IT 11 009
Examensarbete 15 hp
Mars 2011

Analysis and refactoring of the chat architecture in EVE Online

Philip Pettersson

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Analysis and refactoring of the chat architecture in EVE Online

Philip Pettersson

In most commercial software systems the development cycle is a balance between time and code quality. With systems that grow in complexity and usage over time, the need to refactor or completely replace modules of code might arise for a variety of reasons.

This thesis focuses on the feasibility of completely overhauling the aging chat architecture in EVE Online. A prototype was developed using Python, C and Ejabberd that could serve as a replacement system using the open XMPP protocol -- enabling bandwidth savings, increasing scalability and allowing the chat system to be easily extended with new functionality in the future.

Handledare: Christian Lönnholm
Ämnesgranskare: Olle Gällmo
Examinator: Anders Jansson
IT 11 009
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

This report constitutes part of a thesis submitted in partial fulfillment of the requirements for the degree Bachelor of Science in Computer Science from the Department of Information Technology at Uppsala University. The work presented herein has been carried out by Philip Pettersson on behalf of PikkoTekk AB, Uppsala, Sweden and by extension their contractor CCP hf, Reykjavík, Iceland.

1.1 Background

EVE Online is a computer game developed by CCP hf. It is a massive multiplayer online role-playing game *(MMORPG)* set in a sci-fi universe.

Since its release in 2003 it has grown significantly in code-size, with the 14th "expansion pack" (game add-on) arriving in November 2010, but has perhaps more interestingly also grown considerably in the number of active player subscriptions. The player base has seen consistent growth over the past decade with the current active subscriptions numbering over 300,000[1].

When considering market share, EVE Online is dwarfed by other MMORPGs such as Final Fantasy, Lineage, RuneScape and World of Warcraft[2]. There is, however, an important distinction to be made when comparing these games with EVE Online with regards to the world size that players interact with.

Games such as World of Warcraft actively employ server sharding[3] to spread out their player base over a large number of servers, typically housing a few thousand players on each server. These servers are for the most part completely independent of each other and in essence represent different game worlds with differing virtual economies, play styles and most importantly different player bases. In games that employ sharding it is therefore necessary to choose the same server shards as friends if one wishes to collaborate in the same game world as them. Taking World of Warcraft as an example, it is unlikely that any two players will reside in the same game world as they employ more than 200 separate game worlds (servers) in North America alone[4].

In contrast with games that employ server sharding to keep individual server load under manageable levels, EVE Online houses all of its players in the same game world. With a peak concurrent user mark of over 50,000 on

[1] From `http://www.eveonline.com/news.asp?a=single&nid=3044&tid=1`

[2] Market share statistics gathered from http://www.mmogchart.com/

[3] A type of shared-nothing-architecture in distributed computing, where each node is independent and does not communicate with other nodes.

[4] From `http://www.worldofwarcraft.com/info/faq/realms.html`

weekends this poses a great challenge in maintaining acceptable latency levels for the players.

1.2 Problem description

In most commercial software systems the development cycle is a balance between time and code quality[1]. With systems that grow in complexity and usage over time, the need to refactor or completely replace modules of code might arise for a variety of reasons[2].

In the case of EVE Online, there is an on-going effort to optimize parts of the software system to accommodate the ever-growing number of players. The work presented in this report is focused on the in-game chat system in EVE Online, dubbed *Large Scale Chat*. The goal has been to conduct a feasibility study to determine if the existing chat system can be completely replaced by an open standard, and more specifically what advantages or disadvantages a proposed new system would have.

An overview of the current chat implementation is presented in the next chapter, followed by a proposed prototype implementation of a completely rebuilt system using the open *XMPP*[5] standard.

2 Analysis

To analyze and assess a software system that is already in place, one must first make a decision about what properties to evaluate and base the overall assessment on. To enable comparison with the new prototype, three general criterion were considered:

- *Performance* — How efficiently CPU and network resources are utilized.
- *Scalability* — How well the system can cope with an increased number of users.
- *Extendability* — How complicated it is to extend the functionality with new features.

2.1 Python

Python is an interpreted, dynamic scripting language that has been in development since 1991. The design philosophy places great emphasis on code

[5]Extensible Messaging and Presence Protocol

readability while still allowing the use of multiple programming paradigms, such as object oriented, imperative and functional programming styles. With its comprehensive standard library, combined with a myriad of third-party libraries and strict coding guidelines, it enables rapid development without generally sacrificing readability or general code quality.

CCP uses Python extensively in both the EVE Online game client and server for these exact reasons. Using a dynamic scripting language allows flexible extension of the game code and in some cases hot-patching the game without requiring server restarts. This property, together with the increase in development speed when using scripting languages compared to low-level languages such as C, is understandably a desirable characteristic for a game development company that is constantly adding new features to their game.

While interpreted high-level languages like Python have their clear advantages in some aspects, such as increased developer productivity, by their interpreted nature they also do not perform their tasks with the same computational efficiency as low-level compiled languages in terms of processor and memory usage [3]. For data intensive applications this decrease in performance is often unacceptable, while applications that are largely I/O-bound might not spend enough time in CPU-bound code segments for the incurred overhead to have any essential impact.

2.2 LSC

The *Large Scale Chat* system in EVE Online is the central facilitator for real-time direct text-based communication between players.

2.2.1 Features

The LSC architecture is very similar to the *Internet Relay Chat* [4] standard. Similarly to *IRC*, LSC allows group communication in real-time discussion forums called *channels* but also allows one-to-one communication between individual players. There are three main types of "chat-rooms" or *channels*:

- *Conversation* — A private conversation between two players.
- *Conference* — A channel that is joined voluntarily and might have a specific theme, such as being a Korean or German-language channel.
- *Mandatory* — These channels are forcefully joined by the server for gameplay reasons and the player can not part from these channels.

Most types of channels provide real-time presence information about who the current participants of the channel are.

2.2.2 Gameplay mechanics

As LSC is part of the game world itself and involved in some of the gameplay mechanics, it is not strictly speaking exclusively a chat system.

A central part of EVE Online gameplay is the special-purpose *local channel*. To understand how it is so vital to EVE Online gameplay, it is worth briefly summarizing how players interact in the virtual world.

- Avatars are various spaceships that can be piloted by the player.
- A player is always located in a *solar system* and can move around in three dimensions within the system, visiting planets, asteroid belts, moons and other places of interest.
- There are over 5000 solar systems in the EVE Online game world, and most systems are designed to heavily encourage *Player-Versus-Player* gameplay.
- When a player moves from one solar system to another, she is automatically joined to a LSC channel that corresponds to the current solar system.

As LSC allows real-time presence information about the participants of a specific channel, the participant list for a local channel thereby identify other players who are located in the same solar system. This information can, and is, heavily used in EVE Online gameplay to determine if hostile players are present in the system to prepare adequate response measures and judge overall safety.

Another gameplay element of LSC is a ”conversation-fee” that players can choose for themselves. If player *alice* has chosen a fee of 5000 credits and player *bob* wishes to initiate a private conversation with player *alice*, *bob* first needs to deposit the fee before *alice* gets a conversation request. This is commonly used by players to lessen the amount of conversation requests they receive from unknown players, without blocking them completely.

2.2.3 RPC

EVE Online makes heavy use of *Remote Procedure Calls*. All function calls in the client that make some request to the game servers will use the custom

RPC architecture developed in-house. These can be synchronous or asynchronous depending on the specific calls involved. The client architecture is abstracted into compartmentalized local and remote *services*, all of which are independent but the interaction between services combine to provide all the internal functionality in the game client as well as interfaces for remote functionality.

Every service resides in its own thread to provide concurrency and in the case of remote services the local threads simply employ a flexible wrapper around the remote threads. From an implementation perspective, both local and remote services expose a similar interface and the programmer generally does not need to consider that some of the processing is done remotely. All services are registered as being either local or remote on initialization but their inter-process communication is otherwise seamless.
A typical local service is the `gameui` service that provides simple GUI operations to provide status or error information for the player. The following example shows how to create a modal message box and display it to the user in one line of code:

```
eve.LocalSvc("gameui").MessageBox("Hello World!", "Window title",
buttons=uix.OK, icon=triui.WARNING)
```

For this report, the most interesting remote service is naturally LSC. To illustrate the similarity in using local and remote services, the following line of code will send a message to a channel:

```
eve.RemoteSvc("LSC").SendMessage(channelID, message)
```

Behind this line is inevitably a chain of complex function calls that provide the desired functionality. As with any networked remote invocation method, the rough sequence of events is:

1. Client calls the procedure stub with appropriate parameters.
2. The procedure stub encapsulates the parameters in a *marshalled* data packet which serializes the representation of the procedure call.
3. The client sends the marshalled message to the server.
4. The server uses *unmarshalling* to extract the encapsulated request parameters.
5. The server calls the procedure specified in the request with the appropriate parameters.

6. The result is passed on to the client in the same manner, simply changing directions.

The specific RPC architecture in EVE Online is called *machoNet*. It provides all the functionality one would reasonably expect from a mature RPC library, including exception handling, addressing, authentication and encryption. Spanning over 6000 lines of code, a substantial size when considering the terseness of Python code, it is not within the scope or in line with the goal of this feasibility study to analyze the efficiency of the RPC system itself. Instead, we will see how possible overhead is encountered when specifically using the LSC service from the client, with regards to the criterion proposed in the beginning of section 2.

2.2.4 Tracing

For analyzing the chain of function calls that are fired when a remote procedure call is made, code-path tracing was used in conjunction with code review. In a complex system like the EVE Online client it can be daunting to follow the path of execution by looking at code alone, therefore the python `sys` module was used. This module contains a variety of functions associated with manipulating the operating environment of the python interpreter itself, and in particular the `settrace` function can be used to set a global debug tracing function. The function that is provided will then be called on each function call.

Below is a brief example of how one can use this feature to trace execution. Consider a short python program with two functions:

```
def f(x): return x**2
def g(x): return f(x+10)
```

And a simple tracing function that will print debugging information:

```
def trace(frame, event, arg):
    if event == "call" and frame.f_code.co_name not in "<module>":
        code = frame.f_code
        print "Function %r was called" % code.co_name
        arg_values = []
        for i in code.co_varnames[:code.co_argcount]:
            try: arg_values.append("%s=%s"%(i, repr(frame.f_locals[i])))
            except (NameError, KeyError, AttributeError): pass
        print "Arguments: %r" % arg_values
```

After having specified `trace(frame, event, arg)` as the tracing function for `sys.settrace`, the following callstack will be displayed when invoking the function `g(x)`:

```
>>> g(15)
Function 'g' was called
Arguments: ['x=15']
Function 'f' was called
Arguments: ['x=25']
625
```

2.2.5 Sending a message

When sending a short message to a channel containing the word "Hello", a myriad of functions are called to process and handle the corresponding RPC request. Below is a rough callstack:

1. `RemoteSvc`
2. `LocalSvc`
3. `ConnectToService`
4. `StartService`
5. `GetService`
6. `StartServiceAndWaitForRunningState`
7. `StartService`
8. `WaitForServiceObjectRunningState`
9. `Masquerade`
10. `ConnectToService`
11. `ConnectToRemoteService`
12. `RemoteServiceCallWithoutTheStars`
13. `_GetRemoteServiceCallVariables`
14. `GetService`
15. `CallDown`

16. `ToPickle` – The packet is now *pickled*, which is a type of python serialization.
17. `CallDown`
18. `_BlockingCall`
19. `_GetTransports`
20. `MachoDumps` – The packet is now serialized again, with a different layout using the python *marshal* serialization.
21. `SymmetricEncryption` – The packet is encrypted from a C++ DLL using an unknown algorithm.
22. `Write` – The packet is ultimately written to the network, it has now grown to **259** bytes.

As this chain of function calls illustrates, the callstack is quite substantial for such a seemingly simple task — sending a short word to a chat channel.

2.3 Conclusion

2.3.1 Performance

While it might be possible to overlook and ignore this inefficiency in the client, there are still situations where all available processing power should be used for more pressing tasks to maximize game performance. For example, consider a system where a large battle involving hundreds of players is taking place. As there is no priority-based scheduling between micro-threads in the client, it is possible for players to ”spam” the local chat channel to cause deliberate slowdowns in game clients for opponents since all threads have the same priority. These clients are already extremely busy with more important tasks in terms of gameplay experience; physics simulations, graphical effects and sound.

It is important to note that while the performance issues involved in decrypting and unmarshalling the message data are not very serious in the client, the same operations are being carried out by the game servers at a much larger scale, which will then impact game performance as a whole.

2.3.2 Scalability

Some channel types allow delayed presence information, which means that the list of active participants in a channel is only updated when someone

sends a message. Participants who only listen in on the conversation will not have their presence known to other players — this is acceptable for most conversation types but *not* acceptable for the local channel type because of its integration in gameplay mechanics. This effectively forces the local channels to immediately send updates to all members about anyone leaving or joining the channel, which means they cannot benefit from the bandwidth savings associated with delayed presence information.

As with any MMORPG, there are some places in EVE Online where many players congregate. These can be areas where a large conflict is taking place or locations where inter-player trading is commonly conducted. By simply going to one of these solar systems, without being in direct proximity to any other players or otherwise interacting with them, the player still receives many presence information updates per second. As shown in section 2.2.5, the bandwidth overhead incurred by each RPC call or result can then add up with growing channel sizes. Players who connect with mobile broadband or other generally high-cost connections in terms of bandwidth likely want to minimize the increase in data usage that this entails.

On the server side the RPC overhead is also more apparent than in the client, for the simple reason that the hierarchical topology means that the server maintains connections with all clients. For instance, when a new RPC request to send a message to a channel is received, the server must send out asynchronous events to all participants of the channel in question. The overhead associated with each packet is then multiplied by the number of participants and bandwidth is wasted.

Correspondence with CCP has revealed that this is a very real problem for them, as significant parts of their bandwidth and processing power are consumed by processing and routing chat packets.

2.3.3 Extendability

As LSC uses the *machoNet* architecture to handle all network traffic it is intimately woven into the game code of EVE Online itself. All communication over *machoNet* is routed by the game servers and there is no elegant way to implement functionality such as Peer-to-Peer traffic for file transfers. As the RPC library requires an active EVE Online client environment to function, it is impossible to integrate LSC in other pieces of software or otherwise access the chat system from outside the game client.

3 Prototype

The prototype implementation is divided into four interconnected parts that will be described in the following sections. The server-side code is an installation of *ejabberd* running on a Linux server. The client-side code consists of three parts:

- Dynamic run-time python code injector – places the prototype code in the running client.
- Backend jabber interface – provides an interface to the remote jabber server.
- User interface – provides a chat window just like the normal *LSC* implementation in the client.

3.1 Ejabberd

Ejabberd stands for **E**rlang **Jabber D**aemon. As the name implies, it is written in the functional *erlang* language, known for its high scalability. It is a free software implementation of an XMPP [6] standard server distributed under the GNU General Public License, and has a number of high-profile advocates including Facebook. The main draws about ejabberd for this specific prototype are:

- Supports distributed computing by clustering.
- Extendable with custom database authentication modules.

Being able to seamlessly distribute the chat load over several nodes is obviously a very desirable feature compared to the static layout of the current LSC system. Furthermore, as the EVE Online servers already contain all the relevant authorization information and access control lists (channel access, ignore lists etc.), being able to create custom modules for fetching this information from any database type greatly simplifies a transition from LSC to ejabberd on the backend side.

This, together with the fact that ejabberd can be run on a variety of different operating systems, made it the XMPP server with the most desirable characteristics for this planned chat system replacement. Two other jabber servers which fit the requirements of an open source license and platform-independence were also considered but rejected:

- Openfire[6] – Clustering requires a proprietary extension. More focus on ease of installation and maintenance than performance.
- Prosody[7] – Written in Lua, which inherently lowers performance due to it being a scripting language. Does not support clustering.

The XMPP protocol itself was chosen because of its high maturity, open standard and extensible design. It supports equivalent functionality of all the chat features used in LSC through the extension described in [7].

3.2 Dynamic run-time execution of arbitrary code

Development of the prototype was impeded somewhat by the fact that for contractual reasons, access to a proper development environment with full source code, documentation and compile scripts was made impossible.

As the game client does integrity checks on all python bytecode that it loads during initialization, it was not possible to simply drop some addon scripts into the game client file system structure. This meant that the prototype code had to be ”injected” dynamically at run-time into the EVE client process. This was achieved by making a C application that creates a new thread in the process context of the client, which then calls the relevant python functions to load the prototype code into the global scope and execute it. The rough execution outline is as follows:

1. Read in the plain python code and store it in memory
2. Get the process ID for the EVE client process
3. Attach to the process object
4. Allocate a remote thread
5. Populate the remote thread with code
6. Allocate and populate parameters
7. Start the remote thread

   Execution is now taking place in the EVE client process

8. Get a handle for the python DLL
9. Get a handle for

[6]Official website: http://www.igniterealtime.org/
[7]Official website: http://prosody.im/

(a) `PyRun_SimpleString` – Runs a python string in the global interpreter context

(b) `PyGILState_Ensure` – Acquires the python Global Interpreter Lock (*GIL*)

(c) `PyGILState_Release` – Releases the Global Interpreter Lock

10. Acquire the *GIL*

11. Execute the python code

12. Release the *GIL*

The python code has now been injected into the client process and started. This method and other code-injection techniques are described in [5].

3.3 xmpppy

Xmpppy is an XMPP library based on jabberpy[8], which was the first publicly released python jabber library. Development on jabberpy ceased in 2003. Both xmpppy and jabberpy are distributed under the GNU General Public License.

The backend code of the prototype uses a slightly modified version of xmpppy to provide the underlying network communication with the ejabberd server.

The backend code connects to the ejabberd server using a numeric username that is fetched directly from the EVE client, corresponding to the player's user id. When connecting, it specifies a resource name to identify the jabber client being used, in this case EVE. This will enable identification of chat users that are not actually present in the game, an advantage which will be discussed in more detail in a later section. It then joins a test channel and registers callbacks for presence and message information. These will later be instructed to pass the data to the GUI module for user presentation.

3.3.1 Modification

The EVE game client uses stackless python, a modified implementation of CPython that has support for lightweight microthreads called *tasklets*. The client uses these tasklets extensively in many different modules to provide concurrency.

In stackless python there is no support for the common socket operation *select*, which is available in many languages including C and standard python.

[8] http://jabberpy.sourceforge.net/

select enables blocking on one or more socket descriptors until one of them is ready to be written to or read from. The file `xmpp/transports.py` handles the low-level implementations of transports for xmpp-stanzas and uses *select* in one function, mostly as a convenience for custom socket timeouts since it only blocks on one socket descriptor. To remove the dependency on *select* the code was simply rewritten without it which added only a few lines.

Another modification that had to be made was to remove all references to the `socket.ssl` functions which provide built-in seamless SSL encryption for python sockets, since this functionality is also lacking in stackless python.

3.4 GUI

Creating a graphical user interface can be time-consuming if all layout is created through code. The time consumed is naturally exacerbated if the only documentation is sample code, which was the case when constructing the GUI for the prototype.

The prototype GUI consists of a window within the EVE client that almost completely mimics the layout of the normal chat window, which is essentially:

- Message window where new messages are displayed
- User list with the current participants
- Input field for typing

The message window simply formats the incoming messages and displays a timestamp, sender and message. The user list displays each player present in the room and also a special icon next to the player if they are connected via a different, out-of-game client.

The LSC chat can be seen in Fig. 1, with the message window and a user list to identify the participants of the conversation.

Figure 1: A screenshot of the normal LSC chat window.

Figure 2: A screenshot of the prototype jabber chat window.

As can be seen, the jabber chat in Fig. 2 is largely identical to the LSC chat window. Next to one of the chat participants a globe icon can be seen that illustrates that the person is in fact connected from outside the EVE universe, in this case using a free XMPP client called *Pidgin*.

One of the main advantages of using XMPP over LSC is the ability to create bridges between the game world and other entities, be it an instant messaging client as illustrated in Fig. 3 or a separate game world altogether. This will be discussed in section 4.3.

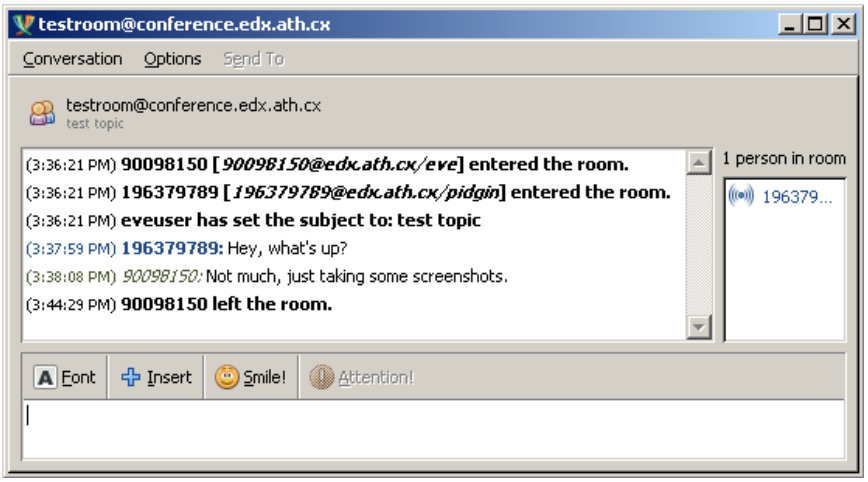

Figure 3: A screenshot of the exchange as viewed from Pidgin.

3.5 Functionality

As shown, the functionality of the prototype jabber chat implementation is largely identical to the LSC chat. The only difference would be the special case of the local channel, where the game must ensure that people are joined to their appropriate local channel at all times. This is easy to accomplish for LSC since it exists on the game server itself and will know the game state of all players at all times. However, the same functionality could be accomplished with ejabberd by simply having it poll the game servers for player location data to determine which channel to force-join them to. It could also be pushed to the ejabberd server by the game servers when state changes occur.

4 Results

4.1 Performance

The current LSC system is inefficient, but this does not pose a great problem for the game client itself since chat is a small part of the computational load on each player's client. The inefficiency becomes a problem on the game servers since seemingly small inefficiencies build up when processing is required for hundreds of thousands of messages. The clients only need to wrap their own message packets and unwrap the packets they receive. The game servers need to unwrap every message in the whole game universe, determine who the recipients are, and then send out new packets to the appropriate players. As LSC exists on the servers as a game service like any

other and is using the same intricate RPC architecture as more advanced game features, it is stealing important processing power that could be better suited for more complex problems than routing chat messages.

LSC:

- Written in a comparatively slow scripting language, Python (including the underlying RPC architecture)
- Not distributed
- Can not be moved off the game servers

Ejabberd was designed from the foundation to be a high-speed jabber implementation that performs the task of routing chat messages between users with minimal overhead and maximum efficiency. Particularly its distributed, fault-tolerant nature makes it a superior alternative to LSC.

Ejabberd:

- Written in a fast language with very high-speed message passing, Erlang
- Distributed
- Can be moved away from the game servers, even to different data centers

4.2 Network

As shown in section 2.2.5, the packet that goes over the wire when a 5-byte message is to be sent will have a size of 259 bytes, excluding IP and TCP headers, which will naturally vary depending on sender and recipient. Even considering it includes some addressing information about the appropriate channel for the message, such a large size is hardly acceptable. It is also worth noting that presence information packets, which simply inform whether a person joined or left a room, will occupy over 250 bytes each because of the overhead associated with the machoNet RPC architecture.

An XMPP message packet with a 5-byte message will have a total data size of about 140 bytes, and the presence information packets will have a data size of around 100 bytes. The overhead in these packets largely stem from the verbose nature of XML, however, since the structure of XML is text it is a prime candidate for data compression which is default when using XMPP over SSL/TSL. TSL[9] is supported in xmpppy, but the library

[9]Transport Layer Security, a cryptographic protocol

uses underlying socket functions that are not present in the stackless python environment that the EVE client uses. The library could however be further extended to support this without using the built-in SSL/TSL functions of python.

4.3 Extendability

The main advantage that a jabber chat platform has over the LSC system is the possibility of "breaking out" of the game world and communicating with people who are not present in the game. A player could log into the chat system and give orders to his comrades by simply using a free instant messaging client or his mobile phone. With XMPP, the chat system could also easily be added to the official web portal *EVE Gate*, which already has support for reading and sending in-game mail with a web browser. This is a great boon to a world like EVE where constant communication is such a vital aspect of gameplay. In the prototype client, users who are not connected via the EVE client will get a globe icon next to their portrait to signify that they are not actually in the game.

CCP has a planned action game in development called *Dust 514* which will take place in the same setting as the EVE Universe, where players in EVE Online will be able to give orders to players of Dust 514. Using an XMPP chat system will allow a unified communication platform for both games, much like Blizzard Entertainment's RealID service allows players of Starcraft 2 to chat with players of World of Warcraft.

5 Conclusion

It is the author's conclusion that this feasibility study has shown that replacing LSC with a Jabber/XMPP system is indeed possible and not prohibitively complicated. The development costs would be small since the only work that needs to be done in order to achieve the final switch is:

- Improve the prototype to meet quality assurance criterion
- Add graphical features to the prototype to make it a complete clone of the LSC equivalent
- Create custom modules for ejabberd to pull data from the game database servers about authorization credentials and access control lists

It would furthermore be possible to extend the XMPP client in EVE to allow connections to any XMPP-compatible service, such as Google Talk or Facebook Chat. This would allow communicating with friends who do not play the game without constantly switching focus to a third party instant messaging client.

XMPP is a highly modular specification and there are extensions that support file transfer, voice chat and video conferencing. By moving onto the XMPP platform, such features could be integrated into the game in the future with the underlying network architecture already being in place.

6 References

[1] William S. Junk, *The Dynamic Balance Between Cost, Schedule, Features, and Quality in Software Development Projects*. Computer Science Department, College of Engineering, University of Idaho, Moscow, ID, USA, 2000.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Boston, MA, USA, 1st Edition, 1999.

[3] Christopher Mueller, Andrew Lumsdaine, *Runtime Synthesis of High-Performance Code from Scripting Languages*. Open Systems Laboratory, Indiana University, Bloomington, IN, USA, 2006.

[4] J. Oikarinen, D. Reed, *Internet Relay Chat Protocol*. IETF RFC 1459, May 1993. Available at: `http://www.ietf.org/rfc/rfc1459.txt`

[5] Jeffrey Richter, Christophe Nasarre, *Windows via C/C++*. Microsoft Press, Redmond, WA, USA, 5th Edition, 2008.

[6] P. Saint-Andre, Ed. Jabber Software Foundation, *Extensible Messaging and Presence Protocol (XMPP): Core*. IETF RFC 3920, October 2004. Available at: `http://www.ietf.org/rfc/rfc3920.txt`

[7] P. Saint-Andre, Ed. Jabber Software Foundation, *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*. IETF RFC 3921, October 2004. Available at: `http://www.ietf.org/rfc/rfc3921.txt`