



Fachhochschule Darmstadt
University of Applied Sciences

A System for Automatic Distributed Execution in Java using Multi-Agents - Concept Development and Implementation

by

Oliver Plohmann
oliver@plohmann.com
(Mat.-Nr. 642646)

April 2004

Referee: Prof. Dr. A. Schütte
Co-Referee: Prof. Dr. P. Wollenweber

A thesis for the submission to the
Department of Computer Science
in partial fulfillment for the degree
“Master of Science in Computer Science” at the
University of Applied Sciences
Darmstadt, Germany

Hiermit versichere ich, Oliver Plohmann, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Frankfurt am Main, 26.04.2004

(Oliver Plohmann)

Abstract

At the time of writing, agents and multi-agent systems are among the most rapidly growing areas of research and development in computer science. Agents are becoming one of the most important topics in distributed and autonomous decentralized systems. The multi-agent paradigm is widely used to provide solutions to a variety of organizational problems related to the collective achievement of one or more tasks.

The idea of this thesis work is to make use of the multi-agent paradigm for the development of an application-level automatic load balancing system. Firstly, this thesis presents concept and development of a platform for cooperative distributed agents. The system is lightweight and efficient. It is completely implemented in Java and XML. It offers an execution environment for distributed agents where agents communicate across process boundaries through the exchange of command objects. Commands are executed by interpreters. Agents decide autonomously which interpreter they choose to respond to a command. Secondly, based on this platform, a system for automatic distributed execution is developed where commands can be interrupted and migrated to other workstations in order to balance load changes.

Contents

ABSTRACT.....	I
INTRODUCTION.....	IX
1 DISTRIBUTED SYSTEMS	1
1.1 DISTRIBUTED OPERATING SYSTEMS.....	1
1.1.1 Amoeba.....	1
1.1.2 Sprite.....	2
1.1.3 MOSIX.....	3
1.2 DISTRIBUTED PROGRAMMING ENVIRONMENTS.....	4
1.2.1 Linda.....	4
1.2.2 JavaSpaces	5
1.3 GRID COMPUTING.....	5
2 LOAD BALANCING AND LOAD SHARING.....	7
2.1 ALGORITHMS FOR AUTOMATIC LOAD BALANCING	7
2.1.1 Distributed Algorithms	7
2.1.2 Centralized Algorithms.....	9
2.1.3 Choosing an Algorithm.....	10
3 AGENTS.....	11
3.1 CHARACTERISTICS OF AGENTS.....	11
3.2 CLASSIFICATION OF AGENTS	13
3.2.1 Interface Agents	13
3.2.2 Intelligent Agents	14
3.2.3 Mobile Agents	16
3.2.4 Multi-Agents	19
3.3 AGENT STANDARDS.....	22
3.3.1 Agent Communication Languages (ACLs)	22
3.3.2 Agent Platform Standards	24
3.3.3 Agent Platforms and Systems	27
4 OVERVIEW OF JANET	29
4.1 LAYERS IN JANET.....	29
4.2 SEMI-TRANSPARENT LOAD BALANCING.....	30
4.3 ARCHITECTURE	30
4.3.1 Development Platform.....	30
4.3.2 Middleware	30
4.3.3 Agent Platforms.....	31
4.3.4 Simplifications	31

4.4	DISTRIBUTED SYSTEMS AND JANET	32
4.4.1	Linda-Style Tuple Space Architectures	33
4.4.2	Distributed Operating Systems	34
4.4.3	Parallel Programming Languages	34
4.4.4	Grid Computing.....	34
5	<u>JANET.CAS</u>	<u>35</u>
5.1	CONCEPTUAL DESIGN.....	35
5.1.1	Commands and Interpreters	35
5.1.2	Applications and Capabilities	36
5.1.3	Schedulers	38
5.1.4	Event Notification	40
5.1.5	Nodes and the Central.....	40
5.2	DETAILED DESIGN AND IMPLEMENTATION.....	42
5.2.1	Schedulers	42
5.2.2	Commands and Interpreters	46
5.2.3	Sending Commands to Agents	48
5.2.4	Object Spaces	61
5.2.5	Event Registries.....	64
5.3	DRIVING APPLICATION: FIBONACCI NUMBERS	68
5.3.1	Fibonacci Numbers.....	68
5.3.2	Defining the Fibonacci Application	69
5.3.3	Node Startup.....	72
5.3.4	Implementing Commands and Interpreters.....	75
5.3.5	Sending Fibonacci Commands.....	77
5.4	DRIVING APPLICATION: REMOTE LOGGER.....	79
5.4.1	Consumer and Producer Capability.....	80
5.4.2	Starting up a Consumer Node	82
5.4.3	Starting up a Producer Node	82
5.4.4	Shutting down a Consumer Node.....	83
5.5	MEASUREMENTS.....	85
5.5.1	Overview	86
5.5.2	Measuring Total Time	86
5.5.3	Janet.CAS Measuring Interface.....	88
5.5.4	Measuring Command Travel Times	89
5.5.5	Measuring Command Execution Times	89
5.5.6	Comparing Average Values	90
6	<u>JANET.ADE.....</u>	<u>93</u>
6.1	CONCEPTUAL DESIGN.....	93
6.1.1	Load Determination.....	93
6.1.2	Distribution Algorithm	94
6.1.3	Executor-Observer-Distributor Triad	95
6.1.4	Leveling Out Workload Imbalances.....	96
6.1.5	Workload Balancing	98
6.1.6	Workload Sharing.....	102
6.2	DETAILED DESIGN AND IMPLEMENTATION.....	103

6.2.1	Executor	103
6.2.2	Observer	111
6.2.3	Distributor	114
6.3	DRIVING APPLICATION: FIBONACCI NUMBERS REVISITED	116
6.3.1	Defining Commands and Interpreters	117
6.3.2	Assembling The Application	122
<u>7</u>	<u>EXTENSIONS FOR JANET</u>	<u>125</u>
7.1	JANET.DIC	125
7.1.1	Mobile Agents vs. Mobile Applications	125
7.1.2	Security	126
7.2	JANET.VOS	126
<u>8</u>	<u>RUNNING SAMPLE APPLICATIONS</u>	<u>127</u>
8.1	OVERVIEW: STARTING UP NODES	127
8.2	SETTING UP SAMPLE APPLICATIONS	128
8.3	RUNNING THE CAS.FIBONACCI SAMPLE	129
8.4	RUNNING THE ADÉ.FIBONACCI SAMPLE	132
8.5	USING JANET WITH ECLIPSE	133
<u>9</u>	<u>SUMMARY AND CONCLUSIONS</u>	<u>139</u>
<u>10</u>	<u>REFERENCES</u>	<u>141</u>
	<u>APPENDIX A: ABBREVIATIONS</u>	<u>145</u>
	<u>APPENDIX B: LINKS</u>	<u>147</u>

List of Figures

Figure 3-1: Untell KQML performative expressed in KIF	23
Figure 3-2: Categories of performatives	24
Figure 3-3: Example of a KQML performative	24
Figure 3-4: Structure of an MASIF agent platform	25
Figure 3-5: The OMG Object Management Architecture	26
Figure 5-1: Applications and capabilities	37
Figure 5-2: Arbitrator class diagram	39
Figure 5-3: Event registries and their scope	40
Figure 5-4: Sample cluster with the central and completely connected nodes	41
Figure 5-5: General-purpose active object classes	43
Figure 5-6: Overview of the scheduler hierarchy	44
Figure 5-7: Basic arbitrator class hierarchy	45
Figure 5-8: The ICommand interface	46
Figure 5-9: The IInterpreter interface	47
Figure 5-10: Sample interpreter execute method	47
Figure 5-11: Aggregation of the CommandAccessor	48
Figure 5-12: Agent dispatcher aggregation	49
Figure 5-13: Using an agent proxy	50
Figure 5-14: Obtaining an agent proxy using an agent path	51
Figure 5-15: Sending a command to the agent itself	52
Figure 5-16: Using a MultiAgentProxy	52
Figure 5-17: Installing a callback handler	53
Figure 5-18: Handling the callback	53
Figure 5-19: CommandEnvelope basic class description	54
Figure 5-20: CommandCallbackEnvelope class hierarchy	55
Figure 5-21: Redefined execute method for class CommandResponseEnvelope	56
Figure 5-22: Invoking the callback handler	56
Figure 5-23: Waiting for an acknowledgement	57
Figure 5-24: Waiting for a reply	58
Figure 5-25: Future objects to block calling threads	61
Figure 5-26: Defining cluster object spaces in the node descriptor file	62
Figure 5-27: Simplified IAttacher interface	63
Figure 5-28: Simplified IDetacher interface	63
Figure 5-29: Using IAttacher and IDetacher objects	63
Figure 5-30: Event registry partial interface	65
Figure 5-31: Obtaining the node event registry	66
Figure 5-32: Cluster event registry with local cluster event registries	67
Figure 5-33: Defining cluster event registries in the node descriptor file	67
Figure 5-34: A node's core interpreters to install permanent node objects	70
Figure 5-35: Skeleton of a node descriptor file	70
Figure 5-36: A node's core interpreters as listed in the node descriptor	71
Figure 5-37: Definition of the CAS_FIBONACCI application	71
Figure 5-38: Registering an application programmatically	72
Figure 5-39: Applications tab of the node main view	73
Figure 5-40: Agent tab of the node main view	74
Figure 5-41: Defining an agent's initial command	74
Figure 5-42: FibonacciCommand class skeleton	75
Figure 5-43: Implementing the ICommand interface for the FibonacciCommand	76

Figure 5-44: Implementation of the FibonacciInterpreter class	77
Figure 5-45: Execute method of the StartFibonacciInterpreter	78
Figure 5-46: Results displayed in log entries tab.....	79
Figure 5-47: Definition of the remote logger consumer capability	81
Figure 5-48: Definition of the remote logger producer capability.....	81
Figure 5-49: Starting up nodes with remote logger capabilities.....	82
Figure 5-50: Extract from node descriptor file to install event handlers	83
Figure 5-51: Shutting down a consumer node.....	84
Figure 5-52: Consumer agent log view	85
Figure 5-53: Total time till received replies from all nodes.....	87
Figure 5-54: Interface to obtain statistical command information	88
Figure 5-55: Depiction of command travel times.....	89
Figure 5-56: Total execution times	90
Figure 5-57: Comparison of different curves for averages	90
Figure 6-1: Cluster Queue Size Category Table.....	97
Figure 6-2: Command distribution within capabilities	97
Figure 6-3: Leveling out commands between nodes	98
Figure 6-4: Waiting queues of different length	99
Figure 6-5: Several executing commands	100
Figure 6-6: Interface for workload-aware commands.....	103
Figure 6-7: IAgentProxy interface extensions.....	104
Figure 6-8: Workload balancing interpreter interface.....	104
Figure 6-9: ExecutorAnchor class diagram.....	106
Figure 6-10: Command arrival message sequence	107
Figure 6-11: Executing an ExecuteWorkloadInterpreter	108
Figure 6-12: System application's additional interpreters of the CORE capability.....	109
Figure 6-13: Executor's main system capability	110
Figure 6-14: Executor capability descriptor defining queue size categories.....	110
Figure 6-15: Simplified observer descriptor definition.....	112
Figure 6-16: Observer capability descriptor.....	112
Figure 6-17: Observer's main view processes tab.....	113
Figure 6-18: Observer's main view settings tab.....	114
Figure 6-19: Distributor descriptor	115
Figure 6-20: ADÉ.Fibonacci test console	117
Figure 6-21: Fibonacci workload commands class hierarchy	118
Figure 6-22: Class hierarchy of the FibonacciBalancingInterpreter.....	119
Figure 6-23: FibonacciBalancingInterpreter execute method	119
Figure 6-24: FibonacciBalancingInterpreter calculateTimes method.....	120
Figure 6-25: FibonacciBalancingInterpreter calculateFibonacci method	120
Figure 6-26: Notify the suspension handler about the suspension	121
Figure 6-27: ADÉ.Fibonacci application description	122
Figure 8-1: Sample CAS.Fibonacci setup	130
Figure 8-2: CAS.Fibonacci Demo View	131
Figure 8-3: Monitor node with one node named AARHUS-0 in the cluster.....	132
Figure 8-4: eclipse launching pads to start or debug nodes with eclipse	134
Figure 8-5: eclipse Classpath tab	135

Introduction

The idea of this thesis work is to develop a system for automatic load balancing. The concept for the system is developed using a *multi-agent paradigm*, where distributed agents cooperatively achieve a common goal. The system is entirely developed in Java. Because of the virtual machine approach of Java, the system can be used in a heterogeneous environment. The minimal system requirement is version 1.4 or later of Java (the run time environment or the JDK). The system consists of two layers. The basic layer provides the agent platform for distributed cooperative agents. This layer is application-independent and may be used as a general multi-agent platform. Another layer, based on the basic one, implements automatic distributed execution. Both layers in conjunction constitute the entire system, which is named Janet¹.

Distributed operating systems like Amoeba or Sprite offer load balancing on operating system-level. On the contrary, the system presented in this thesis is implemented on *application-level*. There are several advantages to application-level load balancing: It is possible for a Java application to make use of workload balancing even if it was not designed for workload balancing to begin with. There is also no need to migrate to a special distributed operating system for which often little software exists and that are even discontinued after they are no longer needed as a research vehicle. There is also no need to upgrade the operating system by installing expensive clustering components. Load distribution has to manage the mapping of distribution objects to distribution units [SCH95]. Distribution objects may correspond, for example, to processes, threads, or data structures whereas distribution units may correspond, for example, to computing nodes or application processes. For load balancing on application-level the mapping of distribution objects to distribution units can be adjusted to the implementation of distributed objects. For distributed operating systems this mapping has to be made automatically, which results in a loss of flexibility and additional overhead. An advantage of distributed operating systems is that they provide workload balancing in general for all applications and not only for individual ones. However, it is believed that there is a place for a system like Janet in areas where distributed operating systems or cluster systems are too effortful or expensive. In addition, Janet is not only a system for workload balancing in Java but also a general-purpose multi-agent platform.

The *first part* of this thesis consists of the chapters 1 – 3. These chapters serve as an introduction into the domain of distributed systems, load balancing, and agent technology. These domains are enabling technologies for the development of the system developed in this thesis or are related domains. The *second part* describes concept development and implementation of an agent-platform for cooperative distributed agents in chapters 4 and 5. The concept for automatic distributed execution and its implementation based on the previously developed platform is described in the *third and last part* in chapter 6.

¹ In autumn 2001 a hefty storm blew over Europe. This was the time when thinking about building a distributed system in Java began. As the name of the storm was Janet, the system to be built was given the code name Janet. Later on it was realised that Janet contained the words Java and network, which made good sense for the project, and therefore the initial code name became the final name for the system.

1 Distributed Systems

This chapter takes a general look at various distributed systems. The major areas of distributed computing covered are distributed operating systems and distributed programming. Grid computing as a new driving application for distributed computing is shortly covered as well. The survey of several interesting distributed systems is considered as an introduction into the application domain and as a basis for the analysis required for the development of Janet.

1.1 Distributed Operating Systems

Distributed operating systems are of special interest in conjunction with workload balancing. Many distributed operating systems offer support for workload balancing or were even designed with workload balancing in mind. There are several distributed operating systems that are especially interesting for this work, which are briefly described in this chapter.

1.1.1 Amoeba

Amoeba is a general-purpose distributed operating system developed by Andrew S. Tanenbaum and Gregory J. Sharp at the Vrije Universiteit Amsterdam [TAN01]. It is designed to take a collection of machines and make them act together as a single integrated system. In general, users are not aware of the number and location of processors that run their commands, nor of the number and location of the file servers that store their files. Amoeba is a distributed system, in which multiple machines can be connected together. These machines need not all be of the same kind. The machines can be spread around a building on a LAN. Amoeba is also a parallel system. This means that a single job or program can use multiple processors to gain speed. For example, a branch and bound problem such as the Traveling Salesman Problem can use tens or even hundreds of CPUs, if available, all working together to solve the problem more quickly.

Microkernel Architecture

Amoeba was designed based on a microkernel architecture. Every machine in an Amoeba system runs a small, identical piece of software called the kernel. The kernel supports the basic process, communication, and object primitives. It also handles raw device I/O and memory management. Everything else is built on top of these fundamentals, usually by user-space server processes. Thus the system is structured as a collection of independent processes. The basic function of the microkernel is to provide an environment in which clients and servers can run and communicate with one another. This modular design makes it easier to understand, maintain, and modify the system.

Threads

In Amoeba, each process has its own address space, but it may contain multiple threads. Each thread has its own program counter and its own stack, but shares code and global data with all the other threads in its process. Having multiple threads inside each process fits into the model of distributed and parallel computing very well. Despite their independent control all the threads can access a common block cache, using semaphores to provide inter-thread synchronization. This design makes programming servers and parallel applications much easier.

Memory Management

The Amoeba memory model is simple and efficient. A process consists of one or more segments mapped onto user-specified virtual addresses. There is no support for swapping, which means that Amoeba can only run programs that fit in physical memory. The advantage of this scheme is simplicity and high performance. The primary drawback is that it is not possible to run programs larger than physical memory.

Objects and Capabilities

There are two fundamental concepts in Amoeba: objects and capabilities. All services and communication are built around them. An object is conceptually an abstract data type. That is, an object is a data structure on which certain operations are defined. Amoeba primarily supports software objects, but hardware objects also exist. A server process to which remote procedure calls can be sent manages each object. Each remote procedure call specifies the object to be used, the operation to be performed, and any parameters to be passed. Capabilities are protected cryptographically to prevent tampering.

Bullet File Server

The standard Amoeba file server has been designed for high performance and is called the Bullet server. It stores files contiguously on disk, and caches whole files contiguously in core. Except for very large files, when a user program needs a file, it will request that the Bullet server send it the entire file in a single RPC. The performance is improved with a larger file cache. The maximum file size is also limited by the amount of physical memory available to the Bullet server.

1.1.2 Sprite

The Sprite Network Operating System is a research operating system developed by John Ousterhout and others at the University of California at Berkeley [OUS88]. The goal for Sprite was to create a system where a collection of networked workstations would behave like a single system with both storage and processing power shared uniformly among the workstations. Initial development on Sprite started in 1985. The final phase of the project ended in 1994. Ousterhout mentions four strong points of Sprite:

Networked File System

At the time the Sprite project was started there were no satisfying networked file systems. A file system was developed for Sprite that allows file sharing while completely hiding the network. I/O devices can be accessed uniformly across the network, and user processes can extend the file system by implementing its I/O and naming protocols using pseudo devices and pseudo file systems.

Process Migration

Sprite supports process migration, which allows processes to be moved transparently between workstations at any time. With process migration a single user can harness the power of many workstations simultaneously, achieving speedups of four or more on common system tasks such as recompilation. The migration mechanism keeps track of idle machines, uses them for

migration, and evicts migrated processes when a workstation's owner returns, so that migration doesn't impact the response time of active users.

Single System Image

Sprite provides a single system image. The file system and process migration provide the most obvious evidence of the single system image since they make storage and processing power sharable among workstations. In many other ways Sprite looks and feels just like a single system. There is one root partition, one password file, one swap area (in the networked file system), one login database, and so on. System administration is not harder with fifty machines in the network than it is with ten, and adding a new machine is no more difficult than adding a new user account. Sprite's single system image also supports different machine architectures in the same cluster.

Log-Structured File System

Another accomplishment of Sprite is its log-structured file system (LFS), which demonstrates a new approach to file system design. LFS treats the disk more like a tape, writing information sequentially in large runs that permit great efficiency. A new garbage collection mechanism was developed that continually opened up large extents of free space on the disk. The result is a system that writes small files to disk an order of magnitude faster than any other existing file system. At the same time it handles other operations, such as reads and large writes, at least as well as other systems. Log-structured file systems have other advantages as well, such as fast crash recovery, the ability to store information on disk in compressed form, and the ability to vary the block size from file to file.

1.1.3 MOSIX

MOSIX (Multicomputer OS for unIX) is a distributed operating system developed by Barak and others [BAR93] at the Hebrew University in Jerusalem. MOSIX originates from the MOS multicomputer OS. It was ported to Linux in 1995 to enhance the Linux kernel with cluster computing capabilities.

MOSIX is a management package that can make a cluster of x86 based Linux servers and workstations (nodes) run almost like a symmetric multi-processing (SMP) system. The main advantages are simplicity of use and near optimal performance. When a process is created, MOSIX will assign (and if necessary reassign) processes automatically and transparently to the best possible nodes to maximize the performance.

The core of MOSIX consists of adaptive management algorithms that monitor and automatically respond to resource requirements of all the processes vs. the available, cluster-wide resources. These *decentralized* algorithms are geared for maximal performance, overhead-free scalability and ease-of-use. The algorithms of MOSIX use preemptive process migration for:

- Parallel processing by automatic work distribution and redistribution
- Moving processes from slower to faster nodes
- Load balancing - for even work distribution
- Memory ushering (migrate processes from a node that run out of main memory) to prevent swapping or thrashing

- High I/O performance - by migrating an intensive I/O process to a file server (rather than the traditional way of bringing the data to the process)

MOSIX uses a preemptive and distributed load balancing algorithm that is essentially based on a Unix load average metric. This means that each node will try to find a less loaded node to which it would try to migrate one of its processes. When a node is low on memory the memory ushering algorithm prevails. Each MOSIX node sends at regular intervals information on its resources at a random subset of nodes. Each node at the same time keeps a small buffer with the most recently information received. The information exchanged includes the following variables: load, number of processors, speed of processor, memory used, raw memory used, total memory.

1.2 Distributed Programming Environments

Distributed systems are a popular and powerful computing paradigm. Distributed programming environments offer integrated frameworks providing the programmer with high-level constructs that significantly simplify the development of distributed applications by hiding as many details of the underlying physical distribution. This section describes some few distributed programming environments in brief that are of interest with regard to the system developed in this work.

1.2.1 Linda

Linda is a virtual shared memory system for heterogeneous networks [GE95]. In the Linda programming object space, processes interact via a tuple space. The tuple space is a common data area into which processes can read and write. An interesting feature of this tuple space is that a process will block until the process that is responsible for creating the required data does so. Thus, a process wishing to read some data will either obtain the data immediately, if it is there, or will block waiting for some other process to terminate and then furnish the tuple space with the required data.

Lazy Linda-Type Architecture

A further refinement of the structure described above allows a “lazy” process model to be defined in which all processes in the system are responsible for creating data in the tuple space [BD93, p.301]. A process will only start its execution if the data it supplies has been requested. A system is therefore initialized with an empty tuple space and a set of uninitialized processes. An external agent then attempts to read some data item. As this item is not present, it causes some process to become active and work towards producing the data. If, in order to do this, some other data is needed from the tuple space then other processes will be activated. This structure can be considered “lazy” as processes only execute when they are needed. The Linda model is *output-driven*. A request for the system to produce a particular data item will result in internal computations to take place, which may in turn dictate input actions. This is contrary to the *input-driven* data flow model where data flow is driven by input flowing into the program. Output will appear if the program’s semantics define an output action.

1.2.2 JavaSpaces

JavaSpaces is a specification by Sun Microsystems [SUN99] for a Linda-style shared memory system for which the company provides a reference implementation named *Outrigger*. A free implementation exists by *JPower* as well as commercial implementations by *GigaSpaces* (which also provides for load balancing and failover) and *Intamission*. JavaSpaces adopts many ideas behind the Linda system and adapts them to the Java object space. The use of the Java object space provides platform independence and the ability to execute objects on remote machines through the Java Remote Method Invocation (RMI).

JavaSpaces is based on Linda objects such as a tuple and a tuple space, which are named Entry and JavaSpace. A JavaSpace is a shared memory space, which can be shared between computers. Objects can be inserted and retrieved from a JavaSpace. Coordination of entries in a JavaSpace is done by JavaSpace servers. All objects in the JavaSpace have self-describing tags. When objects are inserted into the JavaSpace, a JavaSpace server matches tags with CPUs looking for work. The JavaSpace model can be considered a circulation model, or a "flow of objects". Objects circulate through the space. Other objects examine their tags, and use, or execute them as they circulate. The result is a cooperative marketplace where producer store objects in the JavaSpace and consumers look up and take objects from the JavaSpace. The developer designs the solution as a flow of objects rather than a traditional request/reply message-based scenario. An object can circulate through the JavaSpace after its computation is done. It then represents a result. That result is not simply returned to the client. Any objects in the JavaSpace interested in the result may use it. Distribution of objects is done through serialization. All objects, which can be inserted into a JavaSpace, are instances of a serializable object class called `jive.JavaSpace.Entry`. Because these objects are serializable, they can be made persistent. These objects can be transmitted across the network and they retain their behavior and state on other machines.

1.3 Grid Computing

Grid computing enables the virtualization of distributed computing and data resources such as processing, network bandwidth and storage capacity to create a single system image, granting users and applications seamless access to vast IT capabilities. Just as an Internet user views a unified instance of content via the Web, a grid user essentially sees a single, large virtual computer.

At its core, grid computing is based on an open set of standards and protocols - e.g., Open Grid Services Architecture (OGSA) - that enable communication across heterogeneous, geographically dispersed environments. With grid computing, organizations can optimize computing and data resources, pool them for large capacity workloads, share them across networks and enable collaboration.

Core Ideas

Like in the World Wide Web grid computing keeps complexity hidden: multiple users enjoy a single, unified experience. Grid Computing allows users to share files as in peer-to-peer computing. Unlike peer-to-peer computing many-to-many sharing is allowed, and not only files but other resources as well. The Web enables communication whereas Grid Computing enables full collaboration toward common business goals. Unlike clusters and distributed computing,

which need physical proximity and operating homogeneity, grids can be geographically distributed and heterogeneous.

Despite the broad attention grid computing has received, the number of applications utilizing grids are still small. Lanferman writes in [LAN02, p. 5]: “Grid computing has been a buzzword over the last years and computer science institutes have taken great effort in providing software packages to create a Grid infrastructure. The Grid “hype” has been amplified by industrial companies like IBM and SUN moving in and picking up this thread. Still the number of applications, which utilize Grids across multiple organizations remains small”.

Distributed vs. Grid Computing

There are two similar trends moving in tandem: distributed computing and grid computing. Depending on how to look at the market, the two trends either overlap, or distributed computing can be seen as a subset of grid computing. Grid computing received its name because it strives for an ideal scenario in which the CPU cycles and storage of millions of systems across a worldwide network function as a flexible, readily accessible pool that could be harnessed by anyone who needs it, similar to the way power companies and their users share the electrical grid.

Sun defines a computational grid as “a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to computational capabilities.” Grid computing can encompass desktop PCs, but more often its focus is on more powerful workstations, servers, and even mainframes and supercomputers working on problems involving huge datasets that can run for days. Grid computing leans more to dedicated systems, than systems primarily used for other tasks.

Large-scale distributed computing of the variety we are covering usually refers to a similar concept, but is more geared to pooling the resources of hundreds or thousands of networked end-user PCs, which individually are more limited in their memory and processing power, and whose primary purpose is not distributed computing, but rather serving their user. There are various levels and types of distributed computing architectures, and both grid and distributed computing don't have to be implemented on a massive scale. They can be limited to CPUs among a group of users, a department, several departments inside a corporate firewall, or a few trusted partners across the firewall.

2 Load Balancing And Load Sharing

In load balancing a process is moved from a heavily loaded node to a lightly loaded node. This provides potentially faster response time to processes that would otherwise be executed on a heavily loaded node. In order to perform load balancing, it is necessary to identify nodes to which it would be advantageous to move processes.

There are two policies describing remote process execution, *initial placement* and *process migration*. The former policy places a process at a specific site, where it will remain until execution is terminated. Under this policy, the load cannot be fully balanced, since long-lived processes are bound to the site at which they begin execution. Distributing the load using an initial placement policy is called load sharing. An example of such a system is Utopia [ZWD91]. The latter policy, process migration, allows processes to be moved to a different site during execution. In this way, processes can be moved from any site so that the load can be maintained in a balanced state. Distributing the load using process migration is called load balancing. Examples of operating systems that provide process migration include Charlotte [AF89], V [CHE88], Accent [ZAY86], Amoeba [TAN01], MOSIX [BAR93], and Sprite [OUS88].

Load sharing is simpler, but load balancing provides more benefits. In a system with a large number of long running processes, load balancing provides a finer granularity for leveling out the load across the system. Any system that implements load balancing can implement a load sharing, or initial placement, algorithm, by simply adding the constraint that processes are only migrated when they are initially invoked.

2.1 Algorithms for Automatic Load Balancing

There are two general classifications of load balancing algorithms, centralized and distributed. The former algorithm relies on a central server to determine both, when processes should be migrated to balance the load, and where to migrate the processes. A distributed algorithm lets each node make its own decision based on data that is globally available. In the following a summary of the algorithms described in [OSS92] is presented.

The node that invokes a process is called the source node. The machine that executes the process, either through initial placement or process migration, is called the target node. In most of the following algorithms, the source node must exceed a specified load level before it can migrate processes away from itself. This load level is called the threshold, and can be set either statically or dynamically, depending upon the algorithm.

2.1.1 Distributed Algorithms

When a distributed algorithm is used each node is responsible for taking actions. Each node has to decide whether to migrate processes, which ones to migrate, and which target nodes to select. For that purpose status information of all nodes has to be shared between the nodes.

Sender-Initiated

A sender-initiated algorithm is one where an overloaded source node attempts to execute a local process on a remote target node. If load information is broadcasted, the source node can choose a candidate based on this information. Otherwise, the source node must poll several sites to determine the most likely host for remote execution. The two requirements for process migration to occur in this case are:

- The source node must have exceeded the current threshold.
- A target client must be found.

If the first requirement is not met, then there is no need to balance the load any further since all nodes are below the threshold. If the second requirement is not met, then all target nodes are above the minimum threshold, and the system is too heavily loaded for any useful load balancing.

There are several different methods for choosing a site for remote execution when using a sender-initiated algorithm: *Random*, *Threshold*, *Shortest Queue*, and *Broadcast and Bidding*. In each of these methods, the load balancing mechanism is invoked whenever a process is invoked and the load at the local processor is greater than the threshold value. The load balancing mechanism attempts to find another site to execute the process. The first three methods are described by Eager et al. [ELZ86].

In *Random* a target node is chosen at random and the new process is migrated to that node. There is no state information passed between the nodes, so the source node has no knowledge concerning the load of the target node. If the newly arrived process places the target node above the load threshold, the load balancer is used again to find another new site for the process. Thus, it is possible for a process to migrate to several nodes before a suitable execution site is found.

In *Threshold* a target node is chosen at random and polled to see if it can accept an additional process without surpassing the threshold level. If not, the source node must keep the process.

In *Shortest Queue* several foreign sites are polled to determine their load. The site with the load farthest below the threshold, if there is one, is selected from this set to be the target node. This requires more information to be transferred between the nodes than either of the previous two algorithms. Simulations performed show that even a simple algorithm, such as *Random*, is a significant improvement over no load balancing [ELZ86]. *Threshold* gives even better results than *Random*. However, *Shortest Queue* provides very little improvement over *Threshold*, and costs the system more overhead time to poll potential target nodes. Thus, more complex algorithms will not provide any significant performance benefits. Simple algorithms that need a minimum amount of information passed between nodes provide the greatest speedup.

In *Broadcast and Bidding* the source node broadcasts to all nodes in the system when it has exceeded the threshold level. Target nodes that respond bid for the extra process, and the “best” bid receives the process. A simple bidding scheme is to grant the process to the first site that responds, limiting responses to those nodes with a load small enough to accept additional processes. A more complex method allows for differences in bidding weight among potential target nodes that would occur in a network with homogeneous architecture but heterogeneous performance capabilities. A greater weight can be assigned to sites that have greater processing

power, increasing the likelihood that a process will be migrated there. The disadvantage of this algorithm is that a broadcast, followed by the resulting responses, increases network traffic. To avoid network traffic, the number of sites receiving the broadcast must be limited to a small number.

Receiver-Initiated

A *receiver-initiated* algorithm [EAG86] starts at an underloaded node. When the load of a node is below the minimum threshold level, it requests processes from a heavily loaded node. The methods used for obtaining processes are similar to those used to off load processes in *sender-initiated* algorithms. However, these algorithms require process migration policies to be implemented, since it is not likely that an idle processor will locate a heavily loaded node immediately preceding process execution. There are some differences in the benefits of *sender-initiated* and *receiver-initiated* algorithms [EAG86]. The former produces better overall response times when the system load is low, and the latter when the load is high. If the system load is low, idle nodes searching for an overloaded node will consume more resources and generate more network traffic than a few overloaded nodes searching for an idle node and migrating a process. Similarly, in a heavily loaded system, several heavily loaded nodes searching for a few lightly loaded hosts will be more inefficient than a few lightly loaded nodes requesting processes from heavily loaded nodes.

Symmetric

Simulations have shown that the combination of the sender-initiated and receiver-initiated algorithms performs better than either single algorithm [MTS89]. With a small network delay, there is no significant difference between sender-initiated and symmetric at low system loads. A larger network delay resulted in no significant difference until the system load was greater than 80%. Depending on the number of clients that are polled to find a target client, network traffic may be a factor [ELZ86]. For a sender-initiated algorithm, a smaller number of queries is more beneficial in a lightly loaded system. In a heavily loaded system a greater number of queries are necessary. A symmetric algorithm must limit the number of queries made when the system load is low to be comparable to a sender-initiated algorithm. One method of combining these two algorithms is to dynamically choose either the sender-initiated or receiver-initiated algorithm based on the system load. A server detects the global load and broadcasts the algorithm to be used when there is a state change. This method suffers from periodic global polls determining the system load.

2.1.2 Centralized Algorithms

Centralized algorithms use a process scheduler that makes decisions concerning where a process should migrate. In many cases [BK90, THO87], process schedulers take all incoming processes and place them at a lightly loaded client. Workload and client characteristics can be used to determine the overall system load. These characteristics include the number of processes waiting in the CPU queue, CPU utilization, and number of jobs active at the client. These values are translated into vectors that are used to determine the most likely candidate to receive a process.

Sprite is currently using a centralized algorithm for load balancing [DO91]. A global load average daemon maintains each client's load in its own virtual memory. Local load average daemons, executing on each workstation, periodically send the local state to the global load aver-

age daemon. When a process requests migration, the global load average daemon is invoked, and a target client is selected from the list of idle clients. This method is a *sender-initiated* algorithm that relies on the central server to determine the target client. This method provides the additional benefit of monitoring how many processes each client has migrated, and supports leveling out the system wide level of migration for each client.

2.1.3 Choosing an Algorithm

[OSS92] writes on pp. 26: “Studies on Sprite and other distributed object spaces of differing sizes have shown idle hosts are plentiful, 65-85% in a network of over 60 workstation, and greater than 70% in a Michigan State University study with 17 workstations. If this is the case, then the system load is never high enough to merit a receiver-initiated or symmetric algorithm. Depending upon the system and its network, either a centralized or distributed version of the sender-initiated algorithm, using the Threshold method, will provide the best load balancing with the least amount of overhead”.

Shen et al. describe in [SYC02] their conclusions from “the design and implementation of a load balancing system in a Linux cluster that strikes a balance between acquiring enough load information and minimizing system overhead”. Their study concludes that: “1) Random polling based load balancing policies are well-suited for fine-grain network services; 2) A small poll size provides sufficient information for load balancing, while an excessively large poll size may in fact degrade the performance due to polling overhead; 3) Discarding slow-responding polls can further improve system performance”.

Schekenburger writes in [SCH97]: “There are many studies comparing sender- and receiver-initiated algorithms, but the results are contradicting. That means, the results strongly depend on structure and the dynamic behavior of the given system”.

3 Agents

A software agent is some entity that can carry out some task on its own. Once set free and left to its own devices an agent knows what to do and how to get its task accomplished. An agent in this way resembles an employee in a company that knows what he has to do and how to achieve it [BB01]. Such an employee frees his superior from having to observe him and supervise him. As a result the superior has more time to spend for his own duties and becomes more productive. Analogously, a software developer that can leave certain tasks to a set of agents is freed from worrying about many details of the system when developing it. He can think about the design of the system from a higher level of abstraction and, hence, on a lower level of complexity. Developing a system becomes easier in the same way the management of a company becomes easier when all employees know what they have to do and can make their contribution to the company independently from someone that has to motivate them and guide them. In that way, agent technology is a means for reducing the difficulties of developing complex systems. Since managing complexity is the essential problem in software development, it is not astonishing that agent technology has attracted considerable interest.

Reduction of complexity has been an important goal of object-oriented programming that started to attract interest about a decade ago and meanwhile has become a mainstream technology. The approach in object-oriented programming was to reduce complexity by providing further means to the developer to structure her software (such as inheritance) also by making use of information hiding (encapsulation, polymorphism). In agent technology, on the contrary, the emphasis is not on structure but on behavior. An agent knows how to behave to fulfill the goal it was designed for. When the structure of a system is like the skeleton of a body, the behavioral aspects of a system is like the flesh. After the skeleton of a body has been shaped it needs to be filled with the flesh. It seems that developing the skeleton is the easier part while defining the behavior – creating the flesh of the body – is the more demanding one. The developer is left more and more with less structural utilities (encapsulation, inheritance, message-passing) and feels more and more lost. It is therefore no surprise, that there is great confusion about agent technology, which focuses on behavioral aspects. There even seems to be no common consensus about what an agent is. Similar to the beginnings of object-oriented technology where most time was spent with technicalities such as object-oriented languages, object-oriented frameworks, or object-oriented middleware research and development in agent technology was much driven by the development of agent platforms, agent communication languages, or agent standards. To blow life into an agent, to make it become the independent self-controlled employee that knows how to run a business has remained an arduous task. This might be one reason to explain why agent technology so far has not made progress that rapidly as initially projected and why only few examples of well-designed agent systems can be found so far.

3.1 *Characteristics of Agents*

The advent of software agents gave rise to much discussion of just what such an agent is, and of how they differ from programs in general. Unfortunately, there is great confusion about agents: hardly anyone agrees on even the basic question of what an agent is, still less on more contentious issues. There are several reasons for this. As Hyacinth S. Nwana puts it in his paper “Software Agents: An Overview” [NWA96] agent researchers, firstly, do not own the term “agents” in the same way as fuzzy logicians/AI researchers, for example, own the term “fuzzy

logic”. Secondly, even within the software fraternity, the word “agent” is really an umbrella term for a heterogeneous body of research and development.

Stan Franklin and Art Graesser in their paper “Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents” [FG96] have collected various definitions of agents and have worked out common aspects of agents:

- "Autonomous agents are computational systems that inhabit some complex dynamic object space, sense and act autonomously in this object space, and by doing so realize a set of goals or tasks for which they are designed." [MAE95].
- "Intelligent agents are software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in doing so, employ some knowledge or representation of the user's goals or desires." [GILB95].
- "An agent is anything that can be viewed as perceiving its object space through sensors and acting upon that object space through effectors." [RN95].
- "Autonomous agents are systems capable of autonomous, purposeful action in the real world." [BRU91 & FRA95].
- “Intelligent agents continuously perform three functions: perception of dynamic conditions in the object space; action to affect conditions in the object space; and reasoning to interpret perceptions, solve problems, draw inferences, and determine actions.” [HR95].

These definitions make clear that there is no general agreement as to what constitutes an agent, or as to how agents differ from programs. The Software Agents Mailing List on the Internet provides a FAQ (frequently asked questions) that provides a list of attributes often found in agents: autonomous, goal-oriented, collaborative, flexible, self-starting, temporal continuity, character, communicative, adaptive, mobile.

The DARPA Agent Reference Architecture [THO98] provides an overview of agent characteristics that manages to summarize the most common characteristics:

- Adaptiveness: Agents dynamically adapt to and learn about their object space.
- Cooperativeness: Agents coordinate and negotiate to achieve common goals.
- Autonomy: Agents are goal directed and act on their own performing tasks on your behalf.
- Mobility: Agents move to where they are needed.
- Interoperability: Agents interoperate with humans, other, legacy systems, and information sources.

3.2 Classification of Agents

Software agents can be classified using various classification schemes. The major classification schemes that are used are via control structures, via environments (database, file system, network), via language (in which written), or via applications. A comprehensive classification via control structures has been developed by [BRU91]. In the following the major different kinds of agents are presented that have been used in existing systems to provide an overview of various kinds of agents existing in the real world.

3.2.1 Interface Agents

According to Pattie Mae [MAE93] an interface agent is "a computer program to provide assistance to a user dealing with a particular computer application". Interface agents perform different kinds of tasks as they communicate with their local user or other agents, such as, information filtering and retrieving, scheduling meetings, mail management, etc. More advanced interface agents are termed intelligent if they learn by "watching over the shoulder" of the user and detect patterns and regularities in the user's behavior. Intelligence introduces a stronger notion of agency. Intelligence gives an interface agent the ability to learn or understand from experience and make decisions in new situations without interacting with the user.

3.2.1.1 Applications

Interface agents can be used in many ways. In the following several possible applications are listed, which have already been implemented or can be seen as visions.

Assistance in Conducting Daily Tasks

Interface agents make use of methods of artificial intelligence to free the user from conducting daily tasks like reading e-mails, keeping the agenda up-to-date, and filtering of information. They learn by observing the user's activities and by drawing conclusions from them. They are able to improve their capability over time. The idea is that intelligent agents in the future can make a contribution to develop user interfaces for personal computers that make them more flexible and easier to use. It is necessary that the user can trust the interface agent since it collects confidential information when observing the user's activities. Researchers of the IBM Intelligent Agent Group express the opinion that in the near future user interfaces will not be able to compete in the market [IAG1]. Systems already exist that assist the user in filling out web-based forms. For example filling out a form when creating an account with a bank, filling out a tax declaration, or ordering items from an Internet shop.

Network Monitoring

Interface agents are able to automate repetitive tasks for network maintenance and monitoring such as filtering alarm messages and failure detection. The network administrator is freed from daily routine tasks and can concentrate on more demanding and more important tasks. Future interface agents are intended to be able to observe how the network administrator solves specific problems. From the observations made the interface agent is able to carry out the same steps in a recurring situation without assistance by the user.

Entertainment

There are several kinds of entertainment systems that could profit from the concept of interface agents. In ALIVE [MDBP96] users are able to interact with animated agents which are visualized as avatars.

Educational Systems

Educational Systems are intended to assist the student in her learning activity and to replace the teacher at least partially. Such systems are supposed to go beyond systems that can only show the user how many questions were answered correctly or incorrectly. The “digital tutor” acts, for example, as an intelligent search engine that looks up learning material to assist in the learning process.

3.2.2 Intelligent Agents

The term intelligent agent is not unambiguously defined in literature. As a result, the same terms are used in some cases in different ways. Nevertheless, there seems to be consensus that intelligent agents need to be able to act autonomously. In some papers intelligent agents are only required to expose some degree of autonomy. In the following it is assumed that intelligent agents are characterized by some additional properties:

- They have a certain **degree of intelligence** from the evaluation of predefined rules up to self-learning systems of artificial intelligence that are able to draw conclusions.
- Intelligent agents react not only in a reactive way, but in a **proactive** way. That is they are able to take the initiative rather than acting simply in response to their object space.
- They have some **social competence** so that they are able to communicate with the user or other systems to fulfill their duties.
- Even more advanced agents are able to **cooperate** with other agents to get tasks accomplished that go beyond their abilities.
- They might even be able to **move** freely in networks to make use of remote resources or to meet other agents to cooperate with.

Russel & Norvig [RN95] have categorized intelligent agents into several groups starting from very simple table driven agents and ending with advanced BDI agents.

3.2.2.1 Table-Driven Agents

Table-driven agents implement a simple kind of intelligence using lookup tables that contain predefined rules for every kind of event that may happen. Table-driven agents represent the simplest kind of intelligent agents. They can be used in environments where they only have to process a small number of stimuli. Usable results can not be obtained in less simple environments. Russel & Norvig [RN95] mention the example of a table-driven agent playing chess that has no other ability than playing chess. The table for the chess agent would have to contain 35^{100} entries. Creating this table would take a considerable amount of time. In addition, the agent does not have some autonomy, as its behavior is entirely defined from the outside. The

agent is not able to adapt to a changing environment. Even if the agent were given some ability to learn to reach some degree of autonomy, it would take impractical long time till it has built up the values for the new entries in the table

3.2.2.2 Simple Reflex-Agents

As the example of the chess-playing table-driven agent has shown, the option of constructing an explicit lookup table is out of the question. Simple reflex agents summarize portions of the table by noting certain commonly occurring input/output actions. Humans also have created such input/output connections. Russel & Norvig [RN95] mention the example of a braking car: if the car in front brakes, and its brake lights come on, then the driver initiates braking. Such an association is called condition-action-rule. It defines the action in response to an external stimulus:

if car-in-front-brakes **then** initiate-braking

3.2.2.3 Agents that keep track of the world

The simple reflex agent described before can only work when the appropriate decision can be made based on the current perception. If a driver looks into the mirror periodically, he can notice whether the car behind him is about to take over. If the distances to the following car continue to become smaller, the agent has some indication that the following car is going to take over. This information is important to decide whether the current moment is a good moment to change the lane. For an agent to be able to perceive differences in its environment it needs to have an internal state in order to compare perceptions made at different moments in time. The agent needs to collect information about its environment. By comparing information collected at different times it is able to perceive what is not visible (that the following car is about to take over).

3.2.2.4 Goal-Oriented Agents

Knowing about the current state of the environment is not necessarily enough to decide what to do. For example, if a taxi reaches a road junction, it has to decide whether to turn left or right. To be able to take a decision the taxi needs to have a goal. To evaluate several options an agent can compare the results of possible actions. An agent playing chess can assess the quality of a move by taking various aspects into account when evaluating it (position of pieces, worth of the material).

To be able to evaluate the consequences of a series of moves an agent has to search and plan. Searching and planning are the respective sub-disciplines of artificial intelligence that investigate how to find and assess such chains of activities. With this kind of decision-making the agent must take into account what will be in future. It must be able to assess what effects its actions are going to have. This is an important difference to the decision-making of the simple reflex agents. In case it starts to rain the goal-oriented agent is able to update its knowledge about how effectively its brakes will operate. This new knowledge will have an effect on future decisions when driving in rainy weather. On the contrary, for the simple reflex agent many if-then-else rules need to be extended.

3.2.2.5 Utility-Oriented Agents

In case that the agent can make several choices that all lead to the goal the goal-oriented agent has no way to evaluate which alternative is optimal. A goal-oriented agent driving a taxi only knows that there are several ways to reach the destination. To make the optimal decision an agent has to be able to compare the utility of different options.

3.2.2.6 BDI-Agents

BDI agents follow the belief-desire-intention paradigm. This paradigm has been used in various academic and industrial applications, mostly for the implementation of simulators. A BDI agent makes certain assumptions (beliefs) about its environment. It has several desires it strives to fulfill. The desires force it to develop intentions. BDI agents are different from pure deductive systems and other traditional models of artificial intelligence because of the concept of intentionality. A BDI agent develops a plan to pursue its intentions.

3.2.3 Mobile Agents

Mobile agents are programs that can move around in heterogeneous networks performing tasks on behalf of a user. As other type of agents, mobile agents dispose of autonomy. An agent decides on its own based on local conditions whether and where it wants to migrate. When migrating the dynamic process state of the agent is captured and attached together with some context information to a message that is sent to the destination. After arrival at the destination, the process state is recovered and the agent continues execution seamlessly at the location where it was interrupted. In order to perform its task the agent interacts with the respective local environment that is hosting it (the agent platform). The network is only being made use of when migration to another machine is required. The ability to migrate to another machine makes mobile agents interesting, because it enables them to be used in networks with low bandwidth, long latency, and fault-prone communication channels (typically, the agent platform requires comparatively more resources).

3.2.3.1 Infrastructure for Mobile Agents

Interpreted languages are an obvious choice for mobile agents where agents on the agent platform are executed with the use of an interpreter or a virtual machine. The use of an interpreted language makes it possible to overcome platform dependencies in heterogeneous networks. However, agent systems and frameworks exist that, for example, were developed using C++. In these cases an additional effort has to be made to overcome platform dependencies. The implementation of security mechanisms has to be implemented additionally. Java has been accepted more and more as the development platform for mobile agents. Because of the use of a virtual machine platform independence is already achieved. Furthermore, the Java architecture already provides several security mechanisms, APIs, and libraries for the installation of additional security mechanisms.

3.2.3.2 Security Concerns

Security problems are the most serious problem area of mobile agents. In general the threats to mobile agent systems are classified into threats imposed by *malicious agents* and threats imposed by *malicious hosts*. As Mattern states in [MAT] spying out or adulterating data or pro-

gram context of agents during migration can be prevented with the use of classic cryptographic methods such as encryption, signatures, and checksums. With the growth of the Internet the required infrastructure consisting of certification authorities to verify the authenticity of users and certificates is already in the state of construction.

Threats for the Host

For the host danger arises from the agent it executes. For all possibilities of mobile agents to be exploited the agent should be granted access to all resources available to the host. For this reason, the host has to be able to distinguish a benevolent agent from a malicious one. The developer of a malicious agent need not necessarily have malicious intentions. An agent can become malicious due to an unintended programming error.

Threats for the Agent

The agent has to protect itself against a malicious host, which is generally seen to be the more challenging problem. Busch et al. [BRM98] have identified three main security issues that arise from malicious agents:

1. Integrity protection of the agent - in particular the agent's mutable part. Agents might be abused as innocent carriers of illegal or offensive materials. Hosts may also try to delete, replace, or invalidate commitments to the agent, such as terms negotiated in electronic commerce applications. Furthermore, hosts might remove or manipulate offers of competitors already visited by the agent.
2. Maintaining the secrecy of the agent's computations and data, which is a fundamental requirement for fair negotiations as well as for computations on confidential information such as the agent's secret keys.
3. Protecting the integrity of the agent's control flow, which is a precondition for any agent to trust its own decisions. Otherwise, malicious hosts might make agents believe that an offer is acceptable when it is actually not.

In their article Busch et al. [BRM98] mention several references to latest research work on solving these security issues.

The scenario of an agent traveling to several suppliers to find the one with the cheapest offer is legendary. An advanced security problem in this context is the scenario where two or more suppliers form a conspiracy or when agents are being cloned in malicious intention (clones cannot be distinguished from the original). In case any measures against this kind of attacks exist they would require the agent to retreat temporarily into a secure location or to communicate with other authorities (user, bank, etc.). However, doing so would contradict the idea of mobile agents. It would probably be easier for the user to ask every supplier individually for an offer.

3.2.3.3 Applications for Mobile Agents

World Wide Web

The possibilities to use mobile agents for the World Wide Web are plentiful. A typical example is the one of a mobile agent traveling through the Internet and analyzing web pages on behalf of a search engine. After arrival at the origin the mobile agent passes on all indexing information it has collected which is stored in an index database.

Commercial Applications

Mobile agents can be used to travel through the Internet and ask several suppliers what products they offer at what price. With the number of market participants increasing rapidly in the Internet it becomes easily possible to lose oversight. Mobile agents could be used to search for the best offer or for the consumer that pays the highest price. However, there is no other application of mobile agents where security concerns are that serious as with commercial applications (see section 3.2.3.2).

Telecommunication

The autonomy of mobile agents can be used in telecommunication for so-called smart networks or smart messages [MAG96]. In a smart network mobile agents remain stationary. Because of their intelligence they can pursue predefined tasks. These agents can be used with synchronous communication (telephony) or asynchronous communication (mail-oriented systems). In case of smart messages, mobile agents are being used to transport data through the network or it travels from caller to callee and reserves the required resources for the communication.

Network Management

For the support of network management agents are supplied with intelligence and autonomy and assist the network manager by freeing him from monotonous and repetitive tasks. Typical uses of mobile agents in network management are the collection of status information, the modification or configuration of network components.

Mobile Computing

Laptops or similar portable devices mostly dispose of a relatively low processing power and network connection with low capacity. Users only need to create a connection with the network temporarily from different locations. Mobile agents are very well suited for such a setting. During a short connection with the network a mobile agent is sent to the host that carries out some task at the host site. After having completed its task it returns to a central location in the network from where it can return to the user's mobile device. The mobile agent returns to the mobile device when the user logs in next time. The user benefits from the larger processing power of the host system and only needs to transfer the data to her portable device, which the mobile agent has collected.

3.2.4 Multi-Agents

The concept of multi-agent systems (MAS) is defined by two major disciplines: distributed artificial intelligence and artificial life. Ferber writes in [FER01, p. 44]: “The goal of distributed artificial intelligence is to create organizations, which consist of symbol processing systems and are able to solve problems by logical deduction. Artificial life tries to understand and model systems that are artificially equipped with life”.

A multi-agent system is a more or less loosely coupled network of software agents that can solve problems cooperatively, which a single agent could not do, for example, because of missing information. The difficulty lies in breaking down a problem into subproblems and re-assembling solutions to subproblems to a solution of the entire problem. Cooperation between agents has to be designed in such a way that fewest possible interactions are made while working together with highest possible efficiency [UNIBITF94].

3.2.4.1 Coordination in Multi-Agent Systems

Coordination of agents in multi-agent systems is decisive in order to reach a common goal or avoid conflicts. There are two different kinds of cooperation: cooperation and competition. In a cooperative system, agents work together to achieve a common goal whereas in a competitive system, agents pursue different goals with goals that are partially contradicting. For several agents working cooperatively together the question is important how the common goal can be achieved best. This means that individual agents need to carry out different actions to reach the common goal most efficiently. Plans have to be developed. The process of developing plans between agents can be accomplished by all agents acting together or by an agent that creates a plan for all other agents. In case of contradicting goals an important part of coordination is the negotiation between agents and the search for compromises.

3.2.4.2 Applications of Multi-Agent Systems

There are numerous applications for multi-agent systems. In the following four main categories of MAS are observed: problem solving, collective robot systems, multi-agent simulation, and construction of hypothetical worlds.

Problem Solving

Solving problem concerns all situations in which software agents are used that are of use to humans. This category can be distinguished from collective robot systems, because they are pure software agents, which means that they have no physical structure. There are three kinds of problem solving: distributed problem solving, solving distributed problems, and solving problems through coordination. For the first kind, all the knowledge to solve a problem is distributed over all agents. Each agent has only partial abilities to solve the problem. There is cooperation between specialists. For the second kind, the problem itself is distributed. Agents have similar abilities as agents of the first kind. For the third kind, interaction processes are used to solve classic problems.

Distributed Problem Solving

All applications that use the principle of distributed problem solving assume that complex problems can be solved with the use of many specialists. These specialists have specialized complementing expertise. As soon as the knowledge of one agent is exhausted several specialists have to work together to solve the given problem. An example is the construction of a racing car. Different kind of expertise is required to construct it. There are experts for the development of engines, aerodynamics, chassis, tires, etc. Not until all specialists work together and contribute their expertise is it possible to build a racing car that is competitive.

Another example from industrial process control systems is the so-called “Flavors-Paint-Shop-System” to paint trucks [FER01, p. 54]. After leaving the assembly line, the trucks are painted in colors according to customer preference. There are fewer painting stations than colors. For this reason it might become necessary to exchange the paint at a specific painting station, which takes a lot of time and consumes a lot of paint. At the beginning, the system used a classic program to optimize paint consumption. This program was expensive and hard to change. Detailed planning was required that left little room for handling machine failures. In a second attempt, the system was replaced by a multi-agent system in which an agent represented each painting station. When a painting station becomes available it takes a truck out of the waiting queue. The waiting queue contains permanently about 100 trucks. The selection of the truck is defined by the following three rules:

1. Take the first truck out of the queue that has the paint currently in use.
2. In case no truck has to be painted with this paint, take the truck with the highest priority and change to the required paint.
3. In case no truck has highest priority, take the next truck from the queue and change to the required paint.

Even though this system is very simple, the number of paint changes could be reduced by a half compared to the centralized system. More than a million dollar could be saved in a year. In addition, the system is more fault tolerant: the system has no more problems with the breakdown of individual painting stations. This example shows with which simple rules the use of a multi-agent system can be advantageous compared to a centralized system.

Solving Distributed Problems

A distributed problem is being solved in case not only the assignment of tasks is distributed but also the problem area itself. Most applications concerned are analysis, recognition, and error diagnostics – in particular in the area of physically distributed systems. For such systems it is often difficult to develop a completely centralized picture. Examples of such systems are electricity networks or communication networks. Distributed agent systems are well suited to monitor and control such complex distributed structures. The “Ideal-System” developed by Onera and Alcatel-Alsthom is a typical example for a system for monitoring and error diagnostics in telecommunication networks [FER01, p.55]. It defines three kinds of agents: observers localize breakdowns and indicate errors; update agents ensure the coherence of actual network state data and the state data of the agents; maintenance agents, finally, know how to carry out tests and repair network components. The agents operate in a blackboard architecture, which consist of five modules: a communication module for the administration of communica-

tion protocols, an expert module with the knowledge to control networks, a cooperation module for the administration of contact tables and modeling of dialogues, a module with the graphical user interface which enables the user to monitor the activities of the agent and to interfere, and a control module for the administration of the activities of all agents.

Distributed Methods To Solve Problems

Agents can be used in a much more basic way to solve a problem in case the problem is described precisely and all required information is provided. Examples are creation of timetables or the determination of the sequence of steps to find the way out of a labyrinth, to solve a puzzle or to prove a theorem. In this case neither the problem area nor expert knowledge is distributed. However, multi-agent systems make use of logic thinking to solve that kind of problems. The approach used is called *eco-problem-solving*. Further information can be found in [FER01, p. 482].

Multi-Agent Simulation

Simulations deal with describing the real world with theoretical models. Simulations are used in particular in physics, chemistry, biology, geography and social sciences to explain phenomenon and be able to predict them. Scientists develop models for a part of the reality and verify the correctness by executing them on computers. The models mostly describe mathematical relationships between variables. The most commonly used models are differential equations, changeover matrices, etc. These models are based on the interdependency of input and output variables. Numerical simulation techniques share several problems:

- Complexity and realisticness of parameters. Equations contain mostly parameters that can only be estimated imprecisely which causes them not to be compliant with reality.
- Problems with modeling actions. Mathematical models have significant problems to represent individual actions and individual decisions. Most collective phenomenon are the result of actions of many individuals that are being influenced by the decisions of other individuals.
- Qualitative deficiencies. Digital simulations can only take quantitative parameters into account and appear helpless when they are supplied with qualitative data. Mathematical methods quickly reach their limits when they are confronted with complicated habits of conduct of species.

Insofar as individuals, their behavior and their interactions can be modeled directly in multi-agent systems, completely new solutions to perform simulations can emerge. Multi-agents systems depend on the behavior of entities of the real world to be representable in a computerized form. Collective phenomena appear as the result of interactions between autonomous agents. In a multi-agent population model individuals are represented directly by agents.

Construction of Synthetic Worlds

The construction of synthetic worlds is important in research. With the use of such synthetic worlds it becomes possible to analyze in more detail specific interaction mechanisms, as it could be possible in the real world. The analysis of cooperation protocols or the understanding of the influence of individual behavior on the processes in a society can in many cases be ob-

served in more detail in virtual worlds. A typical problem is the hunter-prey-problem. The interested reader is referred to [FER01, p.62] for more information.

Collective Robot Systems

The goal in the construction of collective robot systems is not the construction of robots but the construction of groups of robots that must work together physically to reach a common goal. Contrary to the construction of synthetic worlds, robots in collective robot systems interact physically in the real world. The discipline of distributed robotics is divided into two disciplines: cellular robotics and mobile robotics. Cellular robotics is concerned with the modular construction of robots. A robot is regarded as a multi-agent system and each of its parts as a robot. To make a movement, cooperation of all parts, that is of all agents, is required. In mobile robotics at least two robots are used that must coordinate their moves to reach a common goal, for example controlling a building, discovering distant or dangerous areas, repairing canals. Robots playing soccer in a collective is a well-known competition for researchers developing mobile robot systems².

3.3 Agent Standards

The main idea of multi-agent systems is the distribution of complexity among several agents (that also may be intelligent) and the creation of value added through the co-operation of agents. For this interaction to be possible agents need to be able to communicate. Since agent systems have been developed for various platforms there is a need for standardization so that agents can interoperate in heterogeneous environments. The major standardization efforts are in agent communication languages and in architectures for agent systems, which are presented in the following.

3.3.1 Agent Communication Languages (ACLs)

Where agents need to communicate, they must individually "understand" some agent communication language (ACL). ACLs are designed specifically to facilitate communication between two or more agents. Agents need to communicate information, intentions, goals, and so on to other agents. As Cohen & Levesque [CL95] also note, an ACL should allow agents to enlist the support of others to achieve goals, to monitor their execution, to report progress, success, failure, to acknowledge receipt of messages, to refuse task allocations, and to commit to performing tasks for other agents. Most ACLs derive their inspiration from speech act theory [SE69], which was developed by linguists in an attempt to understand how humans use language in everyday situations, to achieve everyday ends. In speech act theory, human utterances are viewed as actions, in the same sense as actions performed in the everyday physical world (e.g., picking up a block from a table). For example, an utterance may be a request to turn on the heater or an assertion about the temperature. There are many different illocutionary verbs (or performatives) in English: examples include "request", "warn", "inform", and so on. The illocutionary verbs in a natural language like English typically correspond to performatives in some ACL.

For agents that need to communicate about some domain it is necessary to agree on some terminology to describe the domain. For example, an agent is buying a particular item from another agent: the buyer needs to be able to specify unambiguously to the seller the desired prop-

² see www.robocup.org

erties of the item, such as its size. The agents both need to agree on what ‘size’ means, and also what the terms like ‘centimeters’ and ‘inch’ mean.

An ontology is a formal explicit description of concepts in a domain of discourse (or classes), properties of each class describing various features and attributes of the class, and restrictions on properties [SWM03].

3.3.1.1 Knowledge Interchange Format (KIF)

KIF [GF92] is an interlingua for knowledge bases to inter-operate. KIF enables two agents to interoperate which for historic reasons or other reasons use a different knowledge database. Both knowledge databases can be translated to KIF, which is then used as a common database. KIF uses essentially the first-order predicate calculus with a Lisp-like notation. The following example taken from [FIN94] depicts the content slot of an untell KQML performative expressed in KIF:

```
(untell
  :      language KIF
  :      ontology motors
  :      in-reply-to S1
  :      content (= (val (torque motor1) sim-time 5)
    (scalar 12kgf)) )
```

Figure 3-1: Untell KQML performative expressed in KIF

The `content` slot of this message says that the `torque` of object `motor1` at simulation time (`sim-time`) 5 is 12 kgf. It is assumed that the ontology `motors` (mentioned in the `ontology` slot of the message) defines the terms `torque`, `sim-time`, `kgf` and so on. Note that KIF is not an implemented knowledge representation language. Rather, it is an implementation-independent interlingua that allows for precise and unambiguous knowledge representation.

KIF can also be used as a language to define ontologies. Properties of a domain and the relationships between the things in this domain can be declaratively expressed with the use of KIF. As KIF was primarily intended for processing by computers, software tools were developed that allow a user to develop KIF ontologies. The Ontolingua server [FA96] is a Web-based service that was intended to provide a common platform in which ontologies developed by different groups can be shared.

3.3.1.2 Knowledge Query And Manipulation Language (KQML)

The Knowledge Query And Manipulation Language (KQML) is an emerging standard developed as part of the Knowledge Sharing Efforts (KSE) of the DARPA [NEC94]. The KSE is a research program that investigates and develops software for coordination and knowledge exchange of knowledge based systems. KQML is a standard for the exchange of knowledge. At the heart of KQML are more than three dozen performatives that define the allowed "speech acts" that agents may use, and which provide the substrate for constructing more complex coordination and negotiation strategies [NWA96]. These performatives are grouped into nine categories as shown in Figure 3-2.

Category	Reserved performative names
Basic informational performatives	tell, deny, untell, cancel,
Basic query performatives	evaluate, reply, ask-if, ask-about, ask-one, ask-all, sorry
Multi-response query performatives	stream-about, stream-all
Basic effector performatives	achieve, unachieve
Generator performatives	standby, ready, next, rest, discard, generator
Capability definition performatives	advertise
Notification performatives	subscribe, monitor
Networking performatives	register, unregister, forward, broadcast, pipe, break
Facilitation performatives	broker-one, broker-all, recommend-one, recommend-all, recruit-one, recruit-all

Figure 3-2: Categories of performatives

These performatives are neither required nor sufficient for all kinds of agent applications. Nevertheless, developers are asked to make use of them to improve interoperability between applications.

```
(tell
  :      content "cost(bt, service-4, $5788)"
  :      language standard prolog
  :      ontology bt-services-domain
  :      in-reply-to quote service-4
  :      receiver customer-2
  :      sender bt-customer-services)
```

Figure 3-3: Example of a KQML performative

The KQML performative in the message in Figure 3-3 is `tell`, and the agent that is sending it seeks to inform some customer, `customer-2`, of a quote for performing some service, `service-4`, in reply to an earlier request from `customer-2`, to BT customer services. The content of the message is expressed in standard Prolog, and the ontology for BT's services domain is assumed.

3.3.2 Agent Platform Standards

The two major standardization organizations for software agents are the Object Management Group (*OMG*) with the standard MASIF, which focuses on the standardization of mobile agents, and the Foundation for Intelligent Physical Agents (*FIPA*). The FIPA is a non-profit organization from industry and business. Contrary to the *OMG*, FIPA is solely concerned with the standardization of agent technology.

3.3.2.1 MASIF

MASIF (Mobile Agent System Interoperability Facility) is a standard of the Object Management Group (OMG), which was founded in 1989. The OMG pursues the goal of developing standards for distributed object-oriented software architectures. The standard Common Object Request Broker Architecture (CORBA) as well as the Internet-Inter-ORB-Protocol (IIOP) was passed by the OMG. According to [PPW02] experience has shown that MASIF is not sufficient to implement agent systems platform independently.

MASIF is heavily based on CORBA and its services. In that way, MASIF is more an extension of the CORBA architecture. The Object Request Broker (ORB) is responsible that method sends of distributed objects are delivered transparently and independent from the implementation language. An MASIF agent has the inherent property of constantly being mobile. Beside the state of an agent its affiliation to some authority is important. The agent system provides the platform for creation, interpretation, migration, and removal of agents. In addition, it provides a list of all its agents or all agents of a specific authority. It consists of an arbitrary number of slots between which the agent can migrate. A region is an aggregation of agent systems, which servers primarily as a security domain to define permissions of agents.

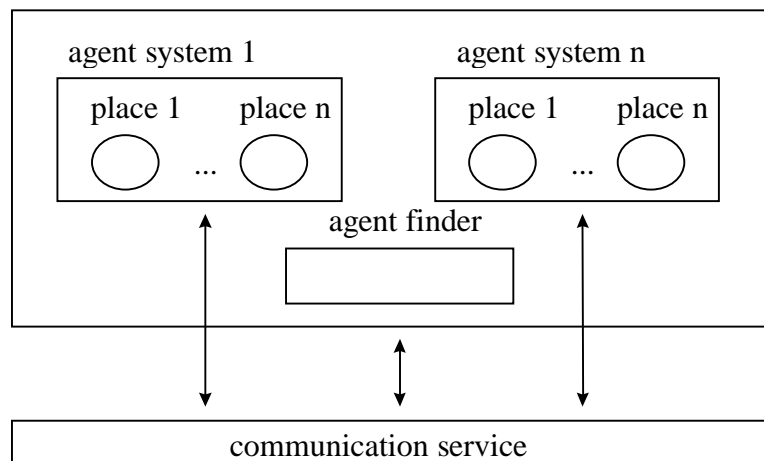


Figure 3-4: Structure of an MASIF agent platform

The agent finder is a directory service for agents, which is defined per region. The interfaces of the agent system and the directory service are defined as IDL interfaces. The IDL (Interchange Definition Language) is a definition language that serves for the definition of objects with the services they provide, which is independent from the implementation language. The IDL interface for the agent system is named `MAFAgentSystem`, for the agent finder it is named `MAF-AgentFinder`. In addition, MASIF defines several data structures, for example for agent names and description of agent systems.

3.3.2.2 FIPA

From the standards described in this document FIPA is the only standard that takes a general approach for the standardization of the agent infrastructure. In FIPA the transport service for moving agents within the same platform is not specified. In that way, rather than being direct opponents, FIPA and MASIF are more complementary in their functionality. The core component of the FIPA model is the agent platform. The agent platform defines the context, in which an agent may be created and provides the required infrastructure. FIPA defines three essential services that form the Agent Management Services:

- The Agent Management System (AMS) provides essential functionality for the creation and removal of agents. It provides a naming service for all agents available on a platform.
- The Directory Facilitator (DF) is a directory service for agents. Agents may register the services they offer with the directory service or search for services offered by other agents.
- The Agent Communication Channel (ACC) carries out communication between agents that reside on different platforms. The ACC is based on the Internet-Inter-ORB protocol (IIOP) of the OMG. FIPA defines an agent communication language (ACL) based on IIOP for the communication between agents.

An additional essential component of the FIPA architecture is the definition of agent interaction with other software systems with the use of Wrapper Agents and the Agent Request Broker.

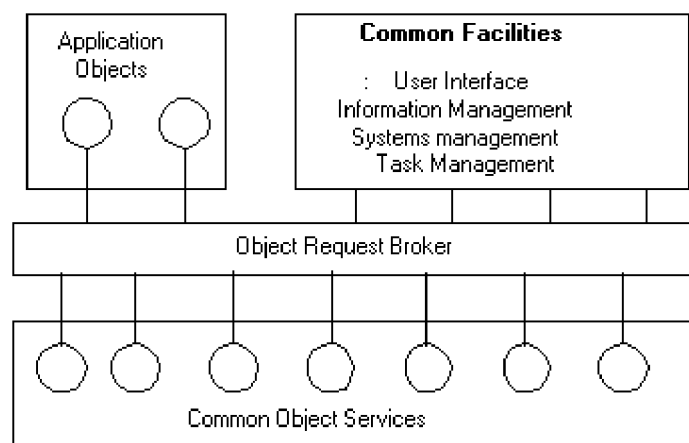


Figure 3-5: The OMG Object Management Architecture

Resource agents encapsulate access to external services and integrate other agents using an ACL interface. The Applications component defines ontologies for selected application areas as well as specifications for services of this domain.

3.3.3 Agent Platforms and Systems

An overview of widely known agent platforms available in the marketplace can be found in [BB01]. The following section describes a few agent platforms in brief.

Aglets [LO98] is a MASIF compliant agent framework developed by IBM Japan. Agents in Aglets are not CORBA objects.

JADE is a FIPA compliant framework for the development of agent-based applications that is widely used in industry and academia developed at the University of Parma, Italy. The goal of JADE is to simplify the development of agents build upon a set of FIPA compliant services and agents. JADE can be seen as an agent middleware, which provides an agent platform and an application-independent development environment.

SeMoA is a mobile agent platform developed at the Fraunhofer Institute in Darmstadt, Germany. It is a Java-based mobile agent platform that takes special care of security problems specific to mobile agents.

4 Overview of Janet

The idea of Janet is to develop a system that provides workload balancing on application-level. Workload balancing has mostly been implemented on operating system-level as in distributed operating systems such as Amoeba [TAN01], Sprite [OUS88] and others. Schnekenburger writes in [SCH95]: “Load distribution is a central problem for parallel programming in distributed systems. Load distribution has to manage the assignment of service demands of a parallel program to distributed resources of the system. More generally, load distribution has to manage the mapping of *distribution objects* to *distribution units*. Distribution objects may correspond for example to processes, threads, or data structures whereas distribution units may correspond for example to computing nodes or application processes”. The idea of Janet is to develop a simple system that provides an object space for developing distributed programs in heterogeneous object spaces. Automatic distributed execution is an application of Janet that is intended to proof the usefulness of the Janet distributed programming object space and to offer a useful system for workload balancing in Java.

As simple as possible, but not simpler.

To use this quote by Albert Einstein, Janet is intended to be made “as simple as possible, but not simpler”. This was also the motto of Niklaus Wirth in the development of the Oberon programming language [RW92]. He says in [RW92] that it is good engineering practice “to strive for economy of means and simplicity of solutions”. One reason for building Janet up from scratch is to use it as a learning vehicle for building an entire system rather than adding new parts to an existing system. The intention here is clearly in having an opportunity to develop such skills mentioned by Wirth rather than developing the “ultimate multi-agent system”. Another reason is that a lot of overhead of existing agent platforms can be avoided if a new system is built using simple means. This is important with regard to workload balancing where a small overall performance penalty of the load balancing system is required for the gains in time out of parallel execution of jobs not to be undone by the sluggishness of the workload balancing system itself.

4.1 Layers in Janet

The goal of this work is to develop a system that can automatically re-distribute jobs to other workstations to balance workload. The core agent system can be separated from the system for workload balancing. This way the core agent system can be used for other purposes as well. It is always good design practice to separate concerns and to strive for a low level of coupling. For this reason, Janet is split into two major layers: Janet.CAS and Janet.ADE which are described in brief in the following.

Janet.CAS

Janet.CAS (Cooperative Agent System) is the core layer that provides a distributed agent-oriented platform where agents communicate through the exchange of commands for which they chose interpreters of their own to execute them. Agents are defined by applications that reside on nodes. All nodes that are connected with each other form a cluster. Janet.CAS provides a platform for which users can write applications. An application is a collection of capabilities. Capabilities consist of interpreters that execute commands. Interpreters constitute the functionality of an agent.

Janet.ADE

Janet.ADE (Automatic Distributed Execution) is based on the Janet.CAS layer. It is the layer that implements automatic workload balancing. Workload balancing is realised using automatic distributed execution of commands. This term originates from distributed operating systems that are based on automatic distribution of operating system-level jobs to balance workload.

4.2 Semi-Transparent Load Balancing

Load balancing requires that a process that was started at a machine can be suspended and can be moved to some other machine where its execution resumes. This way it is possible for the system to readjust to dynamic load changes. The solution implemented in Janet.ADE cannot suspend threads and move them to other machines, as this would require additional information that can only be obtained from the operating system or – in case of Java – from the Java virtual machine. The Java virtual machine can be changed to obtain this additional information as this has been done in the Plurix-Project at the University of Ulm, Germany [SM01]. Such an effortful undertaking is beyond the time and resource frame for this work. In Janet.ADE threads are therefore replaced with the concept of a command. The user is required to provide some assistance when saving the command context before command migration. This way, load balancing can still be implemented on application-level in Java with only a small sacrifice in transparency.

4.3 Architecture

Architecture is one of those terms that is widely overused but poorly defined in software engineering and computer science. This section describes the high-level software framework within which Janet is implemented.

4.3.1 Development Platform

Java has shown to be a good choice for the development of many agent-based systems such as *AgentBuilder*, *AgentOS*, *Aglets*, *JADE*, *JACK*, *SeMoA*, and many others. One reason for this is Java's ability to span heterogeneous environments. For the implementation of mobile agent systems being able to load classes at runtime, which is straightforward in Java, has proven to be very helpful. This ability is important for Janet as well, because interpreters, that execute commands, are dispatched and loaded at runtime. Java has been chosen as the development platform for Janet for the reasons mentioned above and for other reasons as well. For example, to be able to finish the system within the given time it was important to choose a development platform with good developer productivity.

4.3.2 Middleware

To make nodes in a cluster communicate with each other a middleware system is needed. Java Remote Method Invocation (*RMI*) was chosen as the middleware system. Other middleware systems such as *CORBA*, *JMS*, or *XML-RPC* are also possible choices. The advantage of using

RMI is that it is easy to use and thus creates little additional effort to make nodes communicate. This makes it possible to concentrate on design and conceptual issues of the system. If it turns out later that a high-level middleware system such as CORBA is needed, migration from RMI to CORBA would be relatively effortless as these systems are based on a similar user interface. RMI has also been used for the development of JADE, which is widely used in academia and research. Performance measurements of JADE [VQC02] have shown that RMI is well suited to achieve good performance in a distributed agent-oriented system. The same holds true for JavaSpaces, which has been developed with Java and RMI as well. Java RMI, being a Java-centric middleware, is not a good choice to achieve high interoperability with non-Java based systems whereas XML-RPC has been specifically designed for interoperability. XML-RPC is especially interesting since it is widely used to exchange data between web services and clients. With the number of web requests for many sites becoming very large load balancing of web services has become an important issue. It would be an interesting task to develop an XML-RPC interface for Janet to be able to use Janet for load balancing of web requests. However, RMI has been chosen over XML-RPC, because of its simplicity and ease of use, which is important to be able to finish the system within the given time frame. Converting Janet commands objects to XML strings is simple since commands only carry parameters and no code. For this reason, concept and design of Janet would need no modification for any later support of XML-RPC.

4.3.3 Agent Platforms

With the rising popularity of agent-based distributed applications several agent platforms have been developed. JADE has already been mentioned as an agent platform that has been used successfully in several academic projects and research projects. Automatic distributed execution could be implemented using an existing proven agent platform or a shared memory system such as JavaSpaces. An important requirement for a load balancing system is that it is efficient and lightweight. A system that provides workload balancing must not be that inefficient that it counteracts the benefits of workload balancing. When taking the selection of an agent platform for the development of Janet into consideration best possible performance is an important issue. For this reason, it was decided to develop a simple agent platform for automatic distributed execution rather than using an existing heavyweight agent platform. This makes sure that the performance issue can be kept under control. Another reason is that several ideas for the development of an agent platform already existed and automatic distributed execution appeared to be a good reason to turn these ideas into a working agent system.

4.3.4 Simplifications

The goal of this work is to develop a fully functional system that is conceptually clean. It is certainly not possible to develop a system that is complete and offers advanced features that would take a team of developers several years to implement. To be able to concentrate on the main theme of this work, several simplifications had to be defined.

Concentration on Realization of Conceptual Ideas

There is no support for persistence, transactions, or fault tolerance. Security is being taken care of in the way that the system ensures that user-level applications run in a protected environment and cannot interfere each other, nor communicate directly with each other. However, there are no built-in mechanisms for the authentication of sending and receiving commands, which are not encrypted, either. SeMoA from the Fraunhofer Institute, Darmstadt, Germany,

was considered to be used as a secure messaging server for Janet commands. As described in [PPHG02] SeMoA is a Java-based mobile agent platform that protects mobile agents against malicious hosts. Authentication and encryption could be delegated to SeMoA. Because of the SeMoA licensing terms, every user of Janet would have to obtain a license from the Fraunhofer Institute to use SeMoA in conjunction with Janet. As Janet is intended to be a free tool with a liberate license this was considered unacceptable.

No Built-In Support for an Agent Communication Language

For agents to be autonomous they are required to be able to choose their own actions in response to some message they received. Wooldridge states in [WO02, p.164]: “We have two agents i and j , where i has the capability to perform α , which corresponds loosely to a method. (...) There is no concept in the agent-oriented world of an agent j ‘invoking a method’ on i . This is because an agent is an autonomous agent: it has control, over both its state and its behavior”. Janet takes care of this fundamental requirement of the autonomy of agents by executing agents as Java objects that run in their own threads and by decoupling commands from interpreters. An agent sending a command to another agent cannot enforce any action on the recipient agent. The recipient agent decides on his own how to react to this command by choosing the interpreter it considers appropriate.

An agent communication language (ACL) is some lingua franca designed specifically to facilitate communication between two or more agents. Agents need to communicate information, intentions, goals, and so on to other agents. Several ACLs have been developed such as KIF [GF92], KQML [FFMM94], or the FIPA ACL. If agents are to communicate about some domain, then it is necessary for them to agree on the terminology they use to describe this domain.

As Cohen and Levesque have argued in [CL95] KQML, for example, has some deficiencies. The KQML specification is in some points vage and ambiguous. It has no formal semantics, as this is commonly the case for programming languages, which means that it is not guaranteed that a performative is interpreted in the same way by all agent systems that use KQML. ACLs still seem to be in the flux of development and refinement. New ACLs are occurring that seem interesting such as *OWL* (Web Ontology Language) by the W3C group. From this point of view, deciding on an ACL standard appears premature. There seems to be a point in waiting how the development of ACLs is going to continue. For this reason, it was decided that Janet, for the time being, does not make use of an explicit ACL or ontology. Janet system commands are required to know implicitly about the terms they use and how the designer specified them. However, Janet users that develop user-level applications are free to choose their own ACL making use of Janet commands as messengers for ACL statements and develop their own ontologies.

4.4 Distributed Systems and Janet

It is necessary to position Janet against other kinds of distributed systems in order to verify that a system like Janet has applications for which it is well suited and therefore can justify its place.

4.4.1 Linda-Style Tuple Space Architectures

JavaSpaces has been used as a platform for the development of agent-based systems. Engelhardt and Gagnes [EG04] have used JavaSpaces to create an adaptive distributed system where agents adapt to changing demands placed on the system by dynamically requesting their behavior from a JavaSpace. They argue that: "... using JavaSpaces technology is a simple way to create adaptive systems, due to JavaSpaces' ability to support messaging as well as distribution of agent behavior".

Wang has used JavaSpaces to implement a mobile multi-agent system [WA00]. An advantage of using JavaSpaces is that distribution of agents is obtained almost for free. This system was first developed using the Aglets agent framework [LO98] and was then migrated to JavaSpaces. Wang writes that: "... experiences show that JavaSpaces can well be used to provide the same functionality as that provided in other mobile agent frameworks like, e.g. Aglets. (...) In addition, the API in JavaSpaces is very high-level, making it efficient to provide distributed services".

Chen has explored JavaSpaces to coordinate multi-agents [CH02]. He argues that: "... the emerging JavaSpaces technology provides a convenient, yet flexible approach to agent coordination in a multi-agent system object space".

The successful use of JavaSpaces to develop agent-based systems as described in the papers mentioned above makes JavaSpaces an interesting system to use as the implementation technology for Janet. However, it is a drawback that the tuple space approach does not provide a familiar computing objects space. Users are forced to adopt the Linda tuple space computing model, which is a producer/consumer model. A consumer spins off a number of producers, which enter the tuple space, perform their work, and the consumer checks back for results. In JavaSpaces all objects in the JavaSpace are in serialized form. They cannot invoke methods without being first extracted from the JavaSpace. Furthermore, JINI, on which JavaSpaces is built, is a heavyweight API, which simple applications might not need, and it makes using JavaSpaces difficult.

The main drawback of JavaSpaces is that a final decision for the middleware technology is made implicitly with the selection of JavaSpaces. Since JavaSpaces makes use of a very specific computing model it becomes very effortful to change from JavaSpaces to some other middleware system (such as XML-RPC for Web requests). Janet is intended to be middleware independent so that the middleware used to transport commands from agent to agent can be easily exchanged. Middleware independence is considered important, as the past has shown that middleware technologies are in a state of constant change. Adaptability to new emerging technologies such as XML-RPC or SOAP/Axis was given preference. For performance reasons, Janet is intended to perform well and be lightweight as well. This is a requirement for the implementation of a load balancing system to perform well. Simplicity and strive for a lightweight system is another reason not to chose JavaSpaces for the development of Janet.

The disadvantage of not using JavaSpaces is that without a shared memory system agents are restricted to pure message passing as a means of communication. However, since JavaSpaces is based on Java RMI message passing, it is also possible to develop a simple shared memory space for Janet if required.

4.4.2 Distributed Operating Systems

The key rationale of Janet is to build an inexpensive system in little time with which it is possible to do distributed programming in a heterogeneous object space without having to use specialized systems. Specialized distributed operating systems like Amoeba or Sprite are very impressive and some of the most demanding systems to build, but from the beginning they were intended to be research vehicles that will be discontinued one day. Commercial solutions that offer protection for earlier made investments like cluster systems that are extensions of existing commercial UNIX systems must be used constantly to pay off. Janet offers an inexpensive alternative for tasks that need to be solved with the means of distributed programming but do not justify for the massive investment into commercial high-end cluster systems.

4.4.3 Parallel Programming Languages

Other specialized systems such as programming languages for parallel programming - like Occam, Concurrent C, Pascal-FC, pC++, and many others - have the disadvantage that they create their own world. For purposes such as supercomputing there is no need to be able to bridge from specialized parallel programming languages to programming languages for general application programming like Java and living in a world separated from others is therefore not a problem. But there are applications where a rift between programming object spaces would create too big an expenditure and where systems such as Janet come into play. There are other ways to avoid this rift, for example by extending the Java VM to build in support for distributed shared memory (DSM) as this has been done in the Plurix project [STS98] at the university of Ulm, Germany. Since memory is shared in Plurix there is no problem of passing messages beyond process boundaries. This would make the implementation of Janet a lot simpler since communication between agents on different computers would not be an issue. While Plurix is more intended to be a distributed operating system Janet is more geared towards supporting development of distributed end-user applications. For this reason, Plurix and Janet are more complementary to each other rather than competing against each other. DSM provided by Plurix in conjunction with the facilities of Janet for developing multi-agent applications could be well-suited complements just as in conjunction with other agent systems such as JADE or Aglets.

4.4.4 Grid Computing

The idea of grid computing is to integrate computers in a seamless and transparent way to create a single integrated system. This involves the sharing of data and all kinds of resources such as CPUs, memory, storage devices, printers, and others. On the contrary, the idea of Janet is to develop a fundamental system for distributed programming that uses an open pluggable architecture for users to add in their own applications rather than integrating other systems. Janet or some of its concepts may be used as a building block for building larger system for grid computing.

5 Janet.CAS

This chapter begins with the description of the conceptual design of Janet.CAS. The section about conceptual design provides an overview of major design decisions and their realization which form the basis of the Janet.CAS layer. The second part describes the detailed design and implementation where solutions were developed to turn conceptual ideas and decisions into reality. When developing a system it is useful to implement an application that serves for the validation of its concepts. Such an application, called a driving application in this paper, helps to point out missing functionality and to become aware of inappropriate design decisions so that these decisions can be corrected in an early project state. The driving applications developed are an application to calculate Fibonacci numbers in a distributed environment and an application that provides remote logging of log messages. These driving applications are presented in the third part of this chapter. They are supposed to give the reader an introduction into how to develop applications using Janet.CAS. This chapter concludes with the presentation of measurements. Travel times of Janet.CAS commands and the time required to process them is measured in order to obtain an estimate about performance characteristics and scalability of the system.

5.1 Conceptual Design

The conceptual design describes the major conceptual ideas that form the basis of Janet.CAS. These concepts define high-level design approaches on which the entire system is based.

5.1.1 Commands and Interpreters

The core concept in Janet.CAS is the communication between agents through the exchange of commands. Exchange of commands is always asynchronous and therefore is communication between agents. A command is an object that transports parameters between agents and lets the recipient agent know what event has occurred. It is thinkable that a command transports executable code but it is not executable itself. Only the interpreter is executable. The recipient agent invokes an interpreter it chose of its own to handle the command. In the following, the expression is used that “an agent executes a command”. This is a shortcut for not having to state repeatedly, that an interpreter associated with some command is executed. The command-interpreter pair in Janet.CAS is very similar to an event-handler pair. Since the exchange of commands between agents has been developed as an extension of the command pattern as described in [GA95] this text uses the former terminology. Furthermore, commands are a way to notify a specific recipient that an event has occurred rather than a notification to all objects that have subscribed to a certain event and are then notified indirectly by an event notifier. For the latter purpose a separate event-notification mechanism was developed. The idea of enhancing the command pattern by adding an interpreter object was initially developed in a student’s project at the University of Applied Sciences in Darmstadt. In general, to use the command pattern for distributed computing seems to be an obvious idea. Several other persons have chosen this approach independently from each other. Two authors that use this approach in their books (though on a simpler level than in Janet) amongst others are Grosso [GR01] and Farley [FA98].

The principal idea of agents is that they are autonomous and therefore make their own decisions. If a traffic light changes from green to red an agent controlling a car needs to decide of

its own how to react. It may decide to stop the car. However, if it controls an ambulance on an emergency mission it may decide to turn on the siren and cross the red light. The command-interpreter pattern allows for the implementation of this polymorphic behavior as an agent defines itself which interpreter to invoke when a command is received. Upon receipt of a `TrafficLightChangedToRedCommand` command, a car agent would invoke the interpreter `myNamespace.car.TrafficLightChangedToRedInterpreter` whereas an emergency vehicle would invoke the `myNamespace.emergencyVehicle.TrafficLightChangedToRedInterpreter`. The split between command object and interpreter object is necessary to ensure the autonomy of agents. If an agent were simply told by a command, sent by some other agent, what to do, it would only be an object receiving a message rather than an autonomous agent receiving an event and reacting to it in a way it assumes to be appropriate for its purposes. In addition, the split between command object and interpreter object makes sure that an agent only executes code that was defined for it and is therefore save from remotely received commands making it behave as intended by some potentially malicious agent. Due to the separation of concerns between commands and interpreters the command objects need not necessarily be a Java object. It could be encoded in XML, which would make integration with non-Java systems possible (XML-RPC, SOAP).

5.1.2 Applications and Capabilities

Before we can carry on the concept of an application and a capability has to be introduced since these concepts will be referred to later on. As already mentioned earlier, Janet.CAS is intended to serve as a simple agent platform that allows the user to plug in applications developed on their own. The application object in Janet.CAS allows the user to register her application with Janet so that the agents needed by these applications will be instantiated and started. An application consists of several capabilities. A capability defines a set of interpreters. Applications with their capabilities can be defined programmatically or in XML node descriptor files that are parsed when a node is started up. Janet.CAS itself defines nodes that have different roles in a cluster by specifying the applicable interpreters for the core capability of the system application individually as required by the role of the node. In the same way, a user would plug in her applications into the Janet.CAS agent platform. By specifying the names of agents needed for a capability the user defines which and how many agents will be created by Janet.CAS to serve the application. When an agent sends a command to another agent it has to specify the name of the destination agent, its application name and the name of capability that defines the interpreter to execute the command.

Figure 5-1 on page 37 shows the classes `Application`, `Capability`, and `ObjectSpace` with their main attributes and operations. An application is uniquely identified by its name. It has several capabilities and its own object space. The application's agents may use the object space to store permanent information that needs to remain present after an interpreter has finished execution, which allows the agent to keep track of its internal state. Every capability defines 1 or more agents. An agent may only execute commands that are sent to the capability they are associated with and may only send commands to agents that belong to the same capability and application (which may reside on physically separated sites, but need to have the same names and list of interpreters in each capability). This is a kind of protection that makes sure that agents cannot invoke actions on agents of other applications (e.g. denial of service attacks). Agents of different applications can only communicate through events. Agents decide on their own, which events they want to register for. A capability has a set of interpreters. Each interpreter knows the names of the commands it is supposed to interpret. When an interpreter is executed the message `execute` is sent to it with the command as an argument contained in the

`CommandAccessor`. The `CommandAccessor` provides the means for the interpreter to access its capability and application and shields the node from unprivileged access. Through the `CommandAccessor` an interpreter can also access the node itself as far as access is granted. An interpreter executed by a system agent has full access to the node's inner objects, whereas access is restricted for user-defined applications.

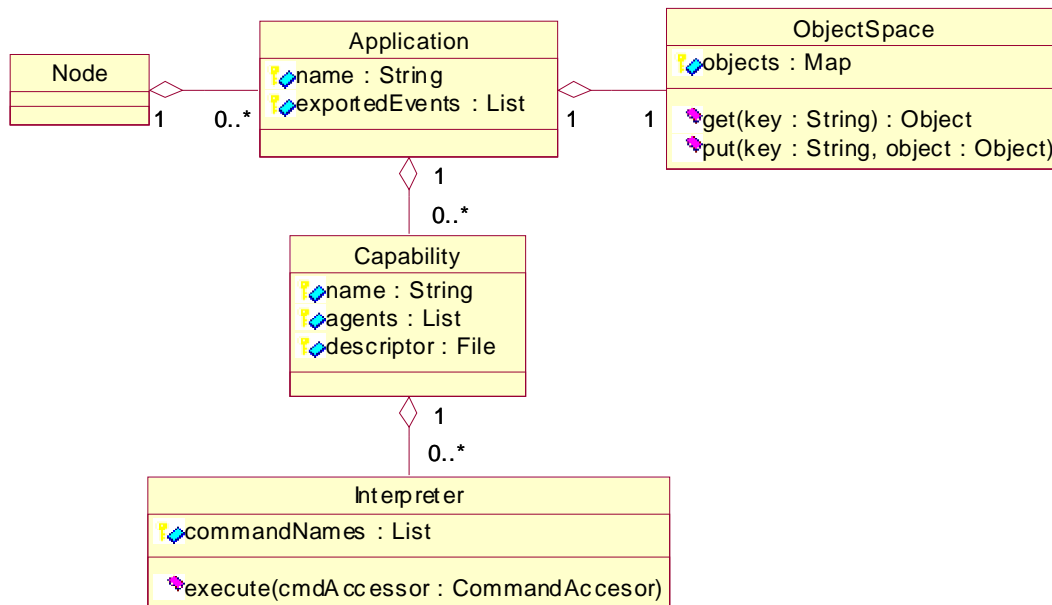


Figure 5-1: Applications and capabilities

The idea of capabilities is inspired from capabilities in Amoeba [TAN01]. Whereas in Amoeba capabilities are descriptors that define user privileges, in Janet they define an agent's executable part. It is thinkable to let an agent dispose of several capabilities. However, this would make the application logic considerably more difficult, especially when evicting commands to other nodes in order to balance load. The agent an evicted command is transferred to needs to have the same capabilities as the agent from which the command was evicted. Otherwise, there is no way to dispatch the interpreter associated with the evicted command at the destination site. In order to strive for simplicity first, an agent can only have one capability, which will remain as is unless evidence appears that shows that it is not possible to cover essential requirements using the initial approach. An application can only be instantiated once. However, the number of agents per capability is unlimited.

Applications and System Application

There is a special application, which is called the system application. The system application defines a capability to run a Janet node, which is called the core capability. An agent that is part of the system application has privilege to access all inner objects of the node.

5.1.3 Schedulers

A common approach to make an agent able to react independently to events is to make it run in its own thread. A scheduler is used to execute interpreters, associated with received commands, on behalf of the agent. Implementation-wise the scheduler is an active object, which means that it runs in its own thread, whereas the agent object merely serves as an anchor object to accept commands and to expose the public interface of an agent. When an agent receives a command from another agent it places the command into the command queue of its scheduler. As long as the scheduler queues contain any commands they are executed one after the other in a consumer-producer-like fashion. Each interpreter runs to completion. The scheduler looks up the interpreter associated with the command, passes the command over to the interpreter - so that the interpreter can extract the required input parameters from the command - and starts it.

Several agents may share the same scheduler, which improves performance considerably in case several thousand agents exist that are running concurrently as shown in [BRHL99]. The more agents share a scheduler the more efficient execution. However, sharing a scheduler reduces the degree of pseudo-parallel execution of commands since a scheduler only executes one command at a time. For shared use of a single scheduler to be efficient, it is important to define commands that execute fast so that crowding out of other application commands is minimized. This also minimizes the likelihood that several commands have to wait long until they can enter a critical section in case it is already occupied by another command, which furthermore improves performance. Since always one command is executed at the same time by one scheduler the number of potential race conditions when accessing shared resources is thereby reduced which simplifies concurrency control and reduces the deadlock potential.

5.1.3.1 Arbitrators and the Supreme Scheduler

Java uses a round-robin thread-scheduling mechanism [OW97]. This means that a thread can interrupt other threads that run at lower priority. However, the behavior of the Java scheduler is not specified in case of several threads running with the same priority. The Java scheduler switches between them but only if the operating system uses time slicing. If it does not, the next thread can only run when the previous one has finished. This is a problem as a thread can cause other threads at the same priority to starve. Janet.CAS as a simple agent platform has to guarantee that an application cannot cause starvation of other applications. This is ensured by the arbitrator that grants each application schedulers a certain amount of time for execution on a time slicing basis. The arbitrator runs at a higher priority than the application schedulers, which makes sure that it always runs when required. The arbitrator controls the application schedulers by changing the priorities of the Java threads that run them. The priority of the single application scheduler that is given the current time slice is increased while the priority of the previously active application scheduler is first decreased.

The arbitrator starts the schedulers of an application once an application scheduler receives the first command. The user cannot start or stop any scheduler. When an application has been registered with the system, the arbitrator may start the schedulers associated with the application at any time but latest when it receives the first command. This makes sure that an agent can always rely on a command to be interpreted by the recipient agent. If this were not the case the application could easily get stuck. Checking whether the application scheduler of an agent is running before sending a command to it would significantly complicate the design of the system and user-level applications. For this reason, registered applications become immediately

visible to all the nodes in the cluster and their agents may always be assumed to be running. Consequently, when an application is deregistered, it is no longer visible to nodes in the cluster.

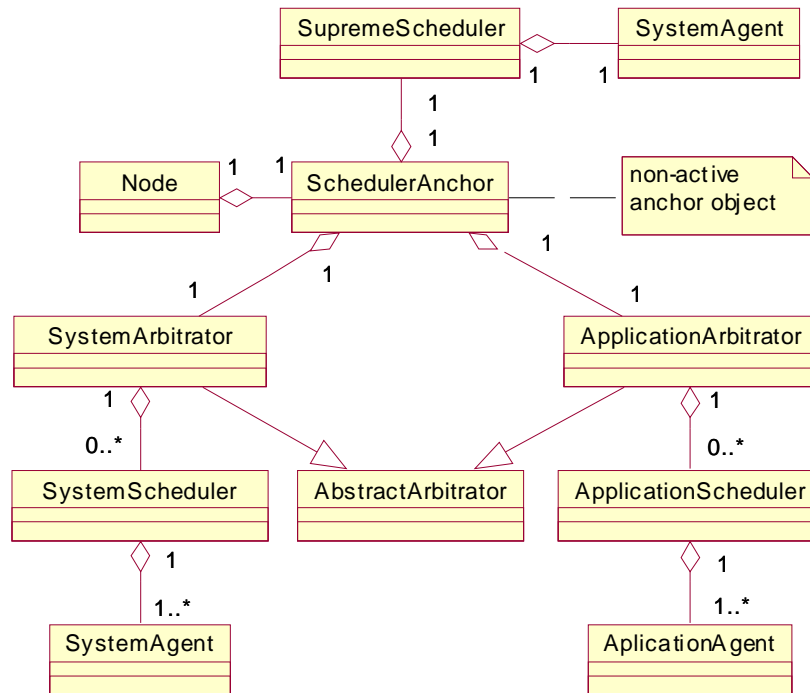


Figure 5-2: Arbitrator class diagram

As shown in Figure 5-2, there are two arbitrators: one to schedule application schedulers and another one to schedule system schedulers. The **SupremeScheduler** is a singleton active object that runs at the highest priority of all schedulers. The supreme scheduler needs to run at a higher priority than all other schedulers so that it can preempt commands of other schedulers and execute its own commands immediately, which is required in some cases for the system to remain responsive. Otherwise, a situation is thinkable where some agent waits for the system to provide some service, which it cannot provide immediately because it is waiting to receive a time slice from an arbitrator.

The system schedulers run at a higher priority than the application schedulers, but at a lower priority than the supreme scheduler. System commands that need not interrupt other commands immediately in order to execute are passed on to a system scheduler. Every agent that has a capability, which is part of the system application, is assigned to a system scheduler. System agents are rarely needed. In fact, they were only introduced to be able to interrupt system commands to implement command eviction and their cancelation. In the following, the term system agent refers to the supreme agent if not explicitly stated otherwise.

5.1.4 Event Notification

Commands in Janet are always asynchronous. They tell an agent that something has happened, but don't tell it what actions to take. This alone is not sufficient to serve as an event-dispatch mechanism since commands are always sent directly to a specific agent. But event and receiver of the event notification need to be decoupled. Events that may occur must be published giving every agent interested in being notified about the event occurrence the opportunity to register for the event. Event registries are a common approach to propagate messages to objects that were not known to the event notifier at compile time. What is specific to Janet.CAS is the question where the event registry should reside and who has to pay for the CPU cost of the execution of the event handler. The cost should be paid by the agent that registered the handler, not by the agent that signaled the event. For that reason the handler is wrapped into a command and sent to the agent that registered for the event. The command containing the handler will then execute the handler and the CPU costs will necessarily have to be paid by the agent that registered for the event. As a result of the handler being executed by an agent as a command, which is executed asynchronously, execution of the event handler is asynchronous as well.

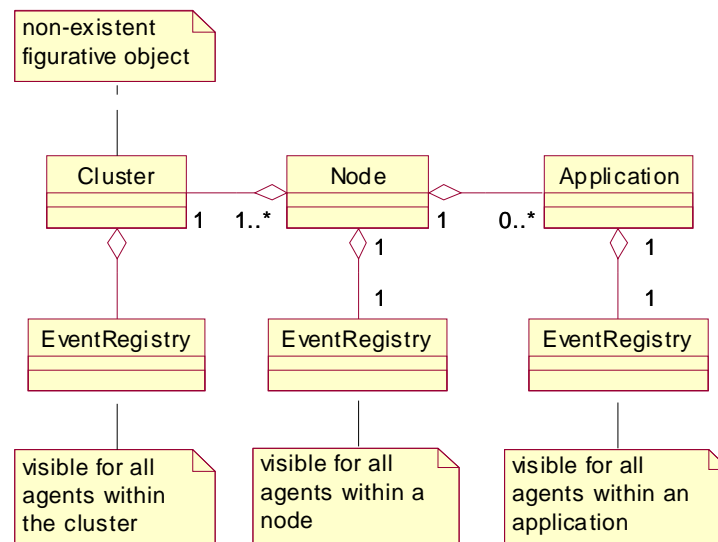


Figure 5-3: Event registries and their scope

Every application has an event registry of its own for events that are not of concern to other applications and which they should not know about. For every node is a node event registry through the use of which agents of different applications residing on the same node may exchange events. Finally, a cluster event registry exists for event registration and notification across node boundaries. An overview of all event registries is displayed in Figure 5-3. These event registries are explained in further detail in the section that describes detailed design and implementation.

5.1.5 Nodes and the Central

The central is a special node that takes care of the management of the cluster. The node is identical to all other nodes in the cluster except that its system application defines a core capa-

bility supplied with the respective interpreters to run a central. The central keeps an image of the cluster with its nodes. When a new node is started up in the cluster it registers with the central, which in turn notifies all already existing nodes in the cluster about the new node. Analogously, when a node shuts down the central removes it from its cluster image and notifies all other nodes about the event. During shutdown race conditions may occur in case two or more nodes are carrying out the shutdown process at the time. The central is needed to synchronize node shutdown.

After a node has registered with the central it registers with all other nodes in the cluster as well. During the registration process nodes exchange descriptors of their applications. In doing so, every node knows about all the applications that are registered with all other nodes in the cluster with their capabilities and agents. An agent is therefore able to look up immediately all existing agents in the cluster that have a specific capability. There is no need to ask a central registry, which takes time. Instant agent lookup is important for being able to transfer commands from a heavily loaded agent to a lightly loaded agent quickly. Nodes may shut down or start up any time, applications with their agents may be newly registered or deregistered while a node is running, agents may be created or closed down any time. This requires agent lookup to be dynamic. Hence, agent lookup cannot be made static in order to optimize performance. Since nodes exchange application descriptors with all other node in the cluster at startup time, startup of a node is a period where command traffic in the cluster is high.

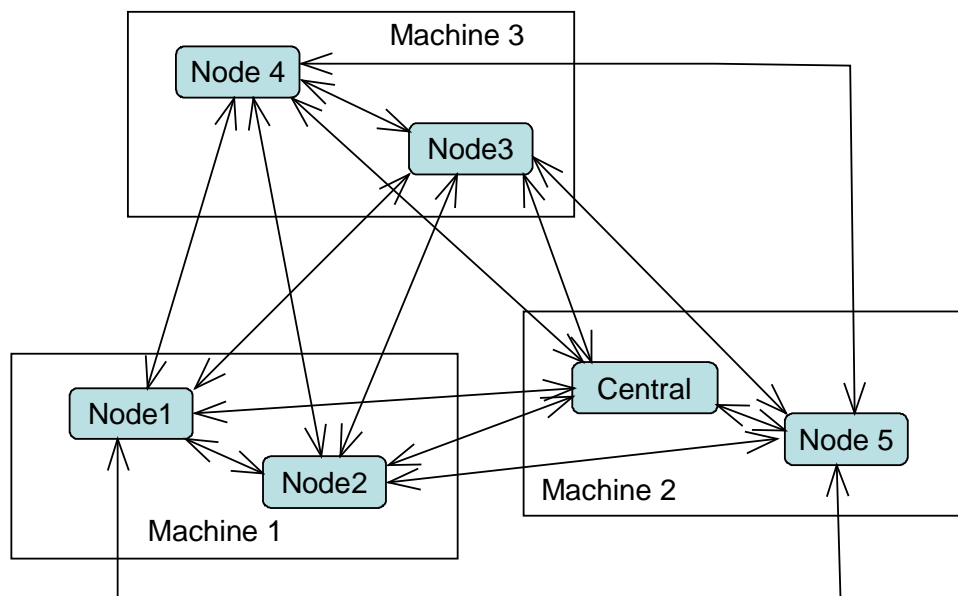


Figure 5-4: Sample cluster with the central and completely connected nodes

The central is required as well to interact with the RMI registry, where RMI node server objects are registered to be visible for other nodes. For security reasons, a RMI server object may only be deregistered from the registry on the same workstation where the RMI registry resides. The central deregisters nodes that may reside on other workstations than the central on their behalf. Another solution would be to let all nodes on the same workstations have their own RMI registry, rather than having a single RMI registry at the location of the central. But this

would result in the user having to specify for every single node in the cluster on which workstations the registries of all other nodes reside to be able to look them up from their RMI registries. Since other middleware might have similar restrictions as RMI for security reasons or other reason, the presence of the central offers the required flexibility that might be needed when plugging in other middleware.

Another reason for the central to exist is to provide some flexibility in case clusters need to be connected. Centrals could help making the connection process of all nodes in every cluster simpler and more efficient.

Figure 5-4 on page 41 shows a sample cluster with the central and several nodes. *It is important to realise that there can be several nodes on one machine and not only a single one.* This is because the user is not restricted to only starting up a single Java virtual machine, but may need to start up several ones, and for each Java virtual machine the user wants to make use of Janet or parts of it. For example, distributed number crunching with more than one node per workstation makes little sense when every machine has a single CPU. On the contrary, one might consider a Janet application that wants to make unused heap memory of a Java virtual machine running a Janet node available to other nodes. If there are several Java virtual machines on one workstation, it makes sense to make the unused heap memory of every Java virtual machine on that workstation available in the whole cluster or for a specific set of nodes.

5.2 Detailed Design and Implementation

Detailed design is concerned with finding detailed design solution to realize conceptual design decisions. This section describes how the conceptual design has been turned into reality and describes the implementation of the most important parts.

5.2.1 Schedulers

Schedulers are active objects, which own their own thread. This thread only executes the methods invoked on an active object by other active objects. Instead of using methods, Janet makes use of the command pattern where agents send commands to other agents. Following the agent-oriented paradigm only agents are active objects in Janet and only agents communicate with other agents. In Janet agents make use of their scheduler, which they may share with other agents for efficiency reasons, for the execution of the commands they received from other agents. This makes scheduling an important task in Janet. Explanation of the detailed design of schedulers is therefore presented first.

All agent schedulers in Janet are derived from class `ActiveSchedulingObject` which itself is derived from `ActiveObject`. These two classes are abstract general-purpose classes defined outside Janet in the subsystem `Objectscope.Common` in the package `org.objectscope.threads` that provides the basic functionality for running an object in its own thread. Class `ActiveObject` has the functionality for starting and stopping the thread owned by the active object and changing its priority. The methods in the Java thread class for thread suspension and resumption must not be used since they are deprecated (as they have shown to be deadlock-prone). Suspending and resuming a thread in Java can therefore currently only be done by blocking and unblocking a thread or by changing the thread's priority. The `ActiveObject` class uses the latter approach since this approach is less intrusive. The stop method in the Java thread class is deprecated for the same reason as the other methods

already mentioned. Stopping is therefore implemented by exiting from the `Runnable` `run` method.

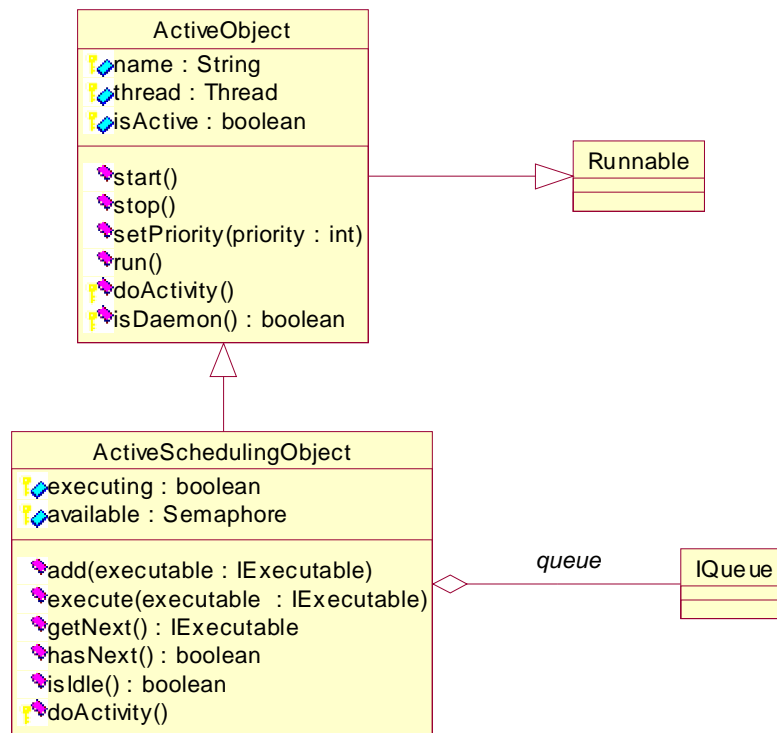


Figure 5-5: General-purpose active object classes

Class `ActiveSchedulingObject` uses a consumer thread that consumes objects from a shared queue and executes them one by one. Agents in Janet act as producers that insert commands they received from other agents into the queue of their scheduler. These schedulers are subclasses of `ActiveSchedulingObject` from which they inherit the functionality for synchronization between threads.

5.2.1.1 Command Schedulers

The kinds of schedulers for executing commands are the supreme scheduler, the system schedulers and the application schedulers. These schedulers are subclasses of the common base class `PriorityCommandScheduler` that inherits from `CommandScheduler`. Class `CommandScheduler` is specialized on scheduling commands. `PriorityCommandScheduler` is a `CommandScheduler` that uses a priority queue. All schedulers are located in the package `org.objectscape.janet.cas.schedulers`. Contrary to the supreme scheduler, the system schedulers and application schedulers interoperate with their arbitrator so that the arbitrator is able to time slice them. These two kinds of schedulers are therefore called arbitrated schedulers.

The arbitrated schedulers notify the arbitrator when they received a command or when they have become idle. A scheduler is considered idle when its queue of commands has become empty and when it is not executing a command. An arbitrated scheduler knows the application

it serves. It is therefore able to calculate the time slice it receives by its arbitrator (depending on the number of agents hosted by the application).

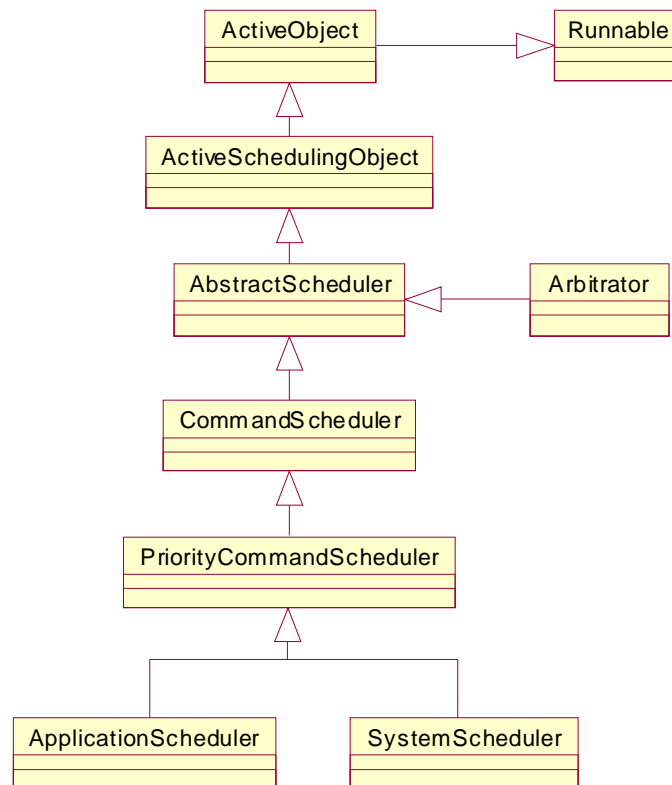


Figure 5-6: Overview of the scheduler hierarchy

It also knows the capability of the agent for logging purposes. The system schedulers run with second highest priority a scheduler must have in the system whereas the application schedulers run with the lowest priority.

5.2.1.2 Arbitrator

The arbitrator makes sure that every arbitrated scheduler receives a fix quantum of the CPU time. When there is only one running arbitrated scheduler present, the arbitrator remains idle as no time slicing is required. The single application scheduler is stored in the attribute `singleScheduler`.

Activating a Sleeping Application Scheduler

When an arbitrated scheduler is created it is added to the list of sleeping schedulers. Whenever a command is added to an arbitrated scheduler the arbitrator is notified by sending the message `commandAddedTo(ApplicationScheduler)`:

- If the arbitrated scheduler was sleeping and there is no running scheduler (neither in the list of running schedulers nor a single running application scheduler), it becomes the single running arbitrated scheduler.

- If the arbitrated scheduler was sleeping and there is a single running arbitrated scheduler present, both arbitrated schedulers are removed from their current locations and added to the list of running schedulers.
- If the arbitrated scheduler was sleeping and there are two or more running arbitrated schedulers, it is added to their list.

No actions need to be taken in case the arbitrated scheduler was not sleeping.

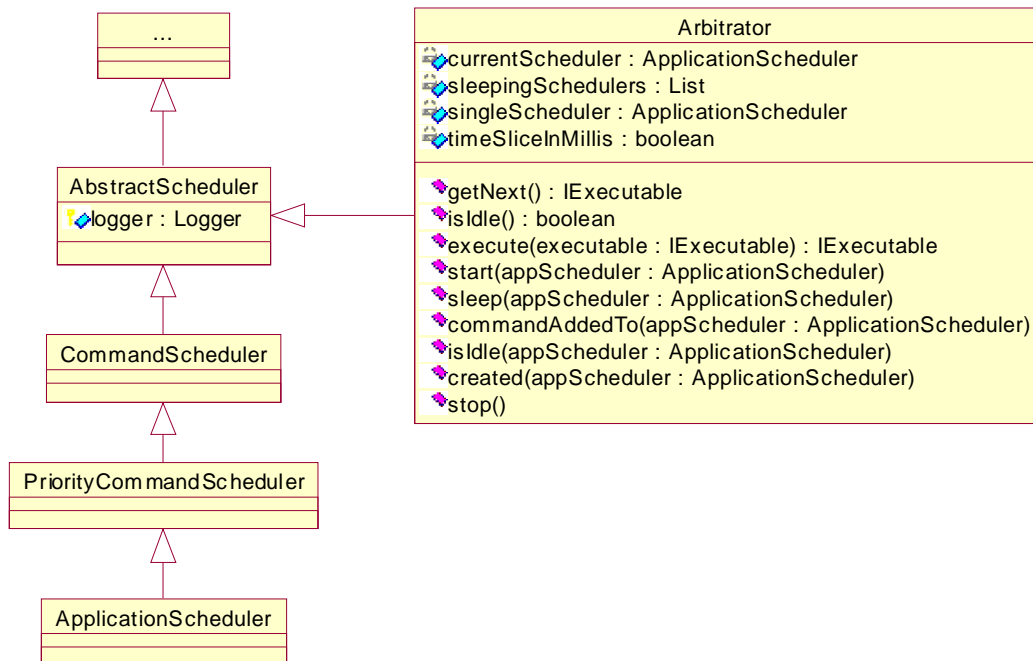


Figure 5-7: Basic arbitrator class hierarchy

Time Slicing Arbitrated Schedulers

The arbitrator starts time slicing arbitrated schedulers when the list of running arbitrated schedulers contains two or more entries. It removes the next running arbitrated scheduler and examines it:

- If the examined arbitrated scheduler is not idle, the arbitrator lowers the priority of the arbitrated scheduler that received the last time slice and raises the priority of the new current arbitrated scheduler. Thereafter, the arbitrator sleeps for the duration of the time slice, which causes the current arbitrated scheduler to be the single running one for the duration of the time slice. The current arbitrated scheduler is then re-inserted at the end of the list of running arbitrated schedulers and the next arbitrated scheduler is picked. If several agents share an arbitrated scheduler, the time slice for the arbitrated scheduler is divided by the number of agents for fairness reasons. If the arbitrated scheduler that receives the current time slice has not been started so far, the arbitrator starts it.

- If the examined arbitrated scheduler has meanwhile become idle, it is added to the list of sleeping schedulers. If this results in a single running arbitrated scheduler being left in the list of running arbitrated schedulers, it is removed from it and becomes the single running arbitrated scheduler and the arbitrator falls asleep till the list of running schedulers contains two or more running arbitrated schedulers.

When an arbitrated scheduler becomes idle it notifies the arbitrator about it sending the message `isIdle(ApplicationScheduler)`. If it was the single running arbitrated scheduler, it is added to the list of sleeping schedulers.

The arbitrator can be asked whether it is idle by sending the message `isIdle` to it. The arbitrator then determines whether any running schedulers and no single running arbitrated scheduler exist.

5.2.2 Commands and Interpreters

Defining an agent's behavior in Janet means to define interpreters and commands. Commands serve as performatives for the agent to invoke applicable interpreters and they serve for carrying input and output parameters of the interpreters. This section describes how to implement commands and interpreters.

5.2.2.1 Implementing Commands and Interpreters

Creating a command that can be sent to an agent requires the construction of a command-interpreter-pair. The user is free to add attributes to a command object. The command object must implement the interface `org.objectscape.janet.cas.schuling.ICommand` and be serializable:

```
public interface ICommand extends Cloneable
{
    public String getQualifiedName();
    public int getPriority();
    public String toLogString();
}
```

Figure 5-8: The ICommand interface

The method `getQualifiedName` must return the fully qualified name of the command the interpreter receives its input parameters from. This name is used by the interpreter associated with the command to inform a scheduler which commands it is intended to interpret.

The method `getPriority` returns the priority of the command according to which it will be inserted into the respective priority queue of the scheduler it was added to. Note that the order of execution with commands of different priority may not be the same as the order of receipt by the agent. If the scheduler queue is already filled with commands, a newly arrived command will not necessarily be executed immediately but remain waiting in the queue like other commands that were received before or after. A command with a high priority can therefore overtake a command with a low priority that arrived before it. It is therefore important not to make assumptions about the order of execution when defining commands with different priorities.

The method `toLogString` must not strictly be defined, but it is recommended to do so. It is used to write log information to the log stream.

The interpreter associated with the command must implement the interface `org.objectscape.janet.cas.schuling.IInterface`:

```
public interface IInterpreter
{
    public void execute(CommandAccessor cmdAccessor);
    public StringVector getCommandNames();
}
```

Figure 5-9: The IInterpreter interface

The method `commandNames` must return a `StringVector` with the command names of the commands that are executed by the interpreter (in most cases the `StringVector` contains a single command name). The schedulers use the command names to look up the associated interpreter to execute a command. The scheduler selects the first hit. It is therefore important to specify the fully qualified command name to rule out ambiguities.

The method `execute` is the core method of every interpreter. It is the point where execution starts and where required parameters are passed on from the command to the interpreter.

5.2.2.2 Executing a Command

Executing an interpreter is the core mechanism to make an agent do something. The definition of the `execute` method of a sample interpreter is therefore explained in further detail.

```
public void execute(CommandAccessor cmdAccessor)
{
    if (cmdAccessor.getCommand() instanceof MyCommand)
    {
        MyCommand cmd = (MyCommand) cmdAccessor.getCommand();
        doSomething(cmd.getInput());
    }
}
```

Figure 5-10: Sample interpreter execute method

The sample interpreter in Figure 5-10 receives its input from the command contained in the method parameter `cmdAccessor` which holds the `CommandAccessor`. After the `doSomething` method has finished execution, execution of the interpreter is finished and the next command in the scheduler's command queue is executed.

5.2.2.3 CommandAccessor

The `CommandAccessor` passed on to the interpreter `execute` method is the entry point for the interpreter to obtain the command it is asked to execute with its input parameters and to the node object with all its inner objects. The `Node` class defines methods that are declared public so that system interpreters that need to access the node's inner objects in order to manage the

node can call them. But the node's inner objects must not be accessed by application interpreters defined by the user as this would allow the user to endanger the functioning of the system, retrieve information she is not privileged to have, or alter other applications. The purpose of the CommandAccessor is to provide guarded access to the node only allowing the user to access objects it has privilege to.

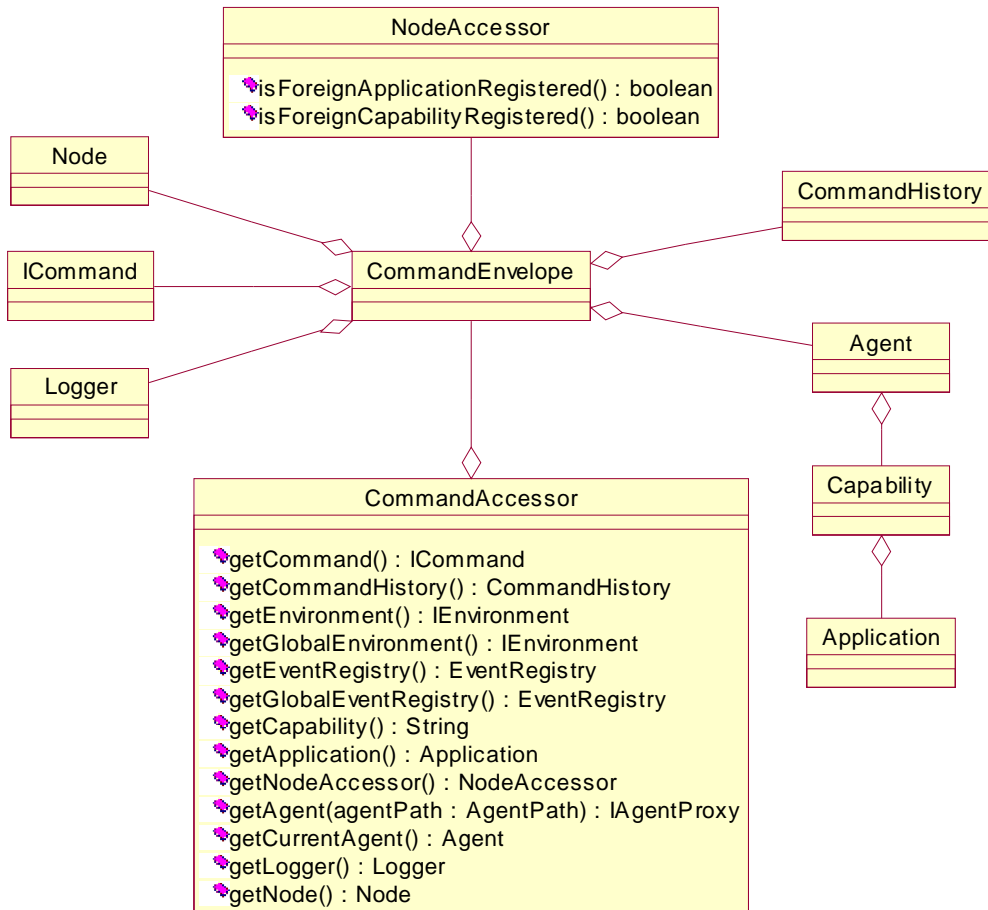


Figure 5-11: Aggregation of the CommandAccessor

If an interpreter executed by an application agent (and not a system agent or the supreme agent) calls methods of the CommandAccessor that give access to the node's inner objects an `InsufficientPrivilegeException` is thrown. On the other hand, the CommandAccessor grants the user full access to all objects referenced by its own application.

5.2.3 Sending Commands to Agents

Sending a command to another agent is the core mechanism to make agents communicate in Janet. This section describes in detail how to narrow another agent and how the agent dispatch mechanism is implemented.

5.2.3.1 Agent Dispatcher

When a command is physically sent across the network it is sent to the agent dispatcher RMI server object that resides on the workstation of the destination agent. With the use of the RMI server object a RMI client is able to acquire a reference to the remote object.

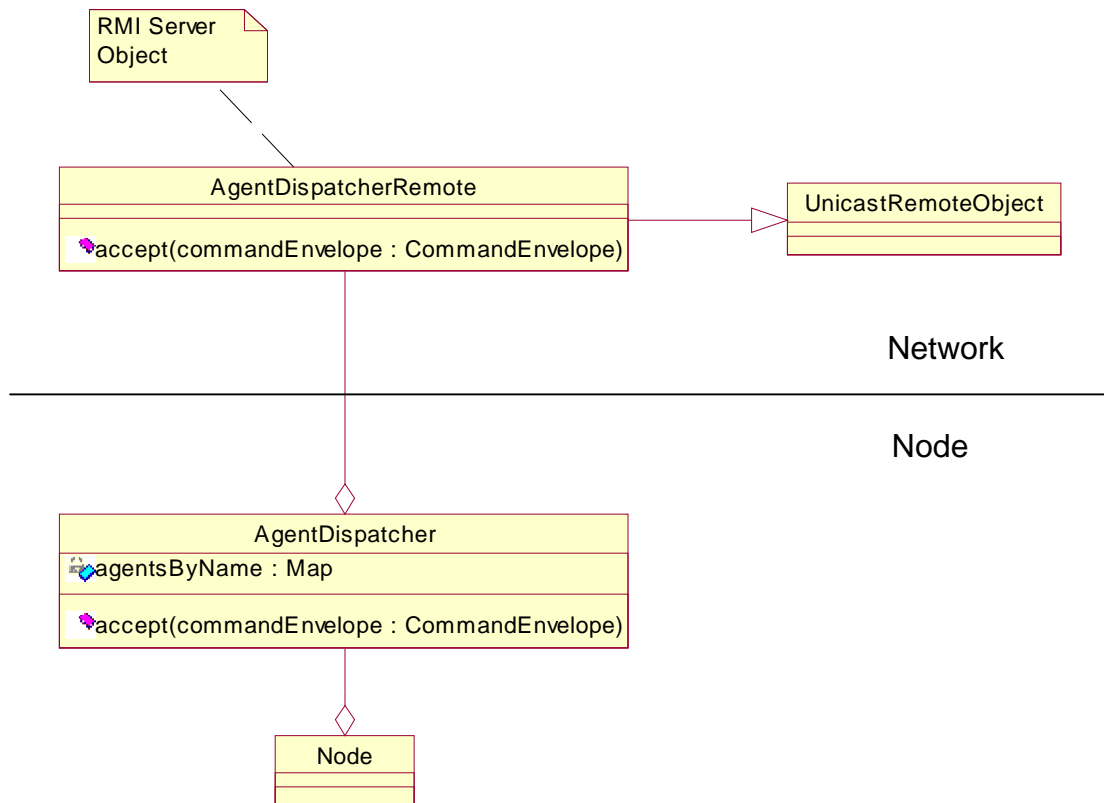


Figure 5-12: Agent dispatcher aggregation

The RMI client passes the command over to the agent dispatcher RMI server object (implemented by class `AgentDispatcherRemote`) that in turn passes it on to the node's agent dispatcher. Class `AgentDispatcherRemote` serves as a network-visible proxy for the agent dispatcher itself. The simplicity of the command pattern as applied in Janet makes it possible for the interface of the agent dispatcher RMI server object to be very simple: it consists of the single method `accept(CommandEnvelope)`. The `CommandEnvelope` is a messenger object that is used to transport a command from agent to agent. It knows the destination agent's location and carries additional information for bookkeeping and other purposes from node to node. The `CommandEnvelope` is explained in further detail later in this chapter.

Need for an Agent Dispatcher

The main purpose of the agent dispatcher is to minimize the number of RMI server objects. The more of them are registered with the RMI registry the more open network connections exist. There should be no limitation to the number of agents that can be started on a node. For this reason, it makes sense to use a single agent dispatcher for every node that receives all com-

mands that are sent to any agents on the same node. The agent dispatcher then passes the command on to the respective agent. Another problem is that binding and unbinding a RMI server object with the RMI registry, which makes it visible within the network or removes it from the network, is a time consuming process that typically takes up to half a minute or more even on a simple network. Quickly creating a new agent when one is needed and registering it would not be possible. In addition, sending an agent from one node to another would require it to be unbound at the source node and be bound at the destination node which would take unacceptably long time. Agents in Janet are not mobile. Nevertheless, the use of an agent dispatcher results in a more flexible design, which leaves options open for sake of extendibility, for example, extensions for mobile agents.

Dispatching Commands

The command object, when sent across the network, is contained by a command envelope. The command envelope contains the fully qualified name of the destination agent. The agent dispatcher contains a map to store agents by their fully qualified name. When the system starts an agent it registers the agent with its fully qualified name with the agent dispatcher. When dispatching a command the agent dispatcher looks up the agent from its store and passes the command on to the agent. Before doing so, the agent dispatcher adds some bookkeeping information to the command envelope's travel history.

5.2.3.2 Sending Commands to Agents: User Interface

Sending a command to an agent with the use of a command envelope and an agent dispatcher requires good understand of the inner workings of Janet.CAS. A user interface has therefore been designed to shield the inner workings from the user when sending commands to agents.

Agent Proxies

Agent proxies are introduced to make sending a command to an agent a simple process for the user. The agent proxy hides the complexity of sending a command to an agent from the user. The user can obtain an agent proxy from the application she created. An agent acts by executing an interpreter in response to a received command. This means that sending a command from an agent to another has to be done within the `execute` method of an interpreter or within methods being called from it.

```
public class MyInterpreter implements IInterpreter
{
    // ...
    public void execute(CommandAccessor cmdAccessor)
    {
        if(cmdAccessor.getCommand() instanceof MyCommand)
        {
            ICommand myCommand = new MyCommand();
            IAgentProxy agent = cmdAccessor.getAgent("nodeName");
            agent.accept(myCommand);
        }
    }
    // ...
}
```

Figure 5-13: Using an agent proxy

Figure 5-13 on page 50 shows some sample code where a command is sent to an agent on the node `nodeName` from within the interpreter `MyInterpreter`. The destination agent has the same agent path as the sending agent, but resides on the node `nodeName`. An agent path is used to uniquely identify an agent within the cluster.

Agent Path

The sample code in Figure 5-14 shows how a command is sent to an agent identified by its agent path. The agent path must contain the destination agent's node name, application name, capability name, and the name of the agent itself.

```
// ...

if(cmdAccessor.getCommand() instanceof MyCommand)
{
    ICommand myCommand = new MyCommand();
    AgentPath agentPath = new AgentPath(
        "nodeName",
        "myApplication",
        "myCapability",
        "myAgentName");
    IAgentProxy agent = cmdAccessor.getAgent(agentPath);
    agent.accept(myCommand);
}

// ...
```

Figure 5-14: Obtaining an agent proxy using an agent path

The agent proxy's `accept` message verifies whether the user has supplied a valid agent path. The node stores in its local cluster image the agent paths to existing agents in the cluster. It knows about them as every node that creates or removes an agent sends a notification about the event to all other nodes in the cluster. When accepting a command the agent proxy looks up in the node's cluster image whether the user-supplied agent path exists. The exception `NoSuchAgentException` is thrown if it this is not the case.

If the agent path is valid the agent proxy checks whether the application name of the sending agent and the recipient agent are identical. If this condition is not met a protection violation has been detected, as agents must not communicate across application boundaries using commands (they may do so with the use of undirected events on a publish-subscribe basis), and the exception `IllegalAgentAccessException` is thrown. The system agents and the supreme agent are allowed to send commands across application boundaries which is required to implement core functionality efficiently. But application-defined agents are forced to use events. Figure 5-15 on page 52 shows how a command is sent to the agent itself that executes the current command using the message `getCurrentAgent`. The required try-catch-blocks to handle these exceptions have been omitted from the sample code for sake of brevity.

The `CommandAccessor` provides several convenience methods for obtaining an agent proxy that are delegate methods of class `AbstractApplication`. These methods fill in the application name, capability name, and agent name depending on which method was called into the agent path.

```
// ...

if(cmdAccessor.getCommand() instanceof MyCommand)
{
    ICommand myCommand = new MyCommand();
    IAgentProxy agent = cmdAccessor.getCurrentAgent();
    agent.accept(myCommand);
}

// ...
```

Figure 5-15: Sending a command to the agent itself

Multi Agent Proxies

If the same command has to be sent to several agents at once the `MultiAgentProxy` iterates over all agents to send the command to them and frees the user from doing so herself. In Figure 5-16 the command is sent to all agents in the cluster with the same agent path as the current agent.

```
// ...

if(cmdAccessor.getCommand() instanceof MyCommand)
{
    ICommand myCommand = new MyCommand();
    IAgentProxy agent = cmdAccessor.getAllForeignAgents();
    agent.accept(myCommand);
}

// ...
```

Figure 5-16: Using a MultiAgentProxy

The `MultiAgentProxy` implements the `IAgentProxy` interface like the `AgentProxy`. It clones a command before sending it to the destination agent. This makes sure that in simulated mode, where all nodes run in the same Java virtual machine, and when commands are sent to agents with the same node (and therefore live on the same Java virtual machine using the same heap space) each recipient agent works on a copy of the original just as in distributed mode or when sending a command to an agent on a different node (where the command is necessarily copied due to the marshalling process). Without cloning the command, each recipient agent living in the same heap space would receive a pointer to the same command that is also being processed by the other recipient agents resulting in command state data to be changed without synchronization.

Installing a Callback Handler

If an agent expects a value to be returned from the destination agent it can pass on a callback handler to the agent proxy's `accept` method as shown in Figure 5-17 on the next page.

```

1  // ...
2
3  if(cmdAccessor.getCommand() instanceof MyCommand)
4  {
5      ICommand myCommand = new MyCommand();
6      IAgentProxy agent = cmdAccessor.getAgent("nodeName");
7      ICallbackHandler handler = new MyCallbackHandler();
8      agent.accept(myCommand, handler);
9  }
10
11 // ...

```

Figure 5-17: Installing a callback handler

After the callback handler has been passed on the thread continues execution. The callback handler will be invoked when the reply of the destination agent has arrived. Figure 5-18 shows how to define a callback handler by implementing the interface `ICallbackHandler` which requires the method `handle` to be defined.

```

public class MyCallbackHandler implements ICallbackHandler
{
    // ...

    public void handle(Object callbackParameter)
    {
        // process result
    }

    // ...
}

```

Figure 5-18: Handling the callback

5.2.3.3 Sending Commands to Agents: Implementation

The previous section described the steps the user has to take to send a command from an agent to another one where agent proxies were used to hide the complexity of the underlying mechanism. The following explains what happens within the agent proxy to send the command to the destination agent.

5.2.3.3.1 General Mechanism

This section describes the mechanism to send a command across the network from one agent to another. The description explains the steps that need to be carried out in Janet.CAS to send a valid command envelope to the destination agent's node agent dispatcher. It does not explain the concept of RMI or inner workings of it.

Command Envelope

When a command is sent from an agent to another agent, and later on is eventually sent back, the system keeps a history of the visited agents and travel times. Bookkeeping travel times is needed for performance measurements. After a command has arrived at an agent it needs to have a reference to the recipient agent as an entry point to access the agent's application, object space and the node itself in case a system command is executed. Adding the required attributes to hold the references to these objects to an abstract superclass of the command class would result in the user not being free to choose the superclass of its command. For this reason, the interface `ICommand` is defined that must be implemented by all commands. The needed attributes are defined in another class that also carries the command from agent to agent which is class `CommandEnvelope`.

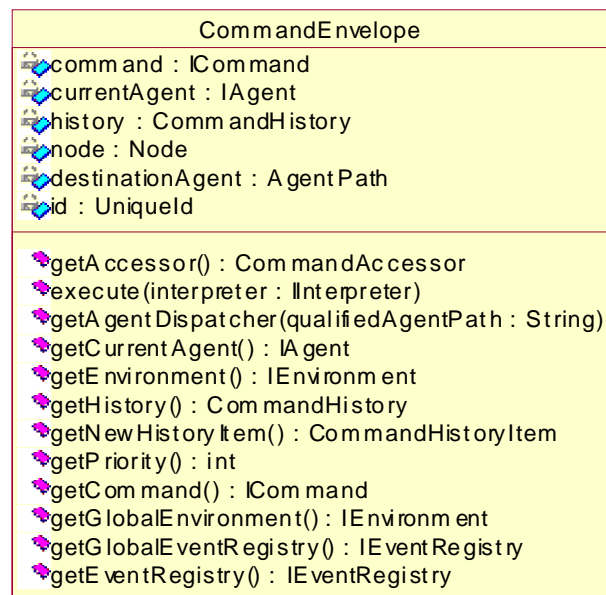


Figure 5-19: `CommandEnvelope` basic class description

Using the Agent Dispatcher to send a Command

After the agent proxy has created the command envelope it adds bookkeeping information to it (e.g., the send time), clears variables storing objects that must not be serialized (like the current agent or the node object). It then sends the `accept` message to the destination agent's agent dispatcher with the command envelope as the parameter which results in the command envelope to be physically sent across the network to the workstation hosting the destination node. The agent dispatcher is a reference to the RMI server object agent dispatcher of the destination agent's node. The agent proxy obtained a reference to the agent dispatcher when it was instantiated from the node's cluster image that holds the RMI server object agent dispatchers of all other nodes, which were added to it during the node startup procedure. To get a reference to the RMI server object agent dispatcher the agent proxy needs to provide the destination agent's node name.

5.2.3.3.2 Extensions for Callbacks

To invoke callback handlers, class `CommandEnvelope` is extended by a specialized command envelope. The abstract class `CommandResponseEnvelope` is introduced as a subclass of `CommandEnvelope` that takes care of sending the command envelope back with some return value to the sending agent after the interpreter, invoked by the destination agent to execute the command, has finished execution.

When the agent proxy accepts a command for which a callback handler has been supplied, it instantiates class `CommandCallbackEnvelope`, which is a subclass of `CommandResponseEnvelope`, instead of class `CommandEnvelope` as in the general case when no callback handler is supplied. It places the callback handler in the agent's application object space and stores the callback handler's application object space lookup key in the attribute `responseId`. The class hierarchy for class `CommandCallbackEnvelope` with most important attributes and methods is displayed in Figure 5-20.

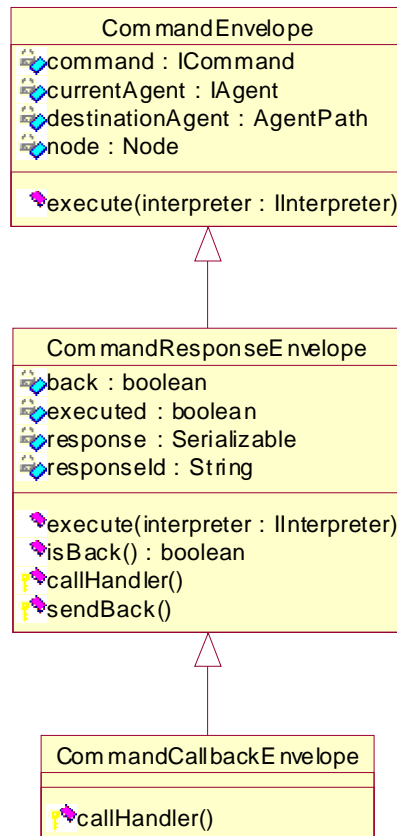


Figure 5-20: `CommandCallbackEnvelope` class hierarchy

The main extension in class `CommandResponseEnvelope` is to redefine the inherited `execute` method.

```

1  public abstract class CommandResponseEnvelope
2  extends CommandEnvelope
3  {
4      // ...
5      public void execute(IInterpreter interpreter)
6      {
7          if (!executed)
8          {
9              super.execute(interpreter);
10             executed = true;
11             sendBack();
12         }
13         else
14         {
15             callHandler();
16         }
17     }
18     // ...
19 }
```

Figure 5-21: Redefined `execute` method for class `CommandResponseEnvelope`

Figure 5-21 shows the `CommandResponseEnvelope`'s `execute` method. If the `CommandResponseEnvelope`'s command has not been executed so far, the inherited `execute` method from class `CommandEnvelope` is called (line 9). The `CommandResponseEnvelope` then marks the command as having been executed (line 10) and sends the command envelope back by calling `sendBack` (line 11). When the command envelope has arrived back at the sending agent it is executed and the method `callHandler` is called (line 15). The method `callHandler` is declared abstract in class `CommandResponseEnvelope`.

```

public class CommandCallbackEnvelope
extends CommandResponseEnvelope implements Serializable
{
    // ...
    protected void callHandler()
    {
        IObjectSpace env = getCurrentAgent().getObjectSpace();
        ICallbackHandler handler =
            (ICallbackHandler) env.remove(responseId);
        handler.handle(response);
    }
    // ...
}
```

Figure 5-22: Invoking the callback handler

It is defined in class `CommandCallbackEnvelope` as displayed in Figure 5-22. The callback handler stored in the sending agent's object space before the command was sent to the destina-

tion agent is taken from the object space and the callback handler is called passing on the response.

5.2.3.4 Convenience Facility: Synchronous Command Sends

All commands sends in Janet are asynchronous following the agent-oriented paradigm. In some situations the application logic requires that an agent needs to wait till some other agent has finished executing a command it had sent to the other agent before it can continue. In Janet.CAS this is for example the case during the startup-process where the system agents has to wait till it has received an acknowledgement from all other nodes in the cluster. Using asynchronous commands makes sure that the application cannot get stuck because of commands creating a deadlock situation. But in some situations it is convenient to use a synchronous command, as this would simplify application logic considerably. Janet.CAS provides a facility for the user that blocks the current thread till the response of the other agent has arrived and then unblocks it. It is left to the responsibility of the user to use this facility only in cases where it is justified not to compromise the agent-oriented paradigm by not using asynchronous commands.

5.2.3.4.1 User Interface

This section describes the user interface for using synchronous commands to receive an acknowledgement by the destination and to receive a reply synchronously. The next section explains the details of the inner workings of the facility to send synchronous commands.

Waiting for Acknowledgement

The sample code in Figure 5-23 shows how to send a command to an agent that requires an acknowledgement by the recipient agent that is sent back after the recipient agent has executed the command.

```

// ...
1  if(cmdAccessor.getCommand() instanceof MyCommand)
2  {
3      ICommand myCommand = new MyCommand();
4      IAgentProxy agent = cmdAccessor.getAgent("nodeName");
5      Acknowledgement ack = new Acknowledgement();
6      agent.accept(myCommand, ack);
7
8      try
9      {
10         long timeOutPeriodInMillis = 2000;
11         ack.acquire(timeOutPeriodInMillis);
12     }
13     catch (TimeoutException e)
14     {
15         // handle timeout
16         return;
17     }
18 }
// ...

```

Figure 5-23: Waiting for an acknowledgement

From the sample code in Figure 5-23 on the previous page it can be seen that the user interface is on principle the same than when sending a command asynchronously. The only difference to the agent proxy's `accept` method is that an acknowledgement object has to be passed on (line 8). After the command has been sent away the current thread has to be blocked till the acknowledgement has arrived (line 13). If no timeout period is specified the sending thread is possibly blocked forever. A `TimeoutException` is thrown if the acknowledgement has not arrived before the timeout period has expired. When a command is sent to a `MultiAgentProxy` and an acknowledgement is requested the `MultiAgentProxy` waits till the acknowledgements from all destination agents have arrived.

Waiting for a Reply

The user interface for waiting for a synchronous reply from the destination agent is on principle the same than when waiting for an acknowledgement. As shown in Figure 5-24 a future result object is passed on instead of an acknowledgement object to the agent proxy's `accept` method (line 8). The result sent back from the destination agent to the sending agent has to be asked from the result object (line 13 + 14) that blocks the current thread until the reply has arrived.

```

1  // ...
2
3  if(cmdAccessor.getCommand() instanceof MyCommand)
4  {
5      ICommand myCommand = new MyCommand();
6      IAgentProxy agent = cmdAccessor.getAgent("nodeName");
7      FutureResult futureResult = new FutureResult();
8      agent.accept(myCommand, futureResult);
9
10     try
11     {
12         long timeOutPeriodInMillis = 2000;
13         Object result =
14             futureResult.getResult(timeOutPeriodInMillis);
15     }
16     catch (TimeoutException e)
17     {
18         // handle timeout
19         return;
20     }
21 }
22
23 // ...

```

Figure 5-24: Waiting for a reply

If no timeout period is specified the sending thread is blocked possibly forever. A `TimeoutException` is thrown if the reply has not arrived before the timeout period has expired.

5.2.3.4.2 *Implementation*

The idea of a `CommandResponseEnvelope` to invoke a callback handler, installed by an agent before sending a command to another agent, was already introduced in section “5.2.3.3.2

Extensions for Callbacks”. The `CommandResponseEnvelope` is sent back from the destination agent after executing the command. Upon receipt the `CommandResponseEnvelope`’s `execute` method retrieves the callback handler from the agent’s object space and passes on the callback value carried with it from the responding agent when invoking the callback handler.

Deadlock Problem due to Synchronicity

The same approach can be used to carry a reply value back synchronously from an agent to another. In addition, the thread running the sending agent has to be blocked till the reply has arrived. Waiting for an acknowledgment can be implemented using the same approach without having to pass on a reply value. This approach works well for implementing asynchronous callbacks. In case of synchronous replies or acknowledgments it has to be extended to prevent the deadlock from happening that would occur as the thread of the sending agent is still blocked at the time the response command arrives. Before the sending agent can execute the response command, which would unblock the agent’s thread, it needs to finish execution of the current command. Because the agent schedulers execute one command till completion till they execute the next one and the current command is blocked waiting for the response command, the response command will never execute and the agent runs into a deadlock.

Solution

There is no solution than can work by adding the response command to the waiting agent’s scheduler queue. One solution would be to let every agent have a shadow agent with its own scheduler that handles response commands. While this approach seems conceptually clean the result would be a duplication of the number of agents in a node which would be inefficient. This suggests that the solution would be to have a single agent to handle response commands. For this reason, an additional system agent is created that serves no other purpose than executing response commands. Since such a response agent is conceptually not important it has not been mentioned so far. It would also be possible to let every application have its response agent. For the time being, the simpler approach of having a single response agent for an entire node is considered sufficient and left as such until proven not to be sufficient. The response agent runs with system scheduler priority to minimize waiting times till a response command can be executed.

Blocking the calling thread is done using class `Latch` from Doug Lea’s `util.concurrent` class library that contains several common classes for thread synchronization and inter-process communication. All synchronization in Janet is done using synchronization classes from this library, which is described in [LEA99].

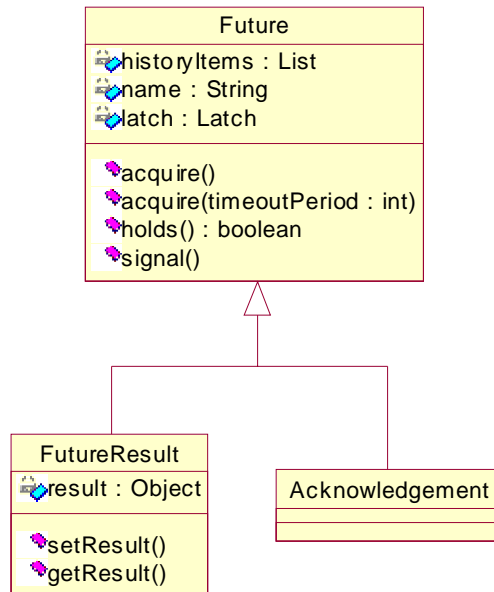


Figure 5-25: Future objects to block calling threads

5.2.4 Object Spaces

The idea of object spaces is to supply agents with a store where they can insert objects that need to be present permanently and must not go out of scope after an interpreter has finished execution. Object spaces can also be used as a convenient way to share information between agents.

5.2.4.1 Overview

There are object spaces on three levels: application-level, node-level, and cluster-level. All object spaces allow agents to exchange objects or store objects that need to remain alive after a command has finished execution. All object spaces implement the same interface and are synchronized.

- **Application Object Space:** Agents that belong to the same application and node have access to the application object space, which can be obtained from the agent's application object.
- **Node Object Space:** Agents that belong to the same node, but not necessarily to the same application, can access the node object space, which can be obtained from the agent's node object or its accessor object.
- **Cluster Object Space:** All agents in the cluster can access the cluster object space independently from the node they reside on or the application they live in. This object space is a RMI server object, which any agent can access directly using the object space's interface without sending commands. The number of cluster object spaces in the cluster is arbitrary. For cluster object spaces to be visible to agents, they must be defined in the node descriptor. The node descriptor defines a node's statically defined applications with their capabilities.

ties and agents. It is read at node startup time. Node descriptors are presented in detail for the driving applications presented in this chapter.

```
<clusterObjectSpaces>
  <clusterObjectSpace name="sharedSpace1"
    hostname="myWorkstation1" port="1099" />
  <clusterObjectSpace name="sharedSpace2"
    hostname="myWorkstation2" port="1099" />
</clusterObjectSpaces>
```

Figure 5-26: Defining cluster object spaces in the node descriptor file

Figure 5-26 shows an excerpt from a node's descriptor file that defines a cluster object space named "sharedSpace1" and "sharedSpace2" that reside on the workstations "myWorkstation1" and "myWorkstation2". The RMI registry, to which they are bound to, is connected to port 1099. When a node starts up, it checks whether it resides on any workstation listed in the `clusterObjectSpaces` section of the node descriptor. If so, it creates one and binds it with the local RMI registry, or looks it up in case it was already bound by some other node on the same workstation that had started up before.

5.2.4.2 Message-Passing vs. Shared Memory

In distributed computing there are two computing models that are based on message-passing and shared memory. Discussing which model is the better one is an arduous task that yields no benefits for this work. In most cases, the decision, which model is the better choice, depends mostly on the given situation (requirements, hardware setup, etc.). Both models have their advantages and disadvantages:

- Message-Passing
 - + Controlled execution: no race conditions, protection
 - + Explicit control makes performance issues clear
 - Difficult to program, especially for irregular, dynamic programs
- Shared Memory
 - + Easier to program, graceful migration path, hence attractive
 - + Works well on moderate-scale tightly-coupled systems
 - Race conditions, synchronization hazards, fault intolerance

Janet is clearly based on the message-passing model where commands take over the role of messages. Other systems, like JavaSpaces, are clearly based on the shared memory model. Janet's cluster object space can be used to a limited extent as a shared memory space in the sense of the shared memory model. However, this would obstruct the intention in which the system was designed and is not recommended. In any event, the cluster object space is by no means that comfortable and rich in functionality as shared memory spaces provided by JavaSpaces or other shared memory systems. Janet's cluster object space intentionally provides no way to register callbacks for events that happen inside the object space. Object spaces in Janet are intended to be used as spaces to store data that needs to survive after a certain action. They

can also be used to exchange information between agents without having to exchange commands to obtain this knowledge when cooperating agents know that the information is present in the object space. For example, if an agent knows that some other agent periodically provides some information to the public by inserting it into the object space, it can obtain this information directly from the shared object space without having to pay for the overhead inherent to message-passing.

5.2.4.3 Interface

The object space interface `org.objectscape.janet.cas.spaces.IObjectSpace` in Janet resembles the interface for an associative container: objects can be stored and retrieved from the object space using a lookup key associated with the object. Being able to store and retrieve objects is not flexible enough in case an agent needs to attach some specific information to an existing object inserted by some other agent. The object space interface therefore allows the user to attach objects to objects already existing in the object space or detach objects from them using `attacher` and `detacher` objects. These objects must implement the interfaces `IAttacher` or `IDetacher` in `org.objectscape.janet.cas.spaces` displayed in simplified form in Figure 5-27 and Figure 5-28.

```
public interface IAttacher
{
    public void attach(Object attachment, Object object);
}
```

Figure 5-27: Simplified IAttacher interface

```
public interface IDetacher
{
    public Object detach(Object object);
}
```

Figure 5-28: Simplified IDetacher interface

```
// ...
ISerializableObjectSpace env1 =
    node1.getClusterObjectSpace("sharedSpace");

ISerializableObjectSpace env2 =
    node2.getClusterObjectSpace ("sharedSpace");

String key = TimeUtility.now();    // return the current time as a string
env1.put(key, new Vector());
String attachment = "foo";
env2.attach(key, attachment, new ListAttacher());
String detachment = (String) env1.detach(key, new ListDetacher());
// ...
```

Figure 5-29: Using IAttacher and IDetacher objects

The code sample in Figure 5-29 shows how to use attacher and detacher objects with a cluster object space to add an object to a `Vector` and remove the first object from the `Vector` using a `ListAttacher` and a `ListDetacher` object that implement `IAttacher` and `IDetacher`. The variables `attachment` and `detachment` end up holding the same value.

Attacher and detacher objects can also be installed in the object space for every object that exists in the object space so that they are always used when `attach(String key, Object object)` or `Object detach(Object object)` in `IObjectSpace` are called. This gives the inserting agents control over what other agents can do with the objects they inserted. When using the cluster object space the interface defines that only instances of `Serializable` may be stored in the object space in contrast to the application object space and the node object space where instances of `Object` may be used as well.

5.2.5 Event Registries

Analogously to object spaces, there are event registries on three levels: application-level, node-level, and cluster-level. The application-level event registry allows agents of the same application to be notified about events. Using the event registry on node-level agents of different applications residing on the same node may notify each other about event occurrences. Finally, the cluster-level event registry allows agents of different applications on different nodes to communicate through events.

Fairness: Who pays for the Execution of an Event Handler?

When the handler of a signaled event is executed the event registry has to make sure that the agent that benefits from the execution of the handler pays for the CPU cost of executing it. The interface for registering an event handler therefore requires that an agent is specified that executes the event handler. After an event was signaled the event registry wraps the event handler with the event parameters received by the signal into a command and sends it to the specified agent.

5.2.5.1 Executing Event Handlers

The event registry uses the system command `ExecuteEventHandlerCommand` as a command wrapper for the event handler. Instead of using user-level agent proxies, the event registry creates the command envelope itself and plugs the `ExecuteEventHandlerInterpreter` in. When the `ExecuteEventHandlerCommand` is executed the interpreter associated with the command will not be looked up from the agent's capability but taken from the command envelope. This is a departure from the concept where an agent chooses the interpreter for executing a command himself. The reason this is done in this case, is to free the user from having to define the `ExecuteEventHandlerInterpreter` in every capability of every application she has defined. To predefine which interpreter is to be chosen for the execution of a command is only permitted on system-level. On application-level the user has no way of doing this as applications do not allow access to the node object, from which the respective agent dispatcher can be obtained to send a command envelope to an agent. The user is forced to use agent proxies instead.

As explained, executing an event handler results in a command being invoked of which the associated interpreter executes the handler in the end. Agents execute commands sequentially.

The idea of events is that events interrupt the program and notifies it about the event. The `ExecuteEventHandlerCommand` is executed with system priority, which means that it overtakes all application-level commands in the agents command queue (the user has no way to assign the system priority to an application-level command). Nevertheless, the `ExecuteEventHandlerCommand` will only execute after the currently running command has finished execution, or after other commands with system priority that might have been inserted into the queue before. If any waiting time till the `ExecuteEventHandlerCommand` is executed cannot be accepted for application-dependent reasons, the user may define a special agent for handling events or handling a specific event to which no other commands are sent.

5.2.5.2 Interface

The event registries of all three levels share the same interface. Obtaining a reference to an event registry is specific for each level. To register for an event the application that defines the executed interpreter has to be specified as well as an agent of this application, beside some other parameters. The agent will execute the `ExecuteEventHandlerCommand` that invokes the event handler. Figure 5-30 shows the registration, signaling, and deregistration of an event using an application-level event registry.

```
public class MyInterpreter implements IInterpreter
{
    // ...
    public void execute(CommandAccessor cmdAccessor)
    {
1       ILocalEventRegistry eventRegistry =
2           cmdAccessor.getApplication().getEventRegistry();
3       ApplicationAccessor app =
4           cmdAccessor.getApplication().getAccessor();
5       AgentPath agentPath =
6           app.getLocalAgentPath("MyCapName", "MyAgentName");
7       EventHandlerEnvelope item =
8           new EventHandlerEnvelope(new MyHandler(), agentPath);
9       HandlerList list = new HandlerList();
10      list.add(item);
11
12      eventRegistry.register(MyParams.EVENT_NAME, list, app);
13      assert(eventRegistry.getHandlerCount(MyParams.EVENT_NAME) == 1);
14
15      IEventParams params = new MyParams();
16      Event event = new Event(MyParams.EVENT_NAME, params);
17      eventRegistry.occurred(event);
18
19      eventRegistry.deregister(MyParams.EVENT_NAME, list);
20      assert(eventRegistry.getHandlerCount(MyParams.EVENT_NAME) == 0);
    }
    // ...
}
```

Figure 5-30: Event registry partial interface

In line 1 + 2 the application event registry of the application that defines the executed interpreter is retrieved. Line 5 + 6 define an agent path of an agent with capability name "MyCapName" and agent name "MyAgentName". Since the agent path is obtained from the application, node name and application name are set by the `getLocalAgentPath` method as appropri-

ate for the application. The event handler defined in line 7 + 8 needs to know which agent will execute the handler. The agent path is therefore passed on to the `EventHandlerEnvelope` constructor. The event handlers are installed in line 12. When the registration method is invoked, it re-inserts node name and application name into the agent paths of the handlers to make sure that the user cannot chisel by specifying an agent path of an agent on a different node or a different application. The event is signaled in line 17 and deregistered in line 19.

```
public class MyInterpreter implements IInterpreter
{
    // ...
    public void execute(CommandAccessor cmdAccessor)
    {
        ILocalEventRegistry eventRegistry =
            cmdAccessor.getEventRegistry();
        // ...
    }
    // ...
}
```

Figure 5-31: Obtaining the node event registry

As shown in Figure 5-30 the application event registry is obtained from the application. The node registry is obtained from the command accessor, which provides a convenience method for that purpose, as the user cannot access the node object, as shown in Figure 5-31.

5.2.5.3 Cluster Event Registry

The cluster event registries of a cluster must be reachable by every node in the cluster. The cluster event registry could therefore be designed as a RMI server object in the same way as the cluster object space. Another possibility would be to use a special node to hold the cluster event registry. This more elaborated approach would provide greater flexibility and extensibility, for example, if more advanced features such as persistence or transactions had to be added. Since the implementation of such advanced features is beyond the scope of this work, the less effortful approach has been chosen, which is sufficient. Abandoning this approach and implementing the cluster event registry as a node later on would not result in a loss of much work.

Local Cluster Event Registries and the Cluster Event Registry

Every node has a local cluster event registry, where applications of the node register and de-register events, which are propagated by the local cluster event registry to the cluster event registry itself. When an event is signaled, the event handler registered in the local cluster event registry is invoked first. Thereafter, the signal is propagated to the cluster event registry. The cluster event registry in turn passes the signal on to all other local cluster event registries on all other nodes that have described for the event.

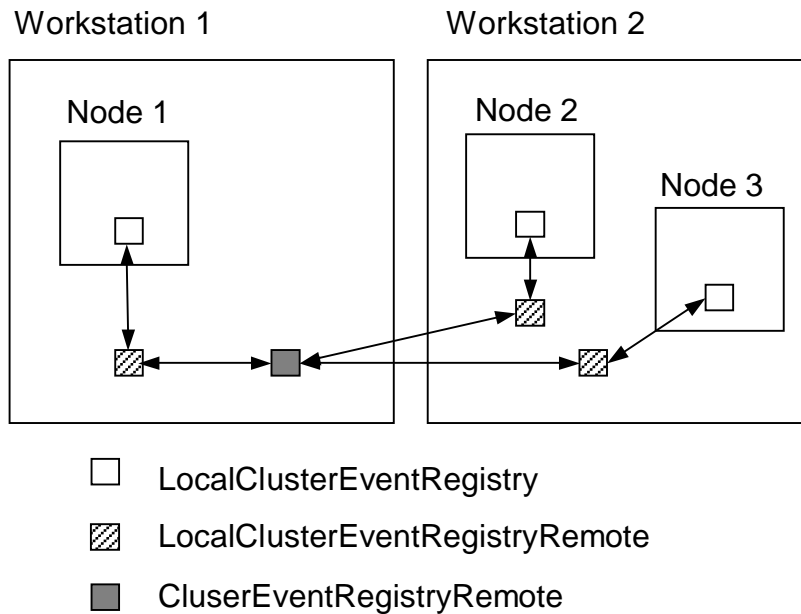


Figure 5-32: Cluster event registry with local cluster event registries

The sequence of actions when a cluster-wide registered event is signaled is display in Figure 5-32. The LocalClusterEventRegistryRemote RMI server object is needed for the ClusterEventRegistryRemote to be able to pass an event on to a local cluster registry.

Node Descriptor Definition

For cluster event registries to be visible to agents, they must be defined in the node descriptor. The definition of cluster event registries is analogous to the definition of cluster object spaces. A sample cluster event registry definition from a node descriptor file is displayed in Figure 5-33.

```
<clusterEventRegistries>
  <clusterEventRegistry name="sharedEventRegistry1"
    hostname="myWorkstation1" port="1099" />
  <clusterEventRegistry name="sharedEventRegistry2"
    hostname="myWorkstation2" port="1099" />
</clusterEventRegistries>
```

Figure 5-33: Defining cluster event registries in the node descriptor file

When the last node is shut down on a workstation where the cluster event registry resides the cluster event registry is also shut down. RMI, for security reasons, only permits an RMI server object to be unbound on the workstation where it is registered. The cluster event registry can therefore not be unbound from a remote workstation. The user needs to take caution when shutting down nodes. The node on the workstation where the cluster event registry resides must be shut down last. It is recommended to add the definition of a cluster event registry to the central's node descriptor as well.

When the central is started up first, the cluster event registry is bound with the local RMI registry on the same workstation as the central. Since the central has to be shut down last, no caution needs to be taken in which sequence to shut down nodes.

5.3 Driving Application: Fibonacci Numbers

It is useful when developing a system that serves as an application platform to develop a sample application, which serves as an example to show the user how to use the application platform. Another reason to develop a sample application is that by stepwise developing the sample application the application platform itself grows through the development process as new requirements become evident that were not clear to begin with. The sample application can also be used to validate architecture, conceptual design, and implementation of the application platform.

Section 8.3 “Running the CAS.Fibonacci Sample” describes how to start up the CAS.Fibonacci sample application and how to use it.

5.3.1 Fibonacci Numbers

Fibonacci numbers are calculated recursively. Calculation of large Fibonacci numbers therefore can take relatively long time, which is the reason that the calculation of these numbers is sometimes used to get a rough performance estimate of a computer. As a simple driving example for Janet.CAS an application is developed for the distributed calculation of Fibonacci numbers. From one Fibonacci agent commands are sent to all other Fibonacci agents on all other nodes to calculate a Fibonacci number. The sending agent calculates a Fibonacci number as well and waits till all results have returned. Thereafter, all results are displayed. The application is solely based on Janet.CAS and does not make use so far of Janet.ADE. This means that the recipient agents to calculate Fibonacci numbers are selected by the sending agent explicitly.

An agent-oriented system like Janet is not well suited for distributed number crunching. Compared to less flexible distributed applications architectures where peer-to-peer connections may be hard-wired, agents induce several additional indirections (such as dynamic agent lookup, use of schedulers, command queues, commands and interpreters), which only cause unnecessary overhead when calculation speed is the only important criteria. An agent-oriented system like Janet is well suited to solve complex problems where agents with their autonomy and ability to solve problems themselves, or part of them in conjunction with other agents, can be used to reduce complexity. Nevertheless, the CAS.Fibonacci sample remains useful since it only serves to explain how to use the most important features of the Janet.CAS layer. In addition, it is very useful to make performance measurements. It becomes possible to compare how much time it takes to calculate the same Fibonacci number on a single workstation n times and how much time it takes when n workstation in the cluster with one node each calculate the same Fibonacci number once. Later on, when the example is extended to serve as a driving application for Janet.ADE, a running Fibonacci command can be preempted and evicted from a node to level out workload imbalances. Because large Fibonacci numbers take several seconds to be calculated there is sufficient time available to provoke a load change that makes command preemption necessary. Since Fibonacci numbers are calculated recursively it is simple to interrupt the command, save its context, and resume it after arrival at the new node. When the system requires the Fibonacci command to suspend execution it stores the current state after the last

iteration, tells the system that it is ready for migration, and after arrival at the new node resumes execution after the last iteration.

This sample application makes use of most features of Janet.CAS. When the Fibonacci commands are sent to all nodes in a round-trip fashion and finally return with the results most features of Janet.CAS have been made use of and are therefore validated and tested. Event notification is needed as well in the end when the results finally arrive and the calling thread has to be unblocked to resume execution. The CAS.Fibonacci sample does not make use of the cluster object space nor the cluster event registry. These features have been validated and tested separately.

5.3.2 Defining the Fibonacci Application

A Janet cluster consists of one central node, which is simply called “the central”, and one or more nodes. The different behavior of the central and the nodes is defined by specifying a system application appropriate for the role the node is meant to fulfill.

System Application

Unlike user-level applications, the system application has a fix structure: it must have a capability named `CORE` and it always has a single supreme agent. The `CORE` capability required to be present in the system application makes sure that the user can only change the behavior of a node, but she cannot change a node’s internal structure and its underlying design such as the number of supreme agents, which always must be one. The node’s internal structure represents fundamental design decisions the user must not change. Therefore, the definition of the system application in a node’s descriptor file is predefined to some extent. The user cannot alter the predefined part of structure. Doing so would result in a node descriptor parse exception to be thrown at node startup time. On the contrary, for a user-level application the user is free to define an arbitrary number of capabilities with an arbitrary number of agents that may also be defined or undefined programmatically at runtime. The role of a node (central or node) is defined by the interpreters listed in the core capability of the node’s system application. This means that there is no difference in the static class structure between a central and a node. Defining different behavior is confined to the interpreters. Interpreters, which are grouped in a capability, collectively define a capability’s functionality. Any permanent node objects must be installed in the system application’s object space. As a central needs different permanent node objects compared to a node (it needs to store different kind of cluster information), the interpreters of the central’s core capability are responsible for installing them in the system application’s object space of their node. In the same way, the interpreters of a node’s core capability carry the same responsibility. The interpreters of a capability must know the type of the permanent objects they use. They are responsible to cast their permanent objects back to their actual type when they are retrieved from an object space as instances of class `Object`.

Figure 5-34 on the next page shows the central’s main permanent node object, the `CentralNodeAnchor`, which was installed by one of the central’s core interpreters into the system application’s object space. A node’s main permanent object, the `NodeAnchor`, was installed in a node’s object space analogously.

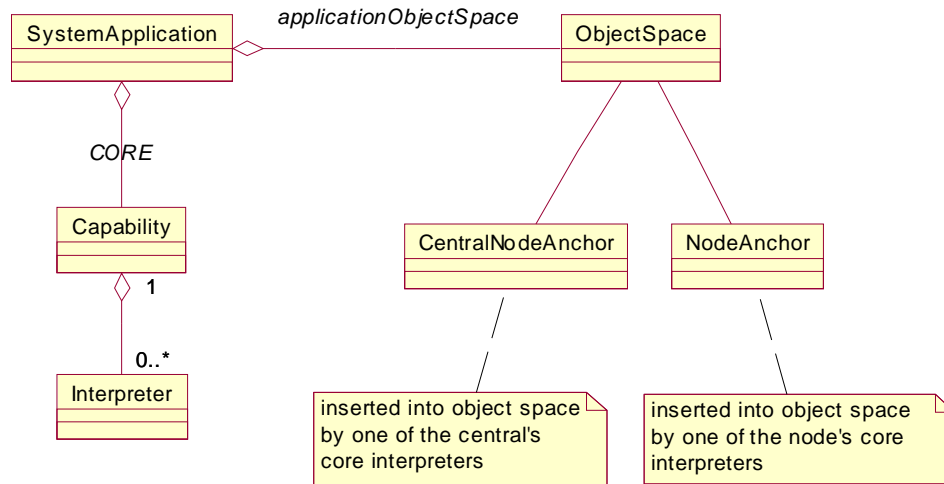


Figure 5-34: A node's core interpreters to install permanent node objects

Defining the Core Capability

To define a node's system application with its core capability the node descriptor has to list the required interpreters that define a node's role. Figure 5-35 displays a basic node descriptor as a skeleton with an empty list of core interpreters.

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- node definition for central -->
<node version="0.1" showGUI="true" exitVMOnNodeShutdown="true">
  <registry centralHostname="myWorkstation" centralPort="1099"
    localPort="1099" />
  <applications>
    <systemApplication>
      <capabilities>
        <capability name="CORE">
          <interpreters>
            <!-- list all interpreters of the system application – see Figure 5-36 -->
          </interpreters>
        </capability>
      </capabilities>
    </systemApplication>
  </applications>
</node>

```

Figure 5-35: Skeleton of a node descriptor file

Figure 5-36 lists all the interpreters of the system application's core capability omitted in Figure 5-35. The package paths are truncated to avoid line breaks. For a node to be able to start up, the fully qualified class names of the interpreters have to be provided so that the class loader can load the interpreter classes and instantiate them. The core capability of the central and a node list the same interpreters as shown in Figure 5-36. The interpreters that are rede-

defined for the special behavior of the central have a package path that ends with “central”. Those that need no redefinition end with the default package name “node”.

```
<interpreter>...central.NodeStartedInterpreter</interpreter>
<interpreter>...central.RegisterNodeInterpreter</interpreter>
<interpreter>...central.DeregisterNodeInterpreter</interpreter>
<interpreter>...node.DeregisterNodeFinalInterpreter</interpreter>
<interpreter>...node.RegisterApplicationInterpreter</interpreter>
<interpreter>...node.DeregisterApplicationInterpreter</interpreter>
<interpreter>...central.NodeShutdownRequestInterpreter</interpreter>
```

Figure 5-36: A node’s core interpreters as listed in the node descriptor

CAS_FIBONACCI Application

The main part of the node descriptor is the section that defines a node’s applications. A node’s applications consist of the system application, which has already been introduced, and applications defined by the user. User-level applications may also be defined programmatically after the node has started up. On the contrary, the system application can only be defined in the node descriptor.

The Fibonacci application is defined as a user-level application in the node descriptor. Its description is shown in Figure 5-37. It is a very simple application named `CAS_FIBONACCI` (application names must not contain dots) that consists of a single capability named `CORE`, which is served by a single agent named `CALCULATOR` and defines two interpreters.

```
<application name="CAS_FIBONACCI">
  <capabilities>
    <capability name="CORE">
      <agents>
        <agent name="CALCULATOR" executeWhenStarted="
          org.objectscope.janet...FibonacciStartViewCommand"
        />
      </agents>
      <interpreters>
        <interpreter>
          org.objectscope.janet...FibonacciStartViewInterpreter
        </interpreter>
        <interpreter>
          org.objectscope.janet...FibonacciInterpreter
        </interpreter>
      </interpreters>
    </capability>
  </capabilities>
</application>
```

Figure 5-37: Definition of the `CAS_FIBONACCI` application

After the node has started up, the `FibonacciStartViewCommand` specified by the `execute-WhenStarted` attribute is executed, which invokes the `FibonacciStartViewInterpreter` that starts the sample's user interface. The interpreter `FibonacciInterpreter` calculates the Fibonacci number and sends the result back to the requesting agent. The full package path of commands and interpreters has been omitted in Figure 5-37 for sake of readability. The full package path is `org.objectscape.janet.cas.demo.fibonacci`.

Figure 5-38 shows how an application is defined programmatically and registered with the central (try-catch-blocks are omitted for brevity).

```
String pathToNodeDescriptor = "./nodes/fibonacci";
NodeAccessor node = NodeStarter.start(pathToNodeDescriptor);
Application app = new Application("CAS_FIBONACCI");
Capability cap = new Capability(app, "CORE", "CALCULATOR");
cap.add(new FibonacciInterpreter());
node.registerApplication(app);
```

Figure 5-38: Registering an application programmatically

After registration (last line in Figure 5-38) the application is known by all nodes in the cluster with all its capabilities and agents. From now on, any agent of the same application on any node in the cluster can communicate with the agents of the newly registered application by sending commands or by signaling events. The user is responsible that the same capabilities of an application are defined the same way on every node. However, not every node may need an application's full set of capabilities. Nevertheless, on all nodes with the same application, a capability must be defined the same way as on another node with the same capability.

5.3.3 Node Startup

At node startup the `NodeStarter` creates the node with all its subobjects and parses the node descriptor. After that the node's applications are created and registered within the cluster. All interpreters defined in application capabilities are loaded with the use of the Java class loader. Every interpreter knows which command it interprets which allows the respective scheduler to dispatch a command to an interpreter.

At the end of the startup process the node's system view is displayed on the screen (the node can be run headless by setting the attribute `showGUI` in the node descriptor to false). For every node the applications tab of the system view shows all applications registered by every node in the cluster. Figure 5-39 displays the node main view application tab of the node "AARHUS-2" on a workstation named "AARHUS". As the application tag shows, the cluster consists of three nodes that all reside on the workstation "AARHUS" (the central is not displayed as the user should not use it to define applications). The three nodes are named "AARHUS-0", "AARHUS-1", and "AARHUS-2" following the Janet node naming convention (which cannot be changed by the user as Janet needs to make sure that a node name is unique within the cluster). All nodes define the application `CAS_FIBONACCI` with the capability `CORE` and one agent named `CALCULATOR`. For the node "AARHUS-2" itself the system application and the interpreters of the capabilities are listed as well.

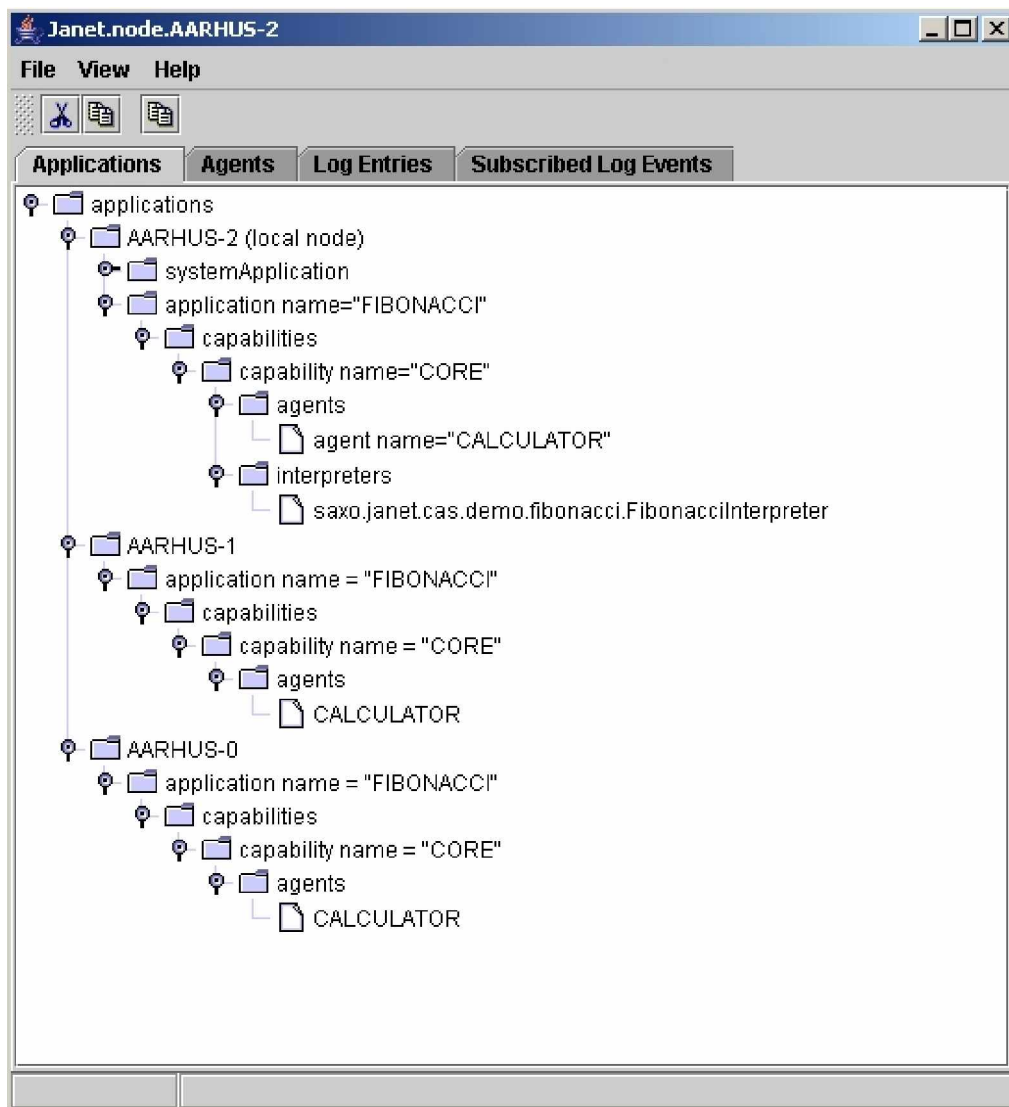


Figure 5-39: Applications tab of the node main view

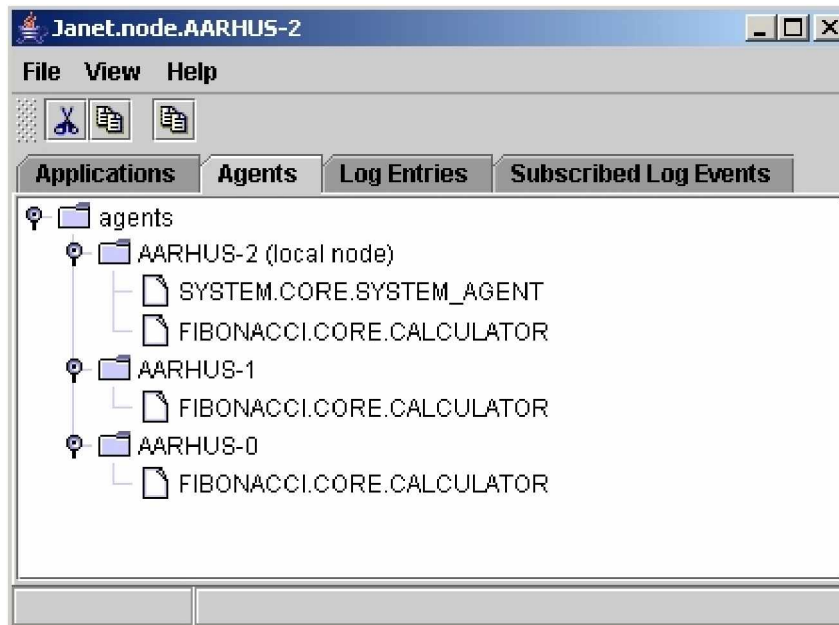


Figure 5-40: Agent tab of the node main view

The agent tab of the node main view displayed in Figure 5-40 shows all nodes in the cluster as defined above with the agents they define. For the node “AARHUS-2” itself the system agent is listed as well.

After node startup all agents are waiting for input. Nothing will happen until some agent sends a command to another agent. Sending a command can, of course, be done programmatically. For convenience, the node descriptor allows to define a command for every agent, which is sent to the agent after its application has been registered.

```
<agents>
  <agent name="Calculator" executeWhenStarted="
    org.objectscape.janet.cas.demo.fibonacci.StartFibonacciCom
    mand">
</agents>
```

Figure 5-41: Defining an agent's initial command

Figure 5-41 shows the definition of the `Fibonacci.Core.Calculator` agent, for which the command `StartFibonacciCommand` has been defined. When node startup has completed, the `StartFibonacciCommand` is sent to the `Fibonacci.Core.Calculator` agent. The `StartFibonacciCommand` has been defined in such a way that it retrieves all the existing `Fibonacci.Core.Calculator` agents from the other nodes in the cluster from its application (which defines an interface for the user for doing this) and sends all of them a `FibonacciCommand`. All agents that receive the `FibonacciCommand` start calculating a Fibonacci number (their capabilities define the same interpreter). The sending agent waits till all calculated Fibonacci numbers have been sent back and displays them on the console.

5.3.4 Implementing Commands and Interpreters

After the Fibonacci application has been defined in the node descriptor the commands and interpreters of the `Fibonacci.Core` capability have to be implemented.

Implementing the `FibonacciCommand`

Commands are typically easy to implement since they simply serve to carry parameters of a command from agent to agent. Figure 5-42 shows the class skeleton of the `FibonacciCommand`. A command must implement the interface `ICommand`. The interface `Serializable` must be implemented as well for the command to be serializable when passed from one agent to another using RMI. The class attribute `input` is used to store which Fibonacci number is to be calculated.

```
import java.io.Serializable;
import org.objectscape.janet.cas.scheduling.ICommand;

public class FibonacciCommand implements ICommand, Serializable
{
    public static final String QualifiedName =
        "org.objectscape.cas.demo.fibonacci.FibonacciCommand";

    protected int input = -1;

    public FibonacciCommand(int input)
    {
        super();
        this.input = input;
    }

    public int getInput()
    {
        return input;
    }

    // other required methods of interface ICommand omitted
}
```

Figure 5-42: `FibonacciCommand` class skeleton

The `ICommand` interface requires several additional methods to be implemented to return the qualified name of the command, the command's priority, how to clone the command, and what display string to use for the command when displayed in a logger. The definitions of these required methods are displayed in Figure 5-43.

```
public String getQualifiedName()
{
    return QualifiedName;
}
```

```

public int getPriority()
{
    return ICommand.ApplicationMaxPriority;
}

public String toLogString()
{
    return "FibonacciCommand"; //$NON-NLS-1$
}

public Object clone() throws CloneNotSupportedException
{
    FibonacciCommand cmd = (FibonacciCommand) super.clone();
    return cmd;
}

```

Figure 5-43: Implementing the ICommand interface for the FibonacciCommand

It has to be noted that the highest priority of a user-defined application command cannot be higher than `ICommand.ApplicationMaxPriority`. If the user specifies a higher priority, the command envelope carrying the command from agent to agent will set back the command's priority to `ICommand.ApplicationMaxPriority` if it is sent to a non-system agent (that is a user-defined application agent). To prevent ambiguities, the qualified command name has to be the fully qualified command name or some other unique id. A command has to be cloned when it is sent to several local agents (which is necessarily the case in simulated mode when several nodes run in the same Java VM) so that every agent receives a command object of which it can change values without affecting other agents. For this reason, a command has to implement the `clone` method from class `Object`.

Implementing the FibonacciInterpreter

An interpreter must implement the interface `IInterpreter`, which only requires two methods to be implemented: `IInterpreter.execute` and `IInterpreter.commandNames`. The implementation of the `FibonacciInterpreter` class is relatively simple. It is displayed in Figure 5-44 below. It displays the full implementation of the `FibonacciInterpreter` (only the `calculate` method is simplified for brevity).

```

import org.objectscope.janet.cas.scheduling.CommandAccessor;
import org.objectscope.janet.cas.scheduling.IInterpreter;
import org.objectscope.util.exception.InvalidMessageException;

public class FibonacciInterpreter implements IInterpreter
{
    public FibonacciInterpreter()
    {
        super();
    }

    public void execute(CommandAccessor cmdAccessor)
    {
        if (cmdAccessor.getCommand() instanceof FibonacciCommand)

```

```

    {
        FibonacciCommand command = (FibonacciCommand)
            cmdAccessor.getCommand();

        try
        {
            BigInteger fibonacciNumber =
                calculate(command.getInput());
            cmdAccessor.setReply(fibonacciNumber);
        }
        catch (InvalidMessageException e)
        {
            cmdAccessor.log(e);
        }
    }
}

protected BigInteger calculate (int nthFibonacciNumber)
{
    // code omitted for brevity
}

public StringVector commandNames()
{
    return new StringVector(FibonacciCommand.QualifiedName);
}
}

```

Figure 5-44: Implementation of the FibonacciInterpreter class

5.3.5 Sending Fibonacci Commands

After the command and interpreter for calculating Fibonacci numbers have been implemented the `StartFibonacciCommand` remains to be implemented (see Figure 5-41) to distribute `FibonacciCommands` to agents in the cluster that have the `Fibonacci.Core` capability. The implementation of the `StartFibonacciCommand` is analogous to the implementation of the `FibonacciCommand` and therefore not explained. The implementation of the `StartFibonacciInterpreter` is of more interest. This interpreter has to get hold of all agents in the cluster that have the `Fibonacci.Core` capability and send them a `FibonacciCommand`. The `StartFibonacciInterpreter` installs callback handlers that are invoked when the calculation results are returned from the `Fibonacci.Core` agents and print the results to the console.

```

public void execute(CommandAccessor cmdAccessor)
{
2      int nthFibonacci = 50000;
3      long startTime = System.currentTimeMillis();
4      NodeAccessor node = cmdAccessor.getNodeAccessor();
5      LoggerAccessor logger = node.getLogAccessor();
6      HistoryTotalsCreator totalsCreator = new HistoryTotalsCreator(
7          agents.size(), nthFibonacci, logger, startTime);
8
9      FibonacciCallbackHandler handler =
        new FibonacciCallbackHandler(logger, totalsCreator);
    IAgentProxy agents = cmdAccessor.getApplication().

```

```

10         getAllAgents(cmdAccessor.getLocalAgentPath());
11         FibonacciCommand cmd = new
12             FibonacciCommand(nthFibonacci), handler);
13         agents.accept(cmd, handler);
14     }

```

Figure 5-45: Execute method of the StartFibonacciInterpreter

The scheduler of an agent invokes the `execute` method of an interpreter in response to a received command. The `execute` method of the `StartFibonacciInterpreter` is displayed in Figure 5-45. Taking all the work into account the `StartFibonacciInterpreter` has to do, its `execute` method looks relatively simple. This is mostly because most of the work is done by two system convenience methods:

- `getAllAgents` (line 11): This method in class `Application` retrieves a list with all agents in the cluster that belong to the same application as the application the `getAllAgents` method was sent to. `cmdAccessor.getLocalAgentPath` in line 11 returns the local agent path of the agent that executes the currently executed interpreter which is `Fibonacci.Core.Calculator`. This local agent path is passed on as a parameter to `getAllAgents`, so that this method knows it has to retrieve all agents in the cluster with the same local agent path on their node. This information is retrieved from the node's anchor cluster image. The `getAllAgents` method returns an instance of class `MultiAgentProxy` (described in section 5.2.3.2: "Sending Commands to Agents: User Interface").
- The `MultiAgentProxy` implements the `IAgentProxy` interface so that the user does not need to be concerned about how to deal with a `MultiAgentProxy`. The user will simply treat the `MultiAgentProxy` the same way as an `IAgentProxy` (using the `accept` message as in line 14). The `MultiAgentProxy` makes sure that a command sent to it, is propagated to all `IAgentProxies` it contains.

After the `MultiAgentProxy` has sent all commands to the destination agents, the thread continues execution. Execution of the `StartFibonacciInterpreter`'s `execute` method has finished and the scheduler will execute the next command in its command queue if there is any. When the calculated Fibonacci numbers have returned the callback handlers created in line 8 + 9 will be invoked. The callback mechanism has been explained in section 5.2.3.3.2: "

Extensions for Callbacks”. The `MultiAgentProxy` makes sure that for every `IAgentProxy` it contains a separate callback handler is installed.

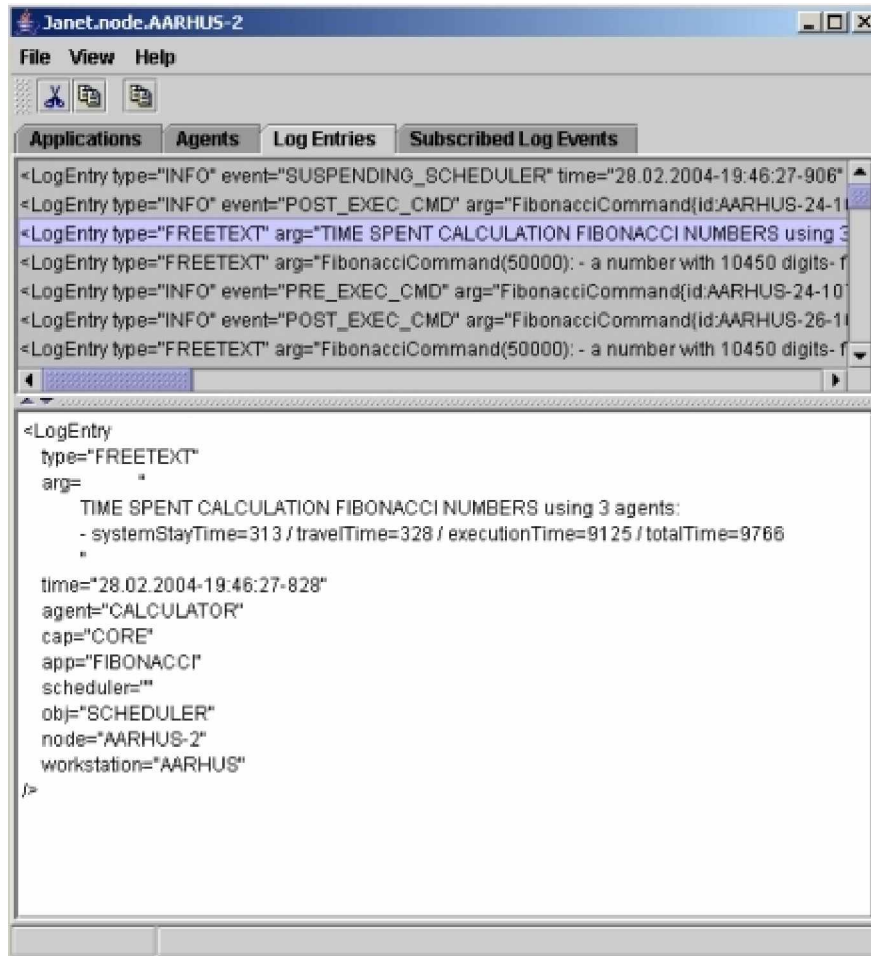


Figure 5-46: Results displayed in log entries tab

When the last calculated Fibonacci number has arrived, the callback handler displays the results in the node's main view log entries tab. The `Logger` used to display log messages is explained in further detail in chapter 5.4: "Driving Application: Remote Logger".

5.4 Driving Application: Remote Logger

Every node has a logger that displays log messages in the node's main view log entries tab. This logger has not been described so far as it is not conceptually important. The remote logger listens to log messages sent to this logger and displays the log messages in a window. The log messages displayed have been sent from all nodes in the cluster that have subscribed to the remote logger service. The remote logger was implemented as a `Janet.CAS` application. It could also have been implemented using existing logging tools for Java such as `log4j`. The former approach was chosen to use the remote logger as a tool to validate concepts of `Janet.CAS` such as the concept of a node, of which the system behavior can be changed according to the role

the node is intended to fulfill, or defining user-level applications in order to add new functionality.

Implementing the remote logger as a node of its own, it could be implemented as a system facility being served by the system agent or as a user-defined application being served by an application agent. Remote logging is not a core facility that is needed to implement a node. Propagating log messages to a remote logger agent produces CPU load the system agent should not have to carry, as it needs to remain available to fulfill tasks critical for running the node. For that reason, the remote logger is not be provided by the system. It is implemented as a user-defined application. This means that the agents of the remote logger facility run at application priority. Heavy logging activity can therefore neither bog down the system agent (which runs at higher priority than the application agents) nor can it cause starvation of other applications, since the arbitrator makes sure that every application obtains the same time slice.

5.4.1 Consumer and Producer Capability

For the remote logger a capability is needed that accepts remote log messages and displays them in a window for the user to see. This capability is called the `Consumer` capability. It needs one agent, which is called the `Logger` agent. The agent path to the consumer logger agent is therefore: `Remote_Logger.Consumer.Logger`. One node is defined with the `Remote_Logger.Consumer` capability as the sole user-defined application capability. This node takes the entire load to display remotely received log messages. It can be run as the sole node on a workstation so that it does not use up resources of other nodes. It is also possible to start up several nodes with a `Remote_Logger.Consumer` capability.

```
<application name="REMOTE_LOGGER">
  <capabilities>
    <capability name="CONSUMER">
      <agents>
        <agent name="LOGGER" executeWhenStarted=
          "org.objectscope.janet.cas.logger.commands.StartView-
            Command" />
      </agents>
      <interpreters>
        <interpreter>
          org.objectscope.janet.cas.logger.interpreters.Display-
            LogEntryInterpreter
        </interpreter>
        <interpreter>
          org.objectscope.janet.cas.logger.interpreters.Start-
            ViewInterpreter
        </interpreter>
      </interpreters>
    </capability>
  </capabilities>
</application>
```

Figure 5-47: Definition of the remote logger consumer capability

As showed in Figure 5-47 the consumer capability defines two interpreters:

- **StartViewInterpreter:** Executed after the node has started up. This interpreter starts up the window to display remotely received log messages. It then sends the `InstallLoggerCommand` to all nodes in the cluster with the remote logger producer capability (introduced below). When the node shuts down the `UnInstallLoggerCommand` is sent to all these nodes.
- **DisplayLogEntryInterpreter:** Displays all the log messages received from another node through the `DisplayLogEntryCommand` in a window.

All nodes of which the log messages are to be displayed by the `Remote_Logger.Consumer.Logger` agent, define a different capability. This capability is called the **Producer** capability since the node it belongs to “produces” log messages. The agent path to the producer logger agent is: `Remote_Logger.Producer.Logger`. This agent installs a listener that listens to log messages sent to the node’s logger and passes them on the consumer agent. When the consumer node shuts down, the listener is uninstalled.

```
<application name="REMOTE_LOGGER">
  <capabilities>
    <capability name="PRODUCER">
      <agents>
        <agent name="LOGGER" />
      </agents>
      <interpreters>
        <interpreter>
          org.objectscope.janet.cas.logger.interpreters.
            InstallLoggerInterpreter
        </interpreter>
        <interpreter>
          org.objectscope.janet.cas.logger.interpreters.
            UnInstallLoggerInterpreter
        </interpreter>
      </interpreters>
    </capability>
  </capabilities>
</application>
```

Figure 5-48: Definition of the remote logger producer capability

The definition of the producer capability is displayed in Figure 5-48. It defines two interpreters:

- **InstallLoggerInterpreter:** Installs a listener in the logger. The listener passes the log messages sent to the logger on to the `Remote_Logger.Consumer.Logger` agent.
- **UnInstallLoggerInterpreter:** Uninstalls the listener.

5.4.2 Starting up a Consumer Node

When a consumer node starts up, the `StartViewCommand`, defined in the `Logger` agent's `executeWhenStarted` attribute, is executed. It starts the consumer window to display log messages and sends an `InstallLoggerCommand` to every producer node in the cluster. This scenario is displayed in Figure 5-49.

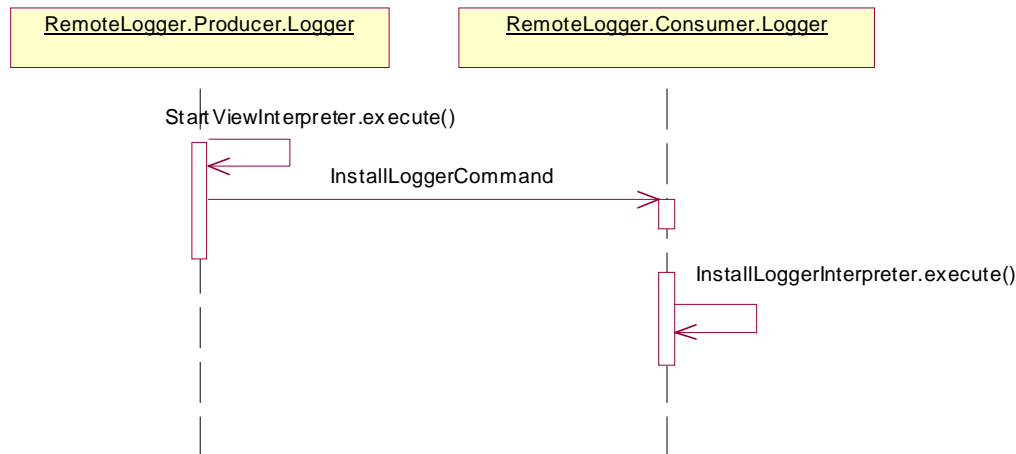


Figure 5-49: Starting up nodes with remote logger capabilities

The consumer agent knows from its node's cluster image which nodes in the cluster have the producer capability. When a producer node starts up and a consumer node is already present, the consumer node needs to receive some kind of notification that a new producer node exists. At startup the consumer node registers for the `FOREIGN_APPLICATION_REGISTERED` event by installing its own `ForeignApplicationRegisteredHandler`. Whenever a node starts up, it tells all other nodes in the cluster about the applications and agents it has. Analogously, when a new application is registered, the hosting node informs all other nodes about the new application and its agents as well. A node that receives such a notification, signals the `FOREIGN_APPLICATION_REGISTERED` event. The consumer agent's `ForeignApplicationRegisteredHandler` is invoked and the consumer agent sends the `InstallLoggerCommand` to the newly started up node.

5.4.3 Starting up a Producer Node

After a producer node has started up it receives the `InstallLoggerCommand` from every consumer node in the cluster. A producer node need not be concerned about looking up a consumer node and registering with it. When it shuts down it does not need to tell the consumer node about it, either. It will simply cease to send log message to the consumer agent for display. When the `InstallLoggerCommand` is executed the log listener is installed in the node's logger. From that moment on, the log listener receives a notification each time a log message is

Chapter 5

sent to the logger and passes the log message on the consumer agent enclosed in `DisplayLogEntryCommand`.

Installing the Log Listener Using An Event

The logger is a system object that may be used by user-defined applications for logging as well. An application that not only uses the logger for logging, but also listens to log messages sent to the logger is able to track down what actions agents of other applications are carrying out. This would infringe the protection of the system application and of the user-defined applications. For this reason, the logger has no public interface to install a log listener. The installation of a log listener therefore has to be done with the use of an event (events have been introduced in section 5.1.4). The installation of the log listener is only initiated by the `InstallLoggerInterpreter`, which signals an event that makes the system install the log listener.

In the `exportedEvents` section of the system application, the user has to define which handler is invoked in case the `EVENT_ADD_LOG_LISTENER` is signaled. The respective section of a producer's node descriptor is shown in Figure 5-50. Since the event is defined for the system application, the system agent will run the event handler, which has the privilege to install a log listener. The node descriptor is under control of the workstation owner. The workstation owner has therefore full control over what should happen when the `EVENT_ADD_LOG_LISTENER` event is signaled as she can define the handler to be invoked in response to the event. The default system handlers can be defined or user-defined ones. If the `exportedEvents` section does not contain an entry for the `EVENT_ADD_LOG_LISTENER` event, nothing will happen when the event is signaled. The handlers are either installed programmatically or by extending the node descriptor file. The default remote logger event handlers simply add or remove a listener to the logger.

```
<!--lines omitted >
<systemApplication>
  <exportedEvents>
    <event name="EVENT_ADD_LOG_LISTENER"
      handler="org.objectscape.util.logging.AddLogListenerHandler" />
    <event name="EVENT_REMOVE_LOG_LISTENER"
      handler="org.objectscape.util.logging.RemoveLogListenerHandler" />
  </exportedEvents>
<!--lines omitted >
</systemApplication>
<!--lines omitted >
```

Figure 5-50: Extract from node descriptor file to install event handlers

Figure 5-50 shows an extract from a node descriptor file that defines event handlers to add and remove log listeners. The default handlers installed are `org.objectscape.util.logging.AddLogListenerHandler` and `org.objectscape.util.logging.RemoveLogListenerHandler`.

5.4.4 Shutting down a Consumer Node

When the consumer node shuts down, it tells all producer agents to disconnect themselves from their node logger, which results in no more log messages being passed on to the consumer

Chapter 5

agent. The consumer agent sends the `UnInstallLoggerCommand` to all producer nodes. The `UnInstallLoggerInterpreters` uninstalls the log listener using the `EVENT_REMOVE_LOG_LISTENER` event.

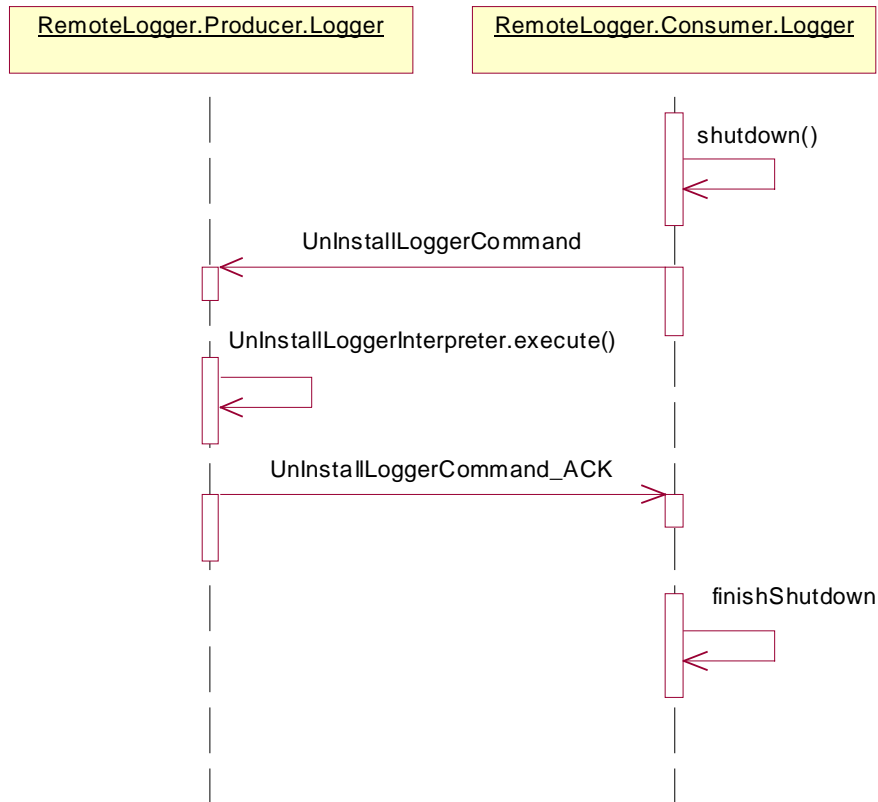


Figure 5-51: Shutting down a consumer node

The consumer node waits till all producer nodes have finished the log listener uninstallation process and then shuts down the node itself, which is shown in Figure 5-51.

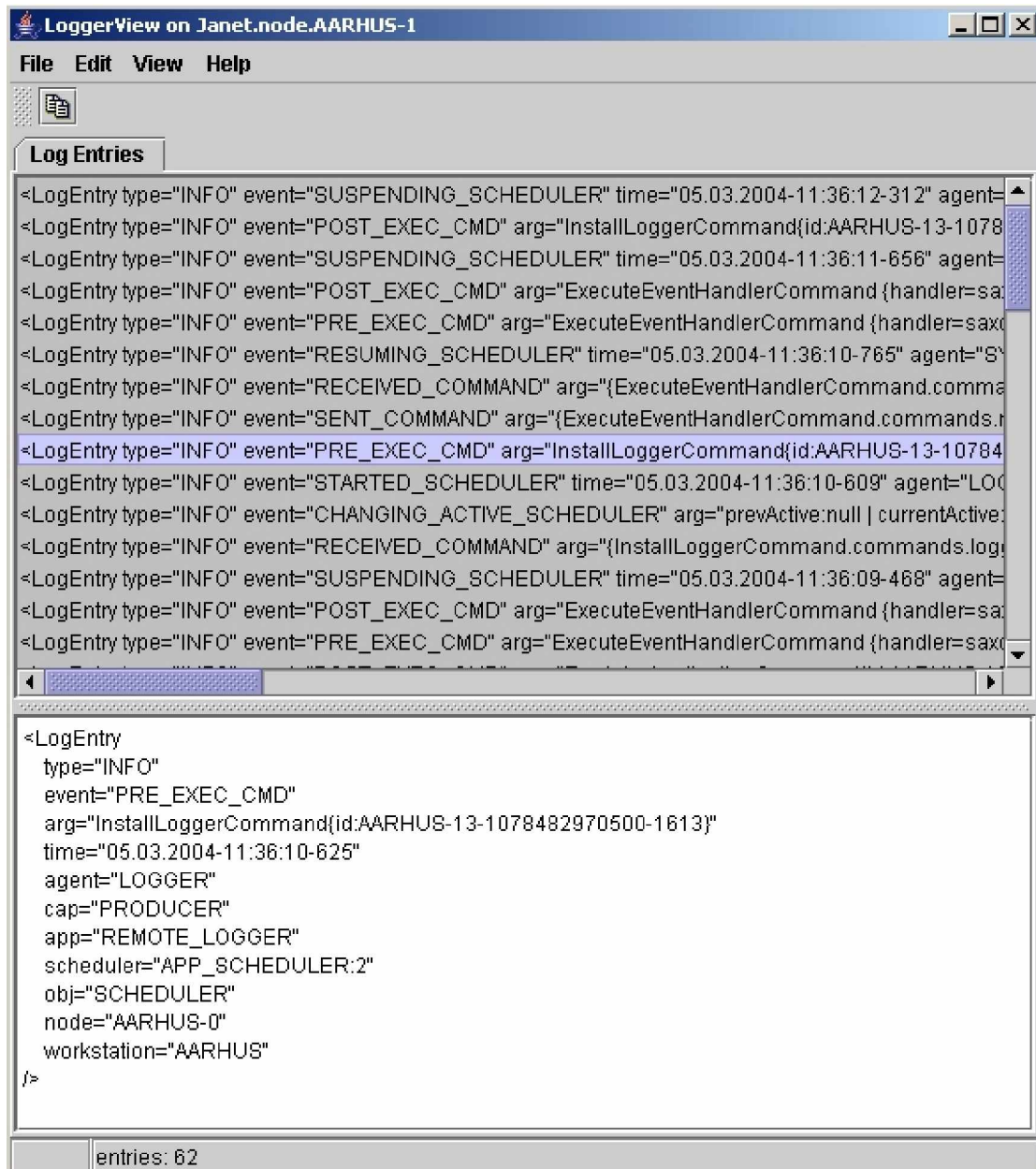


Figure 5-52: Consumer agent log view

Figure 5-52 shows the consumer log view that displays all log messages received from producer nodes. The selected log entry shows a log message from the node AARHUS-0, which indicates that the `InstallLoggerCommand` is about to be executed by the `Remote_Logger.Producer.Logger` agent.

5.5 Measurements

The `CAS.Fibonacci` sample application already presented can be used to measure Janet.CAS system performance. The `CommandEnvelope`, after being sent away to another agent and upon arrival at the destination agent, measures the time that has passed by. These durations can be used to measure command travel times. In the following measurements are taken with more

and more nodes being part of the cluster to see how command travel times are affected when the cluster size grows.

5.5.1 Overview

General Measurement Scenario

The following measurements are based on a scenario where Fibonacci commands are distributed evenly in the cluster. For each new measurement the cluster size is augmented by one node. A Fibonacci agent on one node in the cluster sends a Fibonacci command to itself and to all other Fibonacci agents in the cluster where every workstation in the cluster hosts one node with one Fibonacci agent. Every Fibonacci agent calculates the same Fibonacci number, which is the 50,000th Fibonacci number, and sends the result back to the initially sending agent. To reduce the effect of outliers the calculations are repeated ten times for every cluster size. Maximum, minimum, and average values refer to the maximum, minimum, and average value of ten measurements for a cluster with a given size. All measurements measure overall time.

Setup

All workstations that are used to form a cluster are setup as follows: Pentium III, 512 MB, 1 GHz, SuSE Linux 8.2. Network: 100 Mbit/s Ethernet switched.

Garbage Collection

All measurements are time overall measurements that also include time spent on garbage collection. The Java VM option `-verbose:gc` prints various garbage collection statistics to the console. This is a sample output:

```
[GC 325407K->83000K(776768K), 0.2300771 secs]
[GC 325816K->83372K(776768K), 0.2454258 secs]
[Full GC 267628K->83769K(776768K), 1.8479984 secs]
```

The statistical output is useful to determine whether the garbage collector has become a performance bottleneck for an application. If this is the case, it can be used to verify whether the tuning of the garbage collector shows the expected results. However, the garbage collector statistics is not provided in a way that it can be processed automatically to calculate overall time spent without garbage collection. The time spent on garbage collection is therefore not separated out but included in the measurements.

5.5.2 Measuring Total Time

The total time a Fibonacci commands consumes during its lifetime with more and more nodes being added to the cluster is supposed to give some impression about how well Janet.CAS scales with growing cluster size.

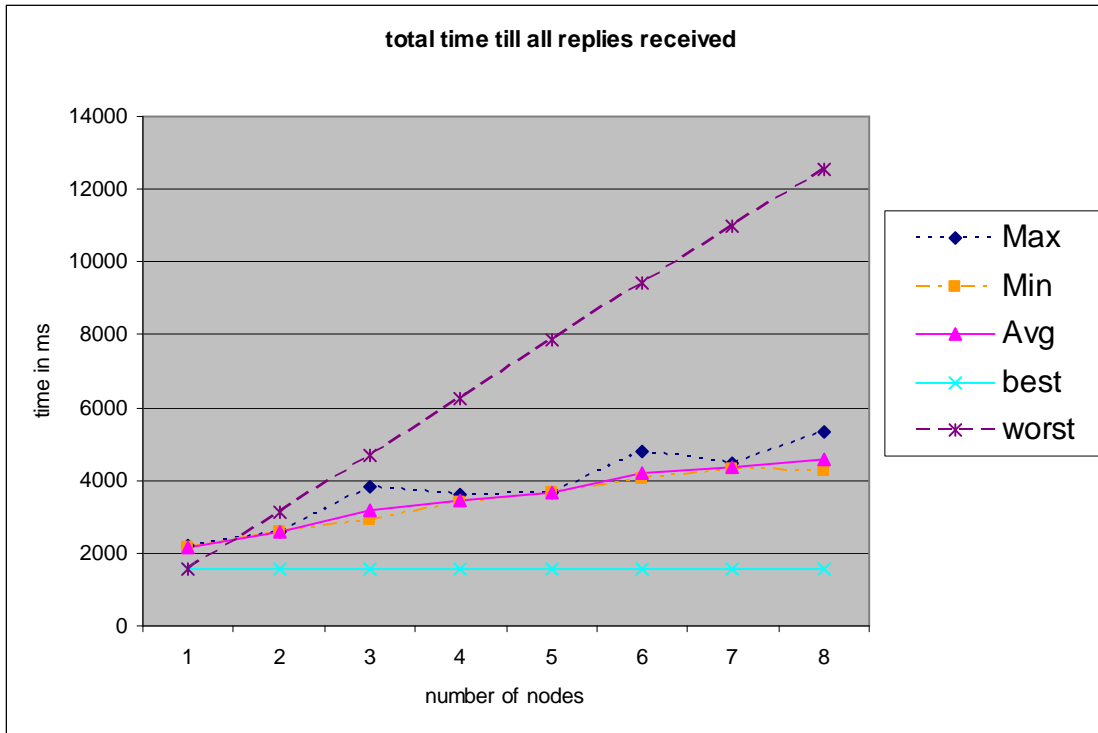


Figure 5-53: Total time till received replies from all nodes

Figure 5-53 shows the total time to calculate Fibonacci commands distributed evenly in the cluster. The total time is measured from the time on the Fibonacci commands where sent away till all replies have arrived.

The *lower horizontal line* shows the time required to calculate the Fibonacci number without the overhead caused by Janet.CAS. A most simple program not using Janet.CAS was written that enters a loop to calculate the Fibonacci number, prints the time spent for calculation to the console and exits. In the ideal case where there is no overhead caused by the cluster software or network the time to calculate n Fibonacci numbers is the same as for calculating one Fibonacci number. The horizontal line represents this ideal case.

The *steepest line* represents the worst case where there is no distributed calculation and calculating n Fibonacci numbers takes n times as much as when calculating it once. This line was obtained by multiplying the time for calculating the Fibonacci number without Janet.CAS by the number of nodes.

The lines for maximum, minimum, and average are in between the lines for the two extreme situations described above. The line with triangular symbols for the average time shows close to linear growth. The line that displays averages is very close to the one for the minimums where the line for maximums is in some points relatively far away from the line for averages. This shows that total time till a Fibonacci command has returned to its sending agent can vary considerably.

5.5.3 Janet.CAS Measuring Interface

When a command is sent from an agent to another agent, Janet.CAS does some bookkeeping to measure the time the command spends during various phases of its life time:

- travel time: the time spent after the command was sent away till it was received by the destination node's agent dispatcher. This is the time consumed by the middleware and the network to transport the command physically to another process.
- execution time: the time the commands spends for the execution of its interpreter.
- system stay time: all the time the command does not spend on traveling from agent to agent and executing its interpreter. This is the time spent after the command was created till it was sent away plus the time spent after reception by the destination node's agent dispatcher till the command was executed. In case the command is sent back to the sending agent with a result the system stay time for sending the command back is added.
- total time: sum of travel time, execution time, and system stay time.

Janet.CAS provides an interface for the user to query travel time, execution time, system stay time, and total time. Figure 5-54 shows a simplified code sample to obtain statistical information from a command.

```
// ...

IAgentProxy agent = getMyAgent();
FutureResult futureResult = new FutureResult();
agent.accept(new FibonacciCommand(50000), futureResult);
BigInteger result = (BigInteger) futureResult.getResult();

ICommandHistorySummary summary = new CommandHistorySummary();
futureResult.createHistorySummary(summary);

System.out.println(summary.getSystemStayTime());
System.out.println(summary.getTravelTime());
System.out.println(summary.getExecutionTime());
System.out.println(summary.getTotalTime());

// ...
```

Figure 5-54: Interface to obtain statistical command information

After the result has arrived the user can query statistical command information from the `FutureResult` object by supplying her implementation of the `ICommandHistorySummary` interface. If supplied to several commands the `ICommandHistorySummary` object contains the totals of all the measured durations. The interface remains the same when using an `Acknowledge` object instead of a `FutureResult` object or when using a callback handler. For the measurements presented in this chapter callback handlers were used instead of `FutureResult` objects.

5.5.4 Measuring Command Travel Times

It can be expected that the time a command spends within the Janet system and the time spent for executing a command remains for the most part constant. However, the more nodes are in the cluster the more commands are sent through the network and the network load increases.

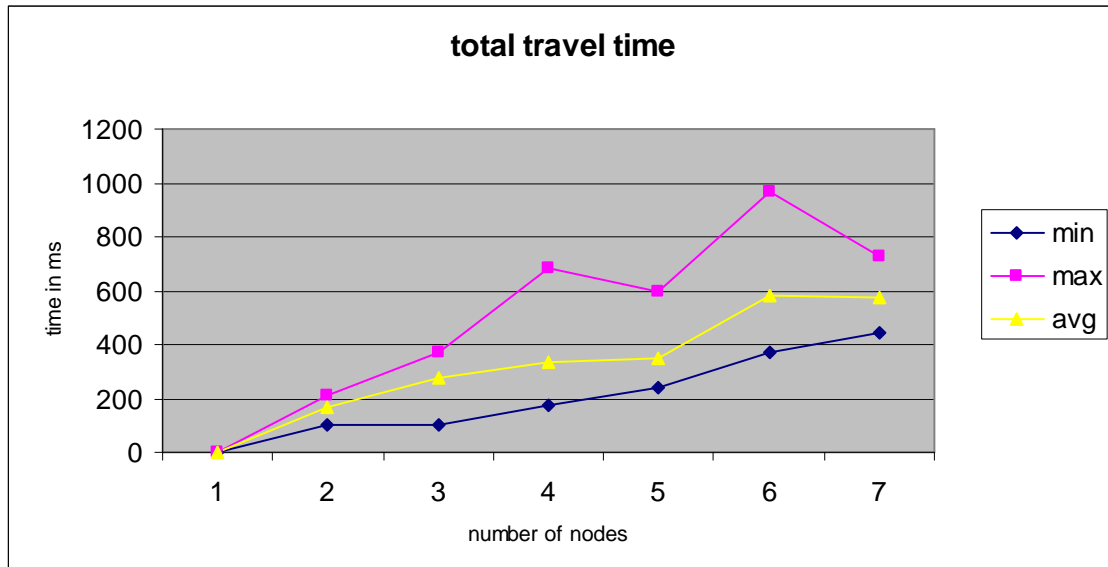


Figure 5-55: Depiction of command travel times

Figure 5-55 shows travel times for the Fibonacci command from 1 to 7 nodes. If only a single node is present, the command is sent to an agent that lives in the single node itself. In this case, sending the command to the agent becomes a simple message send without serialization. As the figure shows, for a single node travel time falls below the smallest unit of measure, which is 1 millisecond. Travel times increase the more nodes exist in the cluster. It can be seen from the figure that the growth of travel times remains linear. It is noteworthy that maximum travel times vary in a relatively large range. If the number of nodes in the cluster is relatively small, the maximum travel time can be up to twice as much as the average travel time.

5.5.5 Measuring Command Execution Times

The execution time of a Fibonacci command is for the general measurement scenario not particularly interesting. Since every node executes only one command at a time and nodes do not share CPUs it can be expected that execution time remains mostly constant.

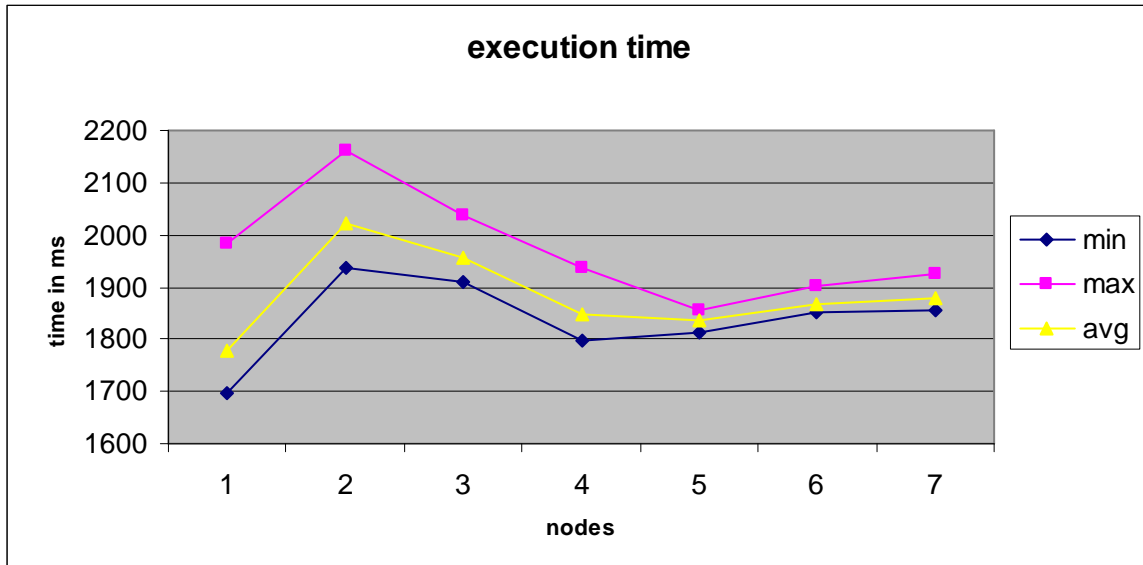


Figure 5-56: Total execution times

Figure 5-56 displays total execution time of commands in clusters varying in size from 1 to 7 nodes. The curves for minimum, average, and maximum execution time are relatively close together indicating that outliers have no significant effect.

5.5.6 Comparing Average Values

In Figure 5-57 the curves for the averages of travel times, system stay times, and total execution time have been depicted in one diagram.

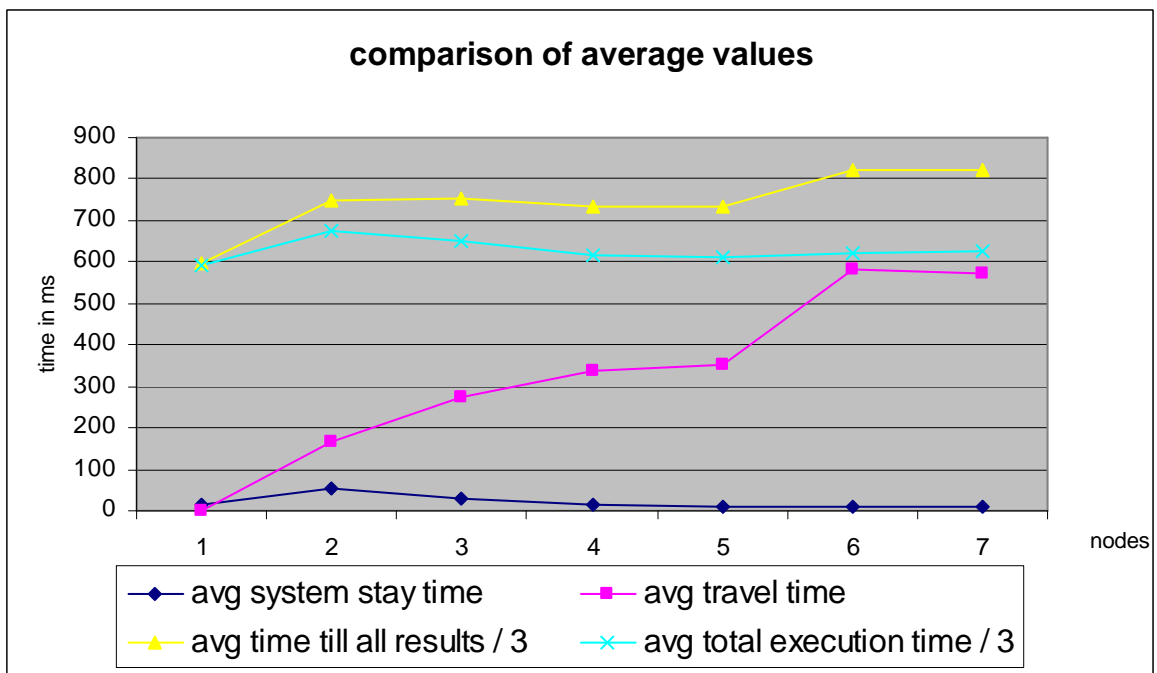


Figure 5-57: Comparison of different curves for averages

The averages for total execution time and the time till all results have been received by the agent sending out Fibonacci commands have been divided by 3 to make them better fit into the figure. The figure shows that average system stay time is relatively low whereas average travel time is high. This indicates that command travel time is the most important time consumer. Since travel times keep growing with the number of nodes in the cluster, travel time is the most important factor for potentially limiting cluster scalability. Average total execution time remains mostly constant, which was expected as for the given general measurement scenario workload for every workstation in the cluster remains the same when the size of the cluster grows.

Average travel time increases sharply from 5 nodes to 6 nodes. It is interesting to see that the curve for the average time till all results have arrived increases sharply at this point as well. It seems to be mostly in parallel with the curve for average travel time with the difference that it does not increase that strong. This confirms the presumption that command travel time is the most important factor in overall response time. The slower increase can be explained by the average total execution time and the average system stay time decreasing slightly with the number of nodes in the cluster rising, which lowers the increase of the overall total.

6 Janet.ADÉ

Janet.ADÉ (**A**utomatic **D**istributed **E**xecution) is the layer that implements workload balancing in Janet. It makes use of the cooperative agent system provided by Janet.CAS to make agents evict commands to less loaded nodes. Janet.ADÉ defines two new command interfaces: `IWorkLoadSharingCommand` and `IWorkLoadBalancingCommand`. The system applies workload sharing if the user defines a command that implements the former command interface which results in the system sending the command to the least loaded node for execution where it resides till execution has completed. In the following, commands that implement this interface are called workload sharing commands. If the user defines a command that implements the latter interface, workload balancing applies. In this case, the command is added to the scheduler queue of the agent it was sent to. If the node hosting this agent is overloaded, the command may be suspended, sent to some other less loaded node, and then be restarted. Such a command is called a workload balancing command in the following. Both kinds of commands are called workload-aware commands. Contrary to workload balancing commands, workload sharing commands cannot be suspended after they have been started. However, they may be evicted as well as long as they remain waiting in an agent's scheduler queue.

Workload-aware commands in Janet.ADÉ are intended for the implementation of long running tasks, for which the overhead caused by workload determination and command eviction can be justified in order to balance overall workload in the cluster. For short running tasks the `ICommand` interface of Janet.CAS is intended to be used.

6.1 Conceptual Design

This section explains the conceptual design of Janet.ADÉ. It describes the roles of the nodes defined to accomplish workload balancing, the balancing algorithm that is used, how it has been developed and what the considerations have been to choose such an algorithm.

6.1.1 Load Determination

As mentioned in chapter 2.1 “Algorithms for Automatic Load Balancing” there are several methods to choose a node to migrate commands to. The most important ones are: Random, Threshold, Shortest Queue, and Broadcast and Bidding. Random delivers significantly better results than without load balancing. Threshold and Shortest Queue deliver even better results. Broadcast and Bidding results in significantly increased network traffic which is especially a problem in Janet since commands in Janet take more time as remote procedure calls in distributed operating systems. Commands in Janet are sent to agents. Then they are added to a scheduler, which processes the command after all previous ones have been processed. Furthermore, dispatch of command and interpreter by the scheduler is dynamic. For this reason, Broadcast and Bidding is considered too costly for Janet and is not further investigated as an option.

Shortest Queue and Queue Size Categories

The Shortest Queue method is especially interesting for workload balancing in Janet. In Janet commands sent to agents are added to a scheduler queue from which they are executed one after the other. Counting the number of queues with their number of commands returns a measure for the load of the system that is simple to obtain and conceptually consistent with the

agent-scheduler-queue paradigm used in Janet, which makes the Shortest Queue method an obvious candidate method to determine a node's load in Janet. The time a command has to wait till it is processed is another measure that can be used to determine a node's load. However, the drawback of the Shortest Queue method is that a node needs to inform a central node about the total size of its queues whenever it changes, which causes more overhead than when using the Threshold method. As already mentioned, commands sends in Janet are relatively costly, which means that an approach resulting in heavy message traffic should be avoided. To address this problem the Shortest Queue method is extended by introducing queue size categories (QSC) where the user can define several categories of queue numbers and queue sizes. For example, for queue size category 0 (QSC0) all command queues are empty and there are no executing commands (the node is idle). For queue size category 1 (QSC1) there is only 1 executing workload-aware command in all of the node's scheduler queues. For queue size category 2 (QSC2) there is at least 1 executing command and at least 3 waiting commands. A node only informs a central node about queue size changes in case it changes state from one QSC to another. Using this approach, the Shortest Queue method can be used for workload determination while keeping the message traffic between nodes and a central balancing node low. Another advantage of queue size categories is that they can be defined individually for every node. The QSC definition for a node that resides on a workstation with considerable more CPU power than other workstations in the cluster can be set up in such a way that queue size categories contain a larger number of executing commands and waiting commands than in the QSC definition of nodes on less powerful workstations.

6.1.2 Distribution Algorithm

As shown by Mirchandaney et al. [MTS89] a symmetric distribution algorithm, which is a combination of sender-initiated and receiver-initiated algorithm, performs better than either single algorithm. In the receiver-initiated approach, an underloaded node requests commands from a heavily loaded node. A node in Janet would therefore have to retrieve heavily loaded nodes from some load data store and request commands from it. It makes sense to have a central load data store to minimize the number of notifications across the network about changes of queue size categories (as described earlier, broadcasts in Janet are costly and should be avoided). The question is whether the central data store should remain a passive store or whether it should be able to carry out actions to assist in the load balancing process. The result of a central data store that remains a passive object is a distributed algorithm, rather than a centralized one. The more "intelligence" is given to it, the more centralized the distribution algorithm becomes. The number of commands exchanged between agents to balance workload increases if nodes have to interact with an additional central data store agent. In the worst case, all communication between nodes goes through the central data store agent, which means that the number of commands exchanged in the cluster doubles compared to direct node-to-node communication. For efficiency reasons, the number of exchanged commands is to be kept at a minimum, which means that the central data store agent receives most of the intelligence or functionality to carry out workload balancing. A centralized approach with a central workload balancer has been chosen to minimize command traffic.

Cost of Command Janet Sends

This paragraph tries to provide a short explanation why command sends in Janet are relatively costly and why high command traffic in Janet is likely to cause a performance problem. Command sends in Janet are more costly than RPC calls used in several distributed operating systems. Java RMI, which is used in Janet for communication between distributed agents, relies

on RPC calls as well. However, Java RMI is a high-level RPC-based layer to provide location transparency for objects. Many more RPC calls are sent on the socket layer to make a distributed object communicate with another one seamlessly than when making a single RPC call in a distributed operating system. In addition, sending a command in Janet from one agent to another involves additional steps beyond transmitting the command using Java RMI. After being accepted by the recipient agent dispatcher the destination agent has to be looked up from the agent dispatcher's agent table. The command is then sent to the agent, which places it in its scheduler's command queue where it waits till all previous commands in the queue have been executed. As it can be seen, command sends in Janet are relatively costly and several times more costly than RPC calls in a distributed operating system. This is an important fact that has to be taken into consideration when developing a conceptual design for workload balancing.

6.1.3 Executor-Observer-Distributor Triad

The executor-observer-distributor triad defines three roles for objects being involved in the process of workload sharing and workload balancing. The executor is an object that is responsible for executing workload-aware commands. The observer observes a workstation's CPU load. More precisely, it observes the CPU load of processes that do not belong to Janet nodes on a particular workstation. If the owner of the workstation reclaims the workstation for purposes other than executing commands using Janet, the observer will observe that the CPU load of non-Janet processes increases. It will then notify one or more distributor objects about the event, which will decide on actions to be taken. If other less loaded nodes exist in the cluster, workload-aware commands will be evicted from all nodes residing on the affected workstation to less loaded nodes on other workstations.

The three kinds of objects of the executor-observer-distributor triad described so far have only been defined in terms of general roles or responsibilities. These roles are likely to be defined in other workload balancing systems as well. No statement has been made whether these kinds of objects can be seen as kinds of nodes or agents, whether there is one or several distributors, etc. The next step consists of the development of the conceptual design, which needs to provide answers to a multitude of questions: Should workload distribution be decentralized as in MOSIX or should it be centralized? Should there be a single distributor object, node, or agent, or should there be several ones? In case the non-Janet CPU load increases, should the observer tell all nodes on the same workstation to evict all commands or should this responsibility be assigned to the distributor? Should an executor send an evicted command to the distributor, which passes it on to the least loaded node, or should it send it directly to some other less loaded agent? There are several questions of this kind for which a conceptual solution is developed in the following.

6.1.3.1 Distributor

It is important to keep the number of commands low that are exchanged to do workload balancing. This is already the reason for the introduction of queue size categories. If there were more than a single distributor in the cluster, all changes of nodes queue size categories would have to be propagated to all distributors in the cluster. Given the high costs of command sends in Janet such a solution would be considered suboptimal. Janet.ADE therefore defines a single distributor node that may define several agents. Whenever a node's QSC changes, the distributor is informed about the state. In MOSIX state change notification is sent to a random number of nodes in order to reduce message traffic. Such an approach is thinkable for Janet.ADE as well. Another idea would be to have a small number of distributors, to which executors send

QSC change notifications in alternating order. The solution chosen so far with a single central distributor represents an initial approach or implementation step, which leaves room for enhancements after this implementation step has been completed.

Load Administrator Agent

There is a special agent that is solely responsible for processing QSC changes, which is called the load administrator agent. Defining a special agent for this purpose makes sure that QSC changes are being taken care of immediately. If a distributor's agent queue were filled with a lot of commands that serve a different purpose than taking care of QSC changes, the QSC change notifications might not be handled in time. In the worst case, a command is not evicted to the currently least loaded agent, because the latest queue size change notifications with the most recent status information needed to determine the least loaded node are located in the command queue behind the eviction request itself and can therefore not be considered in time.

6.1.3.2 Observer

The observer is defined as a single node on a workstation that only serves the purpose of watching a workstation's CPU load of non-Janet processes. In case the workstation owner reclaims the workstation for other purposes than executing Janet commands, the observer notifies the distributor about the CPU load change if a certain threshold value is exceeded (it does so in the opposite case as well when CPU load drops down from above the threshold value to a level below it). The distributor will then decide where workload-aware commands from these nodes will be sent to for further execution. The indirection of going through the distributor could be saved if the observer told all nodes on the workstation directly to evict all commands. During the further development of the conceptual design it will become evident that saving this indirection does not result in significantly reduced response time to CPU load changes and only further complicates the workload balancing process.

6.1.3.3 Executor

An executor is any node that defines the required capabilities of Janet.ADE to enable workload balancing. When these capabilities are defined for a node, Janet.ADE will make use of that node to balance the cluster's workload. A node may benefit from its commands being executed at different sites, which loads some other workstation CPU or may be used by other nodes for the execution of their commands. Workload-aware commands being sent to agents on nodes that have not these capabilities defined will be treated as commands of the Janet.CAS layer that cannot be seen by the workload distribution facility of Janet.ADE and therefore do not take part in the workload balancing process. An observer or a distributor could be used as an executor as well. However, it is recommended not to use these kinds of nodes as executors in addition to their special roles since the additional load would result in reduced response time to workload changes in the cluster.

6.1.4 Leveling Out Workload Imbalances

Whenever the QSC of an executor node changes it notifies the distributor about the new QSC. The distributor is therefore able to construct an image of the workload of nodes in the cluster.

Workstation	Node	NodeQSC	WorkstationQSC	Product
1	1	1	1 x 2	2
1	2	2	1 x 2	2
2	3	1	1 x 3	3
2	4	3	1 x 3	3
3	5	2	2 x 2	4
3	6	2	2 x 2	4

Figure 6-1: Cluster Queue Size Category Table

The table displayed in Figure 6-1 lists all 6 nodes in a cluster with 3 workstations with their QSC. Workstation 1 is the least loaded workstation with a node QSC product of 2. From this workstation node 1 is the least loaded one with the lowest QSC, which should be chosen when determining the least loaded node to assign a workload-aware command to.

Number of workload-aware commands per capability

It needs to be taken into account, that a command can only be sent to an agent that has the capability required to execute the command. This might not necessarily be the case for node 1 of the table in Figure 6-1. For this reason, whenever an executor's QSC changes, it sends a table of capability paths to the distributor with the number of workload-aware commands that were sent to agents with the capability identified by the capability path (a string consisting of the application name of the application the capability is defined in, which is unique, and the capability's name). The table contains two entries for a list with waiting and executing workload-aware commands by capability path and kind of workload-aware command (sharing or balancing). It is not possible to send information about the queue length of every agent that has received workload-aware commands, as the number of agents being hosted by a node may be very large. Frequent transmission of large amount of QSC data from executors to the distributor would cause a high a load. On the contrary, the number of capabilities of a node can be expected to remain comparatively small.

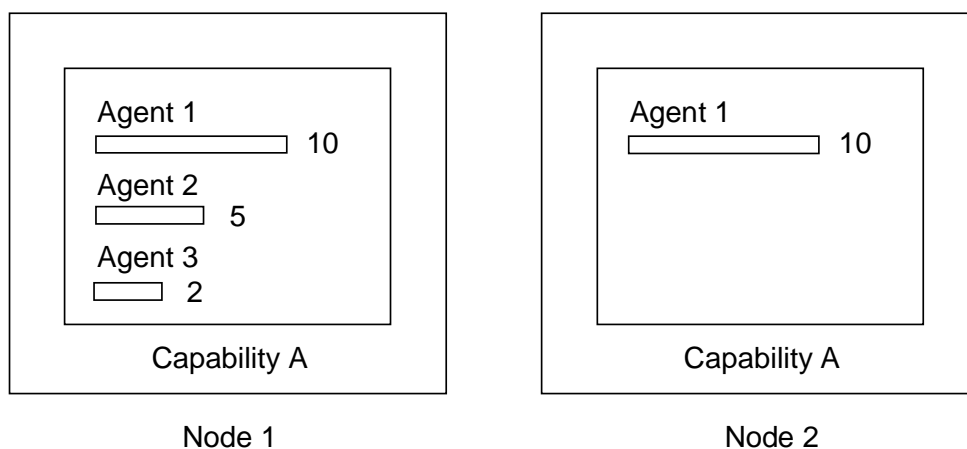


Figure 6-2: Command distribution within capabilities

Figure 6-2 on the previous page shows the same capability A, which exists on the nodes 1 and 2. On node 1, there are 3 agents that have the capability A. On node 1, agent 1 has 10 commands in its queue, agent 2 has 5 and agent 3 has 2. The total number of commands for capability A on node 1 is 17. For node 2 the total number is 10. The distributor knows the total number of workload-aware commands per capability and node. This total number is called the capability queue size (CQS). Whenever an agent is created on a node, through registration of an application that defines certain agents from the beginning or through dynamic creation of an agent at runtime, all nodes in the cluster are notified about the event. The distributor therefore knows about all the agents on all nodes as all other nodes and about their agent paths, which contains the capability name and the name of the agent's application. In the example in Figure 6-2 the distributor knows that the number of commands per agent for capability A is $17 / 3$ for node 1 and $10 / 1$ for node 2. Assuming that commands are to be moved from node 1 to node 2 to distribute workload more evenly, from which agent's queue should commands be taken? Will moving commands from one node to another in this example in the end result in an improvement?

Node	NodeQSC	CQS
node1	5	25
node2	4	10
node3	4	14
node4	6	30

Figure 6-3: Leveling out commands between nodes

There is another problem that is difficult to solve when leveling out commands between nodes. Figure 6-3 shows a table with the QSC for several nodes and their capability queue sizes (CQS). As the table in Figure 6-3 shows, the nodes 2 and 3 are less loaded than the other nodes. To level out commands between nodes, commands could be moved from the heaviest loaded node 4 to the least loaded nodes 1 and 2. The question is how many commands should be transferred. After a certain amount of commands has been transferred, the receiver node's QSC will rise. The transfer then has to be stopped. After how many received commands will QSC start to rise? The distributor has no way to tell as the QSC definition is individual for every node and is not sent to it at startup. If this information were available the calculations to determine the point where the QSC changes are not necessarily simple and may be costly. It is possible that during the process of transferring commands from one node to another a load imbalance occurs, incurred by new workload-aware commands being created, new executors being started or shut down, or by the workstation owner starting non-Janet processes. The transfer of commands would then have to be canceled.

6.1.5 Workload Balancing

The problems described so far make it clear that leveling out load imbalances can only be carried out as an iterative process where a single command is evicted instead of several commands in a row. "Spontaneous" changes in the cluster's workload balance can then not occur during the eviction of several commands at a time of which the agent to migrate to was determined before the last workload change. The newly changed workload balance situation can be taken into account when deciding which command on which node and of which capability to evict

next to which agent. Furthermore, it seems obvious that leveling out commands between nodes that are all already loaded with a certain amount of commands is not simple. The approach taken in Janet is therefore to use a receiver-based approach, where nodes that are about to run empty ask for commands from heavier loaded nodes. Using this approach, there is no need to perform difficult actions to level out workload between nodes. Whenever the distributor realises through received QSC change notifications that a node has fallen onto a certain low QSC level, it transfers one command from a node that is heavier loaded or has a longer capability queue size. The user is advised to define only those commands as workload-aware commands that take several seconds to execute and load a workstation's CPU heavily. If workload-aware commands are defined in such a way, a node will be fully busy even when executing a single command so that there is no urgency to keep a node's capability queue filled with many commands.

By default, QSC0 is defined as a load situation where a node has zero workload-aware commands (is idle). QSC1 is defined by default as a load situation with a single executing command. The distributor will transfer a command to a node that drops onto QSC1. The user has to be aware of this fact when adapting a node's queue size categories.

Adjusting Waiting Queues and Balancing Executing Commands

The receiver-based approach described above results in waiting queues of different queues not to be leveled out. In Figure 6-4 the number of agents per capability on the two nodes 1 and 2 is the same, but the size of the waiting queues is different. In this situation the waiting queue on node 2 is likely to run empty in less time than on node 1 where the waiting queue is longer. Once the waiting queue size on node 2 has dropped to 1, node 2 will notify the distributor that it has fallen onto QSC1. The distributor will then evict one command from node 1, provided its waiting queue is still longer than on node 2, and migrate it to node 2. This process will continue until both waiting queues have run empty. Workload distribution in this scenario is optimal since both nodes constantly remain busy. The different size of the waiting queue is not relevant since both nodes only have a single agent that can only execute one command at a time.

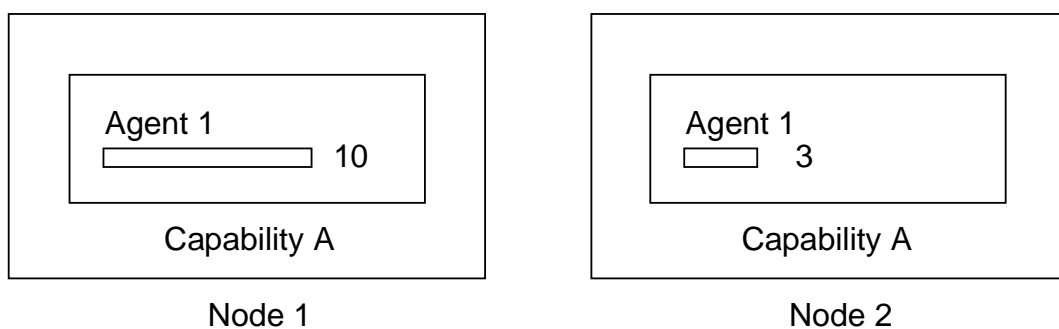


Figure 6-4: Waiting queues of different length

The situation is different when the number of executing commands on different nodes differs. In the situation in Figure 6-5 node 1 has 3 executing commands (each executing command is symbolized by a black filled rectangle) whereas node 2 has 1 executing command.

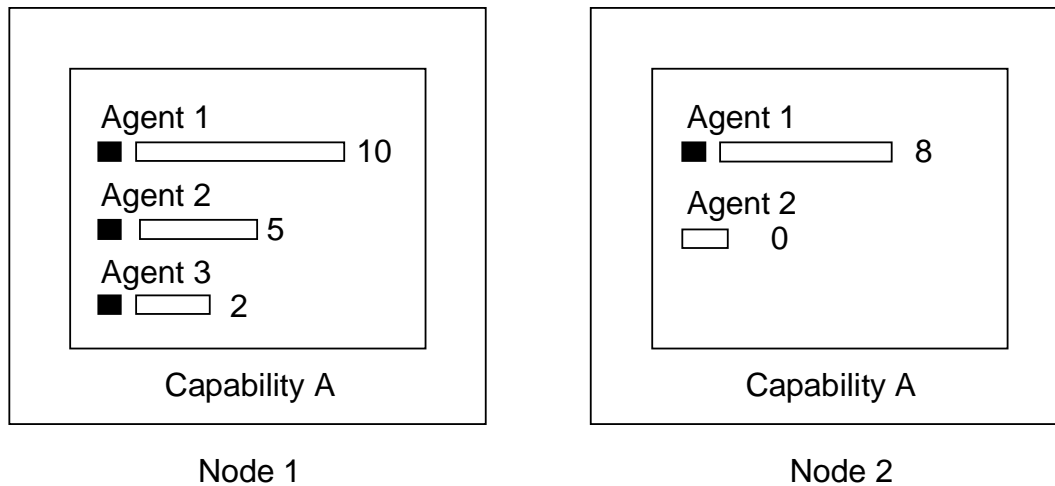


Figure 6-5: Several executing commands

The situation in Figure 6-5 shows an imbalance. A situation with an equal number of executing commands on both nodes would result in a more even workload distribution. Since agent 2 on node 2 is idle, evicting an executing command from node 1 and migrating it to node 2 would remove the imbalance. The workload balancing algorithm presented so far will not do anything about changing the imbalance in the situation in Figure 6-5. With every QSC change notification the distributor receives information about the number of executing commands per capability. It will evict commands and transfer them to nodes with fewer executing commands if possible. Several criteria have to be fulfilled for a node to receive an evicted executing command:

- The node needs to have the required capability to execute a command evicted from some other node.
- The node must not be in the process of a full eviction.
- The node needs to have a lower QSC than the overloaded node. The QSC of other nodes on the same workstation must be considered as well.
- The node needs to have an idle agent (with an empty waiting queue and no executing command) that has the required capability.

From the nodes that fulfill these criteria, the node with the lowest number of executing commands is chosen. If there are several nodes that all have the same lowest number of executing commands the one with the lowest QSC is chosen.

One Node Overloaded, All Other Nodes on QSC0

There is a special case where the receiver-initiated approach does not offer a solution, which means that it has to be handled specifically. In this special case workload balancing commands are sent to a single agent and all other nodes are on QSC0. The recipient agent's waiting queue will start to grow as a result. If all other nodes are on QSC0, no QSC change notification will be sent to the distributor to make it migrate commands from the overloaded node to the nodes

with QSC0. For that reason, whenever a node raises in QSC level above QSC1, the distributor checks whether there is any node on QSC0 to migrate a command to.

6.1.5.1 Full Eviction

When the observer realises that the CPU load of non-Janet processes on a workstation has risen above a certain threshold value all workload-aware commands are evicted from all nodes on the workstation. For the time that this load value remains above the threshold value the nodes on the workstation will not receive any commands from the distributor. When the threshold value is exceeded the observer sends a notification to the distributor. The distributor determines whether any nodes are available to which commands could be transferred to. If any node is available it first locks all nodes on the workstation with the exceeded CPU load threshold value so that they receive no evicted commands. The distributor then sends a command to all executors on the workstation, which start evicting all executing and waiting workload-aware commands. These commands are sent to the distributor, which decides for every command to which agent to pass it on to. It is possible that due to workload changes in the cluster no more nodes are available to transfer evicted commands to. The distributor then sends a cancel eviction command to the nodes evicting commands.

Canceling Full Eviction

Commands in Janet run to completion. Canceling a command that is evicting commands therefore requires the introduction of some additional mechanism. The command that carries out full eviction is the `ReduceLoadCommand`. It is sent from the distributor to the executor's system agent, which runs at highest priority. This makes sure that the `ReduceLoadCommand` is executed immediately. The `ReduceLoadCommand` places a token in the executor's application space and sends the `InterruptibleReduceLoadCommand` to the executor's special executor agent which is managed by the system arbitrator and therefore runs at a higher priority than all application schedulers managed by the application arbitrator. The scheduler thread executing the `InterruptibleReduceLoadCommand` therefore interrupts all lower prioritized scheduler threads that run application-level agents and can any time be interrupted by the thread that runs the system scheduler. This is required in case full eviction has to be canceled. In this case, the `CancelReduceLoadCommand` is sent to the executor's system agent, which will result in the lower prioritized thread running the `InterruptibleReduceLoadCommand` to be suspended and the `CancelReduceLoadCommand` to be executed immediately. The `CancelReduceLoadCommand` removes the token placed by the `ReduceLoadCommand` in the executor's application space. The `InterruptibleReduceLoadCommand` that checks for the existence of this token after every eviction of a command will no longer find it and cancel its operation.

Carrying Out Full Eviction

The purpose of full eviction is to lower the CPU load immediately caused by the processes running the Java virtual machines that run Janet. This task cannot be accomplished if the application agent's schedulers continue to run. Whenever an executing command was suspended and evicted the agent's scheduler takes in the next command from the queue and starts executing it which results in the CPU load to remain high. Clearing waiting workload-aware commands from the scheduler's queue first is no alternative solution in case the number of waiting commands is high. The approach is therefore to stop all application schedulers, suspend all executing workload balancing commands and remove all workload-aware commands from the scheduler's queue. After all workload-aware commands of a scheduler's queue have been

evicted the scheduler is started again. More precisely, instead of stopping and starting a scheduler, an additional semaphore that controls access to the scheduler's queue is closed so that the scheduler has to wait and cannot pick the next command from the queue until the semaphore is opened again. This approach is less effortful and more efficient than starting and stopping a scheduler.

6.1.5.2 Partial Eviction

The term partial eviction refers to the case where an executor node is about to run idle and a command from some other heavier loaded node is transferred to it. In case of partial eviction, contrary to full eviction where simply all workload-aware commands of a node are evicted, a command has to be determined that is transferred to the executor node that is about to run empty. The command queue for eviction is selected using the following criteria:

- Select all workstations that are available for eviction: CPU load threshold for the workstation not exceeded.
- From all available nodes select the ones that have the required capabilities, capability queue size greater than zero, and QSC2 or greater. For all matching nodes on the same workstations create the product of node queue size categories to determine the heaviest loaded workstation.
- From the heaviest loaded workstation select the node with the greatest QSC. Select the capability with the largest number of workload-aware commands. If two or more nodes have the same overall QSC, select the node with the longest capability queue.

After the distributor has selected the node and capability queue to evict a command from, it sends the command `EvictCommand`, that knows from which capability queue to pick a waiting command or suspend an executing command and to which agent to pass it on. The distributor has no knowledge which agent that has the respective capability has the longest scheduler queue. For this reason, the longest scheduler queue is determined by the `EvictCommand` at the executors site.

6.1.6 Workload Sharing

When a workload sharing command is sent to an agent, it is not immediately executed as it would be the case for commands of the Janet.CAS layer but passed on to some agent on the distributor node. This agent is called the sharing distributor agent, which serves for the sole purpose of assigning workload sharing commands to agents that are less loaded. The reason for this special agent to deal with workload sharing commands to exist is that workload sharing commands are different than workload balancing commands in the way that they remain on the node they have initially been assigned to until they have finished execution. Using a special agent for workload sharing makes sure that the distribution process of workload sharing commands does not interfere with the workload balancing tasks and can be treated as a separate part.

The sharing distributor agent has to decide which agent in the cluster a workload sharing command is to be sent to. For that purpose, the sharing distributor agent has to determine the least loaded agent in the cluster. This is the opposite of what the balancing distributor agent has to

do to accomplish workload balancing. The process of determining the least loaded agent is very similar to determining the heaviest loaded one. The steps remain essentially the same except for selecting least loaded workstations, least loaded nodes, and shortest queues instead of selecting the heaviest loaded ones, and longest queues as in case of partial eviction:

- Select all workstations that are available: CPU load threshold for the workstation not exceeded.
- From all available nodes select the ones that have the required capabilities. For all matching nodes on the same workstations create the product of node queue size categories to determine the least loaded workstation.
- From the least loaded workstation select the node with the smallest QSC. Select the capability with the smallest number of workload-aware commands. If two or more nodes have the same overall QSC, select the node with the shortest capability queue.

6.2 Detailed Design and Implementation

This section describes the detailed design and implementation of the three kinds of nodes of the executor-observer-distributor triad.

6.2.1 Executor

The executor is the part in the executor-observer-distributor triad that needs to expose an interface for the user to execute workload-aware commands. It is more illustrative to start detailed design and implementation with a presentation of the user interface.

6.2.1.1 User Interface Extensions

Agent Extensions

The `ICommand` interface for commands from Janet.CAS is extended for workload sharing commands and workload balancing commands.

```
public interface IWorkLoadSharingCommand extends ICommand
{
}

public interface IWorkLoadBalancingCommand extends ICommand
{
}
```

Figure 6-6: Interface for workload-aware commands

As Figure 6-6 shows, both interfaces, `IWorkLoadSharingCommand` and `IWorkLoadBalancingCommand`, are empty. The idea is to make use of method parameter polymorphism in Java. When a workload-aware command implementing either interface is sent to an agent proxy using the `accept` selector the respective method is invoked at runtime.

The additional `accept` method of the `IAgentProxy` are displayed in Figure 6-7 on the next page. Class `AgentProxy`, which implements the `IAgentProxy` interface, implements the additional `accept` selectors suitable for each kind of workload-aware command. The selector `accept(IWorkLoadSharingCommand command)` is implemented in such a way that the workload sharing command is sent to the distributor's sharing distributor agent. If this agent cannot be found, a warning is displayed. The command is then casted to the `ICommand` interface and sent to the receiver agent again, which results in the command to be executed by the recipient agent as a common command of the Janet.CAS layer that cannot be evicted. The other selector `accept(IWorkLoadBalancingCommand command)` is implemented in such a way that the `AgentProxy` checks whether the distributor's balancing distributor agent exists.

```
public interface IAgentProxy
{
    // already described accept selectors omitted

    public void accept(IWorkLoadSharingCommand command)
    throws RemoteException, InvalidCommandException;

    public void accept(IWorkLoadBalancingCommand command)
    throws RemoteException, InvalidCommandException;
}
```

Figure 6-7: `IAgentProxy` interface extensions

If this is not the case a warning is displayed and the workload balancing command is sent to the destination agent represented by the agent proxy. As mentioned earlier, workload balancing commands are only evicted on demand. For this reason, contrary to workload sharing commands, they are not sent to the distributor for distribution to the least loaded applicable agent.

Additional Abstract Interpreters and Interpreter Suspension

There is no additional kind of interpreter needed for the execution of workload sharing commands since this additional kind of command runs to completion in the same way as common commands in Janet.CAS. As workload balancing commands may be interrupted another kind of abstract interpreter is required that permits interruption by another party.

```
public interface IWorkLoadBalancingInterpreter extends IInterpreter
{
    public void suspend(ISuspensionHandler suspensionHandler);
    public Object clone() throws CloneNotSupportedException;
}
```

Figure 6-8: Workload balancing interpreter interface

Figure 6-8 shows the interface of the abstract workload balancing interpreter that must be implemented for the execution of user-defined workload balancing commands. The interface defines in addition to the extended `IInterpreter` interface a `suspend` method. This method is called when an executing workload balancing command has to be evicted. The suspension handler passed on as a parameter has to be signaled upon suspension of the interpreter. A user

implementing any workload balancing command must implement the suspend method. If this is not the case, Janet.ADE has no way to interrupt an executing command and the command will run to completion without being evicted. After suspension, the command carrying out the eviction will then finally evict the command. A common solution to make an interpreter suspendable is to set a flag to suspended in case the suspend method was sent. The interpreter checks the suspended flag periodically. When it detects that this flag has been set to true, it saves permanent information in the command object received through the command accessor, exits the interpreter's execute method and signals the suspension handler.

Every interpreter is instantiated by Janet.CAS once (either at startup time after parsing the node descriptor or when an application is registered programmatically) and then stored in a lookup table that is used to dispatch commands to interpreters. If a user-defined flag like some suspend flag is changed, its value will remain changed (set to suspended) for all subsequent executions of this interpreter, which would result in the execution of the interpreter to suspend immediately whenever the associated command is executed. For this reason, Janet.ADE clones a workload balancing interpreter before every execution. The `IWorkLoadBalancingInterpreter` displayed in Figure 6-8 therefore requires the clone method inherited from class `Object` to be implemented for any workload balancing command.

Additional Agent Extensions

The `IAgentProxy` interface defines for the additional `accept` methods the respective methods to install callback handlers and to send command synchronously as well. Implementing these methods does not require an extension of the existing mechanisms, which rely on a reply command to unblock a waiting thread or on invoking a callback handler. The last entry in the command envelope's history contains the origin agent path to which the command has to be sent after completion. Visiting several nodes will result in the command envelope's history to grow beyond the size of two entries. Nevertheless, the existing mechanisms will continue to work as designed since the last entry in the command envelope's history remains the agent path to the origin agent.

6.2.1.2 Command Tracking

The executor has to keep track of workload-aware commands that have been received. It needs to know into which queues they have been inserted so that it knows from which queues that have to be removed in case of eviction. The executor needs to do some bookkeeping as well, so that it knows which workload-aware commands require which capability to execute and how many commands there are per capability. This information is required to deduct the current QSC.

The `AgentDispatcher` had to be extended to signal a synchronous event whenever a command has been received. This is the only extension that had to be made in the Janet.CAS layer to implement the executor. All other functionality has been implemented with the use of another kind of command envelope that exposes the interface of a command to the existing `CommandEnvelope` while assisting the executor in doing its bookkeeping whenever a user-defined command is about to execute or has finished execution.

The `ExecutorAnchor` is the anchor object of the executor. It stores the bookkeeping information and keeps track of the current QSC. The class diagram of the `ExecutorAnchor` and related classes are displayed in Figure 6-9 on the next page.

Command Arrival

When the `AgentDispatcher` receives a command it signals the respective event, to which the `ExecutorAnchor` has subscribed to. The event handler `CommandArrivedHandler`, the `ExecutorAnchor` has registered to handle the event, notifies the `ExecutorAnchor` about the command arrival in case it is a workload-aware command and sends either the message `workloadSharingCommandArrived` or `workloadBalancingCommandArrived` as appropriate. The `ExecutorAnchor` inserts the command envelope of the command in the respective lists for waiting commands and increments the respective waiting command counters.

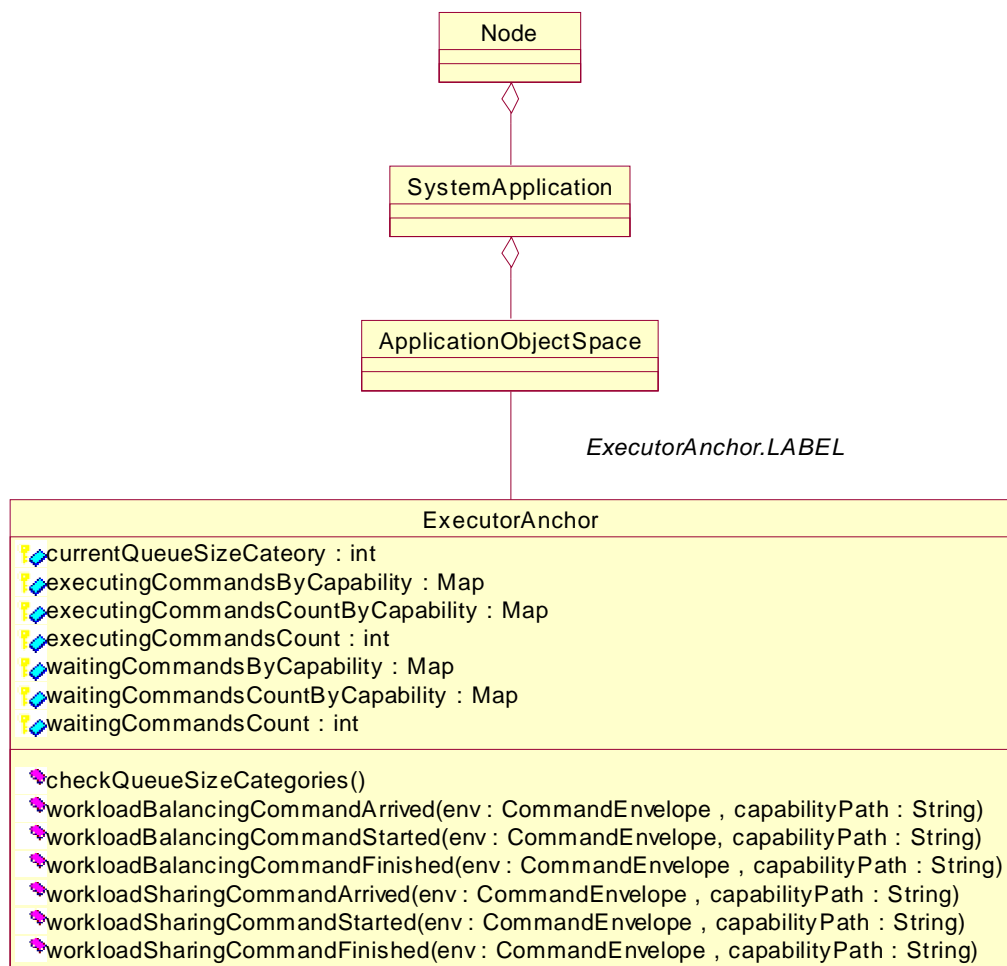


Figure 6-9: `ExecutorAnchor` class diagram

The message sequence that takes place upon command arrival is displayed in Figure 6-10 on the next page.

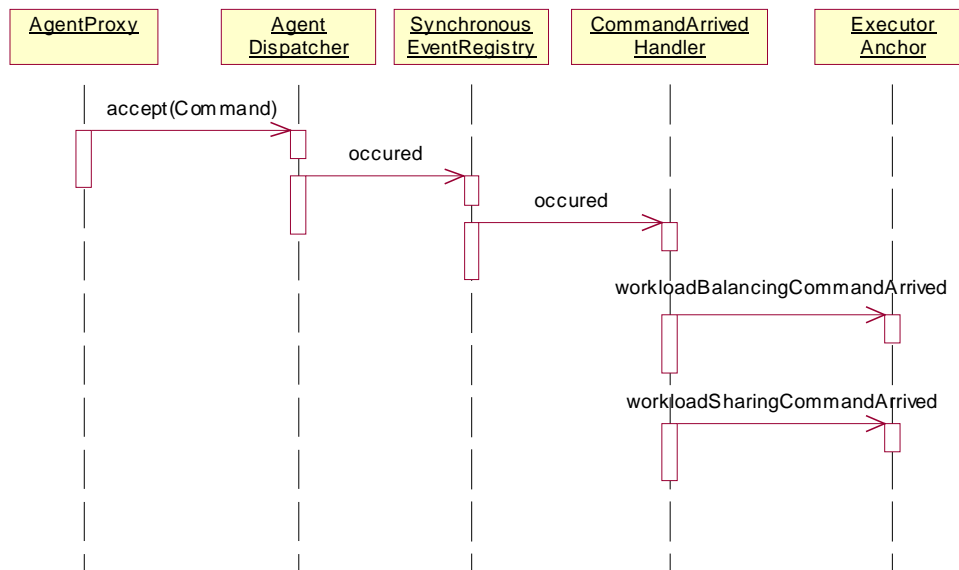


Figure 6-10: Command arrival message sequence

Executing Workload-Aware Commands

When a workload-aware command is sent to an agent proxy, the agent proxy inserts the command into a special command. There is a special command for either kind of workload-aware command: `ExecuteWorkloadSharingCommand` and `ExecuteWorkloadBalancingCommand`. An agent's command scheduler executes either special command as applicable. Both special commands notify the `ExecutorAnchor` when they are about to execute the user-defined command they contain and when they have finished executing it, which enables the `ExecutorAnchor` to keep track of the state of workload-aware commands. Whenever a workload-aware command has arrived, is about to start execution, or has finished execution, the `ExecutorAnchor`'s `checkQueueSizeCategories` method is called, which makes the `ExecutorAnchor` check whether the QSC has changed due to the command state changes. If this is the case, the distributor is notified. The message sequence to execute a workload-aware command is displayed in brief in Figure 6-11.

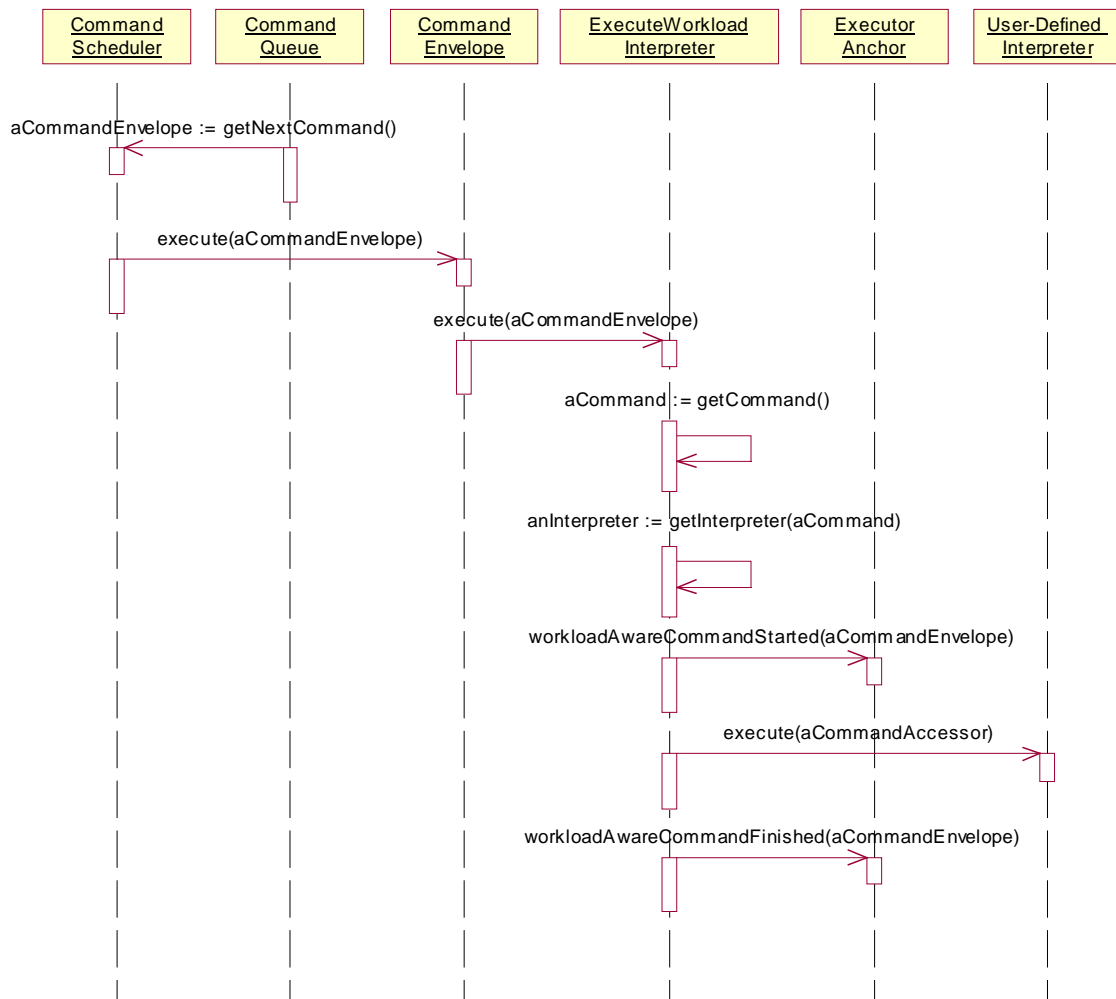


Figure 6-11: Executing an ExecuteWorkloadInterpreter

A command is always executed by an interpreter. The interpreters for the two special commands are named `ExecuteWorkloadSharingInterpreter` and `ExecuteWorkloadBalancingInterpreter`. The interpreter for the user-defined command carried by the special commands may be defined in any user-defined capability. For the interpreters of the special commands to be executable, they have to be added to the respective user-defined capability. To free the user from having to define these interpreters in a capability whenever she has to define an interpreter of a workload-aware command, the agent proxy inserts the respective required interpreter into the command envelope, carrying the user-defined command, before the command is sent to the destination agent. If the command envelope's attribute that holds the interpreter to be used for the execution of a command is not set to null, the scheduler will not look up the interpreter for the command dynamically but use the already specified interpreter. Preventing dynamic interpreter lookup is a compromise for the sake of making the definition of workload-aware command interpreters as simple as possible for the user. There is no way for the user to prevent dynamic interpreter lookup since the respective inner node classes provide no such interface.

6.2.1.3 Executor Descriptor

To define an executor node a node's system application has to be extended. Commands that need to be executed by the supreme agent, that uses the supreme scheduler, have to be defined as part of system applications' CORE capability. For the executor this is the case for the commands `ReduceLoadInterpreter` and `CancelReduceLoadInterpreter`. The `ReduceLoadInterpreter` must be executed immediately to make sure partial or full eviction is carried out without delay. The `CancelReduceLoadInterpreter` to be able to cancel a full eviction must be executed with highest priority as well.

```
<systemApplication>
  <capabilities>
    <capability name="CORE">
      <interpreters>
        <!-- interpreters defined by Janet.CAS omitted -->
        <interpreter>
          org.objectscape.janet.ade.executor.interpreters.ReduceLoadInterpreter
        </interpreter>
        <interpreter>
          org.objectscape.janet.ade.executor.interpreters.CancelReduceLoadInterpreter
        </interpreter>
      </interpreters>
    </capability>
  </capabilities>
</systemApplication>
```

Figure 6-12: System application's additional interpreters of the CORE capability

Figure 6-12 displays the system application's CORE capability extended with the respective interpreters. The remaining commands executed by the executor's agents need to run with system priority. For that reason they are defined as part of an additional system capability so that they will be executed by system agents that use system schedulers. The additional System capability is named `ADE_EXECUTOR`. Its definition is displayed in Figure 6-13.

```
<systemApplication>
  <capabilities>
    <capability name="CORE">
      <!-- interpreters omitted for brevity -->
    </capability>
    <capability name="ADE_EXECUTOR" descriptor="executorDescriptor.xml">
      <agents>
        <agent name="EXECUTOR" executeWhenStarted=
          "org.objectscape.janet.ade.commands.StartCommand" />
      </agents>
      <interpreters>
        <interpreter>
          org.objectscape.janet.ade.executor.interpreters.StartInterpreter
        </interpreter>
        <interpreter>
          org.objectscape.janet.ade.executor.interpreters.
```

```

        InterruptibleReduceLoadInterpreter
    </interpreter>
</interpreters>
</capability>
</capabilities>
</systemApplication>

```

Figure 6-13: Executor's main system capability

The ADE_EXECUTOR capability defines the EXECUTOR agent, which carries out full and partial eviction using the InterruptibleReduceLoadInterpreter. The executor's QSCs are defined in the descriptor of the ADE_EXECUTOR capability, which in Figure 6-13 is stored in the file executorDescriptor.xml.

```

<executor>
  <observerInterval delay="2000" period="2000" />
  <queueSizeCategories>
    <queueSizeCategory name="0" maxWaitingCommands="0"
      maxExecutingCommands="0" />
    <queueSizeCategory name="1" maxWaitingCommands="0"
      maxExecutingCommands="1" />
    <queueSizeCategory name="2" maxWaitingCommands="3"
      maxExecutingCommands="1" />
    <queueSizeCategory name="3" maxWaitingCommands="20"
      maxExecutingCommands="1" />
    <queueSizeCategory name="4" maxWaitingCommands="10"
      maxExecutingCommands="3" />
  </queueSizeCategories>
</executor>

```

Figure 6-14: Executor capability descriptor defining queue size categories

A definition of QSCs is shown in Figure 6-14. QSCs are numbered in ascending order; every QSC defines a maxWaitingCommands and a maxExecutingCommands attribute. When determining the current QSC the executor iterates over the QSC definitions in descending order and checks whether both criteria are met. If it is met the current iterator's QSC name is returned. Otherwise, the iterator moves on to next lower QSC.

6.2.1.4 Command Eviction

During full eviction it is not necessary to identify specific commands in specific queues since simply all waiting and executing commands are moved to the distributor. In case of partial eviction, the evicting command needs to know precisely from which queue of which capability to pick a command for eviction. Partial eviction is therefore the more demanding eviction process, which is covered in more detail than full eviction.

After the ReduceLoadCommand has been accepted by the supreme agent, it sends the InterruptibleReduceLoadCommand to the EXECUTOR agent. The EXECUTOR agent invokes the InterruptibleReduceLoadInterpreter, which retrieves an eviction assignment from its com-

mand. The eviction assignment was inserted by the distributor into the `ReduceLoadCommand`. It defines whether a waiting or executing command has to be evicted, from which capability, and to which agent to transfer the evicted command.

Evicting A Waiting Command

When evicting a waiting command there is no need to suspend it since execution has not started. The waiting command is removed from the queue it is waiting in and transferred to the destination agent. If the waiting queue has run empty after the eviction assignment was sent to the executor, the `InterruptibleReduceLoadInterpreter` has nothing to do and terminates. Otherwise, the agent with the capability specified by the eviction assignment with the longest queue has to be retrieved. All agents with the required capability are obtained from the node's local node image. Since the `IAgentProxy` defines a `size` method that returns the agent's current queue size, the agent with the longest waiting queue can be detected effortlessly. The `ExecutorAnchor` stores waiting commands by their command envelope. The command envelope knows the queue its command is waiting in. The command envelope can therefore be asked to remove its command from its waiting queue. After removal the command is sent to the destination agent. If any monitor node is present in the cluster, it receives a notification that a command has been evicted on a specific node so that the user can follow command movements of commands by watching command counts being incremented and decremented.

After the eviction a QSC change notification is sent to the distributor with the current queue size information even if the QSC has not changed so that the distributor has an accurate picture that includes the recent load adjustment.

Evicting An Executing Command

When an executing command is evicted the eviction process remains essentially the same. The only difference is that the executing command has to be asked to suspend execution. The `EXECUTOR` agent does so by passing on a suspension handler. The `EXECUTOR` agent does therefore not remain waiting till the command has finally been suspended. Only after the suspension handler has been signaled the `EXECUTOR` agent moves the suspended command on to the destination agent. Any monitor node, if present in the cluster, is notified as well. The mechanism to evict a command is described in section 6.3.1 on the basis of the driving `ADÉ.Fibonacci` application. In case several executing agents with required capability exist, the command executed by the agent with the longest waiting queue is chosen for eviction.

6.2.2 Observer

The observer observes a workstation's CPU load periodically to detect whether the workstation has been retracted for purposes other than executing Janet commands. The observer capability is named `ADE_LOAD_OBSERVER` and is defined as a capability of the system application. The observer's agent named `OBSERVER` therefore runs with second highest priority, which makes sure that the observer agent can only be interrupted by the supreme agent. The observer's capability definition is displayed in Figure 6-15.

```
<systemApplication>
  <capabilities>
    <!-- CORE capability omitted for brevity -->
    <capability name="CORE">
```

```

<capability name="ADE_LOAD_OBSERVER" descriptor="observerDescriptor.xml">
  <agents>
    <agent name="OBSERVER" executeWhenStarted=
      "org.objectscape.janet.ade.commands.StartCommand" />
  </agents>
  <interpreters>
    <interpreter>
      org.objectscape.janet.ade.observer.simulation.interpreters.StartInterpreter
    </interpreter>
    <interpreter>
      org.objectscape.janet.ade.observer.simulation.interpreters.
      ObserveWorkstationLoadInterpreter
    </interpreter>
  </interpreters>
</capability>
</capabilities>
</systemApplication>

```

Figure 6-15: Simplified observer descriptor definition

The user can change several settings that are defined in the capability descriptor file of the observer's capability (in the descriptor in Figure 6-15 named `observerDescriptor.xml`). The capability descriptor is displayed in Figure 6-16.

```

<observer>
  <interval periodInMillis="2000" />
  <loadChange significantWhenExceedingPercentage="10"/>
</observer>

```

Figure 6-16: Observer capability descriptor

The user can change the interval period after which the observer checks the current CPU load (`periodInMillis`) and how strong a CPU change needs to be to be considered significant (`significantWhenExceedingPercentage`).

6.2.2.1 Simulated Observer

The `ADE_LOAD_OBSERVER` capability defines two interpreters. The `StartInterpreter` is executed after the node has finished the start-up process, which starts the observer's graphical user interface. The `ObserveWorkstationLoadInterpreter` reads the CPU load periodically to detect load changes and notifies the distributor in case of significant change. Both interpreters are defined in the `org.objectscape.janet.ade.observer.simulation` package since the observer in its current state is a simulation that does not read the CPU load from the host operating system but reads a user-defined value (CPU-LOAD All Others slider in Figure 6-17). The use of a simulation makes it possible to make the system react to load changes that are difficult to create in an unsimulated environment. Using a simulation is a necessity to create an environment where implementation and test can be carried out in a simple and straightforward way.

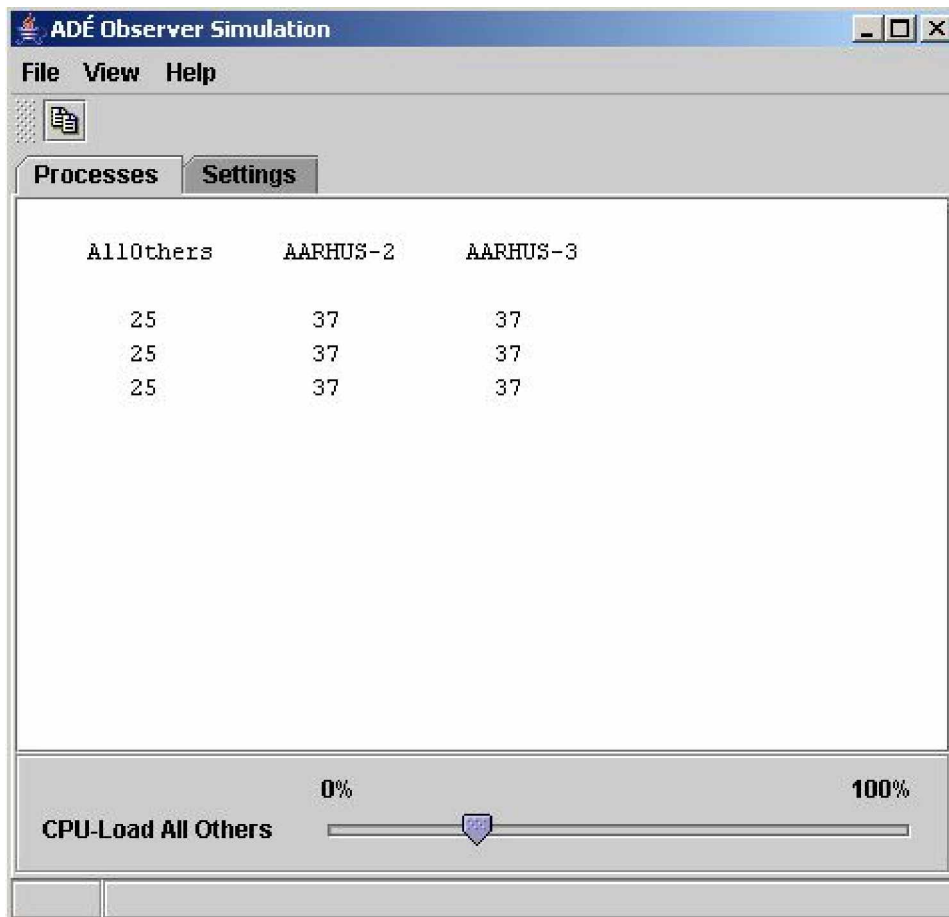


Figure 6-17: Observer's main view processes tab

The observer's main view processes tab displayed in Figure 6-17 displays the value of the simulated CPU load (`AllOthers`) and the CPU load caused by the two executors hosted by the node `AARHUS-2` and `AARHUS-3`, which is derived from the executors QSC. Every executor writes the current QSC into a file following the naming pattern `nodeName.Executor.info`. The simulated observer reads in the current QSC from file for every executor and displays it in the processes tab. In order to implement and test in simulated mode using a single machine, the observer's main view settings tab, displayed in Figure 6-18, allows the user to specify which nodes the observer should assume to be on the same workstation. Clicking the "Look for local Janet nodes" button forces the simulated observer to look again for all `nodeName.Executor.info` files in case new executor nodes have been started up meanwhile.

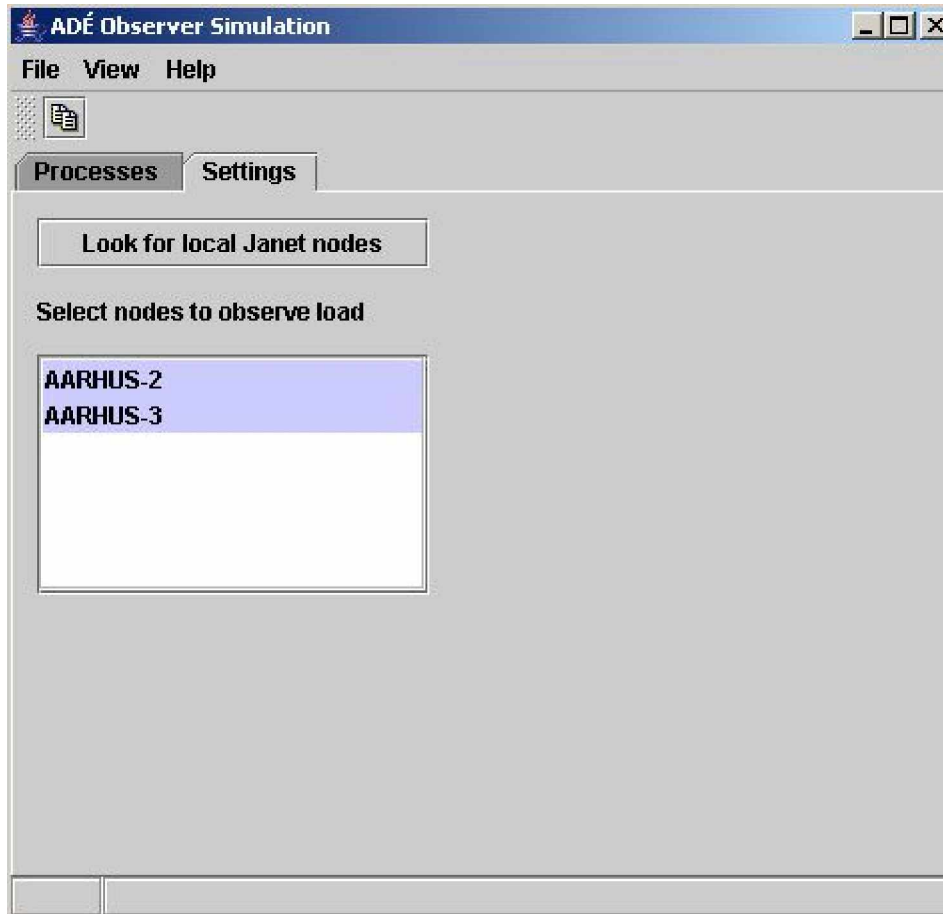


Figure 6-18: Observer's main view settings tab

6.2.3 Distributor

The distributor is the central authority that decides whether commands have to be evicted to level out workload imbalances and decides where commands are evicted and where they are transferred.

The explanation of the distributor's descriptor is a good starting point to describe the distributor's constitution. As mentioned in the conceptual description of the distributor three agents are defined that run as system schedulers with second highest priority. All agents run with the same priority to make sure that one cannot cause starvation of the others. If the `BALANCING_DISTRIBUTOR` agent, that makes the eviction decisions, ran at higher priority than the others, the `LOAD_ADMINISTRATOR` agent might be blocked for a while and could not update the distributor's load image. This would result in the `BALANCING_DISTRIBUTOR` agent making eviction decisions not based on most recent information. Similarly, if the `LOAD_ADMINISTRATOR` agent ran at higher priority, the `BALANCING_DISTRIBUTOR` agent could be blocked not being able to make eviction decision for the distributor to be responsive to workload changes. To make sure that the distributor is not obstructed it is recommended not to define additional system capabilities for a distributor node. If the `LOAD_ADMINISTRATOR` agent

seems to be much more busy than the other agents, adjusting the nodes' QSCs should be considered.

```

<systemApplication>
  <capabilities>
    <!-- CORE capability omitted for brevity -->
    <capability name="ADE_LOAD_DISTRIBUTOR" descriptor="distributorDescriptor.xml">
      <agents>
        <agent name="BALANCING_DISTRIBUTOR"
          executeWhenStarted=
            "org.objectscape.janet.ade.commands.StartCommand" />
        <agent name="SHARING_DISTRIBUTOR" />
        <agent name="LOAD_ADMINISTRATOR" />
      </agents>
      <interpreters>
        <interpreter>
          org.objectscape.janet.ade.distributor.interpreters.
            StartInterpreter
        </interpreter>
        <interpreter>
          org.objectscape.janet.ade.distributor.interpreters.
            WorkloadDistributionInterpreter
        </interpreter>
        <interpreter>
          org.objectscape.janet.ade.distributor.interpreters.
            ProcessQSCChangedInterpreter
        </interpreter>
        <interpreter>
          org.objectscape.janet.ade.distributor.interpreters.
            ProcessCPULoadChangedInterpreter
        </interpreter>
        <interpreter>
          org.objectscape.janet.ade.distributor.interpreters.
            NotifyQSCChangedInterpreter
        </interpreter>
        <interpreter>
          org.objectscape.janet.ade.distributor.interpreters.
            NotifyCPULoadChangedInterpreter
        </interpreter>
        <interpreter>
          org.objectscape.janet.ade.distributor.interpreters.
            ObserverStateChangedInterpreter
        </interpreter>
      </interpreters>
    </capability>
  </capabilities>
</systemApplication>

```

Figure 6-19: Distributor descriptor

The distributor's description displayed in Figure 6-19 defines several interpreters. The `WorkloadDistributionInterpreter` carries out initial placement of a workload sharing

command and places workload-aware commands received through full eviction. The two notification commands `NotifyQSCChangedCommand` and `NotifyCPULoadChangedCommand` notify the distributor about changed conditions concerning the QSC of an executor in the former case and the changed CPU load detected by an observer in the latter case. Both notification commands update the distributor's internal cluster load image. If they detect an imbalance, they create the respective commands that take action to remove the imbalance and send them to the `BALANCING_DISTRIBUTOR` agent: `ProcessQSCChangedCommand` and `ProcessCPULoadChangedCommand`. Only sending these commands in case an imbalance was detected reduces the `BALANCING_DISTRIBUTOR` agent's load and improves responsiveness. The `ProcessQSCChangedInterpreter` tries to level out imbalances in the number of executing commands and feeds an executor with commands in case it runs into QSC1. The `ProcessCPULoadChangedCommand` carries out full eviction or cancels it. `ObserverStateChangedInterpreter` is used to inform the distributor that an executor has been connected to an observer or has been disconnected from an observer.

Displaying Cluster Load Information

The distributor displays in values tab of its node main view the current workload information it has. This information can be used to reproduce eviction decisions.

6.3 *Driving Application: Fibonacci Numbers Revisited*

The Fibonacci sample application developed as a driving application for Janet.CAS is extended with new functionality from Janet.ADE. The driving application is called `ADE.Fibonacci`. Firstly, A Fibonacci workload sharing command is implemented that is transparently placed at a lightly loaded node by the distributor. Secondly, a Fibonacci workload balancing command is implemented that will be evicted from the agent it was initially sent to if required to balance load.

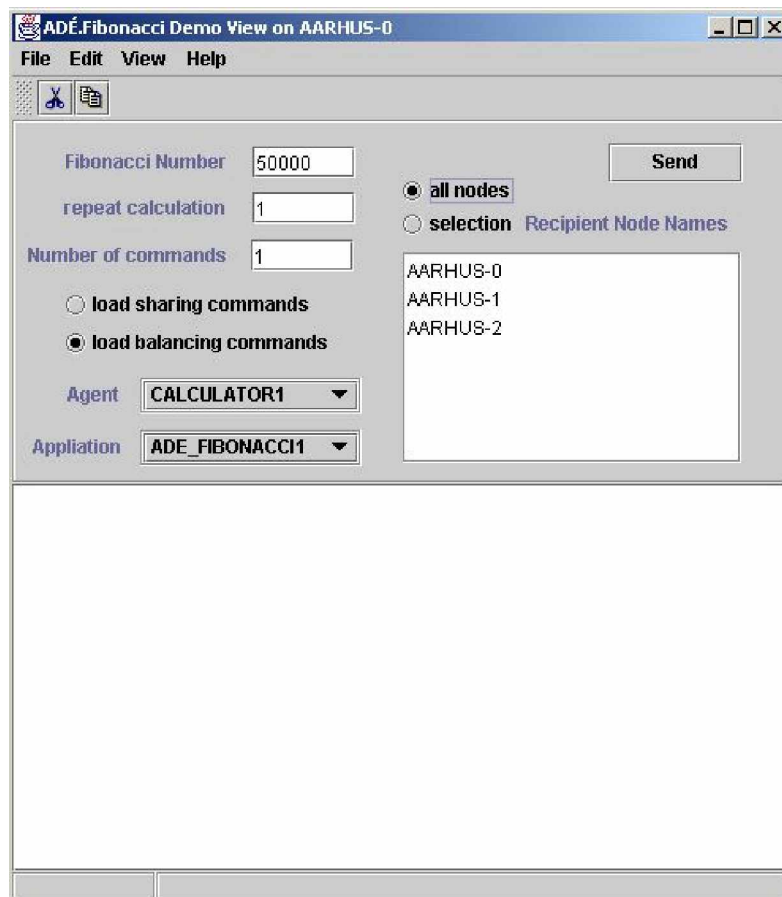


Figure 6-20: ADE.Fibonacci test console

A console is provided to the user with which the user can create various workload scenarios and observe how the system responds in order to level out workload imbalances. The console is displayed in Figure 6-20 on the previous page. The user can define what Fibonacci number has to be calculated by the commands. In Figure 6-20 the 50,000th Fibonacci number will be calculated once, 1 workload balancing command is sent to all nodes in the cluster that have the application ADE_FIBONACCI1. A Fibonacci command can be told to repeat calculation of the Fibonacci number several times by setting the value in the entry field `repeat calculation` accordingly. By entering a value greater 1 the user can make sure that the node receiving the command will be fully loaded for many seconds. This makes it often easier to create specific workload scenarios. It is possible to send commands to all nodes in the cluster with the application selected in the Application combo box or to specific nodes that were selected in the Recipient Agent Names list box. The Application combo box contains the names of three ADE.Fibonacci applications named ADE_FIBONACCI1, ADE_FIBONACCI2, and ADE_FIBONACCI3 defined in the ADE.Fibonacci application description. All these applications are defined in the same way as ADE.Fibonacci applications. By having more than one application it is possible to create scenarios where the system is under load by different applications (these applications have the same definition but different names and are therefore regarded by the distributor as being different applications). The Agent combo box allows specifying to which agent commands are sent. Since every ADE.Fibonacci application defines more than 1 agent it is possible to create scenarios, where several agents with the same capability are executing commands in-

stead of only 1. The `Recipient Node Name` lets the user specify the node of where the recipient agent resides. The number of commands that are sent at once can be specified in the `Number of commands` entry field. For example, when using this feature to send more than one workload sharing command the user can observe on which nodes the distributor places them.

Janet.CAS offers a monitor application that serves as a cluster-wide visible blackboard. Nodes, by default, send status information to the monitor in case it is present in the cluster. Using the monitor the user can follow how workload-aware commands travel. More detailed information is given in section 8.4 “Running the ADÉ.Fibonacci Sample”.

6.3.1 Defining Commands and Interpreters

As for every Janet application command-interpreter pairs have to be defined. For the ADÉ.Fibonacci driving application a Fibonacci workload sharing command and a Fibonacci workload balancing command have to be implemented.

6.3.1.1 Defining Commands

Implementing the commands only consists of defining the required attribute to store input data. A class diagram is displayed in. The full implementation is not displayed here as of little interest. The commands can be found in the package `org.objectscope.janet.ade.demo.fibonacci` named `FibonacciSharingCommand` and `FibonacciBalancingCommand`.

Figure 6-21 on the next page shows the class hierarchy of the Fibonacci commands defined for ADÉ.Fibonacci. Common attributes of both kinds of Fibonacci workload command classes are defined in the abstract class `FibonacciCommand`. Class `FibonacciBalancingCommand` has additional attributes in order to save the current execution state at the time the associated interpreter was suspended for eviction.

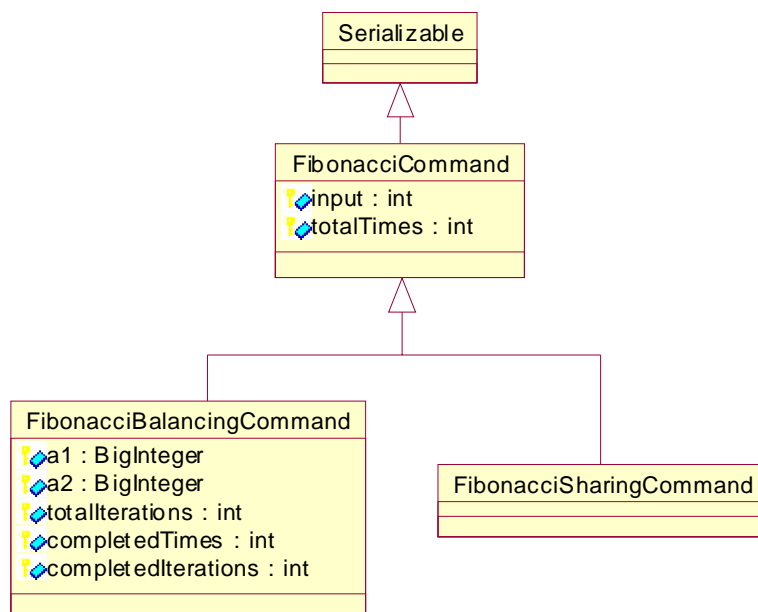
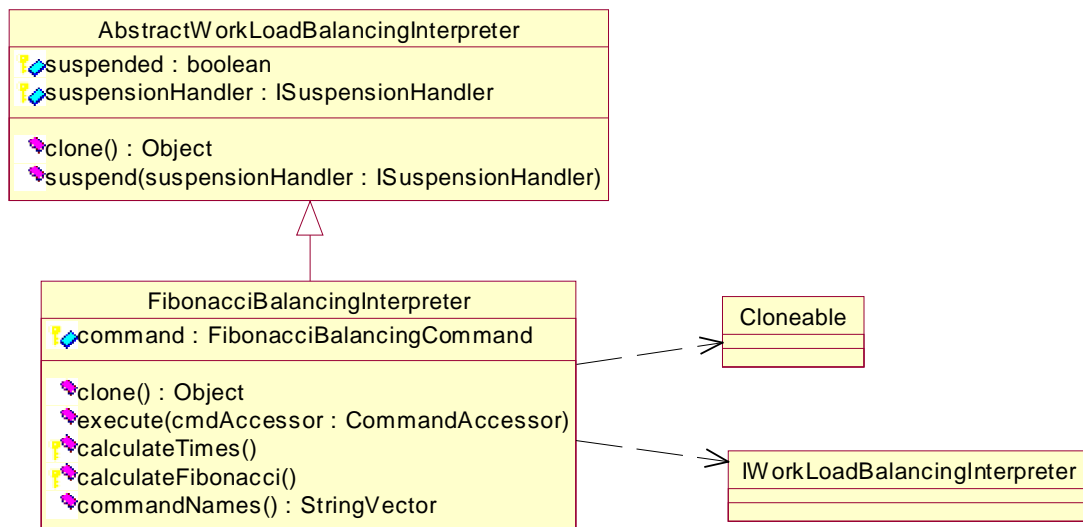


Figure 6-21: Fibonacci workload commands class hierarchy

6.3.1.2 Defining Interpreters

The `FibonacciSharingInterpreter` is defined in the same way as the `FibonacciInterpreter` developed for `CAS.Fibonacci`, but its `commandNames` method specifies the qualified name of the `FibonacciSharingCommand`. For the `FibonacciBalancingInterpreter` an additional effort is needed to make it suspendable.

Figure 6-22: Class hierarchy of the `FibonacciBalancingInterpreter`

Janet.ADE defines an abstract workload balancing interpreter named `AbstractWorkLoadBalancingInterpreter` that is subclassed by the `FibonacciBalancingInterpreter`. The abstract class defines common attributes needed for every workload balancing command. If it is not used it is important to define the `suspend` attribute in the user's workload interpreter class as volatile. If it is not declared volatile the Java compiler might create a cache to store the value of the attribute in, which results in some other thread not to see the changed value but still the cached one. Alternatively, the `suspend` attribute can be protected by a synchronized block, which has a higher cost.

The `FibonacciBalancingInterpreter` class must implement the methods defined in the `IWorkLoadBalancingInterpreter` and be cloneable. Its class hierarchy is displayed in Figure 6-22. The `execute` method it needs to define is in this case simple as can be seen in Figure 6-23.

```

public void execute(CommandAccessor cmdAccessor)
{
    if (cmdAccessor.getCommand() instanceof FibonacciBalancingCommand)
    {
        command = (FibonacciBalancingCommand) cmdAccessor.getCommand();
        calculateTimes();
        handleResult(cmdAccessor);
    }
}

```

```
}

```

Figure 6-23: FibonacciBalancingInterpreter execute method

It stores its command in the `command` attribute and starts the calculation. The calculation methods are of more interest.

```

protected void calculateTimes()
{
1      int completedTimes = command.getCompletedTimes();
2      int totalTimes = command.getTotalTimes();
3
4      for (int i = completedTimes; i < totalTimes; i++) {
5          if(!suspended) {
6              calculateFibonacci();
7              if(!suspended) {
8                  command.setCompletedTimes(i + 1);
9                  command.setCompletedIterations(0);
10             }
11         }
12         else
13         {
14             break;
15         }
16     }
17 }

```

Figure 6-24: FibonacciBalancingInterpreter calculateTimes method

The `execute` method calls the `calculateTimes` method displayed in Figure 6-24. If the user specified in the ADÉ.Fibonacci console that the Fibonacci number should be calculated `n` times, the `calculateTimes` method calls the `calculateFibonacci` method `n` times as long as the interpreter has not been suspended.

```

protected BigInteger calculateFibonacci()
{
1      int nthFibonacciNumber = command.getTotalIterations();
2      if (nthFibonacciNumber < 3)
3          return BigInteger.ZERO;
4
5      BigInteger a1 = command.getA1();
6      BigInteger a2 = command.getA2();
7      BigInteger an = BigInteger.ZERO;
8      int completedIterations = command.getCompletedIterations();
9
10     for (int i = completedIterations; i < nthFibonacciNumber; i++)

```

```

11      {
12          if(suspended)
13          {
14              command.setA1(a1);
15              command.setA2(a2);
16              break;
17          }
18          else
19          {
20              an = a1.add(a2);
21              a1 = a2;
22              a2 = an;
23              command.setCompletedIterations(i + 1);
24          }
25      }
26      return an;

```

Figure 6-25: FibonacciBalancingInterpreter calculateFibonacci method

The `calculateFibonacci` method displayed in Figure 6-25 calculates the Fibonacci number itself. It checks after every iteration whether the interpreter has been asked to suspend execution (line 12). If so, the previous and current Fibonacci numbers, `a1` and `a2`, are stored in the command object (line 14 + 15) and execution of the method terminates, which causes the `calculateTimes` method to store its context-dependent information in the command as well (line 11 + 12 in Figure 6-24) before terminating. When a command is about to be evicted its `suspend` method is called first. The `suspend` method for the `FibonacciBalancingInterpreter` is inherited from class `AbstractWorkLoadBalancingInterpreter`. It stores the `suspensionHandler` passed on as a parameter in the class' `suspensionHandler` attribute and sets the `suspend` attribute to true, which results in the calculation of the Fibonacci number to terminate and the `handleResult` method to be called (see Figure 6-23).

```

protected void handleResult(CommandAccessor cmdAccessor)
{
1      if(suspended)
2      {
3          suspensionHandler.suspended();
4      }
5      else
6      {
7          cmdAccessor.setReply(command.getAn());
      }
}

```

Figure 6-26: Notify the suspension handler about the suspension

The implementation of the `handleResult` method is shown in Figure 6-26. If the calculation has finished without the interpreter having been suspended before, the thread of execution jumps to line 7, where the calculated Fibonacci number is sent back to the agent sending the command (try-catch-block omitted for brevity). The calculation result will be handed over to the callback handler, installed by the agent when sending the command, and the handler will be invoked. If the interpreter has been suspended, the `suspensionHandler` passed on as a method parameter of the `suspend` method is notified that suspension has finished (line 3). Thereafter, the `suspensionHandler` will evict the command.

6.3.2 Assembling The Application

After commands and interpreters have been defined the application definition of ADÉ.Fibonacci has to be assembled and added to the executor's node descriptor (which has been presented in section 6.2.1.3).

```
<application name="ADE_FIBONACCI 1">
  <capabilities>
    <capability name="CORE">
      <agents>
        <agent name="CALCULATOR1" executeWhenStarted=
          "org.objectscape.janet.ade.demo.fibonacci.StartViewCommand" />
      </agents>
    </capability>
  </capabilities>
  <interpreters>
    <interpreter>
      org.objectscape.janet.ade.demo.fibonacci.StartViewInterpreter
    </interpreter>
    <interpreter>
      org.objectscape.janet.ade.demo.fibonacci.FibonacciBalancingInterpreter
    </interpreter>
    <interpreter>
      org.objectscape.janet.ade.demo.fibonacci.FibonacciSharingInterpreter
    </interpreter>
  </interpreters>
</application>
```

Figure 6-27: ADÉ.Fibonacci application description

The definition of the ADÉ.Fibonacci application, that has to be added to the executor's node descriptor, is displayed in Figure 6-27. For ADÉ.Fibonacci 1 agent is defined named CALCULATOR1. After the application has been registered the StartViewCommand specified by the agent's executeWhenStarted attribute is sent to the agent, which executes the StartViewInterpreter. The StartViewInterpreter displays the test console on the screen. The interpreters FibonacciBalancingInterpreter and FibonacciSharingInterpreter have to be added to the capability for the capability's agent to be able to respond to Fibonacci workload sharing and workload balancing commands.

Defining Multiple ADÉ.Fibonacci Applications Per Executor

The executors' node descriptor for the executors as setup in section 8.4 contains three definitions of the ADÉ.Fibonacci application. The application names of these three definitions are ADE_FIBONACCI1, ADE_FIBONACCI2, and ADE_FIBONACCI3 (selectable from the test console Application combo box). There are three different executor node descriptors. The first executor node descriptor defines 1 agent named CALCULATOR1 for each application. The second executor node descriptor defines 2 agent named CALCULATOR1 and CALCULATOR2 for each application. Finally, the third executor node descriptor defines 3 agents named CALCULATOR1, CALCULATOR2, and CALCULATOR3. Using these three different executor node descriptors executors can be started up with a different number of agents per capability (to send commands

to which one can be specified in the test console's `Agent` combo box). With this setup workload scenarios can be created where there is a different number of agents per capability and node: on one executor node three agents of the same capability might be executing a command each, whereas on a different executor node for example only 1 agent might be executing a command at the same time.

7 Extensions for Janet

This chapter describes several ideas for further extensions for Janet. Potential extensions are Janet.DIC to distribute code in Janet (dynamically supplying nodes with applications and capabilities through the network) and Janet.VOS (evicting large objects to nodes with more available memory).

7.1 *Janet.DIC*

Janet.DIC (**D**istributed **C**ode) is the idea of an extension to Janet that allows nodes to be equipped with applications their capabilities remotely through the network. Transferring a capability includes the transfer of the interpreters they define together with the interpreters' byte code that is contained in the interpreters' class files. At the target node, the byte code has to be loaded into program space with the use of the Java class loader.

Using Janet.DIC nodes could be configured dynamically, which would be helpful when configuring nodes at distant locations that are hard to reach. There would be no need to shut down and start up a node to read in additional interpreter class files previously not existing on the node's workstation. The number of interpreters that have to be present on a node's workstation can be minimized. Interpreters could be installed dynamically on a node in case of an agent with a capability previously not present at that node has to be sent to that node. With such a feature it would be possible to use Janet as an agent platform to implement mobile agents or to implement a platform that offers similar functionality as a mobile agent platform without having to accept some of the drawbacks of mobile agents.

7.1.1 Mobile Agents vs. Mobile Applications

Because of Java's ability to distribute code many mobile agent platforms have been developed in Java. Mobile agents are one way to migrate an agent from one platform to another. The advantage of mobile agents is that they can also be started on a minimally equipped destination system as they carry the executable code they need with them. They are therefore not dependent on heavyweight agent platforms and can migrate to low footprint devices such as handhelds, PDAs, mobile phones, etc.

Instead of using mobile agents the approach in Janet.DIC could be to use Janet.CAS as a lightweight agent platform with a small footprint and send a command with applications and their capabilities to a node where these applications are needed. When interpreted by the recipient the command installs the applications and asks the system to start their associated agents. Mobile agents are a special kind of agents that only have an advantage in a special setting - and are therefore not universal - where the destination machine does contain the mobile agent's executable code and environment.

Janet.CAS, on the contrary, is a general-purpose agent platform that together with Janet.DIC can basically offer the same functionality as mobile agents. However, Janet.CAS and Janet.DIC would have to be installed and running on every node, which would cost a certain amount of disk and heap space. A workstation that can accept a mobile agent needs to have a minimal kernel of some mobile agent platform installed so that a mobile agent after arrival can

be started. Since Janet.CAS is a lightweight agent platform the space required by Janet.CAS is probably not higher compared to the minimal mobile agent platform kernel of a real-world mobile agent platform such as SeMoA. Additionally, when an agent in Janet has to migrate between nodes in a cluster it does not need to carry its entire executable code with it as a mobile agent. If it already has visited a node earlier the application it needs is already installed and needs not be sent to that node another time. Many of the security problems of mobile agents arise from agent and code traveling together. It might be worth investigating whether several security problems of mobile agents could be solved easier or could be solved at all if code and agent travel separately.

7.1.2 Security

Security is an important concern with regard to distributed code. The standard approach in Java is to use signed applets. Similarly, for Janet.DIC some kind of signed applications could be introduced to authenticate origin and destination of distributed code.

7.2 Janet.VOS

Distributed operating systems such as MOSIX support memory ushering: if a node runs low on memory, objects can be evicted and be moved to some other node. The idea of Janet.VOS (Virtual Object Space) is to implement memory ushering for Janet as well. It has to be taken into account that a disk drive, used by an operating system's virtual memory system, can swap out objects much faster than it is possible to transfer them through a network – even when the network is a high-speed network. There seems to be no way that Janet.VOS could compete directly with a disk drive as a swap medium. Janet.VOS would therefore need to have a concept that offers advantages an operating system's virtual memory system could not provide. For example, when available memory becomes low, instead of evicting commands, it could evict agents together with all the permanent objects they need to other nodes. A system offering such functionality could react in a very flexible way to changing amounts of available memory. After an agent was evicted, it does not have to be transferred back to the node it was evicted from in case it needs to be accessed in the way that memory pages need to be read in from disk if accessed after having been swapped out. Since agents in Janet are distributed from the beginning, an evicted agent could remain on the node it was migrated to when accessed after eviction. A fair amount of the functionality to evict agents would be provided by Janet.DIC. Janet.VOS, beyond the functionality provided by Janet.DIC, would have to observe the amount of available memory on every node, would have to decide which node to evict an agent to, and initiate the agent eviction process. An agent could also be evicted by some user-defined criteria, other than a node's available memory running low. The considerations described in this section concerning Janet.VOS are not final and more of a brainstorming nature. They are described to display potential extensions of Janet.

8 Running Sample Applications

The sample applications are included on the accompanying CD-ROM. Documentation, sources, binaries, and samples can also be downloaded from www.oliverplohmann.de/janet. In case problems occur downloading the files you can send e-mail to oliver@plohmann.com. The sample applications can be started using the batch files in the samples directory.

8.1 Overview: Starting Up Nodes

Sample Startup Sequence

To start a Janet node a statement with the following pattern has to be executed from a command prompt:

```
java org.objectscope.janet.cas.node.NodeStarter
<nodeDescriptorPathFileName.xml>
```

For every sample application a set of batch files are included that are defined as required. If the path to the bin directory of the installed Java SDK is not included in the system path, it has to be included in every batch file explicitly.

Adapting Node Descriptor To Use Local Workstation Names

All remaining parameters are defined in the node descriptor. Each node descriptor in its registry statement must specify the name of the workstation the central resides on. Before starting any sample application the user must fill in the workstation name of the central in the node descriptor. The registry statement looks like this:

```
<registry
  centralHostname="myWorkstation"
  centralPort="1099"
  localPort="1099" />
```

The user needs to change the `centralHostname` to the name of the workstation she wants to start the central on. *This has to be done for every node descriptor file that is used in the sample applications.* All node descriptor files that belong to a sample application are located in the subdirectory `nodes` of the sample application directory.

The `centralPort` attribute specifies the port used by the registry started by the central on the workstation the central resides. It does not need to be specified if it is not different from the default RMI registry port. The `localPort` attribute specifies the port used by the registry of the node that is started up. It does not need to be specified if it is not different from the default RMI registry port.

If the node descriptors are located on a shared directory, they only need to be adapted to using local workstation names once. Otherwise, they have to be adapted for every workstation they were copied to.

Batch Files To Execute

Which batch files have to be started depends on the sample application itself. However, for every sample application the batch file to start the central has to be started first. For Windows this is the file `<MyJanetDir>/startCentral.bat`. When you shutdown nodes you need to remember to shutdown the central as the last node (the central will refuse to shut down in case other running nodes exist).

8.2 Setting Up Sample Applications

The following list describes the steps that have to be carried out to set up your environment for Java and to set up the sample applications:

1. Make sure you have the Java SDK v1.3 or later installed on your machine. Please note that RMI will not work properly if a firewall is installed between machines with communicating Janet nodes.
2. On the machine where you want to run the central extract the file `binaries_samples.zip` to `<MyJanetDir>`. Make sure that directory names in your `<MyJanetDir>` path does not contain spaces.
3. In the files listed below change the value of the attribute `centralHostname` to the name of the machine you are installing the central (most probably the machine you are using at the moment).

Always required:

- `<MyJanetDir>/samples/nodes/central/nodeDef.xml`
- `<MyJanetDir>/ samples/nodes/logger/nodeDef.xml`
- `<MyJanetDir>/ samples/nodes/monitor/nodeDef.xml`

Required to run the CAS.Fibonacci sample:

- `<MyJanetDir>/ samples/nodes/cas.fibonacci/nodeDef.xml`

Required to run the ADÉ.Fibonacci sample:

- `<MyJanetDir>/ samples/nodes/ade.distributor/nodeDef.xml`
- `<MyJanetDir>/ samples/nodes/ade.observer/nodeDef.xml`
- `<MyJanetDir>/ samples/nodes/ade.fibonacci/nodeDef1.xml`
- `<MyJanetDir>/ samples/nodes/ade.fibonacci/nodeDef2.xml`
- `<MyJanetDir>/ samples/nodes/ade.fibonacci/nodeDef3.xml`

4. For the batch files listed below change the path statement to point your Java SDK bin directory (problems may occur if the path to your Java SDK includes spaces). At the moment batch files to start up Janet nodes are only provided for windows. The existing files for Windows must be adapted manually to run the sample applications on Linux or another platform supported by Java.

Always required:

- <MyJanetDir>/samples/startCentral.bat
- <MyJanetDir>/samples/startLogger.bat
- <MyJanetDir>/samples/startMonitor.bat

Required to run the CAS.Fibonacci sample:

- <MyJanetDir>/samples/startCASFibonacciNode.bat

Required to run the ADÉ.Fibonacci sample:

- <MyJanetDir>/samples/startObserverNode.bat
- <MyJanetDir>/samples/startDistributorNode.bat
- <MyJanetDir>/samples/startADEFibonacciNode1.bat
- <MyJanetDir>/samples/startADEFibonacciNode2.bat
- <MyJanetDir>/samples/startADEFibonacciNode3.bat

Log Files

In case the attribute `active` in the `localLogger` section of a node's descriptor is set to true, the node writes its log messages into the directory it was started up from into a file with the node name as file name and the extension "log". All the messages logged in between startup and shutdown of a node can be observed from these log files. The log files remain locked until the node has shutdown. They contain exception dumps as well, which might be helpful in analyzing setup problems.

8.3 Running the CAS.Fibonacci Sample

This sample application is explained in chapter 5.3: "Driving Application: Fibonacci Numbers". The source for this sample application is included in the `sources.zip` file in the `sources` directory. The classes that belong to this application are located in the package `org.objectscape.janet.cas.demo.fibonacci`.

Running Sample Applications

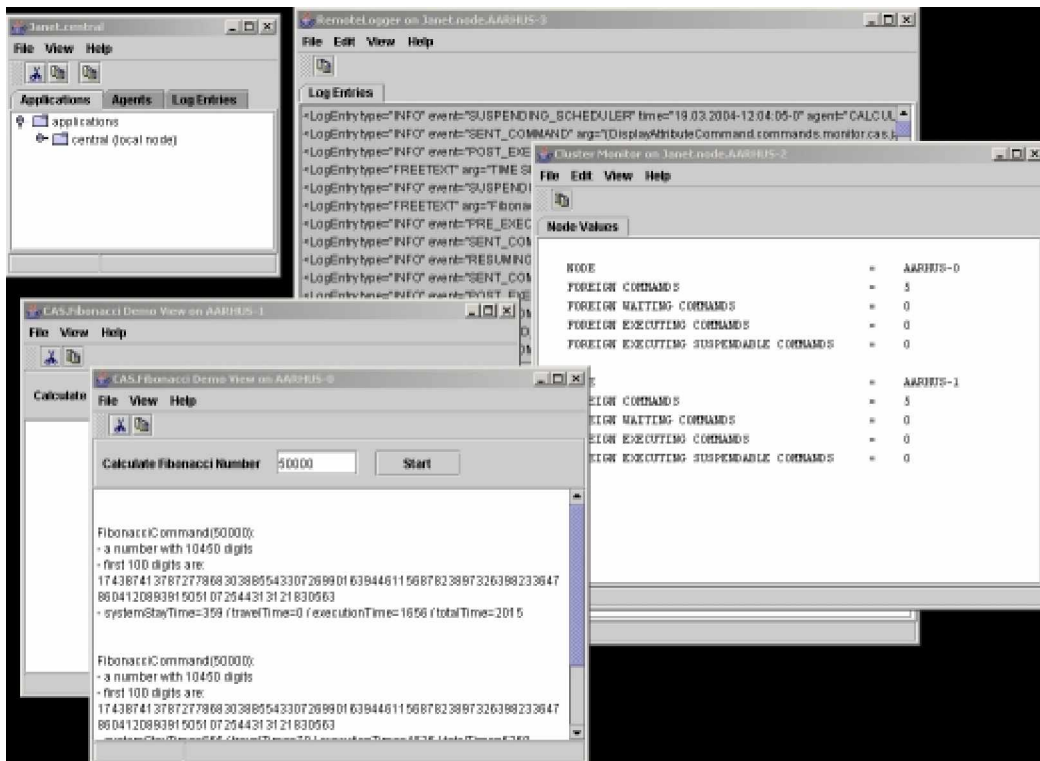


Figure 8-1: Sample CAS.Fibonacci setup

Figure 8-1 displays a sample setup for the CAS.Fibonacci sample application with two Fibonacci nodes, a remote logger node, and a monitor node to monitor command sends within the cluster. To run the sample, carry out the following steps:

1. Run the startCentral batch file. Wait till its main view has appeared on screen.
2. Start as many nodes as you like by running the startCASFibonacciNode batch file. Wait till the main views have appeared on screen.
3. If you want to see a log of all commands executed by any agent within the cluster you can also start up the remote logger by running the startLogger batch file.
4. To initiate calculation of Fibonacci numbers on all nodes in the cluster you can push the start button in any CAS.Fibonacci Demo View. If you started the monitor (which you can still start now) you can then watch the commands being exchanged between nodes to calculate the Fibonacci numbers.

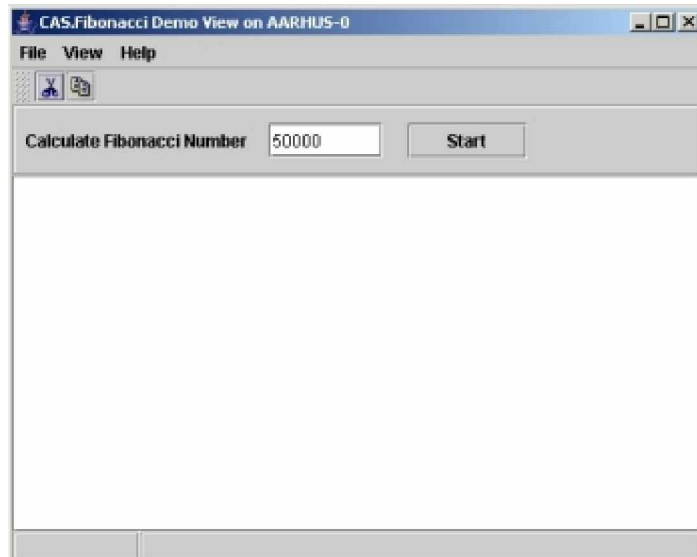


Figure 8-2: CAS.Fibonacci Demo View

5. If you want to follow all the commands that are exchanged between nodes you can also start up the monitor node by running the startMonitor batch file. It is recommended to start up the monitor as the last node. The command counts displayed on the monitor are incremented and decremented by the nodes they belong to. If the monitor node is started up while other nodes are started up at the same time, the command counts on the monitor display may have missed increment or decrement notifications from the node and show an inconsistent picture.

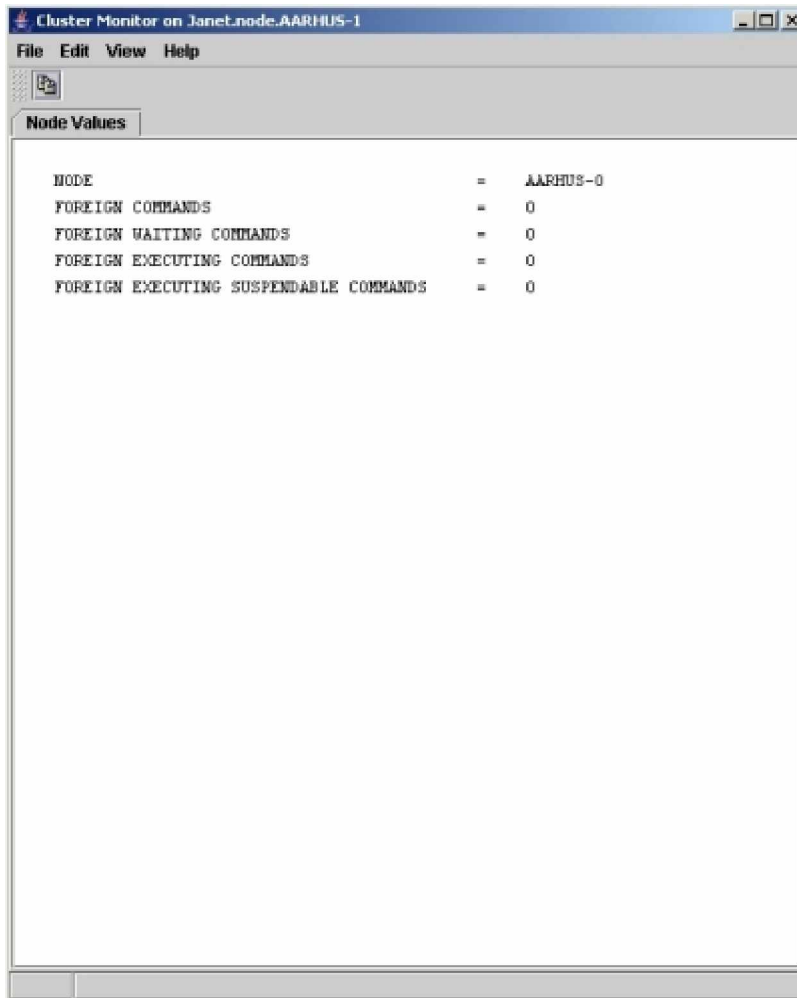


Figure 8-3: Monitor node with one node named AARHUS-0 in the cluster

8.4 Running the *ADÉ.Fibonacci* Sample

This sample application is explained in chapter 6.3: “Driving Application: Fibonacci Numbers Revisited”. The source for this sample application is included in the `sources.zip` file in the `sources` directory. The classes that belong to this application are located in the package `org.objectscope.janet.ade.demo.fibonacci`.

1. Run the `startCentral` batch file. Wait till its main view has appeared on screen.
2. If you want to see a log of all commands executed by any agent within the cluster you can also start up the remote logger by running the `startLogger` batch file.
3. Start the observer by running the `startObserverNode` batch file. It is assumed that all nodes are started on the same workstation. In case the application is run from several workstations, one observer node must be started on every workstation.

4. Start as many ADÉ.Fibonacci executor nodes as you like by running one of the following batch files: `startADEFibonacciNode1`, `startADEFibonacciNode2`, or `startADEFibonacciNode3`. Wait till the main views have appeared on screen.

Starting the `startADEFibonacciNode1` batch file results in an ADÉ.Fibonacci executor node to be started up where every ADÉ.Fibonacci application has 1 calculator agent. Starting the `startADEFibonacciNode2` batch file results in an ADÉ.Fibonacci executor node to be started up where every ADÉ.Fibonacci application has 2 calculator agents. Starting the `startADEFibonacciNode3` batch file results in an ADÉ.Fibonacci executor node to be started up where every ADÉ.Fibonacci application has 3 calculator agent. The idea behind setting up ADÉ.Fibonacci executor nodes is explained at the beginning of section 6.3. Each ADÉ.Fibonacci executor node has a console with which the cluster can be loaded with workload-aware commands. How to use the console is explained at the beginning of section 6.3 as well.

5. After ADÉ.Fibonacci executor nodes have been started up they need to be connected with the observers that are supposed to be observed. All nodes that belong to one observer are considered by the distributor agent to be on the same workstation. The `settings` notebook tab in the observer's main view (see Figure 6-18) lets the user specify which nodes belong to an observer. The observers do not check for double assignments. The user has to take care herself not to assign an executor node to several observers.
6. If you want to follow all the commands that are exchanged between nodes you can also start up the monitor node by running the `startMonitor` batch file. It is recommended to start up the monitor as the last node. The command counts displayed on the monitor are incremented and decremented by the nodes they belong to. If the monitor node is started up while other nodes are started up at the same time, the command counts on the monitor display may have missed increment or decrement notifications from the node and show an inconsistent picture.

8.5 Using Janet with eclipse

Janet has been developed using eclipse. The Janet system consists of several eclipse projects. For the time being, there seems to be no straightforward way to import several projects into eclipse in one go. Therefore, the projects that constitute Janet have to be imported manually one by one. The steps to import these projects are explained later on. As a brute force workaround the eclipse workspace directory and the eclipse `.config` directory of Janet are provided so that they can be copied into an existing eclipse installation. The eclipse workspace directory `.config` directory is based on eclipse M8.

Copying workspace And .config Directories

The file `everything4eclipse.zip` in the Janet/eclipse directory on the accompanying CD-ROM includes the eclipse workspace directory and the `.config` directory of Janet. An easy way to browse code and debug Janet is to copy these directories over your existing local eclipse directories. Make sure to rename or backup the existing directories or use a new eclipse installation instead. After copying these directories into some existing eclipse installation you need to edit the node descriptor files in the `<eclipse>/workspace/nodes` directory to set the value of the `centralHostname` attribute to the name of your workstation as explained in section 8.2.

Running Sample Applications

Thereafter you can make immediate use of the debug launching pads to start nodes from within eclipse or to debug nodes. You can insert breakpoints in the code wherever you want.

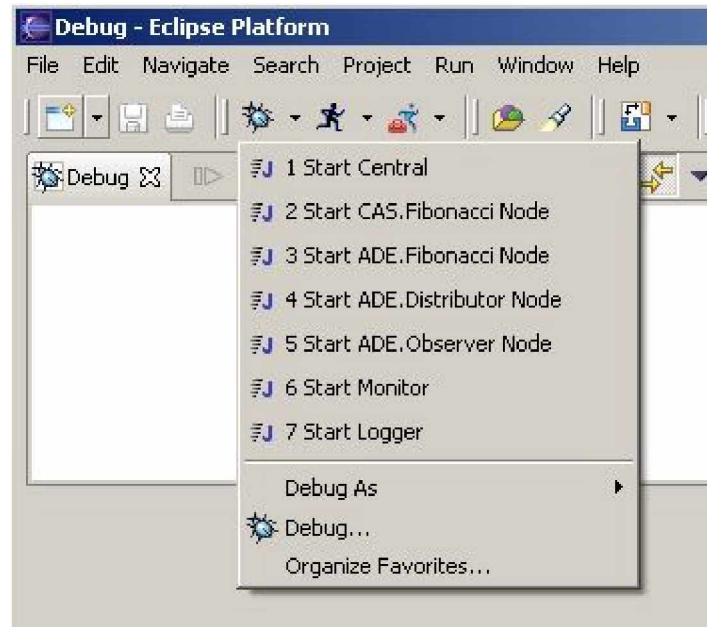


Figure 8-4: eclipse launching pads to start or debug nodes with eclipse

When you rebuild a project the RMI stubs and skeleton files will be removed and have to be re-generated. How to do this is explained later in this chapter in the section “Regenerating RMI stubs and skeleton files”.

Importing Janet Projects into eclipse

Copy all the project directories in workspace.zip from the Janet/eclipse directory on the accompanying CD-ROM into your local eclipse directory. You have to import the following projects from the workspace.zip file in the given order:

- Objectscope.Common
- Objectscope.Janet.CAS
- Objectscope.Janet.CAS.Logger
- Objectscope.Janet.CAS.Monitor
- Objectscope.Janet.CAS.Demo.Fibonacci
- Objectscope.Janet.ADE
- Objectscope.Janet.ADE.Simulation
- Objectscope.Janet.ADE.Demo.Fibonacci

Import the projects as simple eclipse projects from the Java Browsing window selecting the menu items File > New > Project > Simple Project > Next > enter project name in given order.

Regenerating RMI stubs and skeleton files

After importing projects and when rebuilding a project, eclipse scrubs all class files from the output directories and rebuilds them. This will result in all RMI stubs and skeleton class files to disappear. They therefore have to be recreated running the rmi.bat batch file in the <eclipse>/workspace/objectscope.janet.cas directory. The rmi.bat file has to be adapted to point to your local Java SDK installation directory. The RMI compiler is only included in the Java SDK. If you have the Java Runtime Environment (JRE) installed you can get the RMI stubs and skeleton files from the stubs.zip file included in everything4eclipse.zip.

Setting Up Launching Pads

Importing the Janet eclipse projects into eclipse using the New Project wizard does not create the respective launching pads as displayed in Figure 8-4. To start up nodes from within eclipse or debug nodes the launching pads have to be set up.

Every launching pad entry has to be created carrying out the following steps: In a Java Browser window click on the debug icon and select “Debug...”. Select “Java Application”, click “New” and enter the respective values as listed in the following. For some launching pad entries the classpath has to be set to import required projects as displayed in Figure 8-5.

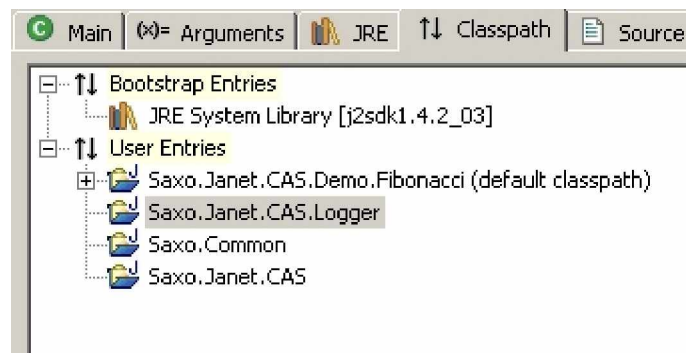


Figure 8-5: eclipse Classpath tab

- Launching Pad Entry for the Central

Name: Start Central

Main tab/Project: Objectscape.Janet.CAS

Main tab/Main class: org.objectscope.janet.cas.node.NodeStarter

Arguments tab/Program arguments: "../nodes/central/nodeDef.xml"

Add to Classpath tab: Objectscape.Janet.CAS.Logger

- Launching Pad Entry for the Remote Logger

Name: Start Remote Logger

Main tab/Project: Objectscape.Janet.CAS.Logger

Main tab/Main class: org.objectscope.janet.cas.node.NodeStarter

Running Sample Applications

Arguments tab/Program arguments: "../nodes/logger/nodeDef.xml"

Add to Classpath tab: no additional entries required

- Launching Pad Entry for the Monitor

Name: Start Monitor

Main tab/Project: Objectscape.Janet.CAS.Monitor

Main tab/Main class: org.objectscapes.janet.cas.node.NodeStarter

Arguments tab/Program arguments: "../nodes/monitor/nodeDef.xml"

Add to Classpath tab: no additional entries required

- Launching Pad Entry for the CAS.Fibonacci node

Name: Start CAS.Fibonacci Node

Main tab/Project: Objectscape.Janet.CAS.Demo.Fibonacci

Main tab/Main class: org.objectscapes.janet.cas.node.NodeStarter

Arguments tab/Program arguments: "../nodes/cas.fibonacci/ nodeDef.xml"

Add to Classpath tab: Objectscape.Janet.CAS.Logger, Objectscape.Janet.CAS.Monitor

- Launching Pad Entry for the ADÉ.Distributor node

Name: Start ADÉ Distributor

Main tab/Project: Objectscape.Janet.ADE

Main tab/Main class: Org.objectscapes.janet.cas.node.NodeStarter

Arguments tab/Program arguments: "../nodes/ade.distributor/nodeDef.xml"

Add to Classpath tab: Objectscape.Janet.CAS.Logger, Objectscape.Janet.CAS.Monitor,
Objectscapes.Janet.ADE.Simulation

- Launching Pad Entry for the ADÉ.Observer node

Name: Start ADÉ Observer

Main tab/Project: Objectscape.Janet.ADE

Main tab/Main class: Org.objectscapes.janet.cas.node.NodeStarter

Arguments tab/Program arguments: "../nodes/ade.observer/nodeDef.xml"

Add to Classpath tab: Objectscape.Janet.CAS.Logger, Objectscape.Janet.CAS.Monitor,
Objectscapes.Janet.ADE.Simulation

- Launching Pad Entry for the ADÉ.Fibonacci 1 node

Name: Start ADÉ Fibonacci

Main tab/Project: Objectscape.Janet.ADE.Demo.Fibonacci

Main tab/Main class: Org.objectscapes.janet.cas.node.NodeStarter

Arguments tab/Program arguments: "../nodes/ade.fibonacci/nodeDef1.xml"

Add to Classpath tab: Objectscape.Janet.CAS.Logger, Objectscape.Janet.CAS.Monitor,
Objectscapes.Janet.ADE.Simulation

- Launching Pad Entry for the ADÉ.Fibonacci 2 node

Name: Start ADÉ Fibonacci

Main tab/Project: Objectscape.Janet.ADE.Demo.Fibonacci

Main tab/Main class: Org.objectscap.janet.cas.node.NodeStarter

Arguments tab/Program arguments: "../nodes/ade.fibonacci/nodeDef2.xml"

Add to Classpath tab: Objectscape.Janet.CAS.Logger, Objectscape.Janet.CAS.Monitor,
Objectscape.Janet.ADE.Simulation

- Launching Pad Entry for the ADÉ.Fibonacci 3 node

Name: Start ADÉ Fibonacci

Main tab/Project: Objectscape.Janet.ADE.Demo.Fibonacci

Main tab/Main class: Org.objectscap.janet.cas.node.NodeStarter

Arguments tab/Program arguments: "../nodes/ade.fibonacci/nodeDef3.xml"

Add to Classpath tab: Objectscape.Janet.CAS.Logger, Objectscape.Janet.CAS.Monitor,
Objectscape.Janet.ADE.Simulation

Debugging Nodes Using Launching Pads

When debugging a node you only need to start the kind of node you want to debug from the eclipse launching pad. If you want to debug the central, you can start it using its launching pad. All other nodes can be started with the use of the respective batch files as explained in the subsequent chapters for the CAS.Fibonacci sample or the ADÉ.Fibonacci sample. If you want to debug a node, you can start up the central running the respective batch file, start an arbitrary number of nodes in addition with batch files, and then start a node for debugging using the respective launching pad. You can also start up the central and all nodes using launching pads.

9 Summary and Conclusions

In this thesis work a platform for distributed cooperative agents in Java has been developed. The platform offers distributed event notification and distributed object spaces. It provides an interface for users to plug in their custom applications that make use of the agent platform. Applications consist of capabilities that are assigned to agents and constitute their behavior. Instead of invoking methods on agents, an alternative paradigm is used where agents communicate through the asynchronous exchange of commands to which they respond with the execution of an interpreter they have chosen autonomously. Interpreters to be executed are scheduled by an agent's scheduler that executes interpreters sequentially. Agents with the same capability are able to communicate directly through commands and may be distributed over several nodes. A node provides the platform's runtime environment, which serves as a habitat for agents residing on a workstation. There is no restriction to the number of nodes per workstation.

The workload distribution layer is completely separated from the agent platform. It allows to level out load imbalances by evicting commands from overloaded agents to lightly loaded agents on other nodes. Since interpreters are inserted into scheduler queues, detecting lightly loaded agents using the ShortestQueue algorithm is straightforward. Initial placement of commands is provided in order to offer workload sharing of commands to the user as well as migration of started commands to offer workload balancing. The user has to assist the balancing system by implementing commands eligible for workload balancing in such a way that the system can suspend them before migration. With this slight sacrifice in transparency workload balancing could be implemented for Java without having to modify the Java virtual machine to provide transparent thread migration.

Summing up, the concept used by the developed system represents an interesting compromise between workload distribution systems on system-level that cannot be tailored to an application's specific requirements and systems that leave load distribution completely to the programmer. The system offers an interesting combination of an agent platform and a workload balancing system. It can be used as a workload balancing system only, as an agent platform only, or as an agent-oriented system that can balance the workload of agent's.

Summary and Conclusions

10 References

- [AF89] Y. Artsy, R. Finkel, “Designing a process migration facility: The Charlotte experience”, *IEEE Computers*, vol. 22, pp. 47–56, September 1989.
- [BAR93] A. Barak, S. Guday, and R. Wheeler, “The MOSIX Distributed Operating System”, *Load Balancing for UNIX. Lecture Notes in Computer Science*, Vol. 672, SpringerVerlag, 1993.
- [BRHL99] P. Busetta, R. Rönquist, A. Hodgson, A. Lucas, “Light-Weight Intelligent Software in Simulation”, *Agent Oriented Software Pty. Ltd.*, Melbourne, Australia, 1999.
- [BB01] J.P. Bigus, J. Bigus, “Constructing Intelligent Agents Using Java”, New York. Wiley 2001.
- [BD93] A. Bruns, G. Davis, “Concurrent Programming”, Addison-Wesley, Reading, Massachusetts, 1993.
- [BK90] F. Bonomi, A. Kumar, “Adaptive optimal load balancing in a nonhomogeneous multiserver system with a central job scheduler”, *IEEE Transactions On Computers*, vol. 39, pp. 1232–1250, October 1990.
- [BRU91] J.C. Brustoloni, “Autonomous Agents: Characterization and Requirements”, *Carnegie Mellon Technical Report CMU-CS-91-204*, Pittsburgh, Carnegie Mellon University, 1991.
- [BRM98] C. Busch, V. Roth, and R. Meister, “Perspectives on electronic commerce with mobile agents”. In: S. Rodinov and M. Vinogradov, editors, *Proc. 11th Amaldi Conference on Problems of Global Security*, pages 89-101, Moscow, Russia, November 1998.
- [CHE88] D. R. Cheriton, “The V distributed system”, *Communications of the ACM*, vol. 31, pp. 314–333, March 1988.
- [CH02] G. Chen et al., “Coordinating Multi-Agents using JavaSpaces”. In: *Proceedings of the 9th International Conference on Parallel and Distributed Systems*, Taiwan, p.63, 2002.
- [CL95] R.E. Cohen, H.J. Levesque, “Communicative Actions for Artificial Agents”. In: *Proceedings of the First International Conference on Multi-Agent Systems*, San Francisco, Cambridge: AAAI Press, p. 65-72, 1995.
- [DO91] F. Douglass, J. Ousterhout, “Transparent process migration: Design alternatives and the Sprite implementation”, *Software—Practice & Experience*, vol. 21, pp. 757–785, Aug. 1991.
- [EAG86] D. Eager et al., “A comparison of receiver-initiated and sender-initiated adaptive load sharing”, *Performance Evaluation*, vol. 6, pp. 53–68, March 1986.
- [EG04] F.B. Engelhardt, T. Gagnes, “Using JavaSpaces to create adaptive distributed systems”, *10th Open European Summer School and IFIP Workshop on the Advances in Fixed and Mobile Networks*, Tampere, Finland, June 2004.
- [ELZ86] D. L. Eager, E. D. Lazowska, and J. Zahorjan, “Adaptive load sharing in homogeneous distributed systems”, *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 662–675, May 1986.
- [FA96] A. Farquhar et al., “The Ontolingua Server: a Tool for Collaborative Ontology Construction”. In: *Proceedings of the 10th Knowledge Acquisition Workshop, KAW'96*, Banff, Canada, November 1996.

References

- [FA98] J. Farley, "Java Distributed Computing", O'Reilly & Associates , 1998.
- [FER01] J. Ferber, "Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence", Addison-Wesley, 2001.
- [FFMM94] T. Finin, R. Fritzson, D. McKay, R. McEntire, "KQML as an Agent Communication Language". In: Proceedings of the Third International Conference on Information and Knowledge Management, ACM Press, November 1994.
- [FIN94] T. Finin, et al., "KQML as an agent communication language". In: Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM), November, New York: ACM Press, 1994.
- [FRA95] S. Franklin, Artificial Minds, Cambridge, Massachusetts, MIT Press, 1995.
- [FG96] S. Franklin, A. Graesser, "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents". Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, <http://www.dfki.uni-sb.de/~jpm/atal96.html>, Springer-Verlag, 1996.
- [GA95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns", Addison-Wesley, Reading, Massachusetts, 1995.
- [GE95] D. Gelernter, "Generative Communication in Linda, ACM Transactions on Programming Languages and Systems", Vol. 7, No. 1, pp. 80-112, January 1995.
- [GF92] M. R. Genesereth, R.E. Fikes, "Knowledge Interchange Format Version 3.0 Reference Manual" (Report of the Knowledge Systems Laboratory KSL 91-1): Stanford University, 1992.
- [GILB95] D. Gilbert et al., "IBM Intelligent Agent Strategy", whitepaper, 1995, <http://activist.gpl.ibm.com:81/WhitePaper/ptc2.htm>
- [GR01] W. Grosso , "Java RMI. Designing and Building Distributed Applications.", O'Reilly & Associates , 2001.
- [HR95] B. Hayes-Roth, "An Architecture for Adaptive Intelligent Systems", Artificial Intelligence: Special Issue on Agents and Interactivity, 72, p 329-365, 1995.
- [IAG1] IBM Corporation, "Intelligent Agent Strategy", <http://activist.gpl.ibm.com:81/WhitePaper/ptc2.htm>
- [LAN02] G. Lanfermann, "Nomadic Migration - A Service Environment For Autonomic Computing On The Grid", PhD Thesis, University of Potsdam, Germany, November 2002, <http://pub.ub.uni-potsdam.de/2003/0018/lanferm.pdf>.
- [LEA99] D. Lea, "Concurrent Programming in Java: Design Principles and Patterns", Addison-Wesley, November 1999, <http://gee.cs.oswego.edu/dl/cpj/>.
- [LO98] D.B. Lange, M. Oshima, "Programming and Deploying Java Mobile Agents with Aglets", Reading, Massachusetts, Addison Wesley 1998.
- [NEC94] R. Neches, "Overview of the DARPA Knowledge Sharing Effort", 1994, www-ksl.stanford.edu/knowledge-sharing/papers/kse-overview.htm.
- [NWA96] H. S. Nwana, "Software Agents: An Overview", Knowledge Engineering Review, Vol. 11, No 3, pp.1-40, Sept 1996. Cambridge University Press, 1996.
- [MA99] M.J.M Ma et al., "JESSICA: Java-Enabled Single-System-Image Computing Architecture", Proceedings of the International Conference

- on Parallel and Distributed Processing Techniques and Applications (PDPTA), June 1999.
- [MAG96] T. Magedanz, K. Rothermel, S. Krause, "Intelligent Agents: AN Emerging Technology for Next Generation Telecommunications?", INFOCOM '96, San Francisco, USA, März, 1996.
- [MAE93] P. Maes et al., "Learning Interface Agents", Proceedings of the 11th National Conference on Artificial Intelligence. AAAI, MIT-Press/AAAI-Press, 1993.
- [MAE95] P. Maes, "Artificial Life Meets Entertainment: Life like Autonomous Agents", Communications of the ACM, 38, 11, 108-114, 1995.
- [MAT] F. Mattern, "Mobile Agenten", Fachbereich Informatik, Technische Universität Darmstadt, www.inf.ethz.ch/~mattern/papers/mobags.html.
- [MDBP96] P. Maes, T. Darrell, B. Blumberg, and A. Pentland, "The ALIVE System: Wireless, Full-Body Interaction with Autonomous Agents," To be published in a Special Issue on Multimedia and Multisensory Virtual Worlds, ACM Multimedia Systems, ACM Press (Spring), 1996, <http://agents.www.media.mit.edu/groups/agents/papers.html>
- [MTS89] R. Mirchandaney, D. Towsley, and J. A. Stankovic, "Analysis of the effects of delays on load sharing", IEEE Transactions on Computers, vol. 38, pp. 1513–1525, November 1989.
- [OSS92] W. Osser, "Automatic Process Selection for Load Balancing", Master Thesis, University of California at Santa Cruz, June 1992.
- [OUS88] J.K. Ousterhout et al.: "The Sprite Network Operating System", Computer Magazine of the Computer Group News of the IEEE Computer Group Society, ACM CR 8905-0314, 1988.
- [OW97] S. Oaks, H. Wong, "Java threads", O'Reilly & Associates, 1997.
- [PPHG02] U. Pinsdorf, J. Peters, M. Hoffmann, and P. Gupta, "Context-aware Services based on Secure Mobile Agents". In: Proceedings of 10th International Conference on Software, Telecommunications & Computer Networks (SoftCOM 2002), pages 366-370, IEEE Communication Society, Ministry of Science and Technology Republic of Croatia and University of Split, R. Boskovica, HR-21000 Split, Croatia, October 2002.
- [PPW02] J. Pichler, R. Plösch, W. Weinrich, "MASIF und FIPA: Standards für Agenten", Informatik-Spektrum, 18. April 2002.
- [RN95] J.S. Russell, P. Norvig, "Artificial Intelligence: A Modern Approach", Englewood Cliffs, New Jersey. Prentice Hall 1995.
- [RW92] M. Reiser, N. Wirth, "Programming in Oberon – Steps beyond Pascal and Modula", Reading, Massachusetts, Addison Wesley, 1992.
- [SCH95] T. Schnekenburger, "The ALDY Load Distribution System", Institute of Computer Science, Munich Technical University, 1995.
- [SCH97] T. Schnekenburger, "Implementing Dynamic Load Distribution Strategies with Orbix". In: International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, Volume II, pp. 996-1005, 1997.
- [SCH97+] T. Schnekenburger, "Cooperating Agents: Language Support and Load Distribution". In: 11th Int. Parallel Processing Symposium, Workshop on High-Level Parallel Programming Models and Supportive Object spaces, Geneva, Switzerland, p. 115-119, IEEE, 1997.
- [SE69] J.R. Searle, "Speech Acts", Cambridge, Massachusetts, Cambridge

References

- University Press, 1969.
- [SM01] M. Schoettner, O. Marquardt, et al., “Architecture of an Object-Oriented Cluster Operating System”, European Conference on Object-Oriented Programming - 4th Workshop on Object-Orientation and Operating Systems, Budapest, Hungary, 2001.
- [SYC98] K. Sycara, “Multiagent Systems”, *AI Magazine* 19(2): 79-92.
- [SYC02] K. Shen, T. Yang, and L. Chu, “Cluster Load Balancing for Fine-Grain Network Services”. In: International Parallel and Distributed Processing Symposium, 2002.
- [STS98] M. Schoettner, S. Traub, and P. Schulthess, “A transactional DSM Operating System in Java”. In: Proceedings of the 4th International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, USA, 1998.
- [SUN99] Sun Microsystems. JavaSpaces TM Specification. Whitepaper, Sun Microsystems, January 25 1999.
- [SWM03] M.K. Smith, C. Welty, and D. McGuinness, “Owl web ontology language guide”, <http://www.w3.org/TR/owl-guide/>, 2003.
- [TAN01] A. Tanenbaum, “Modern Operating Systems”, Prentice Hall 2001.
- [THO87] A. Thomasian, “A performance study of dynamic load balancing in distributed systems”, *IEEE*, pp. 178–184, August 1987.
- [THO98] C. Thompson, “The DARPA Agent Reference Architecture”, Object Services and Consulting Inc., August 1998.
- [UNIBITF94] I. Wachsmuth et al., “Agentensysteme“, Technische Fakultät Universität Bielefeld, www.techfak.uni-bielefeld.de/kvv/WS94-95/kommentare/392112.html, 1994.
- [VQC02] G. Vitaglione, F. Quarta, E. Cortese, “Scalability and Performance of JADE Message Transport System”, presented at AAMAS Workshop on AgentCities, Bologna, 16th July, 2002.
- [WA00] A. I. Wang, “Implementing a Multi-Agent Architecture for Cooperative Software Engineering”. In: Twelfth International Conference on Software Engineering and Knowledge Engineering (SEKE’2000), Chicago, USA, 6-8 July 2000.
- [WO02] M. Wooldridge, “An Introduction to Multiagent Systems”, John Wiley and Sons Ltd, Chichester, England, 2002.
- [ZAY86] E. Zayas, “Attacking the process migration bottleneck”, In: Proceedings of the 1986 fall joint computer conference, pp. 1–23, IEEE, May 1986.
- [ZWD91] S. Zhou, X. Zheng, J. Wang, and P. Delisle, “Utopia: A load sharing system for large, heterogeneous distributed computer systems”, Tech. Rep. CSRI-257, University of Toronto, November 1991.

Appendix A: Abbreviations

General Abbreviations

ACL	Agent Communication Language
LFS	Log-Structured File System
MOSIX	Multicomputer OS for UNIX
NFS	Network File System

Abbreviations Janet

ADÉ	Automatic Distributed Execution
DIC	Distributed Code
CAS	Cooperative Agent System
QSC	Queue Size Category
CQS	Category Queue Size

Appendix A: Abbreviations

Appendix B: Links

Systems

Aglets	http://www.trl.ibm.com/aglets/
Amoeba	http://www.cs.vu.nl/pub/amoeba/
CORBA	http://www.corba.org/
FIPA	http://www.fipa.org/
GigaSpaces	http://www.gigaspaces.com/
Intamission	http://www.intamission.com/
JACK	http://www.agent-software.com/
JADE	http://sharon.csel.it/projects/jade/
JavaSpaces	http://www.javasoft.com/products/javaspaces/
JESSICA	http://www.srg.csis.hku.hk/jessica.htm
JMS	http://java.sun.com/products/jms/
JPower	http://www.jpower.org/
JUnit	http://www.junit.org/
Linda	http://www.cs.yale.edu/Linda/linda.html
MOSIX	http://www.mosix.org/
OMG	http://www.omg.org/
Outrigger	http://wwws.sun.com/software/jini/
OWL	http://www.w3.org/2001/sw/WebOnt/
Plurix	http://www.plurix.de/
RMI	http://java.sun.com/products/jdk/rmi/
SeMoA	http://www.semoa.org/
SOAP	http://www.w3.org/TR/SOAP/
Sprite	http://www.cs.berkeley.edu/projects/sprite/sprite.html
W3C	http://www.w3c.org
XML-RPC	http://www.xml-rpc.org/

Other

Agent FAQ	http://www.ee.mcgill.ca/~belmarc/agent_faq.html
-----------	---