CHALMERS

Jalapeno – Decentralized Grid Computing using Peer-to-Peer Technology

NIKLAS THERNING

Master's Thesis
Computer Science and Engineering Program
CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Engineering
Göteborg 2003

Abstract

This report presents the Jalapeno grid computing system. Jalapeno is implemented using the Java programming language and uses peer-to-peer technology provided by the Project JXTA protocols. Project JXTA defines a set of communication protocols allowing any network connected device to communicate in a peer-to-peer manner.

The system consists of manager, worker and task submitter hosts. The task submitter submits a collection of tasks, called a task bundle, to be processed by the system to a randomly chosen manager. The manager splits the bundle into a set of new, smaller bundles which are forwarded to equally many, randomly chosen, other managers which repeat the process. Each manager have a small (<100) number of connected workers. During task bundle splitting the manager will, depending on its current load, reserve a number of tasks to be processed by its workers. Workers return the results to their managers which will send them to the task submitter.

The system is self configuring: hosts volunteering their computing power will at first become workers only but will eventually become managers if they can not find and connect to another manager within a certain time. The system offers a framework for the development of applications solving embarrassingly parallel type of problems but can also be used for other kinds of problems. The framework automatically splits the problem into smaller sub-problems to be distributed to workers. Furthermore, the system makes it extremely easy for users to participate and volunteer their computing power through the use of Sun's Java Web Start technology.

Keywords: JXTA, P2P, peer-to-peer, grid computing, distributed computing, distributed systems.

Sammanfattning

I denna rapport presenteras resultatet av detta examensarbete, grid-systemet Jalapeno. Jalapeno har implementerats i Java och utnyttjar peer-to-peer-teknologi utvecklad av Project JXTA. De kommunikationsprotokoll som definierats av Project JXTA tillåter i princip alla typer av datorliknande maskiner med en Internet-uppkoppling att delta i ett peer-to-peer-nätverk.

Jalapeno-systemet består av hanterare, arbetare och beräkningsbeställare. Beställaren skickar en samling beräkningar, som skall utföras av systemet till en slumpvis utvald hanterare. Hanteraren delar upp samlingen i ett antal nya, mindre samlingar vilka vidarebefordras till ett antal andra, också de slumpvis utvalda, hanterare. Varje hanterare hanterar ett litet antal (<100) arbetare. Under uppdelningen av en samling av beräkningar i mindre delar kan hanteraren, beroende på hur pass belastad den är, reservera ett antal beräkningar åt sina arbetare. Arbetaren skickar resultatet av en beräkning till sin hanterare som vidarebefordrar resultatet till beställaren.

Systemet är självkonfigurerande: nya värdar som deltar i systemet blir till en början arbetare men kan, efter en viss tid, komma att bli hanterare om ingen annan hanterare finns tillgänglig. Vidare erbjuder systemet ett programmeringsgränssnitt som förenklar utvecklandet av applikationer som löser problem vilka kan delas upp i mindre, oberoende delproblem. Med hjälp av detta programmeringsgränssnitt kan problemet automatiskt delas upp i delproblem. Systemet kan även användas för att lösa andra typer av problem. Slutligen utnyttjar Jalapeno Suns Java Web Start-teknologi vilket gör det extremt enkelt för datoranvändare att delta i systemet.

Nyckelord: JXTA, P2P, peer-to-peer, grid computing, distribuerade beräkningar, distribuerade system.

Preface

The work presented in this report is the result of a Master of Science thesis project in Computer Science and Engineering at the department of Computer Engineering, Chalmers University of Technology, Gothenburg, Sweden. This report as well as system implementation documentation, binaries and all the source code will be made available at `http://jalapeno.therning.org`.

First of all I would like to thank my supervisor Lars Bengtsson for valuable help and feedback and the people at the Computer Engineering department for providing office space and other facilities.

I would also like to express my deepest gratitude to the open-source community. Without their work on projects such as Linux, the NetBeans Java IDE, JXTA, LyX, LaTeX and of course all the extremely useful Java projects available from `http://jakarta.apache.org` this project would never have been possible.

I hope the ideas presented herein will inspire further development.

— Niklas Therning

Gothenburg, Sweden
December, 2003

Contents

Chapter 1

Introduction

This chapter gives a brief introduction to distributed computing. Furthermore the goals of this project as well as the necessary limitations are presented.

1.1 Background

Computers, whether they are office workstations, home PCs or PDAs, are becoming more and more powerful but are still mainly used for tasks requiring very little computation, such as word processing, leaving their CPUs idle most of the time. By connecting these computers in a network it is possible to utilize the otherwise wasted CPU time to solve problems in a distributed manner. The result is a distributed computing system. The definition of a distributed computing system will in this thesis be: *a system providing computing power by the distribution of work.*

Figure 1.1 shows a high-level view of a general distributed computing system and defines the terminology used throughout this thesis. First, the problem has to be split into a set of more or less independent *task*s which can be run in parallel. Each *worker* host receives a task, either directly from the *task submitter* or from a *manager* host acting as task distributor. The worker host executes the task and finally returns the result of the computation, again either directly to the task submitter or by use of the manager. The execution of a task may require direct communication with tasks running on other worker hosts as well as communication with the task submitter host or manager host.

The lower the computation to communication ratio is the more *fine-grained* the problem. Problems requiring no communication at all between tasks (the computation to communication ratio is infinite) are very *coarse-grained* and are also known as *embarrassingly parallel* problems.

Previous and existing distributed computing systems can be divided into a number of classes:

- In a NOW (Network of Workstations) the idle time of the workstations of a small office or university LAN provides computing power. Being on the same LAN enables the participating hosts to communicate directly without having to deal with NAT devices and firewalls. Authentication can be based on the normal login mechanisms used in the network. In most cases the network consists of a homogeneous set of hosts (all are

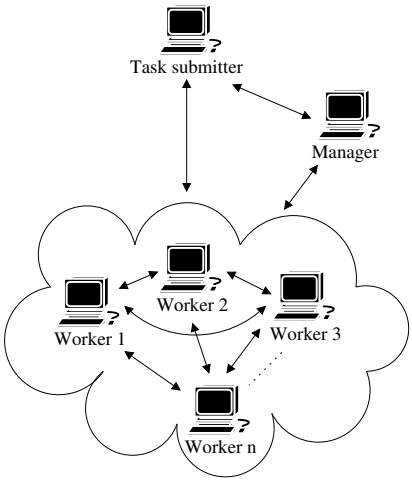

Figure 1.1: High-level view of a distributed computing system. The task submitter either connects to the workers directly or indirectly through a manager. All arrows are double-sided to signify that information may flow in both directions.

of the same architecture and run the same operating system) and can be considered trusted in that they will not try to sabotage the computations or disclose data to non authorized parties.

- Clusters consist of a large number of low-cost PCs connected by a very high-speed network. The PCs are used exclusively to provide computing power and are all of the same architecture and run the same operating system. This kind of system is a cheap alternative to more expensive supercomputers. Clusters are more suitable than NOWs for problems requiring a lot of communication.

- Grids, or grid computing systems, enable the integration of high-speed computer systems such as super-computers and clusters, networks, databases and other resources owned and managed by multiple organizations. Grids often span multiple networks and have to provide secure resource sharing across organizational boundaries. The vision is a global computing power grid where computing power is as accessible as the electricity provided by the electric power grid. In this thesis grid computing refers to systems sharing mainly raw computing power.

1.2 Goals and Limitations

The goal of this project is to develop a grid computing system based on *peer-to-peer* technology. The term peer-to-peer (or P2P) refers to networks not having a fixed number of clients and servers, but a number of peer hosts that function as both clients and servers to the other hosts on the network.

A small application run at every host of the system, the *host application*, utilizes the idle time of that host's CPU and provides the computing power. An

API provides developers with the facilities needed to develop and deploy general applications (not limited to embarrassingly parallel problems) for the system. On top of this API a small framework for embarrassingly parallel problems should be provided facilitating the development and deployment of this kind of applications.

Furthermore the system should provide:

Heterogeneity. The network will most likely consist of machines of different platforms and architectures.

Scalability. The system should be scalable from the size of a small LAN (10^1 hosts) up to the size of the Internet (10^6 hosts and more).

Fault-tolerance. Hosts are expected to be joining and leaving the system at any time thus the system has to be able to deal with host failures.

Security. Tasks should be run in a sand-box environment at the host protecting the host's private data and resources.

Anonymity. The task submitter should not need to have login access to individual hosts. However, it should be possible to require a user to authenticate to connect to the network.

Being a limited-time project some limitations had to be introduced:

- No measures will be taken to prevent sabotage or cheating.
- No sophisticated load balancing will be implemented which considers the performance of workers to schedule tasks.
- Economical aspects in assigning tasks to different workers will not be considered.
- The system will only run on Java-enabled platforms and will require Java J2SE version 1.4 or higher.
- All communication between hosts will be unencrypted.
- Compiled task code downloaded to workers will be saved on the host's file system without being encrypted or protected in any other way.

Chapter 2

Related Work

This chapter begins with a short introduction to a number of programming models for the development of parallel algorithms. The following sections describe previous and present distributed computing systems related to the work done in this thesis project.

2.1 Programming Models for Parallel Algorithms

Parallel architectures, whether they are supercomputers, computer clusters, NOWs or grids, are complicated systems raising the need for abstraction layers, either hardware based or software based, to ease the software development. Parallel multiprocessor computers usually have either a MIMD (Multiple Instruction Multiple Data) or SIMD (Single Instruction Multiple Data) hardware architecture. MIMD means that each processor can execute a separate stream of instructions on its own local data. MIMD systems have either a shared memory common to all processors or a distributed memory where the memory is distributed among the processors. Distributed memory may be private to each node or shared, also called DSM (Distributed Shared Memory). SIMD is a special case of MIMD in which all processors execute the same instruction stream on different pieces of data.

Message passing provided by MPI (Message Passing Interface) [38] and PVM (Parallel Virtual Machine) [39] is widely used in systems like clusters and NOWs. MPI and PVM provide mechanisms for executing tasks remotely and passing messages between hosts to coordinate and exchange data. There is a number of commercial and free MPI and PVM implementations for different architectures and programming languages available at present.

The BSP (Bulk Synchronous Parallel) [34] model is an abstract model of a parallel computer aiming at hiding the physical details of the underlying hardware from programs to make them more portable than programs using MPI or PVM. The BSP model consists of a number of processor-memory pairs. The pairs are connected in a network. Execution of a BSP application proceeds in phases called supersteps and all communication between processors takes place between supersteps. A superstep consists of a number of threads running in parallel on the processors. The threads only access the local memory until they reach a synchronization point, or barrier, which ends the superstep. At a bar-

rier threads must temporarily pause their execution until all other threads have reached the barrier. Before the next superstep begins all accesses to the memory of remote processors take place. BSPlib [29] is a standard C library for BSP programming available for a variety of architectures. BSPlib has also been ported to Java [27].

The tuple-space model of Linda [24] is a higher level abstraction of a distributed shared memory. The tuple-space is a distributed object repository where running processes may store and retrieve objects. In Linda, processes do not communicate directly by exchanging messages or sharing variables; instead they create new data objects (called tuples) and put them in the tuple-space where they may be accessed by other processes. Sun's JavaSpaces and IBM's TSpaces, both discussed in [18], are Java incarnations of the tuple-space model found in Linda.

Software based abstractions also include extensions which add new language elements to popular programming languages, like HPF (High Performance Fortran) and Cilk [16] (a parallel multi-threaded extension of the C language).

Sun's RMI (Remote Method Invocation) and OMG's CORBA (Common Object Request Broker Architecture) are more general technologies for distributed objects and remote method invocation.

2.2 Distributed Computing Systems in Research

The Pirhana system [25] is an example of an early distributed computing system utilizing idle cycles of workstations in a LAN. Pirhana uses the tuple-space model of Linda to achieve interprocess communication. In Pirhana pending applications are stored in a system tuple-space. Whenever a node becomes available it selects an application by random from the system tuple-space and executes it. The most notable problem with the Pirhana implementation described in [25] is that it was unable to span multiple networks incorporating heterogeneous machines.

Cilk-NOW [17] is a runtime system for Cilk applications. All Cilk runtime systems, including Cilk-NOW, use a provably efficient scheduling algorithm, called *work stealing*, which also provides fault tolerance. Whenever a worker in a system using a work stealing scheduling algorithm runs out of tasks it will, through some mechanism, steal tasks from other workers. By having faster workers stealing tasks from slower workers load balancing is achieved. Fault tolerance is a consequence of the work stealing process because a dysfunctional worker is nothing but an infinitely slow worker. Work stealing has been used extensively in other distributed computing systems and is also used in Jalapeno's framework for embarrassingly parallel type of problems. This framework will be further discussed in Chapter 3.

Being a NOW system Cilk-NOW utilized the idle time of workstations in a LAN and could not provide parallel computing on a global scale. The ATLAS system was derived from Cilk-NOW and provided global computing through a combination of Cilk and Java. Scalability was achieved by having a tree-like hierarchy of managers managing a set of workers and using hierarchical work stealing which allowed work stealing from workers in different sub-trees. The ATLAS project seems to have been discontinued.

Like ATLAS Charlotte [15] was one of the first systems to target WANs.

Based on the Java platform the system was able to meet some of the added requirements, such as heterogeneity and security, which a system targeting LANs does not have to meet. Charlotte used an eager scheduling technique very similar to the work stealing scheduling used in ATLAS but unlike ATLAS the Charlotte system used Java applets running in Java-capable web browsers to execute parallel tasks. Charlotte provided a distributed shared memory type of programming model similar to the BSP model.

SuperWeb [14], Javelin [22], Bayanihan [37] and Popcorn [21] are all other examples of distributed computing systems based on Java applets running in Java-capable web browsers. The SuperWeb and Popcorn projects focused on the economics of trading computing resources while the focus of the Javelin project was on supporting various programming models including the Linda tuple-space model and message passing. The authors of the Bayanihan system designed mechanisms for preventing sabotage in distributed computing systems [36] and showed how the BSP programming model could easily be implemented on top of a system for embarrassingly parallel type of problems.

Because of their use of Java applets these systems are essentially ubiquitous: the only requirement on the participants is a Java-capable web browser. No software installation is necessary and participating is as easy as visiting a web site. However, being applet based has its drawbacks. Applets run in a very restrictive runtime environment: they may only open outbound TCP connections to the web server hosting the applet. This limitation severely limits the scalability of the system since all communication between workers must be routed through the web server. This one server may become the single point of failure for the entire system. Applet based systems also require participants to actively start the worker applet every time they login to their workstations. In a corporate network an administrator might want to install workers on a large number of workstations and have them running even if nobody is using the workstation. This is not possible with the applet based approach.

Javelin++ [31] and Javelin 2.0 [32] extend the work originated in Javelin by moving from Java applets to Java applications. These projects examined issues related to scheduling and scalability. The system architecture of Javelin++ and Javelin 2.0 was similar to that of Javelin but included a hierarchy of managers instead of a single manager (the web server) to improve scalability. Some of the people behind the Javelin projects started the CX [23] project, which seems to be in active development. In CX the hierarchy of managers is strictly maintained in a so-called sibling-connected height-balanced fat tree which provides a very robust network topology tolerating a large number of host failures.

JNGI [41, 7] is a more recent system based on peer-to-peer technology. JNGI uses software developed by Project JXTA [9, 13] to communicate in a peer-to-peer fashion. JXTA is also used by Jalapeno and will be further described in Chapter 3. Since JNGI and Jalapeno are similar in many ways JNGI will be further discussed in Chapter 5.

Other projects such as ParaWeb [19] and Manta [40] rely either on extensions to the Java language, modifications of the Java virtual machine or native code to provide seamless migration of threads and objects to other hosts in a network. Requiring such extensions and modifications is undesirable because it may have implications on the platform independence of Java and thus the capability to cope with the heterogeneity of a global computing environment.

2.3 Distributed Computing Systems in Production Use

SETI@home [10] and distributed.net [1] are perhaps the two most famous existing projects which utilize idle Internet-connected computers to solve problems too hard for any desktop computer to solve on its own. The SETI@home project analyzes astronomic measurements collected from radio-telescopes in the search for extra terrestrial intelligent life while the distributed.net project is currently working on the RSA labs 72-bit secret-key challenge (the 56-bit and 64-bit challenges were solved in 1997 and in 2002 respectively). Both of these problems are examples of embarrassingly parallel or very coarse-grained parallel problems having a very high computation to communication ratio. E.g., a client participating in the distributed.net project would get an interval of cryptographic keys to investigate, only a few integers of data, from the server which it would then need hours or days to work on before sending a negative or positive answer back to the server. Similar projects have followed, e.g. Folding@home [4] which analyzes the 3d-structure of proteins to find the cause and cure of different diseases. All of these projects have a centralized server architecture (though not necessarily a single server), are limited to a single very coarse-grained problem and rely on the good-will of participants volunteering their computers' idle cycles.

The Grid MP Global system [5] is a system developed by United Devices [12] mainly working for non-profit on problems such as finding anti-smallpox drugs. It is similar to SETI@home and distributed.net sharing their client-server architecture and reliance on volunteers but it is not limited to a single problem. Volunteers download and install an application which in turn downloads the problem to run and the data. The application seems to be available for the MS Windows operating systems only and the problem code is probably some kind of compiled Windows binary but this has not been confirmed. Because of its client-server architecture the Grid MP Global system is still not suitable for fine-grained parallel problems. United Devices also has systems for building corporate clusters and NOWs.

Software developed by Entropia [2] powered the first phase of the fightAIDS@home [3] research project, a project similar to SETI@home. Entropia also develops software for building NOWs consisting of PCs running MS Windows.

Globus [11] and Legion [8] are research systems in production use. They are more general than most of the previously mentioned systems not only focusing on providing raw computing power. Globus provides a toolkit which is a set of services for resource allocation and process creation, authentication, monitoring and discovery, secondary storage access, etc which can be used to develop useful applications for grids. Legion is an object based virtual computer designed for millions of hosts connected by high-speed links. Users working on their home machines see the illusion of a single computer, with access to all kinds of data and physical resources. Groups of users can construct shared virtual work spaces to exchange information.

2.4 Summary of Related Work

The following list tries to summarize the most important points to be made from the related work presented above:

- The centralized client-server type of system works well for very coarse-grained problems like those solved by the SETI@home and distributed.net systems. For more fine-grained problems, however, some kind of hierarchical, decentralized system is needed to achieve scalability on a global scale. By decentralizing the system it becomes more robust.
- Java has been used extensively in research systems to provide heterogeneity but has not yet been put to the test in global projects like SETI@home or distributed.net.
- Global systems relying on volunteers have to be user friendly. Applets provide ease of installation but put to big restrictions on the applications running on the system and inhibit scalability.

Chapter 3

The Jalapeno Grid Computing System

This chapter describes the design of the Jalapeno grid computing system which is the result of this project. The chapter begins with a short introduction to Project JXTA which was used to provide Jalapeno with peer-to-peer communication abilities. The following sections describe how JXTA is used in Jalapeno and the implementation of the system.

Refer to Appendix A for a detailed user's manual describing how to run the Jalapeno host application. Application programmers will find an example of an application developed for the Jalapeno platform in Appendix B.

3.1 A Brief Introduction to JXTA

Project JXTA [9], pronounced *juxtapose* or *juxta*, defines a set of XML-based protocols allowing any network connected device, ranging from cellphones to servers, to communicate in a peer-to-peer manner. JXTA hosts create a virtual network where any host may communicate directly with any other host regardless of firewalls and NAT devices as shown in Figure 3.1.

The basic building blocks of a JXTA network are *peer*s and *peer group*s. Peers are the individual hosts comprising the network and are grouped into peer groups. Peer groups may have a membership policy disallowing peers with insufficient credentials to join the group. By default all peers automatically become members of the Net Peer Group.

Peers use *pipe*s, which may be of unicast, bi-directional or broadcast type, to communicate with each other. By having many peers in a peer group listening on the same pipe redundancy may be achieved.

The JXTA protocols define how peers discover other peers or network resources, how network resources are advertised and how messages, used by peers to communicate, are routed. When a peer joins a peer group it automatically seeks a *rendezvous peer* for that group or dynamically becomes one if none is found. A rendezvous peer is like any other peer but also forwards discovery requests to help other peers discover resources.

For a peer behind a NAT device to be reachable by peers outside the private LAN it has to register with a *relay peer*. The relay peer temporarily stores

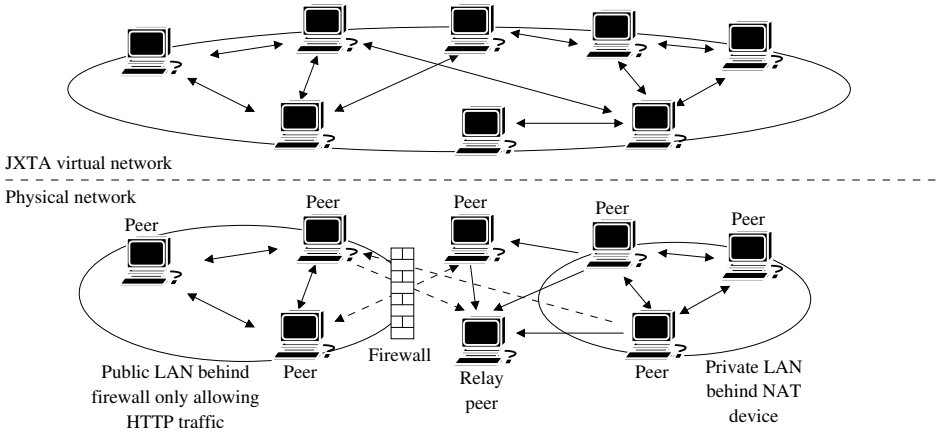

Figure 3.1: The JXTA virtual network. Double-sided arrows are used when direct connections can be initiated by both sides. One-sided arrows are used when only one side may setup a connection. Dashed lines are used when connections must use HTTP to traverse firewalls. The virtual network creates the illusion that all peers are able to communicate directly with any other peer.

messages destined for the peer which, periodically, connects to the relay peer and downloads any new messages.

Firewalls which only allow HTTP traffic may be traversed by configuring JXTA to use HTTP instead of raw TCP/IP connections.

There are currently six JXTA protocols [13]:

- The Peer Discovery Protocol is used by peers to advertise and discover resources such as peer groups, pipes, etc. Resources are described using XML-based *advertisement*s.
- The Peer Information Protocol is used by peers to retrieve status information, such as uptime, from other peers.
- The Peer Resolver Protocol enables peers to send generic queries to other peers within a peer group and receive responses.
- The Pipe Binding Protocol is used to establish pipes between one or more peers.
- The Endpoint Routing Protocol is used by peers to find routes to destination ports on other peers.
- The Rendezvous Protocol is used by peers to register with a rendezvous peer. This protocol is used by the Peer Resolver Protocol and the Pipe Binding Protocol to propagate messages throughout the network.

For more detailed information about JXTA see [13, 33, 42].

The project JXTA web site [9] provides Java (J2SE and J2ME) implementations of the protocols as well as C, Perl and PocketPC implementations. The Jalapeno system uses the Java J2SE implementation.

3.2 System Overview

The Jalapeno system design follows the general description of a distributed computing system as described in Section 1.1, comprising manager peers, worker peers and task submitter peers. Each host may have one or more of these roles. Figure 3.2 shows the structure of a Jalapeno network. To achieve high scalability the system consists of many managers, each managing a small set of workers (<100). Each manager form a peer group which its workers have to join. Within this worker peer group workers may communicate directly with other workers or the manager.

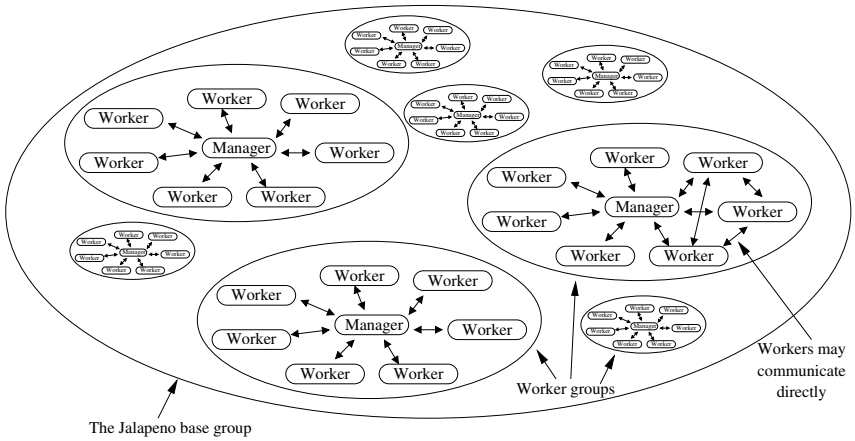

Figure 3.2: The Jalapeno grid computing system.

Initially every host starts a worker peer only. The worker starts to search for available manager peers. When a manager is found the worker tries to connect to it. Managers may only have a limited number of connected workers and will reject any new workers when this limit has been reached. Accepted workers will join the worker group created by the manager and start executing tasks.

If a worker is unable to connect to a manager within a certain time it will start a new manager peer on the local host and connect to it. This makes the system self-configuring and self-healing and provides reliability. The first host will automatically become a manager after some time and start to accept worker peers. New groups of workers will appear spontaneously as new hosts join the network. When a manager becomes unavailable its workers will either find other managers to connect to or become managers themselves and start accepting workers.

To use Jalapeno to solve a problem the task submitter submits a *task bundle*, a collection of tasks, to a randomly chosen manager as in Figure 3.3. The manager splits the received bundle into a number of smaller bundles. Managers keep a limited set of bundles from which tasks are extracted and handed to the connected workers. If the set of current bundles is not full the manager will reserve one bundle during the splitting process to be executed by its connected workers. The rest of the bundles will be forwarded to a number of other, randomly chosen, managers which repeat the process. Bundles which could not be executed nor forwarded are returned to the task submitter. All bundles sub-

mitted to the system include a unique id which the managers use to identify all previously received bundles. Already processed bundles will be rejected to prevent loops during the forwarding process.

When a worker finishes a task it will return the result to its manager which in turn will forward the result to the task submitter.

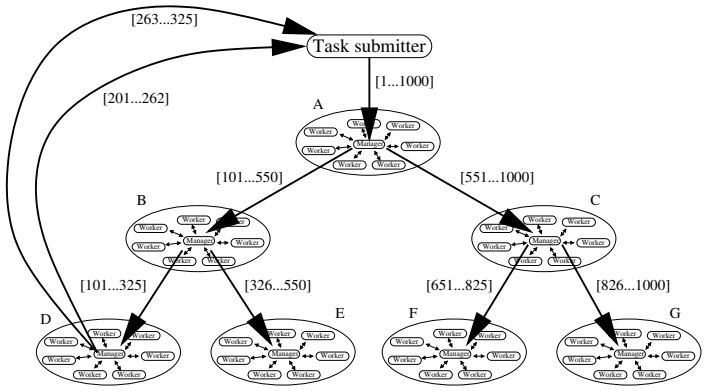

Figure 3.3: The task submission process illustrating how 1000 tasks (the bundle [1...1000]) are distributed throughout a system consisting of seven worker groups. Managers A, C and D reserve 100 tasks to be processed by their workers. Manager B is unable to reserve any tasks while manager E, F and G have enough capacity to process all tasks in the bundles sent to them. Manager D can not find any manager to forward the split off bundles to. Instead, the bundles are returned to the task submitter.

3.3 Implementation

Jalapeno is implemented using the Java programming language. By using Java many of the system requirements presented in Section 1.2 are met. The platform independence of Java makes it ideal for heterogeneous systems. Furthermore, Java has built-in support for preventing executing code to access private data and resources on the host system. The built-in support for object serialization makes it easy to transfer code and data between hosts in a network.

Java applications have a reputation of lagging in performance compared to applications written in languages such as C/C++ and Fortran and compiled for the target machine architecture. This has certainly been true in the past but recent advances in just-in-time (JIT) and adaptive compilation techniques in Java virtual machines make Java increasingly competitive [30, 26].

3.3.1 Core Classes

The implementation is centered around a number of core classes which will be described in the following subsections. The documentation of all classes and interfaces are available online at `http://jalapeno.therning.org`.

`org.therning.jalapeno.Jalapeno`

The `org.therning.jalapeno.Jalapeno` class is used to configure the system and launch worker and manager peers and start the web based user interface. User-defined configuration options are loaded from the file system. Furthermore it initializes JXTA and joins the default peer group which is the base group of worker, manager and task submitter peers.

The `Jalapeno` class is also the entry point of the Jalapeno host application. When the host application starts it will create and launch a new `org.therning.jalapeno.worker.Worker` instance. Appendix A contains a user's manual which further describes how the host application is invoked and configured.

`org.therning.jalapeno.worker.Worker`

The `org.therning.jalapeno.worker.Worker` class implements the worker functionality. When the host application is started a new `Worker` instance will be created and executed in a separate thread.

As described above the worker begins by searching for an available manager peer. When connected to a manager the worker queries the manager periodically for a new task. The tasks are Java objects implementing the `org.therning.jalapeno.task.Task` interface. Task objects are created by the manager, serialized and then transmitted as streams of bytes to the workers. The worker recreates the task object from the byte-stream and executes it in a separate thread by invoking the task's `run` method. While the task is running the main worker thread periodically sends a heartbeat message to the manager to let it know it is still alive. It also periodically checks if the result of the task being executed already has been returned to the manager by another worker. If it has the worker will try to interrupt the currently running task.

On completion the `run` method of the task returns an arbitrary object (the return-type is `java.lang.Object`, the base-class of all Java objects). The result is serialized and returned to the manager.

When there are no tasks available the worker will, every now and then, run a benchmarking task which measures the worker's current performance. The result of the benchmarking task is sent to the manager.

The task object byte-stream transmitted to workers only include the object's fields (data), not its methods (code). To be able to recreate the task objects the corresponding Java class files have to be present on the worker peer. The task submitter, as part of the submission process, uploads a set of Java archive (JAR) files to the manager, containing all necessary class files. The manager forwards the JAR files to its workers as needed.

All classes which are transferred to the worker and needed to execute a task are loaded using the `org.therning.jalapeno.JalapenoClassLoader` class loader. By using a custom class loader a sand-box environment is created preventing tasks to access the local computer's private resources. The Jalapeno class loader maintains a repository of JAR files. The repository may contain different versions of the same archive allowing different versions of a distributed application to coexist in the system. The repository also acts as a local JAR file cache. JAR files already in the worker's repository will never be transferred since they are already available to the worker.

`org.therning.jalapeno.manager.Manager`

The manager functionality is implemented by the `org.therning.jalapeno.manager.Manager` class. Whenever a worker is unable to find an available manager within a certain time it will create and execute a new `org.therning.jalapeno.manager.Manager` instance in a separate thread.

On startup the manager advertises two pipes: the first pipe is used by workers to establish a connection to the manager and the second one is used by task submitters when submitting tasks.

New workers trying to connect will only be accepted if the number of currently connected workers has not yet met the maximum number of workers the manager can handle. As described above workers periodically query the manager for tasks when idle or send a heartbeat while executing tasks. If a worker is inactive, i.e. does not send any data, for a certain amount of time the worker will be disconnected.

Task submitters submit a bundle object implementing the `org.therning.jalapeno.task.TaskBundle` interface by serializing it and transmitting it as a stream of bytes. The manager recreates the original bundle object and splits it using the bundle's `split` method. If the manager's maximum number of current bundles has not yet been reached a part of the bundle will be reserved for the manager's workers. The rest of the original bundle is split into a number of new bundles (the number is configurable by the user running the manager) and each forwarded to another, randomly chosen, manager. In the current implementation managers are chosen with equal probability regardless of how "close" (in terms of geographical location, communication bandwidth, etc) they are to the forwarding manager. The managers receiving the bundles then repeat the process. If there is no manager to forward a bundle to it will be returned to the original task submitter by connecting to the pipe indicated in the bundle object.

Idle workers will receive tasks from the manager's current bundles which may originate from a number of different applications. To be fair bundles will be chosen in a round-robin fashion when idle workers request tasks.

The `JalapenoClassLoader` class loader, used by instances of `Worker`, is once again used to load the classes needed to recreate the bundle object and provides a sand-box environment.

`org.therning.jalapeno.submitter.Submitter`

To submit tasks to the system the developer uses an instance of the task submitter class, `org.therning.jalapeno.submitter.Submitter`. Bundles which are to be submitted are put in a queue along with a set of `org.therning.jalapeno.task.JarDescriptor` instances, each describing a JAR file needed by the application. The task submitter constantly searches for manager peers and as soon as a manager has been found the first bundle will be removed from the queue and submitted to the newly discovered manager.

The developer may register listener objects with the task submitter. The listeners will be notified whenever a result is returned from a task or a bundle could not be submitted. Bundles still in the queue after a certain time will timeout and any listeners will be notified of the failure to submit the bundle.

`org.therning.jalapeno.monitor.Monitor`

The `Monitor` class, `org.therning.jalapeno.monitor.Monitor`, is used to provide network and system status information. The Jalapeno host application will automatically start a monitor instance on startup. If a manager has been started the local monitor will query the manager periodically for its current status and send that information to all other monitors in the system using a propagate pipe.

The monitor also maintains a (partial) list of all managers in the system along with the accumulated performance (the sum of all worker benchmarks) and load of each manager. Since there could potentially be a very large number of managers in the system only the most recent information is kept in the list to use less memory.

3.3.2 Framework for Embarrassingly Parallel Problems

On top of the basic task submitter classes a framework for embarrassingly parallel type of problems has been built. It features automatic task bundle splitting and task distribution among workers, resubmission of bundles which could not be handled by any manager and load balancing through work stealing scheduling.

One restriction is that the total number of entities (e.g. keys in a brute-force encryption key search problem or sub-images in the rendering of an image by a raytracer) to process can not be infinite. The automatic bundle splitting is based on this restriction. The developer describes the solution space using an `org.therning.jalapeno.ep.Space` instance, which holds the dimensionality of the solution space, each dimension's limits and the size in each dimension of a work unit given to workers. A brute-force search for the correct key to decrypt an encrypted message would have a one-dimensional space with the total range of keys as limits. A raytracing application could use a two-dimensional space with the upper left and the lower right corners as limits when rendering images. When rendering movies a third dimension, signifying the frame number, could be added. The framework keeps track of which pieces of the space have been completed.

The developer uses an `org.therning.jalapeno.ep.EPSubmitter` instance handling the submission and resubmission of bundles. The developer must provide the `EPSubmitter` with an object implementing the `org.therning.jalapeno.ep.BundleFactory` interface which handles the creation of new bundles to submit and keeps track of finished tasks and bundles.

The task bundles created by the bundle factory extend the `org.therning.jalapeno.ep.EPTaskBundle` class. The first `EPTaskBundle` being submitted encapsulates the entire solution space. `EPTaskBundle` implements automatic task splitting by invoking the solution space's `split` method which divides the space into a number of sub-spaces. The sub-spaces are used to create equally many new instances of `EPTaskBundle` which are forwarded to other managers. If a manager accepts tasks to be executed by its workers a small number of tasks, equal to the number of workers, each encapsulating a sub-space will be reserved by that manager.

Bundles failing forwarding will be returned to the `EPSubmitter` which will try to merge as many as possible of the sub-spaces of the returned bundles

and then, after some time, submit a new `EPTaskBundle` containing the merged spaces. The task submission process implemented by the framework is illustrated in Figure 3.4.

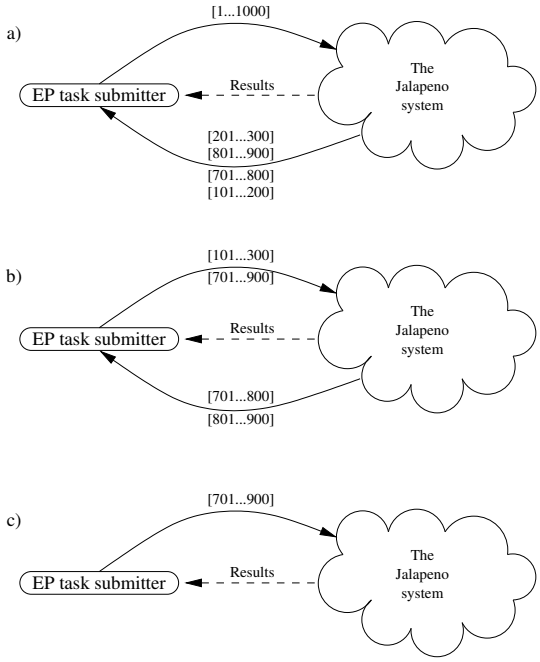

Figure 3.4: The task submission process implemented by the framework. Initially, the entire space of 1000 entities is submitted as shown in step a. The system returns four sub-spaces which could not be processed. In step b the remaining sub-spaces have been merged into two sub-spaces which are submitted. Two sub-spaces are returned by the system. Finally, in step c, the last sub-space is submitted and processed by the system. All sub-spaces have now been completed.

The `EPTaskBundle` class implements a work stealing type of scheduling algorithm to distribute tasks among a manager's workers. It maintains a list of those tasks which have been distributed to workers and are currently being worked on. When no more new tasks are available for distribution to idle workers tasks in this list will be used instead. For more information on work stealing scheduling please refer to Section 2.2.

Task results must implement the `org.therning.jalapeno.ep.EPResult` interface. The `getId` method of the result should return to the id of the `Space` which was processed to obtain the result. This id is used by the `BundleFactory` to mark the finished `Space` instances.

For a detailed description of an embarrassingly parallel application utilizing the framework please refer to Appendix B.

3.3.3 Java Web Start

The Jalapeno host application is available as a Java Web Start [6] application at `http://jalapeno.therning.org`. Java Web Start, developed by Sun, provides a platform-independent, secure, and robust deployment technology. By making the application available on a standard web server developers are able to distribute their work to end-users. Using any web browser, end-users can launch the application. Java Web Start ensures they always have the most recent version of the application.

In a sense Java Web Start is similar to the Java applet technology. The most significant difference is that Java Web Start allows the deployment of real Java applications. By utilizing the Java Web Start technology the Jalapeno system achieves the same level of availability and ease of installation as the applet based systems presented in Section 2.2 without suffering from the restrictions put on such a system.

3.3.4 The User Interface

The Jalapeno host application can be controlled using a web based user interface. At startup the host application will start a small web server on the local host (port 9090 by default). Through the user interface the user may start or stop local worker and manager peers and connect to and disconnect from the JXTA network. There is also a status page giving some statistics on the currently running local worker and manager peers as well as status information on the entire Jalapeno network. By using the configuration page the user may change the current configuration.

Figure 3.5 shows a screen shot of the main page of the user interface.

By default the user interface is only accessible by clients (web browsers) running on the same host as the Jalapeno host application.

3.3.5 Protocols

All the communication protocols used in the system are text-based. A number of more or less well known techniques for remote process communication have been implemented on top of the JXTA protocols, e.g. RMI, the standard Java Remote Method Invocation. Since many of these implementations still are beta software, possibly containing numerous bugs, it was decided that simple text-based protocols, specifically designed for Jalapeno, would be more suitable. The use of text-based protocols also facilitates debugging.

The protocols are very simple and similar to well-established protocols like HTTP and POP3. All non-binary information is encoded using the UTF-8 character set (Unicode). Binary information is sent as raw byte-streams. The sender must initiate any binary transfer by sending the length of the following byte-stream.

Commands, sent by the client, consist of a case-insensitive keyword, possibly followed by one or more arguments. All commands are terminated by the linefeed (UTF-8 code 10) character. Keywords and arguments are each separated by a single space character. There are no restrictions on the number of characters in keywords and arguments. Spaces and backspaces in arguments are escaped using the backspace character.

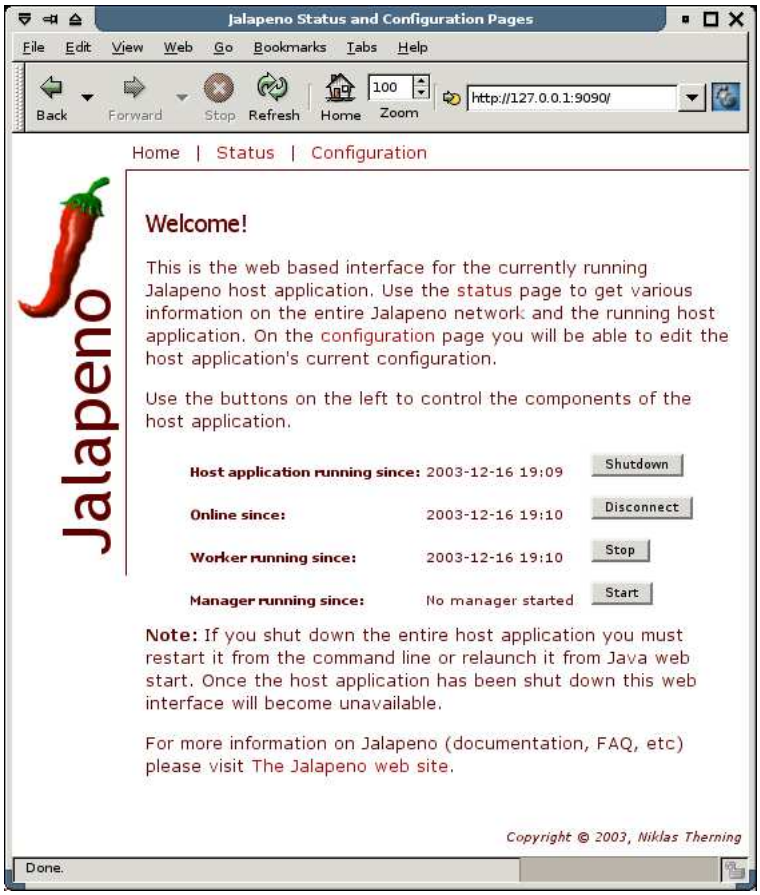

Figure 3.5: The main page of the Jalapeno host application's web interface.

The response to a command consists of a status indicator possibly followed by one or more arguments each separated by a single space character and terminated by the linefeed character. There are two status indicators: positive, `+OK`, and negative, `-ERR`.

Chapter 4

Experimental Results

To determine the performance of the Jalapeno system a number of experiments have been conducted. A homogeneous collection of 8 workstations were used to form the group of workers. The workstations were all equipped with Intel Pentium 4 1.8 GHz processors and 512 Mb of RAM running the FreeBSD 4.8 operating system and Sun's J2SE 1.4.1 version of Java. Except for the Jalapeno host application no other user applications were running on the workstations.

A dedicated manager (i.e. not running a worker) was used accepting as many as 32 workers in total. To show the heterogeneity of the Jalapeno system a Sun UltraSPARC machine running SunOS 5.8 and Sun's J2SE 1.4.2 was chosen as manager while an Intel Pentium III 700 MHz machine running Linux and Sun's J2SE 1.4.2 was chosen as task submitter.

4.1 Scalability

To determine the scalability of the system running an embarrassingly parallel problem a slightly modified version of the RC5 cracking example application presented in Appendix B was used. The tasks distributed to the workers each contained a sub-space of 54,000,000 keys from the total key space which would take 1-2 minutes for a worker to finish. The first key to start the search was chosen to make the correct key end up in the middle of the 32nd sub-space, i.e. a single worker would need to search 31.5 sub-spaces before finding the correct key.

The time T_n ($n = 2, 4, 8$) needed to find the correct key when using n workers was measured and compared with the time T_{seq} needed by the sequential version of the application. Figure 4.1 shows the measured speedup (T_n/T_{seq}) for two different configurations plotted against the ideal situation. Table 4.1 lists the measured running times.

When using the standard configuration the scalability is poor. In the standard configuration the manager accepts a maximum of 4 bundles and each of those bundles will contain as many sub-spaces of keys as there are connected workers. When a worker requests a task the bundle to deliver the task will be chosen from the list of current bundles in a round-robin fashion. This is a fair scheduling technique when multiple applications are running on the system. However, in this experiment the only running application was the RC5 applica-

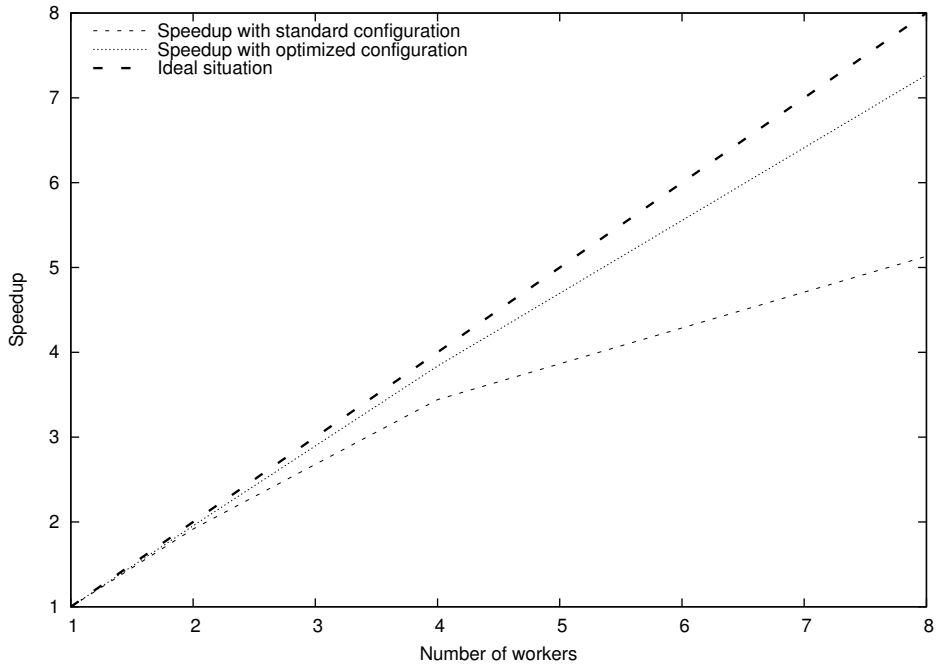

Figure 4.1: Scalability of the system when running the RC5 application using two different configurations.

tion. The bundle selection technique forced the application to search sub-spaces in a non-sequential order, possibly searching more than the first 32 sub-spaces resulting in a poor speedup. The tasks of the first submitted bundle will be processed sequentially because it will be alone in the manager's list of bundles. When the workers finish their first tasks the first bundle will be completed and up to three new bundles will be in the list. Figure 4.2 shows an example of how sub-spaces beyond the 32nd were searched before the 32nd sub-space containing the correct key during a test run using 8 workers and the standard configuration. The figure enumerates the sub-spaces in the order they were assigned to workers. Indeed, if the sub-space chosen to contain the correct key would have been different the speedup using the standard configuration could have been much better, even greater than the ideal, but that would not have been realistic. In real life the sub-space containing the correct key is of course unknown.

The scalability when using an optimized configuration, as seen in Figure 4.1, is very close to the ideal situation. In this configuration the manager could only work on a single bundle at a time. The bundle submitted by the RC5 application contained all of the 32 first tasks. Using these settings the tasks were assigned to workers in order and never would more than 32 sub-spaces have to be searched for the correct key to be found. The obtained speedup was as expected since the problem solved by the RC5 application is very coarse-grained; it is highly computationally intensive while having low bandwidth requirements.

The scalability would probably have been slightly better if the task submitter had been on the same LAN as the manager. In the used setup the task

Workers	Standard	Optimized
1	2491.8	
2	1300.5	1275.0
4	723.9	649.1
8	485.6	342.7

Table 4.1: The running times in seconds, T_{seq} (1 worker) and T_n ($n = 2, 4, 8$), needed to find the correct key using the two different configurations.

submitter needed approximately one minute to discover the manager and connect to it to submit the initial bundle. Having the two peers on the same LAN would have significantly lowered the discovery time decreasing its contribution to the measured speedup. Also, if the number of keys to search before finding the correct one would have been larger the experiments would have needed more time to finish giving more correct speedup figures. However, this was not possible because of time constraints.

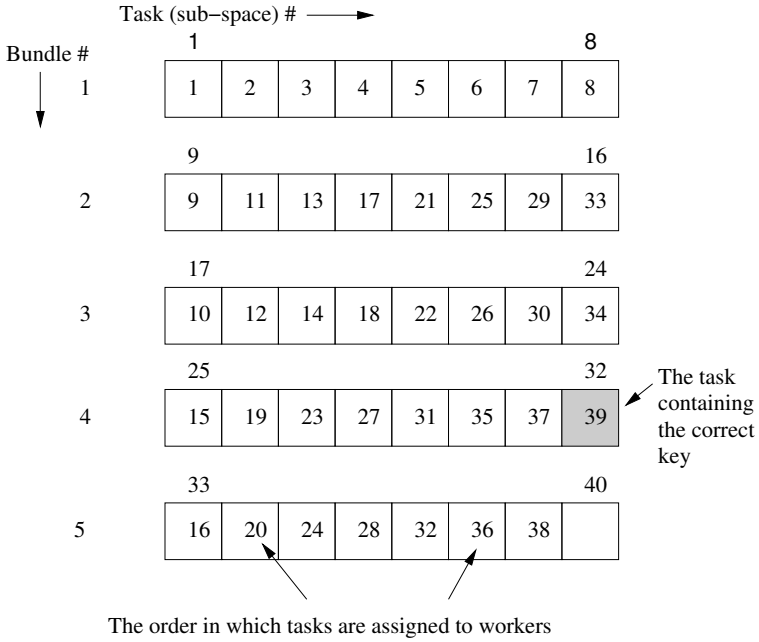

Figure 4.2: An example of how sub-spaces beyond the 32nd will be searched before the correct key can be found. The 32nd sub-space will be the 39th searched by the system.

Chapter 5

Conclusions and Discussion

The result of this thesis project is the Jalapeno grid computing system. By using peer-to-peer technology Jalapeno is completely decentralized.

The system consists of manager, worker and task submitter hosts. The task submitter submits a bundle of tasks to be processed by the system to a randomly chosen manager. The manager splits the bundle into a number of new, smaller bundles which will be forwarded to equally many, randomly chosen, other managers which repeat the process.

Each manager have a small (<100) number of connected workers. During task bundle splitting the manager will (depending on its current load) reserve a number of tasks to be processed by its workers. Workers will return the results of tasks to their managers which will forward them to the original task submitter.

The system is self configuring: hosts volunteering their computing power will at first become workers only but will eventually become managers if they can not find and connect to another manager within a certain time.

5.1 Meeting the Requirements

All the requirements presented in Section 1.2 are fulfilled by the system:

Heterogeneity. Heterogeneity is a side-effect of the choice of implementing the system using pure Java. Java is designed to be platform independent and there are Java virtual machines available for the most of today's popular computer hardware and operating systems.

Scalability. The system scales well for a small number of hosts. The scalability when using a large number of hosts on a global scale is yet to be investigated. However, such tests could prove difficult to carry out in practice due to the large number of hosts needed.

Fault-tolerance. The system provides robustness and fault-tolerance through its self-configurability. Workers leaving the system will be replaced by new workers. The framework for embarrassingly parallel problems ensures that tasks will be reassigned to other workers when a worker abruptly leaves the system without finishing its current task. Managers abruptly leaving the system will in time be replaced by new ones which will start managing workers.

Security. Security is another side-effect of the choice of implementing the system using Java. Java supports mechanisms to prevent parts of the executing code from accessing the local host's file system, network interfaces, etc.

Anonymity. JXTA provides the required anonymity. Peers may join the system regardless of their identity and may submit tasks even if they do not have user accounts on the machines running the worker peers. Jalapeno also provides the ability to create private networks only allowing access by authorized peers (this should work in theory but has not been tested due to time constraints).

As described in Section 3.3.2 a framework for embarrassingly parallel problems have been developed as required.

5.2 Major Contributions

The Jalapeno system has a number of unique features. The most important contributions are:

Automatic solution space partitioning. The automatic task bundle splitting, implemented by the framework for embarrassingly parallel problems, frees the developer from the complicated task of partitioning the solution space, distributing the sub-spaces and keeping track of finished sub-spaces.

Implicit hierarchy. To achieve higher scalability, robustness and fault-tolerance many of the other projects presented in Chapter 2 introduce some kind of hierarchy of managers. However, maintaining the hierarchy is, in most cases, complicated to implement. Jalapeno has an implicit hierarchy of managers which only exists during task submission and changes randomly over time with every task submission. Nothing is required by the system to maintain this hierarchy.

Ease of use by using Java Web Start. By utilizing the Java Web Start technology participating in the network is as easy as visiting a web site. This technology has never been used for this purpose before (at least not to the author's knowledge). The Jalapeno host application is of course available as an ordinary application as well.

5.3 Problems

The current Jalapeno implementation suffers from a number of problems:

- The current monitor implementation uses the JXTA propagate pipe to send status information to all other monitors in the network. In a large network with thousands of managers this could potentially flood the network. Some other technique to maintain status information on as many other managers as possible but still keeping the network traffic low should be used instead. One possibility would be to group managers into smaller manager groups (<100 managers) and have the monitors send status information within those groups only. Some of the peers in each such manager

group would then have to forward the group's accumulated status information to neighboring manager groups on a regular basis.

- In the current implementation results are sent by managers directly to the original task submitter. This could present a bottleneck if a lot of managers try to return results at the same time and could possibly bring down the task submitter. One possible solution would be to distribute the returning of results much in the same way as task bundles are distributed throughout the network. A submitted bundle is forwarded by zero or more managers before it ends up being handled by an available manager. Instead of having managers return results directly to the task submitter they could be returned along the same path of managers used during the submission.
- Jalapeno is self-configuring in that managers are created dynamically as needed. However, in the current implementation managers are never destroyed. This could lead to an excessive number of managers in the system compared to the number of workers. This problem could be overcome by including the number of workers connected to each manager in the status information sent by monitors. If the worker-manager ratio becomes too low the probability that a manager self-destructs will increase.
- The framework for embarrassingly parallel problems can only handle subspaces of equal size. This restriction was made to make the automatic task bundle splitting less complicated but makes it impossible to adjust the size of sub-spaces to suit the processing power or bandwidth of a particular worker.

5.4 Jalapeno vs. JNGI

5.4.1 Introduction to JNGI

JNGI [41, 7] is another distributed computing system utilizing the JXTA protocols. Peer groups are used as a fundamental building block of the system. Figure 5.1 illustrates the architecture of the JNGI network. The system consists of monitor groups, worker groups and task dispatcher groups. The monitor group handles peers joining the system and redirects them to worker groups if they are to become workers. During task submission the task submitter queries the monitor group for an available worker group which will accept its tasks. The worker group consists of workers performing the computations, while the task dispatcher group distributes individual tasks to workers.

Figure 5.1 also illustrates the hierarchy of monitor groups. If only one monitor group would have been used the communication bandwidth within that group could have become a bottleneck. A task submitter always starts the task submission process by contacting a peer in the root monitor group which decides which sub-group the request should be forwarded to. When the request has finally reached a worker group a task dispatcher in the task dispatcher group of that worker group will send a reply to the task submitter. The task submitter may then start to submit tasks to the assigned task dispatcher.

The task dispatcher group consists of a number of task dispatchers each serving a number of the worker group's workers. If a task dispatcher disappears

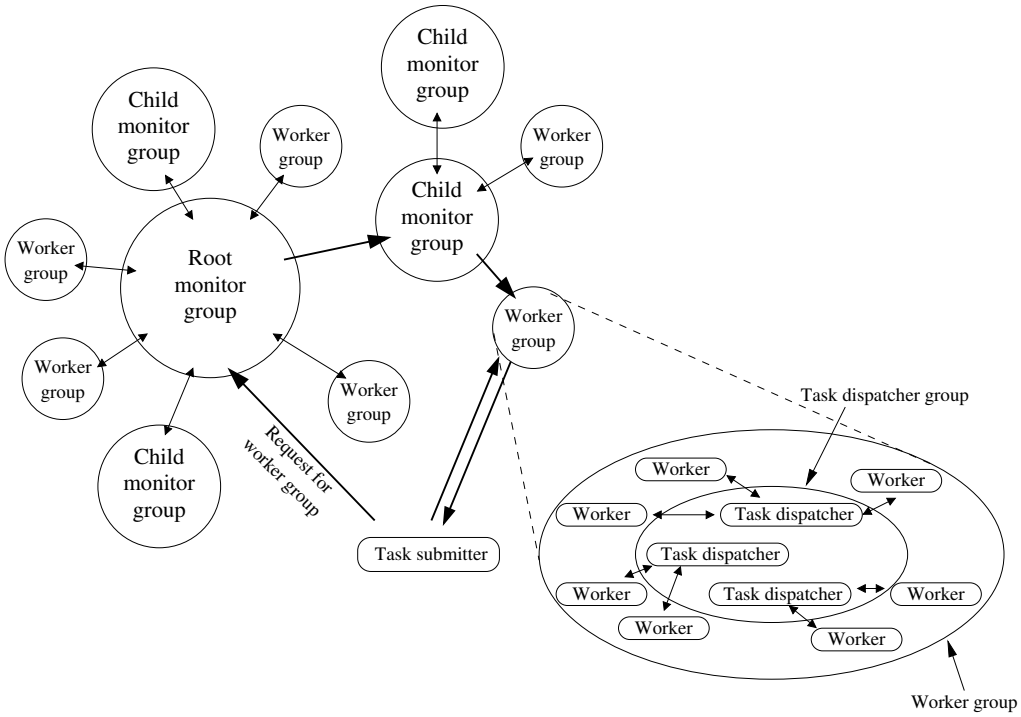

Figure 5.1: Overview of the JNGI system architecture.

the other task dispatchers will invite a worker to become task dispatcher and join the task dispatcher group. Having many task dispatchers provides redundancy. Task dispatchers within a group periodically exchange their latest results; if a task dispatcher becomes unavailable its results will not be lost. Not until all tasks of a task submission have been processed will the results be available for retrieval by the task submitter. The task submitter polls the task dispatcher inquiring whether the tasks previously submitted have been completed.

To develop applications for JNGI developers use the JNGI `RemoteThread` class in much the same way as the standard Java `Thread` class is used. The developer has to create an array of tasks to be executed. The tasks should implement the `Runnable` interface. The array of tasks is used to create the `RemoteThread` instance. By running the `start` method of the `RemoteThread` instance the tasks will be submitted to the system and executed.

JNGI is restricted to embarrassingly parallel type of problems.

5.4.2 Comparison

JNGI and Jalapeno share many similarities. Of course they are both based on the peer-to-peer technology provided by the JXTA protocols. The managers of Jalapeno are almost identical to the task dispatchers of JNGI but do not provide the redundancy provided by the JNGI task dispatcher groups. In the early designs of the Jalapeno system managers were grouped into manager groups identical to the task dispatcher groups of JNGI and exchanged results

periodically. However, this idea was abandoned at an early stage because such a mechanism would not be suitable for all types of applications. E.g. in a raytracing application the partial results exchanged between managers could be very large in size requiring a substantial amount of communication bandwidth.

The most fundamental difference between the two systems is the task submission process. In Jalapeno a collection of tasks is submitted. The tasks are then spread throughout the system without requiring any action from the task submitter. In JNGI the task submitter first has to request access to a worker group and then send the collection of tasks to a task dispatcher in the worker group. JNGI does not provide the ability to distribute a large number of tasks to many worker groups at once. Of course, both approaches have their benefits.

Jalapeno's framework for embarrassingly parallel type of problems provides automatic solution space partitioning. Developers need only to define the dimensionality of the solution space and how large sub-spaces finally processed by workers should be. JNGI is restricted to solving embarrassingly parallel type of problems but still does not provide developers with this kind of facility. However, the kind of solution space splitting implemented by Jalapeno could easily be implemented on top of the JNGI API.

Jalapeno is not restricted to embarrassingly parallel type of problems and could, for instance, easily be extended to support message passing between workers within worker groups.

JNGI offers a less complicated API than the one provided by Jalapeno through the use of the `RemoteThread` class which could make it more attractive to developers. However, Jalapeno is far less complicated to install and use in all other aspects. Jalapeno also provides the option of participating in the system and volunteering one's computer through the use of the Java Web Start technology. This possibility is not provided by JNGI.

Chapter 6

Future Work

This chapter discusses possible directions for future work to enhance the Jalapeno system. Of course, as discussed in Section 5.3 there are some, more or less, severe problems which have to be resolved. Further possible future activities will be covered below.

6.1 Possible System Enhancements

There are a number of possible enhancements that would improve the current system. They are summarized in the following sub-sections.

Load balancing

In the current system there is no way of adapting the size of tasks to suit individual workers' current performance and communication bandwidth. Indeed, the framework for embarrassingly parallel problems implements a work stealing kind of scheduler but the work units distributed to workers still are of equal sizes. The size of the work units has to be specified by the developer.

Also, during task submission, managers "closer" to the task submitter (in terms of geographical location, communication bandwidth, etc) should be preferred over managers farther away. By using JXTA managers on the same LAN as the task submitter are more likely to be used than more remote managers but this is not guaranteed.

Economical Aspects

Introducing a market-based mechanism for trading computing power would let users buy and sell computing power like any other commodity. The fact that the owner of a PC could earn money just by leaving the PC on would probably motivate even more people to join the system. Of course, care must be taken, through the use of strong encryption and other techniques, to prevent fraud in such a system. The economics would be yet another factor to consider when distributing tasks throughout the system and when assigning tasks to workers. The Compute Power Market [20] implements a market-based grid computing system.

Security

Information sent between hosts in the system might be considered sensitive and should be encrypted using standards such as SSL and TLS. This improvement is easily implemented since JXTA already supports TLS encrypted pipes.

Another aspect of security is the protection of proprietary application code. Java class files needed by an application must be downloaded to workers when running the application on the system. However, it is often extremely easy to retrieve the original source code from Java class files by using Java byte-code decompilers. Companies could be discouraged from using a Jalapeno system consisting of untrusted hosts since their private code would be accessible to unauthorized people. In [37] a number of techniques for protecting private Java classes in systems like Jalapeno are discussed.

Another aspect of security is the verification of results returned by workers. Volunteers could try to sabotage the system or cheat by sending false results before the actual results have been finished. This is known to occur in systems like SETI@home, which provides a ranking list of the volunteers contributing the most. By cheating a user could end up higher in the ranking. The most obvious solution would be to duplicate all tasks and assign the identical tasks to different workers. If the results are different one of the workers is probably cheating. Another solution would be to test workers' reliability every now and then by assigning them tasks having known results. If the result sent back by the worker is incorrect the worker could be cheating. A more sophisticated sabotage prevention technique which tries to minimize the number of extra tasks processed while still providing a high degree of sabotage-tolerance is described in [36].

Manager Redundancy

As discussed in Section 5.4.2 the kind of manager (task dispatcher) redundancy implemented by JNGI was considered in the early designs of the Jalapeno system but was later abandoned. The feature could still be added to Jalapeno but there must be a mechanism for developers to specify if a particular application would benefit from it. Managers would then be able to decide if results should be exchanged or not within the manager group. Managers should also decide not to exchange results if the amount of data to transmit is too large.

Message Passing

The Jalapeno system has been designed to allow direct message passing between workers connected to a manager but message passing has not yet been implemented in the prototype system. By adding message passing Jalapeno would be suitable for solving more, less coarse-grained, problems.

API Improvements

The current API is complicated compared to the APIs of other systems and needs to be improved. A JNGI-style `RemoteThread` class could easily be implemented to further simplify the development of Jalapeno applications.

6.2 Further Evaluation

More experiments have to be conducted to further evaluate the performance of the system. One problem with systems like Jalapeno, designed to be used on a global scale, is that it is hard or even impossible to measure their performance in a realistic environment because of the large amount of computers needed. Indeed, the performance on a global scale could be measured by simulation of the system using a carefully designed simulator. However, developing such a simulator during this project was not possible due to time constraints.

The following list presents a collection of questions further experiments would have to answer. These experiments could not be conducted during this thesis project because of time constraints.

- What impact does the worker-manager ratio have on the scalability?
- What is the impact on the performance of applications using the framework for embarrassingly parallel type of problems when the sub-space size is decreased and the computation to communication ratio becomes lower?
- What is the performance of the task submission process when using a large number of managers?

6.3 Porting to J2ME

Today all sorts of home appliances have CPUs and are connected to the Internet, e.g. gaming consoles, digital TV set-top boxes, DVD-players, etc. Many of which are idle most of the time. Porting Jalapeno to these platforms would greatly increase the number of potential volunteers. Many such appliances are able to run Java applications written for the J2ME version of Java and there is an ongoing effort to implement JXTA for J2ME enabled devices. A J2ME version of, at least, the Jalapeno worker functionality should be developed to enable compatible appliances to participate in the Jalapeno system.

References

[1] distributed.net: Project RC5. http://www.distributed.net/rc5/.

[2] Entropia^{TM}, Inc. http://www.entropia.com.

[3] fightAIDS@home. http://fightaidsathome.scripps.edu.

[4] Folding@home. http://www.stanford.edu/group/pandegroup/folding/.

[5] grid.org. http://www.grid.org.

[6] Java Web Start Technology. http://java.sun.com/products/javawebstart/.

[7] JNGI Project Home Page. http://jngi.jxta.org.

[8] Legion: A Worldwide Virtual Computer. http://legion.virginia.edu.

[9] Project JXTA. http://www.jxta.org.

[10] SETI@home. http://setiathome.ssl.berkeley.edu.

[11] The Globus Alliance. http://www.globus.org.

[12] United Devices, Inc.^{TM}. http://www.ud.com.

[13] Project JXTA v2.0: Java^{TM} Programmer's Guide, May 2003. http://www.jxta.org/docs/JxtaProgGuide_v2.pdf.

[14] Albert D. Alexandrov, Max Ibel, Klaus E. Schauser, and Chris J. Scheiman. SuperWeb: Towards a Global Web-Based Parallel Computing Infrastructure. In *11th International Parallel Processing Symposium (IPPS'97)*, pages 100–106. Geneva, Switzerland, April 1997. http://citeseer.nj.nec.com/article/alexandrov97superweb.html.

[15] Arash Baratloo, Mehmet Karaul, Zvi M. Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the Web. In *Proc. of the 9th International Conference on Parallel and Distributed Computing Systems (PDCS-96)*, pages 181–188. Dijon, France, September 1996. http://citeseer.nj.nec.com/baratloo96charlotte.html.

[16] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *The Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996. http://www.cs.utexas.edu/users/rdb/papers/JPDC96.ps.gz.

[17] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and Reliable Parallel Computing on Networks of Workstations. In *Proceedings of the USENIX 1997 Annual Technical Symposium*, pages 133–147. Anaheim, CA, USA, January 1997. `http://citeseer.nj.nec.com/blumofe97adaptive.html`.

[18] Marko Boger. *Java in Distributed Systems: Concurrency, Distribution and Persistence*. John Wiley & Sons, 2001. ISBN 0-471-49838-6.

[19] Tim Brecht, Harjinder Sandhu, Meijuan Shan, and Jimmy Talbot. ParaWeb: Towards World-Wide Supercomputing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, pages 181–188. Connemara, Ireland, September 1996. `http://citeseer.nj.nec.com/brecht96paraweb.html`.

[20] Rajkumar Buyya and Sudharshan Vazhkudai. Compute Power Market: Towards a Market-Oriented Grid. In *Proceedings of the first IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID2001*, pages 574–581, 2001. `http://citeseer.nj.nec.com/buyya01compute.html`.

[21] Noam Camiel, Shmulik London, Noam Nisan, and Ori Regev. The POPCORN Project: Distributed Computation over the Internet in Java. In *6th International World Wide Web Conference*, April 1997. `http://www.cs.huji.ac.il/~noam/popcorn.doc`.

[22] Peter Cappello, Bernd O. Christiansen, Mihai F. Ionescu, Michal O. Neary, Klaus E. Schauser, and Daniel. Wu. Javelin: Internet-Based Parallel Computing Using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, November 1997. `http://javelin.cs.ucsb.edu/docs.html`.

[23] Peter Cappello and Dimitrios Mourloukos. A Scalable, Robust Network for Parallel Computing. In *Proceedings of the ACM Java Grande/ISCOPE Conference*, pages 78–86, June 2001. `http://www.cs.ucsb.edu/projects/cx/CX_publications.html`.

[24] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[25] David Gelernter and David Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *6th ACM International Conference on Supercomputing*, pages 417–427. Washington, D.C., USA, 1992. `http://citeseer.nj.nec.com/gelernter92supercomputing.html`.

[26] Wee Jin Goh. Java vs C Benchmarks, November 2002. `http://members.lycos.co.uk/wjgoh/JavavsC.html`.

[27] Yan Gu, Bu-Sung Lee, and Wentong Cai. JBSP: A BSP Programming Library in Java. *Journal of Parallel and Distributed Computing*, 61(8): 1126–1142, 2001. `http://citeseer.nj.nec.com/499169.html`.

[28] Greg Hewgill. RC5 and Java toys. `http://www.hewgill.com/rc5/`.

[29] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998. `http://citeseer.nj.nec.com/hill98bsplib.html`.

[30] Scott R. Ladd. Market-Based Massively Parallel Internet Computing, January 2003. `http://www.coyotegulch.com/reviews/almabench.html`.

[31] Michael O. Neary, Sean P. Brydon, Paul Kmiec, Sami Rollins, and Peter Cappello. Javelin++: Scalability Issues in Global Computing. *Concurrency: Practice and Experience*, 12(8):727–753, 2000. `http://citeseer.nj.nec.com/article/neary99javelin.html`.

[32] Michael O. Neary, Alan Phipps, Steven Richman, and Peter Cappello. Javelin 2.0: Java-Based Parallel Computing on the Internet. In *Proceedings of Euro-Par 2000*, pages 1231–1238. Munich, Germany, August 2000. `http://javelin.cs.ucsb.edu/docs.html`.

[33] Scott Oaks, Bernard Traversat, and Li Gong. *JXTA in a Nutshell*. O'Reilly, 2002. ISBN 0-596-00236-X.

[34] Dick Pountain. Parallel Processing in Bulk. *Byte Magazine*, pages 71–72, November 1996. `http://www.byte.com/art/9611/sec5/art5.htm`.

[35] Ronald L. Rivest. The RC5 Encryption Algorithm. In *Proceedings of the 1994 Leuven Workshop on Fast Software Encryption*, pages 86–96, 1995. `http://theory.lcs.mit.edu/~rivest/publications.html`.

[36] Luis F. G. Sarmenta. Sabotage-Tolerance Mechanisms for Volunteer Computing Systems. In *Proceedings of the first IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 337–346, 2001. `http://bayanihancomputing.net`.

[37] Luis F. G. Sarmenta. *Volunteer Computing*. PhD thesis, MIT, March 2001. `http://bayanihancomputing.net`.

[38] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, 1996. ISBN 95-80471. `http://www.netlib.org/utk/papers/mpi-book/mpi-book.html`.

[39] Vaidy S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990. `http://citeseer.nj.nec.com/sunderam90pvm.html`.

[40] Rob van Nieuwpoort, Jason Maassen, Henri E. Bal, Thilo Kielmann, and Ronald Veldema. Wide-area parallel computing in Java. In *ACM 1999 Java Grande Conference*, pages 8–14. San Francisco, CA, USA, June 1999. `http://citeseer.nj.nec.com/article/vannieuwpoort99widearea.html`.

[41] Jerome Verbeke, Neelakanth Nadgir, Greg Ruetsch, and Ilya Sharapov. Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment. In *Proceedings of the Third International Workshop on Grid Computing (GRID 2002)*, pages 1–12. Baltimore, MD, USA, January 2002. `http://jngi.jxta.org/jngi-paper.pdf`.

[42] Brendon J. Wilson. *JXTA*. New Riders, 2002. ISBN 0-7357-1234-4. http://www.brendonwilson.com/projects/jxta/.

Glossary

API, Application Programming Interface. A set of definitions of the ways in which one piece of computer software communicates with another.

HTML, HyperText Markup Language. The standard language for creating web pages.

HTTP, HyperText Transfer Protocol. A TCP based protocol mainly used by web clients to communicate with web servers.

J2ME, Java 2 Platform, Micro Edition. A collection of Java APIs used by developers to develop applications for embedded consumer products such as PDAs, cell phones and other consumer appliances.

J2SE, Java 2 Platform, Standard Edition. A collection of Java APIs used by developers to develop applications for workstations.

Java applet, A Java applet is a small program written in the Java programming language that can be included in a web page.

LAN, Local Area Network. A computer network covering a local area, such as an office or a home.

NAT, Network Address Translation. A technique used in computer networking, which relies on rewriting IP addresses of network packets passing through a router or firewall. By using NAT a LAN connected to the Internet may use more IP addresses than it has been assigned.

P2P, Peer-to-peer. Any network that does not have fixed clients and servers, but a number of peer hosts that function as both clients and servers to the other hosts on the network.

SSL, Secure Socket Layer. A protocol designed to provide encrypted communications on the Internet.

TLS, Transport Layer Security. Another protocol designed to provide encrypted communications on the Internet. TLS is based on SSL version 3.0 and extends it with a number of new features.

WAN, Wide Area Network. A computer network covering multiple sites, often across the world.

Appendix A

User's Manual

This appendix contains a manual for users running the Jalapeno host application.

A.1 General Information

The Jalapeno host application is available as a command line application downloadable from `http://jalapeno.therning.org` but can also be launched directly from within the web browser by the use of Java Web Start. The first time the application is invoked it will create the directory `$HOME/.jalapeno` where `$HOME` is the current user's home directory (e.g. `/home/john` on a UNIX system and `C:\Documents and Settings\John` on a Microsoft Windows XP system). Jalapeno will need this directory to store configuration settings, downloaded JAR files and various other files.

Configuration settings are stored in files called `settings.xml`. Jalapeno normally reads configuration settings from `$HOME/.jalapeno/settings.xml` and `$HOME/.jalapeno/<instanceid>/settings.xml` where `instanceid` is `default` by default but may be changed if many instances are to be run on the same host. If some settings have different values in the different `settings.xml` files those in `$HOME/.jalapeno/<instanceid>/settings.xml` will take precedence. All settings have default values which will be used if no `settings.xml` files have been created.

A.2 Launching Jalapeno from the Command Line

The Jalapeno host application's installation file is available either as a gzipped archive, `jalapeno-x.y.z.tar.gz`, or a zipped archive, `jalapeno-x.y.z.zip`, from `http://jalapeno.therning.org`.

A.2.1 Installation

To install the command line application one of the two archives should be downloaded and extracted in the desired installation location (e.g. `C:\Program Files` on a Microsoft Windows system). A new directory, `jalapeno-x.y.z/`, will be created containing a number of files and sub-directories:

`apps/` contains two example applications to be run on the Jalapeno system: the RC5 application, which tries to decipher an encrypted text by doing a brute-force search for the correct key, and the raytracer application which renders raja (`http://raja.sourceforge.net`) 3D scenes. To get help on how to invoke the examples execute the scripts with the `-help` option.

`docs/` contains the system documentation generated from the source code. This is useful for developers developing new application for the Jalapeno system.

`lib/` contains the JAR files required by the application.

`jalapeno.bat` the script used to launch Jalapeno from the command line when running a Microsoft Windows operating system.

`jalapeno.sh` the script used to launch Jalapeno from the command line when running a UNIX-like operating system.

`LICENSE` the Jalapeno license agreement.

`README` a text file detailing the installation and usage of the host application.

A.2.2 Starting the Host Application

The host application is launched by running `jalapeno.bat` or `jalapeno.sh` depending on the type of operating system used. Both scripts require that the location of the `java` command is in the path searched by the system (e.g. on UNIX-like systems it should be listed in the `$PATH` variable). The scripts have the following command line options:

`-dumpconfig` Dumps all default configuration settings to the console and exits. This is useful when creating a custom `settings.xml` file.

`-help` Prints out a help message describing all command line options and exits.

`-instance <id>` Specifies the id of the Jalapeno host application instance to launch. If not set the standard id, `default`, will be used. The directory, `$HOME/.jalapeno/<instanceid>/`, will be used to store downloaded JAR files and various other files. The instance id is useful when running many instances on the same host.

`-log4j <file>` The location of a configuration file used to configure the logging facilities provided by the log4j library. If not specified a default configuration file will be used.

`-manager` Tells the host application to start a manager only (no worker will be started at all). Using this option a manager will be started immediately.

`-notheadless` This option should be set if the host application is launched in a graphical environment, e.g. when running from within Microsoft Windows or when an X display is available. If set a dialog box will

be shown at startup giving the URL to use when accessing the web based user interface.

`-system <path>` The host application normally reads configuration settings from the files `settings.xml` and `<instanceid>/settings.xml` in the directory `$HOME/.jalapeno/`. This option lets a user add a third location of a `settings.xml` file to be read on startup. The settings of the standard `settings.xml` files will take precedence over the settings in the specified file. This option could be useful for system administrators to specify site-wide settings.

When launched the application will output the URL to use when accessing the web based user interface, either by printing it on the console or by showing it in a dialog box (if `-notheadless` has been set).

A.3 Launching Jalapeno using Java Web Start

To launch Jalapeno using Java Web Start the user only has to visit the web site at `http://jalapeno.therning.org` and click the "Launch Jalapeno" link. Java Web Start will download all needed JAR files to the local computer and start the application. Normally, Java Web Start applications run in a very restrictive environment which prevents the applications from accessing any of the local computer's private resources. Since the Jalapeno host application requires access to the local file system and network interfaces the JAR files of the application have been signed. By having the developer sign the JAR files of an application using a private key the application may be granted full access to the system. The first time the signed application is launched using Java Web Start the user will have to specify (by using a dialog box like the one in Figure A.1) if it should be allowed to execute in an unrestricted environment or not.

The web based user interface will start automatically and a dialog box will appear showing the URL to use when accessing it.

Figure A.1: The dialog box which will appear when launching the host application using Java Web Start for the first time.

A.4 The User Interface

The user interface will by default be available at `http://127.0.0.1:9090`. However, the port number is dynamically configured. If port 9090 is unavailable port 9091 will be tried and so on. The URL will, as described earlier, either be printed on the console or shown in a dialog box, like the one in Figure A.2, at startup.

Figure A.2: The dialog box shown when running the host application in none headless mode or from Java Web Start.

The user interface is used to control the running peers (worker and manager peers) and retrieve status information on the running host application and on the entire Jalapeno network. It is also possible to change the current configuration settings.

The user interface should be self-explanatory and will not be described in depth here.

Appendix B

Jalapeno Example Application

This appendix gives a detailed description of the RC5 [35] brute-force key search application which has been ported to the Jalapeno system. The application tries to find the correct key given the cipher text message. RC5 is a fast block cipher using a parameterized algorithm with a variable block size, a variable key size, and a variable number of rounds. In this example a block size of 32 bits, a key size of 64 bits and 12 rounds are used.

The Java implementation of the RC5 decryption algorithm, found in the classes `RC5Algorithm` and `RC5_32_12_8`, has been obtained from the Internet (see [28]). The sources of those classes will not be listed here neither will the source of the `RC5Util` class.

B.1 RC5 – The Main Class

The `RC5` class is the entry point of the application. It extends the `org.therning.jalapeno.JalapenoApplication` class which facilitates the development of Jalapeno applications. In the `main` method the `RC5` class calls the `start` method of the `JalapenoApplication` class which initializes the Jalapeno platform and connects to the Jalapeno network. It also parses the command line arguments which must include the fully qualified path of the JAR file containing the RC5 application's classes.

Eventually the `start` method of the `JalapenoApplication` class will invoke the `start` method of the `RC5` class which creates the one dimensional space of keys to search and starts an `EPSubmitter` instance. The submitter will start submitting instances of the `RC5TaskBundle` class created by the `RC5BundleFactory`.

The `RC5` class registers as `ResultListener` with the `EPSubmitter` instance. When results arrive the `resultsReceived` method will be called which outputs some performance statistics and quits the application if the correct key has been found.

```
import java.math.BigInteger;
import net.jxta.pipe.PipeService;
import net.jxta.protocol.PipeAdvertisement;
```

```
import org.therning.jalapeno.Jalapeno;
import org.therning.jalapeno.JalapenoApplication;
import org.therning.jalapeno.JalapenoException;
import org.therning.jalapeno.ep.EPSubmitter;
import org.therning.jalapeno.ep.Space;
import org.therning.jalapeno.event.ResultEvent;
import org.therning.jalapeno.event.ResultListener;
import org.therning.jalapeno.util.PipeUtil;

/**
 * Example application to run on Jalapeno.
 * Tries to crack the RSA test pseudo-contest named
 * "RSA-32/12/8-test".
 */
public class RC5 extends JalapenoApplication
                 implements ResultListener {

    /**
     * The number of bytes in a 64-bit key.
     */
    public static final int KEY_LENGTH = 8;

    /**
     * The number of keys in a work unit.
     * 100,000,000 keys should take minutes to search.
     * A 700 MHz Intel Pentium III running Sun's JDK
     * v1.4.2 under Linux scans about 400,000 keys/s.
     */
    public static final long UNIT_SIZE = 100000000;

    /**
     * The number of milliseconds between every
     * submission by the submitter.
     */
    public static final long RESUBMIT_TIME = 20*1000;

    /**
     * The number of milliseconds after the last
     * received result before a
     * bundle should be resubmitted.
     */
    public static final long BUNDLE_TIMEOUT = 10*60*1000;

    private EPSubmitter submitter = null;
    private long startTime = 0;
    private long keyCount = 0;

    /**
     * Creates the initial space and starts the
     * submitter.
```

```
 */
public void start() throws JalapenoException {
    try {
        /*
         * Create the pipe used by managers to
         * report back results or send back
         * bundles which couldn't be forwarded.
         */
        PipeAdvertisement pipeAdv =
            PipeUtil.createPipeAdvertisment(
                Jalapeno.getInstance().getBasePeerGroup(),
                PipeService.UnicastType,
                "RC5Pipe");

        /*
         * The space is one dimensional with limits
         * [0x0000000000000000, 0xffffffffffffffff].
         */
        byte[] startKey = new byte[] {
            (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
            (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00
        };
        byte[] endKey   = new byte[] {
            (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff,
            (byte)0xff, (byte)0xff, (byte)0xff, (byte)0xff
        };

        /*
         * Create the space.
         */
        Space space = new Space(
            new BigInteger[] {new BigInteger(1, startKey)},
            new BigInteger[] {new BigInteger(1, endKey)},
            new BigInteger[] {BigInteger.valueOf(UNIT_SIZE)});

        /*
         * Create the bundle factory.
         */
        RC5BundleFactory bundleFactory =
            new RC5BundleFactory(space, BUNDLE_TIMEOUT);

        /*
         * Create the submitter.
         */
        submitter = new EPSubmitter(
            Jalapeno.getInstance().getBasePeerGroup(),
            pipeAdv, bundleFactory, "RC5",
            getJars(), RESUBMIT_TIME);
        submitter.addResultListener(this);
        submitter.start();
```

```
        startTime = System.currentTimeMillis();
    } catch (Throwable t) {
        throw new JalapenoException(t);
    }
}

/**
 * Returns the name of the application.
 *
 * @return the name.
 */
protected String getApplicationCommandName() {
    return "rc5";
}

/**
 * Called whenever results are returned back from
 * the system. Prints out some statistics.
 *
 * @param event the event containing the results.
 */
public void resultsReceived(ResultEvent event) {
    Object[] results = event.getResults();
    boolean done = false;
    BigInteger correctKey = null;
    for (int i=0; i<results.length; i++) {
        RC5Result r = (RC5Result)results[i];
        keyCount += r.getKeyCount();
        if (r.isPositive()) {
            done = true;
            correctKey = r.getCorrectKey();
        }
    }

    long   now        = System.currentTimeMillis();
    long   duration   = now-startTime;
    double secs       = (double)duration/1000.0;
    double keysPerSec = (double)keyCount/secs;

    String message =
        "Time elapsed:  "+secs+" s.\n"+
        "Keys searched: "+keyCount+".\n"+
        "Performance:   "+keysPerSec+" keys/s.\n";
    System.out.print(message);

    /*
     * Quit if finished.
     */
    if (done) {
```

```
            System.out.print("Correct key found: "+
                correctKey.toString(16));
            Jalapeno.getInstance().quit();
            System.exit(0);
        }
    }

    /**
     * Application entry point.
     *
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        new RC5().start(args);
    }
}
```

B.2 RC5BundleFactory

The `RC5BundleFactory` class creates instances of `RC5TaskBundle`. It extends the `org.therning.jalapeno.ep.AbstractBundleFactory` class which implements the `BundleFactory` interface. The `AbstractBundleFactory` class uses the original `Space` object created by the `RC5` class to keep track of finished subspaces returned by managers. It also creates new instances of `RC5TaskBundle` when needed by invoking the `getEPTaskBundle` method. The `space` parameter of the `getEPTaskBundle` method refers to a sub-space of the total key space.

```
import net.jxta.protocol.PipeAdvertisement;
import org.therning.jalapeno.ep.AbstractBundleFactory;
import org.therning.jalapeno.ep.EPTaskBundle;
import org.therning.jalapeno.ep.Space;

/**
 * Creates RC5TaskBundles to be submitted by
 * the submitter.
 */
public class RC5BundleFactory
             extends AbstractBundleFactory {
    /**
     * Creates a new RC5BundleFactory.
     *
     * @param totalSpace    the entire key space to process.
     * @param bundleTimeout the time to wait after
     *                      the last received result
     *                      from a bundle before that
     *                      bundle is resubmitted.
     */
    public RC5BundleFactory(Space totalSpace,
                            long bundleTimeout) {
```

```
        super(totalSpace, bundleTimeout);
    }

    /**
     * Returns the RC5TaskBundle for the
     * supplied space.
     *
     * @param appId     the application id.
     * @param problemId the problem id.
     * @param pipeAdv   the submitters pipe.
     * @param space     the space of keys
     *                  which the tasks created
     *                  by the bundle should
     *                  search.
     */
    protected EPTaskBundle getEPTaskBundle(
            String appId,
            String problemId,
            PipeAdvertisement pipeAdv,
            Space space) {
        return new RC5TaskBundle(
            appId, problemId, pipeAdv, space);
    }
}
```

B.3 RC5TaskBundle

`RC5TaskBundle` extends the `org.therning.jalapeno.ep.EPTaskBundle` class. Instances of the `RC5TaskBundle` class are initialized with a sub-space of the total key space. The `EPTaskBundle` class already implements the `split` method required by all task bundles and provides automatic splitting. This `split` method will divide the sub-space further and create new instanes of the `RC5TaskBundle` class to be forwarded to other managers.

```
import java.math.BigInteger;
import net.jxta.protocol.PipeAdvertisement;
import org.therning.jalapeno.ep.EPTaskBundle;
import org.therning.jalapeno.ep.Space;
import org.therning.jalapeno.task.Task;

/**
 * The task bundle for the RC5 example application.
 */
public class RC5TaskBundle extends EPTaskBundle {
    /**
     * Creates a new RC5TaskBundle.
     *
     * @param appId     the application id.
     * @param problemId the problem id.
```

```
     * @param pipeAdv   the submitters pipe.
     * @param space     the space of keys
     *                  which the tasks created
     *                  by the bundle should
     *                  search.
     */
    public RC5TaskBundle(String appId,
                         String problemId,
                         PipeAdvertisement pipeAdv,
                         Space space) {
        super(appId, problemId, pipeAdv, space);
    }

    /**
     * Creates a new RC5TaskBundle by copying the
     * fields from an other bundle.
     */
    protected RC5TaskBundle(RC5TaskBundle copyFrom,
                            Space space) {
        super(copyFrom, space);
    }

    /**
     * Gets a task which will search the keys of
     * the given space.
     *
     * @param space the space of keys to search.
     */
    protected Task getTask(Space space) {
        return new RC5Task(space);
    }

    /**
     * Gets a bundle which will create tasks searching
     * the keys of the given space.
     *
     * @param space     the space of keys
     *                  which the tasks created
     *                  by the bundle should
     *                  search.
     */
    protected EPTaskBundle getTaskBundle(Space space) {
        return new RC5TaskBundle(this, space);
    }
}
```

B.4 RC5Task

Instances of the `RC5Task` class will be transmitted to workers and executed. The `RC5Task` object is initialized with a sub-space of the total key space to search.

```
import java.math.BigInteger;
import org.therning.jalapeno.ep.EPTask;
import org.therning.jalapeno.ep.Space;

/**
 * The RC5 cracking task.
 *
 * Adopted from Greg Hewgill's code found
 * at http://www.hewgill.com/rc5/.
 *
 * The correct key is 0x82e51b9f9cc718f9.
 */
public class RC5Task extends EPTask {
    /**
     * Cyphertext from the RSA test
     * pseudo-contest "RSA-32/12/8-test".
     */
    private static final long CYPHER_TEXT = 0x496def29b74be041L;
    /**
     * iv data from the RSA test
     * pseudo-contest "RSA-32/12/8-test".
     */
    private static final long IV = 0xc41f78c1f839a5d9L;
    /**
     * This represents the plain text
     * string "The unkn".
     */
    private static final long PLAIN_TEXT = 0x6e6b6e7520656854L;

    private BigInteger startKey = null;
    private long keyCount = 0;

    /**
     * Creates a new RC5Task.
     *
     * @param space the space of keys to
     *              search.
     */
    public RC5Task(Space space) {
        super(space);
        this.startKey = space.getMin(0);
        this.keyCount = space.getMax(0).subtract(
                    space.getMin(0)).longValue();
    }
```

```
public Object run() {
    RC5Algorithm rc5 = new RC5_32_12_8();
    byte[] key = new byte[rc5.keySize()];

    /*
     * Copy the start key to key.
     */
    RC5Util.bigIntegerToByteArray(startKey, key);

    for (long i=0; i<keyCount && !isInterrupted(); i++) {
        rc5.setup(key);
        long pt = rc5.decrypt(CYPHER_TEXT)^IV;
        if (pt == PLAIN_TEXT) {
            /*
             * The key has been found.
             * Stop searching.
             */
            return new RC5Result(getSpace(),
                    RC5Util.byteArrayToBigInteger(key));
        }
        /*
         * Increment the current key.
         * The most significant byte of
         * the key is the one with index 0.
         */
        int j = key.length-1;
        while (j>=0 && ++key[j]==0) {
            j--;
        }
    }

    /*
     * No key found or interrupted.
     * Return a negative result.
     */
    return new RC5Result(getSpace());
}

/**
 * Gets a description of this RC5 task.
 *
 * @return the description.
 */
public String getDescription() {
    String first = getSpace().getMin(0).toString(16);
    String last  = getSpace().getMax(0).toString(16);
    while (first.length()<2*RC5.KEY_LENGTH) {
        first = "0"+first;
    }
    while (last.length()<2*RC5.KEY_LENGTH) {
```

```
            last = "0"+last;
        }
        return "RC5 cracking test. "+
               "This unit searches the key range "+
               "["+first+", "+last+"]";
    }
}
```

B.5 RC5Result

The results returned by `RC5Task` instances are instances of the `RC5Result` class. Results are either positive or negative. If the result is positive the correct key has been found and will be indicated in the result.

```
import java.math.BigInteger;
import org.therning.jalapeno.ep.EPResult;
import org.therning.jalapeno.ep.Space;

/**
 * The result of an RC5 task.
 */
public class RC5Result implements EPResult {
    private boolean positive = false;
    private BigInteger correctKey = null;
    private BigInteger startKey = null;
    private BigInteger endKey = null;
    private String id = "";

    /**
     * Creates a new negative RC5Result.
     *
     * @param space the space searched.
     */
    public RC5Result(Space space) {
        this.id = space.getId();
        this.startKey = space.getMin(0);
        this.endKey = space.getMax(0);
        this.positive = false;
    }

    /**
     * Creates a new positive RC5Result.
     *
     * @param space the space searched.
     * @param correctKey the correct key.
     */
    public RC5Result(Space space, BigInteger correctKey) {
        this.id = space.getId();
        this.startKey = space.getMin(0);
```

```
        this.endKey = correctKey;
        this.positive = true;
        this.correctKey = correctKey;
    }

    /**
     * Returns true if the key has been found.
     *
     * @return true or false.
     */
    public boolean isPositive() {
        return positive;
    }

    /**
     * Gets the id of the task producing
     * this result.
     *
     * @return the id.
     */
    public String getId() {
        return id;
    }

    /**
     * Gets the correct key or null if the
     * result is negative.
     *
     * @return the key.
     */
    public BigInteger getCorrectKey() {
        return correctKey;
    }

    /**
     * Gets the start key.
     *
     * @return the key.
     */
    public BigInteger getStartKey() {
        return startKey;
    }

    /**
     * Gets the end key.
     *
     * @return the key.
     */
    public BigInteger getEndKey() {
        return endKey;
```

```
    }

    /**
     * Gets the number of keys
     * between start key and end key.
     *
     * @return the key count.
     */
    public long getKeyCount() {
        return getEndKey().subtract(
                    getStartKey()).longValue();
    }
}
```

Appendix C

Used Software

The Jalapeno implementation relies on a number of third-party software packages listed below. All of them are using an open-source type of license.

Jakarta Commons CLI
`http://jakarta.apache.org/commons/cli/`
License: Apache Software License
Provides a simple and easy to use API for working with the command line arguments and options.

Jakarta Commons Logging
`http://jakarta.apache.org/commons/logging.html`
License: Apache Software License
Provides an ultra-thin bridge between different logging libraries (like log4j used by Jalapeno). Components may use the Logging API to remove compile-time and run-time dependencies on any particular logging package.

Jakarta Commons Pool
`http://jakarta.apache.org/commons/pool/`
License: Apache Software License
Provides an Object-pooling API used by Jalapeno to create pools of threads for efficient resource usage.

Jakarta Velocity
`http://jakarta.apache.org/velocity/`
License: Apache Software License
A template engine permitting anyone to use the simple yet powerful template language to reference objects defined in Java code.

Jakarta Log4j
`http://jakarta.apache.org/log4j/docs/index.html`
License: Apache Software License
Provides logging facilities. With log4j it is possible to enable logging at runtime without modifying the application binary.

JPreferences
`http://www.unidata.ucar.edu/staff/caron/prefs/index.html`

License: GNU Lesser General Public License
Provides a preferences API. Preferences are stored in XML-files.

Project JXTA

`http://www.jxta.org`
License: Sun Project JXTA Software License
Provides technology allowing any network connected device to communicate in a peer-to-peer manner.

Jetty

`http://jetty.mortbay.org/jetty/`
License: Jetty License
Jetty is a 100% Java HTTP Server used to provide the web based user interface in Jalapeno.

Raja

`http://raja.sourceforge.net/`
License: GNU General Public License
The Raja project intends to build a complete modern raytracer using the Java language. Raja is used in the distributed raytracing example application develop during this project.

RC5 Java

`http://www.hewgill.com/rc5/`
License: Unknown
Java implementation of the RC5 algorithm used by the RC5 key cracking example application.