University of Warsaw
Faculty of Mathematics, Informatics and Mechanics
VU University Amsterdam
Faculty of Sciences

Tomasz Dobek
Student id. no.: 209515 (WU), 1735225 (VU)

Efficient and Flexible Gossip Based Peer-To-Peer Network in ProActive

Master's Thesis
in COMPUTER SCIENCE
in the field of DISTRIBUTED SYSTEMS

Supervisors:
Thilo Kielmann and Kees van Reeuwijk
Dept. of Computer Science,
VU University Amsterdam

Fabrice Huet
I3S and INRIA Sophia Antipolis
University of Nice-Sophia Antipolis

Janina Mincer-Daszkiewicz
Institute of Informatics
University of Warsaw

August 2008

Abstract

ProActive is a middleware for developing distributed applications. It provides a unique programming model based on the active object concept as well as a deployment framework for running the applications. One part of the project is a peer-to-peer network in which participants can donate computational resources creating a dynamic deployment environment for distributed applications. The existing peer-to-peer solution is based on the protocol similar to Gnutella where queries for resources are forwarded to all of the acquaintances and thus number of messages can grow exponentially.

This thesis proposes a new protocol for the peer-to-peer network that uses a gossiping algorithm. It addresses the main issue that the existing implementation suffers from - the unstable bandwidth load. The solution allows to model the network utilization by the peer by specifying the minimum and maximum message size, the number of peers to send the gossip and the frequency of the gossip. In effect we can achieve a compromise between the usage of the bandwidth and speed of dissemination of information in the network. The thesis includes results from experiments where the performance of the peer-to-peer network is assessed in different constraints scenarios.

Keywords

Peer to Peer, Computations, ProActive, Commodity Hardware, Gossiping

Thesis Domain (Socrates-Erasmus subject area codes)

11.3 Computer Science

Subject classification

C. Computer Systems Organization
C.2. Computer - Communication Networks
C.2.4. Distributed Systems

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Background

Nowadays the most demanding computations are conducted in parallel. This fact can be explained by the increasing difficulty of improving performance of existing processing units. Furthermore, many tasks can be easily parallelized. So far, most of the parallel applications are deployed to two different classes of environments: high performance environments consisting of cluster of machines or even grids of clusters and massive amounts of commodity hardware. The first type of environment is dedicated to performing demanding computations. Thus, it has better communication networks and possibly more powerful hardware. Generally, it is easier to deploy applications to such machines as they were designed to serve this purpose. On the other hand, this type of environment had to be separately bought and this may imply considerable costs to the company or research institution that wants to take advantage of such solution. Massive amounts of commodity hardware are on the other hand easier to maintain but harder to coordinate. They are easier to maintain because the price of each machine is very low and often it is more convenient to replace the failing machine than to repair it (examples of such approach: [15], [10]). However, it is difficult to coordinate commodity machines because they are usually used in much bigger numbers than in cluster environments. What is more, the quality of the commodity machines' hardware is worse than the quality of those inside dedicated environments so the failures happen more often. Thus, the designers of distributed applications or the middleware for performing parallel computations must think about fault tolerance issues. The advantage of the second environment is its cost as it can sometimes be near to zero. For example, we could use existing desktop machines in the company to perform computations during nights and weekends.

The distributed computations can be done in a structured or unstructured way as far as the resource allocation is concerned. The first approach implies the presence of some coordinator which decides on the machine allocation. Meta-schedulers are an example of such coordinator. This approach can experience scalability issues when the environment size increases as it happens in centralized solutions. In the unstructured approach, there is a dynamic network of machines without any global coordinator. The machines constitute a so-called peer-to-peer computational network in which the resource they share are not files but the ability to perform computations. This work deals with such network which is suitable for deploying distributed application to the commodity hardware.

The ProActive library is a middleware for the execution of distributed applications. It supports many methods of starting distributed applications providing both structured and unstructured way of allocating the resources. It has the component that uses the unstructured approach which is a peer-to-peer computational network. This component uses the protocol similar to the one that is implemented in the popular peer-to-peer project called Gnutella. Because of that it has scalability issues which are shared by all of the Gnutella-like projects. The problem is caused by the communication mechanism that relies on flooding peers with messages in order to query for resources. When we introduce more peers to the network, this flooding traffic saturates the communication channel. This work proposes a different protocol that the peer-to-peer computational network can use which addresses these issues.

1.2. Contribution of the Thesis

This work contains a description of a new protocol for the peer-to-peer network in ProActive which I designed and made a prototype implementation. This protocol uses ideas from gossiping algorithms in order to perform communication between peers. Because of this approach it is possible to have full control over the internal traffic that the network generates. The user can specify lower and upper limit of the message size. He or she has control also over the frequency of gossiping and number of peers to send gossip to. The network adjusts to the constraints given by the user and is propagating as much knowledge as it can within the given limit possibly delaying the dissemination of some items when there is no room to do this immediately. The experiments conducted with the prototype implementation confirmed this behaviour and proved that the network is using the allocated resources efficiently. The solution is also flexible because it can be used in situations where the network bandwidth is abundant and the speed of dissemination allows us to use more fine grained work distribution.

The rest of this thesis is organized as follows: in Chapter 2 the model of the system in which the proposed solution is contained is described. It starts by presenting the ProActive library and explaining basic concepts that will be used throughout the rest of the thesis. The second part defines the additional assumptions that one has to make when dealing with the peer-to-peer network in ProActive. Chapter 3 contains the description of the existing implementation of the peer-to-peer network in the ProActive library. In Chapter 4 the gossiping based protocol for peer-to-peer network in ProActive which I designed is described. Chapter 5 presents the reference implementation of the concept which turned out to be a general implementation which can be easily customized to test behaviour of different strategies in the generic gossiping algorithm. Chapter 6 contains the results from the experiments using this new approach. Chapter 7 concludes the work and highlights directions for the future improvements. The Appendix covers the ProActive Agent, a Windows system service that makes the deployment of the peer-to-peer network to desktop machines an easy task.

1.3. Related Work

This section presents different projects which aim at allowing to perform distributed computations on massive amounts of desktop machines. Each of the described project realizes this goal in a quite different way and most importantly have different objectives.

1.3.1. BOINC

BOINC stands for Berkeley Open Infrastructure for Network Computing. It is a platform that supports diverse applications to use volunteered computing cycles. The major projects that make use of this framework include Seti@Home, GIMPS (Great Internet Mersenne Prime Search), distributed.net and Folding@home. The software can be also used locally in company networks to form desktop grids. [1]

The BOINC project uses client-server approach in order to manage work. The client software is installed on desktop machines and receives little portions of work from the server. After it finishes work it sends the reply to the server and gets another portion. The server then can use the partial results to compose the whole solution to the problem.

BOINC can be easily integrated with existing applications written in C, C++ or Fortran. Volunteers, in addition to downloading the BOINC distribution, have to install the library that contains the implementation of the project they will be supporting. Participation in projects is set individually on the client side through the settings of the account. The service providers rely on the fact that clients will donate their resources to their project. That is why the name - volunteer computing.

The BOINC project removes from application developers the concern of fault tolerance as it is trying to address this issue by itself. The BOINC scheduler component is responsible for work distribution. This component adds redundancy so every task item will be distributed to more than one machine. When the results from each task copy are obtained they won't be taken into account unless they are all the same [18]. For large work items the BOINC uses the policy called local scheduling in which the scheduler tries to send items to hosts which already have some of the required input files [2].

To sum up, BOINC is a centralized approach for performing massively parallel computations. It tries to handle most of the burden that comes from hiding the distribution character of an application. Because of the centralized nature it is assumed that each unit of work can be done independently.

1.3.2. JXTA

JXTA [19] is a general purpose peer-to-peer network that supports collaboration of a rich variety of devices starting from supercomputers and ending in cellular phones. JXTA actually forms a standard and numerous implementations are available. The standard doesn't limit or make it difficult to implement the specification using different programming languages. JXTA supports developing distributed applications that are using the services exposed by the peers collaborating in the JXTA network.

The JXTA framework is logically divided into 3 layers. The first layer is the platform. It contains the core components that are used to maintain and communicate inside the peer-to-peer network. The second layer is called services and implements various functionalities that a peer can provide or can have access to using JXTA network. This layer uses the platform layer to perform necessary communication inside the peer-to-peer network. The last layer is an application layer in which the application developer creates new software that uses the resources and services from JXTA network using the services layer.

The most important feature of JXTA network is that it supports its own routing algorithm and allows for multi-hop communication between peers. The routing algorithm is necessary because each peer can have access to multiple and different types of networks and we would like to enable collaboration between all of the peers. The multi-hop communication is also necessary when two peers cannot communicate directly. It can happen for example in the situation where two peers are behind a NAT. Another example: a cellular phone connected by Bluetooth cannot talk directly to the PCs that are connected by TCP. These virtual connections between peers are called Pipes.

A Peer uses a notion of advertisements to spread information about the services it provides. This information can describe what categories does a given peer belong to. This knowledge is scattered across peers. Every peer has a limited amount of global state. Specifically it contains information about what is available in its local area network unless it belongs to a special category of peers.

There are several categories of peers:

- A rendezvous peer is a peer that contains information about the advertisements and is used to perform queries for resources. This kind of peer stores information about advertisements beyond its local area network peers. A rendezvous peer doesn't aim at collecting as much information as possible. It can forward the search to the other rendezvous peers if it is appropriate. This kind of peer uses more memory and bandwidth as it has to answer queries and cache the results of queries.
- A router peer is a peer that can generate routes needed for two peers to communicate with each other. The router peer works in the overlay network layer and will try to address issues such as connecting peers that are using different network protocols or use the same protocol but they can't reach each other directly.
- A gateway peer is a peer used as a relay when the Pipe (virtual connection between two peers) requires additional hops. It will be forwarding communication to allow for example two firewalled peers to see each other.

A peer can belong to zero or more of these categories. A peer that doesn't belong to any of these is called an ordinary peer.

All in all, JXTA provides sophisticated means for writing applications that are using underlying peer-to-peer resources. It provides a true peer-to-peer approach making possible the collaboration of many kinds of devices. It has many properties that makes it very convenient to use in business applications such as overcoming connectivity limitations, the ability to form groups of peers and independence from transport layer and the language of implementation. It is a general purpose framework that can be used in arbitrary way by user applications.

Chapter 2

System Model

2.1. Overview

This chapter describes the environment in which the solution is contained. The environment consists of the middleware that the proposed solution will be using. That middleware imposes some model of computation that our implementation will have to use. In our case the middleware is the ProActive library. This package allows us to easily create distributed applications. The model of such an application is based on the concept called an active object which our solution will exploit. The essential description of active objects and presentation of model of the computation that ProActive provides us with will be presented in this chapter. It will be needed to understand the rest of the thesis.

The peer-to-peer network in ProActive is a component of the ProActive library that supports donating computational resources of the machine by contributing in the unstructured network of machines without a single point of failure. The existing implementation of the network is described in Chapter 3. In this chapter, additional assumptions are outlined that one must make before using that component.

The rest of the chapter is organized as follows. First, the concept of an active object is explained. Second, there is an overview of the communication mechanism provided by ProActive along with description of synchronization mechanisms. Third, the concepts that are used to describe the running ProActive application are presented. These include containers for active objects called *nodes* and the containers for *nodes* called *virtual nodes.* The purpose of these objects is explained. The last part of the chapter presents the additional assumptions that the implementation of the peer-to-peer network in ProActive makes about the model of computation.

2.2. Active Object

The active object is a design pattern that can be used in object oriented software that deals with concurrency. Its role boils down to separating method invocation from method execution on the possibly remote party and by doing this simplifying synchronized access to the object. The concept of active object is thoroughly explained in [17].

Every active object has its own thread of control. This thread is responsible for serving incoming requests. When other objects are calling the active object method their threads aren't involved in serving the request. Instead, their requests are recorded in the active object's queue and are served sequentially by the active object's thread. By doing this, the method invocation is decoupled from the method execution. These two activities are done in different time by different threads.

Each active object has the following components:

- A proxy/stub used on a client to access the active object. This element is responsible for calling the actual active object and putting the method invocation request to its queue.
- A method invocation request is the information about the method to be called on the active object side. Depending on the implementation it can be a separate data structure describing parameters of a method call or list of serialized objects possibly in different formats.
- An activation queue is the queue that stores the method requests coming from clients of an active object.
- A scheduler is the component that decides on the order of serving requests. Depending on the implementation of the pattern it could analyze the internal state of the active object to make the decision or rely on some metric that describes the order of serving requests.
- A servant is the service part of the active object. It contains the internal state of the active object and methods that are exposed to the clients. This is the part in which the actual functionality of the active object is implemented.
- A future object is the object returned to the client immediately after calling the active object. It contains (or will contain) the result of the call. Because the method call is decoupled from the method invocation, some time must pass before the client receives the result of the request sent earlier to the active object. During that waiting time the client's thread of control is not interrupted - after sending a request to the active object it can immediately continue execution. However, the returned future object may not contain the result of the method call at that time. As soon as the active object processed the request sent by the client, its future object is updated with the result of the call.

When a client's object calls a method of the active object the following happens:

- The client calls the method of the stub of the active object.
- The stub communicates with the scheduler of the active object and handles the method invocation request
- The scheduler enqueues the request in the activation queue. At this time the client is returned a future object which will be updated with the result of the invocation as soon as it is completed. A client's thread continues to execute next parts of the client's code.
- The scheduler decides which method invocation request should be served.
- The scheduler removes the selected request from the queue and calls the service method which handles that request.

- The scheduler receives the result from the service method and updates the client's future object with the result. From now on, the client has the access to the result of its request.

It is important to be aware that the actual semantic of the active object hugely relies on the implementation of the pattern which might be also influenced by the programming language that is chosen to achieve this goal. For example, the Java programming language makes it easily possible to dynamically generate stubs and active objects from any object requiring from the programmer only to develop the service part of the object. This approach is used in the ProActive framework described in the next section. In this case, the framework allows to 'activate' (i.e. to create the components of the active object) any object existing in the application. This approach has some limitations which can change the semantics of synchronization during method invocations for some specific classes of activated objects.

2.3. ProActive Library

ProActive [3] is an open source middleware dedicated to executing parallel programs on multi-core processors, local area networks, clusters and data centers, on intranet and Internet grids. All of the components of ProActive are written in Java. The framework itself relies only on a compliant implementation of the Java Virtual Machine. This means that it doesn't require changes to the JVM nor preprocessing nor compiler modification. ProActive normally uses the RMI Java standard library as portable transport layer. However, this is not a limitation as other means of transport can be used as well (for example the Ibis communication library [16]).

2.3.1. Active Object

A central concept which the ProActive framework is built upon is an active object pattern[9]. From now on, the active object term will refer to the implementation of the active object pattern in the ProActive library written in the Java language. Any properties of active objects described later in this and next chapters apply to this specific implementation.

An active object is a standard Java object (I will refer to them as passive objects) activated by the ProActive framework. Each active object gains in this process location transparency, activity transparency and synchronization control. An active object has one entry point for all calls made to that object where it can decide on the order of servicing requests. The lifetime of an active object has three phases: activation of the object, performing activities, and deactivation.

In the activation phase the object is activated. A new thread that is attached to that object is started and is used to serve requests. After that stage the active object is running its own activities - the thread is answering requests and if necessary calling other active objects. Synchronization and communication mechanisms will be explained in later sections. When an exceptional situation happens; for instance, an uncaught runtime exception is thrown during thread execution an active object ends its activity. The thread is stopped and it no longer can serve requests.

All of the features that enhance a passive object when it is activated are stored in a component called *the body.* Each active object has its own unique body. The *body* of the object contains the request queue, the thread etc.

Each active object during its lifetime is assigned to one entity called a *node.* A *node* is a container for active objects and this term will be thoroughly explained in Section 2.3.6 of this chapter.

2.3.2. Initialization of the Active Object

A passive object (i.e. not active) can be activated in three distinct ways:

- Class-based approach

  In this case, the user creates a subclass of the class which instances he wants to activate. The subclass has to implement the Active interface. The Active interface is an empty interface that informs the ProActive runtime that a given instance is able to become an active object. In addition to this, the implementation of the subclass can specify some mechanisms that describe the way the active object behaves. For example, by default the service policy serves incoming requests in FIFO based manner. It can be changed in the implementation of the subclass so as to address the user's needs. The section about service policies describes this process in further details. Other examples include specifying initialization and destroy actions taken by the object.

  Example code that creates the active object using described mechanism:

```
public class SomeClassActive extends SomeClass implements Active  { }
Object[] params = new Object[] {"foo", "bar", new Long(12345L)};
SomeClassActive act = (SomeClassActive) PAActiveObject.newActive
                      (SomeClassActive.class.getName(), params, node);
```

- Instantiation-based approach

  This method of creating active objects doesn't require creating subclass as in previous case. Instead, this task is done by ProActive runtime which implicitly creates a proxy subclass during the activation of the passive object. In effect, the developer is relieved of writing any additional code that is needed for the application to be integrated with ProActive middleware.

```
Object[] params = new Object[] {"foo", "bar", new Long(12345L)};
SomeClass act = (SomeClass) PAActiveObject.newActive(
                       SomeClassActive.class.getName(), params, node);
```

- Object-based approach

Both of the so far described ways of creating active objects assumed that this activity is done upon creation of the class instance. The object-based approach removes this requirement by providing a way to transform an existing passive object into an active one.

As far as object-based approach is concerned we deal with a local object we want to activate. The destination of the active object can be a local or remote *node*. The activation algorithm distinguishes these two cases. In the case of local *node* a new proxy is created by ProActive runtime. Different things happen when we want to create an active object remotely. Because we should be able to deal with the local passive object in later time we have to keep it locally so its copy is sent to the remote JVM that the destination *node* belongs to. On that JVM, we keep a copy of the passive object along with the active subclass instance created by ProActive.

```
SomeClass ex = new SomeClass("foo", "bar", 12345L);
ex = (SomeClass) PAActiveObject.turnActive(ex, someNode);
```

2.3.3. Communication

Active objects communicate by means of (possibly remote) method invocations. In the previous section I described methods of creating active objects. In all ways of accomplishing this task the important point is that the active object is a subclass of its passive predecessor. In that subclass the ProActive framework intercepts method invocations allowing different ways of serving requests to be employed.

The method invocations on active objects have different semantics depending on what type of method we want to invoke. In simple words, ProActive framework makes the best effort to provide an asynchronous method call mechanism. The method returns an object called a 'future object'. The user can continue his or hers computations as long as he or she doesn't use that object. When the object is used the computations are stopped until the actual result is received from the original call. However, this solution has some limitations. Before we move on to description of communication protocols used by ProActive let us define an important concepts called reifiable type and object.

A reifiable type is a type which can be stubbed by ProActive to give an immediate result from the method call. The stub is a subclass of the original result class and is hiding the fact that the actual result has not arrived yet. Therefore, a reifiable type has to be a non-final class. Thus, non-reifiable types are final classes and primitive types.

A reifiable object is an object which type is reifiable.

The following communications schemes are provided by ProActive active objects:

- One-way method call

  Only communication from caller to the callee is conducted. That happens when the method doesn't have any return value (i.e. in Java language its return type is void). The protocol can be described as follows:

(a) At the beginning there is a rendez-vous between two objects involved. This rendez-vous has low latency and is performed only to ensure that the method call reaches the callee. In our situation it means that the request is put into the queue on the remote side. As soon as this is satisfied, the rendez-vous ends. ProActive uses by default the RMI protocol for transport layer. Because RMI is a reliable protocol, our communication between those two entities is also reliable.

- Bidirectional synchronous method call

  This type of communication is used when a method returns a non-reifiable object or returns a reifiable object but may throw a checked exception. In this situation we cannot use future objects because either they do not support non-reifiable objects due to Java language limitations or if the exception is thrown we might not intercept it in try/catch block if we use an asynchronous method call. The protocol is very similar to the previous one with the differences outlined below:

  * A future object is not created on the caller side
  * The caller waits until step (d) described in the previous paragraph is finished. This ensures that an exception can be caught inside try/catch block.

- Bidirectional asynchronous method call

  If the method returns an reifiable object and is not throwing any checked exceptions then this scheme is employed. The communication between caller and callee is subject to the following steps:

  (a) The same as (a) in the paragraph above

  (b) While the caller blocks to be notified by the callee on receipt of the call, the future object is created on the caller side. A future object is a subclassed wrapping proxy on the return value from the method invocation. At this time it doesn't contain any proper reply. However, as soon as the reply from the callee reaches the caller it will be updated. There is a synchronization mechanism that is implied in the future objects. It is called wait-by-necessity and will be explained in greater detail in section 2.3.4.

  (c) The callee executes the invocation stored in its body queue. It is up to the callee to decide in which order it wants to serve requests. More on that will be covered in section 2.3.5

  (d) When the call is finished the body of the callee updates the future object with the result of execution

  (e) Caller can use the result from the future object

These are general concepts describing the communication between active objects. It should be emphasized that ProActive also uses a couple of optimizations to ensure a decent performance and to reduce communication overhead. For example, if two active objects reside inside the same virtual machine, those two object would communicate directly without using the RMI stack.

2.3.4. Synchronization

Each active object has its own thread of control. It is used to perform activities of the object or to wait and serve the incoming requests. The synchronization mechanism that is applied when the method on active object is invoked depends on the communication style that is used. The communication variants were described in the previous section.

For one-way communication, there is no synchronization apart from the rendez-vous that ensures that the method invocation call is delivered to the callee's body queue.

For synchronous communication, the caller is stopped until the method execution is finished and the result delivered by the callee.
For asynchronous communication, the synchronization mechanism called wait-by-necessity is employed. This mechanism allows for execution of a caller's thread after the invocation was made and the result was not returned. However it will block the caller when a method on the future object from that invocation is called. This protects the thread from relying on the result from method call that has not returned result yet. As soon as the body of callee updates the future object with a result execution is resumed on the caller side. The following example should clarify the concept of wait-by-necessity:

```
SomeClass activeObject = PAActiveObject.newActive(...);
ResultClass res = activeObject.doSth();
// res at this point might or might not have a result
// next call will not be blocked because we use asynchronous communication
ResultClass res2 = activeObject.doSth();
// however the next call will block the current thread until the result
// (res) is available
res.doSthElse();
```

It is important to note that wait-by-necessity hides from the programmer a concern that comes with asynchronous programming - ensuring that the result of the call is received. It makes programming with ProActive more seamless and it requires less modifications to existing code in order to be integrated with ProActive framework. However, there are mechanisms in ProActive framework that a programmer can use to explicitly ensure that in a given point of thread execution the future object is supplied with invocation result. It is achieved by means of two static methods available in ProActive API:

* PAFuture.isAwaited(futureObject) - returns boolean value indicating whether the future object was updated with the result
* PAFuture.waitFor(futureObject) - blocks current thread until the future object is updated with the result

If the programmer is passing a not updated future object when calling a method on another active object, a copy of that future object is created in the queue of that callee. However, the thread execution of the caller is not stopped until the future object is updated. Another mechanism called automatic continuation is used to prevent that from happening. Automatic continuation guarantees that all of the copies of the future object will be updated when the original callee returns a value.

An automatic continuation is used in the aforementioned case when a not updated future object is used in another call to an active object. It can also occur when an active object returns a result containing a future object which has not been updated with a result. The active object on which side the copy of the future object is serialized remembers mapping from the future object to the body it was sending the copy. As soon as that active object is notified about a return value of the future object it was using, it will also use the stored mappings to notify every body that was sent a copy of that future object. In effect, every copy of that future object will be updated when the original future object is updated.

2.3.5. Control of the Activity

An active object has its own thread of control. By default, ProActive handles this thread execution by waiting and replying for incoming requests in default order. However, the developer can modify the behaviour of the active object's thread. ProActive gives three ways to achieve this:

- Initialization activities

  By implementing the InitActive interface, a developer can specify the initialization actions that will be undertaken when an active object is created. For example, this can include excluding methods that will fall under ProActive synchronization scheme. This will be used in the implementation of peer-to-peer gossiping protocol.

- RunActivity

  By implementing the RunActive interface we can modify the whole behaviour of the active object's thread. To implement this interface, we need to implement the runActivity method which will implement the main function of the thread. If we exit from this method then the life of the active object is over. Inside this method we can specify different mechanisms to handle requests or/and do our own activities. The exemplary policies for handling requests include but are not limited to:

  * serveOldest() - serves the oldest request in the queue
  * serveOldest(String methodName) - serves the oldest request calling methodName
  * serveOldestWithoutBlocking() - serves the oldest request without blocking
  * serveMostRecentFlush() - serves the most recent request and removes the others from the queue
  * serveOldestTimed(int t) - serves the oldest request but not older than t milliseconds

  By default, when a developer does not implement that interface, the messages are served in FIFO manner.

- EndActive

  By implementing the EndActive interface it is possible to specify the actions to take when the lifecycle of an active object ends. That can happen for example, when a Runtime exception is thrown during the course of execution of the active object's thread or the runActive method returns. In that case, we can provide the code that, for example, releases the resources we acquired.

2.3.6. Environment and Implementation

This section describes the environment in which every ProActive application runs. The environment consists of running JVMs, *nodes* and possibly *virtual nodes*. From the user's point of view the ProActive application is running locally and on remote machines (if the application uses remote active objects) inside JVMs. But we know that each distributed application that is relying on the ProActive library creates active objects. These objects can be located on the local machine or be accessed remotely. From the perspective of active objects they exist inside *nodes* and possibly inside *virtual nodes*. These two concepts are described in the following sections.

Nodes

Nodes are basic entities containing active objects. ProActive uses a notion of *nodes* to locate active objects. All of the created active objects must belong to some *node*. A *node* name typically consists of the URL that reveals the location of the machine where the ProActive library is running and the name, for instance: rmi://tomek.inria.fr/SomeNode. This address can be used to perform lookups on active objects for example as we will see in the peer-to-peer implementation. The use of *nodes* in the source code in general is cumbersome because it makes it easy to develop software with hardcoded locations of active objects. Software written like this is difficult to adjust to a new deployment environment. Therefore, we can take advantage of separation between deployment environment and application. It is achieved by means of deployment descriptors which are described in [6]. From the application developer's perspective, this separation is given by the concept of *virtual nodes*.

Virtual Nodes

A *virtual node* is an abstraction that is used by ProActive to group *nodes* into logical entities. This layer is very useful, because it can make the source code of the application fully independent from what environment will be used to deploy the application. Application developers usually want to use some but not specific *node* that satisfies some requirements. By acquiring *nodes* from *virtual nodes* instead of calling *nodes* directly they remove application dependencies on the running environment from the source code. In this situation *virtual nodes* can group *nodes* according to the satisfaction of some requirements. The actual mapping of *nodes* into *virtual nodes* can be done in the part of the source code separate from the application logic (or preferably it can be described in the deployment descriptor, [6]) and changes in the running environment will require to adjust the aforementioned mapping only.

2.4. ProActive Peer-To-Peer Computational Network

Peer-to-peer computational network in ProActive allows users to deploy their ProActive applications to *nodes* that were acquired using the peer-to-peer protocol. The acquisition is conducted by querying for *nodes* from the peer-to-peer network and then mapping them to the specified *virtual node*. Although at first this approach looks like a typical deployment process where *nodes* are created (for example on the cluster machine) and then, they are mapped to *virtual nodes* to be accessed by the application, there are some differences that have to be explained. This section will present both the advantages of using peer-to-peer solution for application developers and additional assumptions that has to be made in order to successfully use this kind of resource acquisition.

2.4.1. Advantages

ProActive peer-to-peer component forms an unstructured network of peers in which everyone donates its computational resources. The unstructured fashion of this network makes it almost impossible to experience failures that will prevent users from using it. This is because the peer-to-peer concept doesn't have any single point of failure. Of course, failures of individual peers are possible to happen but they won't have much impact on the condition of the whole network.

Moreover, the huge amounts of peers that can be accessed by the network provide the environment in which computations can be performed for a long time. In the typical grid environment when we start computing some task we are allocated only the limited amount of time after which we can't use the resources anymore. In this situation however, peers voluntarily donate their resources which could be used for a long time (for example desktop machines outside business hours). Furthermore, when we are using the ProActive peer-to-peer network we don't have any time limits for using the network so the computations can last as long as the application developer wants.

Finally, the configuration of the deployment environment is very simple when using the ProActive peer-to-peer network. In grid environments specifying the machines our application will be using and access methods to them can be very cumbersome. One solution for this is to use meta-schedulers however lack of standard tools to achieve this goal makes it not a scalable solution. The peer-to-peer network gives uniform interface for querying for the resources. In order to be able to use massive amounts of *nodes* coming from different environments the application developer has only to specify how to connect to the peer-to-peer network. Then, the peer-to-peer protocol implementation will search and find the requested resources.

2.4.2. Additional Assumptions

Using the ProActive peer-to-peer network puts some additional assumptions on the applications. First, a user doesn't know where the resources that were requested will come from. Usually this is not the case in the grid environment. For example, if we specify a remote JVM for the deployment we know exactly on which machine it will be run. If we request *nodes* from a cluster we may not know the exact location of the machine that will be used before the application is run, but we know its configuration and capabilities. In the peer-to-peer computational network developed for ProActive we ask only for some number of *nodes* and generally the user has no knowledge about where will they come from and what will be their capabilities. There is a number of works where this issue is tried to be addressed. One example is [11] where the peer-to-peer network in ProActive can be supplied with additional information about the capacity of each peer.

Second, the peer-to-peer network makes the best effort to provide the requested number of *nodes*. However, it is not guaranteed that the asked number will be finally given to the user. There may be many reasons for not fulfilling the demand. For example, there are not enough free *nodes* in the network. Other culprit could be failures of peers or network problems. When we use the peer-to-peer computational network, we cannot assume anything about the transportation network as opposed to the other sources of *nodes* where we can at least expect some quality of service. All in all, an application that is to be deployed to the peer-to-peer network should be able to work with an incomplete number of *nodes*.

Last but not least, when the *nodes* are given to the user from the peer-to-peer network, it is not guaranteed that they will be able to be used for any arbitrary amount of time. When a peer disconnects from the peer-to-peer network, then its donated resources become unavailable instantly regardless of the state that they were in. This can lead to failures in computations that were using that acquired *nodes* when the user application is not expecting this. This fact imposes requirements on a user application to be able to survive such failures.

Chapter 3

ProActive Peer-To-Peer Network

There are many different ways that peer-to-peer networks can be implemented. This chapter presents the design of the existing peer-to-peer system in ProActive. My contribution implementation will resemble some aspects of how the existing solution is organized.

A peer-to-peer computational system is a component of the ProActive library that makes it possible to use resources from desktop machines so as to deploy distributed applications. The system was created with desktop machines in mind, however it is not technically limited to them. In this idea, each participant of the peer-to-peer network donates its resources. These resources are represented in the form of *nodes*. Each peer creates *nodes* that can be used by any distributed application that is using the ProActive library. The distributed application is able to specify that it is willing to use the resources from the ProActive peer-to-peer network by means of, for instance, a deployment descriptor (which is not covered in this thesis, [6]) or by using the peer-to-peer API.

The full description of the existing peer-to-peer infrastructure in ProActive can be found in [8]. The rest of this chapter is organized as follows. First off, the features of peer-to-peer computational network are described in a global scope. Then a general view of the architecture of a single peer is presented from active objects perspective.

3.1. Global Layout

The ProActive peer-to-peer network is an unstructured overlay network of peers. Each peer has a number of acquaintances that are generally chosen randomly (for example by sending an Exploration message). There is a limit on the number of acquaintances for each peer and if that limit is reached, no more acquaintances can be made until the number of acquaintances drops. The network of acquaintances is maintained by sending HeartBeat messages between a pair of acquaintances periodically. If there is no reply from the existing acquaintance after a timeout, it is assumed that that peer had a failure and is deleted from the acquaintances list. In that case, the number of acquaintances drops under the limit and a new search for acquaintances is conducted.

The resources can be acquired by a user application by calling an active object representing the peer on the local machine. When we do this, the peer returns a lookup object which will store references of acquired *nodes*. Next, the peer-to-peer implementation performs the query by sending messages in the peer-to-peer network. That activity is completely transparent for the user. When a local peer is asked for resources it can process the request in two distinctive ways:

* When a requester asks for a single *node* a random walk algorithm is used. First, a peer checks if it itself has free resources. If they are sufficient to fulfill the request, a local *node* is sent to the lookup object and communication ends. If not, one random peer is chosen from the acquaintances list of that peer and the message is forwarded there.
* When a requester asks for more than one *node* a breadth first search algorithm is used to find the resources. First, a peer checks for free resources on itself. If there are enough of them, the process ends similarly to the aforementioned case. If not, a request message is forwarded to all of its acquaintances. Each such message is supplied with: a TTL (Time To Live) number which prevents the peers from infinitely forwarding the message, a number of *nodes* to acquire which is decreased before forwarding by the number of *nodes* offered by the forwarding peer. The latter prevents from flooding the network with the request when it is already fulfilled. In a way it also prevents from flooding the original requester with excessive amounts of available resources.

On the node lookup object side, as soon as it receives a resource it acknowledges the donor of its intention to use it. This is necessary, because it may be that there are more offers of resources than the requester needs. The resource owner after offering the *node* (or *nodes*) waits for the confirmation from the requesting peer. In case of timeout or refusal of the offered resources they become available again.

In order to prevent flooding of the network each peer maintains a list of IDs of recent messages that it has sent. If the received message ID is already on that list the processing is discarded.

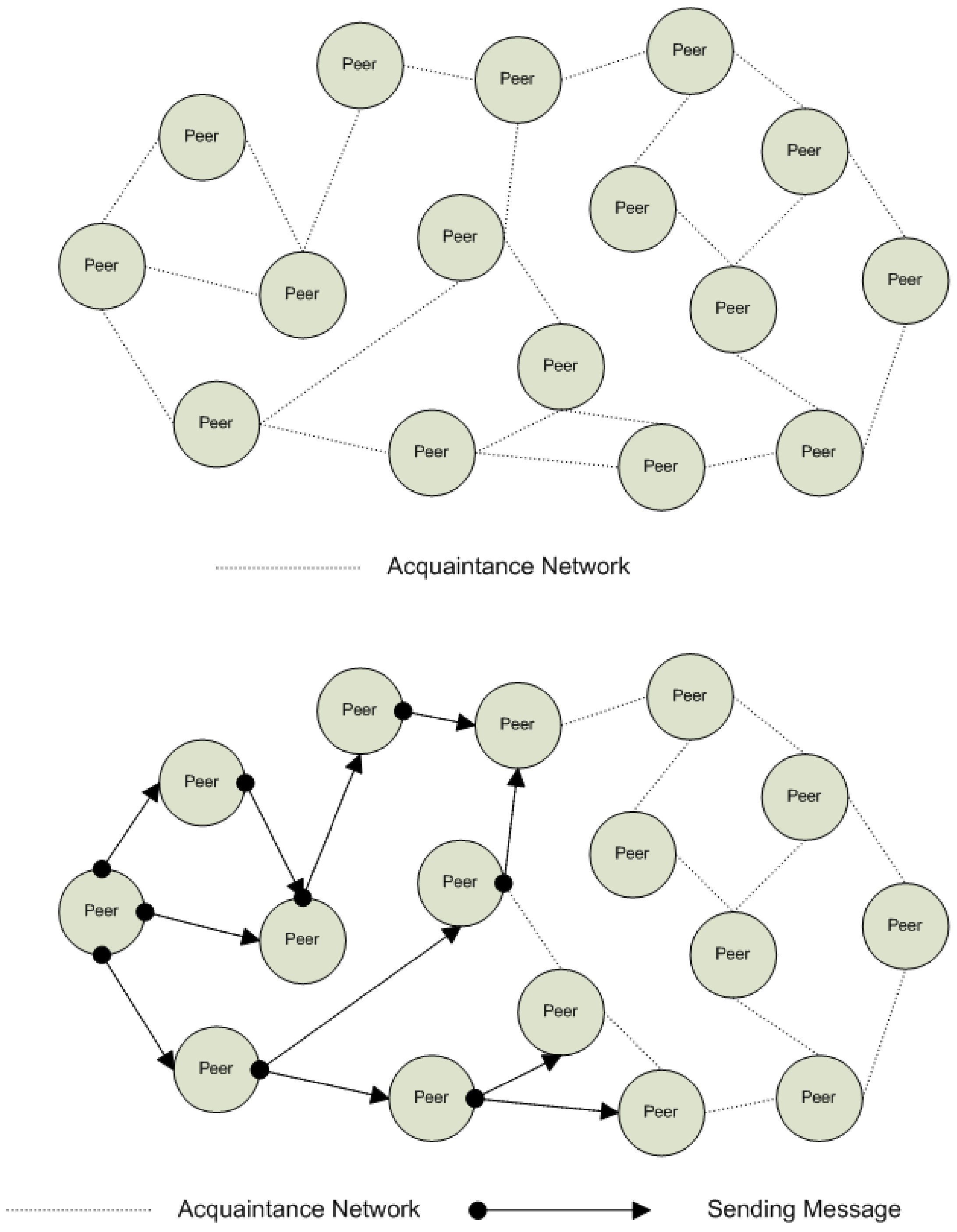
After the user application finished using resources from peer-to-peer network, they have to be given back to the peers where they were acquired from. In case of a failure of the peer which we were using resources from, the *node* that we were using will be already destroyed and inaccessible.

3.2. Peer Architecture

In order to portray a general view of the architecture of a peer, one can notice that the peer-to-peer component representing one peer is divided into three parts. These parts are responsible for different functionalities of the system and each part is represented by a single active object. Thus, a running implementation of a peer is composed at least of three active objects. Apart from that, there could exist lookup objects that are representing queries for resources. These objects are also a part of peer-to-peer infrastructure and they are active objects too. However, the main difference between them and the three aforementioned parts is that the lookup objects represent actions commenced by the user and they are not a building block of the peer.

3.2.1. P2PService

The P2PService active object is an entry point for the user of the peer-to-peer network. This object can be referred as a representative of a peer. It implements the main operations for



At the top: Overlay Network formed by network of acquaintances. In this example for each peer the maximum number of acquaintances equals to 3.

At the bottom: Breadth First Search message flow. Using this scheme a request for resources or exploration message can be processed. In this example the TTL is equal to 3. Thus, we don't have access to all of the peers in the network.

Figure 3.1: Overlay network and resource request in the peer-to-peer network

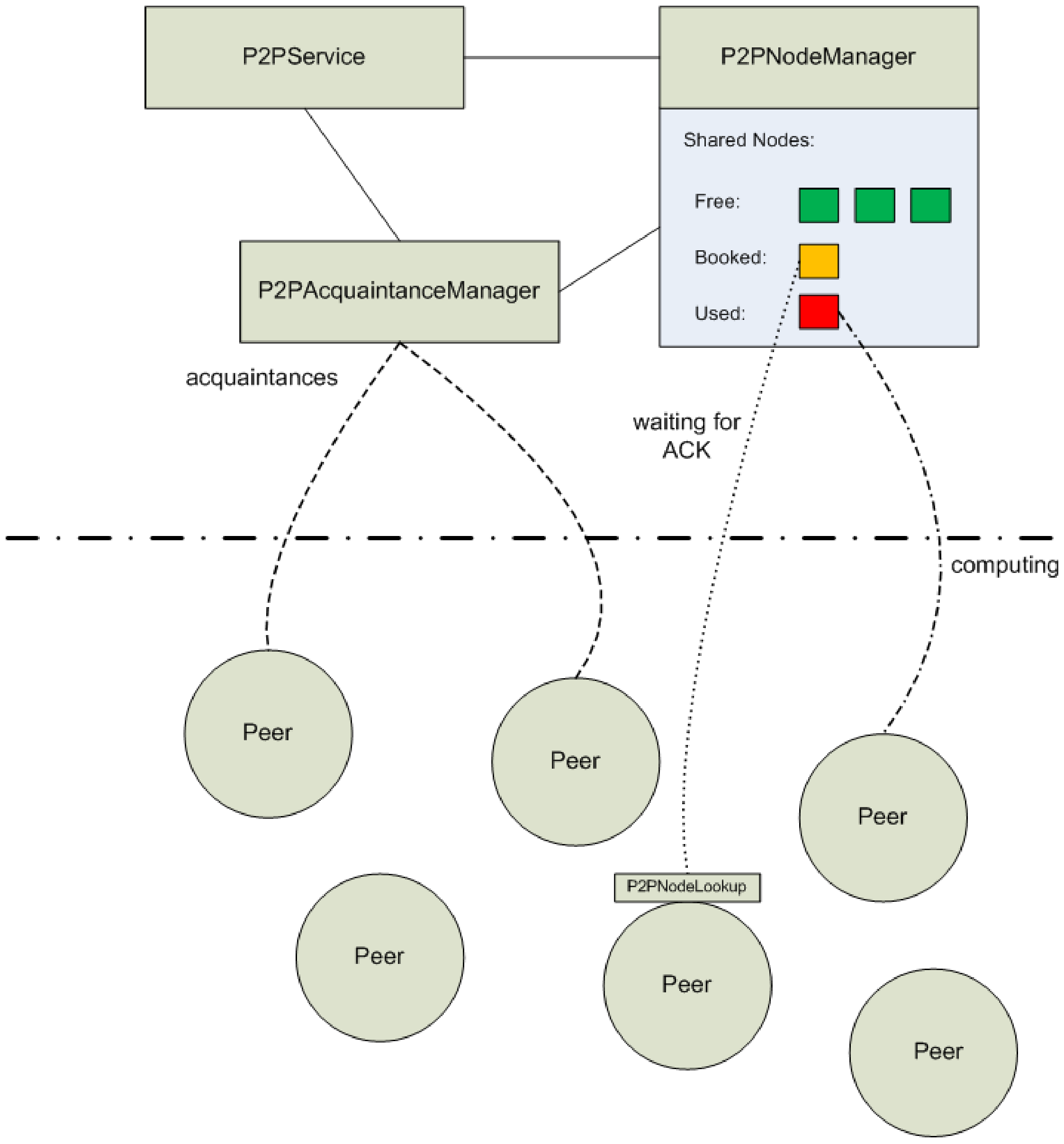


Figure 3.2: Internal architecture of a peer

the user and coordinates the other components. One of its roles is to start the participation in the peer-to-peer network. It is done by initializing this object with a list of peers that should be contacted at the first opportunity. It also exposes methods for acquiring *nodes* from the network.

It is possible for the user not to deal with the P2PService object and still use the resources from the Peer-to-Peer infrastructure. A number of requested *nodes* can be specified in a deployment descriptor. In that situation, a P2PService object will be implicitly requested by the deployment implementation when the ProActive runtime starts.

3.2.2. P2PNodeManager

The computational ability of a peer is represented by means of ProActive *nodes*. Each participant of the network creates a number of such *nodes*. By default, the number of *nodes* is determined by the number of processors multiplied by the number of cores residing on the peer's machine. This allows to use all of the resources that a machine can provide when each *node* is used by one active object and thus one thread. These set of *nodes* used inside peer-to-peer infrastructure are called *shared nodes*.

The P2PNodeManager manages *shared nodes*. This part of the infrastructure is used when a request for resources is processed and the manager is asked how many *nodes* are available. It uses the best effort strategy and gives as much *shared nodes* as possible to fulfill the request. As much as possible means at most the amount that was requested.
A *shared node* can belong to one of three lists:

- List of free *nodes*. These *nodes* are available and ready to be acquired.
- List of booked *nodes*. This list represents *nodes* that are booked. They are blocked from being available to the other requests because the requesting peer has still not confirmed its willingness to use this *node*
- List of *nodes* being used. On this list there are *nodes* that are acquired and confirmed to be used by the requesters.

These three lists represent the state of a peer's resources. The role of the P2PNodeManager is to manage this state according to the events happening in the system.

A P2PNodeManager provides operations that are used internally by the lookup object when processing user requests. The first type of operation has already been described (asking for resources). Other operations that this component is responsible for are releasing the resources and handling notifications about using the *nodes*. The first operation is called from the lookup object when a user releases resources. It moves *nodes* from 'used' into 'free' list. The latter is called when the lookup object confirms that it will be using the offered *nodes*. If this happens, that *node* is moved from the 'booked' to the 'used' list. If the lookup object doesn't want to use the offered resources, the P2PNodeManager is informed that the booking is canceled and *node* is moved back from the 'booked' to the 'free' list.

3.2.3. P2PAcquaintanceManager

A P2PAcquaintance is a component responsible for managing the overlay network - list of acquaintances. The acquaintances can be made in two distinct ways:

- By contacting the peers from the 'preferred contact' list. This list belongs to the P2PService object and is used to join the network during the bootstrap and in emergency situations when the number of acquaintances is not sufficient. At the beginning, this list contains the 'first contact' list of *nodes* from the P2PService object. But as soon as new acquaintances are made outside the 'first contact' list, they are also added to the 'preferred' list. This is done in order to have a larger emergency list of contacts that we could probe when we lose our acquaintances. It could be useful, for example, when we want to restore acquaintanceship with some other peer because of temporary failure of that peer.

- By sending an Exploration message. If the previous action is not able to restore the given number of acquaintances, an Exploration message is sent to all acquaintances. An Exploration message is a request for acquaintanceship sent to the network using a breadth first search algorithm. It is possible that there are peers that are not known for us and are also looking for new acquaintances. If such a peer receives such message it treats it like it was contacted directly with a request for acquaintanceship and starts the acquaintance handshake (which will be explained in the next paragraph).

Acquaintanceship is a bidirectional relation. If a peer A has a peer B on its acquaintance list then the peer B has also the peer A on its own list. This has a consequence that making acquaintances requires an agreement between both sides. To satisfy this, there exists a handshake protocol which ensures that the number of acquaintances on both sides is lower than the maximum level. The algorithm has the following steps:

- The requesting peer is contacting a potential acquaintance candidate to inform about its intentions.

- The requesting peer stores the information about the sent request in its awaited replies list.

- The requested peer analyzes the request and its own state. If the requester is already on the list of acquaintances of that peer the request is denied. If not, it computes the potential number of acquaintances in order to decide on this. That value is equal to the number of existing acquaintances plus the size of the list of awaiting replies (the same as in previous point for the caller). Now:

  - If the potential number of acquaintances > 2*[limit of acquaintances] then the request is denied
  - If the potential number of acquaintances > [limit] and < [2 * limit] then a request is accepted with a probability equal to the value of a linear function of potential numbers of acquaintances which has the following properties:

    $f(limit - 1) = 1$
    $f(2 * limit) = 0$

  - If potential number of acquaintances < [limit] then the request is accepted

  From this point one can see that a peer is able to accept the acquaintances above its limit. If this is the case a peer will periodically remove random acquaintances in order to keep the number of acquaintances within the limit. All of these activities are done in order to aid situations in which the existing members of the network form a topology in which they are all well connected and have reached their limit of acquaintances. In such situations it is nearly impossible for new peers to find acquaintances. By allowing existing peers to accept new acquaintances above the limit temporarily we can connect new peers to existing ones. Later, when existing peers remove random acquaintances they will restructure the overlay network to include new members in the topology.

- In either case (request accepted or denied) it will try to notify the peer about its decision
- If successful, the requester adds a new acquaintance. If not, the request is removed from the awaited list.
- Periodically, the requester removes timeouted requests from awaited list (in case no reply was sent)

As the peer often forwards requests to all of its acquaintances this kind of communication can be optimized. Instead of sequentially contacting every peer from the acquaintance list we could do a broadcast to all of them simultaneously. The ProActive library provides us with a mechanism to perform this kind of behaviour. It is called typed group communication [5], [4]. This mechanism allows to communicate with a group of active objects and it is used in peer-to-peer network implementation to forward breadth first search messages.

3.2.4. P2PNodeLookup

The P2PNodeLookup object represents a resources lookup. It is returned by the P2PService after a user asks it for *nodes*. The requested resources will be retrieved by this object. It is the responsibility of the user application (or deployment descriptor framework implementation) to unload nodes from this object and use them in the application. Similarly, the user is responsible for returning *nodes* back by calling killAllNodes() on this object when the computation is finished. Unless the latter is done, the *node* cannot be reused by the other peers anymore. The following example shows how to use the peer-to-peer computational engine to acquire resources manually in the application code:

```
P2PService serviceP2P = startServiceP2P.getP2PService();

P2PNodeLookup p2pNodeLookup = serviceP2P.getNodes(40, "workerVN", "jobID");
// we are asking for 40 nodes

List<Node> nodes = new ArrayList<Node>();

while (!p2pNodeLookup.allArrived()) {
        nodes.addAll(createNodes(p2pNodeLookup.getAndRemoveNodes()));
        System.out.println("Got " + arrivedNodes.size() + " nodes!");
        Thread.sleep(10000);
}
nodes.addAll(createNodes(p2pNodeLookup.getAndRemoveNodes()));
// Do something with nodes
//...
// release nodes
p2pNodeLookup.killAllNodes();
```

The first line is an initialization of the peer-to-peer service on our machine. Then a lookup is generated: we are asking for 40 *nodes* that we want to be contained in the *virtual node* called "workerVN". The last parameter is a property defined in every ProActive *node* but to keep the explanation simple the description of this field will be omitted. In the loop we wait until all *nodes* arrive (in this example we don't assume that we can receive incomplete number of *nodes* to keep it simple). The getAndRemoveNodes() method removes all the new *nodes* that arrived. After we finished using received *nodes* we have to release them using killAllNodes() method of the lookup object.

Chapter 4

Gossiping Protocol

The main part of my contribution to the ProActive project was a design and implementation of different approach in peer-to-peer computing - the peer-to-peer computational network based on gossiping protocol. The existing implementation of peer-to-peer network uses protocol similar to Gnutella [14] to ask for resources. One of the qualities specific to that protocol is that the messages are broadcast on demand: if a peer needs an acquaintance it starts a breadth first search message to the peers that it already knows. Similarly, when a peer asks for multiple resources the breadth first search is performed in the network. Intuitively, it all happens because the peer doesn't store any knowledge about the network locally so it can't do anything more clever than blindly send a request to all the peers it knows hoping that the message will finally reach the matching peer. Unfortunately, this type of behaviour can lead to huge fluctuations in the bandwidth usage of the peer which sometimes can lead to flooding the peer.

The proposed solution is a protocol based on gossiping algorithm concept and the reference implementation of that protocol in ProActive (described in Chapter 5). In this gossiping approach peers are periodically exchanging the knowledge about the network by sending gossip messages. Because gossiping is a periodic activity we can easily control the network bandwidth used by the gossiping. Moreover, relying on the local knowledge a peer can talk to remote peers that are likely to have resources and ask for them directly.

In order to fully control the size of the communication channel allowed for the internal peer-to-peer communication this solution proposes strategies that keep the gossip message size within the limit. The minimum and maximum message size along with other parameters of the network can constrain the size of internal communication of the network. The results of the experiment described in Chapter 6 prove this feature of the network. In effect, a designer of a peer-to-peer deployment given an estimate of the network bandwidth the peer-to-peer communication is allowed to use, can adjust parameters of the network in order to use restrain the communication.

As each peer only depends on its local state it is important to keep the state as consistent as possible. Achieving this goal greatly depends on the speed of dissemination of the state changes in the network and the frequency of those changes. Limiting the communication in gossiping network can lead to the decreased speed of dissemination of the state of the network. These issues will be addressed in this chapter when strategies for keeping the message size within the limit will be presented. In general, the more strict limit on communication is applied, the network propagates state changes more slowly during rapid changes. These situations will be described and analyzed further in Chapter 6.

The description will start from defining the state of the peer-to-peer gossiping network. Each peer maintains that state and the messages sent to the network by the peer are based on recent changes to that state.

4.1. Peer State

The state of the network is described by the following two lists:

- List of available peers - that list contains entries with peers which are believed to have some available resources.
- List of busy peers - that list contains entries with peers which are believed to have all the resources currently unavailable.

The difference between the gossip approach and the standard peer-to-peer approach in ProActive is that in the original solution a peer doesn't maintain any such state. Instead, the peer is storing only information about its acquaintances. Therefore, in order to ask for resources a peer must contact its acquaintances and it is likely that they will have to forward this request further. In the gossip approach a peer will directly talk with peers which are believed to have necessary resources. Also, a peer doesn't have any special set of peers called acquaintances. Instead, it will try to contact some peers from the two above lists in order to send the gossip.

The entry of each list is composed of the following:

- Peer ID - a unique identifier containing information about the peer's URL
- Timestamp - a notion of time specifying the moment of time when the entry information (whether peer is in the 'available' or 'busy' list) was generated. The value is taken from the peer that this entry refers to.

4.2. Communication

The peers communicate with each other to exchange the knowledge about the system and to request computational resources. Similarly to the existing solution, the peer-to-peer component is divided into three active objects: P2PService, P2PGossipEngine and P2PNodeManager. Instead of P2PAcquaintanceManager, there is a P2PGossipEngine because in this approach a peer doesn't manage a fixed number of acquaintances. More on the architecture will be covered in the next sections. What is important for communication is that depending on request type the peer will be talking to different active object. If the communication concerns gossip exchange then P2PGossipEngine will be called directly. If resources are requested, the communication will involve P2PNodeManager. This way of communicating is different than existing solution (in which all the requests have to call P2PService) and is aimed at simplifying synchronization between threads. Let us recall that each active object has its own thread of control. Then using one thread to process one kind of request dramatically simplifies the concurrent architecture as the logic becomes decoupled. We only have to worry about synchronizing data collections which can be accessed by multiple active objects when they are referencing each other. It will be explained further that there is only one such collection in the implementation. It is called EventQueue. By shifting the burden of synchronization from managing manually threads of active objects to the data structures it becomes more easy to produce a safe solution. What we are relying on is just the synchronization mechanisms available in the Java language.

There are two types of messages that are sent between peers:

4.2.1. Gossiping Message

A gossiping message which is send periodically from a peer to some number of peers taken from its state. Basically, this message contains the state change of that peer since the last gossip. For example, the message can contain a list of peers that were added/moved to 'Busy' and 'Available' lists. Apart from containing this information, the contents of the message will be adjusted subject to a couple strategies explained below. These strategies are used in order to control the size of the sent message. For now, we can assume that this message contains only the whole change of the state. That is:

- A list of new peers that were added to the 'Available' list since the last gossip
- A list of new peers that were added to the 'Busy' list since the last gossip

The expression 'new peers' means here the new entries on a given list. These entries may come from new peers that were added to the list or existing peers after changing their state (i.e. moving from one list to another).

Gossiping is conducted periodically and the time between sending gossip messages is configurable. In the meantime, a peer receives messages which can modify its state. These changes will be the part of a next gossip message coming from that peer.
The following happens on a receipt of the gossip message by a peer:

- The entries of each list are analyzed. For each entry:
    1. If a PeerID does not exist in local state, then the entry is added to the appropriate local list.
    2. In the opposite case the timestamp value is analyzed. If the value is lower or equal to the value of existing entry in the peer's state, then that entry is ignored. If it is higher, then existing entry is removed and the entry from the message is added to the appropriate list.
- Any changes to the local state are recorded so as to create a new message that will be used when new gossip is to be sent.

Before sending the gossip message the following policies are applied in order to control the traffic generated by gossiping:

- Minimum message size policy - This policy ensures that a message contains at least some number of entries. That number is a parameter of the peer-to-peer network and is global to all peers. The implementation of this policy is following:
    - If the message size is below the threshold, a peer attempts to add an entry about itself to the message.

  - If that is not sufficient, a peer attempts to add randomly chosen entries from its state to satisfy the limit. In the worst case, the message size will be below the threshold because the peer has too few peers in its state to make message big enough.

- Maximum message size policy - This policy ensures that a message contains at most some number of entries. That number is also a global parameter of the peer-to-peer network. The implementation of this policy is described below:
  - The 'prepared message' means the message that was about to be sent without applying this policy
  - If the size of 'prepared message' is below or equal to the threshold then it is left unchanged.
  - In other case, a different message is prepared individually for each recipient. Every such message contains a threshold number of entries chosen randomly from all entries of 'prepared message'.

The gossip messages are sent to some number of peers. These peers are randomly chosen from the state of gossiping peer. In this solution the number of recipients is a constant that is a global parameter of the peer-to-peer network. In the future, it would be wise to examine if a dynamic behavior would provide benefits in some situations. If there are no peers to choose from (the state of a peer is empty without including information about itself) then gossiping is discarded for this time and changes continue to accumulate and will be used along with existing ones to create next message in the future.

4.2.2. Request Message

A request message is sent by a peer that wants to acquire resources from the network. One of the feature of gossiping approach is that each peer has its own view of the state of the network and will use only this knowledge to request resources. That means that resource query is performed by calling a requested peer directly. As a result, the communication involved in this operation is much more simplified than the existing peer-to-peer solution. One of the consequences is that a peer almost immediately gets the requested resources.

The request message contains the number of *nodes* that the peer wants. That message is then sent sequentially to all of the peers that are on the 'Available' list until the request is fulfilled. Each peer asked for resources analyzes how many available *nodes* it has. The peer maintains the state of its own resources similarly to the P2PNodeManager described before. The only difference is that it has two lists instead of three: list of used and free *nodes*. There is no 'booked' list because the requester talks sequentially and with one peer at time. The answer comes immediately and is awaited by the requester before the next call so there is no possibility that excessive resources will be given to the requester. Thus, a NodeManager can assume that if it gives available local resources they will be used.

The outcome of the request for resources operation is used to update the state of the requester and the requested peer. In the reply from a request the requester receives:

- A list of *nodes* that it can use.
- Number of available *nodes* after replying to the request on that peer.

- A timestamp for that information.

On the requester side, the information from the message is used to update the entry referring to the asked peer. Even if the peer stays in the same list (for example it was marked as 'Available' and after the query it still has some resources available) the timeout value will be updated to reflect the freshness of the information. That change can be used in the next gossip exchange. The algorithm works as follows:

- If a request was denied (reply contains 0 *nodes*), then that peer is moved to 'Busy' list with timestamp from the reply
- If a request was successful (some resources were acquired) and the remote peer has 0 *nodes* left, than that peer is moved to 'Busy' list with timestamp from the reply.
- If a request was successful and the remote peer has a positive number of *nodes* after the query it stays in 'Available' list but its timestamp is updated with that from the query result.

The same actions happen on the remote peer. When its resources become unavailable it moves itself to the 'Busy' list. In the opposite case, the reverse action is taken accordingly. The important thing is that both requested peer and the requester use the same timestamp value for their modifications. If we removed this requirement, it would result in unnecessary gossiping about the state of the requested peer. Instead of propagating this information from two independent sources, the more recent version would try to take over and replace the older one.

After the peer receives the asked number of *nodes* or the list containing peers to ask becomes empty the request operation ends. The result is a list of acquired *nodes*. The peer makes the best effort to provide the requested number of *nodes*. However, it is not guaranteed that requirement will be fulfilled. The result list might contain less *nodes* then the requesting application asked for when they are not available according to the knowledge of peer. Taking into account that each peer uses only its knowledge to acquire the resources, it is important for the new information to disseminate across peers as fast as possible. The proposed protocol addresses this requirement and evaluation will be provided in Chapter 6.

4.3. Bootstrapping

The important aspect of peer-to-peer system is the process of becoming a part of the network. This poses a challenge in the gossip based approach as we have to obtain the state of the whole network as fast as possible. However, we cannot conduct behaviour similar to exploration messages available in the existing solution because we would start sending messages on demand which is not a desired behaviour. The proposed approach along with the described strategies of gossiping solve this challenge. Its performance will be evaluated in Chapter 6.

The proposed solution is a simple bootstrapping component. It contains a list of peers that will be used to make the first contact (the similar concept as 'first contact list' in existing solution) with other peers. The bootstrapping algorithm will be started whenever the number of peers to send gossip message to drops to zero. The steps of that algorithm are as follows:

1. A new gossip message is created. That message contains the nodes from the bootstrap list and they are stored in the 'Available' list. Each of the entries of this message has

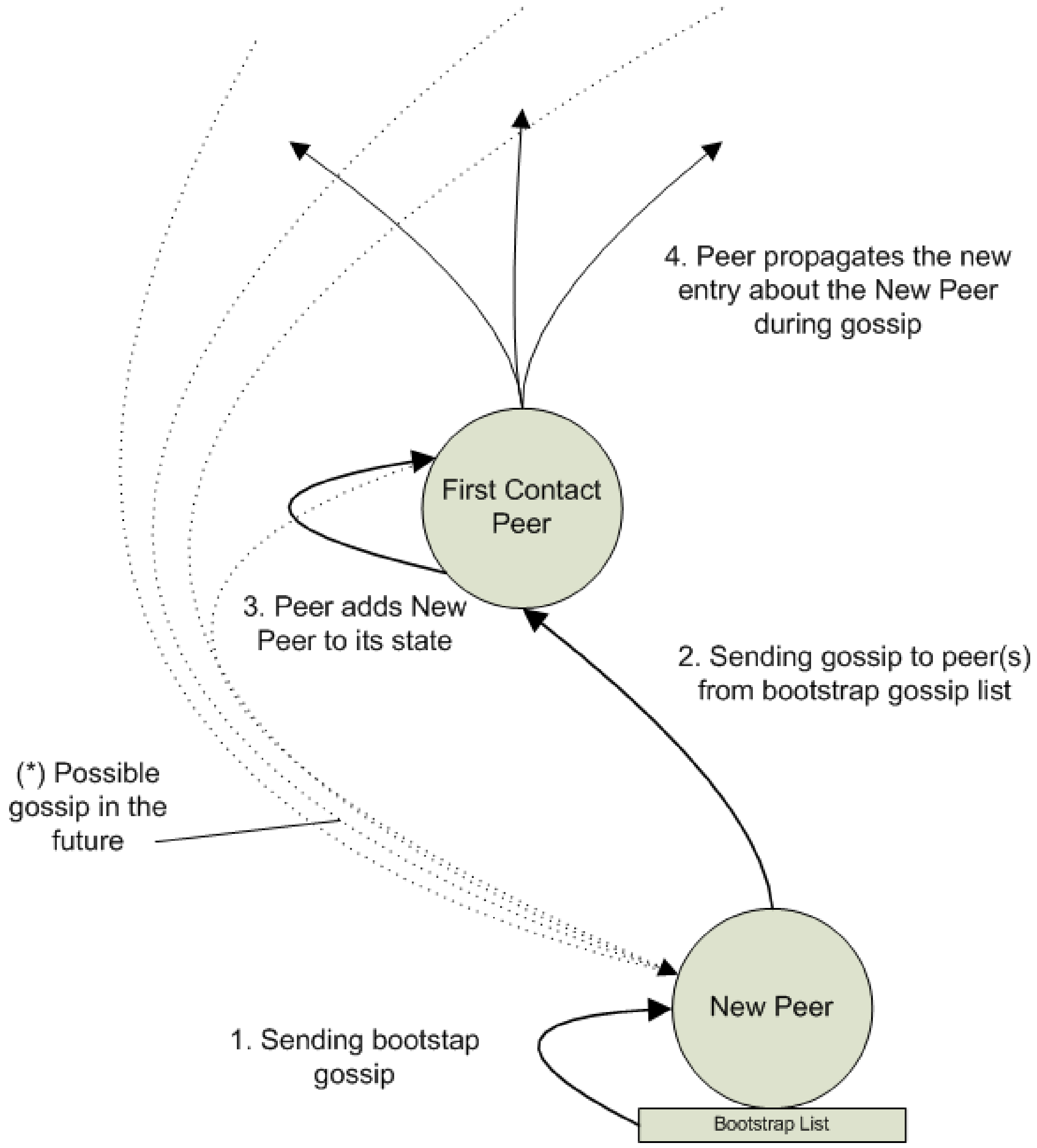

4. Peer propagates the new entry about the New Peer during gossip
First Contact Peer
3. Peer adds New Peer to its state
2. Sending gossip to peer(s) from bootstrap gossip list
(*) Possible gossip in the future
New Peer
1. Sending bootstap gossip
Bootstrap List

Figure 4.1: Bootstrap in Gossip Peer-To-Peer

'incorrect' timestamp of the following property: each 'correct' (not set by the bootstrap mechanism) timestamp is greater than the 'incorrect' timestamp. Furthermore, all of the 'incorrect' timestamps have the same value.

2. The bootstrap list is cleared.

3. Such a message is sent to the local P2PGossipEngine mimicking a gossip exchange.

4. Later on, on any connection error while attempting to contact any peer whether to exchange gossip or ask for resources, that peer will be removed from the local state and will be put to the bootstrap list. That list will be used again in the future when the number of peers that can receive gossip messages drops to zero.

Let us examine what happens when a peer receives a bootstrapping gossip. Because initially its state only contains information about itself, it will be populated with all of the entries of the message. After that, during the next gossip exchange it will have peers that it will be able to send a gossip message. After the message is sent, each recipient will discard the IDs with 'incorrect' timestamps that it already has knowledge of. Any new entries will be added and will extend the potential candidates for gossip or resource acquisitions. The 'invalid' timestamp ensures that any information received later about those peers will replace existing entries. Assuming that the recipient of a gossip doesn't know the sending peer, information about that peer will also be added. Thus, gossiping between these two peers will be possible in the future and the state of a new peer will be gradually updated.

The update of a new peer is important to be performed as fast as possible because as long as it is not complete, the names of many peers which can potentially have available resources are not known. The problem in this solution was that if nothing happened in the network, then initially no messages would be exchanged between peers. As a result, it was difficult to join such network for a new peer. Fortunately, now different policies are applied that guarantee a decent update of the state. First, there is a minimum message size policy that guarantees a minimal amount of information in each message. The new peer will be therefore informed about existing entries in the states of existing peers even when there is no or very little new knowledge spreading across peers. We can have an impact on speed of integration of a new peer with network by adjusting this parameter. One may think that making an initial exchange with limited number of peers (we can assume that first contact list is small in relation to the size of the network) will slow the gathering process down. Indeed, the bigger the network the less probability that the next recipient of the message will be a new peer as they are chosen randomly from all known peers. However, we must be aware that the peers from the 'first contact' list will know about a new peer and will be able to propagate this knowledge further with very high probability. Figure 4.1 is an example of such situation. During step 2 the new peer sends information about itself to the peer from the bootstrap list. After adding the information to the state (step 3) the recipient forwards this entry during the next gossip exchange to some peers (step 4 - in our situation 3 randomly chosen peers). After this, there are four peers that could send the gossip to our new peer. There will be more such peers in the future as three peers that received the information about our new peer will propagate this knowledge further during their next gossip.

It is important to note that only the bootstrap gossip message can exceed the amount specified in the maximum message size policy. However, this communication doesn't account for bandwidth usage as it is performed locally within one peer.

Chapter 5

Gossiping Protocol Implementation

One of the contributions of this thesis is an implementation of the gossiping protocol in the ProActive library described in the previous chapter. The new implementation exists along with the existing one and a user is able to choose which implementation to use. The code that is implementing this mechanism was written not only to implement the protocol. The design decisions that have been taken before starting implementing the service are outlined in the next section. In effect, there exist a generic implementation of peer-to-peer service using a generic gossiping algorithm and the described mechanisms form one instance of such service. The design of this generic framework will be described in the 'Interfaces' section. The overview of implementation of these interfaces is provided in the section called 'Strategies'.

5.1. Design Decisions

In general, the design of the component should not only strive to provide the desired functionality. As we know the quality of software lies in its design and software projects prove to be successful when they can be easily maintained and extended for a very long time. The most important point in making a good design is to forecast future points of extension and to prevent architectural barriers from jeopardizing the future growth of the software. During my curriculum I attended couple of classes dedicated to designing software. I wanted to use best of my knowledge to provide implementation of the peer-to-peer solution not only correct with respect to functional criteria but also make it maintainable and extensible from the developer point of view. During design and development of the component I relied on design patterns described in [13]. I also created software incrementally with respect to some guidelines from Extreme Programming [7], [12]. One of these was to provide a minimal interfaces and implementation of features.

During the design of the peer-to-peer gossiping implementation the following points were taken under consideration:

- All of the entities that the peer-to-peer component is composed of should be decoupled from each other. Ideally, each of the entities has separate responsibilities and works independent from each other and is replaceable. In the context of the implementation of the peer-to-peer system that would be for example a part of the system that is responsible for answering queries and gossiping.

- The component should be divided into parts that describe the behaviour of the system in high-level abstract way. It should be possible to use low-level parts, combine them and experience different behaviour without changing the high-level code. In our case, we should be able to make use of low-level components and by combining them implement different algorithms for peer-to-peer communication.

- The low-level activities should be also described in an abstract way providing opportunities to easily replace small parts of the system (for example implementing independent small functionalities) as well as plug in additional functionalities. For example, adding new policy implementations.
- The system in general should provide an open framework that other components can plug into. For example, a monitoring system could be used without changing the existing code of the peer-to-peer implementation.
- The system should be implemented in the fashion that resembles 'separation of concerns' approach. That means the functionality should be reasonably separated across classes. Ideally, each class has less than 300 lines. That requirement makes it easy to extend the software in the future and makes the code easy to understand. Therefore, it provides decent maintainability of the implementation.

5.2. The System from the Big Picture

The gossiping peer-to-peer solution consists of three main active objects. These objects can be initialized with different policy implementations and thus provide different behaviour of the network. The figure 5.1 shows these three main objects with dependencies between them. As we can see, these three sub-components are almost independent from each other. The only dependencies that exist are from the P2PService to the P2PNodeManager and P2PGossipEngine. However, these dependencies are very limited. They are exploited only during initialization phase when the peer instance is created. Such independence is achieved thanks to the fourth sub-component that joins the other sub-components together, the EventQueue. In this design objects talk to each other by sending events and subscribing for them. Thanks to this, one component does not need to know what other sub-components are, making it easier to extend the service as additional sub-components might be introduced. In addition to this, the synchronization issues are no longer a concern since sub-components do not talk directly with each other. Furthermore, some other services that work along with peer-to-peer can react on peer-to-peer events easily. For example, a monitoring service is implemented in this fashion.

In the next sections the responsibilities of each sub-component are described and simple overview of the internal communication between them is provided:

5.2.1. P2PService

P2PService is the main component which responsibility is to provide services to the user. It stores the 'url' of the ProActive peer where sub-components are located. Apart from that, it provides the main and the only operation of peer-to-peer network exposed to the end user - acquiring resources. The interface of that method is the following:

```
public List<NodeInformation> getNodes(int howMany, String vnName)
```

where:

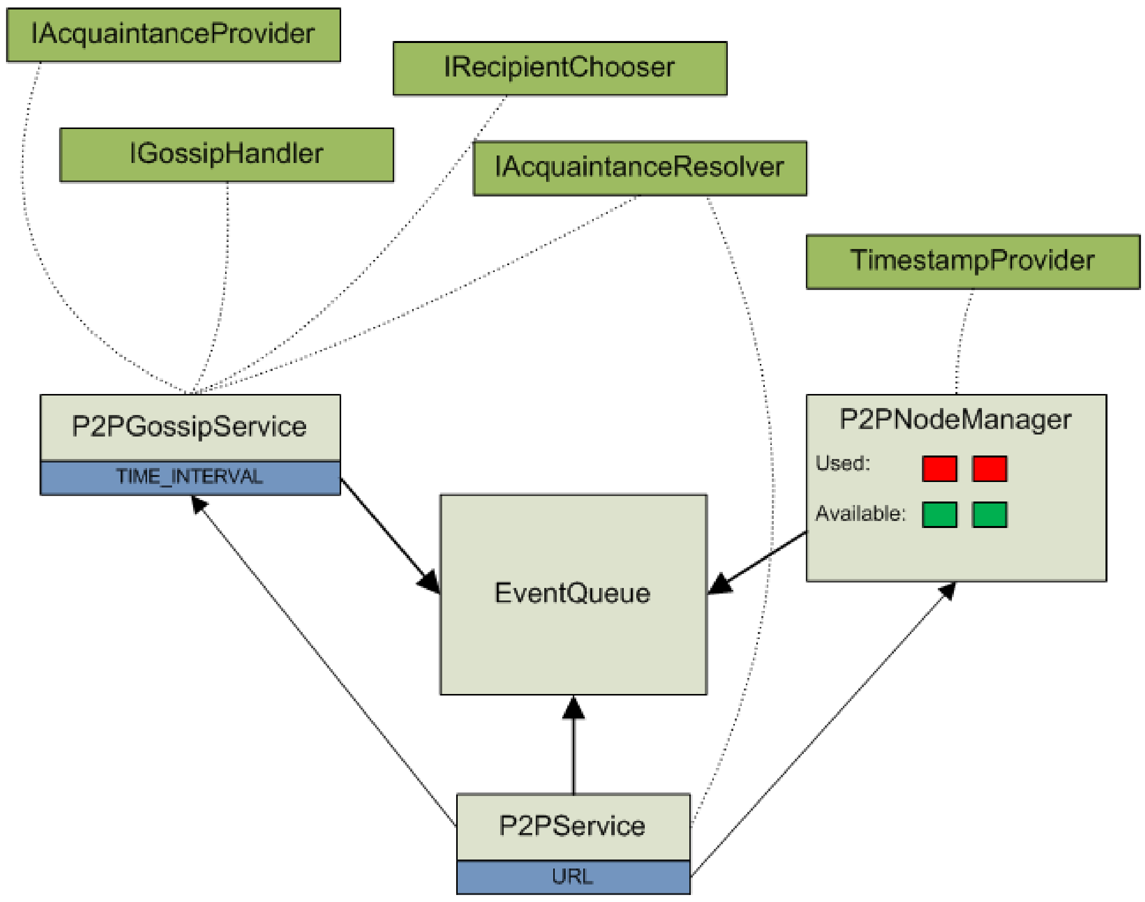


Figure 5.1: Gossip Peer-To-Peer Architecture

- howMany is a number of requested *nodes*
- vnName is a name of *virtual node* which acquired *nodes* should belong to
- The method returns a list of NodeInformation. NodeInformation stores necessary information about the *node*: the Node object itself and remote P2PNodeManager in order to be able to release the resource when we no longer need it. In fact, the resource release mechanism is implemented directly in the NodeInformation class to help user getting rid of that concern.

P2PService implements this method using two strategies which are described in the next section. These are: delivering candidates that will be asked for resources (implementation of the ICandidatesDelivery interface) and a resolver for remote active objects (implementation of the IAcquaintanceResolver). The resolver is used to get stubs to communicate with remote P2PNodeManagers when sending requests.

5.2.2. P2PNodeManager

P2PNodeManager manages the local resources that a peer is making available to the users. It maintains two lists described before: the 'Available' list of *nodes* and 'Busy' list of *nodes*. Its main functionality boils down to answering to resource requests. Here is the interface of that method:

```
public NodeQueryResult queryForNodes(int howMany, String vnName)
```

where:

- howMany and vnName are the same parameters as in getNodes(..) method
- the result contains the information returned with the request described in the section devoted to communication in gossiping solution: list of *nodes*, number of *nodes* that are still available and timestamp of that operation.

The P2PNodeManager uses timestamp generator (LocalTimestampProvider interface) that is described in the 'Strategies' section.

Second activity that this sub-component does is releasing the resources. The proper method is:

```
public void giveBackNode(Node givenNode)
```

where:

- given Node is a *node* that was acquired earlier from this peer

This method moves the specified *node* back from the 'Busy' to the 'Available' list. If the 'Available' list is empty before this operation, it will send a new event to the EventQueue signaling that the peer has some available resources again. Based on that, its state will change to 'Available'.

5.2.3. P2PGossipEngine

P2PGossipEngine is a component dedicated to manage communication between peers. It implements a generic gossiping algorithm that can be customized to achieve different behaviors by using strategies. Its main responsibility is to receive and send gossip. The implementation of the algorithm is very simple:

```
public void runActivity(Body body) {
   Service service = new Service(body);
   while (body.isActive()) {
      serveRequestsAndWait(service);
      sendGossip();
   }
}

public void receiveGossip(AbstractGossipMessage msg) {
   eventQueue.sendEvent(EventType.GOSSIP_MSG_RECV, msg);
}
```

The runActivity method as described in Chapter 2 is a main function of the thread of a given active object. During its lifetime (the code inside the while loop) it is listening to incoming requests for some amount of time. That amount of time equals to the interval between gossiping and is a parameter of the P2PGossipEngine object. After that, a next gossip is sent. The implementation of sending gossip is only sending an event to the EventQueue. One of the strategies that subscribed to this event handles this operation. The reason for such implementation style is to make the internal state of a peer independent from algorithm for communication. If we added a method to this class or make a reference to another object we would add a dependency from the P2PGossipEngine. As a result, we would have to provide some information about the state manager in order to use it to send the gossip. But on the other hand, we would like to have an algorithm that only describes the scheme for communication. We then will have enough flexibility to apply different state management or even different state definition in the future without modifying the algorithm itself. Also, having a state implementation it will be equally easy to change the communication algorithm if the new one is implemented in the similar fashion.

P2PGossipEngine object uses following strategies to send gossip: acquaintance manager to provide information about all possible recipients of a gossip (IAcquaintanceMgr interface), recipient chooser to limit this list with some criteria (IReceipientChooser interface), resolver of peers (IAcquaintanceResolver interface) and finally a strategy that prepares a gossip message based on the state (IGossipHandler interface). These strategies will be described in the next section.

5.2.4. EventQueue

EventQueue is a linking point between sub-components. It is used to notify other entities about events happening in the system. For example, when an attempt for resource acquisition fails (coming from P2PService) the strategy that maintains state of the peer (IGossipHandler interface) would like to know about this outcome to update the state. The components register to the queue so as to be able to handle notifications. In order to be able to cooperate with the queue they have to implement ISubscriber interface that contains one method:

Event Name	Description
URL_REGISTERED	The URL of the local peer is known. It is set during initialization.
NO_FREE_NODES	The local number of available *nodes* dropped to zero.
FREE_NODE_AVAILABLE	The local list of available stopped being empty.
NODE_REQUEST_DECLINED	The request for resources returned 0 *nodes* from some peer.
PEER_STILL_HAVE_RESOURCES	The request for resources was successful and remote peer still have free *nodes*.
GOSSIP_MSG_RECV	Gossip message has been received.
GOSSIP_MSG_SENT	Gossip message has been sent.
NODE_CONN_FAILURE	There was an error while connecting to some peer.
NO_ACQUAINTANCES	System don't have any peers that it can send gossip to.
NEW_ACQUAINTANCE	System got information about new peer.

Table 5.1: Event Types

```
public void onEvent(EventType type, Object payLoad);
```

which handles incoming notifications. The subscriber has then to register itself in the queue. For example:

```
eventQueue.register(this, new EventType[] { EventType.URL_REGISTERED });
```

This code registers the object itself by putting this to the EventQueue, and will be notified about any URL_REGISTERED event.

In order to send an event the object has to call:

```
public synchronized void sendEvent(EventType eventType, Object payLoad)
```

method in EventQueue. The parameters are self-explanatory. The method is synchronized in order to keep the queue in a consistent state. It is the only place in the peer-to-peer implementation where different threads might compete for resources. All other activities are performed by active objects independently.

5.3. Strategies

Strategies are small building blocks that the gossiping peer-to-peer service is constructed from. Each such block represents a different aspect of peer's behaviour. The strategies are provided in the form of Java interfaces and thus allow the developer to easily replace or extend the functionality of existing strategies. This can be achieved for example by subclassing existing strategy or using Decorator design pattern. What is more, if some approach naturally combines different strategies' implementations in one class, nothing prevents us from implementing more than one interface at a time because of the Java language. The current implementation of gossiping algorithm supports the following strategies:

5.3.1. IAcquaintanceMgr

IAcquaintanceMgr is an interface that is able to provide a list of all peers that can be contacted in order to send gossip message. In some sense it resembles the old P2PAcquaintanceMgr active object because it is used to communicate with peers but its functionality is greatly reduced in this implementation. The one single method that has to be implemented by this interface is:

```
 public List<? extends AbstractNodeInfo> getAcquaintances();
```

In currently used implementation the manager is listening to incoming gossip messages by notifications coming from the EventQueue and is adding all new peers to the list of possible recipients of messages. The peer will be removed from the list on NODE_CONN_FAILURE event when the peer is unable to be contacted.

5.3.2. IAcquaintanceResolver

IAcquaintanceResolver is used to transform Node IDs into active object stubs that can be used to contact remote parties in order to send messages. Because this strategy is used internally by the implementation of peer to peer network, it provides resolutions only to two types of active objects:

```
public List<P2PGossipEngine> resolveGossipEngines
               (List <? extends AbstractNodeInfo> nodes);
public List<P2PNodeManager> resolveNodeManagers
               (List <? extends AbstractNodeInfo> nodes);
```

The third type of active object (P2PService) provides an interface used only by the end user, thus raises no interest to internal implementation. The implementation of the methods starts by analyzing each Peer ID. It was mentioned earlier that each Peer ID contains information about the URL of the remote entity. Therefore that URL is passed to ProActive internals which then generate a stub of the remote object residing on the peer. The returned list is composed of stubs of remote active objects referring to corresponding node information. In the gossiping approach, the use of typed group communication mechanism [5], [4] was dropped because a new group would have to be created every time the communication is conducted as a peer randomly chooses recipients from all of the known peers.

5.3.3. IBootstrapManager

IBootstrapManager is responsible for bootstrapping and recovery processes during the lifetime of a peer. This strategy is completely independent from the peer-to-peer implementation as it is not referred from any of three main active objects nor EventQueue. Theoretically, it might not be used at all. The interface is used to put the first contact list and initialize the bootstrap:

```
public void addFirstContacts(List<String> urls);
public void startBootstrap();
```

The second method could be actually optional because it is possible to use EventQueue to be notified when the peer has no acquaintances in order to start the bootstrap (NO_ACQUAINTANCES event). However, to keep a descriptive role of the interface it was not removed. In addition to this, the current implementation reacts on the NODE_CONN_FAILURE event and puts such peer into bootstrap list.

5.3.4. ICandidatesDelivery

ICandidatesDelivery - is a strategy that allows to get a set of peers that probably have the required resources. The interface contains one method:

```
public List<? extends AbstractNodeInfo> getCandidates();
```

The current implementation analyzes existing state and returns all peers from the 'Available' list.

5.3.5. IGossipHandler

IGossipHandler - is the strategy used to manage the peer state. The interface defines two methods:

```
public void handleMessage(AbstractGossipMessage msg);
public AbstractGossipMessage prepareGossip();
```

In the current implementation the handleMessage(..) method is not called externally (outside the object implementing IGossipHandler interface) as the object relies on EventQueue notifications - the method could just be omitted but to retain the descriptive function of interface it was not deleted. The purpose of the handleMessage(..) method is to update the internal state when a gossip message is received. In current implementation it does exactly what was described in section 4.2.1 in the 'Communication' part when a gossip message is received. The prepareGossip() method is used when a new message has to be sent. In current implementation the created message contains only changes in peer's state and the minimal size of message policy is applied. The policy needs to be defined here because it relies on the peer's internal state.

5.3.6. IReceipientChooser

IReceipientChooser is a strategy used to filter out the initial list of all possible recipients to provide the actual receivers of a gossip message. The interface contains one method:

```
public List<? extends AbstractNodeInfo> getReceipients
                       (List<? extends AbstractNodeInfo> l);
```

There are currently two implementation of this interface. One is used just to filter out the local peer from the list. It is used internally for example when asking for resources to exclude ourselves from the search. The second one is an implementation of static number of recipients for the message policy. In this case, the first implementation is used at beginning to filter out local entry (this is the example of use of Decorator pattern). Then, a given number of recipients are chosen randomly from the outstanding list. That number equals to the initialization parameter of the implementation class.

5.3.7. LocalTimestampProvider

LocalTimestampProvider provides a logical clock to the peer. It is used frequently when updating the state about local *nodes*. The interface is again very simple:

```
public long getTimestamp();
```

The current implementation uses incremental timestamp values and the value is updated on every call of the method. That method is synchronized to avoid concurrency issues.

Chapter 6

Assessment of Gossiping Protocol Performance

This chapter contains results from experiments performed with my implementation of the proposed gossiping peer-to-peer protocol. First, the monitoring framework that was very helpful to generate results from the experiments is described. Thanks to the open architecture the integration of this layer was seamless and required only writing very little amount of code. Second, experiment results are revealed. These experiments proved the expectations that the proposed solution efficiently uses network resources to maintain the peer-to-peer overlay. The interpretation of the results contains the explanation of the role of each configurable parameter of the network.

6.1. Monitoring Facilities in the Peer-To-Peer Implementation

One of the goals of the implementation of the gossip peer-to-peer network in ProActive is to provide an open architecture which other components can plug into without modifying the code of the peer-to-peer classes. This feature was heavily used during the experiments as the monitoring code is such a component. The EventQueue component of the peer-to-peer implementation is used to notify the monitoring class about the events happening inside the network. Because of this, the whole implementation of the monitoring class contains a handler for notifications coming from the EventQueue. The monitoring facilities use only one type of event: GOSSIP_MSG_RECV which contains the message received by a peer. The whole role of the monitoring component is to log the contents of the message. The logged information is then used to analyze the behaviour of the network. The information was written using the CSV format in order to make it easy to import it to a database or a spreadsheet.

6.2. Statistical Analysis Based on Sent Messages

6.2.1. Experiment Description

The conducted experiment had three goals:

- To prove/disprove that one of the features of the proposed peer-to-peer network is the control over the bandwidth usage of the network. In particular, the number of messages sent by the peer should not exceed the specified amount in the configuration.

- To prove/disprove that it is possible to alter settings of the proposed peer-to-peer network to successfully adapt it to different requirements set by the user applications which can use different granularity of the work items to be completed by the resources from peer-to-peer network.
- To analyze relationships between parameters of the network and how they impact the performance of the network.

The following experiment was conducted to realize these goals. The experiment consisted of 72 simulations launched with different parameters of the peer-to-peer network. The parameters of the network include:

- Minimum size of the gossip message (I used: 1, 2, 5 entries)
- Maximum size of the gossip message (I used: 5, 10, 20, 40 entries)
- Number of peers that are sent the gossip (I used: 3, 5)
- Interval between gossip (I used 1 second)

Such ranges of parameters were used in order to show how they influence the network performance, to prove that they provide a way to model the behaviour of the network making the solution adaptable to different usage scenarios and to observe the behaviour of the solution in different situations.
The experiment was launched in a network consisting of 140 peers. 120 peers were passive during the course of the experiment. Their role was limited to giving resources and gossiping. The other 20 peers were simulating 'users' of the network. Their role was to periodically acquire and release resources. In all of the configurations they were acquiring resources for 10 seconds, then releasing them and then waiting another 10 seconds. These actions were repeated infinitely. Every 'user' was requesting the same number of *nodes*. The total number of requested resources differed between configurations:

- Variant 1: Users requested 50% of the resources in total
- Variant 2: Users requested 75% of the resources in total
- Variant 3: Users requested 100% of the resources in total

There were 72 configurations in total. These included every combination of minimum, maximum size of the message, number of recipients of the gossip and the percentage of resources acquired. Every combination was monitored using the facilities described in the previous section during the simulation. Each simulation lasted for 10 minutes and was performed on 20 nodes in a cluster. There were 7 peers running on each node of the cluster. The nodes were not exclusively used for the purpose of the experiment. The nodes of the cluster were communicating using 100 Mbit Fast-Ethernet network.

The monitoring facilities were tracking each entry of every message that was received during the gossip globally. Each peer generated its own output file by the monitoring facilities. After that, the results were gathered in one file and then processed.

Because of the amount of data to process (ranging between 200 and 700 MB) I created scripts that performed this job automatically. Because the processing took up to 2 hours on one node of the cluster, I completed this process in parallel using 16 nodes from the cluster in 5 stages. The following diagram shows the details of the processing process:

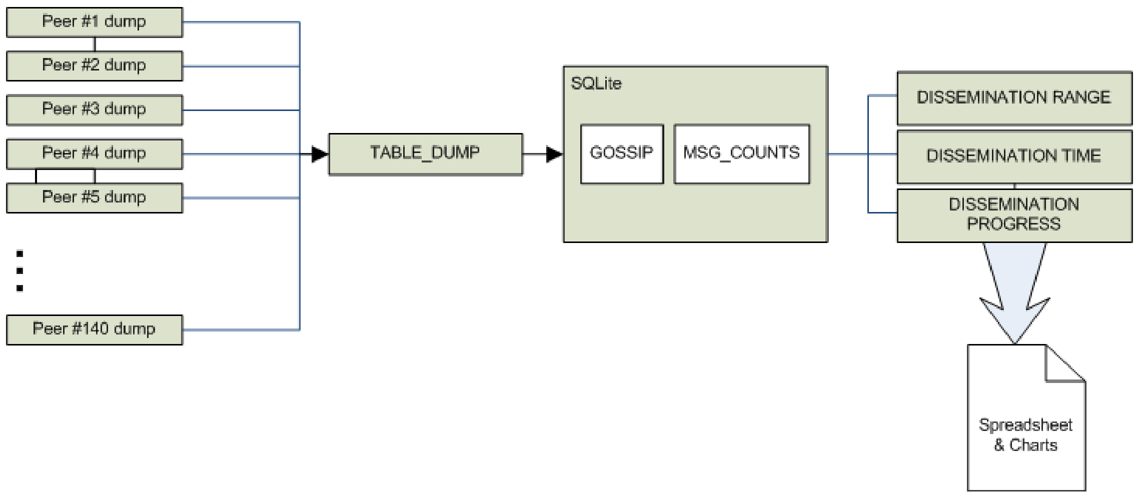


Figure 6.1: Experiment Processing

First, the simulations were performed automatically in 72 configurations (not shown on the diagram). Each simulation generated 140 dump files from each peer. These files contained information about the entries of every message the given peer received. After the simulation these files were merged into one big file containing all the messages that were spread in the network during the experiment (the TABLE_DUMP file on the diagram). Then the TABLE_DUMP file was processed in order to be imported to the database table. Each table entry contained:

- PeerID - ID of the peer from the message entry
- Timestamp - timestamp from the message entry
- Time - the system time of the message receipt. The time was taken from the local clock of the peer. As the simulation was performed inside the cluster the differences between clocks of the peers were minimal.
- ID - ID of the peer that received the message
- Available - flag if the entry was about available or busy peer

During the preprocessing stage the file was modified to add another column containing concatenated PeerID and Timestamp in order to improve the performance of database queries on the table (that pair was used to join tables).

After that, the file was imported to the GOSSIP table and another table was generated called MSG_COUNTS which consisted of the concatenated column and number of messages that were spread which were mentioning that pair of PeerID and timestamp. Let us recall that in the reference implementation of the gossip peer-to-peer a peer can increase its timestamp value without changing state information about itself. Such timestamp update will not be propagated to the network. This table was used to filter out these timestamp values when no messages were sent.

During the next stage the data was processed and the result files were created: DISSEMINATION_RANGE, DISSEMINATION_TIME and DISSEMINATION_PROGRESS. Each file contained the result from different analysis:

- DISSEMINATION_RANGE measured how many peers were aware of the given message entry. That is the number of distinct peers that received the message. The range doesn't contain the peer that the message entry was about. For each pair of PeerID and timestamp value that was in MSG_COUNTS table the range number was computed.
- DISSEMINATION_TIME measured how much time a given message was circulating in the network. For each pair of PeerID and timestamp value the interval between the reception of the first and last message was calculated. The time was taken from the local clock of the peer that received the message.
- DISSEMINATION_PROGRESS examined the process of spreading the message. The result file is similar to the DISSEMINATION_RANGE but contains the number of peers that were aware of the message during the first X milliseconds after the first message was received. I generated statistics for the milliseconds ranging from 500 to 8000 in the intervals of 500 ms. Because of the complexity of the query I limited generating these statistics to one randomly chosen peer. This is important because the average range at the end of time interval can differ slightly from the average counted for all the peers in the DISSEMINATION_RANGE file.

Each output file was generated in the CSV format. The SQLite client provided a convenient way to achieve this goal as it is capable of redirecting the output of the queries to files and choosing the output format. The CSV format is convenient because it is possible to load that file again to another table in SQLite and continue processing the data or import the data to spreadsheet and generate the report.

The last phase of the experiment was analyzing results in the spreadsheet. To achieve this goal I created a generic spreadsheet where graphs were generated automatically after the data from CSV files was imported.

Our experiment is able to realize all of the three presented goals because:

- We can analyze messages that were circulating in the network during each simulation and conclude if the network keeps the communication constrained conforming to the user specification. The vast range of scenarios analyzed additionally supports our conclusions.
- We can observe the performance of the network in each scenario so as to analyze the speed of dissemination of the information in the network. By doing this we can conclude if applying different configuration parameters has impact on the speed of the dissemination and thus whether it is possible to adapt the network parameters to user applications using different level of granularity of work items.
- The choice of scenarios reflecting different network configurations allows us to analyze each parameter of the network in isolation. Thus, it is possible to make conclusions about the impact of each parameter of the network on the overall performance measures of the proposed peer-to-peer protocol.

6.2.2. Interpretation of the Results

This section will present the results from experiments. Because of the number of them some of them will not be presented here. The partial results presented here are to outline some of the properties of the proposed solution.

Definition 1 *Let X-Y scenario means the set of simulations that were performed with the minimum message size X and maximum message size Y.*

Global Statistics

In every scenario I measured the total number of messages that were circulating in the network. Because in all cases the simulation was running for 10 minutes and there were 140 hosts it is easy to calculate the average number of messages received by one peer during one round (the interval between gossiping).

In all cases the number of distinct messages depended only on the number of acquired resources. This is a reasonable observation because the size of communication channel should only influence the speed of the dissemination of the information. For 50% resource acquisition scenario there were 7,000 distinct messages on average, for 75% there were about 10,000 messages and for 100% acquisition about 13,000 messages were recorded. The interesting statistic is the usage of the allocated communication channel as it can indicate in which situations the given communication constraint was sufficient and where it was delaying the spread of the knowledge.

For the most constrained scenario in which the message size differed between 1 and 5 (I will be referring to this as 1-5 scenario - see definition 1 in section 6.2.2) the channel was completely saturated. One reason for that is that the the channel usage didn't follow the increase of the resource acquisition parameter. For 3 neighbours scenario and 50, 75, 100 % of resource acquisition the channel was used in respectively: 78, 79 and 80 percent. The usage didn't reach 100% for a couple of reasons. First, the peers in the simulation were not started simultaneously on the cluster. Each node of the cluster had 7 peers to run. The initialization time in this scenario when 7 JVMs was started in ProActive was considerable and it was experienced by me when I looked at the output of the simulation. The longest part was initializing the local RMI registry for each peer. Although the command to start peers was launched at the same time, the time before all of the peers were initialized completely could last up to 2 minutes and the initialization time for peers differed substantially which could have influence on the results. Second, the logic of the peer-to-peer network makes it almost impossible to use the channel in 100%. If the entry from the message is not new for the peer that is receiving it, it will not forward that entry further. In our 1-5 scenario each peer would receive every second 15 entries on average if the channel was used in 100%. The peer could forward 5 of these entries and thus use its outbound channel fully only if there were 5 entries of those 15 received that were new to that peer. It can happen often but might not always be the case.

The 80% usage of the channel is a sign of high saturation. The only results that were higher than this was during 5-5 scenario (where theoretically the usage should be always 100%) where the usage for 3 neighbour scenario was 84% in all of the cases and for 5 neighbour case ranging from 80 to 81 percent. When we compare it to the scenarios where larger messages were allowed we can see that the difference between usage of the channel for different resource acquisition variants is growing. In case of 1-10 scenario this was for 3 neighbours from 67 to

74%; for 5 neighbours: 73-77%. In 1-20 scenario for 3 neighbours: 50-65%; for 5 neighbours: 51-70%. And in 1-40 scenario for 3 neighbours: 28-47% and for 5 neighbours: 29-50%. We can see that in the last case the upper limit of the message size was sufficient for the knowledge to spread naturally without inhibiting the communication. It will also be seen in other results.

The Impact of the Maximum Message Size

In this section the influence that maximum message size parameter has on the performance of the network is analyzed. With the maximum message size parameter one can limit the communication below some threshold. Therefore, on average the usage of the bandwidth by the network will be below some level. This behaviour is in contrast to the previous implementation of the peer-to-peer network where a peer could be flooded with messages when the requests were sent often. Thus, the proposed solution gives the user more predictable consumption of the bandwidth. On the other hand, if the maximum message size is set too low we may suspect that the knowledge will disseminate slower than in the optimal conditions. Effectively this could reduce the usefulness of the peer-to-peer network for tasks that are very fine grained.

The results from simulations proves our suspicions. The results differ greatly depending on the maximum message size parameter. This parameter has the biggest impact on the performance of the network. The role of the 'user' peers in the experiment scenario was not specifically coarse grained. The requirement for the network to disseminate the new state (which is changing every 10 seconds) in 10 seconds is a very ambitious goal and needs a lot of communication. In real life situations the requirements put on our network will be much smaller. The experiments revealed that there are two barriers that our network could face - which one depends on the value of the maximum message size parameter and the workload. One of the barriers is the channel size which appears when the maximum message size is set too low to handle the workload. The second one appears when the communication channel is big enough and the maximum message policy implementation doesn't have to be applied too often. In such situation we reach the theoretical barrier that comes from the exponential growth of the dissemination and the maximum speed of dissemination depends on the number of neighbours we are sending gossip to.

In Figure 6.2 we see the results of the simulation launched with different maximum message sizes. Let us recall that range means the number of peers that received a given message entry. From the graphs we can notice the difference in performance depending on the value of the maximum message size. When the message size was limited to 5, the network had difficulties in disseminating the new state in time. The average number of peers that were aware of the given entry was below the half of the network. The question is if that percentage is sufficient. The answer is - it depends. If there is one user that needs 100% of the resources the answer will be of course no. But when there are 100 users and each of them only wants to acquire 1% of the resources, the probability that all of them achieve the goal is much higher due to randomness of the dissemination of information (each peer knows about different available peers). In other scenarios higher values of the maximum message size allowed knowledge to spread more broadly in a given time constraint. For the size of 10, the average message was acknowledged by 80 peers. For 20, that number was about 110. And for 40, the number of peers that the message was spread to equaled to 120. The curves that are expressing the progress of the dissemination are growing slower in the course of time. For the scenario with the maximum size of 5, at the end of the plot the curve is growing faster than it is in the

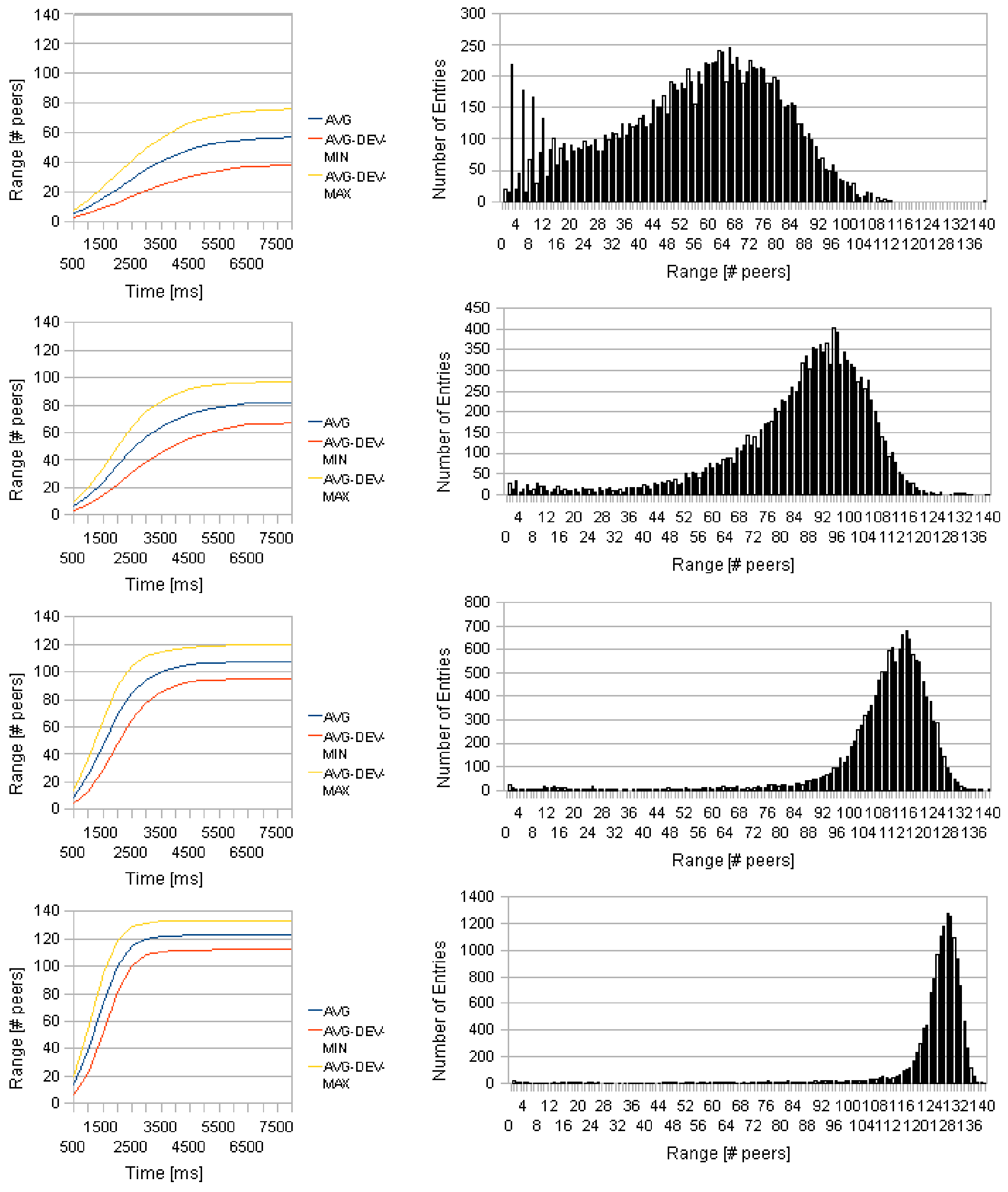


Figure 6.2: Each row represents a different scenario from top: 1-5, 1-10, 1-20, 1-40. Each case was launched using 3 neighbours to forward gossip and 100% of resource acquisition. The first graph shows the growth of the average range size during time [in ms] along with average absolute deviations. The second graph presents the distribution of the range across all messages.

other scenarios because the knowledge was still not spread to most of the peers. However, it is obvious that it won't be growing much after 10 seconds as it is highly probable that each peer will change its state (as we acquire 100% of resources). When the maximum message size is 40, the curve stops growing after about 5 seconds after the first entry of the message was received indicating that the dissemination is over. This also indicates that the limit of 40 for the message size is sufficient for our workload to update the state of the network. We can see it in the distribution graph for this scenario where the most of the entries' ranges are located between 120 and 140 peers with a high and a narrow peak at 130. The smaller the maximum message size parameter is the wider the peak of frequent range values.

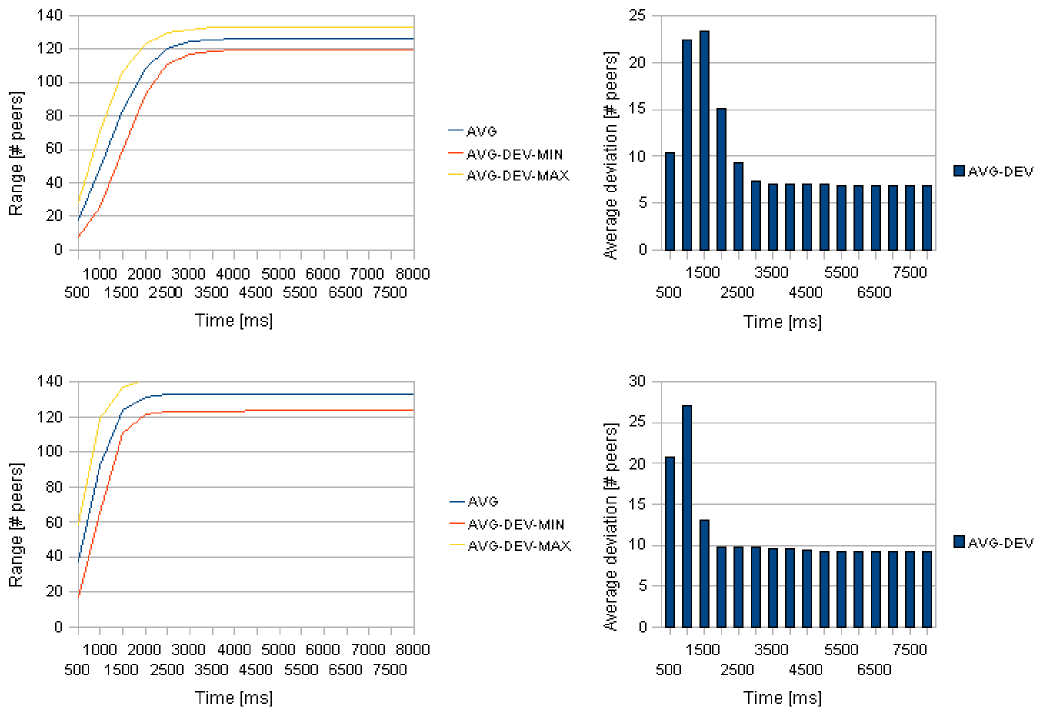


Figure 6.3: The progress of the range and the progess of the average deviation graphs. The first row covers 1-40, 3 neighbours and 50% of acquisition scenario. The second is the same but 5 neighbours are used.

From the previous graph we see that the limitations of the network usage can create a bar-

Scenario	Number of acquired resources		
	50%	75%	100%
1-5, 3 neighbours	81,89	63,95	55,04
1-10, 3 neighbours	109,27	94,14	85,25
1-5, 5 neighbours	82,8	81,93	80,14

Scenario	Number of acquired resources		
	50%	75%	100%
1-10, 3 neighbours	109,27	94,14	85,25
1-20, 3 neighbours	124,37	116,75	109,71
1-10, 5 neighbours	128,05	119,72	108,68

Scenario	Number of acquired resources		
	50%	75%	100%
1-20, 3 neighbours	124,37	116,75	109,71
1-40, 3 neighbours	126,75	125,27	124,32
1-20, 5 neighbours	132,76	131,77	128,36

Table 6.1: The average range increase after increasing maximum message size and after increasing the number of neighbours

rier for the spread of the information. The other barrier comes from the theoretical limit of dissemination of information. The entries are disseminated exponentially. It means that the number of neighbours can have impact on the theoretical limit of dissemination. The experiments proved this observation. If we compare the progress of the range in two extremely optimal situations: the maximum message size is 40 and we acquire 50% of resources using 3 neighbours to disseminate information and the same scenario with 5 neighbours the difference in the curve shape comes from this theoretical limit as network resources are not limiting the dissemination. This can be seen on the Figure 6.3. As a consequence, we can see a faster drop in the deviation from average range in the scenario with 5 neighbours. The high value of deviation means the dissemination is in 'process' and some messages are spread and some are still spreading.

The Impact of the Number of Neighbours

We know that the number of neighbours influences the speed of dissemination when there is enough bandwidth for spreading the knowledge naturally. However, the number of neighbours is increasing the network consumption of the peer. In our case, increasing the neighbours parameter from 3 to 5 increases the bandwidth required by 66%. It is interesting to compare the performance of the network when the bandwidth allowance was increased by changing the maximum message size parameter with the case where the bandwidth allowance was increased by increasing the number of neighbours. In our configurations the scenarios where the maximum size of the message was doubled are taken into account as it is not possible to compare increasing number of neighbours with a case where the bandwidth was also increased by 66%. In the table 6.1 there is a comparison of the final range values in some scenarios.

We can see in the first table that there is clear advantage of having larger maximum message size. The interesting thing is that the values for the 5 neighbours scenario are almost not falling. For example in the next table, in all cases the range is falling. I think the explanation for this is that the low value of the message size inhibited the maximum value of the range for 5 neighbours scenario. It looks like the 5 neighbour scenario reached some kind of maximum which it is not able to beat. What supports this hypothesis is that the value is not decreasing for the increasing amount of workload when the rate of acquisition is adjusted.

In the next table the similar comparison is presented but the allowed message sizes were doubled. In this situation the 5 neighbour scenario behaves competitively with the increased message size variant and because it is using 66% more resources and not 100% this variant is more efficient.

In the last table the allowed message size was doubled again. Here, in the 40 maximum message size we can see that we reached the theoretical limit of disseminating information because the range values for this situation are barely falling. The 5 neighbours scenario behaves better than doubled maximum message size one but we can see an increased drop of the performance for 100% acquisition case which indicates that this case is slightly below the ideal limit for the dissemination with 5 neighbours. However, in this scenario we are still outperforming the doubling the maximum message size strategy.

To sum up, we can see that an increased number of neighbours can give a performance boost for the network if it is not too overburdened. When the maximum message size policy is applied in a too strict way there is no advantage of increasing the number of recipients of our gossip.

The Impact of the Minimum Message Size

The minimum message size helps new peers that are joining the network in populating its state. Let us recall that a new peer that joins the network has an empty state. When it contacts new peers it can later receive gossip from them. However, initially this gossip covers only the recent changes that happened in the network. Thus, when nothing happens a new peer cannot get any information from any peer. The minimum message size policy helps new peers to populate the state quicker as it is providing them with entries already existing inside peers. During the simulations I wanted to test how the behaviour of the network changes when this parameter is increased.

The statistic that changed the most by increasing the minimum message size was the distribution of the time that a given entry was spreading in the network. This can be easily explained by the fact that the 'old' entries from peers were 'reused' to compose the message (possibly the entries that were for a very long time in the state). That increased the total time of the message that was exchanged in the network. It can be seen on the figure 6.4.

Because of the change of the time distribution especially in the situation where the communication is heavily constrained one may fear that the performance of the network is compromised when using the increased value of minimum message size. However, the statistics prove us wrong - the performance is the same. In table 6.2 we can see differences between average range of the message. The values are almost the same when the minimum message size changes. Thus, the increased value of this parameter doesn't influence the performance of the network.

Scenario	Minimum message size	
	1	5
Max msg size: 5, 3 neighbours, 75% resources	63,95	65,26
Max msg size: 10, 5 neighbours, 75% resources	119,72	119,79
Max msg size: 40, 5 neighbours, 75% resources	134,04	133,56

Table 6.2: Range size in the selected scenarios

These two facts combined lead to another question. The difference in the time distribution is because the 'old' entries are resent in the future. But this doesn't have a great impact on performance because it is comparable with the situation when a little number of such entries is sent. We can conclude that the most of such resent messages aren't that important and are not sent further by the recipient (most probably because it has it already in its state). However, sending additional messages even if it is within the message size limit increases usage of the bandwidth. The results show that increasing the minimum message size doesn't increase the average message size much. For the scenarios listed in table 6.2 the message size is increased respectively by: 6%, -1% and 10%. The -1% increase is because we are comparing two different simulations. During these, the initialization time for the peers may have been different changing the total number of messages sent. Increasing the minimal number of entries in each message doesn't have any negative impact on performance of the network and thus can safely be used to increase the speed of initialization of peers.

6.2.3. Summary

The experiment gave answers to the three questions mentioned in the goals. Specifically:

- The user has control over the bandwidth usage of internal communication of the proposed peer-to-peer network. The experiments showed that the number of entries received by each peer on average didn't exceed the maximum defined by the parameters of the network. Of course, the number of entries sent by each peer also didn't exceed the maximum because of the policies that our solution is using. Because the size of each message entry is similar we can conclude that by adjusting network parameters a user has control over the average network bandwidth usage of the peer-to-peer network.

- The proposed protocol for peer-to-peer communication can be adjusted to address different needs of user applications such as different work item granularity and different ways of acquiring resources from the network.

  First, the value of ranges even for heavily constrained scenarios showed usefulness of the network in scenarios where the resources aren't acquired and released too often as our scenario was typical of the fine grained computations. The average number of message entries received by a given peer within a gossip interval ranged from 11 to 100 depending on the configuration of the network. The average number of peers that were aware of a given entry ranged from 55 to 134. From this we can conclude that different network parameters have an impact on the range value which depends on the speed of dissemination in the network. Thus, the speed of dissemination can be adjusted by

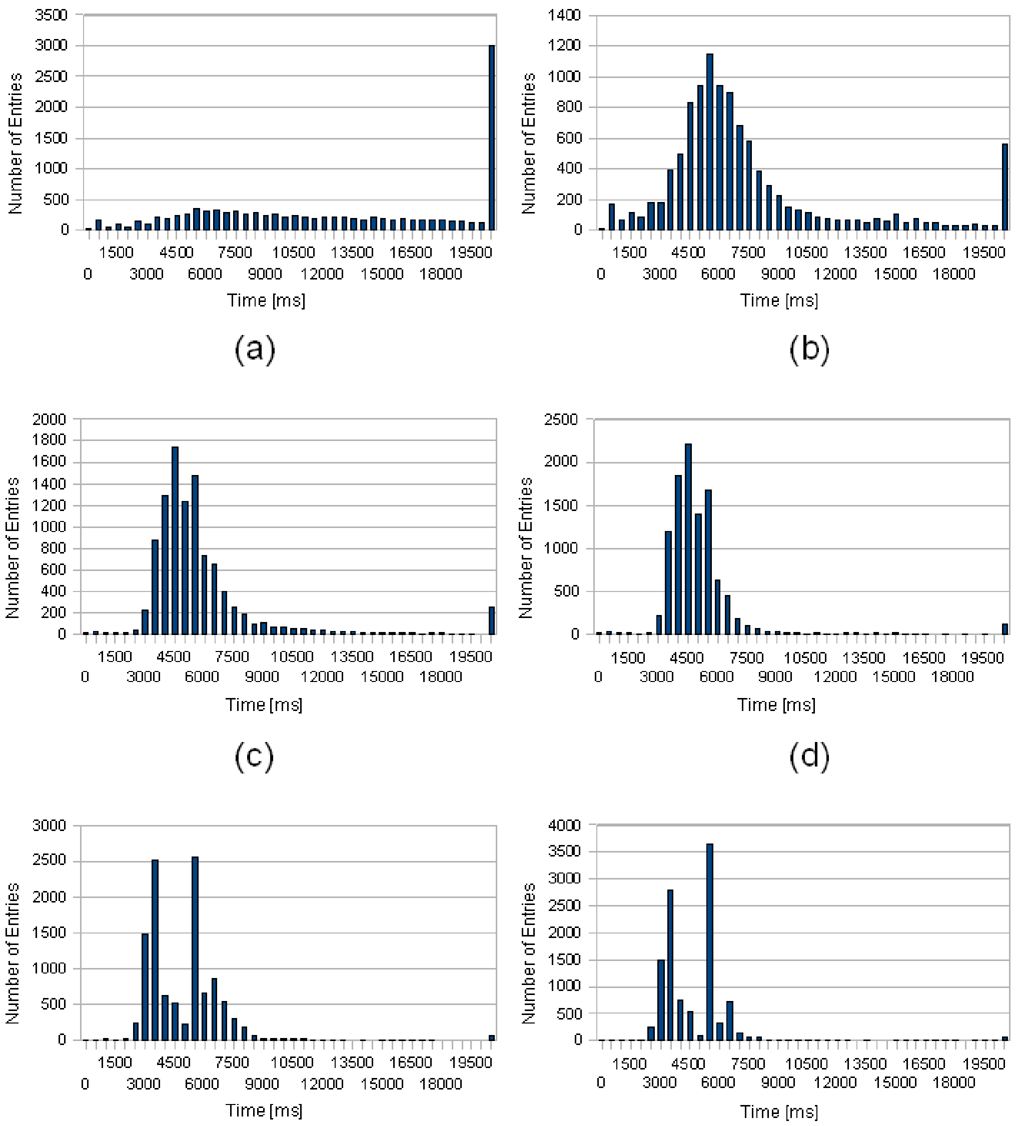


Figure 6.4: The plots show the distribution of amount of time the given entry was exchanged in the network in the following scenarios: (a) 5-5, 3 neighbours, 75% resources, (b) 1-5, 3 neighbours, 75% resources, (c) 5-10, 5 neighbours, 75% resources, (d) 1-10, 5 neighbours, 75% resources, (e) 5-40, 5 neighbours, 75% resources, (f) 1-40, 5 neighbours, 75% resources The X axis represents time and Y axis number of entries.

modifying network parameters of the peer-to-peer network addressing different needs of user application as for work item granularity. The experiment proved that faster dissemination comes at a cost of less efficient communication usage.

Second, changing parameters of the peer-to-peer implementation can address different ways of acquiring resources from the network by user applications. For example, even 40% average dissemination is useful in some computational scenarios where a single peer doesn't have to acquire the bulk of the resources available in the network and many peers take part in acquiring the resources. In such situations heavily constraining the communication of the peer-to-peer network saves the network bandwidth without the impact on the user applications performance.

- The experiment revealed the following relations between network parameters:

  The maximum message size parameter has the greatest impact on the speed of the dissemination. The increase of this value always leads to the increased performance of the network in terms of the average range of dissemination. Furthermore, the average deviation from this statistic is smaller when the maximum message size grows.

  An increased number of neighbours leads also to the increased performance. However, the final outcome of the change depends on the current workload of the network. If it is heavily overloaded then the increase of the number of neighbours might not lead to any gain in performance. But when the message size range is better adjusted to the usage style of the network we can experience performance boost achieved with lower cost. It is because we can achieve the same performance boost by increasing the overall communication channel size less than in case of increasing the maximum message size. Furthermore, by increasing the number of neighbours we can increase the theoretical maximum speed of dissemination of the knowledge in the network.

  The minimum message size policy that was introduced in order to help new peers to populate their state doesn't interfere with the performance of the network. Although it can be seen that some of the old entries from the state of peers are resent to the network, the overall range of dissemination and communication channel usage are not affected by this.

Chapter 7

Conclusions and Future Work

7.1. Conclusions

Distributed computations are entering the world of desktop machines where the unused CPU cycles of these machines can be used to provide computational resources for performing large computations. We could see with the BOINC [1] and its predecessor project Seti@Home how successful this model can prove to be. Now, the computations on desktop machines are on demand in the world of corporations which can use its in-house desktop machines to do the computations adding value to their business.

The proposed solution enables the desktop machines to form an unstructured peer-to-peer network driven by a gossiping protocol to efficiently allow them to be used as a computational resources without one single point of failure. The ProActive framework which the proposed solution is contained within allows to use the resources acquired from the proposed peer-to-peer network in a flexible way as it is not implying any distributed computation scheme on application developers (such as Master/Worker scheme in BOINC). Therefore, it fits perfectly to be applied in the corporation environment where different types of communication schemes can be efficiently exploited.

The proposed solution provides a peer-to-peer computational network which can be customized by the administrator according to the requirements of applications that rely on the resources acquired from the network. The administrator can change the minimum and maximum message sizes, number of recipients and the frequency of the gossip to adjust the network to the grain of computation of the application. These parameters combined can define the maximum average traffic that the network will use for the internal communication. The network itself strives to use the allocated resources efficiently which was shown in Chapter 6 where the behaviour of the network was analyzed given different parameters during the experiment. One of the indicators of the efficiency was the drop in the allocated network use when the allocated communication channel size was enough to spread the knowledge naturally (using exponential distribution). The experiments also revealed relations between these parameters of the network and their impact on the performance of the network. These observation could be used to extend the features of the proposed solution presented in the next section.

Finally, the concept of a new approach in ProActive peer-to-peer network lead to design of a new object model of the peer-to-peer framework. The proposed solution has a unique design that allows to easily extend the solution and modify the strategies without fear of making the code unstable. Moreover, the proposed design lead to simplification of the concurrency concerns which were typical of applications where some number of active objects are communicating with each other such as existing peer-to-peer implementation. The decoupling was done also at the low level allowing different strategies to be written and easily integrated with the existing parts of the system only by changing the class used for implementing a given interface. This allows to change the small part of solution without threatening the correctness of the whole peer-to-peer component.

Unfortunately, due to changes in the architecture of the ProActive library that happened while this thesis was being written the proposed gossip based peer-to-peer implementation is still not included in the current major release of the library. However, apart from including this package in the ProActive library the author is considering releasing the implementation as a standalone open-source project.

Apart from the design and development the implementation of the protocol for peer-to-peer computations I also developed software that tries to achieve real integration of Windows desktop machines with the ProActive library - ProActive Agent. One of the way that this integration is provided by the ProActive Agent is by collaborating in the proposed in this thesis peer-to-peer network. This application is aimed at providing the machine's computational resources in the spare time of its user(s). The Appendix A provides the description of this software.

7.2. Future Work

The gossiping approach opened a lot of new opportunities in improving the performance of the communication algorithm. Because of limited time I wasn't able to perform all of the experiments that raised interest. Here is the list of possible extensions of the protocol which may improve the properties of the Peer-to-Peer network:

Detection of Patterns in the Peer-to-Peer Network Communication

During exchanges of gossip between peers one can observe patterns in the communication. One possible point of observation is the size of each message. We can detect situations when message is capped due to excessive size. We also can find out the moment when the message size decreases because the knowledge of recent changes is spread out and we would be able to pass information that was capped earlier and not sent to anyone. The other interesting stage is the moment when a peer joins the network and receives first messages from existing members. By detecting these situations we can customize the behaviour of the peer to optimize its current activity according to the detected state.
For example, we could cache the entries from the gossip that we were unable to forward due to the reduction of message size. We could maintain such buffer (with limited capacity) and during the time when there is free space in the message we could add these entries to assure that recent knowledge is spread.

Dynamic Number of Recipients of the Message

If a given gossip exchange is sent to a greater number of recipients the knowledge contained in that message will be disseminated faster (i.e. in fewer number of gossips everyone will be aware of that state change). This behaviour could be exploited without impact on bandwidth use when the size of the message is relatively small. We could try to send this message to as many peers as possible while still meeting the bandwidth limits as opposed to filling the message with the entries from our own state (as it is done now). I believe it is possible to combine this strategy of increasing number of recipients with the one described in the previous section and try to apply increase of number of recipients when the buffer containing entries that were not previously disseminated is empty or near to empty.

Dynamic Number of Maximum and Minimum Message Size

If we apply statistical analysis to the message flow in the Peer-to-Peer network we can see that the fixed limits on the message size artificially reduces the possible range of message size. Because of that the factor that is variable is the average message size. This value is of course between minimal and maximum message size but we cannot control it directly. There could be alternative approach in which we could try to guarantee a fixed average message size with changing minimal and maximal message size factors. The Peer-to-Peer infrastructure could track the distribution of message sizes and increase the message window size to encourage bigger exchange of state in situations when the big exchanges appear rarely with smaller ones happening often. In the reverse situation when the big exchanges start to dominate the communication we could reduce window size to reduce the message size and thus to try to achieve our goal of a fixed average message size possibly delaying the state dissemination in time.

Compression of Messages

This optimization would allow to use the same bandwidth to send more information than it is now possible. The currently used RMI protocol is not very efficient in terms of the message size. The proposed solution is to intercept communication that is coming through RMI protocol and before sending a message compress its contents and before passing the received message to the RMI infrastructure decompress it.

Bibliography

[1] David P. Anderson. BOINC: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, November 2004.

[2] David P. Anderson and John McLeod VII. Local scheduling for volunteer computing. In *Workshop on Large-Scale, Volatile Desktop Grids (PCGrid 2007) held in conjunction with the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Long Beach, USA, March 2007.

[3] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[4] Laurent Baduel, Francoise Baude, and Denis Caromel. Efficient, flexible, and typed group communications in Java. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 28–36, Seattle, 2002. ACM Press.

[5] Laurent Baduel, Francoise Baude, and Denis Caromel. Asynchronous typed object groups for grid programming. *International Journal of Parallel Programming*, 35(6):573–614, 2007.

[6] Franoise Baude, Denis Caromel, Fabrice Huet, Lionel Mestre, and Julien Vayssire. Interactive and descriptor-based deployment of object-oriented grid applications. In *HPDC*, pages 93–102, 2002.

[7] Kent Beck and Martin Fowler. *Planning Extreme Programming*. Addison-Wesley, 2000.

[8] Denis Caromel, Alexandre di Costanzo, and Clement Mathieu. Peer-to-peer for computational grids: Mixing clusters and desktop machines. *Parallel Computing Journal on Large Scale Grid*, 2007.

[9] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Object*. Springer-Verlag, 2005.

[10] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] Imen Filali, Fabrice Huet, and Christophe Vergoni. A simple cache based mechanism for peer to peer resource discovery in grid environments. In *CCGRID*, pages 602–608, 2008.

[12] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994.

[14] Gnutella. http://www.gnutella.com.

[15] Howard Gobioff, Sanjay Ghemawat, and Shun-Tak Leung. The Google File System. In *19th ACM Symposium on Operating Systems Principles*, Lake George, NY, October 2003.

[16] Fabrice Huet, Denis Caromel, and Henri Bal. A high performance Java middleware with a real application. In *SuperComputing 2004*, Denver - USA, October 2004.

[17] R. Greg Lavender and Douglas C. Schmidt. Active object. an object behavioral pattern for concurrent programming. *Pattern Languages of Program Design 2*, 1996.

[18] M. Taufer, D. Anderson, P. Cicotti, and C.L. Brooks III. Homogeneous redundancy: a technique to ensure integrity of molecular simulation results using public computing. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) Heterogeneous Computing Workshop*, Denver, USA, April 2005.

[19] Bernard Traversat, Ahkil Arora, Mohamed Abdelaziz, Mike Duigou, Carl Haywood, Jean-Christophe Hugly, Eric Pouyoul, and Bill Yeager. Project JXTA 2.0 super-peer virtual network, May 2003.

Appendix A

ProActive Agent

The efficient utilization of desktop machines as computational nodes requires software that will enable the use of local resources for deployment of distributed applications. Since most of the desktop machines have a Windows operating system installed, from now on I assume that desktop machine has that operating system installed. In the context of ProActive project, the user of the desktop machine should be provided with an easy to use tool that will register that machine in the computational network upon some events like inactivity of user. Although it is possible to launch start scripts for ProActive manually, this implies a lot of limitations. First, user may forget about launching such script when he or she is inactive. Second, stopping such script might not shut the ProActive runtime down correctly. For example, if a *node* that was used in computations launched another native Windows process, that process will not be killed since Windows operating system does not provide a direct way to kill processes recursively. As a result CPU of the machine will still be heavily loaded even though the user becomes active.

I have developed an utility program that makes this process easy. ProActive agent is a software that act as Windows system service and is able to expose computational abilities of the system in the form of ProActive *nodes*. Then, it is possible to join the ProActive peer-to-peer computational network, register in ProActive Resource Manager (which will be described in the next section), or just create a local *node*. The agent is configurable by the XML configuration file where the administrator can define time intervals during which the system should become part of the network. The example configuration is to activate the agent outside the office hours on the desktop machines in the company. When it is done, all of the desktop machines will automatically become available as a computational resources during nights on week days and throughout the whole weekend to be used for distributed computing.

A.1. Goals of the Software

The system has several goals and constraints:

- The system should be able to register a local ProActive *node* in :
  - a ProActive peer-to-peer computational network
  - (or) in ProActive Resource Manager
  - (or) local RMI registry

for a given amount of time.

- The system should be configurable to schedule the above activities periodically or on user's inactivity.
- On disconnection from computational network all of the processes that originated from ProActive runtime launched by ProActive Agent should be killed.

A ProActive Resource Manager is an extension to ProActive library that is able to maintain a list of available ProActive *nodes*. These lists can be used later by an distributed application to acquire necessary resources. Thus, it is another way of obtaining computational resources along with already presented peer-to-peer computational network and deployment descriptors. This way differs from the two above because we deal with single point of failure (the Resource Manager is running on one machine) unlikely to the peer-to-peer solution and we have additional layer of indirection when compared with deployment descriptors. The Resource Manager is used often with software called ProActive Scheduler which makes it possible to define tasks to be completed and to run them on the chosen available machines. There is a GUI client for ProActive Scheduler where a user can just click to view available resources and observe the progress of computations.

A.2. Software Architecture

The agent is divided into following parts:

- ProActive Agent System Service

  ProActive Agent system service is the main component of the software. It works as a system service and is responsible for reading the configuration file and for triggering the ProActive runtime upon events. When the service is started the Agent reads configuration and sets up the timers used for triggering calendar events. It also listens to external commands which can trigger starting or stopping ProActive runtime as well. The configuration architecture is presented in the diagram A.1.

  The general configuration distinguishes the following components:

  - Events - on which the Action will be taken. The event symbolizes some of the circumstances upon which the ProActive runtime will be started. Currently, there is one type of events called Calendar Events. Each calendar event specifies the time when the runtime should be launched. Each such event contains:
    - Day of the week, hour, minute, seconds when the event should start. The system assumes that the events are recurring every week.
    - The duration of the event. Again, the user specifies number of days, hours, minutes and seconds.
  - Action - an object unique for the configuration describing the activity that will be initiated during duration of the event. There are three types of actions:
    - P2PAction - which starts a local peer and joins the peer-to-peer network.
    - RMAction - which starts a local ProActive *node* and registers itself in Resource Manager

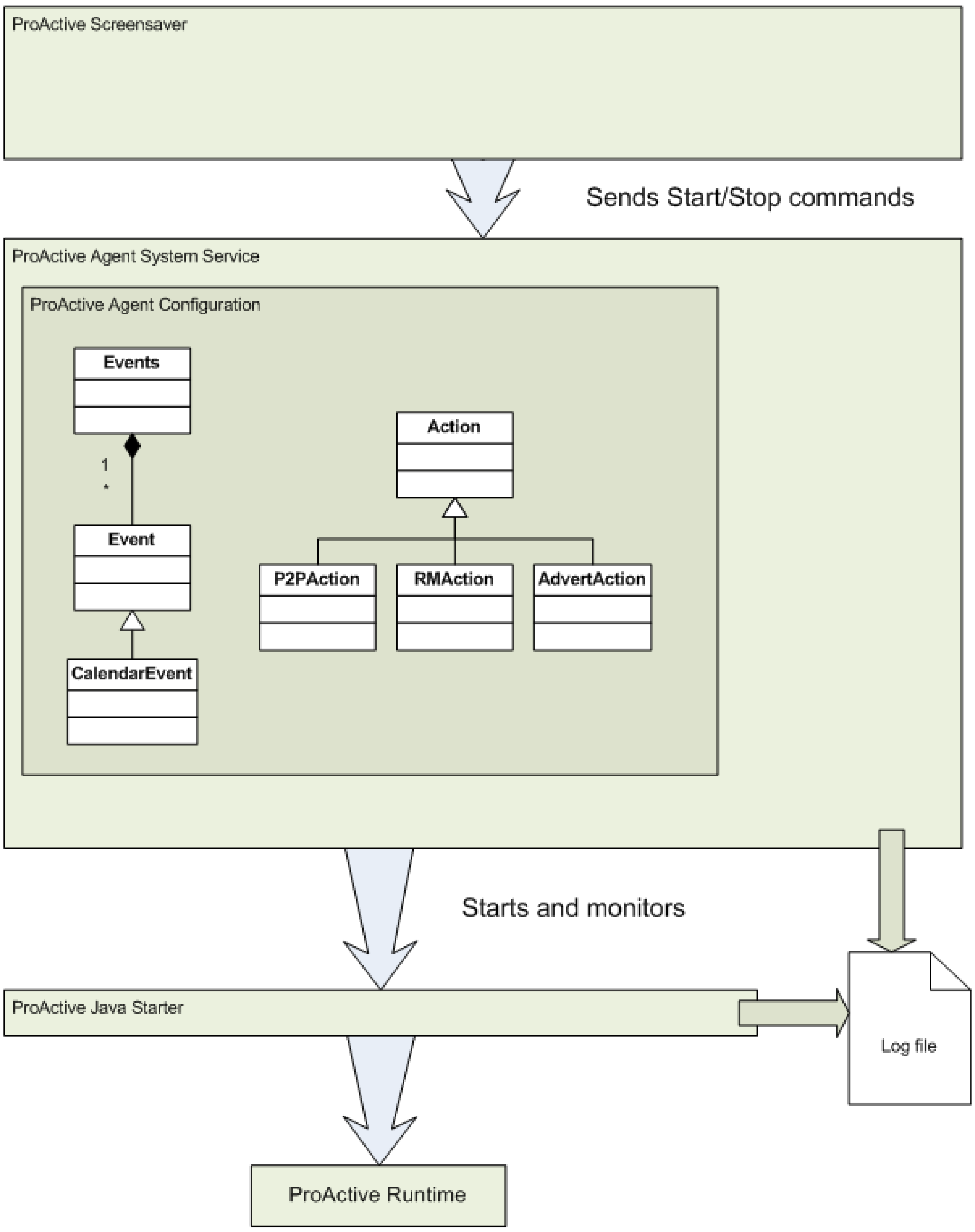


Figure A.1: The architecture of ProActive Agent

    * AdvertAction - which starts a local ProActive *node* and advertises itself in a local RMI registry

  As a system service, ProActive Agent works inside the environment called Service Control Manager. Service Control Manager is a RPC server that is maintaining all of the System Services in the Windows operating system. Each such service responds to the default service commands (such as Start, Stop, Pause, Resume Service requests) and custom commands containing a numerical code that can be send by other Windows applications. In our case, we are using custom commands to initiate and stop Action to provide support to external application. This feature will be exploited by ProActive ScreenSaver in order to start Action on user's inactivity.

  Apart from all of above, the service is also responsible for monitoring the created ProActive instance and logging the content of standard output of that instance.

- ProActive Java Starter

  The ProActive Java Starter is a small Java application that is called by the ProActive Agent Service when the Action is initiated. The application accepts parameters that specify the desired activity to be started. The application can:

  – Start the local ProActive peer-to-peer peer and join the existing network. The service has to provide 'first contact' list to be used.
  – Start a local *node* and register itself in Resource Manger. The service specifies URL of the Resource Manager instance.
  – Start a local *node* and register itself in local RMI registry. The service can optionally provide the name of the *node*.

- ProActive Screensaver

  The ProActive Screensaver is a very simple application that can be launched by Windows when user is inactive in the system for a specified amount of time. Upon launch it sends command to ProActive Agent Service to start the Action. Like usual screensaver, it will terminate when the user moves mouse or types something on the keyboard. At the termination of Screensaver, it issues a command to end the action in ProActive Agent System Service.

A.3. Communication

This section will provide an overview of communication happening inside ProActive Agent component. This involves sending commands to the Service as well as launching and killing the created processes by the service.

ProActive Agent Service allows to run at most one ProActive instance that originated from the service. Thus, while the action is still happening (it has not ended yet) all of the subsequent start commands sent any program including calendar events will be ignored. This however introduces a new problem. Let us consider the following scenario: a screensaver started the action and while it was running the calendar event was triggered. Immediately after that a screensaver is shutting down. Under this semantics it will shut down the ProActive runtime preventing from using the resources possibly for next couple of hours which was probably set in calendar event. To avoid such scenarios we introduce another additional assumptions about commands: each command sent to the service has the same priority. Moreover, the commands accumulate. That means that if two start commands were sent to the service, then two stop commands will be required to stop it. Thus, the implementation keeps track of how many commands has been sent using counter that is indicating the number of stop commands required to stop the whole ProActive runtime.

The other responsibility of ProActive Agent Service is to monitor the running instance of ProActive runtime. Apart from counting number of necessary stop commands the service has to log the output of the ProActive instance. This is achieved by creating a monitoring thread that is periodically reading from redirected output of the ProActive child process.

The last and the most important part of the communication of ProActive Agent is management of the created ProActive instance. The process creation is easy as a batch script sets up the environment and starts the ProActive Java Starter component. The challenging part is shutting down the instance. The main problem is that Windows operating system doesn't support an explicit API for killing processes recursively. What is more, in many works one can read that Windows operating system doesn't support hierarchy of processes making it almost impossible to track the hierarchy of processes. The problem has been solved by using extended API provided in a library that is partly documented by Microsoft and is called Native API (it is not Windows API). It is possible to access an internal structure of processes from one of the calls available in this library and access the field called InheritedFromProcessId which is similar to PPID in Unix operating systems. Having this information it is enough to perform kill of the whole process tree. I created a dynamic linked library that is used by the ProActive System Service using Platform Invoke mechanism (the service itself is written using .NET framework). Apart from this, I created a command-line tool that is able to kill a given process recursively providing its PID as a parameter.

A.4. Realization of Goals

All of the software goals that were set are fulfilled:

- The system is able to register a local ProActive *node* with all of the method described by using ProActive Java Starter
- The system is configurable by an XML configuration file which will be described in the last section of this chapter. The configuration allows to start the ProActive runtime based on calendar events. Apart from this, the ProActive Screensaver is able to start ProActive instance on user inactivity.
- On disconnection from computational network all of the processes that originated from ProActive runtime launched by ProActive Agent are killed. It is done by using the unmanaged code that is calling a Native API routine detecting parent-child relationship by analyzing the list of the processes.

A.5. Installation

The one of the non-functional goals behind this application was to be able to be installed and configured silently without user interaction. The ProActive Agent has following system requirements:

- .NET 2.0 framework or newer
- prerequisites for ProActive

I assumed that the system has the prerequisites for ProActive already installed. Depending on the requirements it is usually only Java VM. The additional software required is then only .NET 2.0 framework. It can be installed silently by calling a setup with /q command line option and optionally specifying temporary installation files path and destination path.

The second stage of installation deals directly with installing the ProActive Agent System Service on the desktop. It is achieved with help of some .NET 2.0 utilities and there are batch scripts for full installation and uninstallation of ProActive Agent System Service in the system. They are not interactive and can be launched remotely.

The initial configuration of the service is in the form of XML file which is easy to distribute across machines.

The screensaver requires no configuration apart from copying necessary files into system folder and changing screen saver settings by editing the windows registry entry. The appropriate tool has been created.

A.6. Configuration

The configuration file specifies the settings of ProActive Agent Service. Following is the example of the configuration file:

```
<?xml version="1.0"?>
<agent>
<launcher>
<script location="c:\proactive\scripts\windows\p2p\agentservice.bat"/>
<workDir location="c:\proactive\scripts\windows\p2p"/>
</launcher>

<events>
<event type="calendar">
<start day="monday" hour="19" minutes="00" seconds="00"/>
<duration days="0" hours="12" minutes="00" seconds="00"/>
</event>
<!-- .. and other events -->
</events>

<action>
<rmRegistration url="rmi://cheypa:1099"/>

<!-- or
<p2pAction>
<peerList>
```

```
<peer>rmi://cheypa:1099</peer>
...
</peerList>
</p2pAction> -->

<!-- or <advert nodeName="Agent_Node"/> -->

</action>
</agent>
```

The file is divided into three sections:

- The launcher section which provides information about location of script that is launching ProActive Java Starter
- The events section which contains a list of calendar events. Each event is configured by providing the start time and the duration of the activity.
- The action section which describes what services to start upon event.