UNIVERSITY OF TARTU
Faculty of Mathematics and Computer Science
Institute of Computer Science
Speciality of Computer Science

Keio Kraaner

Friend-to-Friend Computing

Master Thesis (20 CP)

Supervisors: Ulrich Norbisrath
Eero Vainikko

Author: ……………………………………….. "……" May 2008
Supervisor: ………………………………..... "……" May 2008

Allow to Defense:
Professor .... …………………… "……" May 2008

TARTU 2008

Contents

Chapter 1

Introduction

This master thesis is the result of my work during one year in the Distributed Systems group of Eero Vainikko at the computer science department of the University of Tartu. The main ideas of this work were first published in [24] that I wrote together with Ulrich Norbisrath, Eero Vainikko, and Oleg Batrašev (these are the persons to whom I refer as *we* hereafter). The most important differences between this paper and my thesis are the sections 2.3, 2.4, 3.2.6, 3.2.7, 3.2.8, 5.1, 5.3 and the chapter 4.

Current trends in scientific computing and computational science show the increase of the needs for more computational power. Some examples are large-scale simulations in environmental and engineering science, pharmacy, and chemistry. At the same time, computer architectures move more and more toward distributed and multi-core processor architectures. Therefore, *parallel* and *distributed computing* becomes more important. One way to facilitate increased computational power is super computing or cluster computing.

Another way growing in popularity is *Grid computing*. Grid computing means that several computers are used as one. It networks single computers, servers, supercomputers, clusters and special devices into a global resource [13]. This approach enables to build a virtual supercomputer at a fraction of the cost of traditional supercomputers.

The working principle of Grid computing is that a computational problem (called *job*) that needs to be solved is split into small parts (called *tasks*), the processing of each part is done on an individual computer, the results are collected and combined into a solution of the problem. If the processing of the tasks does not require any communication between them the problem is called *embarrassingly parallel*.

Various frameworks exist to support Grid computing, like the Globus Toolkit [14], UNICORE [18], or the Condor Project [28]. Grids that are based on these frameworks network different computing centers, companies, and universities. These Grids work well but have a major drawback - they are very heavyweight and complex. They need huge administrative capacities, just to keep the Grid infrastructure running. If only a small job should be submitted, this administrative overhead is not justified. The initial metaphor for the computational Grid being as easy as a power grid is still unfulfilled [30].

For several years now, most of the world's computing power is no longer primarily concentrated in computing centers, companies, and universities. It is distributed in hundreds of millions of PCs and other computer equipment belonging to the ordinary people. The number of Internet-connected PCs is growing rapidly, and is expected to reach 1 billion in 2008. Together, these PCs could provide several PetaFLOPs of computing power. This, and the fact that most of today's PCs are heavily underutilized (typically less than 10% of their processor power is used), and the ubiquity of the Internet has given birth to an idea to connect usual PCs over the Internet into a global computing system. Such a system is called *desktop Grid*. Previously mentioned frameworks are too complex to be used in desktop Grids: people without information technology (IT) background are not able to install these systems on their PCs.

A step to making networked computing power more accessible to the public is *public-resource computing* (or *volunteer computing*) as proposed in the BOINC [11] approach. It networks home-PCs to a centralized Grid. Famous examples, using this, are SETI@home [12] and Einstein@home [5]. This approach is targeted at small research groups trying to solve problems which take enormous computational effort in terms of computational time, usually needing years to compute. BOINC makes it easy to donate computational power to some project, but creation and management of such a project still needs quite big administration effort.

To facilitate a public usage of Grids even more, we extend Grid computing by using techniques from *peer-to-peer* (P2P) *computing* and *instant messaging* (IM). IM is used by the vast majority of computer users. We assume that using IM techniques for the administrative part of setting up a desktop Grid will open up Grid capabilities to the public. Imagine if you could use the power of your MSN friends' PCs to run an application much faster! If the simplicity of forming social networks with mutual trust (to some level) in current IM systems would be extended with the possibility to easily unite the computing power of the participants within one social network, lots of people would have access to a desktop Grid. Analogous to peer-to-peer computing, the here applied paradigm is called *friend-to-friend* (*F2F*) *computing* as a contraction of Grid and P2P computing, and friends relating to the spontaneously formed social networks. As a F2F Grid is formed with friends in the IM system, the participants share some kind of trust and thereby authentication and authorization issues can play a minor role. The F2F computing paradigm will enable single researchers, very small research groups, and small companies to combine their computational power with their friends or colleagues and to solve their computationally intensive problems much faster than today. This paradigm will probably have an impact on scalability of current Grid projects and will yield results for simplified Grid application development. New F2F Grid applications, that become popular within masses, will widen the circle of Grid users by ordinary people without any IT background.

My research was driven by implementing and evaluating the following example scenarios:

1. Simulation Boost: A chemistry (further planned scenarios apply to fields

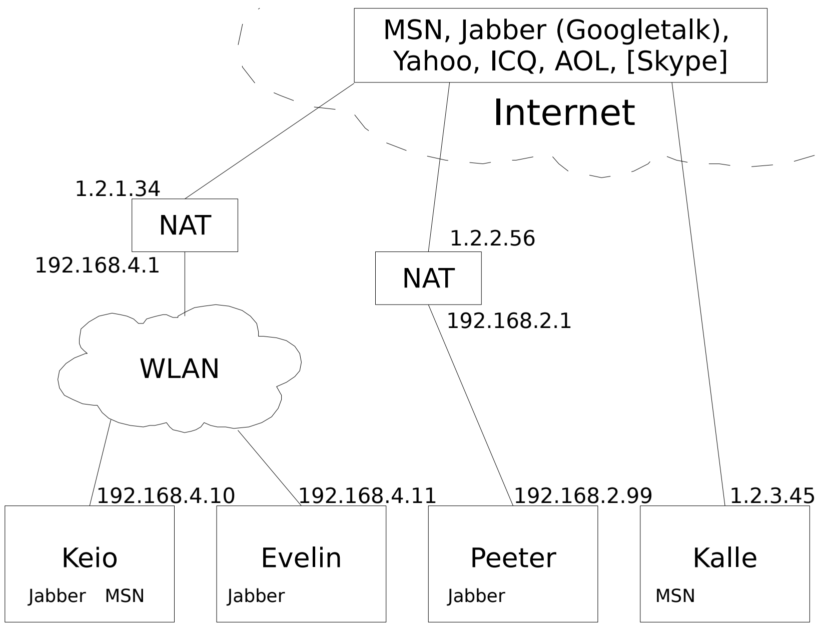


Figure 1.1: Overview of a typical layout of a small social network

of mathematics, bio-informatics, physics, or biology) student needs to run a simulation of a reaction. The student has to do this multiple times with different input parameters. To accelerate getting the results, the student asks colleagues and friends to share their computational power.

2. Dynamic Rendering Grid: The employees of a small movie studio want to render a part of an animation movie (see for example Elephant's dream and Blender [6, 3]). To speed up this process they create in their IM application a chat group and use the formed F2F Grid to carry out the distributed rendering task.

The goal of my work behind this thesis was to implement the first version of a framework that would support F2F computing. This framework is called F2F Computing. The main focus was to get the basic functionality of the framework working. This involves the ability of starting up the F2F Computing framework, forming a F2F Grid together with the job submission and communication capabilities between different Grid nodes and tasks. I have not dealt deeply with the issues of security, heterogeneity, and other issues except the base functionality of the framework. My main goal is to demonstrate the simplicity of this lightweight Grid implementation and application development.

In this work I will look at the example network setup depicted in Figure 1.1. It shows the connections between the computers of four people. Keio and Evelin are wirelessly connected via a network address translation firewall (NAT) to the Internet. Peeter has an Ethernet connection via a different NAT to the Internet and Kalle is connected directly (for example via a dial-up) without firewall to the Internet. They all have access to the global chat servers located in the Internet. Keio uses two instant messaging protocols: Jabber and

MSN. Evelin and Peeter use only Jabber. Kalle uses only MSN. This means that Keio can chat with all other three participants, but Kalle cannot chat with Evelin and Peeter. For the rest of the thesis I imagine that Keio needs to run a computationally intensive task - render a movie - and wants to run it together with Evelin, Peeter, and Kalle.

The thesis is structured as follows. Chapter 2 will give an overview of current desktop Grid systems and P2P related techniques, which guided the prototyping of the F2F Computing framework. Chapter 3 will cover the architecture and the implementation details of the framework. Chapter 4 will describe the application programming interface (API) of F2F Computing framework and guides how to use it. Chapter 5 describes how to use the framework in practice and my current experience with it. As the current version of the F2F Computing framework is far from being complete, I conclude this work with the summary of current state and directions for further improvements.

Chapter 2

Related Work

This section will give an overview of current desktop Grid systems and P2P related techniques, which guided the prototyping of a new lightweight P2P Grid framework. It will explain the main concepts of these approaches and how they influenced the decisions in the development of the F2F Computing framework.

2.1 BOINC

BOINC (Berkeley Open Infrastructure for Network Computing) [11] is a non-commercial desktop Grid platform that enables the creation and management of computational intensive projects to which ordinary PC owners can donate cycles of their processors. It is the generalization of the SETI@home project [12]. BOINC's computational model is very simple and based on a flat client-server architecture: a BOINC Grid has a central server that maintains small independent units of work (tasks) of an embarrassingly parallel compute-intensive project and serves the tasks to voluntary contributors who process them and return the results to the server. PC owners are able to configure which resources and how much they contribute. BOINC deals with batch-processes and does not allow communication between the workers.

BOINC is one of the first projects that successfully uses home computers to solve computational intensive problems. To facilitate the use of home computers for solving these kind of problems is also one of the goals of my thesis. To achieve a high number of participants in a particular project, incentives are used. On the one hand, the advertisement of the good and valuable nature of the problem to solve is important. On the other hand, BOINC also introduces a credit point system for the participants to prove their share in the solution of the problem and allows comparison to other participants. These incentives are essential in gaining people's interest in participating. In F2F Computing approach, these incentives will be provided via wanting to help a friend and being convinced to do this with an initial chat to advertise the need for solving the problem together. Like in BOINC, the configuration of shared resources has to be taken into account in F2F Computing.

In BOINC users must entrust their computers to the big institution where code and computation data is received from. For preventing malicious code execution by a server intruder all executables are signed by the private key, which must be stored separately. Sandboxing is not provided, but BOINC may determine that the application is using too much memory or disk space and interrupt it. To ensure that returned results received by the server are correct and to forestall failure of workers, BOINC supports redundant computing - one task is sent to multiple workers, and a canonical result is selected by the server.

Even, if BOINC claims to be a P2P project, it is definitely a client-server architecture. A communication between nodes is not possible. The architecture is not a random network; it is a star. In F2F Computing, server infrastructure is only used to do connection topology detection and as the last way to have a connection between Grid nodes if direct P2P communication is not possible. Furthermore, only already available and reliable servers that are maintained by big IM providers may be used. Because of this, there will be no administrative overhead.

2.2 JXTA

Sun Microsystems started the JXTA project (derived from the word *juxtapose*: examine side by side, collate) in 2001 to make an open P2P communication platform that would support and simplify the development of distributed applications [15]. JXTA is a set of protocols that define simple overlay networks on top of physical networks. Sun provided the first implementation for Java, which is now developed by the JXTA Community. It can be used to create P2P solutions written in Java. It allows the creation of dynamic virtual networks in which different resources can be published and searched.

JXTA's goal is to enable any inter-connected computers to collaborate and exchange messages independently of the underlying physical network topology in a decentralized manner by providing a simple overlay network on top of the physical network. It defines several protocols which have the core abstractions peer, peer group and pipe. A *peer* is a node that is connected to the JXTA's network; it is identified by an unique ID. *Peer groups* are logical collections of peers sharing some kind of common interest. A peer may simultaneously belong to several peer groups. Figure 2.1 shows the JXTA's overlay network on top of the physical network. *Pipes* are virtual uni-directional communication channels with an unique ID. Pipes are used for exchanging messages between peers. They may be unreliable like UDP or secure and reliable like tunneled TCP connections are. *Protocols* determine how peers, groups, or pipes can be discovered, how messages can be exchanged, and how groups and pipes can be managed. The JXTA project provides implementations of its protocols in C and Java.

JXTA's possibility to offer a unique network layer on top of nearly every available other network layer including any kind of P2P connections would be just the right technology to use for the F2F Computing communication. Due to problems described in section 3.2.5, I could only use JXTA's concept

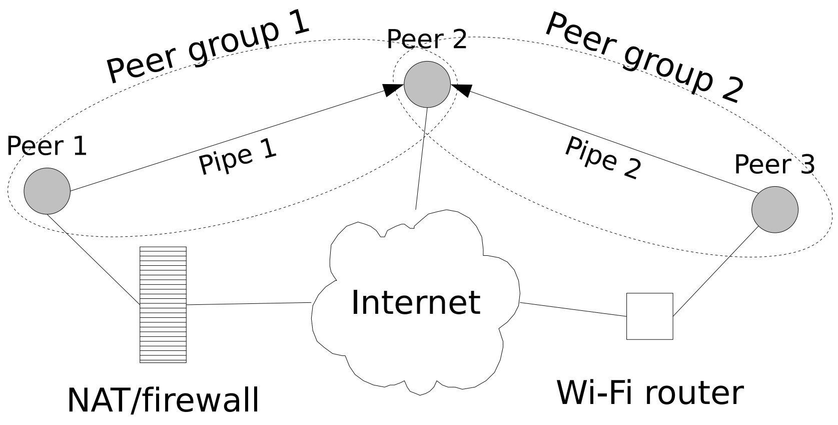


Figure 2.1: JXTA concepts: peers, groups, and pipes

as an inspiration and had to freeze the efforts to use JXTA as a connection provider in F2F Computing framework. Nevertheless, it is worth to keep an eye on JXTA because its concept is great, and should be taken into use if its implementation will improve.

2.3 JNGI

JNGI [9, 29] is a P2P desktop Grid framework that is robust, dynamic and scalable. Its idea is to form spontaneously a global virtual supercomputer. Everybody should be able to contribute to this supercomputer. Everybody should be able to use this supercomputer. This should be possible very conveniently, without big administrative overhead.

JNGI is based on JXTA (see section 2.2), and it contains the following types of peer groups: *monitor* group, *worker* group and *task dispatcher* group. A node can belong to multiple groups and there may be several instances of peer groups in the JNGI's Grid. The monitor group is coordinating the operation of the Grid in the most general level. It analyses the Grid all the time and dynamically assigns positions to the peers so that the Grid would work in the best possible way. The monitor group decides what position should a new peer have, addresses responsibilities of peers which are not doing their work well or have left the network to other peers, and decides where a job should be executed. The worker group is the place where a job is being executed. Each worker group contains the task dispatcher group which distributes tasks to worker peers and communicates with job submitters to receive new tasks and return processed tasks. For better reliability all the information in the Grid is replicated and distributed between several nodes, so nothing is lost if a node crashes.

JNGI provides a simple API for developing embarrassingly parallel applications that can be run on the Grid. The idea of the framework is really great but it did not get popular because its realization relies entirely on JXTA and contributors have no control how given resources are used (one can not limit who or which applications can use the resources and how much of them can be used; contribute nothing or everything), and the project is not alive since 2004. I tried to get the framework running, but failed because of some dependency problems I could not resolve.

The simplicity of JNGI's API and the ideas how easy it should be to use the Grid are the points that are essential for the implementation of F2F Computing. JNGI's dynamic nature is something that should be considered in F2F Computing. Also, JNGI is a good example that shows that if a desktop Grid is not secure and configurable it will not be used. Therefore these issues definitely have to be tackled in the F2F Computing.

2.4 Alchemi

> "Alchemi is a .NET-based framework that provides the runtime machinery and programming environment required to construct desktop Grids and develop Grid applications. It allows flexible application composition by supporting an object-oriented application programming model in addition to a file-based job model. Cross-platform support is provided via a web services interface and a flexible execution model supports dedicated and non-dedicated (voluntary) execution by Grid nodes" [22].

Alchemi [1] uses a *master-worker* pattern of parallel computing. Worker nodes get tasks from a central server node. To protect the system from malicious users, Alchemi uses a role-based security model. The server node maintains a list of permissions that represent different activities which have to be secured, a list of roles which contain a set of permissions, and a list of users with any number of roles. If a user application wants to perform an activity that needs to be authorized, it must provide a user name and a corresponding password. If the given credentials are valid (the name and password match) and the user has a role that contains the particular permission, the requested activity is performed, otherwise not. The principles of this security model are worth to be considered when designing the security model for F2F Computing.

One goal of Alchemi is to ease the development of new Grid applications. Alchemi provides a .NET API which can be used by developers to fulfill this goal. I have used the following idea from Alchemi's API while designing the API of F2F Computing: Grid tasks can be initialized with input data before sending them to the friends' machines for execution, which lowers the communication needs between the tasks. Alchemi supports Grid-enabling legacy applications. This is also a functionality that is worth to be supported by the F2F Computing framework.

The setup of an Alchemi-based Grid is not very simple. The server and worker nodes have to be installed and configured carefully. This is hard for non-IT people. The F2F Computing framework is supposed to make this step easier.

2.5 GridKit

The lightweight Grid middleware GridKit, proposed in [16] deals with highly heterogeneous and (re)configurable interaction paradigms patterns and networking technologies in future Grids. The core of GridKit contains an interaction framework and an overlay framework. The interaction framework is an environment for pluggable interaction paradigms that interface with the pluggable overlay networks in the overlay framework.

In the F2F Computing framework, heterogeneous protocols for communication (different communication providers) are used but the framework does not have plug-in architecture. In the future, developers of the framework might look at the configuration and reconfiguration strategies, support for different communication patterns, and support for different programming languages (usage of OMG IDL in the framework's API) that is done by the GridKit. At the moment, support for different operating systems is bound to the fact that the F2F Computing framework is written in Java.

2.6 Turtle F2F

Another project that uses the friend-to-friend term is Turtle F2F [26], which is designed as a secure environment for friends to exchange information. Turtle F2F is intended mainly for the safe sharing of sensitive data in a P2P manner, aiming to support mainly the right to freedom of speech. The similarity with my use of the term F2F is that trust relationships between pre-existing friendship peers are used as basis for their secure network. The difference with the F2F concept that is described in this thesis is that anonymity built into their system has nothing to do with the friend relationships we have in mind. Turtle F2F does not use IM either.

2.7 NAT traversal

Direct P2P communication across network borders can be very difficult. Especially if the connection should cross multiple network address translator devices (NATs). Compare, for example, the connection from Keio to Peeter in Figure 1.1. To make a direct connection between their computers two NATs have to be crossed. Neither Keio's nor Peeter's computer have a public IP-address. The technique enabling this crossing is called *NAT traversal*. There are various approaches, which describe this technique. The most complete document at the moment collecting these is the Interactive Connectivity Establishment (ICE) draft [27] that is developed by the Internet Engineering Task Force's (IETF) Multiparty Multimedia Session Control working group. Unfortunately I could not find any suitable library that would provide NAT traversal for F2F Computing framework, thus this functionality has to be implemented.

2.8 Instant messaging

IM describes a form of real-time communication between two or more (usually two) persons. In contrast to emails, messages are delivered immediately. Also the availability of the parties is usually visible. Received messages are cached in a local cache and can be read later if the receiving party is not currently near the receiving computer. IM is usually offered by a service provider as messages are usually sent to its server infrastructure and then forwarded to the respective receiver client. Some examples of these service providers are Microsoft with MSN Messaging, Yahoo with Yahoo! Messaging, Google with Googletalk (which is actually Jabber), or Skype with Skype chat. Nevertheless, there are also P2P IM solutions not dependent on a central server infrastructure (like the old Unix-talk). These have usually a much smaller user base than the big ones formerly mentioned.

The applications allowing access to the IM networks are called IM *clients*. There are clients which allow only access to one IM network like the MSN Messenger or multi-protocol clients like Miranda, Trillian, or Pidgin (for a comparison of IM clients see this Wikipedia page [4]). A remarkable multi-protocol client is SIP Communicator [19], which supports like Skype also Voice over Internet Protocol (VoIP). It supports popular IM and telephony protocols such as SIP, Jabber, AIM/ICQ, MSN, Yahoo, Bonjour, and IRC. SIP Communicator is a free open source project initiated by Emil Ivov at the Louis Pasteur University in Strasbourg, France. It is developed completely in Java. As it is based on the OSGi [25, 2] component framework, it is very easy to extend it with new plug-ins. These are the reasons why SIP Communicator was the IM application that was chosen to be used in the F2F Computing development. Also being developed in Java it facilitates the deployment of distributed interoperable Java applications.

2.9 Important concepts for F2F

Our main idea is to use IM as a triggering distributed application for setting up a computational Grid between friends, which should minimize the administrative efforts. The installation of an F2F-compatible IM client program and forming a chat group with the friends should be all that is needed to set up a Grid. The resulting environment is called an *F2F Grid* (or, abbreviated, *frid*), which allows quickly to start parallel applications. The creator of the frid starts an F2F application and each peer has to answer a simple question: "Do you allow {alias of your friend} to use your PC?" (it should be possible to configure the system so that this question is answered automatically in some cases (for instance you might want to allow your best friend to use your PC always)).

Another important concept is the ability of computing nodes to interact via sending and receiving messages. This opens a door for much wider class of distributed applications than simply embarrassingly parallel computations tackled by most of the existing desktop Grid solutions so far. Distributed applications that involve P2P communication can be developed. An IM protocol is used for starting up the connection between Grid nodes, but the communication channels are built up thereafter independently using the fastest available protocol, with the option to use even NAT traversal methods.

The choice of the Java based free SIP Communicator allows rapid prototyping of heterogeneous Grid applications. Also its extensibility and multi-protocol support simplifies the development of the F2F Computing framework a lot.

To ease the development of applications that can be run on top of the F2F Computing framework, it is essential to provide a simple but powerful API that is well documented. The API should allow the usage of different programming models like Alchemi (see section 2.4), and support various interaction patterns like GridKit (see section 2.5).

The security and configuration issues have to be dealt with, otherwise the F2F Computing framework will not be accepted by the wider public. Owners of PCs should have precise control over their resources at any moment like in BOINC's approach (see section 2.1). Java may be helpful in providing a sandbox environment by limiting access to certain resources, or even our own virtual machine might be developed. The trust between friends may be something that can be used while dealing with the security issues. We trust our friends, but even they can have bugs in their code.

The framework should provide the functionality that helps applications to survive the failure of some nodes. It may include replication of tasks so that one task is running on more than one node at a time, and checkpointing and recovering the states of this task. This thesis does not cover the fault tolerance, security, and configuration matters, therefore the next versions of the F2F Computing framework should handle them.

This thesis focuses on getting the basic functionality of the framework working. This involves the ability of starting up the F2F Computing framework, forming a frid together with the job submission and communication capabilities between different frid nodes and tasks.

Chapter 3

F2F Computing framework

This section will cover the architecture and the implementation of the F2F Computing framework.

3.1 Architecture

For the following description please refer to Figure 3.1. The framework consists of three layers: communication, computation, and application. The communication layer provides the connection that is used to exchange messages between the nodes in a frid. The core functionality is located in the computation layer that uses communication layer. The computation layer provides abstractions for F2F Computing applications in an interface, called the `F2F Computing API`. It provides access to the concepts Peer, Job, and Task. For a closer look at these concepts see Figure 3.2.

A *Peer* corresponds to the entity in control of one of the users participating in solving a respective computational task. If Keio and Evelin would like to compute a problem together and both of them are logged in to their IM application on one computer each, it makes sense to regard Keio's computer as one Peer and Evelin's as the other one. The relation between different Peers is called `is friend`, as in our example of the social network between Keio and Evelin; these are regarded to be the friends who have the ability to do F2F computing together. The problem Keio and Evelin want to compute together is the *Job*. One of the both can create this Job (or also multiple Jobs - regard the `is created by`-relation in Figure 3.2). The executed parts making up a Job are called *Task*s. This means the Job consists of (compare association `consists of` in Figure 3.2) multiple Tasks which run on (see association `runs on`) specific Peers. Each Peer has information about the Jobs (compare the `knows`-relation) which have at least one Task running on the Peer.

The main functionality the API has to provide for the application layer is the following:

- create a new Job;
- submit the Tasks forming this Job into specific Peers;

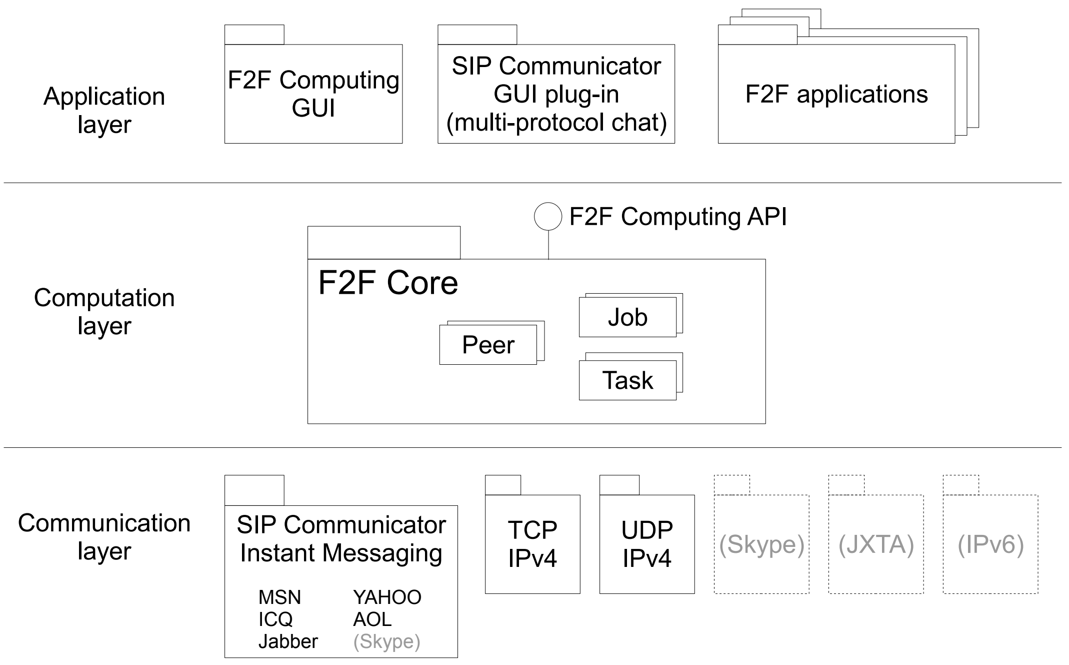


Figure 3.1: F2F Computing architecture

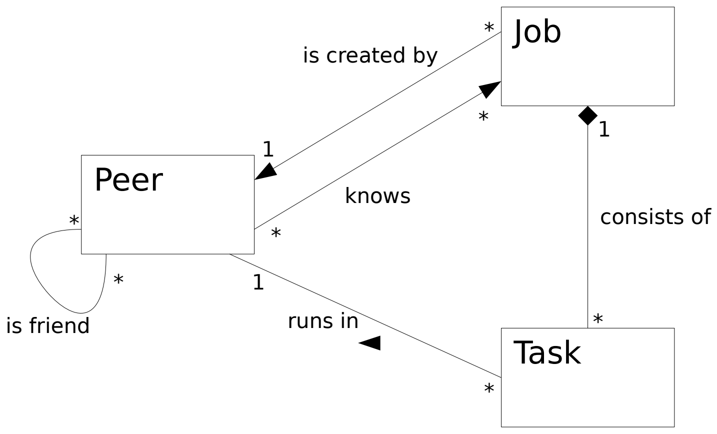


Figure 3.2: Relations between peers, jobs, and tasks

- work out the ability of the Tasks to exchange messages.

Chapter 4 will describe the F2F Computing API in detail.

On the communication layer are the different connection providers via which the F2F Computing can operate. The first communication channel is over IM protocols and this is provided by IM connection provider that is based on SIP Communicator (see section 3.2.1). SIP Communicator itself already supports the usage of MSN, Yahoo, ICQ, AOL, Jabber and many more protocols. The design is planned so that the connection providers can be exchanged on the fly; the framework uses always the best available connection provider. This is reasonable because the IM connection is usually very slow and a more direct connection is preferable. The easiest case is just to switch to a TCP-IP socket connection as it would be possible between Keio and Evelin or between Kalle and anybody else shown in the initial Figure 1.1. For a fast and efficient connection between Keio and Peeter, there needs to be a hole punched in either the NAT 1.2.1.34 or the NAT 1.2.2.56. This is supported by the UDP connection provider (see sections 2.7 and 3.2.3). Alternatives here are the usage of connection providers that make use of technologies like Skype (see section 3.2.4), which uses hole punching itself, JXTA (see section 3.2.5), or later IPv6. However, any other connection provider could be plugged in here, making F2F Computing easily extendable with new technologies in the future.

On the application layer are the programs that make use of the F2F Computing API. These are the F2F Computing GUI, the SIP Communicator GUI plug-in, and the F2F applications implementing the Jobs. The F2F Computing GUI is described in section 3.2.8. The SIP Communicator GUI plug-in is presented in section 3.2.7. It is an OSGi-based multi-protocol group chat plug-in that provides connection between SIP Communicator and the F2F Computing framework (for the description of the plug-in development for SIP Communicator compare the developer documentation on [19]). A simple example of the implementation of an F2F application is given in section 5.1.

3.2 Implementation

This section describes different ways and techniques that are used in the implementation of the F2F Computing framework. The main focus is on describing the different connection types of the communication layer between the peers. Each type of connection has a priority and the framework uses the connection with the highest priority.

3.2.1 Instant messaging connection

The first communication channel between two peers is via some IM protocol that is supported by the SIP Communicator (like MSN, Jabber, or ICQ). I wanted to use the IM channel for exchanging the F2F Computing framework messages between the peers. To achieve this, I implemented a solution that serializes the framework messages and surrounds them with a special XML tag. In this form the messages can be sent via the IM channel like ordinary chat messages. The solution includes a filter mechanism for SIP Communicator which filters out the messages with the tag, and forwards them to the framework. This communication channel ensures that, if friends see each other in their contact list as online and can chat, the F2F Computing framework can exchange messages between these peers. Of course the framework tries to establish a better link between the peers. This will be covered in the next paragraphs.

3.2.2 TCP connection

When the first communication channel to a remote peer has been established (this is momentarily the IM connection, but it could be any other type of connection theoretically), the framework tries to create the TCP connection between the local and the remote peer. The used algorithm is the following:

- (preconditions) peers A and B know their local IPs and are listening on a port on each IP (peer A on sockets A.IP1:P1, ..., A.IPn:Pn, and peer B on sockets B.IP1:P1, ..., B.IPm:Pm);
- A and B exchange the information about the sockets that they are listening on over the existing communication channel;
- both try to make a socket connection to the remote peer in parallel (m threads start doing this on peer A, and n threads start doing this on peer B);
- each peer selects the first valid connection that is made or none if all the tries fail;
- and, if both peers succeed, they agree which connection will be used finally.

The TCP connection has higher priority than a IM connection, meaning that if a peer is contactable via TCP and IM channels TCP is used.

3.2.3 UDP connection with NAT traversal

When the first communication channel to a remote peer has been established (this is usually the IM connection, but it could be any other type of connection theoretically), the framework tries to create also the UDP connection to this peer. Artjom Lind has designed and implemented the NAT traversal module that tries to establish a UDP connection between peers. His work is described in detail in [21]. In general, the following algorithm is used:

- (preconditions) peers have gathered their STUN info;

- peers exchange their STUN info over the existing communication channel;
- based on the data in the STUN info the peers decide if the traversal is possible and how it is exactly done (it depends on the types of used NAT and firewall devices).

On top of the UDP connection provided by the NAT traversal module I implemented a layer guaranteeing the reliability of the connection. There is still work to do to optimize package size and transmission rate. The UDP connection has priority which is higher than the IM connection but lower than the TCP connection.

3.2.4 Skype connection

When I started to implement the F2F Computing framework the SIP Communicator was not used. Instead of this, Skype was used as the provider of social networks (friends) and the communication channel. This was achieved with the usage of the Skype API in the communication layer. This proved that a frid can be composed on top of a Skype network. But as the licensing issues with the Skype API are not very simple, the development that uses the Skype API was freezed until the licensing issues are fixed. Licensing issues are solved now. Still problems with Linux adapter. However, the student Janno Toots is trying to solve these problems in his bachelor thesis.

3.2.5 JXTA connection

When I started to use the SIP Communicator as the provider of social networks, at first I tried to use the JXTA community Java implementation for the connection provider. Unfortunately, setting up the JXTA infrastructure is not straightforward and requires knowledge of the network topology. Without intermediate nodes (rendezvous peers), JXTA is unable to traverse NATs, which is a necessity for the F2F Computing framework. Due to the lack of concise examples and an up-to-date tutorial, programming with JXTA was a very time consuming and error prone process. When I was dealing with JXTA, the tutorial was for JXTA release 2.3, the stable release was version 2.4, and examples where only available from the source repository for the unstable 2.5 version. The complex JXTA architecture and protocols made understanding failures and debugging new code very difficult. Additional difficulties arose even using JXTA within the same local network: initialization of connections took up to 60 seconds and would fail often. The problems with JXTA apply also to other projects. One of them is Xeerkat [10] which finally replaced JXTA with the XMPP (the successor of Jabber) communication protocol. Because of the experienced unreliability and the complexity of the JXTA project the efforts on using JXTA in the communication layer of F2F Computing framework were frozen, and I started to introduce the IM, TCP and UDP connection providers. Nevertheless, it is worth to keep an eye on JXTA because its concept is great, and should be taken into use if its implementation will improve.

3.2.6 Core of the F2F Computing framework

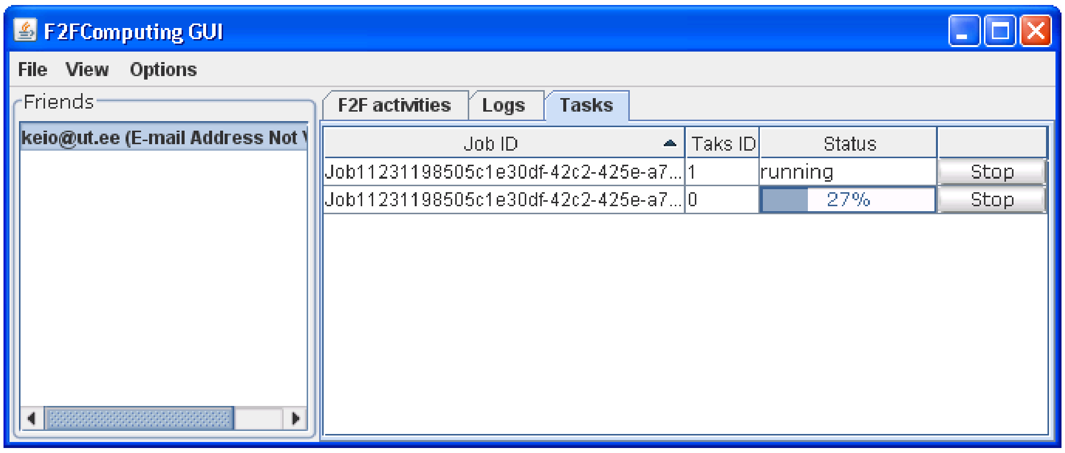
As described in section 3.1 the core functionality of the F2F Computing framework resides in the computation layer which provides access to the concepts Peer, Job, and Task via the F2F Computing API (see Figure 3.1 and Figure 3.2). Besides the main functionality (see the listing in section 3.1) the computation layer provides some features that make it flexible and extendable. The design is planned so that any module, that may be developed and added later, can easily send and receive custom messages between Peers. A module can register itself with the framework to listen for the specific type of messages, and if such a message is received it is forwarded to the module. For instance, the TCP and the UDP communication modules use this approach to exchange the data that is needed to run the processes that test if the respective connection can be established. Additionally, a module can listen to the events when a Peer becomes online or goes offline, and when a task starts or stops running. More detailed description of these features can be found in section 4.

3.2.7 SIP Communicator GUI plug-in

The SIP Communicator GUI plug-in is the tool which allows to create a frid in an intuitive way. This plug-in allows to create multi-protocol chat rooms (meaning that contacts via different kind of IM protocols can take part in one chat room) and enables the creator of the chat room to run an F2F application with the help of the participants of the chat room (of course, the participants have to allow this application to be run on their PCs, as already mentioned in section 2.9). Only the creator of the chat room can add new contacts to the chat room. If he/she tries to add a contact to the chat room the framework checks if this contact is F2F-capable, and if the checking succeeds (the `is friend`-relation is made, see Figure 3.2) the friend is invited to join the chat room. If the creator has gathered all the wanted friends to the chat room he/she can start the desired F2F application in the following simple way: open the dialog in which the F2F application has to be specified and start it. This process is described more deeply with illustrating screen-shots in section 5.2.

3.2.8 F2F Computing GUI

For the following description please refer to Figure 3.3. At the beginning of the project the F2F Computing GUI was meant to be the application that allows users to start F2F programs. The idea to use the group chat window in SIP Communicator came up in a later stage of the project. At this point this functionality was removed from the F2F Computing GUI. The main purpose of the GUI at the moment is to show different kind of debugging information (in the logs and the F2F activities tabs, plus in the window that can be opened from the View menu) and the state of tasks running on the local machine (in the Tasks tab). Additionally, from the Tasks tab the user can interrupt the tasks by stopping them. The Friends section in the GUI shows these friends which are detected by the framework to be F2F-capable. From the Options



keio@ut.ee (E-mail Address Not \

Job ID	Taks ID	Status	
Job11231198505c1e30df-42c2-425e-a7...	1	running	Stop
Job11231198505c1e30df-42c2-425e-a7...	0	27%	Stop

Figure 3.3: F2F Computing GUI

menu the users can select if their friends are allowed to use their PC by default (if the user chooses to allow, the pop-up message that asks for the permission is not shown each time a friend starts an F2F application).

In the future the F2F Computing GUI should be developed to be the tool that shows rich monitoring information like state of tasks and the resources they use (CPU, RAM, hard disk, network bandwidth), network and application topology. Also, the GUI must allow the PC owners precisely to configure and control the resources that are made available for the friends.

3.3 Roundup

This chapter was explaining the concepts used for the F2F Computing framework in detail. It showed the architecture, explained in detail the different connection types, which are possible in the framework, and presented the GUIs I have implemented. To avoid confusion, it should be mentioned again, that security and configuration issues are not part of this work and will be included in later versions of the framework.

Chapter 4

API and programmer's guide

This section describes the API of F2F Computing framework and guides how to use it.

4.1 API for F2F application developer

The F2F Computing framework supports programs that are implemented using a *master-slave programming paradigm* [20]. As described in section 3.1 a computational problem that friends want to compute together is called the job. A job is created if an F2F application is started. The process of starting an F2F application is described with illustrating screen-shots in section 5.2. A job may contain several tasks and each task may be running on a different computer. The master-slave programming model considers one of these tasks to be special. This special task is called the *master task*, and others are called *slave tasks*. The master task is the first task that is started if a new job is created. It runs on the job creator's node. The purpose of the master task should be the creation of slave tasks, the distribution of initial data (preparation of slave tasks), and the end-result assembly. The intention of the slave tasks should be to calculate parts of the end-result and to return them to the master task.

For example, if Keio starts an F2F application to render a movie which consists of 120 frames together with Evelin, Peeter and Kalle (see Figure 2.1), the application might work in the following manner: The master task of the application is started in Keio's computer. The purpose of the master task is to divide the rendering job between slave tasks, which render different parts of the movie, to execute these tasks on separate nodes, gather the rendered movie clips and join them into the final movie. Therefore, the master task creates four slave tasks - the first one in the same computer (meaning that the master task and one slave task will be running on the same machine), the second one on Evelin's computer, the third one on Peeter's computer, and the fourth one on Kalle's computer. After this, the master task sends rendering input data to the slave tasks and tells to each slave task which frames it should render - the first slave task should render frames 1 to 30, the second slave task should render frames 31 to 60, the second slave should render task frames 61 to 90, and the fourth slave task should render frames 91 to 120. The slave tasks render the movie clips, send them to the master task, and the master creates the desired movie from them. A pseudo implementation of this F2F application is described in detail in section 5.1, and the according real implementation is given in Appendix A.

The F2F Computing API provides several Java classes that form the base on top of which F2F applications can be built. The following sections describe these classes.

4.1.1 Task

The `Task` (the exact class name is `ee.ut.f2f.core.Task`) class is the base class for all tasks that can be run by the F2F Computing framework, both master and slave. This class represents the Task concept described in section 3.1. Technically a task is a thread which is started by the framework, therefore the `Task` class extends `java.lang.Thread` class. By extending the `Task` class and implementing the `runTask()` method one can create an F2F application. An F2F program must contain at least the implementation of the master task. The number of different types of slave tasks in an F2F program is not limited. The most important methods of the `Task` class are the following:

- `void runTask()`
  This is an abstract method which is invoked by the framework if the task is started. Its realization should describe the steps of the task's algorithm. This is the place to where one can put their code and let it be executed by the framework.

- `boolean isStopped()`
  This method returns `true` until the framework allows the task to run. If this method starts to return `false` the task should exit the `runTask()` method and stop all the sub-threads it has started. For instance, this method starts to return `false` after a user has chosen to stop the task from the F2F Computing GUI (see Figure 3.3).

- `void taskStoppedEvent()`
  This method is called by the framework if the task should stop running. At the same time the `isStopped()` method starts to return `false`. This method should be overwritten if the task wants to react quickly (which is polite) on the event that asks it to stop.

- `void setProgress(int progress)`
  Often the user of an F2F application wants to know the progress of the application. The progress should indicate how much of the task is completed. The value of the task's progress can be asked via the `getProgress()` method; the default progress value is `-1`. This method sets the progress value of the task equal to `progress` and informs the listener objects (described in section 4.2.6) which are interested in this kind of events. The Tasks tab in the F2FComputing GUI (see Figure 3.3)

is one of those listeners. It interprets the value of tasks progress in the following way: the value less than `0` means the progress is not determined (the GUI does not show the progress bar in this case), the value from `0` to `100` means the task is 0 to 100% completed (the GUI shows according progress bar), the value greater than `100` is considered equal to `100`.

- `String getTaskID()`
  This method returns the unique identification (ID) of the task. Each task has the unique ID in the job. At the moment the framework gives IDs to tasks by the following algorithm: the master task has ID `"0"` and the slave tasks have IDs from `"1"` to `"N"`, where N is the number of slave tasks. The ID is used for communication and for debugging.

- `TaskProxy getTaskProxy(String taskID)`
  This method tries to return a proxy to a remote task which ID is `taskID`. If the proxy to the remote task is asked the first time, it is created and saved for later use. The framework checks if a task with the given ID exists (in the context of the job this task belongs to) and then returns the proxy to it; if such a task does not exist, `null` is returned. The `TaskProxy` class is meant to be used to exchange messages between the tasks; it is described more in section 4.1.2.

- `void messageReceivedEvent(String remoteTaskID)`
  This method is called by the framework if a message is received from a remote task with the ID `remoteTaskID`. The message can be read from the corresponding task proxy. An implementation of the `Task` class that wants to be notified about incoming messages should overwrite this method and take the actions it wants (probably read the message and process it).

- `Job getJob()`
  Each task is part of a job, like described in section 3.1. This method returns the job this task belongs to. The `Job` class is described in section 4.1.3.

4.1.2 TaskProxy

The `TaskProxy` (the exact class name is `ee.ut.f2f.core.TaskProxy`) class represents the link between two tasks. It should be used to send/receive messages to/from the task. The received messages are being held in a FIFO queue until they are read. For example, lets assume that task A is running on Keio's node and task B is running on Evelin's node and they are exchanging messages. In that case, the task A holds a proxy to the task B and the task B holds a proxy to the task A, and they are using these proxies to exchange the messages. The most important methods of this class are the following:

- `String getRemoteTaskID()`
  This method returns the ID of the task to which this proxy links to. The proxy that the task A holds returns the ID of the task B.

- `void sendMessage(Object message)`
  This method tries to send the `message` object to the corresponding task. The `message` object has to be serializable. This method just sends the message out and returns. It means that it does not guarantee that the message has reached the destination task.

- `void sendMessageBlocking(Object message)`
  Like previous method this method tries to sends the `message` to the corresponding task. The difference is that this method blocks until the receiver has got the message. This means the method waits until the framework has made sure the message has reached the destination machine and can be read by the destination task.

- `Object receiveMessage(long timeoutInMillis)`
  This method should be used to receive a message from the corresponding task. If the task A wants to read a message from the task B, it may use this method to do this. The parameter `timeoutInMillis` defines the timeout (in milliseconds) how long the method should wait if a message has not been received yet; `0` or negative value causes the method to wait until a message is received. The method returns the first message from the incoming message queue (and removes it from the queue) or `null` if there are no messages after the timeout.

- `Object receiveMessage()`
  This method is equivalent to the invocation of `receiveMessage(0)`.

- `boolean hasMessage()`
  This method returns `true` if there is at least one message received from the task which this proxy belongs to, otherwise `false` is returned. If the task A wants to read a message from the task B, it may use this method of the proxy to the task B to check if there are any messages to read.

- `int getMessageCount()`
  This method returns the size of the incoming message queue. If the task B has sent 3 messages to the task A and the task A has not read any of them, and the task A invokes this method of the proxy to the task B, it should return `3`. If the task A then reads one message and calls this method again, it should return `2`.

4.1.3 Job

The `Job` (the exact class name is `ee.ut.f2f.core.Job`) class represents the Job concept described in section 3.1. The purpose of this class is to hold the jar-files that are needed for the tasks of this job, to keep track about the tasks that make up the job and in which peers they are running, and to allow the master task to submit new tasks. The most important methods of the `Job` class are the following:

- `Collection<F2FPeer> getPeers()`
  This method returns the collection of peers who were present in the group chat when the job was started. These are the peers into which the master task of the job may submit the slave tasks with the `submitTasks()` method (if the peers allow to use their PC). The `F2FPeer` class is described in section 4.2.2.

- `String getMasterTaskID()`
  This method returns the ID of the master task of the job. This ID can be used to get the proxy to the master task using the `Task.getTaskProxy()` method.

- `Collection<String> getTaskIDs()`
  This method returns the IDs of all the tasks this job consists of. The IDs can be used to get the proxies to the corresponding tasks using the `Task.getTaskProxy()` method.

- `void submitTasks(String className, int taskCount, Collection<F2FPeer> peers)`
  This method of a job is one of the two methods that can be used to submit new tasks to the job. I mention again that only the master task of the job may use this method; if a slave task tries to submit new tasks an exception is thrown. The parameter `className` defines the name of a Java class implementing the `Task` class. The parameter `taskCount` defines the number of new tasks the master wants to submit. The parameter `peers` defines the collection of peers in which the new tasks should be executed. The `peers` collection has to be a subset of the `getPeers()` return value, meaning that tasks can only be submitted to those peers who were present in the group chat when the job was created. The method waits until `taskCount` peers of the `peers` have allowed to use their PC (the requests were sent out during the job creation to all the peers who were in the group chat at that moment), then new tasks are added to the job and executed in corresponding peers. If not enough peers have allowed to use their PC in 1 minute, the method throws an exception.

- `void submitTasks(Collection<Task> tasks, Collection<F2FPeer> peers)`
  This method is similar to the previous one. The difference is that this method allows to submit already initialized tasks. This approach allows the master task to prepare the new tasks with the initial data before they are sent to the peers for execution. The parameter `tasks` defines the collection of prepared tasks.

4.1.4 Roundup for F2F application developer

The key points a developer who wants to create new F2F applications should remember from the previous sections are the following:

- an F2F application consists of tasks;
- each task has to extend the `Task` class and implement the `runTask()` method, which is called when the task is started;
- one of these tasks is the master task, it is the task that is started if the application is executed;
- the master task can submit slave tasks by invoking the `getJob().submitTasks()` method;
- a task can exchange messages with other tasks with the help of proxies that it can get using the `getTaskProxy()` method.

4.2 API for the F2F Computing framework developer

As mentioned in section 3.2.6 the computation layer of the F2F Computing framework is designed to be easily extendable. Any module, that may be developed and added later, can easily send and receive custom messages between peers. A module can register itself with the framework to listen for the specific type of messages, and if such a message is received it is forwarded to the module. Additionally, a module can listen to the events when a Peer becomes online or goes offline, and when a task starts or stops running. Also, as mentioned in section 3.1, it is quite easy to interface a new connection provider module to the framework.

The following paragraphs describe the classes and interfaces of the F2F Computing API that provide these features, and explain how a developer can use them to extend the framework.

4.2.1 F2FComputing

The `F2FComputing` (the exact class name is `ee.ut.f2f.core.F2FComputing`) class is the very center of the computation layer. Because of that the methods of this class comprise all the features described in section 4.2 and use lots of different classes and interfaces. It is more readable and understandable if I will not describe the methods here as one list, but start to describe the classes and interfaces and include the descriptions of the related `F2FComputing` class methods in there.

4.2.2 F2FPeer

The `F2FPeer` (the exact class name is `ee.ut.f2f.core.F2FPeer`) class represents the Peer concept described in section 3.1. The purpose of this class is to provide a logical link to the corresponding friend's PC. This link can be used to send custom data to the peer. The user of this class does not have to worry about the real connection that is used for the data transfer. The framework takes care of this and ensures that the best possible way of communication is used. The most important methods of the `F2FPeer` class are the following:

- `String getDisplayName()`
  This method returns the display name of the peer. This is the same name that a contact has in the SIP Communicator contact list.

- `UUID getID()`
  Each peer has an unique ID in the framework. This method returns the ID of the peer.

- `void sendMessage(Object message)`
  This method tries to send the `message` object to the peer. The `message` object has to be serializable. The method just sends the message out and returns. It does not guarantee that the message has reached the destination peer. The `TaskProxy.sendMessage()` method described in section 4.1.2 uses this method.

- `void sendMessageBlocking(Object message, long timeout)`
  Like previous method this method tries to sends the `message` to the corresponding peer. But this method is blocking, it returns only when the message has reached the destination (or throws an exception in case of a failure or timeout). If the timeout is reached the `MessageNotDeliveredException` is thrown. This exception does not mean that the message was definitely not received by the peer, but means that the acknowledgment from the peer was not received (so the message may have been reached the destination or not). The `timeout` parameter defines the maximum amount of time to wait (in milliseconds) before throwing the exception. If the `timeout <= 0`, the timeout is never reached.

- `void sendMessageBlocking(Object message)`
  This method is equivalent to the invocation of `sendMessageBlocking(message, 0)`. The `TaskProxy.sendMessageBlocking()` method uses this method.

The methods of the `F2FComputing` class related to the `F2FPeer` class are the following:

- `F2FPeer getLocalPeer()`
  This method returns the `F2FPeer` object that represents the local machine.

- `Collection<F2FPeer> getPeers()`
  This method returns the collection of peers that are known to the local machine. These peers can be used to do F2F computing together.

4.2.3 F2FMessageListener

If a module of the F2F Computing framework wants to exchange some messages between the peers it can use the `F2FPeer` class for sending the messages out. Somehow the messages have to reach the desired module in the receiver peer. The `F2FMessageListener` interface enables this. The `F2FMessageListener` interface declares the following method:

- `void messageReceived(Object message, F2FPeer sender)`

The module has to define a message class that it uses as the data carrier, realize a class implementing the `F2FMessageListener` interface, and register this class to be the listener for the messages of the defined type. The following methods of the `F2FComputing` class deal with the registration:

- `void addMessageListener(Class messageType,`
  `F2FMessageListener listener)`
  This method registers the `listener` object to be interested in messages of the type `messageType`. If now a message of the type `messageType` is received it is forwarded to the `listener` by invoking the `listener.messageReceived()` method. Many listeners may listen for the same type of messages. An incoming message is forwarded to all of them.

- `void removeMessageListener(Class messageType,`
  `F2FMessageListener listener)`
  This method unregisters the `listener` object to be interested in messages of the type `messageType`.

For instance, the module that checks if the TCP connection is possible between the peers uses this approach to exchange the information about the IP addresses of the peers.

4.2.4 PeerPresenceListener

A module may be interested in knowing if a peer comes online or goes offline. If a peer comes online the messages can be sent to the peer. If a peer goes offline the messages sending is not possible any more. For instance, the module that checks if the TCP connection is possible between the peers is such a module. If a peer becomes online the module starts the process that checks if the TCP connection can be made to the peer. To allow all interested modules to be notified about the events concerning the presence of peers I created the `PeerPresenceListener` interface. This interface declares the following methods:

- `void peerContacted(F2FPeer peer)`

- `void peerUnContacted(F2FPeer peer)`

A module that wants to be notified about these events has to realize a class implementing the `PeerPresenceListener` interface, and register this class to be the listener for the events. The following methods of the `F2FComputing` class deal with the registration:

- `void addPeerPresenceListener(PeerPresenceListener listener)`
  This method registers the `listener` object to be interested in events concerning the presence of peers. If a peer comes online all the listeners get notified as the `listener.peerContacted()` method is called for all of them. If a peer goes offline all the listeners get notified as the `listener.peerUnContacted()` method is called for all of them.

- `void removePeerPresenceListener(PeerPresenceListener listener)`
  This method unregisters the `listener` object to be interested in events concerning the presence of peers.

4.2.5 CommunicationProvider

As described in section 3.1 the architecture of the F2F Computing framework contains the communication layer (see Figure 3.1) which provides the connection that is used to exchange messages between the peers. The communication layer may contain several communication providers. The framework uses the best available communication provider that is available between two peers for the data exchange. To ease the development of new communication providers and make them to easily interface with the computation layer I created the `CommunicationProvider` interface. This interface declares the following methods:

- `int getWeight()`
  This method returns the priority of the communication provider. The framework uses this value to detect the best available communication provider. The higher this value is, the better connection should be. The TCP communication provider has implemented this method to return 1000, the UDP communication provider has implemented this method to return 500, and the IM communication provider has implemented this method to return 10.

- `void sendMessage(UUID destinationPeer, Object message)`
  This method should try to send the `message` object to the peer with the ID `destinationPeer`. This method is used by the `F2FPeer.sendMessage()` method, described in section 4.2.2.

- `void sendMessageBlocking(UUID destinationPeer, Object message, long timeout)`
  This method is similar to the previous one, the difference is that it should block until the `message` object has definitely reached the peer with the ID `destinationPeer`. If the timeout is reached the

`MessageNotDeliveredException` should be thrown. The `timeout` parameter defines the maximum amount of time to wait (in milliseconds) before throwing the exception. If the `timeout <= 0`, the timeout should never be reached.. This method is used by the `F2FPeer.sendMessage()`method, described in section 4.2.2.

A new communication provider has to implement the `CommunicationProvider` interface, and to notify the computation layer which peers are available via this communication provider. For this it should use the following methods of the `F2FComputing` class:

- `void peerContacted(UUID peerID, String displayName, CommunicationProvider comm)`
  This method tells to the computation layer that the peer with the ID `peerID` (and with the display name `displayName`) is reachable via the `comm` communication provider. If now according `F2FPeer` object's `sendMessage()` or `sendMessageBlocking()` is used the `comm` communication provider might be used (if it is the best one available) for the data transfer. If the peer was not reachable before via any communication provider the `PeerPresenceListener` objects are notified about a new peer (the `PeerPresenceListener.peerContacted()` method is called, like described in section 4.2.4).

- `void peerUnContacted(UUID peerID, CommunicationProvider comm)`
  This method tells to the computation layer that the peer with the ID `peerID` is not any more reachable via the `comm` communication provider. After this the `comm` communication provider is not used to send messages to the according peer. If the peer is not reachable any more via any communication provider the `PeerPresenceListener` objects are notified that the peer went offline (the `PeerPresenceListener.peerUnContacted()` method is called, like described in section 4.2.4).

If a communication provider receives a message from a peer (for instance the TCP communication provider receives it from a TCP socket and the IM communication provider receives a specially tagged chat message) it must use the following method of the `F2FComputing` class to forward the message to the computation layer:

- `void messageReceived(Object message, UUID senderID)`
  This method forwards the `message` object to the computation layer, and says that the peer with the ID `senderID` has sent it. Then the computation layer forwards it to the proper `F2FMessageListener` objects (the `F2FMessageListener.messageReceived()` method is called, like described in section 4.2.3).

Additionally, if the received message was being sent with the `sendMessageBlocking()` method, the communication provider must send back the acknowledgment that it has received the message.

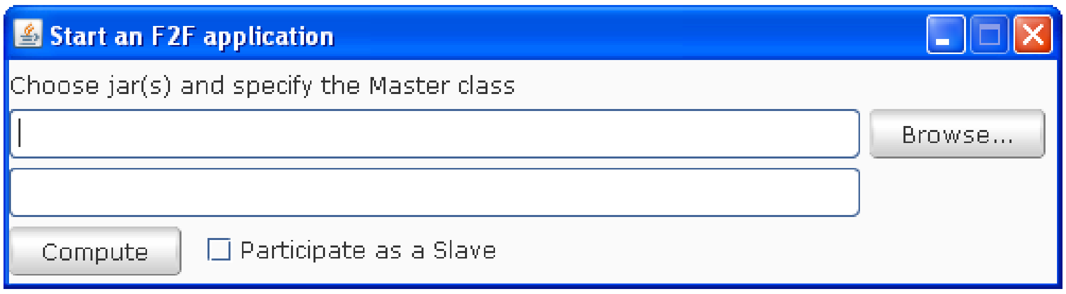


Figure 4.1: Screen-shot, dialog box for the start of an F2F application

4.2.6 TaskListener

A module may be interested in knowing if a task starts to run (the `runTask()` method of the task is called), stops (the `runTask()` method of the task exits) or progresses (the task changes its progress via calling the `setProgress()` method). For instance, the Tasks tab in the F2F Computing GUI (see Figure 3.3) is such a module. To allow all interested modules to be notified about the events concerning the state of tasks I created the `TaskListener` interface. This interface declares the following methods:

- `void taskStarted(Task task)`
- `void taskStopped(Task task)`
- `void taskProgressed(Task task)`

A module that wants to be notified about these events has to realize a class implementing the `TaskListener` interface, and register this class to be the listener for the events. The following methods of the `F2FComputing` class deal with the registration:

- `void addTaskListener(TaskListener listener)`
  This method registers the `listener` object to be interested in events concerning the status of tasks. If a task starts to run all the listeners get notified as the `listener.taskStarted()` method is called for all of them. If a task stops all the listeners get notified as the `listener.taskStopped()` method is called for all of them. If a task updates its progress all the listeners get notified as the `listener.taskProgressed()` method is called for all of them.

- `void removeTaskListener(TaskListener listener)`
  This method unregisters the `listener` object to be interested in events concerning the status of tasks.

4.2.7 JobCreator

For the following description please refer to Figure 4.1. The figure shows the dialog box that is an instance of the `JobCreator` class. The dialog provides the only way for starting an F2F application. In the dialog, the application specific jar-file/files has/have to be named (separated by semicolon), and the name of the master task (this is usually specified in the manifest file and read from there automatically) has to be given. If the `Compute` button is clicked the application is started, which means the master task is initiated in the local machine and its `runTask()` method is called. If the `Participate as a Slave` check-box is marked the master task may submit slave tasks to the local machine, otherwise not. The `JobCreator` class is a private member of the F2FComputing class, thus to open the dialog the following method of the `F2FComputing` class should be invoked:

- `void startJob(Collection<F2FPeer> peers)`
  This method opens a new job creation dialog. The `peers` parameter specifies the peers to where slave tasks may be submitted. If the `Participate as a Slave` check-box is marked, the local peer is added to the collection (if it is not present already). If it is not marked and the peers collection contains the local peer, it is removed.

4.3 MPI support

The Message Passing Interface (MPI) [17] has become the *de facto* standard for programming distributed computationally and communicationally intensive applications. If the API of the F2F Computing framework would support this standard it would be a great virtue. This would enable the developers who are MPI specialists easily to create new F2F applications. Also, it would support the porting of existing MPI applications to the F2F Computing framework. Andres Luuk has designed and implemented the MPI module for the F2F Computing framework. His work is described in detail in [23].

Chapter 5

F2F Computing framework in practice

This section describes how to use F2F Computing in practice and my current experience with it. Section 5.1 will describe an example F2F application and section 5.3 will present some results that I got while running the application on different computers. Section 5.2 will describe how to start an F2F application.

5.1 Example F2F application: Distributed Blender

The example program this paragraph describes was already introduced in section 4.1. This application is called the Distributed Blender. It is based on a free open source 3D content creation suite called the Blender [3]. One of the features the Blender provides is movie rendering: one can render a movie which is described in a blend-file (the file extension is .blend, this thesis does not cover how to create these files). The rendering of a movie might take lots of hours in one computer. The purpose of the Distributed Blender is to allow to render a movie using the power of all the computers in a frid. The idea is that each computer renders a different part of the movie. The prerequisites for the Distributed Blender are that the Blender must be installed and the installation directory must be in the PATH environment variable of the operating system.

If the application is started, the master task (see Listing 5.1) asks initial data from the user (line 10). The user has to specify the input blend-file, select the output location and the format of the resulting movie, and change the number of the start and/or end frames if the rendering of all frames is not desired (see Figure 5.1). This data is stored in the `RenderJob` object. Based on the data given by the user, the master prepares the slave tasks (lines 13-29). Each one is assigned to render a different part of the movie. If the preparation is done, the slave tasks are submitted (line 32). Then the master task waits until the slave tasks have returned the rendered movie clips to it (lines 36-39 and 51-61), and joins the clips into the desired movie (lines 42-46).

```java
public class BlenderMasterTask extends Task
{
  // here the master task collects the results the slave tasks have rendered
  private List<RenderResult> renderedResults =
      new ArrayList<RenderResult>();

  public void runTask()
  {
    // get the initial data from the user
    RenderJob renderJob = getRenderJobFromUser();

    // prepare the slave tasks
    Collection<Task> tasks = new ArrayList<Task>();
    long[] partLengths = splitTask(
        renderJob.getEndFrame() - renderJob.getStartFrame() + 1,
        this.getJob().getPeers().size());
    long startFrame = renderJob.getStartFrame();
    for (int i = 0; i < this.getJob().getPeers().size(); i++)
    {
      BlenderSlaveTask slaveTask =
        new BlenderSlaveTask(
          renderJob.getInputFileName(),
          renderJob.getInputFile(),
          renderJob.getOutputFormat(),
          startFrame,
          startFrame + partLengths[i] - 1);
      tasks.add(slaveTask);
      startFrame += partLengths[i];
    }

    // submit the slave tasks
    this.getJob().submitTasks(tasks, this.getJob().getPeers());

    // wait until the slaves have rendered and returned the movie clips
    // the messageReceivedEvent() method collects the results
    synchronized (renderedResults)
    {
      renderedResults.wait();
    }

    // compose the resulting movie
    composeResult(
        renderedResults,
        renderJob.getOutputLocation(),
        renderJob.getInputFileName(),
        renderJob.getExtension());
  }

  // handle messages from the slave tasks
  // this means collecting the rendered parts
  public void messageReceivedEvent(String remoteTaskID)
  {
    TaskProxy proxy = this.getTaskProxy(remoteTaskID);
    RenderResult renderResult = (RenderResult) proxy.receiveMessage();
    synchronized (renderedResults)
    {
      renderedResults.add(renderResult);
      if (renderedResults.size() == this.getJob().getPeers().size())
        renderedResults.notifyAll();
    }
  }
}
```

Listing 5.1: The Distributed Blender Master Task

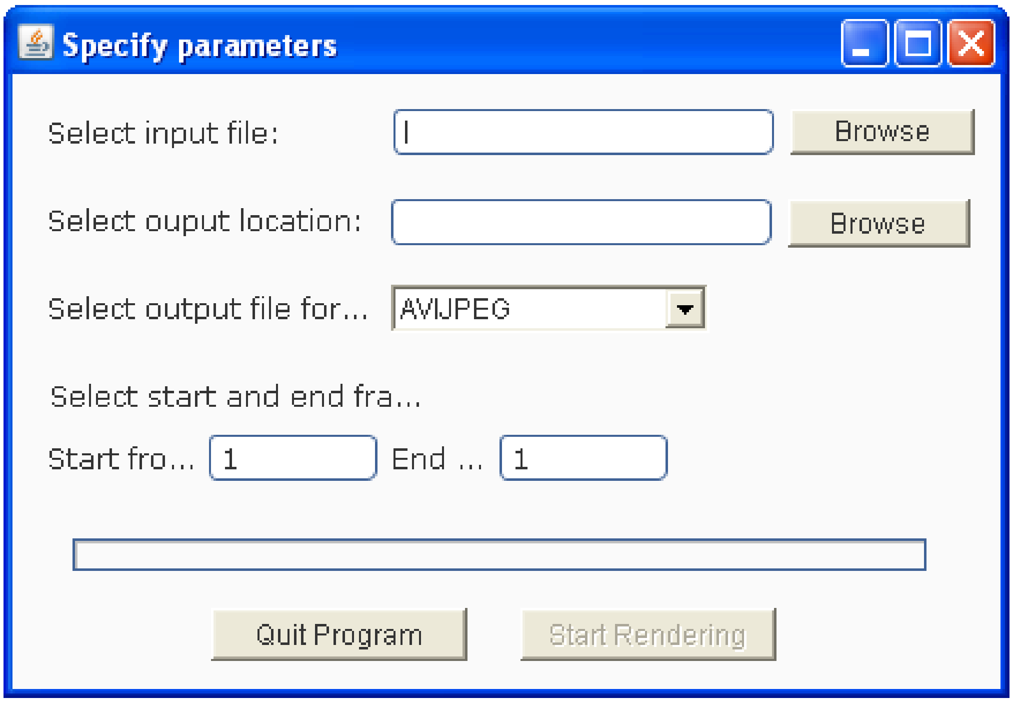


Figure 5.1: Screen-shot, the GUI of Distributed Blender

The slave task (see Listing 5.2) renders the part of the movie the master task assigned to it (lines 10-18). This is done by starting the Blender executable with the appropriate arguments (lines 23-28; such approach is an security issue of the F2F Computing framework, as at the moment tasks can start any external application). After the Blender has finished the rendering (lines 31-50), the slave task sends the resulting movie clip to the master task (lines 53-64).

The compiled application should be packaged in a jar-file. In the manifest file the name of the master task class should be specified with the `F2F-MasterTask` attribute. In our case this would be the following: `F2F-MasterTask: BlenderMasterTask`.

5.2 Starting an F2F application

This paragraph describes how the process of starting an F2F application should look like.

A user of the SIP Communicator, Keio, wants to run the Distributed Blender application, which he just has developed. He sees that some of his friends are online (see Figure 5.2) and decides to use the power of their PCs to complete the rendering quicker. An F2F application can be started from an F2F chat room. Keio creates a chat room and calls it “Keio’s rendering group” (see Figure 5.3). Then, he opens the chat room and drags and drops the friends he wants to join from the contact list into the group chat window (see Figure 5.4). Keio can chat with them and explain the application he wants

```java
public class BlenderSlaveTask extends Task implements Serializable
{
  // the data for rendering
  private String inputFileName;
  private byte[] inputFile;
  private String outputFormat;
  private long startFrame;
  private long endFrame;

  public BlenderSlaveTask(String inputFileName, byte[] inputFile,
      String outputFormat, long startFrame, long endFrame)
  {
    this.inputFileName = inputFileName;
    this.inputFile = inputFile;
    this.outputFormat = outputFormat;
    this.startFrame = startFrame;
    this.endFrame = endFrame;
  }

  public void runTask()
  {
    // start to render the movie clip
    String tempDir = System.getProperties().getProperty("java.io.tmpdir");
    File file = saveFile(this.inputFile, tempDir + this.inputFileName);
    String[] cmdarr = { "blender", "-b", file.getName(), "-o", tempDir,
        "-F", this.outputFormat, "-s", String.valueOf(this.startFrame),
        "-e", String.valueOf(this.endFrame), "-a", "-x", "1" };
    Process proc = Runtime.getRuntime().exec(cmdarr);

    // wait until the frames get rendered
    BufferedReader br = new BufferedReader(
        new InputStreamReader(proc.getInputStream()));
    String line;
    while ((line = br.readLine()) != null && !isStopped())
    {
      if (line.startsWith("'blender' is not recognized")
          || line.indexOf("blender") != -1
          && line.indexOf("not found") != -1)
      {
        System.out.println("Blender not found!");
        break;
      }
      else if (line.startsWith("Append frame")
          || line.startsWith("added frame")
          || line.startsWith("Writing frame"))
      {
        System.out.println("Rendered frame " + line.split(" +")[2]);
      }
    }
    proc.destroy();

    // send the rendered movie clip to the master task
    byte[] renderedFile = loadFile(tempDir +
        generateOutputFileName(
          renderTask.getStartFrame(),
          renderTask.getEndFrame(),
          renderTask.getExtension()));
    RenderResult result =
      new RenderResult(
          this.startFrame,
          this.endFrame,
          renderedFile);
    TaskProxy masterProxy = this.getTaskProxy(this.getJob().getMasterTaskID());
    masterProxy.sendMessage(result);
  }
}
```

Listing 5.2: The Distributed Blender Slave Task

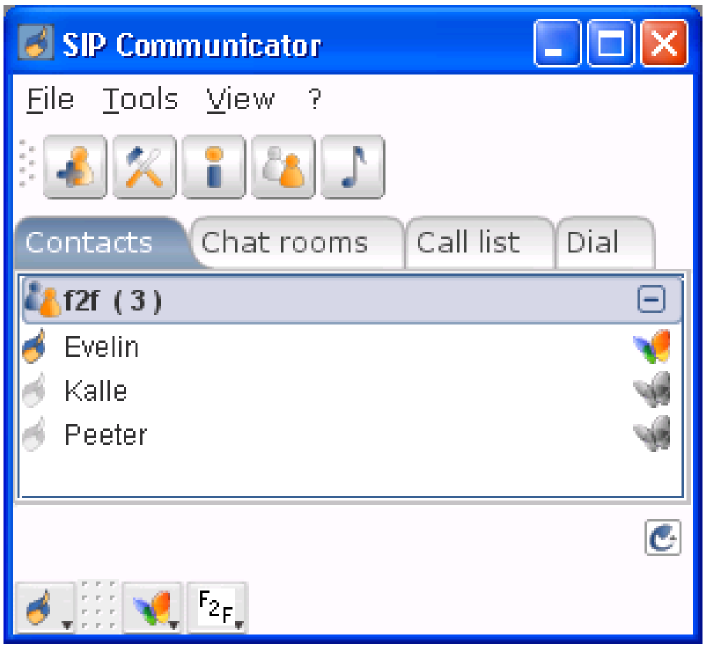
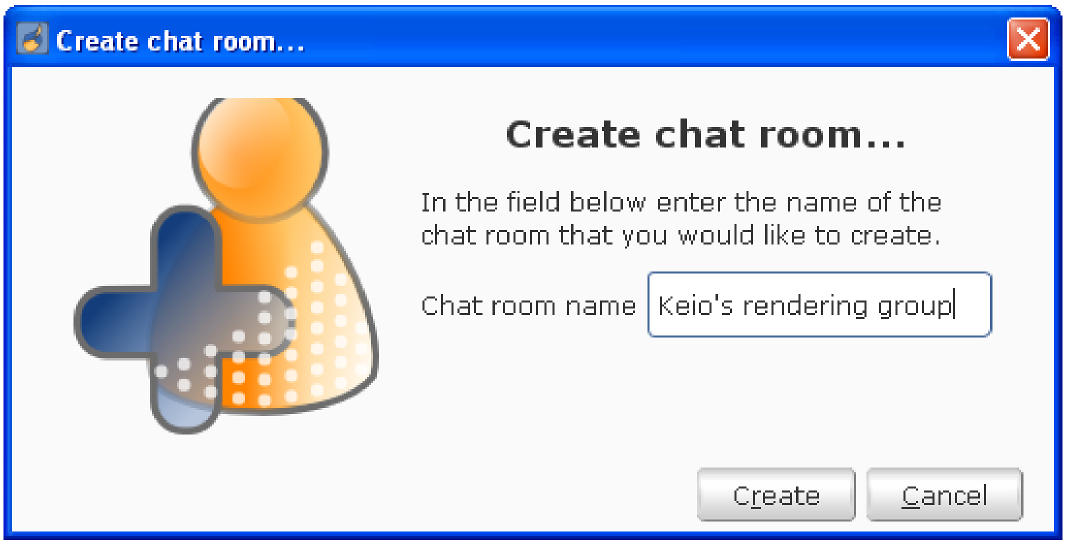


Figure 5.2: Screen-shot, SIP Communicator contact list (also used as a source from where to drag contacts and drop to a chat group)



Create chat room...

In the field below enter the name of the chat room that you would like to create.

Chat room name: Keio's rendering group

Figure 5.3: Screen-shot, create an F2F chat room

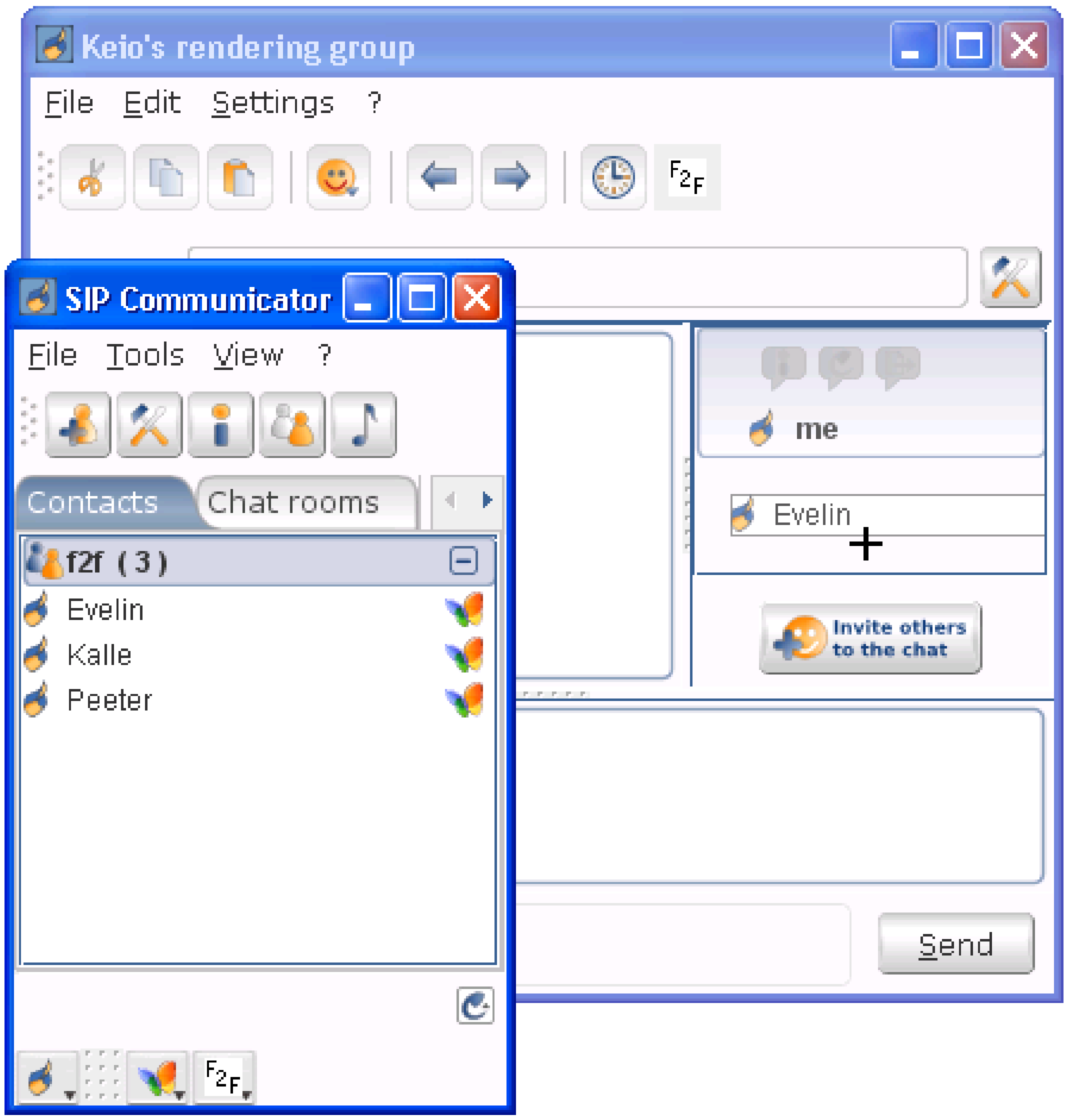
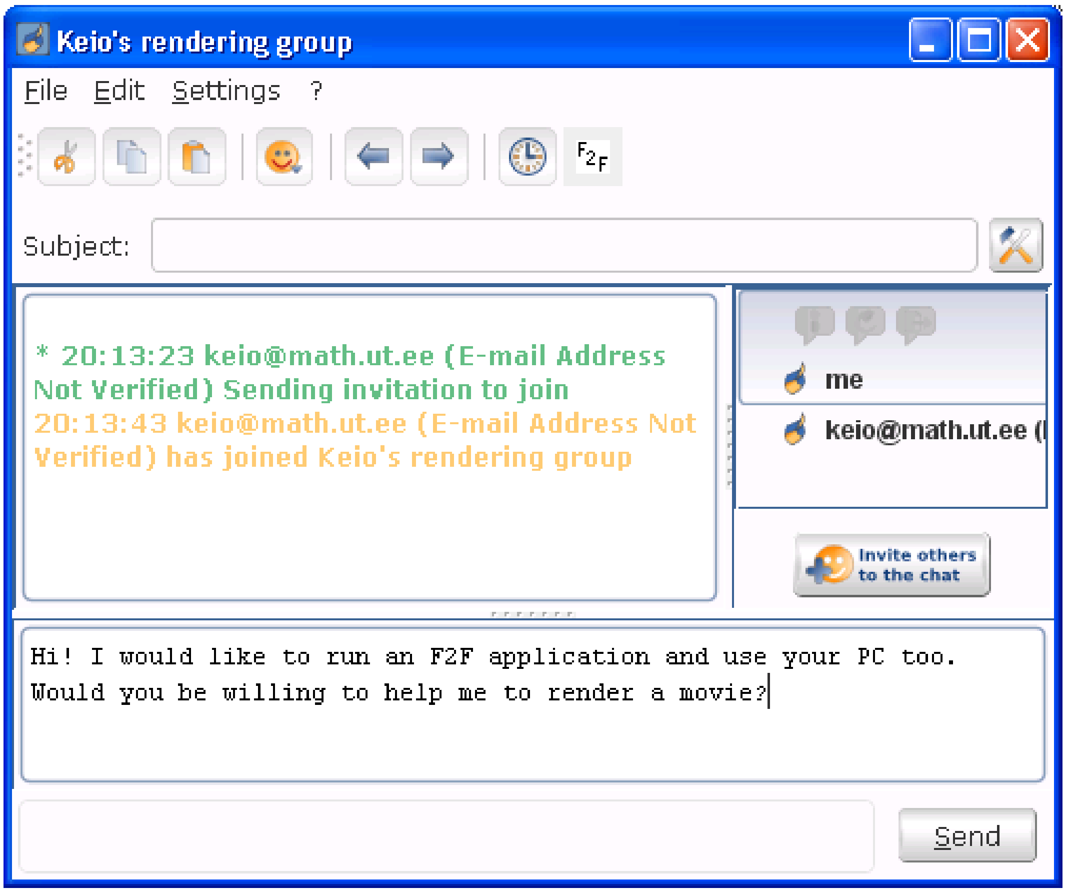


Figure 5.4: Screen-shot, Keio drags Evelin to the group chat



* 20:13:23 keio@math.ut.ee (E-mail Address Not Verified) Sending invitation to join
20:13:43 keio@math.ut.ee (E-mail Address Not Verified) has joined Keio's rendering group

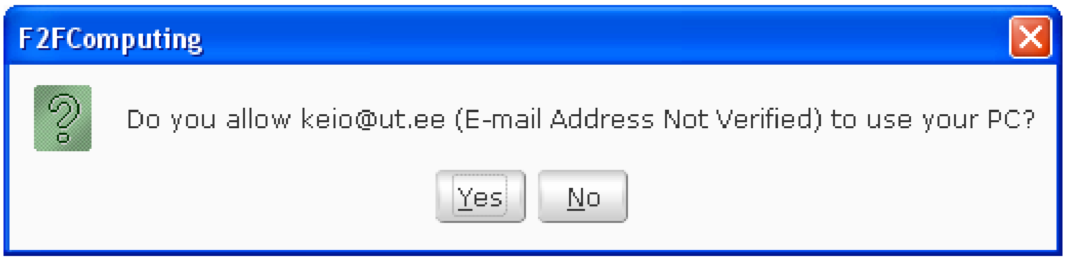
```
Hi! I would like to run an F2F application and use your PC too.
Would you be willing to help me to render a movie?
```

Figure 5.5: Screen-shot, two participants in Keio's rendering chat room

to run (see Figure 5.5). The next step for Keio is to click on the F2F button in the chat room toolbar. This opens a dialog via which F2F applications can be started. This dialog is described in section 4.2.7. In the dialog, Keio browses for the jar-file(s) that are needed for the Distributed Blender program, specifies the name of the master task (this is usually specified in the manifest file and read from there automatically) and whether he wants a slave task to be run on his PC or not. When Keio has filled the dialog with appropriate values, he clicks the `Compute` button and the application is started. As an initial level of security, the framework asks each chat room participant if he/she allows Keio to run the application (see Figure 5.6). If someone does not allow it, the framework does not allow any tasks of the application to be sent to the corresponding PC.

5.3 Rendering movie results

This paragraph compares how much time it takes to render the same movie with 1 or 3 PCs using the Distributed Blender program described in section 5.1. Lets name the computers M1, M2 and M3. The blend-file I used (taken from http://wiki.blender.org/index.php/Image:BSoD-ItCA-final.blend) describes an animation which contains 120 frames and all the


F2FComputing

Do you allow keio@ut.ee (E-mail Address Not Verified) to use your PC?

Figure 5.6: Authorization to run a job.

frames are equally difficult to render (frame rendering time should be the same for all frames).

At first I rendered the movie separately on each machine. M1 finished the rendering in 310 seconds, M2 in 321 seconds and M3 in 309.

Then I rendered the movie using all three machines. M1 and M2 were located in the network of University of Tartu. M3 was in my home (good Internet connection). When I formed a frid with the computers (using three MSN accounts) the creation of TCP connection between them succeeded. This meant that the master task could distribute the slave tasks quickly and the slave tasks could send the rendered results to the master task fast. The network overhead was not significant. The rendering was finished in 112 seconds. As expected, the rendering on three computers took almost three times less time than it took on the slowest machine.

Chapter 6

Summary and outlook

This thesis presented a lightweight desktop Grid framework which enables to set up a computational friend-to-friend Grid environment with the help of using instant messaging systems. We call it the F2F Computing framework.

I created the first version of the framework, which is realized as a plug-in to the multi-protocol instant messenger SIP Communicator. The framework enables the creation of small desktop Grids with no administrative overhead. If you want to use the framework, you just have to install the version of SIP Communicator which includes the F2F Computing plug-in; no additional administrative effort is needed. The framework shows the potential of combining instant messaging with Grid techniques. Setting up Grids spontaneously in small social communities will also accelerate the turnaround for testing and migrating Grid applications on this platform and so enable improvement in the development of Grid applications. Furthermore, opening the Grids to the public will lead to Grid applications contributed by public communities and increase the possibility to re-use components of existing Grid applications to re-combine them into new applications.

The described framework has its strength in the simplicity for set-up and use in small communities consisting of close friends who know each other in person. The framework allows you to use the power of your friends' PCs. To do this, you just have to create a group chat, invite your friends to the chat (your and their PCs will form an F2F Grid (*frid*)), and start the desired F2F application which will run on the formed frid. However, it is possible that your closest friends have in turn their friends on their list whom you do not need to know but who would be willing to run an application of a friend of their friend. Enabling such socially more loose but bigger communities to be combined into next generation frids is one feature that should be considered when developing the next version of the F2F Computing framework. To widen the range of the F2F Computing even more, it would be useful to supply adapters to different other Grid middlewares to be able to participate in F2F computations through some real user having credentials to do so. For example, your friend who has access to a Globus Toolkit based Grid system might be able to use his right to submit there an F2F computational task. However, this may be also a local computer cluster, to which one could easily add the F2F pool in such way.

The framework provides a simple but still powerful API to support the development of distributed applications that can be run on frids. The API enables computing nodes to interact via sending and receiving messages, opening a door for creation of not only embarrassingly parallel distributed applications. Additionally, the API supports the MPI standard, which has become the *de facto* standard for programming distributed computationally and communicationally intensive applications. This enables the developers who are MPI specialists to easily create new F2F applications. Also, it supports the porting of existing MPI applications to the F2F Computing framework. An instant messaging protocol is used for starting up the connection between Grid nodes, but the communication channels are built up thereafter independently using the fastest available protocol, with the option to use even NAT traversal methods.

The security and configuration issues have not been covered in my work, therefore next version(s) of F2F Computing framework should handle them, otherwise the F2F Computing will not be accepted by the wider public (see section 2.3). The owners of PCs should have precise control over their resources at any moment. Java may be helpful in providing sandbox environment by limiting access to certain resources, or even our own virtual machine might be developed. The trust between friends may be something that can be used while dealing with the security issues. Also, in the future the framework might deal with the dynamic and heterogeneous nature of Grids. For instance, it would be nice to be able to submit an F2F application and order the framework to use the power of your friends' computers, and the framework dynamically assigns each task to the most suitable computer, automatically reassigns tasks which were running on PCs that have left the Grid, and even uses the power of these friends' PCs who come online later.

To prove that the framework can actually be used and to ease the first steps in doing it, I described how to start F2F applications, presented an example F2F application for movie rendering, and showed that a movie can be rendered with a significant speedup.

This thesis shows the potential of combining instant messaging, Grid, and P2P techniques. I believe that the introduced approach of uniting the computational power of social communities has a great future. Improving the framework with the primary focus on security and configurability and creating some popular F2F applications will widen the circle of Grid users by ordinary people without any IT background. Thus making Grid computing popular for the public.

The F2F Computing framework is open-source and available under an LGPL license [8]. You can monitor it and contribute to it at the Friend-to-Friend Computing homepage [7].

Sõprusraalimine

Magistritöö (20 AP)
Keio Kraaner
Resümee

Hetkel on võrkraalimiseks (*Grid computing*) kasutatavad süsteemid väga keerulised ja raskekaalulised. Ainuüksi nende töökorras hoidmine nõuab suurt administratiivset tööd, rääkimata installeerimisest. Seetõttu ei ole võrkraalimissüsteemid laiemalt levinud. Samas on väga populaarsed ja laialdaselt kasutatavad kiirsuhtlust (*instant messaging*) võimaldavad süsteemid nagu MSN Messenger ja Skype.

Minu magistritöö eesmärgiks oli ühendada võrkraalimiseks ja kiirsuhtuseks kasutatavaid tehnikaid ning luua nende baasil uus võrkraalimise raamistik, mida oleks kerge hallata ja mis võimaldaks kiirsuhtluse kasutajatel hõlpsalt oma arvutite võimsust ühendada ja mingit arvutuslikku probleemi kiiremini lahendada. See lähenemine sai nimetatud sõprusraalimiseks ja loodava raamistiku nimeks sai F2F Computing.

Koostöös Ulrich Norbisrathi, Eero Vainikko ja Oleg Batraševiga kirjutasin sõprusraalimise ja F2F Computing raamistiku peamisi põhimõtteid kajastava artikli [24], mis võeti selle aasta juunis Ateenas toimuva konverentsi *The Third International Conference on Internet and Web Applications and Services* (ICIW 2008) kavasse.

F2F Computing raamistiku arendamise baasiks sai valitud SIP Communicator, mis on Java programmeerimiskeeles arendatav avatud lähtekoodiga modulaarse arhitektuuriga kiirsuhtlusprogramm. Realiseerisin raamistiku selle suhtlusprogrammi pistikprogrammina. Seetõttu on raamistiku installeerimine sama lihtne kui SIP Communicator'i installeerimine.

Tegin F2F Computing raamistiku selliselt, et see lisab SIP Communicator'i kasutajatele võimaluse luua erilisi jututubasid, kuhu saab kutsuda sõpru kasutatavast suhtlusprotokollist (MSN, ICQ jne) sõltumata, nendega seal suhelda ja käivitada raamistikuga ühilduvaid rakendusprogramme, mis siis püüavad jutuoaga seotud inimeste arvutite võimsust ära kasutada. Loomulikult peavad tuttavad oma arvuti kasutamist lubama.

Selleks, et F2F Computing raamistikuga ühilduvate rakenduste arendamine oleks võimalikult lihtne, sisaldab raamistik rakendusliidest (*Application Programming Interface*). Selle abil saab ka olemasolevaid programme raamistikuga ühilduvaks kohaldada. Tähelepanuväärne on see, et rakendusliides võimaldab hajusprogrammi osadel omavahel sõnumeid vahetades suhelda (enamus võrkraalimissüsteeme seda ei võimalda). Rakendusliides toetab ka MPI (*Message Passing Interface*) standardit, mis on laialdaselt kasutatav arvutuslikult ja infovahetuslikult intensiivsete hajusrakenduste programmeerimiseks.

Tutvustasin oma töös F2F Computing raamistikku. Kirjeldasin selle tähtsamaid osi, töö- ja kasutuspõhimõtteid ning -võimalusi. Näitasin kiirsuhtluseks ja võrkraalimiseks kasutatavate tehnikate ühendamise potentsiaalsust. Ma usun, et see lähenemine omab suurt tulevikku. Kui F2F Computing raamistikku parendatakse (eeskätt turvalisust ja konfigureeritavust silmas pidades) ning tehakse mõned populaarsed raamistikuga ühilduvad rakendusprogrammid, on vägagi tõenäoline, et võrkraalimist kasutavate inimeste hulk laieneb ja jõuab sõprusraalimise kaudu ka nendeni, kellel puudub infotehnoloogiaga lähem side.

Appendix A

Source code

The source code related to the F2F Computing framework is being held in an SVN repository that is maintained by Google. The home page address of the F2F repository is http://spontaneous-desktop-grid.googlecode.com/. There you will find up to date information about the framework, the source code, and how to start using it. Also, the latest release should be available.

Bibliography

[1] Alchemi. http://www.alchemi.net/.

[2] Apache Felix. http://felix.apache.org/.

[3] Blender. http://www.blender.org/.

[4] Comparison of instant messaging clients on wikipedia. http://en.wikipedia.org/wiki/Comparison_of_instant_messaging_clients.

[5] Einstein@home. http://einstein.phys.uwm.edu/.

[6] Elephant's dream. http://www.elephantsdream.org/.

[7] Friend-to-friend (F2F) computing. http://f2f.ulno.net/.

[8] GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html.

[9] JNGI. https://jxta-jngi.dev.java.net/.

[10] xeerkat - google code. http://code.google.com/p/xeerkat/.

[11] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 4–10, Nov. 2004.

[12] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45:56–61, 2002.

[13] F. Berman, G. Fox, and A. J. G. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley and Sons, 2003.

[14] I. Foster. Globus: a metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications*, 11:115–128, June 1997.

[15] L. Gong. Jxta: a network programming environment. *IEEE Internet Computing*, 5:88–95, 2001.

[16] Grace, Coulson, Blair, and Porter. Deep middleware for the divergent grid, 2005. http://dx.doi.org/10.1007/11587552_17.

[17] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, 1994.

[18] V. Huber. Unicore: A grid computing environment for distributed and parallel computing. pages 258–265. Springer-Verlag, 2001.

[19] E. Ivov and sip-communicator community. Sip communicator. http://www.sip-communicator.org/.

[20] S. Kleiman, D. Shah, and B. Smaalders. *Programming with threads*, chapter 16. SunSoft Press, Mountain View, CA, USA, 1996.

[21] A. Lind. NAT Traversal in P2P systems in Java, 2008, University of Tartu.

[22] A. Luther, R. Buyya, R. Ranjan, and S. Venugopal. *High Performance Computing: Paradigm and Infrastructure*, chapter Peer-to-Peer Grid Computing and a .NET-Based Alchemi Framework, pages 403–429. Wiley Press, 2005.

[23] A. Luuk. Porting MPI applications to the F2F Computing framework, 2008, University of Tartu.

[24] U. Norbisrath, K. Kraaner, E. Vainikko, and O. Batrasev. Friend-to-Friend Computing - Instant Messaging Based Spontaneous Desktop Grid. In *The Third International Conference on Internet and Web Applications and Services (ICIW 2008)*, 2008.

[25] Osgi-Alliance. *Osgi Service Platform, Release 3.* IOS Press, Inc, 2003.

[26] B. C. Popescu, B. Crispo, and A. S. Tanenbaum. Safe and private data sharing with turtle: Friends team-up and beat the system. *Proc. of the 12th Cambridge Intl. Workshop on Security Protocols*, 2004.

[27] J. Rosenberg. draft-ietf-mmusic-ice - interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols. http://tools.ietf.org/html/draft-ietf-mmusic-ice.

[28] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency and Computation: Practice and Experience*, 17:323–356, 2005.

[29] J. Verbeke, N. Nadgir, G. Ruetsch, and I. Sharapov. Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment. In *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, pages 1–12, London, UK, 2002. Springer-Verlag.

[30] B. Vinter. The grid taken literally. In *The 8th Hellenic European Research on Computer Mathematics & its Applications*, Athens - Greece, Sept. 2007.