition, client made separate connection to a space server. Because of this, it needed to authenticate also separately to the space server. Also, if the peer server connection was broken for some reason (partial system reboot, for example), the connection to the space server could still be active. This occasionally led to broken sessions in the server.

When user used a *navigator* to find a room she wanted to enter, address to the virtual space server operating this room was provided by the peer server. Selection of the space server was simply based on the modulo of the room id: as long as all peer servers had consistent knowledge of the available space servers, user was forwarded to the correct space server. When a space server crashed (usually for Out of Memory or a Java VM bug), this information could become outdated and forwarding tables to the space servers inconsistent. These were big problems as it could lead to one room having two incarnations (runtime instances) running in two separate servers. In this kind of situation, maintaining room state was not reliable.

3.2 Communication between servers

Initially, communication between servers was implemented using Java Remote Method Invocation (RMI). RMI is a neat and simple way to implement remote method invocation. However, the simplicity was deceptive, as our light-hearted usage of RMI lead to serious scalability problems. There were two major issues: (1) RMI does not support asynchronous calls, even if the method call does not have a return value; (2) the level of concurrency is not limited, which leads to *thread starvation* under high load. The latter problem is explained in detail below.

3.3 Issue: Java RMI and threading

When a Java RMI client invokes a remote method on a RMI server, a new thread is created in the server to handle the request, unless there is an old thread in the pool available to handle the request. This is basically a good system, because it prevents slow method calls from blocking quicker calls to execute. However, there is a trap: if for some reason several requests are blocking (usually because they cause a database query to be executed and database might be temporarily under heavy load), the number of threads on the server may grow indefinitely. In the beginning of the decade, Java threading scalability was notoriously poor and having hundreds of threads - even if they were not doing anything - lead to serious resource starvation and running out of memory.

The unability to control the number of threads was a big problem for us, because in our architecture we had a singleton service Central Enterprise Server that was used by several clients (peer servers and the virtual space servers). The number of threads in the peers was equal to the number of client connections. Each client connection did RMI calls to the Central Enterprise Server and if the CES was blocking (due to database for example), the amount of pending requests in it could grow to equal the number of clients connected. As we had over thousands of simultaneous users already in the early days, this lead to frequent crashes under high load.

Solutions: a good but more complicated solution was to replace the RMI calls by a single socket connection between peers and enterprise servers (and virtual space servers). An object-stream protocol was used for the communication. This solution limited the number of threads in CES to the number of servers communicating with it. This also enabled asynchronous messaging as some of the calls did not require a response from the server. Another quick solution is to synchronize all the RMI-calls on the client side, but this does not bring the benefit of asynchronous calls. This was later on used with communication to a central cache server.

3.4 Issue: synchronous I/O and Java threading

Java did not offer asynchronous I/O before version 1.2, but for each connection, you were required to have one thread for reading and possibly another thread for writing to the socket. Synchronous I/O is more convenient for the programmer and can be more efficient in a fast network. But early Java virtual machines were also poor in threading, especially on Linux. Running over hundred threads slowed down the system considerably and over 500 threads was usually already too much. Splitting the system to more virtual machines, each handling less connection, is the obvious solution. However, the memory overhead of a Java VM is big and also managing big number of VMs can be difficult.

In addition, some clients could block indefinitely: if I recall correctly, a crashed Macintosh OS 9 did not close a socket connection and this in turn made server writing to this socket block indefinitely. Thus, a separate thread to watch for socket-writes taking too long was needed.

Solutions: there were external Java Native Interface (JNI) libraries available to enable asynchronous reading from sockets. This combined with a thread pool to handle incoming messages improved performance and scalability considerably. In general, thread pooling was and still is an important method to limit the number of concurrency in a natural way.

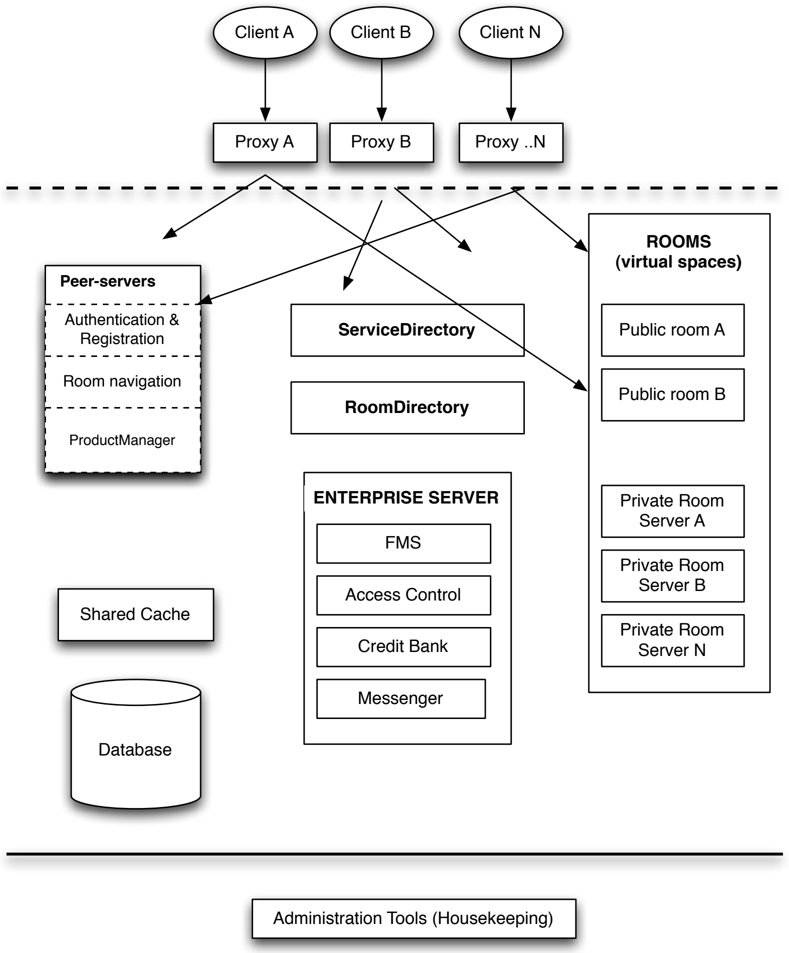
3.5 Final remarks

The first generation of the architecture was clearly inadequate in many ways. Lack of session management and proper management of the server components lead to many problems with data consistency and failure recovery. On the other hand, the model of separating game-like real-time functionality of the game servers from the request-and-response based functionality was good and is still valid in Habbo.

4 Second generation: 2003-2005

The next generation of Habbo server architecture took full advantage of the new asynchronous I/O of Java version 1.2 (NIO). Instead of clients directly connecting to the *peer services* and *virtual space servers*, a new component group *proxies* was introduced. It

Figure 2: Habbo architecture: version 2 (10/2002)



was now the sole responsibility of proxies to communicate with the client and mediate the client requests to the *subsystems*. A sketch of the architecture is shown in figure 2. This model has several immediate benefits, including:

- virtual space servers are isolated from slow I/O with clients
- user "session" is now bound to one connection, easier to manage
- proxies can be on different side of the backend firewall, this increases security
- allows rebooting of subsystems without disconnecting clients.

The new model allowed us to create subsystems that only did one thing, such as instant messaging, and then we could allocate resources to different functionality more precisely.

At the same time, our service usage grew dramatically, thanks to the quick expansion of broadband internet to households. Now our main worry became the database, which remained as a singleton service. This is discussed in the next subsection.

4.1 Database performance issues

Habbo is a highly dynamic service, where information changes and is created constantly. Like other Massively Multiplayer Online Game (MMOG) developers, the database has always been a major bottleneck for the scalability. One reason is that we were not able to anticipate the issues early enough. We should have designed the data model to be more readily *sharded* into several instances. I will now outline some of the challenges and how we moved to solve them.

Caching: caching data objects, such as "User" and "Room" on the application level reduced database load significantly. About 90 percent of the database reads were eliminated by introducing a *shared cache*. SharedCache functioned as a shared hashtable and was used with Java RMI. This system was good when the unique identifier of a data object was known, such as user id or name. For more complicated queries, database was necessary to be consulted. Caching in a distributed environment is problematic because of data consistency issues. If subsystem A is holding instance of user Foo and subsystem B then changes information of user Foo, how to tell subsystem A about the change? In this version of architecture, we did not solve this issue but instead had a policy that an object must be loaded again immediatelly before it is updated to database. This prevents misaligned data in most cases, but as we do not have distributed transactions (actually, at that time we did not even have database-level transactions), with bad luck, collisions still occur. Also programmer errors were common with the policy. Clearly, a policy for deciding which subsystem owns each object was needed. However, introducing such policy on a legacy system is very difficult!

Table level locking: at this time, MySQL still had not officially introduced InnoDB table engine or the version of InnoDB that time was too limited for us. We were thus bound to use MyISAM, the default table type of MySQL. MyISAM is very efficient for reading data, but it scales very poorly if the ratio of updates to reads is high. For us, this ratio was high, almost 50 percent. The problem with MyISAM is table level locking: when a row is updated in a table, no other reads or updates are able to proceed. For big tables, where updates are expensive, this becomes a real bottleneck. One way to solve this is to split a table to several sub-tables. For example our table for *friends* (user X is friend of user Y etc.) was split to ten sub-tables friends0, friends1, ..., friends9, where the modulo of ten of user id was used to select the table to use. The performance improvement was very big, but it was a compromise from the administration point of view. For example, now it was much more difficult to count how many rows of friends there were in total.

4.2 Messaging

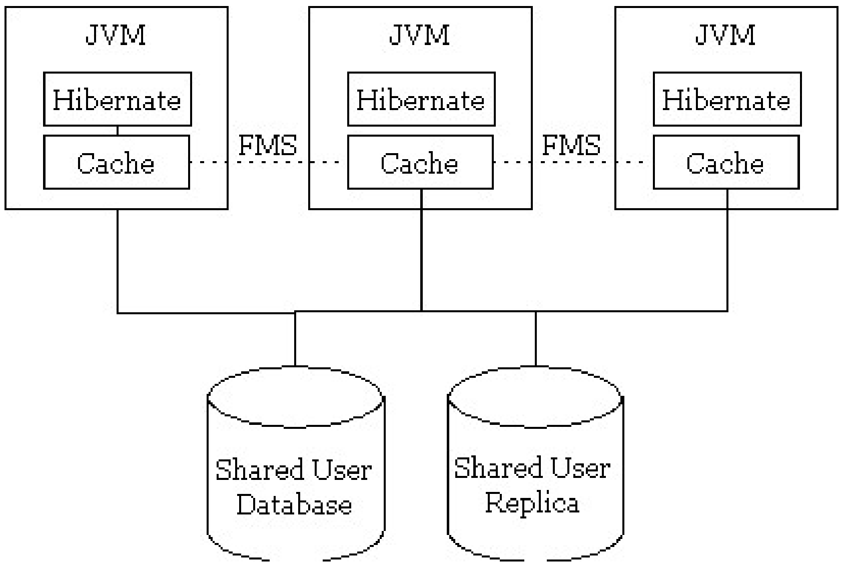
Java RMI, as explained above, is a synchronous method for communication between services. For many cases, asynchronous communication is more efficient and convenient. Also RMI does not support broadcasting, i.e sending a message to several recipients at once. This version of the architecture incorporated a system-wide topic-subscribe messaging system. It was implemented by a proprietary solution, Fuse Messaging System (FMS), written by myself, but after several years it was replaced by standard Java Messaging System (JMS). The decision to create own messaging at first was based on (in addition to my strong interest to write such a system!) the immaturity of the open source solutions available. Also, JMS had many features we did not need, so our own simpler solution was also more efficient.

I still remain a big fan of message-based architectures. They are easier to develop and more resilient to failures. In addition, they are often easier to administer as messaging counterparts do not need to know about each other, only about the message broker. On the other hand, synchronous remote method invocation is still sometimes needed, and RMI is well suited for that. We use both mechanisms in the system.

4.3 Final remarks

This version of our architecture was a very significant improvement. We were also greatly helped by the improved performance and reliability of the Java technology. Biggest issues that remained hard challenges were the database performance and data consistency issues. Along with success, we also got receiving more and more security attacks by hackers. They still remain a major challenge to our development. However, discussing security is not in the scope of this article.

Figure 3: Database object caching (04/2006)



5 Third generation: 2005-

The next big goal was to make the database layer more scalable. This was addressed on two fronts: on the one hand improving caching of data with a distributed caching architecture and on the other hand by using database replication to balance load. In addition, we started quickly adopting popular open source technologies such as Spring Framework and Hibernate persistence framework.

5.1 Database replication and caching

Figure 3 shows an outline of how our persistence layer worked. Each Java Virtual Machine (JVM) could access the database using the Hibernate persistence framework. To shield the database from repetitive queries of the same *data objects*, a Hibernate *second level cache* was employed. The cache is practically a hash table were data object class and identifier form the unique key. The obvious issue is of course how to guarantee that caches do not have stale data. This was solved with best-effort strategy: when a JVM updated a data object, it broadcasted a message to all another JVMs asking them to revoke their cached version of the object. In addition, Hibernate stores a version number for each database object in the database: if one tries to update a stale object (with a newer version than the current copy), an exception is thrown. In most cases, the exception can be handled by simply reloading the object and making the update again. This model is not perfect, but implementing real distributed transactions would cause far too much overhead.

Database replication can be used to share database loads between several database instances. One of the instances is the *master* that broadcasts all the updates to all the *slaves*. The slaves serve only database queries and all updates are directed to the *master*. For some time we used slave-master replication in the normal operations, but practically administering the system proved to be quite tedious because of bugs in the replication software for MySQL. Nowadays we use replication only with our statistics database: to protect the main database from heavy statistics queries, the statistics database acts a slave to the master database.

5.2 Move to open technologies

In the first half of this century, there were growing frustration in the industry with the Java Enterprise Edition (J2EE) platform. J2EE, especially Enterprise Java Beans (EJB) software was tedious and complicated to write and the persistence layer of EJB was very inefficient. This frustration resulted in the still continuing popularity of e.g. Spring Framework component framework and Hibernate persistence layer, in addition to many open source projects by the Apache community. These technologies also proved to be perfect match for the Habbo development.

Although we are nowadays increasingly satisfied with the technologies mentioned above, they did not come without problems. This again proved to me that it is very important to know how external software is implemented before adopting it with full force. For example, Hibernate had (and still has) many performance issues if it is used "blindly", especially regarding foreign references. This is one reason that open source software can be more safe option than commercial software: at least we are able to debug and fix the problems ourselves.

One big change we did was to move from our proprietary FMS messaging system to standard Java Messaging System. The change was driven by the team's urge to use standard technologies. But after the change, we still bumped on small and obscure, but severe bugs that could worsen the performance to unbearable levels and even crash the JMS broker. It is not always better to use third-party technology, just because it is widely adopted in the industry. This is certainly the case when one has exceptionally high requirements, like we have with Habbo.

6 Links

Spring Framework is a popular *inversion of control (IoC)* based component framework. More info at: `http://www.springframework.org`.

Hibernate is very popular persistence framework for Java and .NET. It provides powerful object-relational mapping and query capabilities and works very well together with Spring. `http://www.hibernate.org`.

MySQL is free (or almost free) relational database which is very popular in web applications. MySQL was acquired by Sun Microsystems Inc. in the beginning of year 2008. `http://www.mysql.com`.