UNIVERSITY OF TARTU
Faculty of Mathematics and Computer Science
Institute of Computer Science
Specialty of Information Technology

Oleg Knut

A Low Level Virtual Machine Back-end for F2F Computing

Master Thesis (20 CP)

Supervisor: Ulrich Norbisrath

Author: …….. "……" January 2010

Supervisor: ……... "……" January 2010

Allow to Defense:
Professor .....…….… "……" January 2010

TARTU 2010

Contents

Acknowledgments

I would like to thank my parents, my mom Galina and dad Igor. As well my wife Anastassia, whom I love very much and without her support I would never get so far.

I would like to thank whole Distributed Systems group at University of Tartu for giving me this challenging task to solve. Special thanks goes to F2F project main contributors, my supervisor Ulrich Norbisrath and Artjom Lind, who supported me very much during the research, development and implementation of the thesis subject.

Chapter 1

Introduction

Behind acronym F2F[1] stay the words Friend to Friend, which is a simple Peer to Peer network allowing users (friends) to share computational power between peers. F2F is written in the University of Tartu and is used for academic needs. Main areas of usage are to show students in an easy form how to create distributed networks and exchange information between nodes. Nodes can initiate "jobs", which is machine code that should be activated on every friend node and this way uses the friend's computer to create some result. It will be transferred back to the initiator or shared between next nodes in the network (based on job logic).

The current approach of the F2F supports only one programming language to create jobs. This language is Python[2] – dynamic scripting language. It lets easily write some jobs for a network of friends, but there is a direct statement that it is not enough to be a universal tool for academic needs.

The idea of the thesis comes from the wish to let students create jobs in a language they would like to write in and without requiring them to have a knowledge of Python. During the research done by the Distributed Systems work group, the framework they chose to use was LLVM[3], which seemed to be promising and satisfied most of the needs.

The LLVM technology allows you create, what they call, IR (intermediate representation)[4]. This is a language that can be translated into machine operations, using only the instructions given by LLVM the framework. Creators of some popular languages already made compilers that allow creating IR from C, C++, Fortran, Java and others.

The general subject of the thesis is to create a virtual machine, the so-called sandbox, which will be allowed to run jobs that are shared inside F2F network. Also, the author will look into security aspects of the topic, to be ensured that no harmful code can be run on client computers.

1.1 Area of Research

Nowadays everything is so heavily computerized that there are no industries left where computers do not help humans. They get more powerful every day and scientists are not sleeping in laboratories while inventing new processors and computer hardware. A positive news is that in recent years they think more

and more about the environment, and try to invent techniques that use less power, effort and materials to produce.

Unfortunately, it is still hard to maximize the usage of computers. The reason for this is most probably the lack of experience in working with electronic devices. Yes it is true, if we compare the time electronic devices have been used to the time that human beings have been living on the Earth. It is indisputable that one day we reach the top of experience, but now we still have a long way to go.

The main direction of this academic work is the sharing of computational power. In this area there is a wide selection of tools and techniques which allow users to share computational power between each other. It should be clear to everyone that such techniques will increase the proper usage of every watt that is spent on producing computations. A common situation is when computers are used in organizations during office hours and idle for the weekends or night time. Electricity spent on this downtime is just wasted. In this area there are many improvements that can be made.

For scientific needs maybe it is not enough to just use single computers that might be free during a limited time. There is a solution for that - to connect computers into a network and create a distributed cluster that is not used in a particular time. This way it is possible to use computers very efficiently and leave a particular group of people without a need for their own cluster, supercomputer or any other computational devices.

Letting others use your computers brings up a lot of questions. The wide-spread usage of Internet allows us to connect almost all computers in the world into one network. But how safe is that? Safety and security is one of the main dilemmas. It is clear that allowing others to connect your computer with full trust and access will end up with an information leak or even a hardware crash. In this case there is no point in sharing computational power, because it is not worth it and the risk is too high. Probably there is no easy way, and the challenge is to create a network that is secure enough and highly reliable for end users.

The goals of scientists who work in these areas are:

- To encapsulate the real usage of hardware and its sharing context;
- To ensure against maximum possible crashes or failovers, which might be caused by the sharing;
- To create time and rights management tools that would be easy for the end-users to work with, and which would encourage them to share computational power with others.

To achieve the listed goals, a lot of work has to be done. But it is just a matter of time when engineers and developers will be able to present a tool which will satisfy all the needs.

1.2 Back-end

Well designed software should always be divided into logical parts for easier maintenance. Common design of application has to be divided into at least two parts: front-end and back-end. Back-end is the part that is responsible for processes that do not require any user interface. It usually interacts with the front-end,

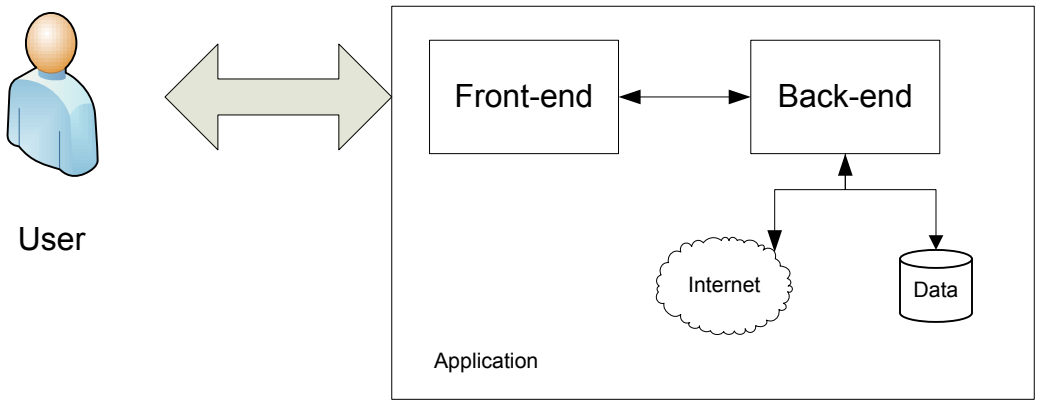

Figure 1.1: Back-end and front-end in action

communication interfaces and APIs, which is in turn responsible for representation of job is done. As seen on Figure 1.1 the structure of the application becomes more clear and easier to understand.

Typical back-end tasks are:

- Business logic handling - enterprise development;
- Code generation, execution and optimization - compilers;
- Logical core - operation systems.

In this thesis we will look into the development process of a typical back-end. We will go through the basic steps that need to be performed to achieve the goal. The back-end will be responsible for code transformation, basic optimization, security and execution. All previous keywords can be gather under one term - Virtual machine[5].

1.3 Structure of the Work

A brief explanation of the thesis will follow in this short paragraph, to give a better overview of the work that has been done. In Chapter 2 we will talk about the common problems of peer-to-peer networks, discuss virtualization, compare technologies and describe the project that the back-end will be built for. Also some theoretical aspects will be delivered: pluses and minuses, the future of the project, common areas of usage. Chapter 3 will follow with a description of the back-end design, generally explaining why LLVM was used, what is platform independency and how it will be implemented. Other topics discussed in depth are virtual machines, just-in-time compiling and communication with front-end. The next chapter is a step-by-step instruction of the implementation with explanations of decisions and its analysis. In the end take a look at the real life examples of the job done and how it can be used.

Chapter 2

Simple Peer-to-Peer or Friend-to-Friend

2.1 F2F or P2P for Friends

A friend is a person whom you know and trust. Such a person should be someone with whom you want to share your emotions. In this kind of relations trust is very important. You have to be sure that your friend will support you in any situation, not just share the good moments with you. These definitions were taken over to computer science and used as a basis in this cutting edge technology.

Peer-to-peer network consist of peers, hardware units that have a special software installed and are somehow connected to each other. Connections between computers can be divided into two main groups - wireline and wireless. Subgroups of them are well known data transmission channels: Integrated Services Data Network (ISDN), Asymmetric Digital Subscriber Line (ADSL), modems, access points, etc. Generally, data exchange takes place either via the Internet or some sort of private networks.

A typical P2P network is used to share data in text or binary. Usually the goal is to create a transmission channel that does not have any intermediate points. This approach allows users to be straight connected, which assumes faster and safer data exchange. Usual peer-to-peer network does not care about the number of peers in the node, but is more interested in the content every peer can share. Without a good content

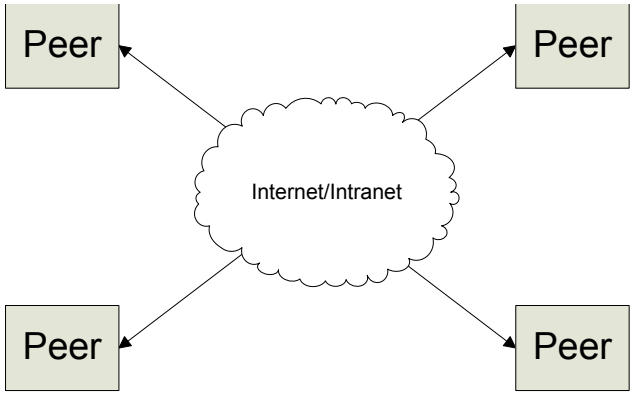

Figure 2.1: P2P workflow

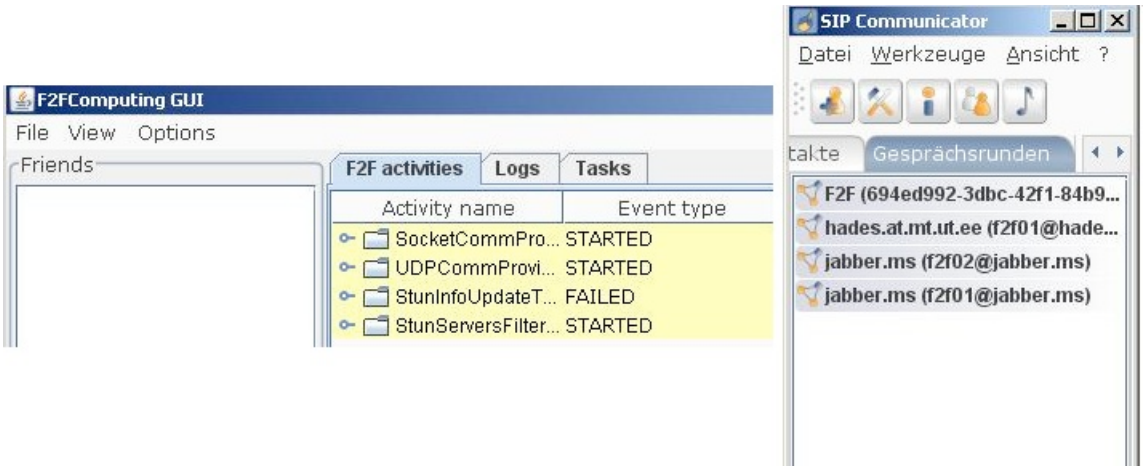

Figure 2.2: F2F as plug-in in SIP Communication

to share there is no point in such networks. In some cases the content can be computer resources. This makes a point of the sharing network without any particular need for valuable content.[6]

Who is interested in sharing computer resources and what exactly will be shared? Most probably one of the main sharing options is the computational power of computer processors. Typical groups who may be interested in having access to other users' resources are young researchers, students, and mid-size businesses. This is very efficient when you do not need your own powerful hardware - you just need a lot of friends (peers).

Friend-to-friend (F2F) computing network was invented for bringing computational power sharing to the masses. F2F was founded in University of Tartu in 2007 by Ulrich Norbisrath on behalf of the Distributed Systems work group. Its main goal was to simplify the creation of such networks and make the process transparent for the students. Currently, the software is used for academic purposes, allowing students to easily understand the workflow and architecture of the modern distributed system.

2.2 Version History

The first version of F2F was written in Java and implemented as a plug-in for SIP Communicator. It was a standalone application with multi-platform (Linux, Windows, Mac) and multi-protocol (XMPP, ICQ, MSN, etc.) support.

Considering the orientation of the product (scientific computations), the implementation in Java quickly became unusable. Java is a great object-oriented language, but it is run inside a virtual machine, which slows it down and does not allow using power resources reasonably. When planning the new version of F2F, increase in speed was needed, and the choice fell on rewriting it in C. This resolution took communication and execution layers to a very low level and greatly increased the speed. Like other popular software, the new version had to have a code name and it was decided to name new version - F2F New Generation (NG).

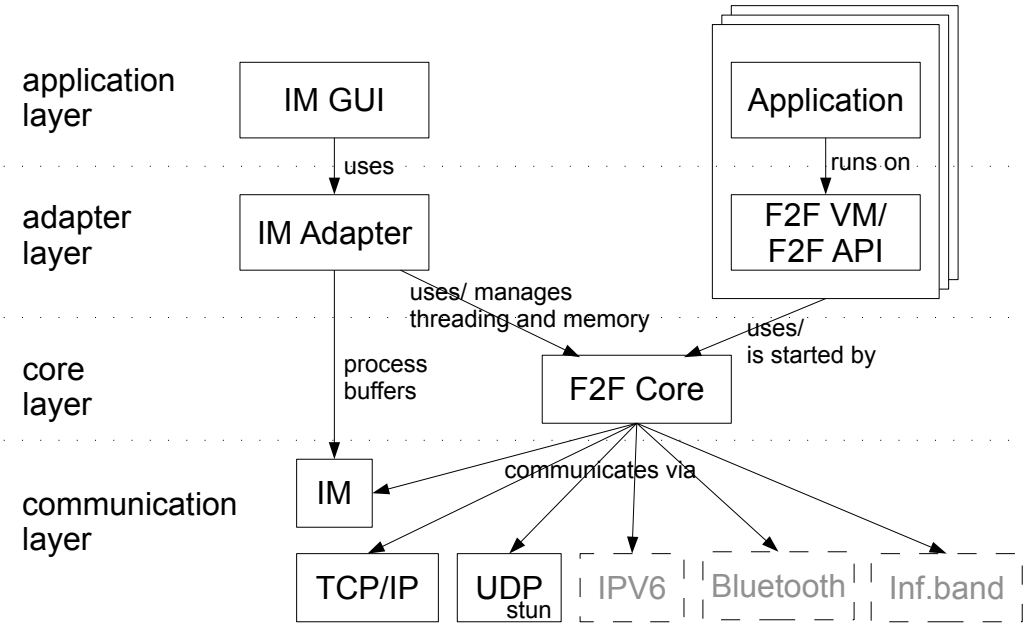

Figure 2.3: New model of F2F

2.3 F2F New Generation

Active development phase moved from alpha to beta status in December of 2009. In this version core was totally separated from main code, which give almost free hands for developers to write their own communication adapters, to be not restricted with any particular software like it was with previous version (SIP Communicator and Java).

Figure 2.3 illustrates the new application architecture. As mentioned before, new application was changing an implementation language and had to rewritten from scratch in C language. Following sections in this chapter will explain more in depth different application modules and layers.

2.4 Communication Principles

In general, F2F tries to solve in its own manner the two communication problems that every distributed system has to deal with. These are:

- Establishing direct connection between two nodes;
- In case establishing a direct connection is not possible, using indirect connection;
- Content delivery.

The next three sections will describe application architecture against mentioned problems.

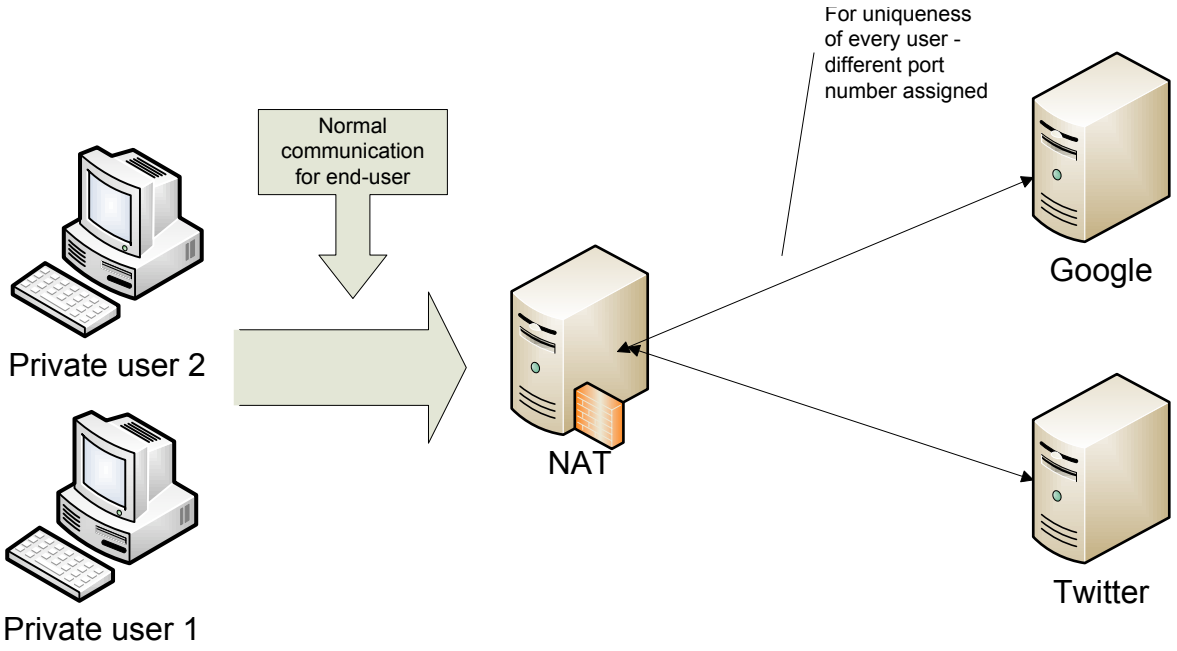

Figure 2.4: Communication with the Internet through NAT

2.4.1 Establishing Direct Connection

Establishing a direct connection means that every part that is participating in communication knows the addresses of the others. This not always a trivial task, because achieving direct communication is not always easy. There are a lot of obstacles that prevent users from making it happen. It is a very common situation when users share their public addresses. An ordinary Internet provider infrastructure collects a defined group of service users into a private network and allows access to the Internet using same public Internet Protocol (IP) address[7]. Then communication goes through so called Network Address Translators (NAT)[8] gateways, which handles every outgoing and ingoing packet. Such restrictions are used due to the small number of the IPv4 addresses.

There are a lot of different possibilities on how to still be able establish a connection between nodes. The general term for them is "NAT traversal", which are a number of techniques that allow establishing a connection between users behind NAT[9]. Such techniques can be easily implemented as one of the communication adapters for F2F.

2.4.2 Establishing Indirect Connection

Sometime it is not possible to detect the addresses of the nodes. In this case, a good distributed system should use public communication channels. These are usually public communication services like chats. Well-known channels are very popular chat software (MSN, ICQ, GTalk, Skype), which at the same time have their open protocols for communication. Such networks are always available for almost every Internet user, because they have open servers that act like relay for every message that has to be delivered. In this case there is extra time needed for delivery and due to the high load of such servers the content delivery can take much longer than a direct connection.

2.4.3 Content Delivery

Depending on the possibilities of the communication adapters, F2F has to always choose the fastest way available. For example, the slowest way that can be used is to relay content through some public chat servers (MSN, ICQ, etc.), which have a long delay due tohigh load. Next is using the same thing, but with your own private server (XMPP, IRC) - this will definitely increase productivity, but can still be slow for a bigger amount of data. For the quickest way there is a need for direct connection (both sides have to know each other's addresses and they have to be fixed). Using direct connection it is possible to use well-known fast protocols for data sharing like Transmission Control Protocol over Internet Protocol (TCP/IP)[10], or User Datagram Protocol (UDP)[11].

F2F has an abstract layer that allows delivering content independently from the communication channel. Communication adapter is responsible for content integrity and for any other problems that might come up during delivery (packet loss, bad signature, denial of service, etc.).

One content package of F2F abstract delivery layer should indicate the following fields:

- Recipient (single, group) - determines where the content should be delivered;
- Sender - the initiator of the content sharing;
- Content handler identification - identifies which handler needs to be used when content is delivered;
- Loading status - current status of loading the message;
- Content - partial or full content of the package.

Based on the fields described above it is possible to make the core independent from the communication adapters.

2.5 Job

When the connection has been set up and the distributed system is functioning, the participants should somehow be instructed with tasks they will perform. The term 'job' is used in this case, which is a single instruction set that will be delivered and executed on every node participating in the group. Job is also responsible for further communication between nodes after the connection has been established. After the initialization period there are mechanisms in the client that store information about the group and participants taking part in the current network of friends. Several functions are provided to allow the job to automatically communicate with other peers by delivering information to them or requesting some information from them.

To get instructions executed on every node they have to pass a specific life cycle in F2F.

1. Job is received;
2. Communication adapter makes sure that the job is not partial, otherwise waits until the whole job is delivered;

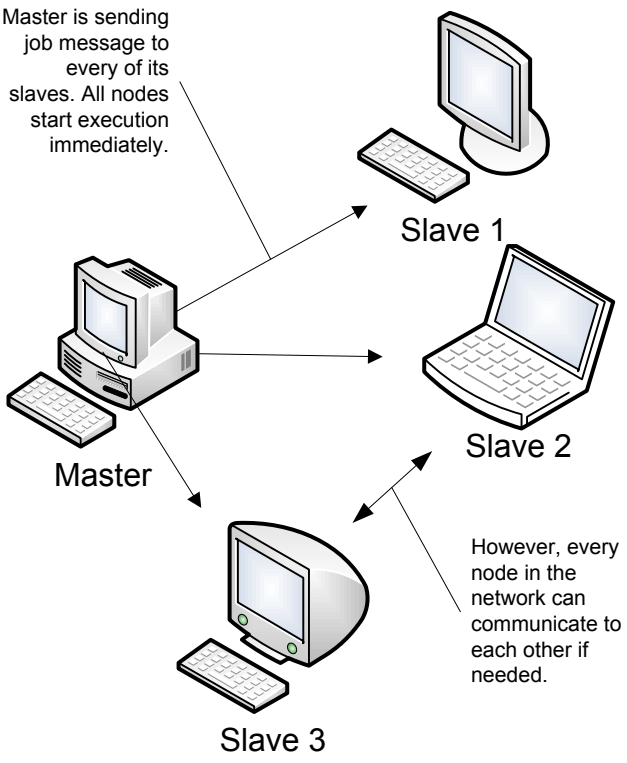

Figure 2.5: Sending job to the nodes

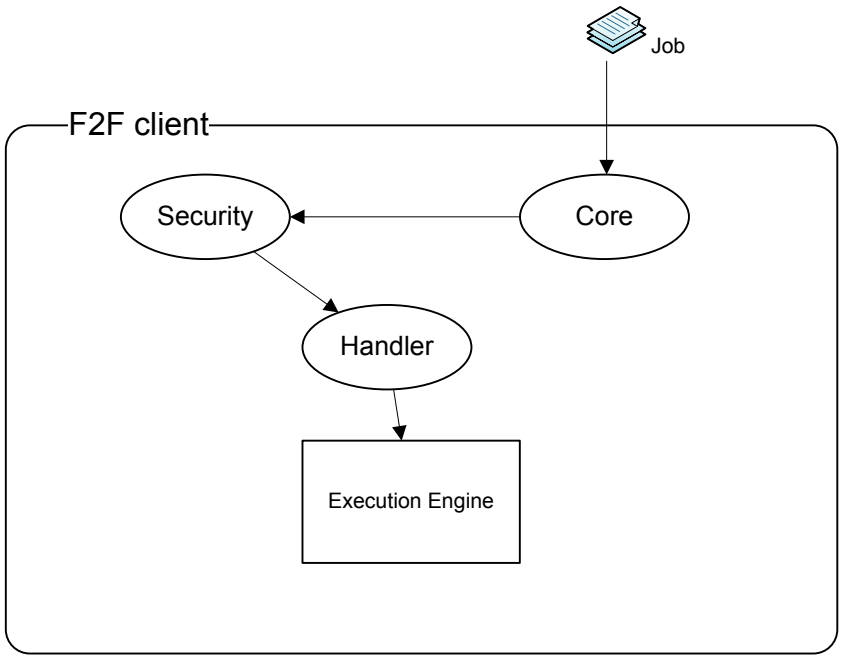

Figure 2.6: Job life cycle inside client application

Function prototype	Description
F2FWord32 f2fJobGetInitiator(F2FJob* job)	Return identification number of the initiator in this group.
F2FWord32 f2fJobGetMyself(F2FJob* job)	Return self id.
F2FWord32 f2fJobGetPeerCount(F2FJob* job)	Return size of the group currently participating.
void f2fJobLog(F2FJob* job, const char* jobLog)	Creates output to the trace. Useful for debugging.
char* f2fJobGetContentPt(F2FJob* job)	Return content of the job.
F2FWord32 f2fJobWaitForMessage(F2FJob* job, F2FString buffer, F2FSize size, F2FWord32* sender)	Wait for receiving a message.
F2FWord32 f2fJobSendMessage(F2FJob* job, F2FWord32 dest, const F2FString msg, F2FSize size)	Send message out either to particular recipient or whole group.

Table 2.1: Provided functions for communication

3. Job is passed to security module (decrypted, verified);
4. Job handler executes the code in the execution engine;
5. Handler waits for job to be finished (during this time the job is able to communicate with other peers in the group);
6. When the job is finished the client releases the virtual machine and waits for another job to receive.

2.5.1 Handler Functions

To allow communication between nodes, special functions are provided for a job to use. Through this interface it is possible to send and receive messages from others and also request some information about self. In Table 2.1 the provided functions are listed with a brief explanation of their task.

Using these functions in the job code it is possible to communicate with the whole group and receive messages from them during the job execution in the handler.

2.6 Core

In the context of software development, core is a central part of the code which implements the functionality of the principal. It participates in the whole life cycle of the application and includes functions that are required globally and/or is responsible for linking and unlinking different modules during execution.

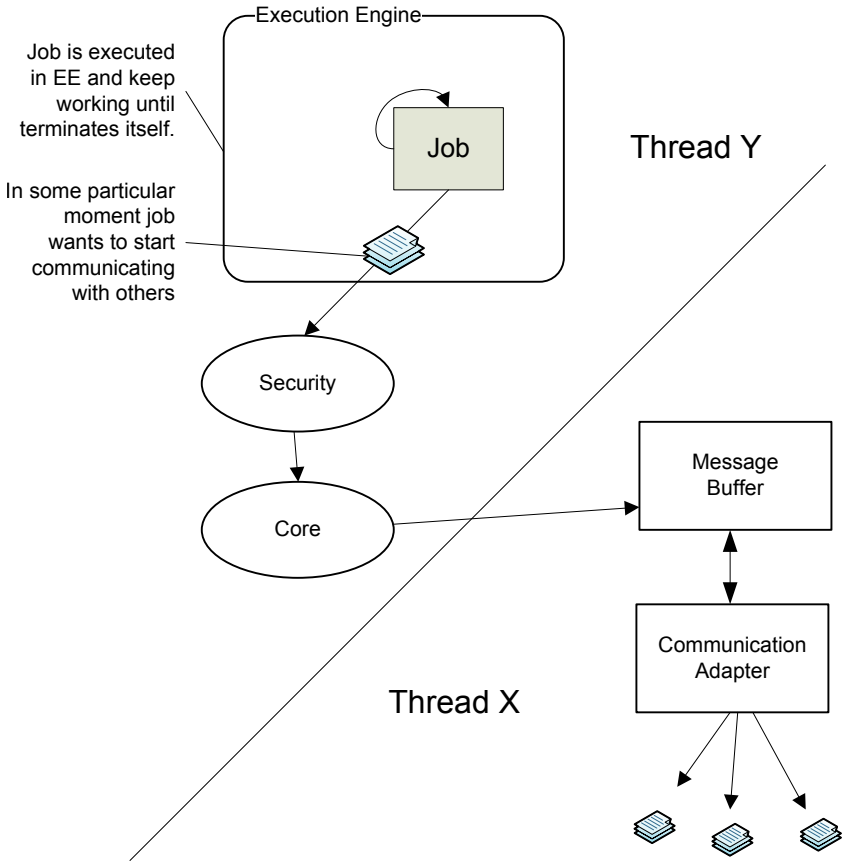

Figure 2.7: Job communication

The core in F2F follows these rules and additionally, in the mean time reacts as a buffer for all incoming and outgoing messages.

The main contribution here is to be the head of data transferring in the network. Core implements Message Passing Interface (MPI)[12] that allows computers to communicate with each other.

2.7 Security in F2F

This complicated topic can be generally divided into two parts that handle security in their own manner. The following subsections will describe in general how F2F deals with security aspects.

2.7.1 Secure Message Transfer

To secure the transferring of information between nodes, encryption and decryption mechanism was proposed for F2F. During the writing of this work the solution was only proposed and did not have a real implementation yet.

Off-the-Record (OTR) message encryption support is planned. Such messaging allows having private conversations over the Internet. Implementation of the described algorithm is either already part of the popular instant messengers or is provided as library in many popular languages (Java, C, C++). Actual implementation of the OTR is not enough to complete the task, because it provides the possibility of secure message transferring in single pairs. In F2F, Group Off-the-Record (GOTR) will be used, to allow secure message transferring in the group with a shared key.

2.7.2 Client-side Sandbox

Why does the client or, in this context, friend need to be secured?

1. Not all friends are your real friends;
2. You still want to be sure that nothing can happen to your stored data and hardware;
3. You do not want to share some of the content on your computer ;
4. System crashes because of bugs or harmful code have to be minimized.

One possibility to solve this problem is to use virtualization techniques. The so-called “Sandbox” application, which creates a chest over it and allows working with computer resources only through its Application Programming Interface (API). It keeps an eye on every move of the code which is run inside it and tries to minimize the risk.

Sandbox is a term for the so-called Application Virtual Machine (VM). Started in a single process, application inside the Operation System (OS), that creates a proper environment that can run another code inside, by reading it from memory or other storage devices. VM as an input requires some sort of high-level language code or some abstraction (bytecode) that can be translated into machine-code, which will be already understandable for the computer hardware where application is run.

The most known virtual machines are:

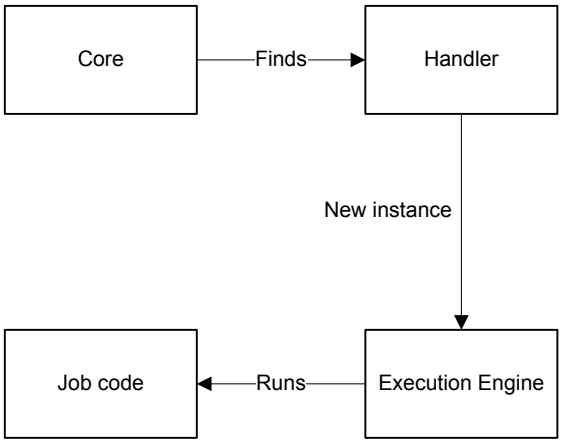

Figure 2.8: Job execution in back-end

- Java Virtual Machine (JVM)[13] - created and contributed by Sun Corporation, to allow running code written in Java language. Java abstract language (Java bytecode) compiled from code written in Java is absolutely platform-independent and can be run without any changes on every platform that is supported by JVM;
- Common Language Runtime (CLR)[14] - created and contributed by Microsoft Corporation to allow running the Common Intermediate Language (CIL), which is also a platform-independent low abstract language to support .NET Framework languages (C#, VB.NET, etc.). Previously CIL could only run on Windows family operating systems, but thanks to open-source communities, it is also possible to run it on Linux Operation System (Mono Project)[15]. Thanks to this it can also be a multi-platform virtual machine.

2.8 Back-ends for Execution

This term is used for the piece of code in the job handler module that will run the code of the job. The part responsible for running the code is called Execution Engine (EE).

Back-ends that are already implemented in F2F are:

- Python VM for execution of Python script code;
- Low Level Virtual Machine (LLVM) VM for execution of LLVM intermediate representation language.

Next chapters will explain the basics of the LLVM structure and the advantages of this implementation with others back-ends.

2.9 Sample Scenarios

F2F provides the possibility to share the computational power between friends in the network. It should be clear that it has to be used somehow. Scientists are the ones who is interested in creation of distributed

networks and tasks where they will try to solve very different tasks, for example from mathematical theory proving to global climate warming visualization. Some of the tasks can be arranged into typical categories that are described below.

- Controller - there are different kinds of problems that might be caused by different computer platforms and hardware manufacturers. Job can execute benchmark tests on nodes to verify that all of the nodes can execute the code correctly and can be reliable in some future tasks;
- Information gathering - the initiator of the job is the master of group. By submitting a job to the network group, all other nodes become the slaves of initiator. Master can ask slaves to send some particular instructions and send results back to the master. When using this type of job there is no communication between slaves, they always communicate directly with the master.
- Distributed computation - is when each node in the network participate in the communication equally and there is no master node and results are needed for every node or nodes that execute a particular task to continue with computation of another.

To summarize, let us take an example where F2F can be used. Imagine that you need to render a clip with a simulation of an atomic bomb explosion. There are a lot of formulas and algorithms for doing that. But to start doing this on your personal computer, you will need to do a cycle and render every frame one by one. If you have a friends network, you can submit a job that will divide rendering between nodes, and then every node will have its own frame to render. When a frame is finished, the content of the frame can be sent to the initiator and another frame can be requested for rendering. Using such approach we can gather more than one frame at once and now the delivery of the whole clip is just a matter of nodes' count in the network. The more friends you have, faster it gets done.

Chapter 3

LLVM Back-end

At the time this work had no meaning or any structured ideas about the new back-end, the F2F network already had one back-end for dynamic Python code execution. It was not an ideal solution, because supported job executions were written only in Python and it was tightly connected with a client application which was also written in Python. This approach did not allow creating a good modular system, where the job handler and the communication layer would not be connected to each other. It was clear that the project was in need of something more universal. There was a need for a new back-end that would support the possibility of writing jobs in more than one language and with better extensibility.

The main contributors of F2F arrived at the idea that there is a need for a new back-end that has to be very efficient. Having already an experience in writing jobs in Java, it would be logical to start thinking of creating a Java Virtual Machine (JVM) back-end for execution of Java bytecode. In fact, this solution was not ideal for a number of reasons:

1. JVM is virtual machine with a long history and is highly reliable. However, it is not that easy to integrate it into a third party applications, because the producer does not support it natively;
2. Implementation of such a back-end seemed to be very time-consuming and not very clear.

Another important thing was to support languages that are very popular in the science world, so developers who already have implementations of some algorithms will not have to rewrite them again or learn new programming languages. By the fact these languages are FORTRAN and C. The new application should support job writing in these languages and hopefully provide good performance and optimization possibility.

It was decided to look around new technologies and find out if there is a tool that could satisfy F2F's needs and would be more transparent in the developing of the virtual machine. At that time, decision fell on a highly promising project which seemed to satisfy most of the needs and, by description, was in fact more specialized than JVM. The project name was LLVM Compiler Infrastructure.

Figure 3.1: LLVM official logo

3.1 Low Level Virtual Machine

The first version of the Low Level Virtual Machine (LLVM) project was originally developed as a master's thesis in the University of Illinois by Chris Lattner in 2000. It is a set of code that can generally be called compiler infrastructure. It includes sources and tools that help the developer build his own compiler, analyzer, interpreter or optimizer. It is widely used in all kinds of projects by giants like Apple, Google and Adobe. Nowadays, the main evangelist of the LLVM is Apple Inc. who uses this product in the latest versions of their operation systems (Mac OS X family) for personal computers and hand-held gadgets/devices.

More in depth this set provides a number of reusable components for building compilers. It is worth mentioning that the advantage of using such components is that they reduce the time and costs spent on creating a new compiler and are highly effective, because of the well-tested and proven code. It makes life easier when there is a need to use static compilation techniques. Using the framework it has never been so easy to create one's own optimizer for better performance.

The main version of LLVM consists of:

1. Virtual instruction set handler - handled instructions are platform-independent, they will be resolved equally by all computer processors;
2. Collection of integrated libraries - allow performing operations with the virtual instruction set or executing it dynamically on the target platform;

3. Collection of tools to work with instructions - a number of tools that allow working with the virtual instruction set and produce static libraries or executable machine code. Available tools can perform the following actions with code: assembly, disassembly, optimization, linking, archiving, profiling, analysis.

It is worth explaining that there was a number of reasons why the LLVM was chosen to build a new back-end:

- Highly effective tools for creating an engine for execution on many platforms (x86, x86-64, ARM, PowerPC, SPARC, MIPS, IA-64, Alpha);
  - Such variety of platforms give flexibility to F2F and allow creating clients that will work on most operation systems and hardware.
- Libraries for creating applications that allow dynamic execution (Just-in-Time compilation);
  - Exactly what is needed for new job handler back-end, a library to create a real-time compilator and intepretator.
- The framework allows code optimization and profiling;
  - This one is definitely the case for further development. Profile should help verify application execution and also track down job execution to help analyze it.
  - Optimization techniques allow making the execution of binary code faster and prevent typical mistakes.
- Wide range of front-ends available for generating virtual instruction sets for LLVM;
  - Adding multilanguage support to F2F was the main goal of the thesis.
- Very active developers community, which is supported by the giants of the computer industry;
- Distributed under Illinois Open Source License[16], which means free access to the source code.

LLVM is a cutting edge technology that is used more and more all over the world in projects with various types of complexity. During a short period of time it has been proven that this is not just another student project and got supported by very big corporations, who included this technology into the tools they use every day.

3.2 Intermediate Representation

LLVM has its own form of code representation. They call it Intermediate Representation (IR) - the set of virtual instructions. This is input/output language for LLVM, accepted for code generation and brought to simplify analysis and optimizations. The IR can have a different forms for its easier maintenance:

1. In memory - hierarchical object model created in Random Access Memory (RAM);

2. As binary - not human readable bitcode; suitable for quick execution in a virtual machine;
3. As text - a human readable form of IR, allows seeing and understanding what is actually going on during code execution.

IR was designed as a low-level language, but at the same time it has the extensibility and effectiveness of high-level languages. LLVM team is trying to keep this language as low as possible, so it can be a front-end for maximum possible languages.

The features of IR are:

- Low level instructions with a strong type definition;
- Static Single Assignment (SSA)[17] form, well-known representation for compilers, used by many popular compilers;
- Abstraction of hardware registers;
- Memory model with own implementation of stack and heap;
- Reusable functions support.

To have a better understanding of a typical transformation into SSA form, let us take a look at the Listing 3.1 with a small C function first.

```
1  int fac(int digit)
2  {
3          int answer = 1;
4          int i;
5          for(i = 2; i <= digit; i++)
6          {
7                  answer = answer * i;
8          }
9          return answer;
10 }
```

Listing 3.1: Example C function

Using LLVM-GCC compiler we can produce LLVM IR from the C code and in Listing3.2 is the result that is already in SSA form. Let us look more into the details of what was produced.

The first line starts with a definition of the function (instruction *define*). The definition has to include return type (*i32*), name (has to start with @) and entry parameters (pairs of type and name starting with *%*). Inside the body of the function there are 3 labels defined (no indent and ends with *:*). Entry label is the entry point of the function and will be executed by function call automatically. Entry point has a simple check inside, to make sure that other label code is necessary or if it is better to execute return label and finish function execution. The check is done with the *br* instruction by controlling the value of *%tmp919* and deciding which label to go to (*i1* is a type which is integer with the size of one bit, which

```
1  define i32 @fac(i32 %digit) nounwind
2  {
3  entry:
4          %tmp919 = icmp slt i32 %digit, 2
5          br i1 %tmp919, label %return, label %loop
6  loop:
7          %indvar = phi i32 [ 0, %entry ], [ %indvar.next, %loop ]
8          %answer.0.reg2mem.0 = phi i32 [ 1, %entry ], [ %tmp3, %loop ]
9          %i.0.reg2mem.0 = add i32 %indvar, 2
10         %tmp5 = add i32 %indvar, 3
11         %tmp3 = mul i32 %answer.0.reg2mem.0, %i.0.reg2mem.0
12         %tmp9 = icmp sgt i32 %tmp5, %digit
13         %indvar.next = add i32 %indvar, 1
14         br i1 %tmp9, label %return, label %loop
15 return:
16         %answer.0.reg2mem.1 = phi i32 [ 1, %entry ], [ %tmp3, %loop ]
17         ret i32 %answer.0.reg2mem.1
18 }
```

Listing 3.2: Example transformed into IR

can be interpreted as being equal to a Boolean value). Next, look at the body of the *loop* label, which implements the execution of the *for* cycle. In the end of that instruction is *br* (short for 'branch') which checks if it is time to move into *return* label, where the final statement is returned out of the function with the *ret* instruction.

In general, LLVM IR is composed of modules - the object files. Each may include the following parts: meta-data, variable definitions (globals, locals), function definitions. Meta-data may include some sort of special information, provide possibility to attach arbitrary data to the code without a need of changing program behaviour. Variables that are defined globally are specified with @ in front to distinguish them from local variables, which always start with the % symbol. It is possible to also specify the linkage type of each variable. There are many different linkage options, including *internal*, *private*, *import*, *common*, *weak*, etc. The function definition includes in the body a set of labels with instructions in them. Every function label inside the function should always end either with return from the function (*ret* instruction) or stepping into another label (*br* or *switch* instruction). Functions consists of instructions, which take value type and variable as arguments. Functions consists of the instructions, which take as an arguments value type and variable.

3.2.1 Value Types

As mentioned previously, the LLVM IR is a strongly typed low-level language. In this section let us take a look at some of the types that are used in LLVM. Table 3.1 gives a brief overview of types used in this language[4]. Types are divided into two general categories - primitive types (simple types, never include other types in them) and derived types (complex types that can be sets of other types).

IR type	Name	Description
i[1..]	Integer (primitive)	It is possible to create a digit with a size of specified bits. Specifying a number of bits is the number of bits, that will be used. E.g. *i1*, can hold only one bit, so possible values can be 1 or 0. *i32* is a standard integer, that is implemented in almost languages.
float	Float (primitive)	Value with floating point with total size of 32 bits.
double	Double (primitive)	Floating point value with total size of 64 bits per value.
void	Void (primitive)	Type without size.
label	Label (primitive)	Represent code markers.
[[1..] x [type]]	Array (derived)	Array of the data with fixed data type and size. First argument of complex type is size and second is type of data included.
{ [type], [type] }	Structure (derived)	Structured complex data type.
[type] *	Pointer (derived)	Pointer represent link to another object in the memory.

Table 3.1: LLVM value types

3.2.2 Instructions

Instructions are trivial commands that will be translated into machine code that can be run on the target computer. LLVM team tries to keep them as simple as possible to make sure that instructions will be available for a very wide variety of front-ends. Generally, instructions can be divided into four categories: terminator, binary, bitwise binary, memory and other instructions. In Table 3.2 we will take a look at a few of them from every category to have a better understanding of how it works[4].

Using these instruction it is possible to perform all operations that are needed in modern programming. Theoretically it is possible to describe any software that can be written in any other language.

3.3 Just-in-Time Compilation

Besides the low-level programming language (it is lower than the well-known C language), LLVM has to offer code libraries, which gives developers the possibility to work with LLVM IR. Especially the framework allows to easily create compilers for translating IR into machine code or frond-ends for translating other languages to IR.

Compiler is a software that translates programming language, usually some sort of high-level language, into a form that is understandable for the client computer. Sometimes compilers can also include parsers (e.g. prevent syntax mistakes) and optimizators (e.g. use some techniques to speed up code execution on a particular computer model).

Another type of compilators are the ones that have the possibility to also be interpretators, the ones that will actually execute the program that was just compiled. Sometimes it is needed that compilation and interpretation processes occur simultaneously. Software that is able to compile and execute code during runtime is called a Just-in-Time (JIT) compilator[18].

IR instruction	Name	Description
Terminator instructions		
ret [type] [value]	Return	Return from function.
br [type] [condition] label [true], label [false] or br label [name]	Branch	Branch to another label on condition. Condition is either 0 or 1 to proceed.
switch [type] [value], label [default] [[type] [value] [destination], ...]	Switch	Switch between multiple values. Also available default destination.
Binary instructions		
add [type] [operand1], [operand2]	Add	Adding together two operands. Values must be with equal type.
sub [type] [operand1], [operand2]	Subtract	Subtract two operands.
mul [type], [operand1], [operand2]	Multiplication	Multiply two operands.
udiv [type] [operand1], [operand2]	Unsigned division	Divide two operands. If values was signed, they become unsigned.
Bitwise binary instructions		
lshr [type] [operand1], [operand2]	Logical shift right	Shift bit to the right.
and [type] [operand1], [operand2]	Logical and	Standard interpretation of logical AND.
or [type] [operand1], [operand2]	Logical or	Standard interpretation of logical OR.
Memory instructions		
alloca [type]	Memory allocation	Allocate memory for given type and return pointer to this memory slot.
load [type]* [pointer]	Load from memory	Load value from memory by given pointer address.
store [type] [value], [type]* [pointer]	Store to memory	Store value to the given address in memory.
Other instructions		
icmp [condition] [type] [operand1], [operand2]	Comparison	Function compare two operands. Have pre-set conditions: *eq, neq, ugt, uge, sgt, sge*, etc.
phi [type] [[value], [label]]	Phi	Special function for SSA form. Depending on condition it either stays in the same label or moves to another one.

Table 3.2: LLVM instructions

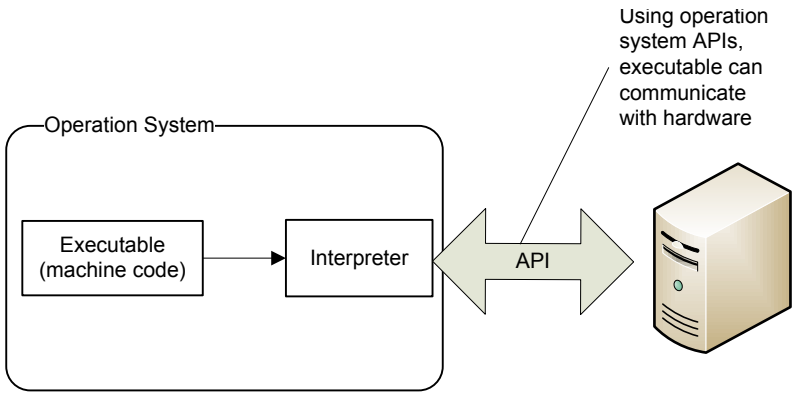

Figure 3.2: Common application execution

This kind of software was invented to solve common problems like:

- Compiled machine code is not editable and it is impossible to make any changes to the code without recompiling it for the target platform;
- Impossible to improve performance of the application with common or target specific optimizations;
- Machine code has to be compiled separately for every platform where it is used;
- Hard to collect profiling information about the running application.

JIT technique has been invented to solve problems described above.

- Code is compiled into machine code right before execution of the program, so it is possible to make changes into instructions right before any execution;
- Code can be analyzed and changed in the moment of execution, to improve application running on target machine;
- JIT can be developed in a way that input language can be totally or partly platform independent (depends on the task);
- When code is available for editing, then it is possible to include profiling information on the fly, so that developers do not need to include it while writing the application.

Figure 3.3 describes how the execution of the code is processed in JIT.

This technique is widely used by virtual machines. Using JIT allows a very flexible management of tools for code execution. It allows tweaking the performance on the fly and gathering information about the application during runtime.

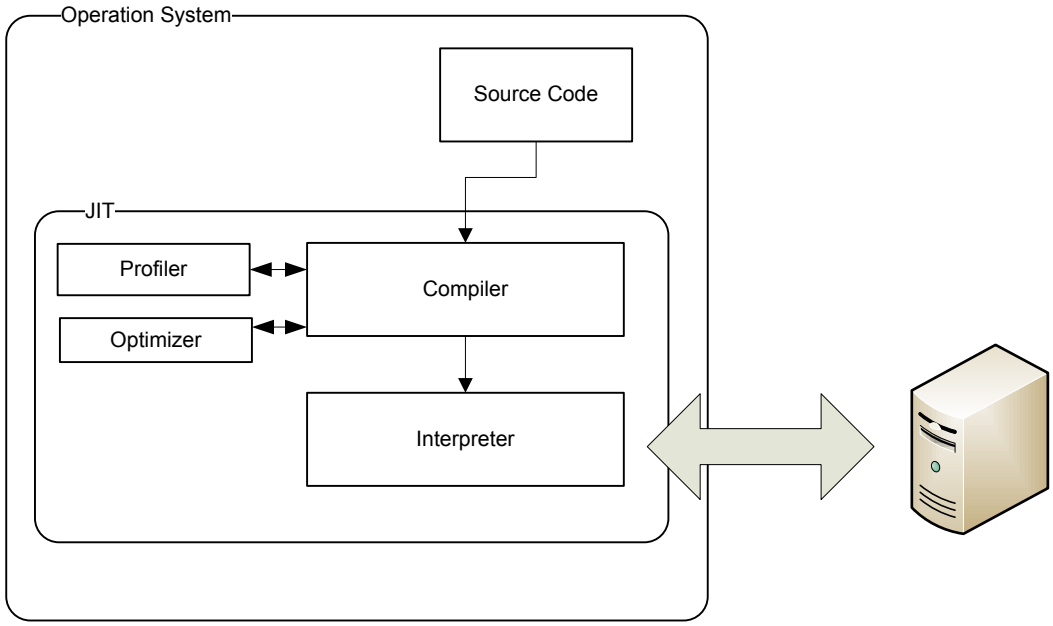

Figure 3.3: Execution in JIT

3.4 Optimizations

LLVM provides a wide range of different optimizers. The goal of optimization is to ensure that code is well formed and of course make it more reliable on runtime. There are a lot of different ways to increase code performance. Framework allows creating "Passes" that can be grouped into "Pass Managers", to allow applying of multiple passes on the code[19].

Pass is a special interface that allows iterating over the object model of instructions. When instructions of IR are loaded from textual representations into computer memory, they are represented as set of hierarchically aligned objects. Using such model it is very easy to iterate over it and retrieve the information needed for some particular pass. At the same time it is equally easy to implement some changes in the code using the framework library class provided by LLVM.

```
1 bool ModuleSecurityPass::runOnModule(llvm::Module &M)
2 {
3         for (llvm::Module::iterator I = M.begin(), E = M.end(); I != E;
              ++I)
4         {
5                 llvm::Function& F = *I;
6                 ...
```

Sneak peak into the pass source code

3.5 Security of Virtual Machine

Security is a huge problem in all computerized areas. Whenever there is something to hide, there will always be someone who will try to access it without permission. In a virtual machine security is a very daunting problem, because they execute, but the origin is not always clear. In this case it is very important that the virtual machine prevent or keep under control:

- Access to client storage devices (hard drives, solid drives, etc.);
- Size of allocated virtual memory for the application;
- Usage of the Central Processor Unit (CPU).

Every implementation of a virtual machine tries to solve the problems described above. There are many limitations as to why these tasks cannot be solved completely. The main problem is that it is not very easy to control the allocation of resources without harming the performance too much.

In the next chapter we will walk through the creation of the virtual machine and also explain how to solve the listed security problems.

Chapter 4

Implementation of the Back-end

In this chapter we will take a look at the development process of the back-end for the Friend-to-Friend computing network, which will use Low Level Virtual Machine framework. Using LLVM technologies we will learn how to build the execution engine for F2F, which is built using "Sandbox" methodology and it will become a virtual machine wit hits own secure infrastructure.

LLVM framework does not dictate how to create a virtual machine and it does not suggest any proven architecture, which is why I developed the structure personally. I was trying to keep the class structure as simple as possible and not create any overhead, so the application would not get much slower because of the C++ code that was included.

4.1 Requirements

Collecting technical requirements is one of the first logical steps in software development. Let's start with listing of the basic tasks.

The basic requirements for the new back-end are:

- Learn about the F2F modular system and propose the separation of execution engine as module;
- Add another back-end into the list of back-ends using LLVM infrastructure;
- Extend the number of supported languages for writing a job - as mentioned in Chapter A, the new handler has to definitely support C and Fortran (previous research on LLVM showed that it already has a front-end for these languages);
- Provide a security concept for the F2F execution engine - it has a variety of problems that was discussed in Section 3.5. The new virtual machine should try to solve a maximum number of the problems concerning security;
- Extend the core so that it meets modular structure and also think on future extensions;
- Modify the build scripts - most of the changes will need some extensions of SCons (Software Construction tool)[20] build scripts to include linkage of the new libraries.

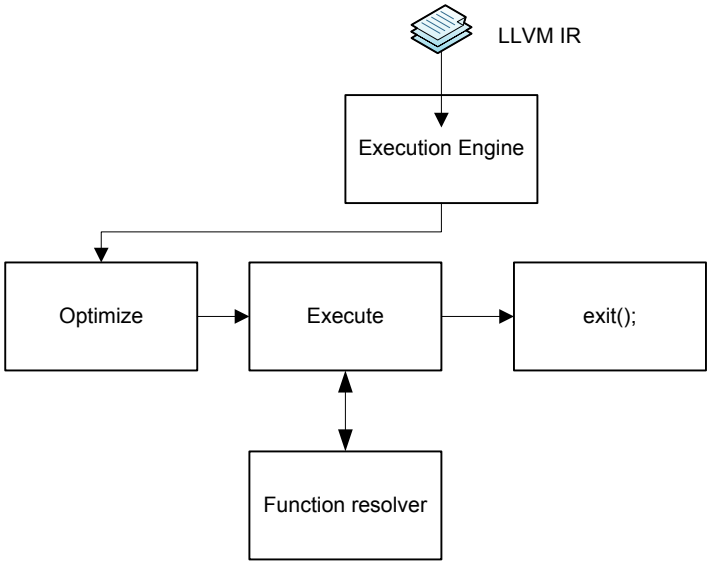

Figure 4.1: LLVM back-end

These requirements were collected during a set of conversations between different members of the Distributed Systems workgroup. Another source of ideas were seminar sessions where it was also possible to discuss some of the problems and ideas.

Graphically, the back-end should look like shown in Figure 4.1.

4.2 Used Technologies

F2F NG is written purely in C language. However, LLVM is written in C++ and its framework allows writing applications using its libraries only in C++.

LLVM is very powerful and provides a variety of tools for building a virtual machine, so there is almost no need for any external libraries. From general C++ only standard library (STD) was used - this was necessary for making it possible to log the output and for storing information in a comfortable form.

For the new handler some of the LLVM libraries were used:

- System - library is a wrapper for some particular operation system interfaces, mostly containing classes that allow communication with operating system;
- Assembly - library for working with the LLVM IR, as we will work a lot with intermediate language, we are in need of this library;
- Analysis - library consists of different analysis passes,needed for performing optimization. The set of optimizations that are applied on every job before execution are described in subsection4.4.2;
- Support - library with various helper tools for easier developing, very useful tools like buffering content into memory can help in many cases when data storage needed;

- ExecutionEngine - library that supports all kinds of code execution, for us only one part is interesting - the particular set of classes that is responsible for real time compilation. Library and compiler both have the same names in JIT.

To find out which exact libraries are needed and which classes have to be used to create a virtual machine, the documentation provided by the LLVM community was read. It is also possible to find some examples that are very useful to follow[21, 22].

4.3 Class Diagram and Source Code Structure

To get a better overview of the application source code, at first the steps described in Appendix B should be followed. However, this information about structure might be valid only at the time of writing this work, because it can be refactored later and might need to be specified from up-to-date documentation.

The first level of directories, in cloned code from repository, represent module names.

```
.
..
adapters
core
exec
old
```

Actual handler code is situated under the *core* module. It has a subdirectory with sources *src/libf2fcore*, where all core header and implementation files can be found. The job handler of F2F has its own subdirectory *handlers* and the particular handler source code for LLVM jobs is under *llvm* directory.

```
f2fjit.cc              // llvm handler source
f2fjit.h               // llvm handler header
f2fjit_wrapper.cc      // wrapper for execution of C++ class in C
f2fjit_wrapper.h       // wrapper header file
f2fllvmjobhandler.cc   // handler register in the core
f2fllvmjobhandler.h     // handler register header
f2fsecurefunctions.cc  // implementation of secure functions
f2fsecurefunctions.h   // secure functions header
f2fsecuritypass.cc     // implementation of experemental security pass
   for "black list" method
f2fsecuritypass.h      // security pass "black list" header
```

Code has inline comments, mostly in implementation parts, for easier lookup.

Figure 4.2 represents a class diagram with comments about its attributes and methods.

Besides that, several single static attributes and methods were used. For additional explanations of the code, please read the inline comments of the implementation files (*f2fjit.cc*). For LLVM class references, the official documentation of the API can be found on their homepage[21].

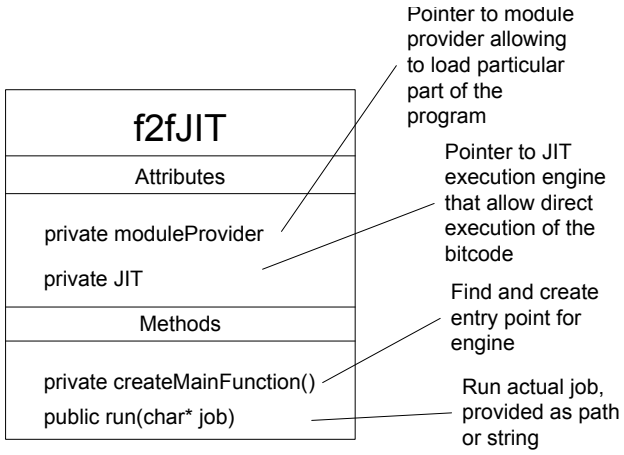

Figure 4.2: LLVM back-end class diagram

4.4 Functionality

During the virtual machine's life cycle several important steps can be extracted - optimization of the code, execution of the code and security mechanisms.

4.4.1 Execution

To execute a job, which is actually just a bunch of code, the core finds the appropriate handler for the job, then creates an instance of the handler and passes the job content to it. A job executed in the LLVM back-end will pass the following steps:

1. Instance of the *f2fJIT* class is created;
2. Object executes *run* method that has one input parameter with possible values:
   (a) Job code represented as array of characters;
   (b) Path to the file with the code of the job.

   Several class instances from LLVM namespace are created here to prepare JIT virtual machine for execution. First we create the *llvm::Module* instance, which will become a provider of the execution.
3. Read LLVM IR into memory;

   LLVM framework provides different helper methods for common tasks. Let us use one of such methods from the *Support* library. To read a job binary into memory we use the *llvm::MemoryBuffer::getFile* static method.
4. JIT module lookup for entry method in job;

   When all objects have been read into memory, then it is possible to look up the entry method function (Subsection 4.4.1.1). If the method is not found, execution is aborted.
5. Validate that the code is correct and ready to run;

6. Optimizing the code;

   For validation and optimization llvm::PassManager is created. It includes all of the optimizations described in subsection 4.4.2.

7. Entry method is executed (which also executes the job);

   Execution engine provides methods to call functions by providing a reference to it. As we have already resolved function in previous steps, it needs to be passed to the *llvm::ExecutionEngine::runFunction*.

8. After entry method execution is done, the module object is destructed;

9. Job handling in the handler is finished.

4.4.1.1 Job Entry Method

Every software needs to have a starting point where its execution starts. In the context of F2F we call it the entry method - this starts the execution of the code in the job. Going together Similarly to other programming methods (like in C, C++ or Java languages), the LLVM back-end requires a job to have a method called *main*. This is a very common practise and is widely used in various programming languages where execution of the program starts with that method.

4.4.2 Optimization

LLVM has a lot of optimization passes included in its framework. Additionally, there are tools for creating one's own passes if there is a specific need for that.

Some selected optimization techniques were used in this project:

- VerifierPass - ensures that the LLVM IR is correct and well-formed (no syntax mistakes), also tries to find logical mistakes;
- LowerSetJmpPass - lower set jump - lowers *setjmp* and *longjmp* to use the LLVM invoke and unwind instructions as necessary;
- RaiseAllocationPass - transforms the usage of the *malloc* and *free* function calls into appropriate instructions of IR;
- CFGSimplificationPass - simplifying the Context-free grammar (CFG) - perform dead code elimination and basic block merging;
- PromoteMemoryToRegisterPass - promotes memory references to be register references;
- GlobalOptimizerPass - performs a search to find global variables that are not used and eliminates them;
- GlobalDCEPass - eliminates dead code globally;
- FunctionInliningPass - turns inline functions into global functions and replaces them with function calls.

The main reason of optimization is reduce the size of the code for interpretation and increase execution speed. Also, it is possible to prevent threats or mistakes that were made by developers. [23]

4.4.3 Security

In fact this task should meet the needs that were described in Section 3.5. LLVM framework does not provide any ready-made security solutions for virtual machines. In the scope of this work some of the security elements will be implemented for the project to ensure that harmful code cannot be run on a client machine.

The idea of the security layer is that we want to keep under control the code that is run inside the virtual machine. The most important feature is that it should not be possible to access computer interfaces and storages (another thing is if the user wants to allow controlled access to these things). The second issue is memory allocation - this can cause issues when the code tries to use all of the operation system's virtual memory and which slows the processes down and can even cause all working processes to crash.

4.4.3.1 Control Executed Functions

At first there was a requirement that job developers should not be able to run functions that are potentially unsafe for execution on computers. Under this category belong all functions that perform:

- memory allocations - from C these functions are: *malloc*, *alloc* and others which can lead to buffer overflow;
- data reading from storage device - there are a lot of different functions for manipulating the file system, like *fopen*, *fputs*, *remove*, *rename* and others.

There was actually two ways to implement that functionality:

1. The "black list" method;
2. The "white list" method.

Black List Method Black list means that there is a list of functions that cannot be executed and execution engine will prevent them from running. This method was implemented as the first version of secure function controller. A special optimization pass was created to provide security. It was iterating all over the module and resolved recursively all function names and tried to look up for them in the black list. In case a blacklisted function was found, it was replaced with a *void* pointer. Unfortunately, this method was not highly reliable because it was almost impossible to keep track of all functions that needed to be on the list.

White List Method LLVM JIT library provides a handy way to switch off the resolving of any external functions and additionally sets the delegate function that will be responsible for every resolving. Getting string as parameter with function name, the main task of the resolver delegate is to return a function pointer that will then be executed, or not to resolve this symbol. Now the virtual machine does not allow any function execution besides the ones that are provided by the environment. The so-called "white list" ensures that only the provided functions will be resolved. If the called function is not on the list,

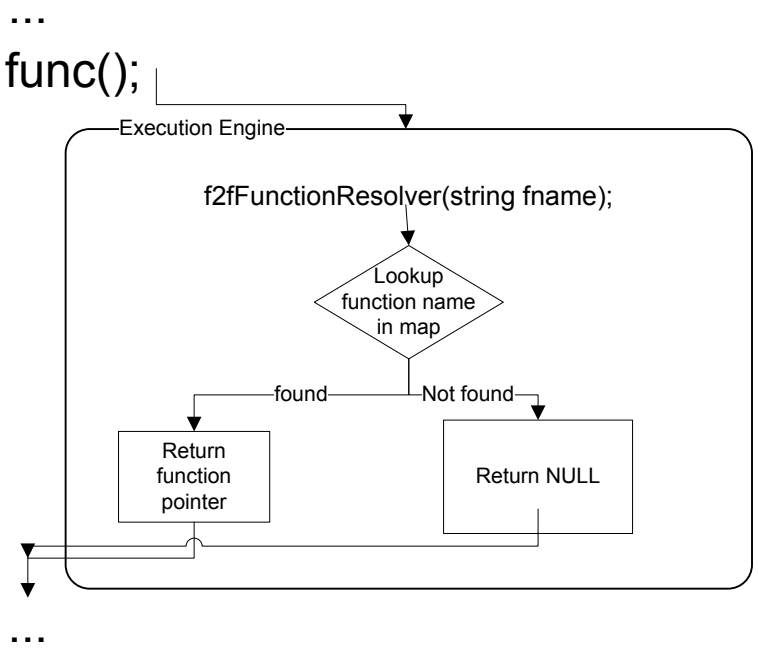

Figure 4.3: Function resolving in job

Function prototype	Description
void f2fSleep(int mseconds)	Sleep given amount of seconds.
int f2fStrLen(char* str)	Return length of the given char array.
float f2fSqrt(float number)	Compute square root.
void f2fWriteFile(char* fname, char* content)	Write some content on disk.

Table 4.1: White list functions

then it will be replaced with an empty function and can lead to code execution failure because of other dependencies inside the code.

Job handler provides some functions for job writers to use in their programs, to allow communication with other peers in the network. Some more functions will be provided for developers to make job writing more reliable and comfortable.

The scope of this work had to include the creation of a mechanism that would allow white listing in the application. Currently only some basic functions were provided, just to give an idea of possible function types that could be implemented. However, this topic needs a much deeper research in this area to find out exactly what kind of functions the developers need in the area of scientific computing. This kind of research is left for future development and can be done in the scope of another thesis.

These functions are basically wrappers of the original library functions that are well-known and does not need an introduction. The main idea of creating such wrappers is that now it is possible to also keep the input under control. Checking input against different threats and not allowing performing of any suspicious operations. E.g. *f2fSleep* can have an extra check for maximum length, which is not implemented in the original function, which ensures that developers cannot set the application to sleep for too long. The *f2fWriteFile* should have a similar behaviour - by taking only the file name as parameter,

it is not possible to specify a full path to some particular directory, and writing job files is never allowed anywhere but in the allowed locations.

4.4.3.2 Control of Virtual Memory Allocation

At the beginning of the development this seems to be an easy task to complete, because JVM virtual machine behaviour was expected. Unforeseeably there were a lot of differences, that came out only during the development phase. JVM does not allow code t be directly allocated from the memory of the computer, but forces to use the machine's own general heap, which can be limited by machine settings. However, LLVM does not provide ready-made solutions for such management, and at the time of writing this thesis no one had submitted any examples of its implementation.

Research has been done to include memory management into a F2F virtual machine, but due to the lack of experience in this area, the implementation was not finished and was left for the future releases.

In the previous subsection 3.2.2it was mentioned that LLVM IR has commands that instruct the computer on how to execute the program. Some of them control memory allocation - *alloca*, *store*. As all these instructions will be represented in the object model of the LLVM IR, it is theoretically possible to write a pass for executed code and measure how much memory will be executed for a particular job even before the execution of the code. There is also another problem: memory can be allocated dynamically during actual execution. The solution for this is to provide special allocation functions. Such functions in C are *malloc*, *realloc*, *calloc.*As we have only while listed functions allowed, it is possible to provide special allocation functions for F2F that will together with standard functionality also count how much memory was allocated and deallocated. The counter will track how much memory the application used dynamically and stop execution if the limit is reached.

4.5 Implementation Environment

The project was developed in Eclipse IDE "Ganymede" version 3.4.0 on Mac OS X (10.5.8). Additionally, useful plug-ins were installed on top of Eclipse - C++ development tools and PyDev development tools. The *scons* tool was used as a building environment. Project development was committed to public revision control system *GiT*.

4.6 Building Environment

During implementation testing there is an actual need for running the code that was produced. Simple code compilation and execution is not always a very trivial task, because applications have a lot of different dependencies and have to include a lot of libraries at linking stage. In such situation it is very helpful to use construction tools for compiling, linking, and execution. The most well-known tools are make, automake, autoconf, etc. However, the little-known tools are sometimes better and more user-friendly. One such tool is *SCons*[20], which is written in Python and allows constructing very complicated software. The description of the building steps has to be written down in Python language as a file that has to

specified in the *SCons* before build. A common practise for the F2F team is to hold these instructions in the *sconsbuild.py* file, which usually located in the *build* or *src* folder of the project.

When support for LLVM was added to the *core,* some changes had to be made in the building environment, because LLVM required a lot of header files and libraries that could not be resolved automatically. Here I would like to mention one very nice feature of about *SCons* - it is able to detect source code origination automatically and choose the proper compilator automatically from the environment of the operation system. This means adding C++ files and headers into the project did not require almost any changes in build scripts, because it can automatically detect what kind of source code is provided and build it with the correct compiler. After the changes were completed and the needed headers were included into the script, it ran well on the development environment. The problems began when I wanted to try out virtual machine on Linux platform and it would not run anymore.

Compilation of all libraries went well, but during execution the application failed with "Undefined reference to symbol" by *ld* tool (dynamic linker/loader). Usually such an error means that the included directories path that was given during compilation does not contain the needed libraries. Trying to understand this bug took a lot of time, because there was an environment where exactly the same code had worked fine. In the end, the origin of the problem was found. The position of included libraries is important to LLVM and if you do not include them in the right order, the dynamic linker will not be able to resolve all dependencies and find the proper symbols. This situation seems to be very uncommon, as there is no logical explanation why the order of linked libraries should matter so much that the execution fails. Nevertheless looking into the problem from the right perspective lead to a nice tool provided by the LLVM team - *llvm-config*. This tool is made specially to resolve problems with linking and compiling. It has various keys that can return parameters for the compiler (linking flags, compiler flags, libraries). By doing something like

```
llvm-config --libs engine bcreader scalaropts
```

it would actually return a list of libraries to standard output that was needed for these modules to be in the proper order.

SCons does not support reading additions from standard output of the command, and requires libraries to be as array in the script. Build script was extended with parsing results of the *llvm-config* tool into an array.

4.7 Supported Platforms

Theoretically, F2F with the new back-end should support three platforms:

- Linux;
- Mac;
- Windows (using MinGW).

Practically, the code was tested on two platforms with specific versions:

- Linux Ubuntu (9.x) and Fedora (10) distributions;
- Mac OS X (10.x).

A full network test was done only on Linux. There was no native support for Pidgin on Mac, so it was just tested with function executions on client without testing it inside the network.

4.8 Visibility of Refactoring and Improvements

Virtual machine and back-end in general are in need of a good configuration manager that would let the user configure different options of them. This task can be solved either via a configuration file or even a special graphical interface for setup.

Virtual memory allocation and CPU speed management are topics that need some special dedication. These topics are difficult to solve and might be implemented in future.

Chapter 5

Working with LLVM Back-end in F2F

The previous chapters were very theoretical and described workflows with short snippets of code which have not given any understanding of the product. The following sections will approach the topic in a more "how-to" manner , in order to give a better understanding of what really happens when we have a real network of friends who would like to share some computational power between each other.

In Appendix A a virtual disk image can be found. The disk contains a preinstalled Linux operation system with all the required tools and code preinstalled to start using F2F with the new back-end.

5.1 Exploring Environment

Operation system credentials are:

- Username: llvm
- Password: llvm
- Architecture: x86

Learn more about running a virtual hard drive with VirtualBox on the software's official homepage[24].

The virtual hard drive consists of:

1. Linux Ubuntu 9.10 (Karmic Koala) Desktop edition;
2. C/C++ Building Essentials;
3. GiT tools;
4. LLVM 2.5 libraries and tools;
5. Python 2.6;
6. Pidgin Internet Messenger 2.6.2;

7. SCons tool;
8. SWIG tool;
9. OpenFire XMPP server.

Additionally installed packages list: build-essential, git, git-core, swig, scons, python, python-dev.

Karmic is Ubuntu's latest version and has its own package sources with the newest software. At the time of writing this thesis LLVM version 2.6 was also available, but the application code needed to be built against version 2.5. The packages for the 2.5 version are available in the software source channel of the previous Ubuntu version - Jaunty. There is a need of manual installation of older debs (Debian software package) downloaded from Jaunty sources: llvm, llvm-dev, llvm-gcc[25].

To reduce the amount of external tools that might be used, a local Jabber (OpenFire) server was also installed and some users were prepared on the server.

The home directory for the thesis-related sources is "/home/llvm/Documents/f2f/". In this folder several project folders can be found:

- adapters - folder with available communication adapters;
- core - F2F core library;
- exec - execution engines;
- old - files related to first version of F2F written in Java.

All source files were cloned from F2F project public GiT repository[26].

5.2 Sample Job

Let's walk through the steps that are needed to write a job that can be executed in the LLVM back-end. Sample job was written in C language to test execution in a virtual machine. The test job implemented the "ring topology" idea where every peer resends the message to next one in the group. For better understanding, refer to Figure 5.1 which shows the process graphically.

When the program is ready, the next logical step is to try to compile it into a proper form for usage in a virtual machine. For sending it to other peers, the binary LLVM IR is needed. For that, the LLVM-GCC compiler will be used. It is provided by the creators of the LLVM framework and is kind of a showcase of its functionality. LLVM-GCC provides the same functionality as a standard GCC compiler,the difference being that is uses the LLVM framework to work with the code. One very important feature of the LLVM-GCC is that it is also a front-end for C and C++ allowing transformation of source code into LLVM IR. Option "-emit-llvm" of the compiler preserve creation of the machine code from source and leave it as LLVM IR in binary format.

To compile the provided job code into LLVM IR, we will need to use the following command line:

```
1  int nextPeer(int count, int peer){
2          return (peer+1) % count;
3  }
4
5  int main() {
6
7          F2FJob* job = f2fGetJob();
8
9          if (job->content)
10                 f2fJobLog(job, job->content);
11
12         int myid = f2fJobGetMyself(job);
13         int master = f2fJobGetInitiator(job);
14         int count = f2fJobGetPeerCount(job);
15
16         if (myid == master){
17                 //@TODO: replace delay
18                 // make sure job started on remote peers
19                 sleep(20);
20
21                 // Init send
22                 f2fJobLog(job, "Send␣Init␣message␣...");
23                 char* msg = "Test␣Message\0";
24                 f2fJobSendMessage(job,nextPeer(count,myid), msg, strlen(
                       msg));
25         }
26
27         //Circular send 10 times
28         int idx;
29         for(idx=0; idx<10; idx++){
30                 f2fJobLog(job, "Receiving␣...");
31                 char buffer[F2F_MAX_JOB_MESSAGE_SIZE];
32                 int size =0;
33                 int sender = -1;
34                 size = f2fJobWaitForMessage(job,buffer,
                       F2F_MAX_JOB_MESSAGE_SIZE,&sender);
35                 buffer[size] = 0;
36
37                 f2fJobLog(job, "Received␣msg");
38                 f2fJobLog(job, buffer);
39
40                 f2fJobLog(job, "Send␣for␣next␣...");
41                 f2fJobSendMessage(job,nextPeer(count,myid), buffer, size
                       );
42         }
43
44         f2fJobLog(job, "Job␣End");
45         return 0;
46
47 }
```

Listing 5.1: Test job source code

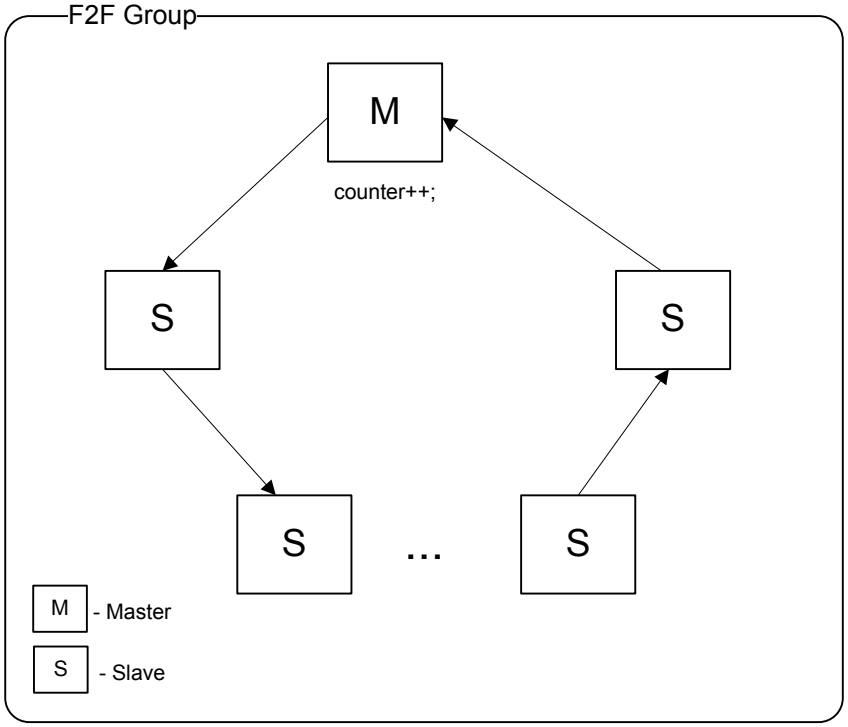

Figure 5.1: Ring topology test job workflow

```
llvm-gcc -emit-llvm -c test64-mesh.c -I/home/llvm/Documents/f2f/core/src
   /libf2fcore/ -I/home/llvm/Documents/f2f/core/src/libf2fcore/
   jobhandlers/
```

Besides providing the file with the source code we also have to provide two directories with header files, so that the compiler will be aware of the special functions that were used in the job.

The execution command will produce the file "test64-mesh.o", which is already in required form in the required binary format that is understandable for a virtual machine. This is a file whose content can be sent to other peers in the network for execution.

If there is a need to verify the created LLVM IR, there is a special tool provided for that - *llvm-dis*. Taking a stream as parameter it will disassemble binary code into a human readable form to standard output.

```
llvm-dis < test64-mesh.o
```

5.3 Sending Job With Pidgin Plug-in

F2F provides a comfortable interface for managing groups of peers and making job sending very easy. Working as a plugin for Pidgin Instant Messenger, it extends software functionality with additional features, allowing to create special groups of F2F users and enable content delivery between them.

The sample environment provides a preconfigured Pidgin software with all users and the needed group presets. You will find program launchers on the Ubuntu desktop - "Pidgin F2F001" and "Pidgin F2F002".

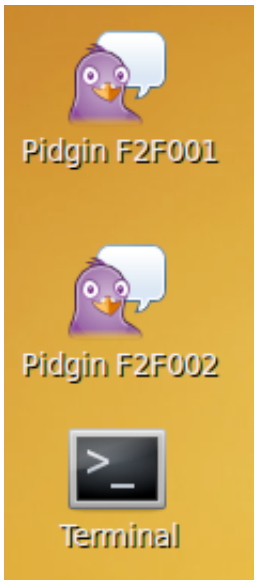

Figure 5.2: Pidgin launchers

Two buddies will communicate between each other using the locally installed "OpenFire" Jabber server.

Figure 5.3 represents how it should look like for two contact lists with preset users. "f2f001" have single contact "f2f002" and vice versa.

We have to try to create a test group and send out the job. The easiest way is to use the already existing chat group "test", which can be seen on Figure 5.3. Double-clicking on test group will open the chat window and you will need to add another peer to the group by dragging it into the chat window. Now the group should be created and initialized. The new state can be seen on Figure 5.3.

When everything is prepared, the LLVM job can be sent to the peers. The following steps need to be done:

1. In the chat window context menu, click "Conversation" tab;
2. Follow the submenu "More";
3. Find menu "Submit Job";
4. Choose the file you want to submit;
5. Follow the trace log in the debug window.

Our simple job will write logs into the debug output window where job execution can be followed.

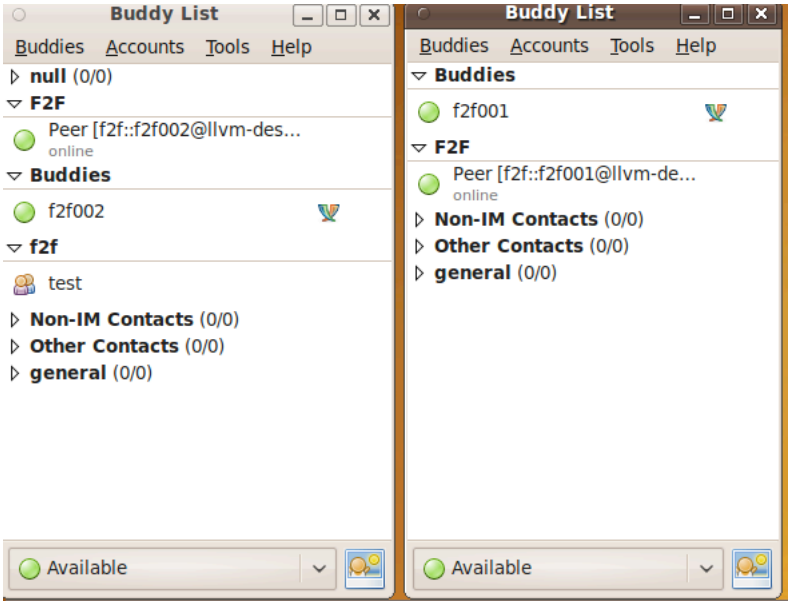

Figure 5.3: Pidgin software ready state

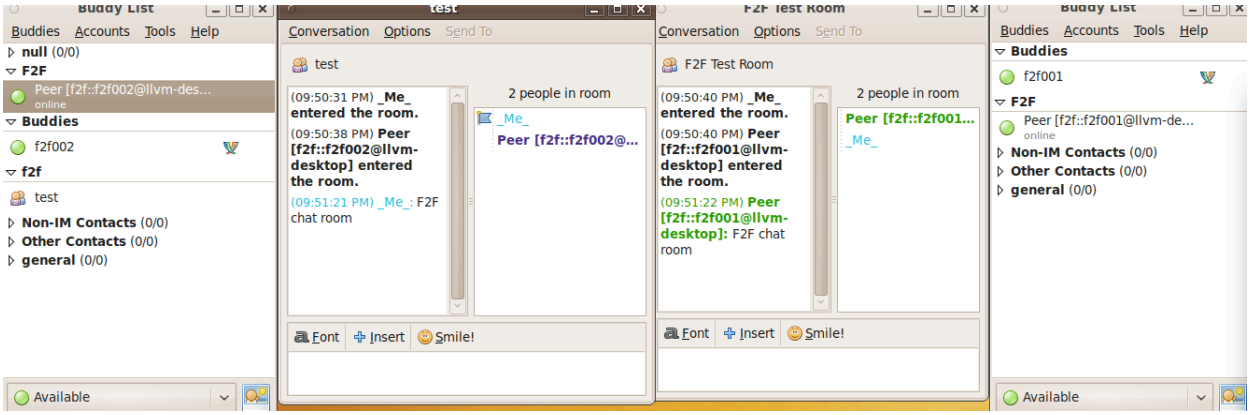

Figure 5.4: Chat group created

Chapter 6

Conclusion

The Main practical goal of this thesis was to create new a back-end for a Friend-to-Friend computing network. It had to implement the virtual machine that was built with the Low Level Virtual Machine framework, and support the execution of LLVM Intermediate Representation, a low level language designed for easiest conversion to machine code.

LLVM is a framework and a set of tools for creating compilers, optimizers, analyzers and interpreters. One of the main goals of this research was to create Just-in-Time compilers and extend F2F with an execution engine based on this technology.

The total time spent on this work can be divided into three logical parts:

- Research - I had to learn about the LLVM framework, platform independent code, virtual machines, virtualization techniques, compilers, interpreters and many more related things;
- Learn F2F - had to dive into the world of distributed networks and computing, find out the workflow and problems in this area, understand the needs of the project and search for the best solution;
- Development - F2F is still under heavy development and I was involved in that process too. The actual system had no modular system in the beginning, but during working on the system it became available what we call now "job handler".

Additionally, I learned a lot about how an application works on the low level, how library linkage works, compilation on different platforms, static and dynamic libraries compilation, code optimizations and low level networking.

I truly believe that the LLVM framework is a great thing and adding support of it was really worth it. There are a lot of possibilities to improve different aspects of the virtual machine, like security, usability, and performance. All of these points could be studied in greater depth and be topics for individual research papers in the future. There is chance that in the end F2F will only support the LLVM execution engine because of its versatility.

I see F2F being used in various places in all kind of areas. In a few years I see it not only as an learning accessory, but a stand-alone product, that can be used in huge distributed networks. I will continue supporting this project and help its contributors to extend its popularity and community.

Resümee (Eesti keeles)

Magistritöö (20 AP)

"Low Level Virtual Machine" virtuaal masin F2F arvutusvõrgu jaoks

Oleg Knut

Sisukokkuvõte

Akronüümi F2F taga on inglisekeelsed sõnad *Friend to Friend* (sõbralt sõbrale). See on lihtne kasutajalt-kasutajale võrgustik, mis lubab kasutajatele (sõpradele) vahetada arvuti jõudluse kasutamist. F2F on kirjutatud Tartu Ülikoolis ja seda kasutatakse õppevahendina. Põhiline tema kasutusala on näidata üliõpilastele kuidas saab luua hajussüsteeme ja luua andme edastus kanaled kasutajate vahel. Et kasutajad saaksid hakkata kasutama võrgustiku eeliseid peavad nad selleks kirjutama koodi programmi nn "töö" ja toimetama seda läbivõrgu kõikidele gruppi kasutajatele ehk sõpradele. "Töö" saab olema käivitatud kõikidel arvutitel ja vastavalt etterkirjutatud operatsioonidele genereerib tulemuse, mida saadab töö alustajale.

Praegune F2F võrgustik toetab töö kirjutamist vaid ühes programmeerimis keeles Python[2]. See keel on dünamiline skriptimise keel ja annab kergesti kirjutada erinevaid applikatsioone sõprade võrgu jaoks, kuid on selge, et ainult ühest keelest ei piisa ja eriti õpikeskkonna on vaja neid rohkem.

Magistritöö idee tuleneb sellest, et lubada kirjutada töid F2F võrgu jaoks võimalikult mitmel keelel, mis lubaks neil ise valida, mis keeles nad kirjutada tahavad ja et nad tingimata ei peaks juurde õppima Python keele. Uurimiste käigus, mis olid tehtud Tartu Ülikooli Hajussüsteemide rühmas olid valitud raamistik LLVM, mis lubas täiendada olemas oleva klient rakenduse nii, et see hakkaks toetama rohkem keeli, milles saaks töid kirjutada.

Tänu LLVM tehnoloogiale on võimalik luua nn "vahepealne esindus". See on programmerimis keel, mis on instruktsioonide jada ja on loodud selleks, et maksimaalselt mugavalt luua sellest masina koodi. Selle keele tähtis omadus, et sama raamistik pakub vahendeid, et luua programme, mis oskavad genereerida mõnes teisest programmeerimis keelest "vahepealset teisendust". Juba praegu on saadaval kompilaatorid,

mis oskavad genereerida “vahepealset teisendust” populaarsetest keeltes nagu C, C++, Python, Fortran, Java.

Selle töö põhieesmärgiks oli uurida kuidas on võimalik teha virtuaal masin F2F töö koodi käivitamiseks. Uurimustest saadud informatsiooni kasutada praktikal ja täiendada F2F, nii et see hakkaks toetama LLVM “vahepealset esinduse” käivitamist, mis omakorda annab võimaluse kirjutada F2F tööd väga paljudes keeltes.

References

[1] Friend to Friend computing. http://f2f.ulno.net/ (Last accessed: January, 2010).

[2] Python Programming Language Official Website. http://www.python.org/ (Last accessed: January, 2010).

[3] LLVM Homepage. http://www.llvm.org/ (Last accessed: January, 2010).

[4] LLVM Assembly Language Reference. http://llvm.org/docs/LangRef.html (Last accessed: January, 2010).

[5] Virtual Machine. http://en.wikipedia.org/wiki/Virtual_machine (Last accessed: January, 2010).

[6] Peer to Peer. http://en.wikipedia.org/wiki/Peer-to-peer (Last accessed: January, 2010).

[7] Internet Protocol address. http://en.wikipedia.org/wiki/IP_address (Last accessed: January, 2010).

[8] Network Address Translation. http://en.wikipedia.org/wiki/Network_address_translation (Last accessed: January, 2010).

[9] JPunch. http://ulno.net/projects/jpunch (Last accessed: January, 2010).

[10] Internet Protocol Suite. http://en.wikipedia.org/wiki/Internet_Protocol_Suite (Last accessed: January, 2010).

[11] User Datagram Protocol. http://en.wikipedia.org/wiki/User_Datagram_Protocol (Last accessed: January, 2010).

[12] Message Passing Interface. http://en.wikipedia.org/wiki/Message_Passing_Interface (Last accessed: January, 2010).

[13] Java Virtual Machine. http://en.wikipedia.org/wiki/Java_Virtual_Machine (Last accessed: January, 2010).

[14] Common Language Runtime. http://en.wikipedia.org/wiki/Common_Language_Runtime (Last accessed: January, 2010).

[15] Mono Project. http://www.mono-project.com/Main_Page (Last accessed: January, 2010).

[16] Illinois Open Source License. http://www.opensource.org/licenses/UoI-NCSA.php (Last accessed: January, 2010).

[17] Static Single Assignment form. http://en.wikipedia.org/wiki/Static_single_assignment_form (Last accessed: January, 2010).

[18] Just in Time compilation. http://en.wikipedia.org/wiki/Just-in-time_compilation (Last accessed: January, 2010).

[19] Writing an LLVM Pass. http://llvm.org/docs/WritingAnLLVMPass.html (Last accessed: January, 2010).

[20] SCons: A software construction tool. http://www.scons.org (Last accessed: January, 2010).

[21] LLVM API Documentation. http://llvm.org/doxygen/ (Last accessed: January, 2010).

[22] LLVM Kaleidoscope JIT tutorial. http://llvm.org/docs/tutorial/LangImpl4.html (Last accessed: January, 2010).

[23] LLVM Analysis and Transformation Passes. http://llvm.org/docs/Passes.html (Last accessed: January, 2010).

[24] Inc. VirtualBox by Sun Microsystems. http://www.virtualbox.org/ (Last accessed: January, 2010).

[25] Package llvm in Jaunty. http://packages.ubuntu.com/jaunty/llvm (Last accessed: January, 2010).

[26] F2F Public Repository. http://git.f2f.ulno.net (Last accessed: January, 2010).

Appendix A

Virtual Drive With Preinstalled Environment

Here you find on attached DVD with virtual drive for VirtualBox virtualization software with preinstalled Ubuntu Linux on it.

URL for download: http://f2f.ulno.net

Please follow next steps:

1. Unarchive or use version from DVD the .vdi file;
2. Install and open VirtualBox software (can be downloaded at http://virtualbox.org);
3. Create New Virtual Machine;
4. Set name and choose Linux and version Ubuntu from operation systems dropdown lists;
5. Setup values for RAM;
6. Specify location of .vdi file;
7. You are done, now just run virtual machine from the list.

Appendix B

Public Source Code Access

In order to have access to all the source code of F2F you will need to have some tools preinstalled on your computer. F2F project is using GiT repository for saving revision histories. You will need install either GiT tools or use some other client. If you use Linux of Mac OS, then GiT can be found from their software repositories (available as package named *git-core*). More info about GiT please read at their home page - http://git-scm.com/.

To clone F2F source code with standard command line tools:

```
git clone http://git.f2f.ulno.net f2f
```

This command will create a local copy of repository under "f2f" directory.