Orc Protocol Parser and Generator

Facilitating communication via the Orc protocol

TOBIAS ERIKSSON

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2007

COS/CCS 2007-17

Kungliga Tekniska Högskolan
Royal Institute of Technology

Master Thesis Report Draft
Final Version
Date: 2007-04-25

Orc Protocol Parser and Generator

Facilitating communication via the Orc protocol

Master Thesis Report

Tobias Eriksson
<toe@kth.se>

Examiner and academic supervisor
Professor Gerald Q. Maguire Jr., School of Information and Communication Technology

Supervisor
Peter Eriksson, Orc Software

Abstract

This master thesis project took place at Orc Software. This company provides technology for advanced trading, market making, and brokerage. The Orc System is based on a client/server architecture. The ordinary way to communicate with the Orc Server System is via the Orc Client Applications, such as Orc Trader or Orc Broker. Additionally, there is another way to communicate with the Orc Server System without using an Orc Client Application. There is a service within the Orc Server System which provides an interface for communication with the Orc Server System. Clients can communicate via this interface using the Orc Protocol (OP).

Banks and brokers usually have different systems that are specialized for different needs. Often there is a need to integrate these systems with the Orc Server. In order to simplify the integration for customers with modest programming experience in TCP/IP and parsing techniques, Orc Software would like to provide an example parser/generator capable of communication with the Orc Server System free of charge.

This thesis introduces a toolkit consisting of a parser/generator and a sample application. The application provides several examples as well as serves as verification to the customers of how simple it is to develop their own applications by utilizing the different OP messages.

A comparison was made between the newly created OP parser/generator and a manually generated FIX client using the FIX gateway which ORC Software AB also sells. This evaluation shows that OP parser/generator is both faster and less memory demanding than the manually generated FIX client.

Sammanfattning

Det här examensarbetet är utfört på Orc Software, som utvecklar system för avancerad handel, market making samt mäkleri. Detta system är baserat på en klient/server arkitektur. Normalt sker kommunikationen med Orc Servern via Orc klient applikationer som Orc Trader eller Orc Broker. Men det finns även ytterligare ett sätt att kommunicera med Orc Servern utan att använda Orc klient applikationer. Det finns en tjänst i Orc Servern som tillhandahåller ett gränssnitt som går att kommunicera med genom att använda Orc Protocol (OP) meddelanden.

Banker och mäklare har vanligtvis flera olika system som alla är specialiserade för olika behov. Detta gör att det ofta finns ett behov att integrera dessa system med Orc Servern. För att kunna underlätta integrationen för kunder med låga kunskaper i TCP/IP och parsing teknik, vill Orc Software tillhandahålla en gratis parser/genererare som kan kommunicera med Orc Server Systemet.

Examensarbetet introducerar ett paket innehållande en parser/genererare och ett exempelprogram. Programmet visar ett par exempel samt fungerar som bekräftelse på hur enkelt det kan vara att utveckla ett eget program som använder sig av del olika OP meddelanden.

Avslutningsvis presenteras en utvärderingsstudie mellan den utvecklade parser/generator och en manuellt genererad FIX klient som använder en FIX gateway som Orc Software också säljer. Utvärderingen visar att parser/genereraren är både snabbare och använder mindre minne än FIX klienten.

Acknowledgements

I would like to thank my examiner and advisor Professor Gerald Q. Maguire Jr. at the Royal Institute of Technology for his advice and valuable ideas. I have really appreciated the short response time and the correction of my English.

Moreover, I would also like to thank Peter Eriksson at Orc Software for giving me the opportunity to work on this project and for all the support he gave me during my work. Many thanks to Thomas Erlandsson who also gave me feedback and really helped me progress in my work.

Table of Contents

1 Introduction

This section outlines the thesis, first giving some background and then defining the specific problem which is to be addressed.

1.1 Background

This thesis project is being conducted at Orc Software. The company provides technology for advanced trading, market making, and brokerage.

1.1.1 Basic financial terms

Below are some financial concepts that will be helpful to understand before reading this report.

Bond	is a dept that the issuer owes the holder and, depending on type, has to pay interest and at maturity repay. [1]
Broker	a member of an exchange that is allowed to sell or buy **financial instrument** according to the orders from the customer on that particular **market**.
Cash instrument	The value of these instruments is set on the **market**. It can be securities such as **corporate stocks**, **mutual funds**, or **bonds**.
Corporate stocks	The Company issues **shares** to the **market** in order to raise capital. Each shareholder owns a share of the company. The size of the share is related to the number of stocks owned.
Derivative instrument	The value of these instruments is derived from the underlying asset. Market participants make an agreement to make an exchange (money, assets, or another value) in the future for an underlying asset. Common derivate instruments are **futures**, **forward**, **options**, and **swaps**.
Exchange	is the opposite of **over-the-counter** and can, for example, be a stock exchange (trading stocks). A corporation provides a place where **brokers** and **traders** can **trade**.
Financial instrument	is a legal agreement that involves an economic value. There are two sort of instrument: **cash instrument** and **derivate instruments**.
Forward	A forward is a contract between two actors. For example, one part agrees to deliver X tons wheat in November at a price Y per ton, whereas the other part agrees to buy the

	wheat at that price. Forward contracts are often traded **Over-the-Counter**. [2]
Future	is almost the same as a **forward** contract except the contract is revalued on a day-to-day basis (**Marking to Market**). The future contracts are often traded on an **Exchange**.
Market	is a place allowing buyers and sellers to share information and perform exchanges of goods or services. Three special markets are the stock market, bond market, and commodities markets.
Marking to Market	This means that an instrument is revalued to reflect the current market conditions.
Mutual fund	is a collective instrument where many investors together, managed by a fund manager, invests money in.
Option	An option is either a call or a put option. The buyer of a call option has the option to purchase the underlying asset at a particular date in the future at a specific price. A person buying a put option, on the other hand, has the option to sell the underlying asset.
Order	An order is instructions to a **broker** from a customer wanting to buy or sell something on an **exchange**.
Over-the-counter	this means that **financial instruments** are directly traded between two parties.
Position	is a commitment to buy or sell a given amount of securities or commodities. A market position means that the commitment is put on the **market**.
Portfolio	when someone owns, for example, a number of stocks it is said that he/she has a portfolio of stocks. This is often done to lower the risk involving owing only one instrument.
Quote	often refers to the latest price of a security traded. It could also be a price which is being offered for a trade.
Swap	When two parties agree to exchange cash flows in the future, they are said to make a swap agreement. For example, one party changes its variable income to a fixed

	one and the other party will get a variable income according a prearrange formula.
Trade	is exchanging of goods or services on a **market**.
Trader	is someone buying or selling **financial instrument** in the financial **market**.

1.1.2 The Orc System

The Orc System [3] is based on a client/server architecture. The Orc Server System consists of a number of components, including a database. This system supplies market connectivity to the Orc trading applications, this means that clients can use this application to access market data and conduct transactions in this market. The ordinary way to communicate with the Orc Server System is via the Orc Client Applications, such as Orc Trader or Orc Broker. A simple model of the Orc System, where an Orc Server provides an Orc client with a connection to a market gateway, is illustrated in the figure below.

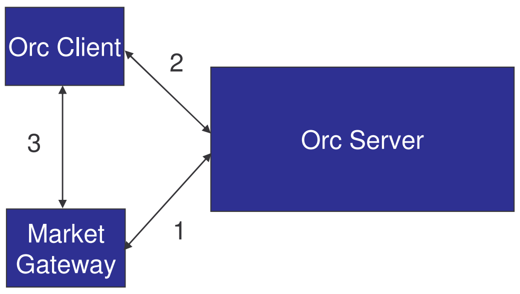


Figure 1 - A simple model of the Orc System.

First, the Market gateway registers itself in the Environment Management Daemon (EMD) and the Port-Mapper daemon (PMD) on the Orc Server. When an Orc client wants to connect to a Market Gateway, it makes a request to the Orc Server which provides the appropriate information. Finally, the Orc Client connects directly to the Market Gateway.

In addition to the Orc Client Application, banks and brokers usually have different systems that are specialized for different needs. Often there is a need to integrate these systems so that they can share the same view of, for example, the market or a position. Earlier, some customers wrote their own application in order to directly communicate with the Orc Server System's database (called CDS), e.g. making low-level access to the database. However, this can, for example, cause problems when there is an upgrade to the Orc Server System.

Fortunately, there is another way to communicate with the Orc Server System without using an Orc Client Application. There is a service within the Orc Server System which provides an interface for communication with the Orc Server System. Clients can communicate via this interface using the Orc Protocol (OP) [4]. Thus applications are able to access the Orc Server System's functionalities such as: enter orders, perform instrument downloads, get positions from a portfolio, and perform theoretical calculations on instruments. This is sketched in the figure below.

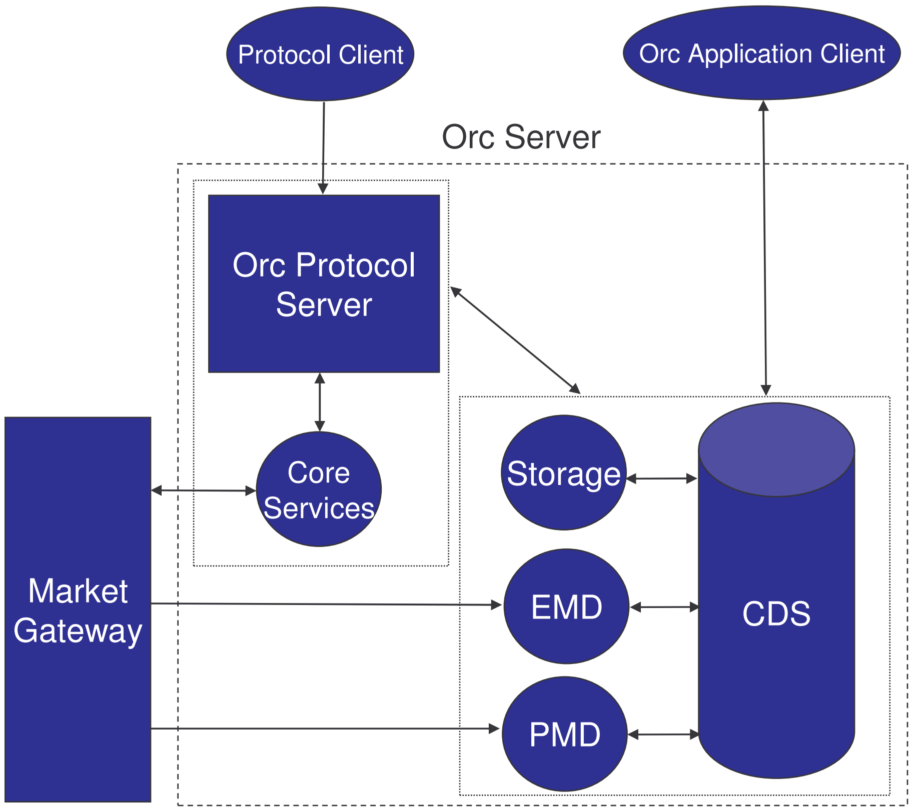


Figure 2 - An overview of the Orc Protocol Interface and its relationship to the other components of the Orc Server System

1.2 Problem Description

Some customers who want to interact with the Orc System using OP find it difficult to create their own parser and generator for this protocol. Thus the integration needed by the customer must typically be done using a consultancy or with support from Orc Software. In order to simplify the integration for customers with modest programming experience in TCP/IP and parsing techniques, Orc Software would like to provide an example parser/generator capable of communication with the Orc Server System free of charge.

Orc Software wants to provide a toolkit consisting of a parser/generator and a sample application. The application should provide a few examples as well as serve as verification to the customers of how simple it is to develop their own applications by utilizing the different OP messages. Therefore, it is important that the parser/generator and the example application are easy for the customer to understand. Orc Software hopes to reduce the perceived difficulty in communicating with their Orc Server System and thus attract additional customers and to encourage their existing customers to more tightly integrate with their system.

1.3 Tasks within the project

The first task was to implement an Orc Protocol parser in Java to enable external programs to communicate with the Orc Protocol. As the Orc Protocol is the gateway to the Orc System. The specifications for this parser are:

- It should be well documented
- The source will be delivered to customers
- Thread safe
- Fast
- Small memory footprint
- The communication should be TCP/IP based
- It should use the Java Message Service interface (see Section 2.2.3)

The second task was to implement an Orc Protocol generator. Since many applications will need to send data to the Orc system there is a need for a generator too.

The third task was to create an example application with some Orc protocol messages using the Orc Protocol Parser and Generator.

The final task was to contrast and compare the created Orc Protocol parser and generator to a FIX client using the FIX (Financial Information eXchange) gateway which Orc Software AB also sells. Some of the metrics that are relevant are performance and functionality.

2 Overview

This section provides the necessary background to understand both the problems described in section 1.2 and to understand the technology which will be used to solve these problems. It will start by describing how parsing works, along with the functionalities and interfaces to the various subsystems. In sections 2.3 and 2.4 two techniques to create parsers will be illustrated. In order to write an OP parser it is necessary to understand how the Orc Protocol works, this will be described in section 2.5.

2.1 Parsing

According to Steven John Metsker [5] there are two ways to separate human interaction from computers; humans work with text and computers with objects. To close this gap, between the human and the computers, we use parsers. For each language that is to be used to interact with the computer, a parser is needed to create a human interface that can transform this language into objects which the computer can understand.

Parsing or syntactic analysis is a well established part of computer science. It is used in several different sub-disciplines: compiler construction, database interfaces, self-describing databases, and artificial intelligence. Parsing is generally used to process a linear representation according a given grammar (here we will exclude parsing of two dimensional or higher dimensional representations such as images and volumes and will strictly focus on linear textual representations). This is a broad definition that gives room for various interpretations. For example, a linear representation can be a sentence, a computer program, or a sequence of financial data. The grammar, on the other hand, controls the relationships between the elements. [6]

There are many different languages, such as simple data languages, queries, logical, and imperative languages, all attempting to make it easy for the user to interact with the computer. Today a popular programming language is Java. Such a language is convenient when building programs to give the computer direct commands. However, when programming Web pages other languages such as HTML (Hypertext Markup Language) and XML (Extensible Markup Language) are more suitable.

As mentioned above, one area that parsers are used in is the compiling process. To explain how a parser work, I will describe the part of the compiling process where parsing is involved.

2.1.1 Compiling process

A compiler translates a program from one language into another. [7] To be able to successfully complete that task the compiler first needs to understand the structure and meaning of the source program. The figure below illustrates a simple model of the compiling process.

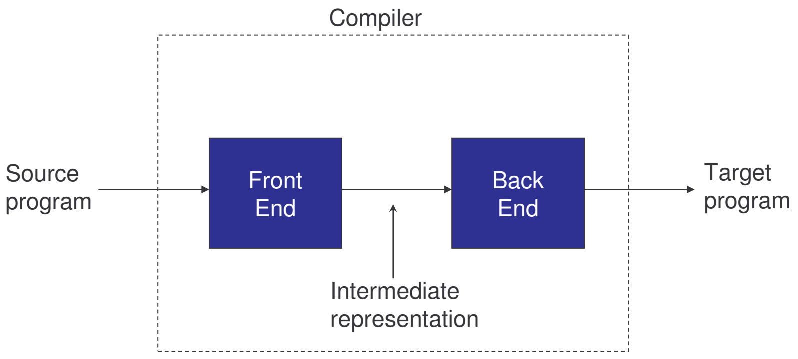


Figure 3 – A simple model of the compiling process

2.1.1.1 Front End

The front end consists of a **Lexical analysis**, **Syntax analysis**, and a **Semantic analysis**. This is shown in the figure below.

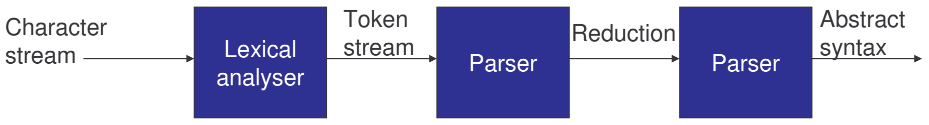


Figure 4 - Outline of the stages in the Front End of a compiler

A language is made up of strings that each has a finite number of symbols. The symbols are from a finite alphabet. In lexical analysis, the compiler breaks the stream of characters into words known as lexical tokens. After this is done the Syntax analysis begins to parse the phrase structure of the program. Lastly, Semantic analysis is done and this is where the actual meaning of the program is considered.

2.1.1.1.1 Lexical Analysis

A lexical token is a unit in the grammar of a language. Examples of tokens in a language could be ID, NUM, and REAL. Some units are simpler than others to represent. For example, punctuation marks like colons, semi-colons, commas, parentheses, and square brackets can be represented using their unique character representations. We also have keywords like IF and THEN that have a unique spelling.

Other tokens may have very complicated representations. For example, as will be explained in section 2.5, a value in the Orc Protocol can be an integer, a float, a string, a date, a time, or a dictionary depending on the key. The dictionary, in turn, can have one or more key/value combinations.

Consequently, there is a need for rules capable of producing all possible combinations of lexical tokens. A powerful notation to specify these lexical rules is regular expressions. Each regular expression often matches several different strings. There are rules of how to describe a regular expression that matches certain strings and thus, a token. Some of these rules are listed in the table below:

Table 1- Rules for Regular Expressions

A | B	Alternation, choosing from A or B.
A · B	Concatenation, an A followed by an B.
AB	An implicit way to indicate concatenation.
A *	Repetition, zero or more times.
A +	Repetition, one or more times.
A ?	Optional, zero or one occurrence of A.
[a-zA-Z]	Character set alternation.

To describe a token one can combine the different rules and create a regular expression that matches the strings that you want.

Table 2 - Illustrates how a Token is described using Regular Expressions

Token	**Regular Expression**
IF	if
ID	[a – z] [a – z 0 – 9] *
NUM	[0 – 9] +
REAL	([0 – 9] + "." [0 – 9] *) | ([0 – 9] * "." [0 – 9] +)

In the above table the token ID has to start with a letter and then zero of more letters or numbers. The token NUM consists of one of more numbers from 0 to 9. Looking at the token REAL it can be a decimal number greater than 1 or a decimal number less than 1.

One important aspect to consider when writing a regular expression is ambiguity. In this case, ambiguity means that some strings could match more than one token. The regular expression above is ambiguous because strings like if8 can be an **IF** token followed by an NUM token, or only an **ID** token. Therefore, there is a need for other rules for a lexical analyzer to be able to choose the right token for a string. The most used disambiguation rules used today are Longest match and Rule priority. Longest match means the longest initial substring of the input that can match any regular expression specified is chosen as the next token. Rule priority on the other hand means that when the longest match is found the lexical analyzer checks in the list of lexical tokens for the first regular expression that can match. For this reason, it is important to carefully write the regular-expression rules.

Regular expressions are suitable for specifying lexical tokens, but when they are to be implemented by a computer we need to be more formal. This is where Finite Automata comes in, as they provide a way to describe regular expressions using different states. As the name implies the description is an automata which has a finite set of states. The states are connected using edges. These edges are labeled with a symbol and lead from one state to another as can be seen in the figure below. The figure shows the lexical token **ID** where state 1 is the start state and state 2 is the final state.

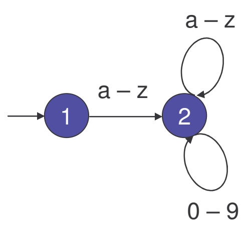


Figure 5 - Finite Automaton for the Regular Expression describing the Token ID

There are two different classes of automatons: Deterministic Finite Automaton (DFA) and Non Deterministic Finite Automaton (NFA). The difference is that the NFA can have two (or more) edges going from one state to another labeled with the same symbol while a DFA can not.

To construct a DFA or an NFA by hand requires hard work. Therefore, a lexical-analyzer generator such as JavaCC or SableCC, both written in Java, can be used. Both take a regular expression with tokens as an input and produce a lexical-analyzer program. These two programs can also be used as a parser generator where the input is a context-free grammar and the output is a parser program. JavaCC will be described in more detailed in section 2.3.

2.1.1.1.2 Syntax Analysis

A traditional notation for specifying syntactical structure is a context-free (CF) grammar. This is used in the next step in the compiler process, Syntax analysis. To be able to parse a language described by a CF grammar we need something more powerful than finite automata. Why this is the case can be explained by the following example:

Let us start by defining the lexical tokens digits and sum with the help of regular expressions.

digits = [0 – 9] +
sum = (digits "+") + digits

In this example a sum could for example be 67+39+7. However, if we wanted to define an expression like (67+ (39+7)), where the parentheses are balanced, it wouldn't be practical to use finite automaton. Finite automaton requires one additional state per open parentheses. Below is an example grammar of the second expression:

digits = [0 – 9] +
sum = expr "+" expr
expr = "(" sum ")" | digits

Thus, a parenthesis-nesting with the depth N requires the memory to able handle N states.

In this stage, the symbols are seen by the parser as lexical tokens and the alphabet is the different token-types that are set by the lexical analyzer. A CF grammar describes the language based on productions that consists of symbols. A symbol can be either terminal or nonterminal.

A formal description of a grammar G is [8]:

G = (T, NT, S, P) where,

T – represents terminal symbols, or words, in the language. Terminal symbols are the basic units of grammatical sentences. For example, in the expression above, the terminal symbols would be digit and sum, thus words revealed in the lexical analysis.

NT – stands for non-terminal symbols, or syntactic variables. These appear in the rules of the grammar and are made up of all the symbols in the rules except those already expressed using a terminal.

S – is the start symbol and a member of NT . If we want to derive a sentence from G it must begin with S.

P – corresponds to a set of productions. For example, P: NT → (T, NT), thus the rules of P decide the syntactic structure of the grammar. To make sure the grammar is context-free we can only allow one non-terminal on the left hand side.

This is illustrated in Figure 6. As the figure shows, a terminal symbol is a token from the alphabet of strings and belongs on the right side of a production. The non-terminal symbol on the other hand, is instead on the left hand side in a production.

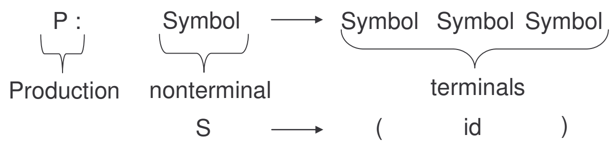


Figure 6 - Showing the differences between non-terminal and terminal symbols.

There are many different algorithms to parse a grammar file. These algorithms are often divided into deterministic top-down methods and deterministic bottom-up methods. [6] A deterministic parser has the property that there is always only one possibility to choose from. This means the grammar's right-hand side starts with a terminal symbol. If this is true, a predict step will always be followed by a match step. These parsers are faster than non-deterministic ones that have to search. However, there is one drawback, the number of grammars that can be handled using this method are fewer and more limited.

2.1.1.1.3 Deterministic top-down methods

The algorithm LL(1) is a deterministic top-down method. LL(1) means the parser operates from Left to right, produce a Left-most derivation, and uses a look-ahead of one symbol. Writing this in mathematical symbols a context-free grammar is called LL(1) when:

(1) A -> $\alpha_1, \alpha_2, ..., \alpha_n$

(2) B = FIRST(α_1 x#), FIRST(α_2 x#), …, FIRST(α_n x#)

(3) $\bigcap B = 0$

A is in this case a non-terminal with right-hand side $\alpha_1, \alpha_2, ..., \alpha_n$. The second line states that the sets FIRST(α_1 x#), FIRST(α_2 x#), …, FIRST(α_n x#) belongs to B. The third line means that any prediction for Ax# has no symbol that is a member of more than one set.

It is possible to expand LL(1) to any finite look-ahead, LL(k). Having a greater look-ahead makes the method more powerful, but also not as fast.

2.1.1.1.4 Deterministic bottom-up methods

One bottom-up algorithm is called LR(k), and it can delay the decision to choose the production until all input tokens corresponding to the complete right-hand side of a production have been found. LR(k) stands for left-to-right parse, rightmost-derivation, k-token lookahead. The k indicates that it can propone the decision k input tokens further than the production itself. This means, for example, that bottom-up parsers can’t have semantic actions at the beginning of a production.

To work properly a bottom-up parser needs a stack and an input. Using shift and reduce the parser steps through the input. Below is an easy example showing how the parser works:

Input: X Y Z
Grammar rule: A -> X Y Z

Shift: Move the first input X to the top of the stack. X can not match any production so we put Y and Z onto the stack as well.

Reduce: Pick the grammar rule A -> X Y Z by popping Z, Y, X from the top of the stack and then push A onto the stack.

Postponing the decision until the last moment makes this method more powerful than the top-down method. However, this makes it slower as a modest grammar might require hundreds of thousands or even millions of states making the parsing table huge.

Therefore, an algorithm called LALR(1) [9] was introduced which stands for Look Ahead LR(1). This algorithm decreases the parser table by merging two states with identical items but which differ in lookahead sets. LALR(1) parsing is the most-used parsing method today.

Although it is possible to build a parser by hand the most convenient way is to use a parser-generator program to build the parser. For example, a program like JavaCC can be applied. JavaCC uses the parsing algorithm LL(1) and will be described in detail in chapter 2.3.

2.1.1.1.5 Semantic analysis

In most situations it is not enough for the parser to just recognize whether a sentence belongs to the language of a grammar or not. There is a need for a mechanism that can make use of the phrases that are parsed. This is where the next stage in the compiler process called semantic analysis comes into action.

A first step is often to create a parse tree which is a structure that can be used later. In a parse tree the input tokens are leaves of the tree and the grammar rules are internal nodes. In JavaCC there is a tool called Java Tree Builder that automatically creates such a tree. If a tree should be constructed or if it is better to use an event driven approach will be discussed more in chapter 2.2.

2.1.1.2 Back End

The next steps (producing assembler and machine code) in the compilation process have no real connection to this master thesis and will therefore not be described more in detail.

2.2 Interfaces

Before starting to develop the Orc protocol parser generator it is important to consider how it should use the information received from the Orc System and what is the nature of the sample application. Should it use the idée behind DOM, i.e. creating a tree of objects of the whole message, or should it use the event handling of SAX?

Therefore, this section will begin by introducing both SAX and DOM. Moreover, this section will discuss whether the Java Message Service (JMS) interface could be useful for the Orc protocol parser generator to use for its implementation.

2.2.1 DOM

The Document Object Model (DOM) [10] is an application programming interface (API) for HTML and XML documents. The DOM interface was initially a specification to make JavaScript scripts and Java programs possible to use in different Web browsers. The DOM working group consisted of people from World Wide Web Consortium (W3C) [11] and other vendors working with related technologies.

The DOM interface reads the entire XML document into the memory as a tree representation. This makes it possible for an application implementing the DOM interface to access any element in the tree to change, delete, or add information.

In the case of the parser generator, this would make an object of every key/value combination and dictionary in the Orc Protocol (for a description of this protocol see section 2.5). When the entire, or at least most, of the information received from the Orc Server System is needed, this approach would be most desirable.

However, one has to consider that DOM is memory demanding and time intense when the input is large. Most of the customers using the Orc System have multi-market access which means that there are a lot of instruments available. For example, the reply from an instrument_download message from the Saxess market (5 000 contracts) is about 10.5 MByte. There are customers that have access to more than 600 000 contracts, which means almost 1.3 GByte to process. Today memory is very inexpensive, so that might not be a limited factor. However, it is time consuming to process such huge chunks of data.

2.2.2 SAX

The Simple API for XML [12] (SAX) is used for parsing XML documents using events. Instead of saving the entire document in a tree, as the DOM interface, the SAX interface uses event handlers the parser can use for reporting information. It is then up to the application to handle these SAX events, e.g. choose what information is relevant and should be saved as an object. SAX is today a SourceForge product and the current version is SAX 2.0.1 [13].

If the parser generator should utilize SAX it would need an efficient way to be able to fetch the important parts of the Orc Protocol message, e.g. Message Type, without having to read through the entire message. Otherwise one might just as well use DOM, because the speed and memory efficiency of SAX can not be used. As the Orc Protocol is currently implemented there is no way to figure out where certain parts of the message reside without parsing the entire message. The reason is that the order of the key/value combinations can be changed by Orc Software without notice. For further information about the Orc Protocol please see section 2.5.

2.2.3 Java Message Service

One possibility could be to use the Java Message Service (JMS) [14] interface for the ORC protocol parser generator. The JMS API is a message standard that makes it possible for different software applications or components to communicate. One limitation is that the software has to be based on the Java 2 Platform, Enterprise Edition (J2EE) in order to be able to create, send, receive, and read messages. Messages can be either synchronous, using the receive method, or asynchronous, using a message listener. Basic tasks for a JMS application are creating a connection and a session, creating a message consumer and producer, and sending and receiving messages.

A JMS application is made up of a **JMS provider**, **JMS clients**, **Messages**, **Administered objects**, and **Native clients**.

JMS Provider	is a messaging system providing administrative and control features.

JMS clients	are the one producing or consuming the messages.
Messages	are the object that carries the information the JMS clients communicates.
Administered objects	are either destination or connection factories.
Native clients	are programs that do not use the JMS interface but the native API.

The figure below shows the JMS API architecture.

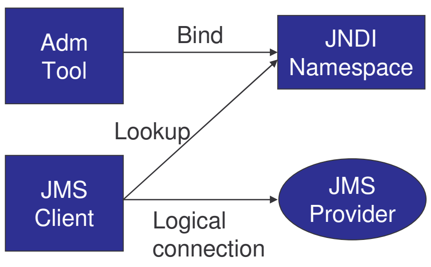


Figure 7 - The architecture of the JMS API

First of all you have to bind, using the administrative tools, destinations and connection factories into a Java Naming and Directory Interface (JNDI) API [15] namespace. The JNDI API provides directory and naming functionality to a Java application. This means any Java application using the JNDI can access a range of directory services, e.g. LDAP, in a simple way. The client then uses JNDI to look up the Administered object.

When the right Administered object is found the client can establish a logical connection to that object through a JMS provider.

There are two domains of messaging: **point-to-point** and **publish/subscribe**. A JMS provider must at least implement one of these messaging domains.

The **point-to-point** domain uses the concept of queues, senders, and receivers. This is illustrated in the figure below.

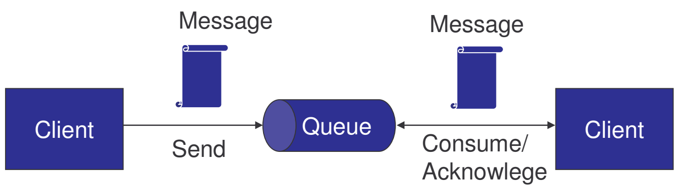


Figure 8 - How a message is sent and received in the point-to-point domain.

One client sends a message to a queue whereas the other client consumes the messages received form the queue. This means that the sender can keep sending messages even though the receiver is not fetching the messages. A given message can only be sent to one receiver. Once the receiver is online it can fetch the message from the queue in the same order as they were sent. The receiver also has to acknowledge each received message. The use of point-to-point messaging is useful when the message should be read by only one consumer.

In the **publish/subscribe** messaging domain the client sends a message addressing it to a topic. In this domain the actual sending is called publishing and to receive you have to first subscribe. The messaging in the publish/subscribe domain can be seen in the figure below.

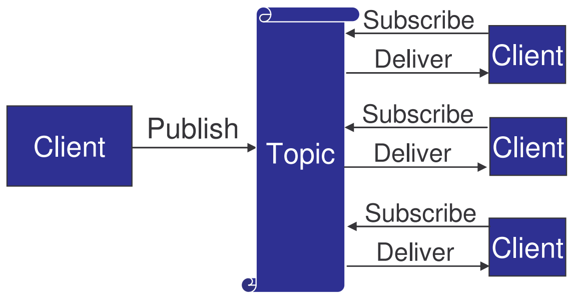


Figure 9 - The Publish/subscribe messaging domain.

The client that wants to send a message publishes it to the Topic. The Topic then delivers the message to the clients that have previously subscribed. The difference compared to the point-to-point messaging is that the message is not retained in the Topic longer than the time it takes to distribute the message to the subscribers. Thus, if a client subscribes later it can not get previously sent messages. However, in JMS there are something called durable subscribers that can receive messages even though they are offline for a while. This means that if a message should be delivered to many clients the publish/subscribe messaging should be chosen.

An application can be made up of different building blocks. In JMS these are **Administered objects** (**connection factories** and **destinations**), **Connections**, **Sessions**, **Message producers**, **Message consumers**, and **Messages**. The JMS API programming model can be seen in the figure below.

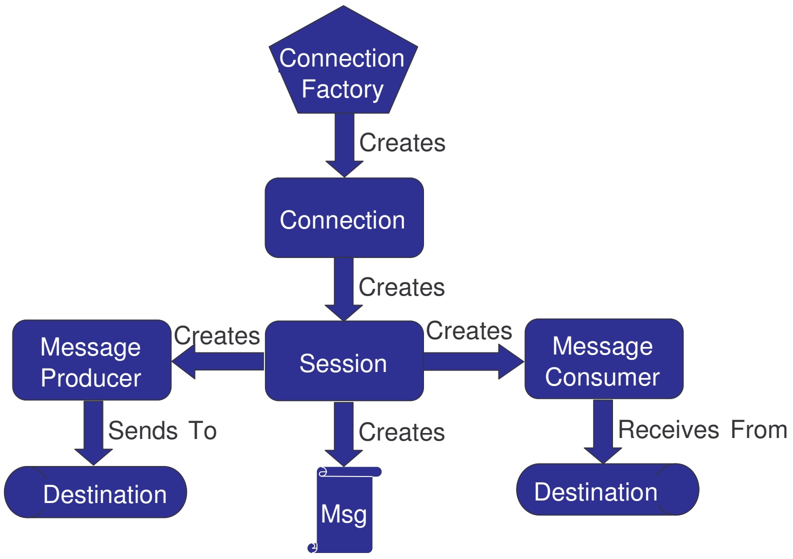


Figure 10 - The JMS API programming model

Connection Factory
The client uses the object connection factory when creating a connection with a provider. To create a new connection factory in the point-to-point domain you can use the administrative tool j2eeadmin:

```
j2eeadmin -addJmsFactory jndi_name queue
```

The connection factory is either an instance of a QueueConnectionFactory or a TopicConnectionFactory interface depending on the messaging domain. The following code creates an instance of a connection factory in the JMS client, using the point-to-point message domain:

```
QueueConnectionFactory queueConnectionFactory =
(QueueConnectionFactory) ctx.lookup("QueueConnectionFactory");
```

The line `ctx.lookup("QueueConnectionFactory")` uses JNDI to look up a QueueConnectionFactory and assign it to queueConnectionFactory.

Destination
Destinations are called queues or topics depending on messaging domain. In the destination object the client saves an address to the target when a message is produced and an address of the source when a message is received. By using the tool j2eeadmin you can, for example, create a queue:

```
j2eeadmin -addJmsDestination "MyQueue" queue
```

JMS has no naming policy, and the administrator can place an administered object anywhere in a namespace. However, JMS does not provide a JMS client to create, administer, or delete queues. This is most of the time not a problem since nearly all clients use statically defined queues. Instead, a JMS client application has to look up a queue using JNDI:

```
Queue myQueue = (Queue) ctx.lookup("MyQueue");
```

Connection
The connection object is used to create one or more session objects. You use it to encapsulate a virtual connection with a JMS provider. By using the connection factory you can now create a connection:

```
QueueConnection queueConnection =
queueConnectionFactory.createQueueConnection();
```

Session
Once the connection is made, you use a session to create messages, message producers, and message consumers:

```
QueueSession queueSession =queueConnection.createQueueSession(true, 0);
```

Message consumers and message producers
The objects *message producers* and *message consumers* are used for sending the messages to a destination. The following commands create a message producer and a message consumer and start the sending and receiving respectively. As you can see you first have to connect to be able to receive a message.

```
QueueSender queueSender = queueSession.createSender(myQueue);
QueueReceiver queueReceiver = queueSession.createReceiver(myQueue);

queueSender.send(message); //send the message

queueConnection.start(); //start the connection
Message m = queueReceiver.receive(); //receive the message
```

Listener
In JMS there is a possibility to receive messages asynchronously. This is accomplished using a message listener which is an object that uses event handling for messages. To make it work you have to register the listener with the message consumer.

```
QueueListener queueListener = new QueueListener();
queueReceiver.setMessageListener(queueListener);
```

Message
A JMS message is made up of three parts: a header, properties, and a body. The only thing that is required is the header, the other parts are optional. The header field contains, for example, a MessageID, the name of the destination queue, etc. In the properties you can set further values (i.e., those not included in the header). JMS supports a couple of

message types. For example, if the message type in the body is set to TextMessage the body contains a java.lang.String object. Below is the code used to send a TextMessage to a queue.

```
TextMessage message = queueSession.createTextMessage();
message.setText(my_string);
queueSender.send(message);
```

At the other end the message received is always a generic Message object and has to be typecasted to the right format.

```
Message m = queueReceiver.receive();
if (m instanceof TextMessage) {TextMessage message = (TextMessage) m;}
```

Implementing JMS
A powerful reason why the sample application and the parser generator should implement the JMS interface is that Orc Software has other development projects that will be using a subset of the JMS interface. Another reason is that JMS is increasing in popularity, thus the chances that customers will be familiar with this interface are higher.

The idea is not to implement the entire JMS, rather pick some useful objects and methods. The application suite should be as simple as possible for the customers to understand. Initially, it is better to implement just a few methods in order to see how the customers like the idea. If there is a need to include the whole JMS interface this can be done in the future.

2.3 JavaCC

The Java Compiler Compiler (JavaCC) and is a generator for producing both a parser and a lexical analyzer written in Java. As explained in section 2.1, a lexical analyzer takes a stream of characters and breaks into tokens. The parser, on the other hand, accepts these tokens as input and determines if it matches the grammar of the language.

Sreeni Viswanadha and Sriram Sankar created JavaCC [16] and distributed it through the company WebGain. However, since June 2003, JavaCC is open source and the source code can be downloaded free from java.net.

The input to JavaCC is a file, called a grammar file, containing both a parser and a lexical analyzer according to JavaCC specifications. The code for a simple example called Test.jj can be seen in Figure 11.

```
PARSER_BEGIN(Test)
public class Test {
  public static void main(String args[]) throws ParseException {
    Test parser = new Test(System.in);
    parser.Production1();
  }
}
PARSER_END(Test)

TOKEN:
{
      <DIGIT: [ "0"-"9" ]>
|     <LETTER: [ "A"-"Z", "a"-"z" ]>
|     <KEY: [ "0"-"9" ] | [ "A"-"Z", "a"-"z" ]>
}

void Production1() :
{}
{
  "{" Production2() "}" <EOF>
}
void Production2() :
{}
{
   <DIGIT> | <LETTER> | <KEY>
}
```

Figure 11 - A simple example of a grammar file for JavaCC.

The region between PARSER_BEGIN and PARSER_END is called the Java compilation unit. The main program creates a parser object and calls the non-terminal Production1(). TOKEN starts the section in the grammar file called the lexical specification region. This is where the lexical tokens, matched against the character input stream, are specified. The three lexical tokens are in this case DIGIT, LETTER, and KEY, and the regular expressions for them are ["0"-"9"] and ["A"-"Z", "a"-"z"]. SKIP, SPECIAL_TOKEN, and MORE are additional lexical specifications that could be used. [17]

The non-terminals Production1() and Production2() are called grammar productions or BNF productions. They indicate that the character stream must start with a left brace, followed by a digit or a letter, a right brace, and end with an end of line token.

When this grammar file is processed by *javacc* it creates three main files: Test.java, TestConstants.java, and TestTokenManager.java.

Test.java is the generated parser for this grammar file. This file is the one to run to start the parsing of a stream of character.

TestTokenManager.java is the Token Manager [18] or lexical analyzer. It tries to match the maximum number of character from the input file with the regular expressions specified in the grammar file. If there is more than one longest match, the regular expression that comes first in the lexical specification section in the grammar file is chosen. Thus, in the code above, the regular expression to the token KEY will never be matched even though it should match any letter or digit.

TestConstants.java is an interface consisting of a table of the constants that are used by the parser and the token manager.

In addition, *javacc* creates a couple of java files needed to handle exceptions and representation of tokens.

Entering the compilation command at the command prompt looks as follows:

```
D:\javacc\javacc Test.jj

Reading from file Test.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "TestCharStream.java" does not exist.  Will create one.
Parser generated successfully.
```

JavaCC is based on the LL(1) [19] algorithm which was explained in detail in section 2.1. However, JavaCC allows you to use grammars that are not LL(1) by using look-ahead. The specifications for look-ahead build into JavaCC can be used where the LL(1) rules are not satisfactory. The grammar could be ambiguous, meaning it can be matched in two ways. Often JavaCC gives you a warning message when compiling if ambiguity is likely, for example:

```
Choice conflict in (...)* construct at line 25, column 8.
Expansion nested within construct and expansion following construct
have common prefixes, one of which is: ","
Consider using a lookahead of 2 or more for nested expansion.
```

The general structure of a LOOKAHEAD specification is:

```
LOOKAHEAD( amount, expansion, { boolean_expression } )
```

amount – states the number of tokens to LOOKAHEAD
expansion – give the expansion to use to perform syntactic LOOKAHEAD
boolean_expression – is the expression to use for semantic LOOKAHEAD

When specifying LOOKAHEAD one of the three entries must at minimum be present. There are four different ways to set LOOKAHEAD: Global, local, syntactic, and semantic.

In the above example one way to solve the conflict would be to set the option LOOKAHEAD to 2. This is a global LOOKAHEAD and means the parsing algorithm in essence becomes LL(2), i.e. during parsing two tokens will be looked at before making a choice.

However, a global choice is often not the best way when it comes to performance. By converting the entire grammar to LL(2) the part of the grammar that still could work using LL(1) will not be as effective.

Instead of using global LOOKAHEAD perhaps a local LOOKAHEAD could be sufficient to solve the conflict. Then the LOOKAHEAD is set at a specific point in the grammar file. This means that most part of the grammar is still LL(1) and thus, performs better. Local LOOKAHEAD could for example look like this:

```
void PRODUCTION1() :{}
{
LOOKAHEAD(2)
      <NUM> "{" word() "}"
|
      <DIGIT>
}
```

In the above example the LOOKAHEAD of 2 only affects the first choice <NUM> "{" word() "}". The other choice is still LL(1).

A third choice to solve the conflict could be by using something called syntactic LOOKAHEAD. In this case an expansion is specified that should be evaluated. If the expansion is true, the following choice is taken.

The last alternative is using semantic LOOKAHEAD. It works by specifying a Boolean expression whose evaluation determines what actions to take.

The example below shows both syntactic- and semantic LOOKAHED:

```
void Production1() :{}
{
 LOOKAHEAD( Production2() ) Production2()
|
 LOOKAHEAD( getToken(1).kind == "Product3" ) Production3()
|
 Production4()
}
```

In this example the first LOOKAHEAD uses syntactic LOOKAHEAD and states that if the input of tokens matches Production2, go to Production2(). The second LOOKAHEAD uses semantic LOOKAHEAD and tells us that if the next token is Product3 go to Production3().

Running the parser now will only check if the stream of character matches the grammar specified. Usually, there is a need to do more than that. JavaCC has a tool, JJTree, which can expand the use of the grammar.

JJTree [20] creates the tree structure consisting of nodes of all non-terminals in the grammar file. This can be edited manually, e.g. skipping a node of a non-terminal or creating one for a terminal. The nodes have to implement a Java interface to enable setting the parent node and adding and retrieving children.

It is possible to run JJTree in two different modes, simple or multi, depending on the complexity needed for the Node object. To construct the bottom-up tree it uses a stack that is possible to manage from inside the grammar file.

JJTree constructs two different nodes: a definite node or a conditional node. A definite node has a specific number of children whereas a conditional node can have all the nodes within its node scope as children depending on whether or not the condition set evaluates to true. One way to describe a conditional node is using the shorthand **indefinitenode** which means any node. This node is the default way JJTree creates nodes of each nonterminal if nothing else is specified. For example, to avoid creating nodes write a #void after the production name or set the NODE_DEFAULT_VOID option to true.

When the JJTree file is configured to save the necessary nodes in a tree, the command rootnode.dump() prints the whole tree. However, by default there is no information stored in the nodes without configuring the SimpleNode file (if we run jjTree in simple mode) with two added methods called, for example, setText() and getText(). This will enable you to get and set the information you want in the nodes. [21]

2.4 Building Parsers

Instead of using JavaCC, as explained in the previous section, you can write the Java code directly from the BNF (Backus Naur Form) grammar. Thus, you do not have to work with two languages and can start writing Java directly. Sequences, alternations, and repetitions in the grammar are in this case Sequence, Alternation, and Repetition objects.

2.4.1 Building blocks

The parser can be seen as an object that recognizes the elements of a language and then translates it to the correct format. All languages have a certain pattern. To check if a particular stream of characters follows that pattern is one of the basic responsibilities for a parser.

Steven John Metsker [5] suggests that for a functional parser, three classes are needed: **Assembly**-, **Assembler**-, and a **Parser** class. As stated before, the parser is an object that recognizes if a certain stream of character follows the grammar of a language. In this setup, the Assembler helps the parser to build a result, and the Assembly offers the parser a place to work.

The Assembly Class
The Assembly assigns an index to the string it reads. It also gives the parser a stack and a target object to work on. The Assembly class must implement the interfaces PubliclyCloneable and Enumeration. This is needed because the Assembly often has to record the progress of the parser and clone itself when there are multiple ways to progress (see Table 3).

Table 3 - Shows the progress of the Assembly from an input string.

String	" Put a Car in the garage"
index	3
Stack	Car
Target	a PutCommand object

In Table 3, the input string can be seen in the first row. The index indicates the parser has identified three words so far. The third word, Car, is put onto the Stack and the target object now is PutCommand.

When the Enumeration interface is implemented, the Assembly must have the two methods hasMoreElements() and nextElement(). This is done in two subclasses; TokenAssembly and CharacterAssembly. These two subclasses are needed if the parser should be able to parse a text as a string of tokens.

Tokenizing a text is, as described in chapter 4, the way of taking apart a text into logical pieces, such as numbers, words, and punctuations. This is called the lexical analysis step in the progress of reacting to the input.

Parser
A parser is an object that recognizes a stream of characters. In this example, there are two kinds of parsers, either a Terminal or a Composite. A terminal parser uses no other parsers when recognizing a string. The composite parser, on the other hand, uses other parsers to carry out the same task.

Assembler
The Assembler assigns meaning or semantics to the parser. Up to this point, we have been able to recognize if a string matches a particular language pattern. Now, we want to be able to react to the input, and this is where the Assembler comes in.

This is how it works: When a parser has a match against an assembly, the method workOn() in the assembler is called. The assembler now knows the parser has a match and could start to work on the assembly. First, it could pop the token from the stack of the assembly and then get the target object and, using a suitable method, set some attributes. Figure 12 shows an example of an assembler taken from the book *Building Parsers with Java* [5].

```
/**
* This assembler pops a string and sets the target
* coffee's country to this string.
*/
public class CountryAssembler extends Assembler {

  public void workOn(Assembly a) {
  Token t = (Token) a.pop();
  Coffee c = (Coffee) a.getTarget();
  c.setCountry(t.sval().trim());
  }
}
```

Figure 12 – Code example of an Assembler

2.4.2 Building a Parser

First there is a need for a grammar. The grammar is the rules of the language the string has to follow to be accepted by the parser. After the grammar is written the next step is to write the Java code that acts upon the pattern of the grammar.

Metsker suggests starting by writing some sample sentences to test against a simple grammar consisting of only a few rules. Next, you can expand the grammar to fit all the rules in the language. This means using an iterative approach for coding and testing.

2.5 Orc Protocol

As explained in the introduction, the Orc Server System has an interface, the Orc Protocol (OP). This enables other applications, for example, to access the CDS database in the server. This is illustrated in the figure below.

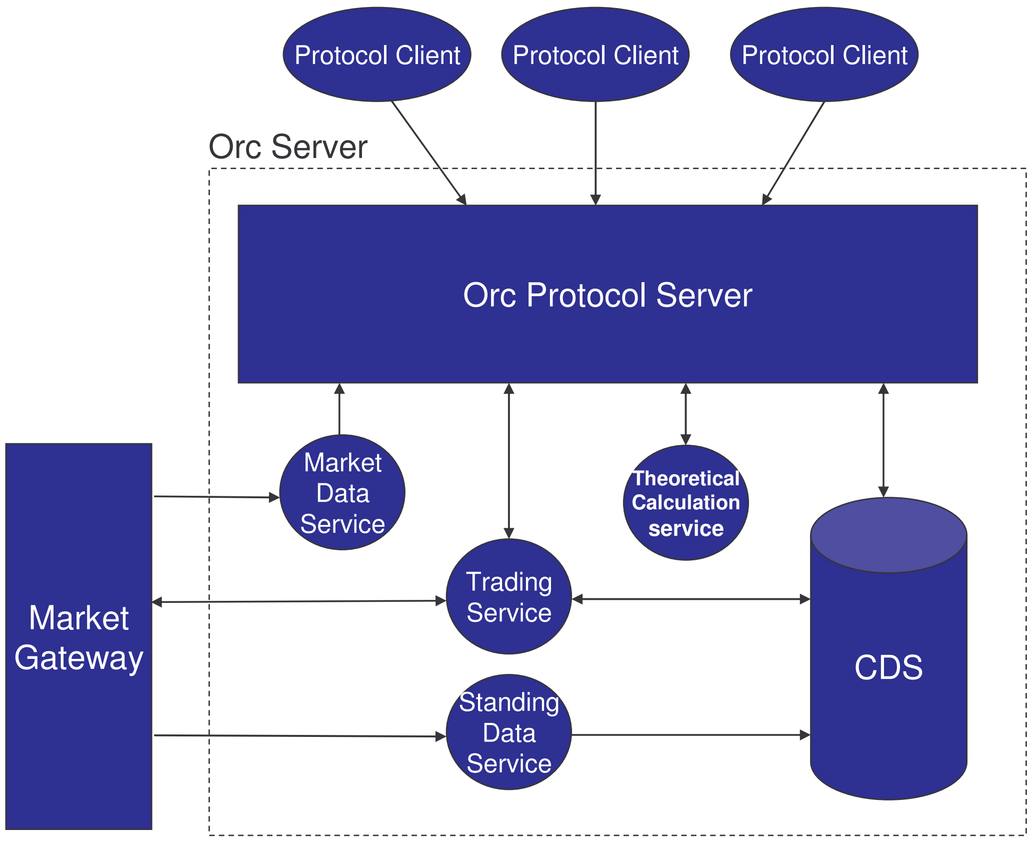


Figure 13 – An overview of the Orc Protocol Interface. [3]

As can be seen in the figure above there are four processes running on the Orc server: Trading Service, Market Data Service, Standing Data Service, and Theoretical Calculation Service. These services are called the Orc Core Services and provide an alternative to existing services that are accessible using an Orc client such as Orc Trader and Orc Broker.

The Trading Service can handle orders, trades, and quotes for connected Orc Protocol clients. The Market Data Service provides market data and, for example, news messages. The Standing Data Service handles automation of standing data such as: automatic download, or deletion/creation of dynamic combinations. The Theoretical Calculation Service takes care of theoretical calculations and theoretical price feeds.

OP provides many messages to communicate with the OP process within the Orc Server. This communication is TCP/IP based and the messages are sent in ASCII format. OP is designed to be independent of both the underlying programming language and platform.

OP consists of four parts: the Orc Protocol server, a communication model, a message format, and a set of messages. [4]

2.5.1 Orc Protocol server

The OP server is included in the standard Orc Software distribution. As explained earlier, the OP server is always waiting for new clients to connect; it can then communicate with other Orc server components as needed. Each client executes in its own thread.

2.5.2 Communication model

As mentioned, when a client connects to the OP server it uses a standard TCP/IP connection. OP does not have an IANA [22] assigned port. Both the hostname and port number for the OP server are configured by the customer. The OP server requires username/password authentication when clients log in to the server. When the client is authorized, it can start exchanging data with the OP.

The OP server communicates in two ways with the client, either synchronous or asynchronous. Normally the communication is synchronous. However, if there is a need for a real time trade feed, such as price, order, or news feed, an order_feed_toggle message has to be sent to the OP server. In this way the client will receive asynchronous messages. Trade feed messages from the OP are now mixed in between the other, normal, replies. The message from the client can be marked with a unique identifier which the server uses in the reply message.

2.5.3 Message format

The message format for OP is specified using BNF notation which contains a set of meta-symbols. These meta-symbols are displayed in the table below and are also matched against the symbols used in JavaCC. This compare is done to make it easier making a grammar file in JavaCC.

Table 4 - Meta Symbols and their corresponding description in OP and JavaCC

Meta Symbols OP	Descriptions	Meta Symbols JavaCC
::=	is defined as	:
|	or	|
()	group items together	()
[]	enclose an optional item	[] or ()?
{}	enclose a repetitive item	()* or ()+
""	enclose a terminal item	""

An OP message format is described in BNF:

```
message ::= message_length dictionary [whitespaces]
message_length ::= 10*digit
dictionary ::= "{" key_value_combinations "}"
digit ::= "0" | "1" | ... | "9"
key_value_combinations ::=
key_value_combination { "|" key_value_combination }
key_value_combination ::=
[whitespaces] key [whitespaces] "=" [whitespaces] value
[whitespaces]
whitespaces ::= whitespace { whitespace }
whitespace ::= tab | linefeed | carriage_return | space
key ::= letter {letter | digit | underscore}
value ::= string | integer | float | boolean | date | time | dictionary
| enum | empty_value
empty_value ::= "[None]"
letter ::= uppercase_letter | lowercase_letter
uppercase_letter ::= "A" | "B" | ... | "Z"
lowercase_letter ::= "a" | "b" | ... | "z"
underscore ::= "_"
string ::=legal_character{legal_character}|empty_string
empty_string ::= citation citation
citation ::= """
integer ::= [sign] digit
[digit[digit[digit[digit[digit[digit[digit[digit[digit]]]]]]]]]
float ::= [sign] digit "." 12*digit ("E" | "e") [sign] 2*digit [digit]
sign ::= "+" | "-"
boolean ::= "FALSE"|"TRUE"
date ::= 4*digit "-" 2*digit "-" 2*digit /* yyyy-mm-dd */
time ::= 2*digit ":" 2*digit ":" 2*digit /*hh:mm:ss */
enum ::= A list of values of same type (most likely strings)
```

Figure 14 - The BNF description of the OP

Looking in the table there are some explanations in order. First, a legal character in OP is any characters except "{", "|" and "}". If there is a need to send an empty string to the OP it has to be specified using two consecutive citation marks (""). An integer can be up to 11 characters long including the sign and has a max/min value of +/- 2,147,483,648. While OP does not support the Long format, an Integer longer than 11 digits will be interpreted as a String. Last, the message length in the OP refers to the message length in bytes and not symbol length. All messages sent to and from OP should be UTF8 encoded. The problem is that sometimes UTF8 encoded strings have a symbol length that is not equal to byte length, thus it is important to ensure the byte length is used.

It is also important to know that parsing on key positions in messages is discouraged as Orc Software could change these positions without notice. Most probably the best way to parse is parsing the characters "{", "}", and "|". However, if a value has to contain the parsing characters there is a way in the OP to represent them without risking a parseexception. The characters "{", "}", "\" and "|" should be represented as "\[", "\]", "\\" and "\/" respectively.

Another issue to take into account is that OP version 5.1 and later has no spaces in the messages, that is the message structure is {key1=value1|key2=value2} and not { key1 =

value1 | key2 = value2 } as for previous versions. However, if there is a need to use the older format one can always set the configuration setting old_style_output_spacing to true.

2.5.4 OP messages

As can be seen in the table above, a message in the Orc Protocol has two parts; a message length and a dictionary. First in the message comes the message length, which is a ten character long string of digits which indicates the byte length of the message excluding the message length field itself. The rest of the message consists of a dictionary that can have one or more key/value combinations. All messages from the client have different key/value combinations depending on message type, but they all have to include a message_info key/value combination. The message_info key/value combination in a reply from the server is replaced with a reply_to key/value combination. As explained above, the order of appearance of key/value combinations is not specified, meaning that key/value pairs can occur in any order.

An example of an OP message is:

```
1234567890{my_string=astring|some_int=123|a_float=-1.2345678901234567E-
123}
```

The message length is followed by the message (always a dictionary). The first key/value combination is a key called "my_string" with the value "a string". A key references some logical data and implicitly gives the type of the value, in this case a String. Next, the Meta symbol "|" indicates there are additional key value combinations. The second key is "some_int" and the value is an integer. As illustrated in the OP BNF table above, a value can be an integer, a float, a string, a date, a time or a dictionary depending upon which key it is.

If we instead look at a real OP message it could look like this:

```
0000000130{message_info={message_type=order_insert}|order={buy_or_sell=
buy|instrument_id={market=Saxess|feedcode=101}|price=29|volume=2000}}
```

The message length in this example has the same symbol length as byte length, 130 characters. As described above, a message must include a message_info key. The message_info key has a dictionary as value which consists of the key message_type. This message is an order_insert type to buy 2000 Ericsson B (feedcode = 101) stocks in the Saxess market at the price of 29 SEK each.

The third example is a combination of a message sent to the OP Server and a reply message:

```
0000000123{ message_info = { message_type = login } | login_id =
tobias| allow_ping = True | ping_timeout = 120 | ping_interval = 30 }

0000000100 { reply_to={ message_type=login } | login_id=tobias |
version=5.2.11b | utc_offset=3600 | error=0 }
```

This is a login to the OP server for the user tobias, and then a reply message with error=0 indicating that the login was successful. The difference between the client message and the server message is the message_info key is changed to a reply_to key.

3 Results

This section offers a proposed solution for the problem. It describes the design and the implementation of an OP client. Following this an evaluation on this OP client is presented.

3.1 Analysis and design

We begin by implementing a solution using the underlying idea behind DOM, thus it stores all objects in a tree after the parsing is done. This decision was made because there was no efficient way to be able to foresee what message was sent without parsing the entire message. As stated in section 2.5, the Orc Protocol uses key/value combinations whose order can be changed by Orc Software without notice. Thus, while the message type today is the first dictionary, this need not be the case in the future.

A subset of the JMS interface has also been implemented. As a start, it will use the message format MapMessage, to send and receive messages. As mentioned in section 2.2.3, the main reason why the OP client implements the JMS interface is that Orc Software has other development projects that will be using the JMS interface. Of course, the fact that JMS is increasing in popularity has also been a factor in choosing to use JMS.

An overview of the proposed solution can be seen in the figure below.

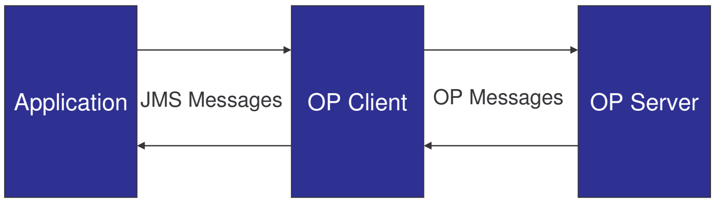


Figure 15 - An overview over the proposed solution.

To send a message to the OP Server the customer only has to create a MapMessage that will result in a specific OP message. An example of a code creating an order insert message can be seen below.

```
/*
 * Sends a order_insert message to the OP server
 * Checks the reply.
 */
 public void order_insert(){
  /*
   * Create producer, consumer, and a MapMessage.
   */

   producer = session.createProducer();
   consumer=session.createConsumer();

   //Create a new MapMessage and set the Request number.
   order_insert = session.createMapMessage();
   aRequestNr = aRequestNr+1;

  /*
   * The message that will be sent is:
   * message_info = { message_type = order_insert | private =
   * aRequestNr} | activate = exchange | order = { buy_or_sell = buy |
   * instrument_id = { market = *Saxess | feedcode = 101 } | price =
   * 22 | volume = 1000 } }
   */

  /*
   * SetMessageType() loads an order_insert message with parameters:
   * message_info = { message_type = order_insert | private=aRequestNr}
   *
   * If you want to be able to retrieve the reply message using a
   * consumer, you will have to provide a requestNr as second input
   * to the function setMessageType().
   */

  order_insert.setMessageType("order_insert", aRequestNr);

  // | activate = exchange
  order_insert.setString("activate", "exchange");

  String feedCode = Settings.getSettingsResource(Settings.FEEDCODE);
  String market = Settings.getSettingsResource(Settings.MARKET);
  String volume = Settings.getSettingsResource(Settings.VOLUME);
  String price = Settings.getSettingsResource(Settings.PRICE);
  String buyOrSell =Settings.getSettingsResource(Settings.BUY_OR_SELL);

  //| order = {buy_or_sell = buy | instrument_id ={market = Saxess |
  //  feedcode = 101}

  Dictionary order=new Dictionary();
  order.setString("buy_or_sell",buyOrSell);

  Dictionary instrument_id=new Dictionary();
  instrument_id.setString("market", market);
  instrument_id.setString("feedcode", feedCode);
  order.setDictionary("instrument_id", instrument_id);

  //| price = 1 | volume = 1}}
  order.setString("price", price);
  order.setString("volume", volume);
  order_insert.setDictionary("order",order);

  // Send the message to Orc Server
  producer.send(order_insert);

  /*
   * Receive a message reply
   * Get some information from the message. Print it.
   */
   aReply = consumer.receive(aRequestNr);

   if ((aReply.getString("error").equalsIgnoreCase("0"))){
        String order_tag=aReply.getString("order_tag");
        String activate=aReply.getString("activate");
        System.out.println("The order tag is = "+order_tag);
        System.out.println("Activate = "+activate);
    }
    else{
    System.out.println("ERROR:"+aReply.getString("error_description");
    }
}
```

Figure 16 - The code to create a MapMessage for an order insert OP message.

This message is then transformed into an OP message and delivered to the OP Server. The reply from the OP Server is, in its turn, transformed into a MapMessage and sent back to the client. The client can then retrieve the information needed in the same way as he/she constructed the MapMessage. The components of the OP client are displayed in the figure below.

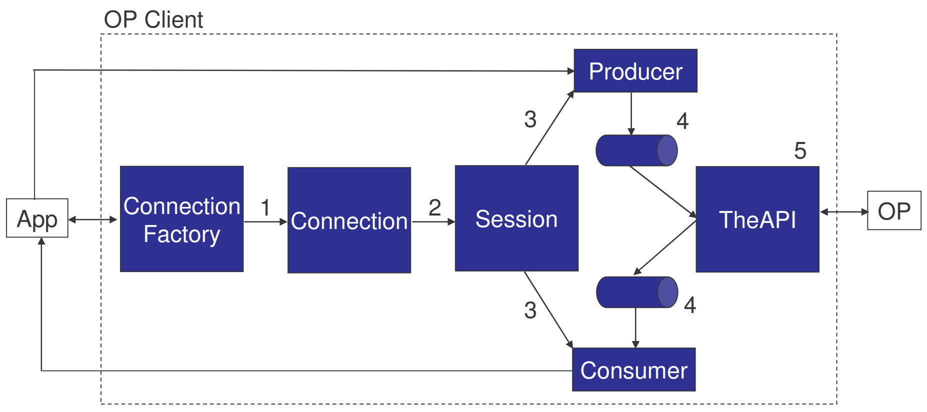


Figure 17 - The components in the OP Client

First of all the client program (App in the figure above) has to create a Connection Factory object. By using the Connection Factory object, a Connection object can be created (1). The Connection creates a Session object (2) and a connection with the OP Server using TheAPI object. The Session, on the other hand, creates both a Producer and a Consumer (3). Both the Consumer and the Producer have a queue object to store messages (4). TheAPI controls the communication with the OP server and puts and polls messages from the queues (5).

The communication between a client application and the OP Server can be illustrated more in detail using the figure below.

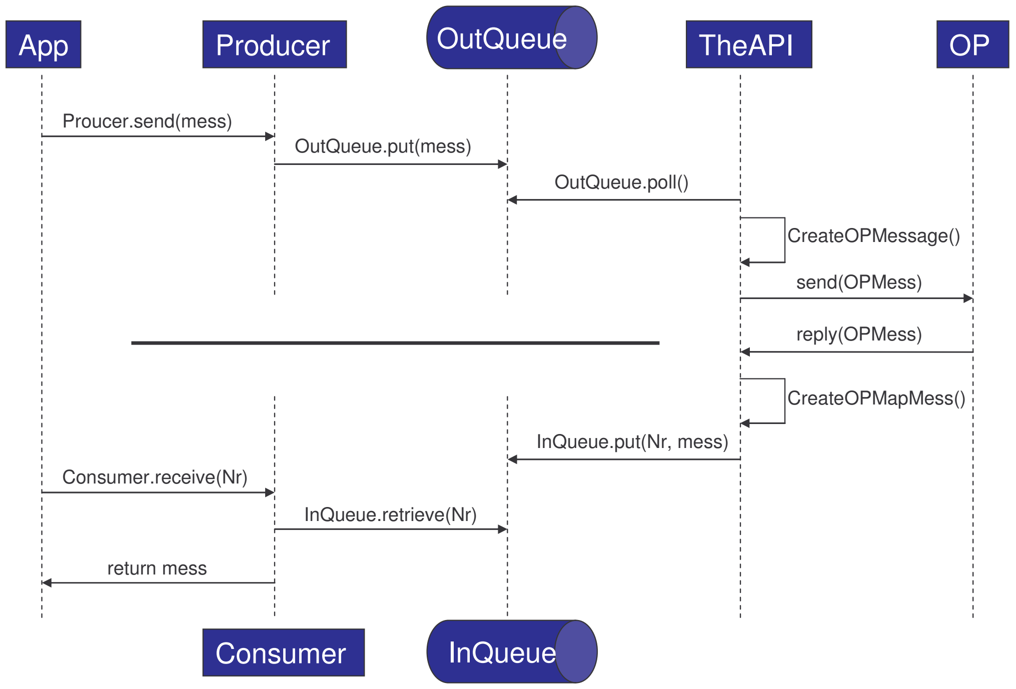


Figure 18 - The communication between an application and the OP Server.

The client sends messages using the Producer which then puts the messages in a OutQueue. TheAPI thread checks the OutQueue for new messages, and then pulls them out of the OutQueue. TheAPI also translates the MapMessage to an OP message before sending to the OP Server. When there is a reply from the OP Server, TheAPI translates the messages into MapMessages and puts them in an InQueue. The client then uses the Consumer object to retrieve the MapMessage. The Consumer checks the InQueue and returns the message to the client.

The second example is when a client wants to receive asynchronous messages, so called feed toggle messages, from the OP Server. The clients have to connect to a Topic using Subscribers which is illustrated in the figure below.

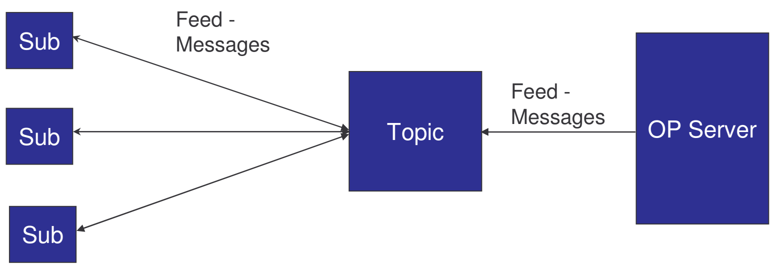


Figure 19 - An overview of the solution with asynchronous messages.

By using a Topic, a message is only sent to that Topic which then distributes the message to its subscribers. The OP Server does not need to know which clients should receive the feed message. One example of Feed Toggle Messages is PriceFeed_Toogle where a client receives price updates on a financial instrument continuously. Using the same illustration as before, the components in the OP Client when subscribing using Topics are shown in Figure 20.

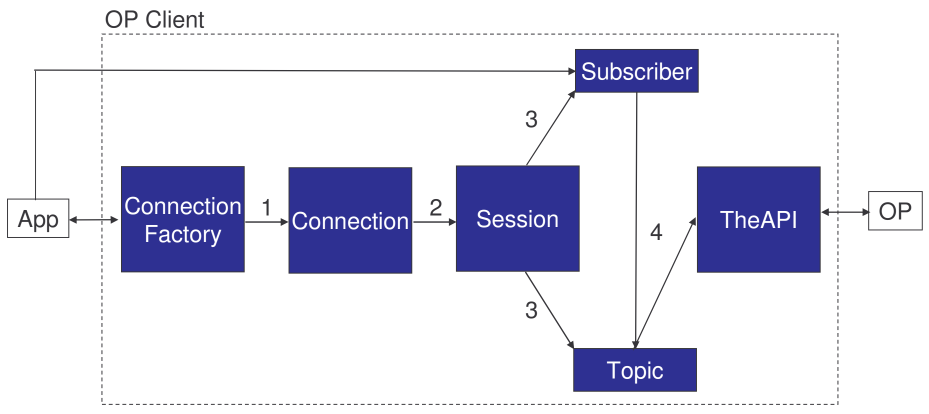


Figure 20 - The components in the OP client when using Topics.

The steps 1 and 2 in the Figure 17 are the same as when sending synchronous messages. In this case the Session creates both a Subscriber and a Topic (3). The Subscriber handles the connection to the client application and the Topic sends subscriptions to the TheAPI that, in turn, sends the message to the OP Server (4).

Figure 21 illustrates more in detail the communication between the client and the OP Server when the client subscribes to a feed message.

The client sends a Listener to the Subscriber, which will listen to a certain feed message. When the Subscriber was created using the Session, as explained above, the message was stored at the Subscriber. This message and the Listener are sent to the Topic. The Topic sends the message to TheAPI and waits for a hash code to identify a feed message later. This hash code is created by TheAPI. TheAPI also creates an OP message and sends it to the OP server. When feed messages arrive to theAPI the hash code is calculated and set as a parameter in the MapMessage that is also created. The message is then sent to the Topic. The Topic searches for the Listeners that should receive the message using the hash code. Last, the OnMessage() function in the Listener can retrieve what is needed from the message.

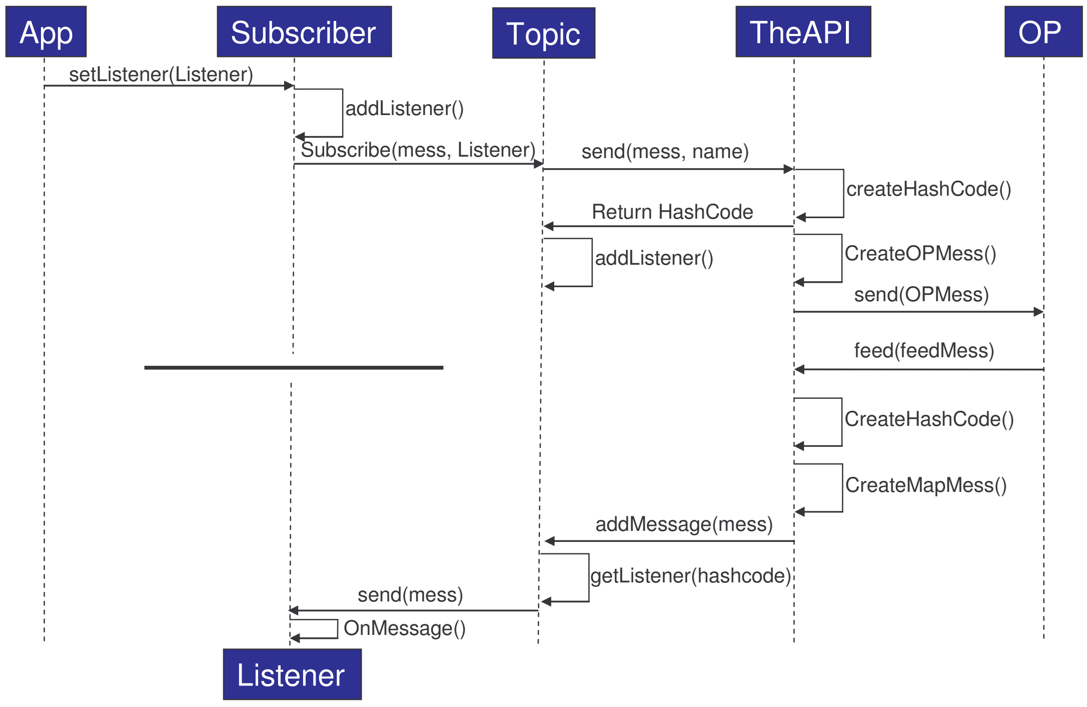


Figure 21 - The communication between the client and the OP server when the client subscribes to a feed message.

3.1.1 Scalability

When multiple clients run on the same machine, the customers can choose between two different scenarios using the OP client. First, they can create a new object for each client wanting access to the OP server, or second, they can share the OP client between each other. The scalability with respect to multiple clients running on the same machine depends on which scenario used.

If the customer uses the first scenario there will be many threads running on the same machine. The OP client itself uses three threads to be able to handle requests efficiently. The first scenario, exemplified using three clients, means that 12 threads are used. This is illustrated in the figure below.

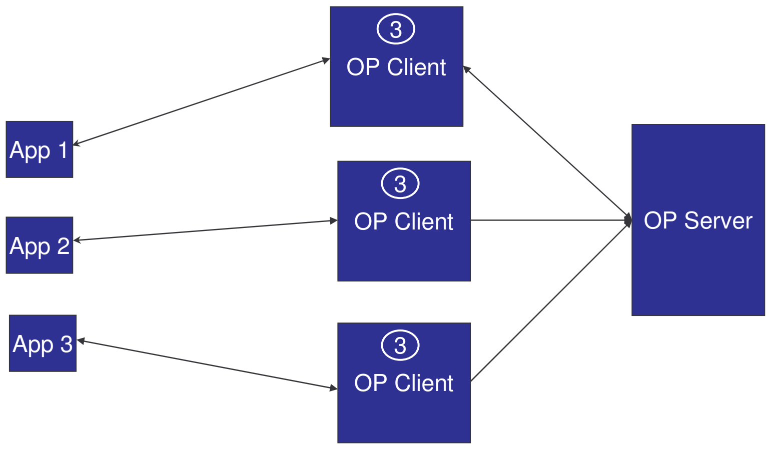


Figure 22 - Multiple clients run on the same machine and each uses an own OP client.

This scenario is best to use when the traffic load is high because OP will serve the client using three processes. It will also scale better on high end multiple CPU machine. However, because it uses many threads it will not scale so well when the number of client increases.

The second scenario is illustrated in the figure below, showing three clients connecting to one OP client.

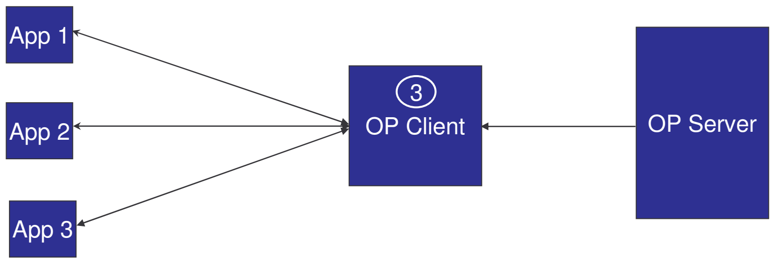


Figure 23 - Multiple clients run on the same machine and each uses the same OP client.

This setup will scale better when there are multiple clients connected. The problem is that when the load is high it will be slower because OP will serve the single client using only one process.

3.2 Parser construction

The Java class TheAPI uses an OP parser/generator to fetch an OP message and to create a MapMessage. When a string is received from OP it is passed as input to the Parser from TheAPI. This string is then parsed into different objects. These objects are then used to build a MapMessage. This can be illustrated in the following figure.

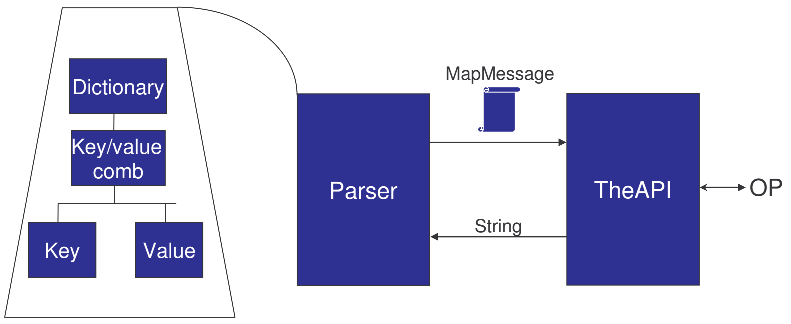


Figure 24 - The setup between the Parser and TheAPI.

The OP parser/generator has been done using JavaCC. The tool JJTree is used to build a tree of objects after parsing a character stream. The JavaCC code for the parser/generator is illustrated in the figure below.

```
PARSER_BEGIN(Parser1)
public class Parser1 {
}
PARSER_END(Parser1)
<*>
SKIP :{
  " "
| "\t"
| "\n"
| "\r"
}
<*>
TOKEN :{
      <#DIGIT: [ "0"-"9" ]>
|     <#LETTER: [ "A"-"Z", "a"-"z" ]>
|     <START_DICT: ["{"]>:DEFAULT
|     <CLOSE_DICT: ["}"]>:DEFAULT
|     <NEW_COMB: ["|"]>:DEFAULT
}
<DEFAULT>
TOKEN :{
      <MESSAGE_LENGTH: (<DIGIT>){10}>
|     <#UNDERSCORE: "_">
|     <KEY: <LETTER>| ( <LETTER> | <DIGIT> | <UNDERSCORE> )*>
|     <GO_TO_VALUE_STATE: "=">:VALUE_STATE
}
<VALUE_STATE>
TOKEN :{
      <VALUE: ( <STRING> | <CHARACTER> | <INTEGER> | <FLOAT> |
<BOOLEAN> | <DATE> | <TIME> | <EMPTY_VALUE> )>
|     <#INTEGER: (<SIGN>)? (<DIGIT>){1,10}>
|     <#FLOAT: (<SIGN>)? <DIGIT> "." (<DIGIT>){12} ("E" | "e")
(<SIGN>)? (<DIGIT>){2}>
|     <#SIGN: "+" | "-">
|     <#BOOLEAN: "FALSE"|"TRUE">
|     <#DATE: (<DIGIT>){4} "-" (<DIGIT>){2} "-" (<DIGIT>){2}>
|     <#TIME: (<DIGIT>){2} ":" (<DIGIT>){2} ":" (<DIGIT>){2}>
|     <#STRING: (<LEGAL_CHAR> ( <LEGAL_CHAR2> )* <LEGAL_CHAR>) |
<EMPTY_STRING>>
|     <#CHARACTER: (<LETTER>)*>
|     <#EMPTY_STRING: <CITATION> <CITATION>>
|     <#CITATION: "\"">
|     <#EMPTY_VALUE: "[NONE]">
|     <#LEGAL_CHAR: ~["{","}","|"," "]>
|     <#LEGAL_CHAR2: ~["{","}","|"]>
}

SimpleNode Message() :{}{
  <MESSAGE_LENGTH>
  { jjtThis.setKey( "Message" );jjtThis.setValue( "Dictionary" );}
  Dictionary()
  {return jjtThis;}
}

void Dictionary() :{}{
  <START_DICT> Key_value_combinations() <CLOSE_DICT>
  { jjtThis.setKey( "EMPTY" );jjtThis.setValue( "key/value comb" );}
}

void Key_value_combinations() #void:{}{
  Key_value_combination() (<NEW_COMB> Key_value_combination())*
}

void Key_value_combination() #Name :{Token key, value;}{
  key=<KEY>{jjtThis.setKey(key.image);} "=" ( value = <VALUE>
{jjtThis.setValue (value.image);} | Dictionary()
{jjtThis.setValue("Dictionary");})
}
```

Figure 25 - The JavaCC code for the OP Parser/Generator.

As can be seen in the code, states are used to enable the parsing to process these messages. The parse remains in the DEFAULT state until a "=" character is read, then, the parser switches to the VALUE state. The parser switches back to DEFAULT state whenever a "{", "}", or "|" character appear in the input stream. This is done because a VALUE token could be misinterpreted as a KEY token, or the other way around depending on the order, if both tokens were to exist in the same state.

The parser creates objects of Message, Dictionary, and Key_value_combination. In this way the information stored in the tree can be used to create MapMessages according to the JMS specification (see section 2.2.3). For example, a tree for the below OP message is shown in Figure 26.

```
{ message_info = { message_type = order_insert } | activate = exchange
| order = { buy_or_sell = buy | instrument_id = { market=Saxess |
FeedCode=101} | Price = 10 | Volume = 1000 }}
```

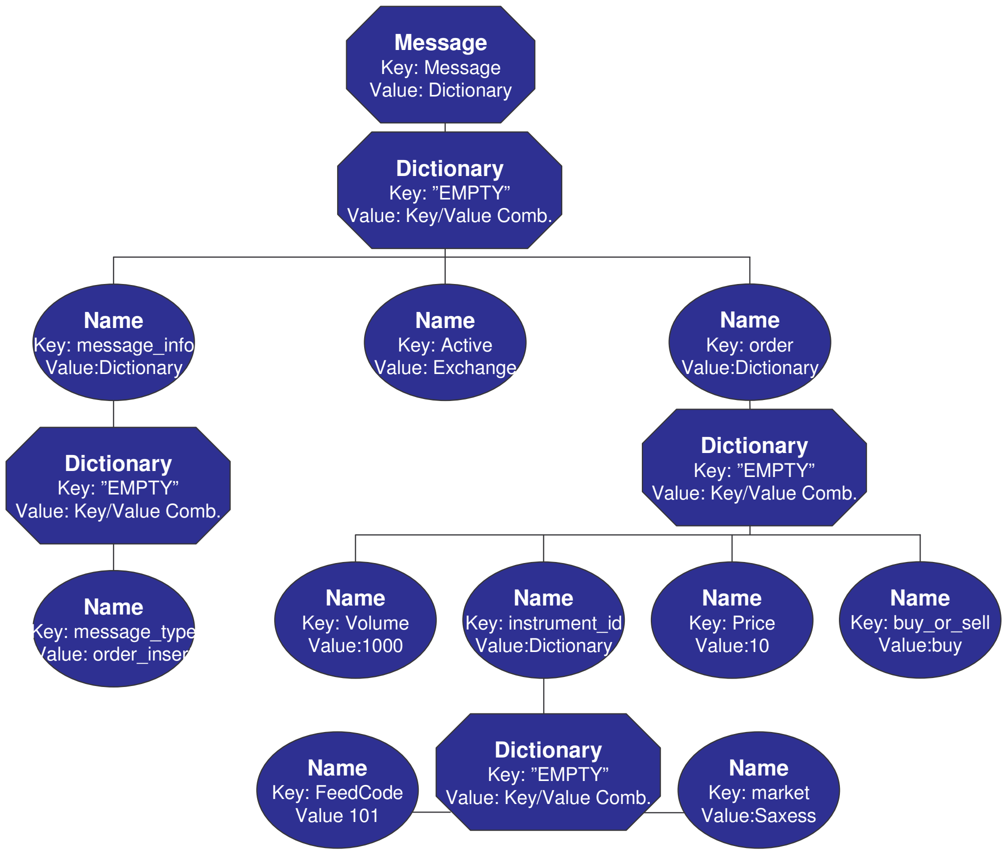


Figure 26 - The object tree created by the parser.

As stated, the relevant parts of the tree in the figure above are turned into a MapMessage. This MapMessage is illustrated in the table below.

Table 5 - An example of a MapMessage for an order_insert OP message.

MapMessage	
Message_info	**Dictionary**
Active	"Exchange"
Order	**Dictionary**
Message_info	
Message_type	"order_insert"
Order	
Volume	"1000"
Price	"10"
Buy_or_sell	"buy"
Instrument_id	**Dictionary**
Instrument_id	
Market	"Saxess"
Feedcode	"101"

3.3 Evaluation

There are two major metrics to be evaluated for an OP client. First, its speed and second, the space required. By speed means the time it takes for the program to complete a request, and by space, the memory used up by the program.

The expected results are that the OP client will be faster and less memory intensive than the FIX client. This is because the FIX client stores more information locally than the OP client.

3.3.1 Generic setup

The OP client will be evaluated against a FIX client using a FIX implementation already sold by Orc Software. The test will be to insert 1000 orders to buy a common stock at market price. The test starts when the first order is inserted and is completed when the last order is active on the market. This will be repeated until the observed standard deviation of the results is less than 5 percent, or a minimum of 10 times. The reason why we want to test a number of times is to minimize the disturbance from, for example, temporary network delays. The test will not be conducted on an external test market such as Saxess-Test market. Instead, it will be carried out on an internal market which means that there will be no VPN connections to the market or other tests that can interfere.

3.3.2 OP client setup

First, an order feed toggle message will be sent to the OP server to monitor orders. Second, 1000 order insert MapMessages will be sent to the OP client. The time starts when the first order is sent, and stops when the last order feed is received from OP stating that the order is in the market.

3.3.3 FIX client setup

In this setup a so called single order FIX message will be sent to a FIX Gateway. The FIX message sent to the FIX Gateway can be seen below.

```
[8=FIX.4.2 9=116 35=D 49=bowmore 56=fixgwHead 34=2 52=20070228-09:52:18
22=8 48=11160_oim 100=RMP 38=1 44=1 54=1 40=2 11=-9030888640□10=000]
```

This FIX message is explained more in detail in Table 6.

Table 6 - Fix Tag Number and the corresponding Field Name and description. [23]

FIX Tag Num	FIX Field Name	Description
8	BeginString	Identifies the beginning of new message and protocol version.
9	BodyLength	Message length in bytes.
35	MsgType	Defines the message type. D stands for a single order.
49	SenderCompID	Identifies the firm sending the message.
56	TargetCompID	Identifies the receiving firm.
34	MsgSeqNum	The sequence number
52	SendingTime	The time the message was transmitted.
22	SecurityIDSource	Identifies the source of the SecurityID tag. 8 stands for Exchange Symbol.
48	SecurityID	Identifies the contract, in this case, using the feedcode 11160_oim
100	ExDestination	The destination for execution. The market is RMP.
38	OrderQty	The number of shares for the equity.
44	Price	The price per unit of quantity.
54	Side	Side of the order. For example, 1 for buy and 2 for sell.
40	OrdType	OrderType. 2 stands for Limit
11	ClOrdID	Client order ID. Unique identifier assigned by the buy-side.
10	CheckSum	The checksum for the message.

3.3.4 Speed Test Results

The speed test results from inserting 1000 orders using the OP client and the FIX client are shown in Table 7.

Table 7 – Test results when inserting 1000 orders.

FIX Client			OP Client
Date	2007-03-16		2007-03-16
Server	Birka		Birka
Instrument	11160_oim		11160_oim
Market	RMP		RMP
	[ms]		[ms]
1	27 906		21 250
2	26 437		20 078
3	27 047		19 937
4	27 390		20 907
5	27 797		20 250
6	28 016		19 875
7	27 094		20 062
8	27 219		20 235
9	26 828		20 359
10	26 938		20 187
Average	27 267		20 314
Variance	259 538		190 032
Std deviation	509		436
Std in percent	1,87%		2,15%

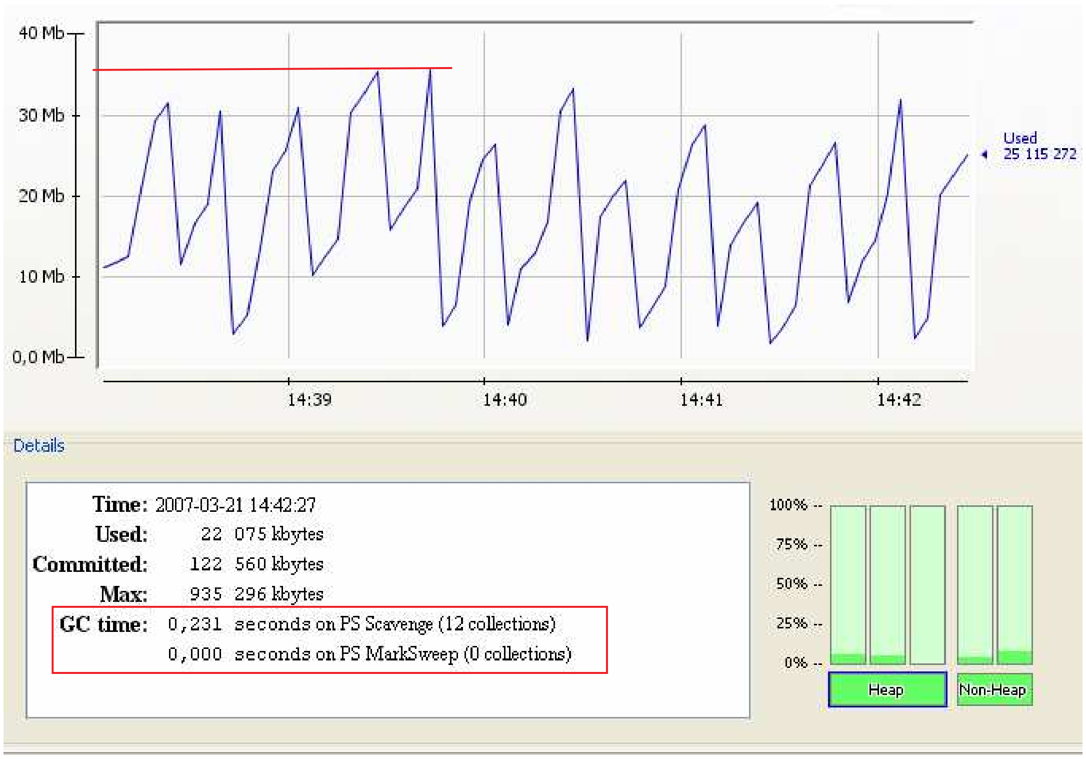
When looking in the table above, we can see that the OP client is faster than FIX client. On average, it inserts 1000 orders approximately 26 percent faster and the variance is smaller. There is no reason not to assume that this is true when it comes to other FIX and OP messages.

3.3.5 Memory consumption test results

The memory consumption test uses the Java SE Monitoring and Management service [24]. This is as service included in the Java SE 5.0 platform that, for example, provides tools to monitor current, peak, and threshold memory usage.

Figure 27 below shows the Heap Memory Usage for the OP client. We can see that the maximum memory usage was around 35 Mb. The cumulative time spent on Garbage Collection (GC) and the total number of garbage collection invocations can be seen in the "Details" area.

GC is the process of releasing objects that are no longer referenced. GC often divides memory into different segments and assigns a memory pool to each. When a segment has allocated all memory a partial GC, called minor collection, is performed. There are three segments all holding objects with different ages: **Young**, **Tenured**, and **Permanent** [25]. In the segment holding the younger objects the infant mortality rate is high, and thus, most object die there. The periodic behavior in Figure 27 is not caused by a timer; instead it is the **Young** and **Tenured** segments that are full. GC uses different algorithms to release memory in an efficient way. In this case, the PS Scavenge is used 12 times consuming 231 milliseconds.



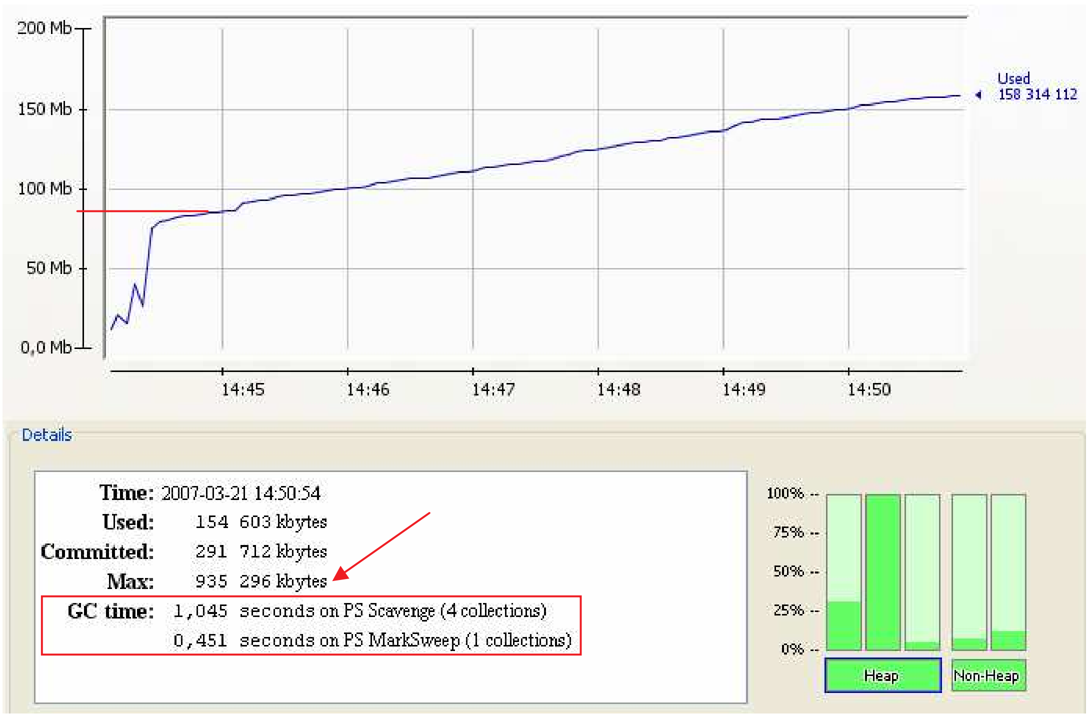
```
     Time: 2007-03-21 14:42:27
     Used:      22 075 kbytes
Committed:     122 560 kbytes
      Max:     935 296 kbytes
  GC time:   0,231 seconds on PS Scavenge (12 collections)
             0,000 seconds on PS MarkSweep (0 collections)
```

Figure 27 - The Heap Memory Usage of the OP client.

There are three more measurements in the "Details" area: **Used** is the amount of memory currently use by the OP client, **Committed** is the amount of memory that is guarantied to be available for the Java VM, and **Max** is the maximum amount of memory that the Java VM can use.

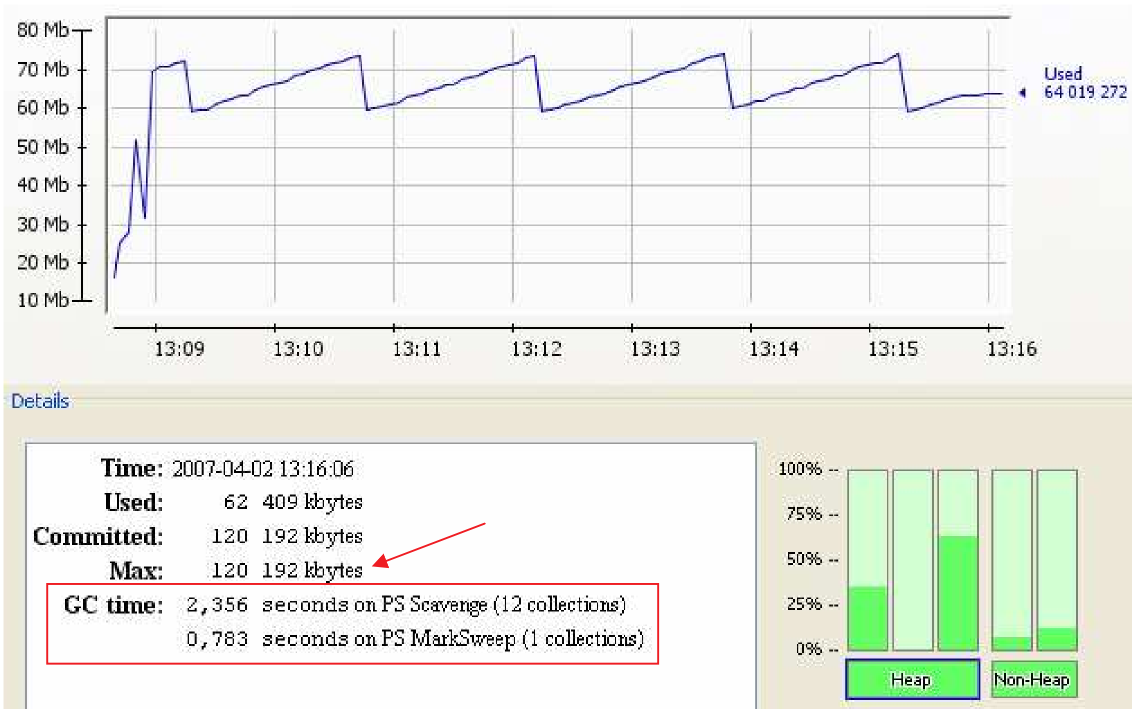
In Figure 28, the FIX client Heap Memory Usage is illustrated. We can see that around 90 Mb of memory is used in the beginning and by the end of the test 158 Mb of memory is used. This is far more memory usage than the OP client. Even though GC is performed five times this can not be seen as less heap memory usage in the graph. This can perhaps be explained if we look at the **Max** memory in the "Detail" area. The maximum memory allocated is 935Mb and as long as the FIX client is not close to this number it does not release any memory.

This hypothesis can be challenged if we lower the **Max** allowed memory to 120 Mb. In Figure 29, we can see that in this scenario approximately 10 Mb of memory are released when GC is invocated.



```
       Time: 2007-03-21 14:50:54
       Used:     154 603 kbytes
  Committed:     291 712 kbytes
        Max:     935 296 kbytes
    GC time:   1,045 seconds on PS Scavenge (4 collections)
               0,451 seconds on PS MarkSweep (1 collections)
```

Figure 28 - The Heap Memory Usage for the FIX client.



```
       Time: 2007-04-02 13:16:06
       Used:      62 409 kbytes
  Committed:     120 192 kbytes
        Max:     120 192 kbytes
    GC time:   2,356 seconds on PS Scavenge (12 collections)
               0,783 seconds on PS MarkSweep (1 collections)
```

Figure 29 - The Heap Memory Usage for the FIX client when Maximum memory is set to 120 Mb.

Figure 30 shows an overview of the Heap Memory Usage, the number of Threads and Classes used, and CPU usage when executing first the OP client followed by the FIX client. The red arrows mark the end of the OP client and the start of the FIX client.

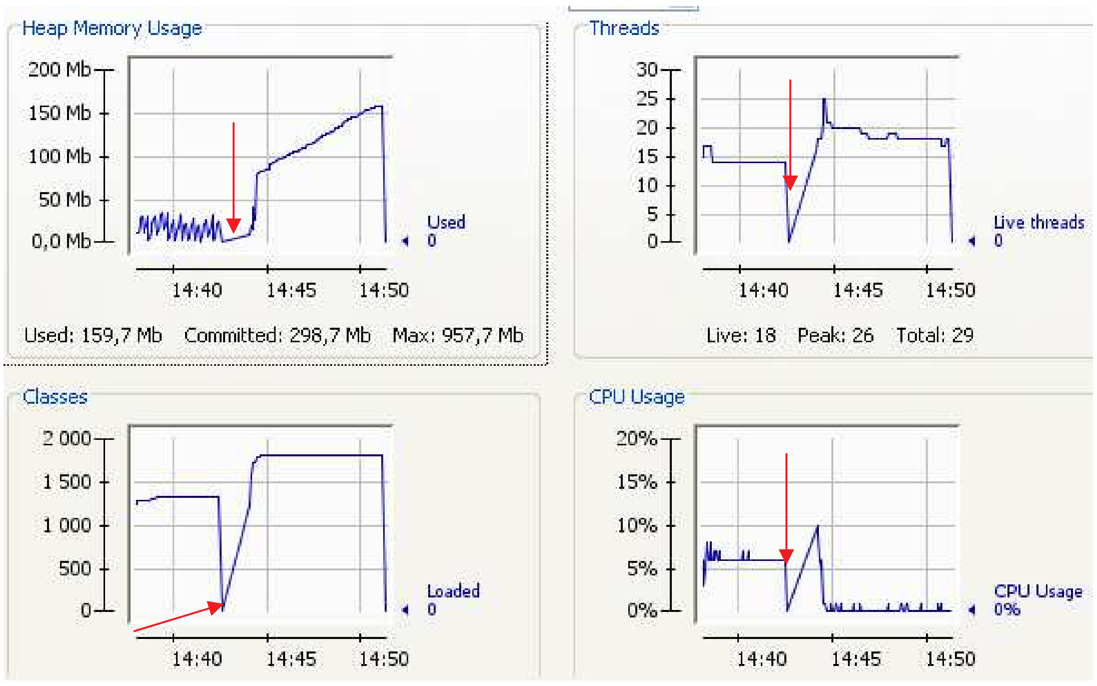


Figure 30 - Overview of the memory usage, the number of classes and threads, and the CPU usage.

As already stated, the FIX client uses far more memory than the OP client. One reason for this is that the FIX client uses both more threads and classes than the OP client. When we look at the CPU usage we can see that the OP client uses more CPU than the FIX client. The OP client uses around six percent of the CPU while the FIX client uses as low as 1 percent. When the FIX client is waiting for the orders to enter the market the CPU usages is down to zero.

3.3.6 Test summery

To summarize the findings of the evaluation study we can say that the OP client is both faster and uses less memory than the FIX client. However, when it comes to CPU utilization the OP client uses more CPU than the FIX client.

As shown, the GC invocations occur more often for the OP client than the FIX client. This could mean that the OP client uses objects that are short lived and resides in the **Young** segment. The FIX client, on the other hand, caches more objects, thus having more objects in the **Permanent** segment.

The CPU utilization could be related to the GC invocations. While the OP client has more invocations than the FIX client this could explain the higher CPU usage. However, the majority of Orc Software's customer insert around 5000 orders per day, thus, there is still safe to say that the OP client scale enough to be able to handle this volume of transactions.

In the finance sector speed is a very important aspect. This means that the OP client is preferable, since orders are placed faster and thus the customer's order is executed faster.

4 Conclusion and Future Work

In this section the conclusions that can be drawn based on this thesis are discussed. There are also some ideas presented to further develop the effectiveness of the parser/generator.

This thesis presented a solution for the initial problem. The solution meets the requirements, such as being well documented, thread safe, fast, small memory footprint, and implementing a JMS subset. In the toolkit provided to the customers, a sample application providing a few examples utilizing different OP messages will be included. This fulfills the desire by Orc Software to reduce the perceived difficulty in communicating with their Orc Server System.

The evaluation study showed that the OP parser/generator is both faster and more memory efficient than the FIX client which uses the FIX gateway. Thus, the OP parser/generator performs well enough to be competitive in the market; hence it is a good alternative to the other existing OP parser/generator.

There are alternatives for making an even more efficient OP parser/generator. For future work, an implementation where the OP specification is read as a template automatically and object stubs are created should be evaluated. This would make it easy to have objects for all messages and not have to maintain them when changes occur. In this way the customers have to know even less about the OP protocol, than they have to when using the solution presented in this thesis.

An extension to the parser/generator could be to implement a function that could handle, for example, FIX messages as well. FIX is a protocol that continues to increase in popularity which means that the parser/generator can be used in many environments. An extension to the parser/generator can be implemented in perhaps 160 man-hours.

To conclude, this thesis has shown that using of a parser/generator can lead to a high performance solution, which is easy to understand and correctly implement with limited effort.

5 References

[1] Wikipedia, http://en.wikipedia.org/wiki/Bond_%28finance%29, last accessed 2007-01-29.
[2] John C. Hull, *Options, Futures, & Other Derivatives Fourth Edition*, Prentice Hall, 2003.
[3] Orc Software, *Orc 5.2 Server Administration Manual*, September 2006.
[4] Orc Software, *Orc Protocol 5.2 Specification*, September 26, 2006.
[5] Steven John Metsker. *Building Parsers with Java*, Addison Wesley, 2001.
[6] Dick Grune and Ceriel J. Jacobs, *Parsing Techniques - A Practical Guide*, Ellis Horwoord Ltd, 1990.
[7] Andrew W. Appel and Jens Palsberg, *Modern Compiler Implementation in Java 2nd edition*, Cambridge University Press, 2002.
[8] Keith D Cooper and Linda Torczon, *Engineering A Compiler*, Morgan Kaufmann, 2003.
[9] Doug Lea and James Power, Concurrent programming in Java, Department of Computer Science National University of Ireland, *Maynooth, Formal Language Theory and Parsing*, http://www.cs.nuim.ie/~jpower/Courses/parsing/, last accessed 2006-12-27.
[10] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne, *Document Object Model (DOM) Level 2 Specification*, http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/DOM2-Core.pdf, last accessed 2007-01-10.
[11] The World Wide Web Consortium, http://www.w3.org/, last accessed 2007-01-30.
[12] Saxproject, http://www.saxproject.org/quickstart.html, last accessed 2007-01-10.
[13] Sourceforge.net, http://sourceforge.net/projects/sax/, last accessed 2007-01-26.
[14] JMS Tutorial, http://java.sun.com/products/jms/tutorial/1_3_1-fcs/doc/copyright.html, last accessed 2007-01-02.
[15] Sun Microsystems, Inc, *Java Naming and Directory Interface Application Programming Interface (JNDI API)*, http://java.sun.com/j2se/1.5/pdf/jndi.pdf, last accessed 2007-01-29.
[16] Theodore S. Norvell., *Javacc Tutorial*, Memorial University, http://www.engr.mun.ca/~theo/JavaCC-Tutorial/javacc-tutorial.pdf, last accessed 2007-01-08.
[17] Java.net, *JavaCC [tm]: Grammar Files*, https://javacc.dev.java.net/doc/javaccgrm.html, last accessed 2007-01-13.
[18] Java.net, *JavaCC [tm]: TokenManager MiniTutorial*, https://javacc.dev.java.net/doc/tokenmanager.html, last accessed 2007-01-15.
[19] Java.net, *JavaCC [tm]: LOOKAHEAD MiniTutorial*, https://javacc.dev.java.net/doc/lookahead.html, last accessed 2007-01-15.
[20] Java.net, *JavaCC [tm]: JJTree Reference Documentation*, https://javacc.dev.java.net/doc/JJTree.html, last accessed 2007-01-15.
[21] Howard Katz, *JavaCC, parse trees, and the XQuery grammar, Part 2*, http://www-128.ibm.com/developerworks/xml/library/x-javacc2/, last accessed 2007-01-15.
[22] Internet Assigned Number Authority, *Port Numbers*, http://www.iana.org/assignments/port-numbers, last accessed 2007-01-26.
[23] FixProtocol.org, *FIXimate 4.4*, http://www.fixprotocol.org/specifications/fix4.4fiximate/index.html, last accessed 2007-02-28.
[24] Java SE Monitoring and Management Guide, http://java.sun.com/javase/6/docs/technotes/guides/management/toc.html, last accessed 2007-03-16.
[25] Sun Developer Network, *Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine*, http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html#1.1.Sizing%20the%20Generations%7Coutline, last accessed 2007-04-03.

Appendix A - Abbreviations and Acronyms

OP	Orc Protocol
EMD	Environment Management Daemon
PMD	Port-Mapper daemon
FIX	Financial Information eXchange
HTML	Hypertext Markup Language
XML	Extensible Markup Language
DFA	Deterministic Finite Automaton
NFA	Non Deterministic Finite Automaton
CF	Context-Free
JMS	Java Message Service
SAX	Simple API for XML
DOM	Document Object Model
JNDI	Java Naming and Directory Interface
JavaCC	Java Compiler Compiler
BNF	Backus Naur Form

Appendix B – Quick Start Guide

This Quick Start Guide will describe how to setup a Trading Client Application connecting to the OP Server and using this connection to get different feed messages, and then send order_insert messages. This setup is illustrated in Figure 31, where the different OP messages that will be sent to the OP Server also are displayed.

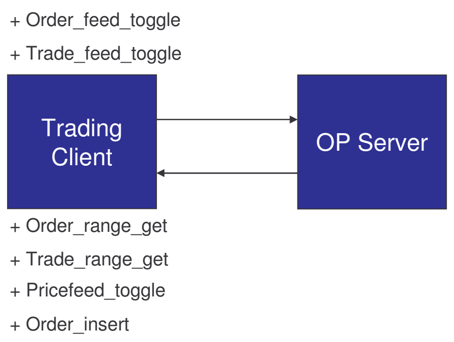


Figure 31 - The setup with a Trading Client Application and the OP Server.

B.1 *Trading Client*

This section explains how to create the Trading Client Application. First, we connect to the OP Server, and then we create both an order_feed_toggle message and a trade_feed_toggle message. Next, we will create an order_range_get message and a trade_range_get message. This will give us a view of the current status of the orders and trades. Last, we will create a price_feed_toggle message and then, using the prices in the price_feed messages, we will create an order_insert message.

B.1.1 Connect to the OP Server

To initiate a connection with the OP Server you first have to create a ConnectionFactory object. The input to the ConnectionFactory is the hostname and port of the OP Server.

```
connectionFactory = new ConnectionFactory( HOST, PORT);
```

The next step will be to create a Connection object using the ConnectionFactory object. If you provide a username and a password as input, a login message will automatically be sent to the OP Server. If a username and a password are not provided, you first have to create a login message before exchanging information with the OP Server.

```
connection = connectionFactory.createConnection(userName, password);
```

The last step in the initiate face is to create a Session object using the Connection object. There is no need for any input when creating a session. The Session object is later used to create a MapMessage, Producer, Consumer, Topic, and Subscriber.

```
session = connection.createSession();
```

B.1.2 Create an order_feed_toggle message

When we want to create an order_feed_toggle message we first have to create a Topic using the Session object. The input to the function createTopic() should be the same message type as the requested feed toggle message specified in the OP Specification except that you leave out "_toggle" in the message type.

```
Topic order_feed = session.createTopic("order_feed");
```

At this point there are no publisher sending messages to this Topic. Therefore, the next step is to create an order_feed_toggle message and send to the OP Server. The message that should be sent is a JMS message standard called MapMessage. This is done using the Session object.

```
MapMessage order_feed_toggle = session.createMapMessage();
```

When the MapMessage is created you first have to set the message type of the message. Use the same message type as specified in the OP Specification, except that you also in this case leave out the "_toggle" in the message type. If you want to be able to retrieve the reply message using a consumer (see section 1B.3) you have to provide a unique number as second input to the function setMessageType().

```
order_feed_toggle.setMessageType("order_feed", unique_Nr);
```

When this is done it is time to start filling the MapMessage with the key/value combinations and Dictionaries that are needed. The MapMessage follows the same message format as an OP message, thus, a particular Dictionary have to have certain key/value combinations. The code below shows how to insert a string into a MapMessage. In this case, the order_feed_toggle message will request order_feed messages from the Saxess market.

```
Order_feed_toggle.setString("market", "Saxess");
```

After the Topic and MapMessage are created it is time to create a Subscriber that will be used to register a Listener that will be listening to the asynchronous messages received by the Topic. Use the Session object to create a Subscriber using the createSubscriber() function. Both the Topic and the MapMessage should be as input.

```
subscriber = session.createSubscriber(order_feed, order_feed_toggle);
```

The next step is to create a Listener which should be registered with the Topic. This Listener must include a function named onMessage() that will have a MapMessage as input. The onMessage() function will execute when a feed message for this Listener is received by the Topic. In this case we don't have to do that because the Trading Client Application implements the Listener interface.

We use the created Subscriber to enable the listener to start listening to the Topic. This is done using the function setMessageListener(). The input in this example is the Trading client itself. When the Listener is registered with the Topic, the Topic will send the newly created MapMessage to the OP Server.

```
subscriber.setMessageListener(this);
```

When this is done you will start receiving feed messages from the OP Server. If you want to unsubscribe see section 1B.2.

B.1.3 Create an trade_feed_toggle message

To create a trade_feed_toggle message you use the same approach as when you created the order_feed_toggle message. Remember that the input when creating the Topic should be without the "_toggle".

```
Topic trade_feed = session.createTopic("trade_feed");
```

B.1.4 Create a order_range_get message

The first step is to create a MapMessage. This is, as explained, done using the Session object.

```
MapMessage order_range_get = session.createMapMessage();
```

When the MapMessage is created you first have to set the message type of the message. Use the same message type as specified in the OP Specification. As mentioned before, if you want to be able to retrieve the reply message using a consumer (see section 1B.3) you have to provide a unique number as second input to the function setMessageType().

```
order_range_get.setMessageType("order_range_get", unique_Nr);
```

When this is done it is time to start filling the MapMessage with the key/value combinations and Dictionaries that are needed. In this case, we want to receive all the orders for the date 2007-04-24 in the Saxess market that we have created.

```
order_range_get.setString("market", "Saxess");
order_range_get.setString("startdate", "2007-04-24");
order_range_get.setString("enddate", "2007-04-24");
order_range_get.setString("owner", userName);
```

When the MapMessage is loaded, it is time to send the message to the OP Server. First, you have to create a Producer that is responsible for sending the message. Then use the send() function, with the MapMessage as input, to send the message.

```
producer = session.createProducer();
producer.send(order_range_get);
```

Now the message is sent to the OP Server. To retrieve the reply message see section 1B.3.

B.1.5 Create a trade_range_get message

To send a trade_range_get message use the same approach as when creating an order_range_get message. If you don't provide a start and an end date the default will be the current date.

B.1.6 Create an pricefeed_toggle message

First, we have to create the Topic using the Session object.

```
pricefeed = session.createTopic("pricefeed");
```

In this scenario, the price_feeds that we are interested in are from the contract Ericsson B (feedcode=101) on the Saxess market. As before, start by creating a MapMessage, set the message type, and then load the message. In this example, we will create an instrument_id Dictionary where we put the feedcode, market, and currency using the setString() function. Next, we put the Dictionary in the MapMessage using the setDictionary() function.

```
MapMessage ericb = session.createMapMessage();
ericb.setMessageType("pricefeed", "ericb");

Dictionary instrument_id =new Dictionary();
instrument_id.setString("feedcode", "101");
instrument_id.setString("market", "Saxess");
instrument_id.setString("currency", "SEK");

ericb.setDictionary("instrument_id", instrument_id);
```

When the MapMessage is created you follow the same steps as you did when sending an order_feed and trade_feed messages.

B.1.7 Create a order_insert message

Now we want to use the prices we receive from the price_feed messages to buy Ericsson B stocks.

Price_feed messages from the OP Server are picked up by the onMessage() function in the listener. To check the latest price you use the getString() functions.

```
String last=message.getString("last");
```

The next step is to create an order_insert MapMessage buying 1000 Ericsson B stocks on the Saxess market for the latest price.

```
order_insert = session.createMapMessage();
order_insert.setMessageType("order_insert", unique_Nr);

order_insert.setString("activate", "exchange");

Dictionary order=new Dictionary();
order.setString("buy_or_sell","buy");
order.setString("price", last);
order.setString("volume", "1000");

Dictionary instrument_id=new Dictionary();
instrument_id.setString("market", "saxess");
instrument_id.setString("feedcode", "101");

order.setDictionary("instrument_id", instrument_id);

order_insert.setDictionary("order",order);
```

The MapMessage we have just created represents the OP message shown below. If we look inside the MapMessage it will look as displayed in Table 8.

```
{ message_info = { message_type = order_insert } | activate =
exchange | order = { buy_or_sell = buy | instrument_id = {
market=Saxess | FeedCode=101} | Price = last | Volume = 1000 }}
```

MapMessage	
Message_info	**Dictionary**
Active	“Exchange”
Order	**Dictionary**
Message_info	
Message_type	“order_insert”
Order	
Volume	“1000”
Price	“10”
Buy_or_sell	“buy”
Instrument_id	**Dictionary**
Instrument_id	
Market	“Saxess”
Feedcode	“101”

Table 8 - An order insert MapMessage.

When the MapMessage is loaded, it is time to send the message to the OP Server. This is done the same way as when sending the order_range_get and trade_range_get messages.

B.2 Unsubscribe from a Topic

To unsubscribe from a feed message use the Topic object and the function unsubscribe(). The input to this function is the Listener and the MapMessage used when subscribing. For example, to unsubscribe the feed message created in section B.1.2 we would write:

```
order_feed.unsubscribe(this , order_feed_toggle);
```

B.3 Retrieving a reply message

To be able to retrieve a reply message from the OP Server you first have to create a consumer object. This is done using the Session object.

```
consumer=session.createConsumer();
```

When the consumer is created you can start receiving messages. Use the function receive() with a unique number as input. This unique number should be the same as you used when sending the message to the OP Server.

```
MapMessage aReply = consumer.receive(unique_Nr);
```

When the message is received you can start using the information stored in the MapMessage. The functions getDictionary() and getString() will give you the Dictionaries and key/value combinations needed.

B.4 Closing the connection

When you are finished exchanging information with the OP Server you need to close the connection. This is very simple using the Connection object.

```
connection.close();
```

COS/CCS 2007-17