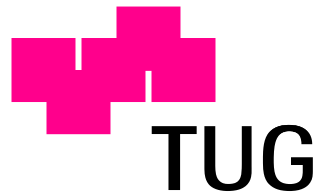
Magisterarbeit

Load Distribution for Embedded Smart Cameras based on Mobile Agents

Thomas Winkler

Institut für Technische Informatik
Technische Universität Graz
Vorstand: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß



Begutachter: Ao. Univ.-Prof. Dipl.-Ing. Dr. Bernhard Rinner
Betreuer: Dipl.-Ing. Michael Bramberger

Graz, im Mai 2005

Abstract

In third generation video surveillance systems the overall processing is distributed among the whole system. Current technology provides enough computational power to digitize, analyze, and process video data directly on the cameras of the surveillance system. The SmartCam project is dedicated to the development of such a high-performance camera that is equipped with multiple digital signal processors (DSPs) for video processing and an XScale network processor. The network processor controls the DSPs and is responsible for intra- and inter-camera communication.

In a typical surveillance system many cameras are deployed and a lot of different surveillance tasks have to be executed. The cameras are grouped into clusters in order to support the allocation of these tasks to cameras. Not every task is executed on every camera of a cluster. Since there is no dedicated central system, the allocation of tasks to cameras has to be managed in a decentralized way.

This work focuses on the design of a dynamic load distribution system for such embedded smart cameras. The load distribution is realized using mobile agents. Mobile agents represent surveillance tasks and can migrate from one camera to another at runtime. The mobile agent system is implemented on the network processor running under GNU/Linux. The image processing is done on the DSPs, hence the DSP applications are included in the agent's payload.

The load distribution problem is modeled as a distributed constraint satisfaction problem that is solved in parallel on all cameras of a cluster. The partial solutions are then merged resulting in a set of complete allocations of agents to the cameras of the cluster. To select the optimal solution, costs are assigned to the different placements of the agents. The proposed load distribution mechanisms were implemented in a prototype system. Various experiments demonstrated the feasibility of this load distribution mechanism.

Zusammenfassung

Video Überwachungssysteme der dritten Generation zeichnen sich durch eine verteilte Verarbeitung der Überwachungsaufgaben aus. Aktuelle Technologien besitzen ausreichend Rechenleistung um die Digitalisierung, Analyse und Verarbeitung der Videodaten direkt auf den Kameras durchzuführen. Ziel des SmartCam Projekts ist die Entwicklung einer solchen Kamera basierend auf mehreren digitalen Signalprozessoren (DSPs) für die Bildverarbeitung und einem XScale Netzwerk-Prozessor. Der Netzwerk-Prozessor steuert die DSPs und kontrolliert die Kommunikation sowohl innerhalb des Systems als auch zwischen Kameras.

In einem typischen Überwachungssytem werden mehrere Kameras eingesetzt, denen eine Reihe von Aufgaben zugewiesen wird. Um die Zuweisung von Aufgaben zu erleichtern, werden die Kameras zu so genannten Clustern zusammengefasst. Nicht jede Aufgabe wird dabei auf jeder Kamera ausgeführt. Da kein zentrales System vorgesehen ist, das die Zuteilung von Aufgaben zu Kameras durchführt, muss dies verteilt erfolgen.

Der Fokus dieser Arbeit liegt auf dem Entwurf eines dynamischen Lastverteilungssystems für eingebettete intelligente Kameras. Die Lastverteilung basiert auf dem Konzept von mobilen Agenten. Überwachungsaufgaben werden als mobile Agenten realisiert die zur Laufzeit zwischen Kameras migrieren können. Das Agentensystem läuft auf dem Netzwerk Prozessor, auf dem GNU/Linux als Betriebssystem zum Einsatz kommt. Die DSP Applikationen sind Teil der Agenten.

Das Problem der Lastverteilung wird als verteiltes "Constraint Satisfaction Problem" (CSP) modelliert das parallel auf allen Kameras eines Clusters gelöst wird. Die sich dabei ergebenden partiellen Lösungen werden anschließend zu einer Gesamtlösung verschmolzen. Diese Gesamtlösung beinhaltet die vollständigen Zuweisungen von Agenten zu den Kameras des Clusters. Um aus den vollständigen Lösungen die Optimale auszuwählen, werden den einzelnen Lösungen Kosten zugewiesen. Das vorgeschlagenen Lastverteilungssystem wurde in einem Prototypensystem umgesetzt. Darauf aufbauende Experimente und Messungen haben die Realisierbarkeit eines solchen Systems unter Beweis gestellt.

Danksagung

Diese Magisterarbeit wurde im Studienjahr 2004/2005 am Institut für Technische Informatik an der Technischen Universität Graz verfasst. Die Durchführung der Arbeit wurde vom Austian Research Center Seibersdorf (ARCS) unterstützt.

Als erstes geht mein Dank an meinen Betreuer Michael Bramberger für die gute Zusammenarbeit und seine tatkräftige Unterstützung in allen Phasen der Magisterarbeit. Weiters möchte ich mich bei Herrn Professor Bernhard Rinner für die gute Betreuung bedanken. Außerdem gilt mein Dank allen Mitgliedern der SmartCam Gruppe für die gute Atmosphäre.

Mein ganz besonderer Dank gilt meinen Eltern für ihre Unterstützung und ihr Verständnis während meines gesamten Studiums.

Graz, Mai 2005 Thomas Winkler

Contents

Contents **1**

List of Figures **4**

1 Problem Domain and Motivation **6**

1.1 Smart Cameras 6

1.2 Load Distribution Using Mobile Agents 7

1.3 Thesis Organization 7

2 Background **9**

2.1 Agent Based Software Development 9

2.1.1 Intelligent Agents 10

2.1.2 Distributed and Mobile Agents 10

2.1.3 Agent Technologies 11

2.1.4 Agent Environments 12

2.1.5 Agent Interoperability Standards 14

2.1.6 Advantages and Disadvantages of Mobile Agents 16

2.2 Java in Embedded Environments 17

2.2.1 Java 2 Micro Edition 18

2.2.2 JamaicaVM - Ahead of Time Compilation 20

2.2.3 Free Java Environments 21

2.3 Load Distribution 22

2.3.1 Requirements for Load Distribution 22

2.3.2 Types of Load Distribution 22

2.3.3 Levels of Load Distribution 23

2.3.4 Dynamic Load Distribution 24

2.4 Constraint Satisfaction Problems 29
2.4.1 Definition of Constraint Satisfaction Problems 29
2.4.2 Solving Constraint Satisfaction Problems 30
2.4.3 Distributed Constraint Satisfaction Problems 32

3 Related Work 33
3.1 Load Distribution 33
3.1.1 Resource Monitoring 33
3.1.2 Mobile Agent Based Load Distribution 34
3.1.3 Load Distribution as Constraint Satisfaction Problems 35
3.2 Mobile Agents in Embedded Environments 36

4 The SmartCam Project 37
4.1 Hardware Architecture 37
4.2 Software Architecture 38
4.3 Surveillance System Architecture 39

5 Design 41
5.1 Basic Agent Infrastructure 41
5.2 Load Distribution 43
5.2.1 Resource Requirements of Agents 43
5.2.2 Agent Placement: A Distributed Constraint Satisfaction Problem . . 44
5.2.3 Selecting a Solution Based Upon its Costs 45
5.2.4 Using Mobile Agents for Task Allocation 51
5.2.5 Inter-Cluster Communication 53
5.2.6 Event-Based Redistribution of Load 56
5.2.7 Comparison With Other Load Distribution Systems 60

6 Implementation Aspects 61
6.1 Diet Agents Platform 61
6.1.1 Available Functionality 62
6.1.2 Extensions for the SmartCam Project 63
6.2 Implementation of the Distributed CSP 65
6.2.1 Computing the Agent Placement for Every Node 66

6.2.2 Merging Solutions . . . 68
6.2.3 Enhancing Java Object Serialization . . . 70

7 Evaluation **71**
7.1 Testbed . . . 71
7.1.1 Java Runtime Environments . . . 71
7.1.2 Code Size . . . 73
7.1.3 DSP Applications . . . 73
7.1.4 Migration Performance . . . 74
7.2 Evaluation Scenarios . . . 75
7.2.1 Single Cluster Scenario . . . 75
7.2.2 Overlapping Clusters Scenario . . . 84
7.2.3 Cost Factors . . . 88

8 Conclusion and Future Work **89**
8.1 Future Work . . . 89

Bibliography **91**

List of Figures

2.1 The structure of MASIF compliant agent systems. 15

2.2 The FIPA agents platform with its components and communication paths. . 16

2.3 Relationship between the Java 2 Standard Edition (J2SE) and the Java 2 Micro Edition (J2ME). 18

2.4 The profiles for the Connected Device Configuration (CDC) and the Connected Limited Device Configuration (CLDC). 19

2.5 The JamaicaVM build process. 21

2.6 Components of dynamic load distribution systems. 23

2.7 The different levels of load distribution. 24

2.8 Graph coloring - a constraint satisfaction problem. 30

2.9 Variable based decomposition of constraint satisfaction problems. 32

2.10 Domain based decomposition of constraint satisfaction problems. 32

4.1 The SmartCam prototype. 38

4.2 The SmartCam hardware architecture. 39

4.3 Deployment of SmartCams in clusters. 40

5.1 The SmartCam agent infrastructure. 42

5.2 The user assigns several agents to the cluster. 51

5.3 Load distribution worker agents are created on every node. 51

5.4 The initiator selects pairs of workers for merging. 52

5.5 Worker agents migrate to the host of their merge partner. 52

5.6 Two overlapping clusters. 53

5.7 Flowchart of the load distribution process. 55

6.1 The `CommunicationAgent` provides remote communication mechanisms. . . 63

6.2 The SmartCam console and visualization tool. 65

6.3 Generation of partial solutions of the CSP. 66
6.4 Application of backtracking to solve constraint satisfaction problems. 67
6.5 Merging two solution sets of the constraint satisfaction problem. 69

7.1 A single cluster containing three nodes. 76
7.2 Running times until the final solution is received at the initiator. 78
7.3 Three overlapping clusters on four nodes. 84

Chapter 1

Problem Domain and Motivation

The volume of traffic on roads and highways continuously increases. Intelligent traffic surveillance and guidance systems are required to handle this increase and to ensure the safety of road users. The key to intelligent traffic surveillance systems are smart cameras. Such smart cameras relieve systems operators by preprocessing and analyzing video data, warning of critical events and taking over routine jobs.

1.1 Smart Cameras

Processing power of embedded systems is increasing rapidly, opening up a wide range of possible new applications. First and second generation traffic surveillance systems essentially employ analog video cameras to observe and capture scenes. The digitization and processing of the content is typically done at some centralized backend system [RRF01].

Today's technology not only allows for digital capturing and on-board video compression, but offers enough resources to add additional functionality to surveillance cameras. Such third generation cameras, also referred to as smart cameras, provide sufficient computing power to do on-board analysis of the captured content.

In practice, surveillance cameras are not deployed isolated, but multiple cameras are installed. A typical application is the observation of a highway section such as a tunnel. Although today's cameras are equipped with a high amount of processing power, they are still limited in resources. In addition to that, other constraints such as low power consumption and low heat dissipation are important topics for embedded systems. Having more than one camera observing the same section opens the possibility to distribute tasks among those cameras to achieve a better overall utilization of system resources. Such a set of cameras that shares a number of tasks is referred to as a cluster.

The types of tasks in traffic surveillance cover a broad spectrum. They range from traffic statistics gathering including lane occupancy, vehicle count or average speed, to the detection of stationary vehicles, fire and smoke, traffic jams and lost cargo. For most of these tasks it is not critical to be executed on a specific camera. It is sufficient if such a task is executed on one of the cameras of the cluster, all observing the same scene. Another

type of task that is of great interest, is the tracking of vehicles such as dangerous goods transports. The handling of such tracking applications, that follow the tracked vehicle from camera to camera is covered in [Qua05].

The goal of the *SmartCam* project [BRS04,RBB+04] is to implement such an intelligent camera. The basis of the project is a multi-processor platform equipped with an XScale base board and multiple digital signal processors used for video processing. The hardware and software architecture of the prototype system is covered in chapter 4. The system is not designed to use a central host for coordination, but to operate fully distributed. This implies, that the management of the clusters and the assignment of tasks to specific cameras of a cluster, must be implemented in a distributed fashion.

1.2 Load Distribution Using Mobile Agents

The mobile agents software development paradigm has become a major research topic in recent years. A mobile agent is a piece of software that is capable of migrating from one machine to another. In addition to that, an important topic in agent development is communication between agents. Both aspects, the migration and the communication capabilities of agents, make the agent paradigm an excellent choice for the development of the load distribution system of the SmartCam project. This load distribution system not only has to organize and implement the allocation of tasks to cameras of a cluster, it also has to handle dynamic changes of the load of a cluster at runtime.

The tasks executed by a cluster are realtime tasks. This means that the tasks have specific resource requirements. The load distribution mechanisms must ensure that these requirements are met. Otherwise, the realtime constraints are violated and the tasks are not able to run properly. This thesis addresses the design and implementation of the load distribution system on top of the SmartCam prototype platform.

1.3 Thesis Organization

Chapter 2 provides a survey of the technologies and concepts that form the background of this work. The mobile agent paradigm is covered in section 2.1 followed by a discussion of Java environments suitable for embedded systems in section 2.2. Section 2.3 covers terminology and techniques of load distribution systems. The chapter is concluded by an outline of the structure of constraint satisfaction problems in section 2.4.

Chapter 3 focuses on related work in the field of load distribution based on mobile agent technology. It also covers the applications of agent systems in embedded environments.

Chapter 4 presents the hardware and software architecture of the SmartCam prototype platform.

Chapter 5 provides a detailed discussion of the design of the load distribution system for the SmartCam project. It covers the basic infrastructure (section 5.1) and discusses the mechanisms designed for optimal task placement in sections 5.2 to 5.2.3. Section 5.2.4 describes how these concepts are mapped to mobile agents.

Chapter 6 selects specific aspects of the implementation of the load distribution system that are discussed in more detail. These topics include extensions of the mobile agent platform and the implementation of the constraint satisfaction problem solving and merging algorithms.

Chapter 7 presents a discussion of test results and measurements conducted with the load distribution system on the SmartCam prototype.

Finally, chapter 8 presents a summary and conclusion of the work. Possible future extensions and improvements are discussed.

Chapter 2

Background

This chapter provides a survey of the theoretical background and the technologies that are relevant to this work. Section 2.1 starts with an introduction to the concept of software agents and presents the basic elements of agent systems. The majority of agent systems is developed by using the Java programming language. Section 2.2 discusses Java environments available for embedded systems. It is followed by the fundamentals of the second main subject of this thesis, load distribution mechanisms, in section 2.3. Many problems in computer science, including load distribution, can be modeled as constraint satisfaction problems. Section 2.4 provides the concepts of constraint satisfaction problems and shows how they can be solved.

2.1 Agent Based Software Development

Agent based software development has become a major research topic in recent years. In literature, many definitions of the agent concept exist. This section tries to identify the different definitions, followed by a discussion of technologies and standards related to agents. It concludes with a summary of the advantages and disadvantages of the agent paradigm.

A basic definition of the term agent is the following:

> Agent: A representative who acts on behalf of other persons or organizations. [AgD]

Although this definition is not focused on computer science, it provides a first insight into what the agent notion in information technology is about: An agent is some type of entity, typically a software program, that performs one or more tasks on its own. It acts on behalf of a user and pursues predefined goals. This suggests that agents contain some form of artificial intelligence. While this is true of some implementations of agents, it is not an absolute requirement. Other agent definitions focus on mobility and distribution. Some researchers see the agent concept as an extension to object oriented software development and propose a new type of software engineering approach, an agent oriented development.

2.1.1 Intelligent Agents

The concept of intelligent agents has its origin in the field of artificial intelligence and knowledge engineering.

An intelligent agent, as defined in [Tve00], is a piece of software that is *authorized to act for, or in place of another*. The term intelligent in this context is summarized as follows: *intelligent behavior is the selection of actions based on knowledge*. Intelligent agents have a number of properties[WJ94]: (1) They are autonomous and do not require human intervention. (2) They have full control over their internal state and their actions. (3) Agents can either act "proactive" or respond to events of their environment. (4) Intelligent agents are able to interact with other agents using a common language and they can migrate between hosts. Other characteristics of intelligent agents are that they do not deliberately provide false information or do not pursue goals conflicting with those defined by the user.

Agent-Oriented Programming (AOP) or Agent-Oriented Software Engineering (AOSE) is described as an extension to the widely established object-oriented programing (OOP) paradigm [Tve00, Sho93]. An object in OOP consists of several attributes and methods that operate on those attributes. In AOP, agents are the equivalent to objects, but agents are active entities which act autonomously. They are equipped with a mental state consisting of beliefs or commitments.

Agents in agent-oriented software engineering have a number of features [Jen99]. They are seen as problem solving entities with clearly defined interfaces. In a typical scenario, an agent is embedded in an environment it only has partial control over and limited knowledge about. Their goal is to fulfill a set of defined objectives by acting autonomously. Many situations and problems require not only one agent, but a larger number of them. Having more than one agent, poses the problem of inter-agent communication. Agents often have to work in a cooperative manner to reach their individual objectives. This aspect of agent programming is often subsumed by the term social interaction. Since it is not possible to predict all possible agent interaction scenarios, a high-level agent communication language (ACL) is required.

Intelligent agents and agent-oriented programming have a strong focus on artificial intelligence aspects. These aspects, however, are not very important to many practical applications of agents. Hence, in literature agents are often referred to as software agents instead of intelligent agents.

2.1.2 Distributed and Mobile Agents

An important topic in the field of agent based development is the distribution and mobility of agents. Being mobile means that an agent is able to move from one host to another. This requires that the agent is able to suspend its execution, save its current state, migrate to the foreign host, restore its state and resume execution. There are two different types of agent migration [IKWK00]:

- *Strong Migration.* If an agent decides to move to a remote host, its current state has to be captured and transferred. This state not only includes the agent's code and object data, but also the execution data. This involves the stack, the program counter and references to other objects including recursive references. Strong migration means high complexity, because it requires mechanisms to freeze the thread of the agent and capture a dump of the frozen state. This can only be done from the outside using a debugging interface, but not from within the application.

- *Weak Migration.* Since strong migration is difficult to implement, most agent systems use a simplified migration technique, the so called weak migration. A very common form of implementation is to only migrate code and object data. The stack and the program counter are not transferred. A special suspend method of the agent is called, giving it the chance to store required data into member variables before the agent is stopped. After the agent has been moved, a resume function is executed and the agent can continue its work.

In case of multi-threaded agents, the agency has to make sure that all threads, created by the agent, are suspended and migrated as well.

Not all agents are necessarily mobile. A common application of agents are sensor networks [MJT+01]. In such a scenario, agents are distributed among the nodes of a network where they monitor certain parameters. If these numbers reach critical values, the agents can either report them to a central decision maker or act autonomously. An agent could modify parameters on its node or get in contact with other agents by means of a message interface. In this way agents are able to handle situations on their own without the requirement to be mobile.

Stationary agents are easier to design than mobile agents. Depending on the problem, the additional effort required for mobile agents can increase the performance of a system. If an agent is able to travel to a remote host, it is able to conduct a more intense dialog with the host than it could do via messages over the network [LO99]. This approach not only allows the agent to make use of local resources of a host, but it also can reduce the network traffic significantly.

2.1.3 Agent Technologies

As most technologies in computer science, agents and agent systems are built upon and surrounded by a number of other technologies. Agent environments, also called *agencies*, provide a number of basic services which an agent requires to achieve its goals. To be able to integrate agent environments from different manufacturers, interoperability standards were defined such as the Mobile Agent System Interoperability Facility (MASIF) by the Object Management Group (OMG). Related to those standards are agent communication languages allowing vendor-independent inter-agent communication.

2.1.4 Agent Environments

Agencies, as well as the agents, should be able to execute on a wide range of different platforms, including different hardware architectures or operating systems without the need of re-compilation. This requirement narrows the choice of programming environments to either scripting languages that are interpreted at runtime, or platforms such as Java or Microsoft .NET [PCVM99]. In case of Java and .NET the source code is compiled to a special type of intermediate code, which then is executed without modifications on every platform the runtime environment is available for. The execution is either done by interpreting the intermediate code or with a just-in-time compiler (JIT). Employing a just-in-time compiler provides higher performance. While SUN Microsystems is covering a wider range of platforms with Java, Microsoft .NET is only available for the Microsoft Windows platform. The Mono project [Mon] aims to provide a free, open source implementation of the .NET framework for other operating systems, such as Linux, MacOS and other UNIX derivatives. With the further development of Mono, the .NET environment might become a serious choice for agent platforms in the near future.

The following services, required by agents to accomplish their goals, are provided by an agency:

- *Naming and Lookup.* To allow agents to locate other agents and agencies, it is required that they can be uniquely identified. For this purpose a distinct label has to be assigned to a newly created agent. In addition to that, some type of usually centralized registry mechanism is required, where agents and available services are listed. This allows agents to locate other agents or services, which they require to accomplish their objectives.

- *Communication and Messaging Infrastructure.* Whenever an agent has located a service or another agent providing the required functionality, it has to communicate with it. For this purpose, an agent environment has to provide a common infrastructure that allows agents to exchange messages. This can be done in a proprietary fashion or in compliance with agent communication language standards, such as the FIPA agent communication language (ACL) (see also section 2.1.5). The components used to implement the messaging infrastructure are typically simple socket communication or advanced mechanisms such as remote procedure calls (RPC).

- *Migration Facilities.* For mobile agents the agency has to provide functionality that allows agents to migrate from one agency to another. This means that it must be possible to suspend the execution thread of an agent, capture the current internal state and move it to the destination. At the new host the agent will be restored and can resume its operation.

- *Persistence Services.* Agents are able to modify their goals at runtime, gather knowledge throughout their lifetime or move to different hosts. In case of a system failure, such as a power outage or an agency failure, agents cannot easily be recreated by re-instantiating a certain class. To overcome the potential loss of valuable information, the agency can offer a persistence service. This means that agents are stored in a non-volatile storage either periodically, upon request or triggered by special events.

- *Agent Cloning.* Cloning in the field of agent technology means to create an identical copy of an agent. A new instance is created that has the same internal state as the original agent [SSCJ98]. The only thing in which the original agent and the clone differ is their unique identifier.

Agent and Agency Security In addition to the security risks familiar from conventional network based systems, the mobile agent concept introduces a number of additional security problems [GBH98]. Two fundamental kinds of problems can be identified: The first are threats for the hosting agency. An agent might abuse the resources of the agency and thus cause major harm to its host and to the other agents on that system. The other target of an attack are the agents. In case the agency has been corrupted by an attacker, it could modify or even destroy agents.

The following list provides a survey of the most important security threats.

- *Damage.* Without proper protection mechanisms, an agent has access to the file system of its host system and could therefore modify or delete data. A common solution of this is a monitoring facility that intercepts agent actions based on a set of permissions assigned to the agent. Likewise a corrupt agency could modify or delete an agent at will.

- *Denial of Service.* Not all harm caused by agents has to result in damaged files or data. An agent could consume large amounts of CPU time or saturate the local network connection by sending garbage. That way, the agency has difficulties to serve other agents. Having said that, an agency could just as well refuse to grant resources to an agent which would cause the agent to fail. Introducing CPU-time-, resource access- or network bandwidth limits can significantly reduce the threat of denial of service attacks by agents.

- *Data Confidentiality.* In e-commerce applications for example, confidential data is transmitted over the network. This data is part of the internal state of an agent. Counter measures, such as encryption, have to be taken to prevent the theft of such information.

- *Social Engineering.* A compromised agent platform could provide agents with incorrect information. That way, the creator of the agent could be tricked into giving away private information to a supposedly trustworthy agency.

An important mechanism to establish mutual trust is the introduction of certificates for agents and agencies. Both parties could be signed using public key cryptography, which permits to identify the other party unambiguously. Since agents change their internal state at runtime, they have to be re-signed when they leave the platform.

2.1.5 Agent Interoperability Standards

The idea of agent interoperability standards is to make agent systems from different vendors compatible. To achieve this, common aspects such as agent transfer, class transfer or agent communication need to be standardized.

Mobile Agent System Interoperability Facility – MASIF

The Mobile Agent System Interoperability Facility (MASIF) is closely related to the *Common Object Request Broker Architecture* (CORBA) [PPW02]. Both standards are defined and maintained by the Object Management Group (OMG[1]). The core component of every CORBA, and hence MASIF, compliant system is the Object Request Broker (ORB). The ORB handles all requests of clients. When a client object performs a remote call, it is unaware of the real location of the destination object. The ORB will look up the location of the called object and forward the call to it. ORBs communicate by using the Internet Inter ORB Protocol (IIOP). The IIOP is also standardized by the OMG which therefore enables interoperability of ORBs from different vendors.

MASIF is designed as a *CORBAfacility* and makes use of *CORBAservices*. CORBAservices offer basic services such as naming, object lifecycle management and object persistence. The term CORBAfacilities sums up application specific services. Figure 2.1 presents the components of a CORBA/MASIF setup and shows their relationship. The MASIF agent system provides the environment for creating, executing, migrating and destroying agents. An agent system can be divided into places and agents can be assigned to a specific place within the system. Multiple agent systems form regions.

MASIF is focused on the interoperability of agent platforms, but does not cover the issue of the programming language. It only ensures that agents, that have been developed for one agent platform, can also exist in another MASIF compliant agent platform if both platforms were developed using the same programming language. The following topics are addressed by the MASIF specification:

- *Agent Management.* A set of management functions that allow a system administrator to create, suspend or terminate agents is defined. Agent management should work the same way regardless of the platform vendor.
- *Agent Transfer.* Agents should be able to travel from one platform to another, no matter who designed and developed the system. Related to agent transfer is class transfer. To make agent interoperability possible, mechanisms have to be defined of how to find and fetch classes required to restore agents after migration. This MASIF service can be implemented using the CORBA lifecycle management facility.
- *Agent and Agent System Names.* To be able to locate agents or to execute administrative commands involving agents, a mechanism is required to assign globally unique names to agents and agent systems. The CORBA naming service already provides such a mechanism.

[1] Object Management Group (OMG) website: http://www.omg.org/

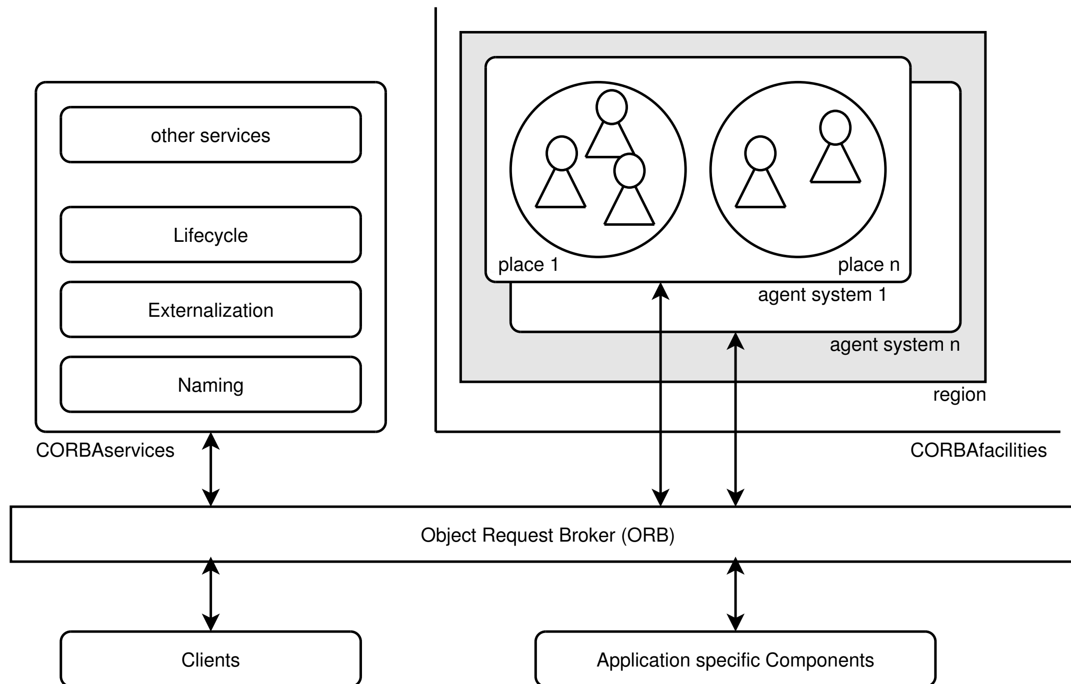


Figure 2.1: The MASIF agent systems are defined as CORBAfacilites. The MASIF standard makes use of standard CORBA services such as naming or lifecycle management.

Technically, the MASIF standard defines interfaces that allow agents and agent platforms to interact in a standardized way. The MAFFinder interface specifies a registry facility with operations including the (un-)registration of agents and agent systems, or the lookup of agents. MAFAgentSystem defines operations relevant to agent systems such as creating, suspending, receiving or terminating agents.

Foundation for Intelligent Physical Agents – FIPA

The Foundation for Intelligent Physical Agents (FIPA[2]) is an organization focused on developing standards for agent interoperability [PPW02]. The term physical in its name is misleading: when FIPA was founded, one of its goals was not only to develop software agents but also hardware agents, such as robots. Over time, the objectives have changed and FIPA is now entirely dedicated to software agents.

Having agent systems from multiple vendors, means that there must be an agreement of how these systems can inter-operate. The core topics covered by the FIPA standards are agent management, agent naming and lookup, and agent communication. The AgentManagement System (AMS), as defined by FIPA, provides basic functionality such as agent creation and deletion. It also maintains a directory of all agents currently executed on the system (white pages). The Directory Facilitator is a lookup service for agents (yellow pages). Agents can register themselves with the Directory Facilitator or search for ser-

[2]Foundation for Intelligent Physical Agents (FIPA) website: http://www.fipa.org

vices provided by other agents. Within a platform, agents interact using a vendor specific protocol, not specified by FIPA. For communication between agents of different platforms, Agent Communication Channels (ACC) are defined. The protocol used by the agents to communicate is called an Agent Communication Language (ACL). Resource management is done by the Agent Resource Broker. Figure 2.2 shows the components of a FIPA agent platform.

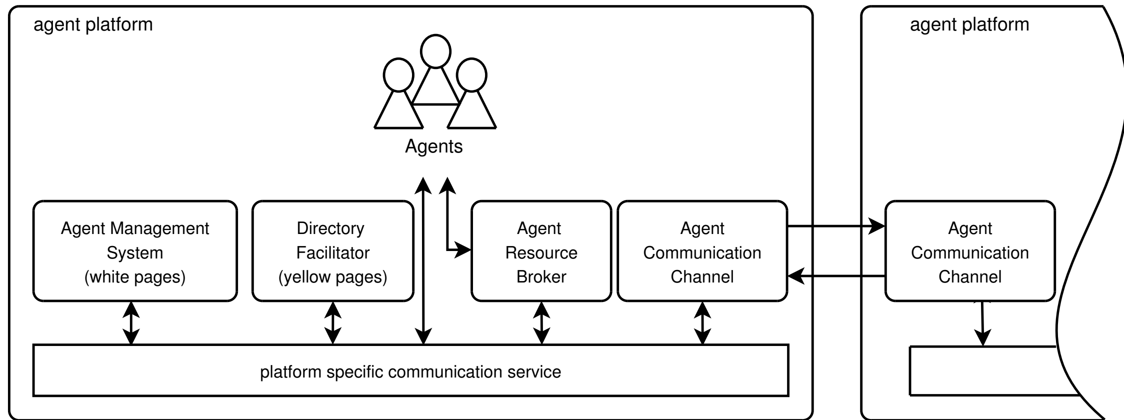


Figure 2.2: The FIPA agents platform with its components and communication paths.

In contrast to MASIF, FIPA tries to specify the entire agent infrastructure including agent communication. Agent mobility is supported by FIPA, but it is not as important as in MASIF. Both standards define an agent platform with its basic operations. The directory facilitator of FIPA is the equivalent to the MAFFinder in MASIF. Both define a service for agent lookup. The agent communication channel used by FIPA for inter platform communication can be compared to the object request broker of MASIF/CORBA systems. Because of the similarities of the two standards, OMG and FIPA decided to coordinate their efforts [PPW02].

2.1.6 Advantages and Disadvantages of Mobile Agents

As every other technical system, mobile agents have a number of advantages and disadvantages [LO99, Jen99, Vig04]. This section discusses both of those aspects.

- ⊕ *Network Load.* This is one of the most often mentioned benefits of mobile agents. When processing large amounts of data, such as results of database queries, it might be highly inefficient to transfer all the data over the network, especially if the agent only needs to extract and aggregate certain elements. It is much more efficient to transfer the agent to the remote location, let it do the data processing there, and then return with the results to its host of origin.

- ⊕ *Network Latency.* Especially in large networks latency becomes an issue in the field of real time applications. Instead of a central control station that sends commands

to clients periodically, those command sequences can be integrated into an agent. An agent instance is created by the control station, and is then sent to the client where it starts its operation. Since the agent containing the control information now is located directly at the client system, network latency is not of further concern.

⊕ *Autonomous Execution.* Since it is the nature of agents to act on their own, this opens up a number of different applications. A user is able to express a certain task which he hands to an agent for processing. The agent now works on its own to meet the targets without further interaction with the user. It is even possible for a user to disconnect from the network and collect the results the agent came up with at a later point. This feature is especially of interest for wireless applications with expensive or low quality network connections.

⊕ *Fault Tolerance.* Mobile agents are not bound to a specific host. In case the host, an agent is running on has to be shut down due to some external event, the agent can move to another host and resume its operation there. This assumes that there is time to forewarn the agent ahead of the event.

⊖ *Unpredictability.* By acting more or less on their own, the behavior of agents can be difficult to predict. It is the task of the agent developer to balance the proactive and reactive aspects of an agent. Being too reactive for example, combined with a highly dynamic environment, could lead to executing irrelevant tasks by reacting to every minor event. Generally it is nearly impossible to do any prediction about timing, migration or interaction patterns.

⊖ *Overhead.* Especially in small setups the usage of agents is not always an advantage over well proven concepts. The agency has to be running on every node of system, requiring a certain amount of system resources, which is of special concern on low end systems. If agents are used only to exchange a limited number of messages, without the need to migrate between hosts, a more light weight peer-to-peer or client/server model might be better.

⊖ *Bad Performance.* Although agent literature often claims the contrary, in many situations agents do not perform well. The advantage of lower network traffic is often more than compensated by the amount of data that needs to be transferred during agent migration.

⊖ *Difficult Testing and Debugging.* Mutli-threaded, distributed software is hard to debug. Adding the mobility aspect pushes the complexity even further.

⊖ *Missing Practical Experience.* Although the concept of agents has been known for more than a decade, it still lacks wide scale adoption. This results in missing practical experience compared to other, traditional, types of software engineering.

2.2 Java in Embedded Environments

Although the capabilities of embedded systems continuously increase, they are still limited in terms of computing power compared to general purpose computers. In addition to

that, other limitations regarding available memory, communication facilities and power consumption have to be dealt with. Therefore, a mobile agent system, deployed on an embedded device has to be as lightweight as possible. The same is true of the programming language and runtime environment, which is used for development.

For mobile agent applications, the Java programming language has become one of the most popular platforms for a number of reasons. Java code is not compiled to native machine code but to an intermediate code, also known as byte code. This byte code is executed on a virtual machine, called the Java virtual machine (JVM). This allows the deployment of Java applications on every hardware platform and operating system, a JVM is available for. The Java byte code is either interpreted by the virtual machine or compiled to native code using a just-in-time (JIT) compiler. Other aspects that make Java suitable for mobile agent applications are its fully object oriented nature, built in serialization and networking capabilities, and its overall ease of use.

The object serialization of Java defines the process of converting the internal state of an object into a byte or string representation. This serialized object can be transferred over the network, or written to some permanent storage. The reconstruction of the object from its serialized form is known as object de-serialization. In contrast to other programming languages, Java does not require the developer to write serialization and de-serialization code for every class. Object serialization is a standard feature of the Java platform and can be used for agent migration.

2.2.1 Java 2 Micro Edition

The Java 2 Standard Edition (J2SE) is typically too large and resource consuming for embedded devices. For this reason the Java 2 Micro Edition (J2ME) was developed with small, low-resource devices in mind [Ort04].

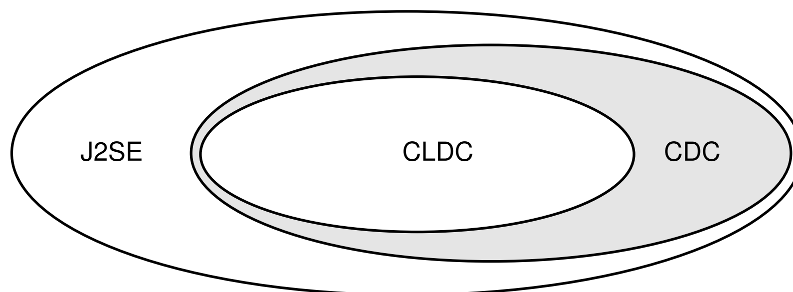


Figure 2.3: Relationship between the Java 2 Standard Edition (J2SE) and the Java 2 Micro Edition (J2ME) in its two flavors: Connected Limited Device Configuration (CLDC) and Connected Device Configuration (CDC).

The basic building blocks of the J2ME are called configurations. Configurations consist of a virtual machine and a set of core classes surrounding it. There are two different configurations available:

- *Connected Limited Device Configuration (CLDC).* This configuration is intended for low power devices with slow, non-permanent network connections. It offers a small memory footprint and is designed for slow CPUs.
- *Connected Device Configuration (CDC).* This configuration is designed for more powerful devices such as Personal Digital Assistants (PDAs). As shown in figure 2.3, the CDC is a super set of CLDC.

Both, the CLDC and the CDC virtual machines, do not have a just-in-time compiler, but use an interpreter. On top of the configurations there are so called profiles. These profiles are extension packages that provide additional functionality known from the J2SE class library. For the CLDC, there are two profiles available (see figure 2.4). The first one is called *mobile device information profile* (MIDP) and is targeted towards devices with graphical user interfaces. The second one is the *information module profile* (IMP) and is a subset of MIDP that omits the graphical elements.

A major drawback of the CLDC is the lack of serialization and reflections support, and the missing Java Native Interface (JNI). The JNI is used to make functions of the underlying system available to Java applications. Object serialization can also be done by hand [Gig02], but this means that the application developer would have to write serialization code for every class, whose instances are transmitted over the network. This renders the CLDC inapplicable for mobile agent applications.

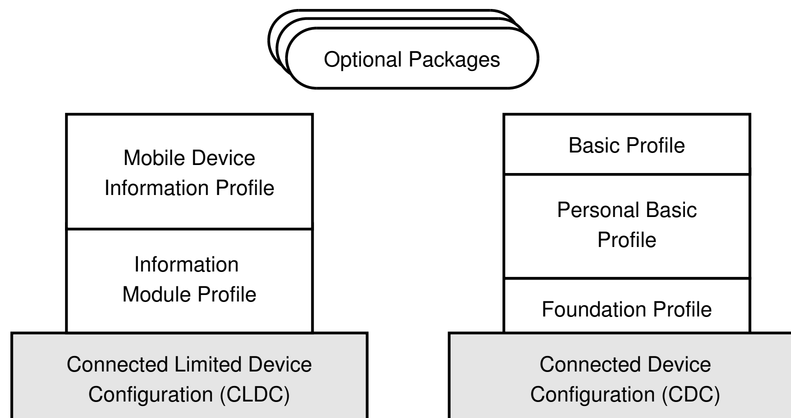


Figure 2.4: The profiles for the Connected Device Configuration (CDC) and the Connected Limited Device Configuration (CLDC).

For the connected device configuration, there are three profiles available (see figure 2.4). The basic one is called the *foundation profile* (FP). It includes many of the features supported by the Java 2 Standard Edition including the reflection and serialization APIs as well as JNI support [Sun01]. However, it does not include the Advanced Windowing Toolkit (AWT) or other GUI components. Basic AWT support is added by the *personal basic profile* (PBP) running on top of the foundation profile. A further extension is the *basic profile* (BP), which adds additional GUI components.

Finally, there is a number of optional packages available, to add functionality not found in any of the profiles. Among those add-ons there are a bluetooth package, a wireless messaging API as well as Remote Method Invocation (RMI) and database access APIs.

2.2.2 JamaicaVM - Ahead of Time Compilation

Most Java virtual machines either interpret the Java byte code or use a just-in-time compiler to compile the byte code to native code at execution time. JITs not necessarily compile the entire program, but only those parts frequently executed.

Another approach is ahead of time compilation. This approach is taken by the commercial JamaicaVM [Aic04] from Aicas GmbH. JamaicaVM is especially targeted at real time and embedded systems. As other VMs, also JamaicVM has an integrated interpreter that is optimized for performance. A JIT compiler is not applicable for realtime systems since it introduced unpredictable delays for the just-in-time compilation process. JamaicaVM solves this problem by ahead of time (AOT) compilation of performance critical code. It generates intermediate C-code from the Java byte code, which is then compiled for the target platform. Java byte code is relatively small compared to native compiled code. For embedded systems not only performance and real time constraints are important, but also the size of resulting executable. To reduce the size of the executable, JamaicaVM is able to mix interpreted code and pre-compiled code. The JamaicaVM documentation claims that already the ahead of time compilation of a relatively small percentage of the code, can result in execution times comparable to those of hundred percent native compiled code.

To decide which parts of the code should be compiled using AOT compilation, JamaicaVM provides a profiling mechanism. The application is first compiled in a special profiling mode. The resulting application is the executed and a profiling log is generated. The profiling data contains information which functions are called frequently and how much execution time is consumed by the individual parts of the application. In a second compiler run, this information is used to generate an optimized executable. Parts of the application that have proven to be performance critical during profiling, are compiled using the AOT compiler. Other parts of the application will be interpreted. The user is able to specify the percentage of the code that should be compiled ahead of time. The higher this percentage is, the larger the resulting executable file will be. A complete ahead of time compilation of the entire application is not required. Already relatively small percentages of AOT compiled code result in a significant speed up of the application, while the size of the executable remains small. The Jamaica build process is presented in figure 2.5.

JamaicaVM not only tries to address performance issues of embedded systems, but also is designed with real time applications in mind. The biggest problem of Java applications in realtime environment is the automatic garbage collection mechanism of Java. Periodically, memory that is not referenced by the application program is detected and freed. The freed memory blocks are then compacted to avoid fragmentation of the heap memory. The garbage collector is usually implemented as thread of its own, blocking all threads of the user's application while the garbage collection is running. This is problematic for realtime application since it cannot be predicted when the garbage collector becomes

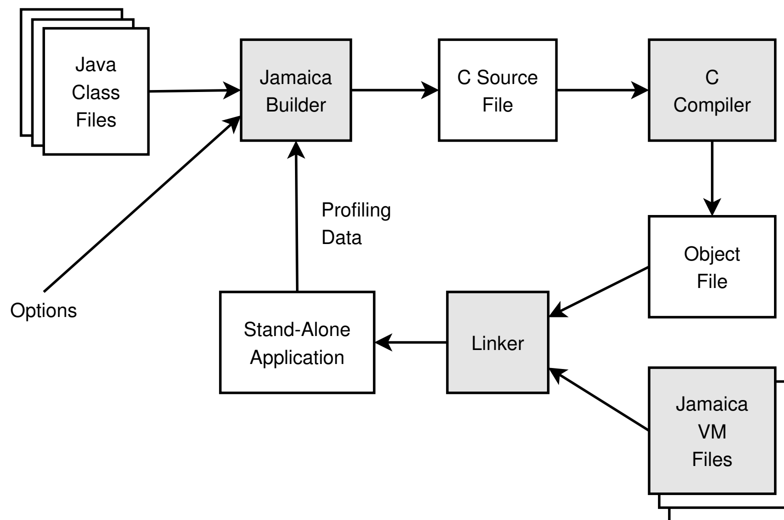


Figure 2.5: The JamaicaVM builder generates intermediate C source code from Java byte code. The C code is then compiled to native code.

active and for how long it will block the application threads. To overcome this problem, the Real-Time Specification for Java (RTSJ) was defined. RTSJ introduces threads that cannot be preempted by the garbage collector because they are given a higher priority. Those realtime threads cannot allocate objects on the normal heap. This complicates the communication between realtime threads and normal threads. JamaicaVM does not require the distinction of realtime and non realtime threads [Hol04]. The activation of the JamaicaVM garbage collector is predictable. The JamaicaVM garbage collector is always preemptable by other threads.

2.2.3 Free Java Environments

In addition to the official Java environments, provided by SUN Microsystems, and other commercial implementations such as JamaicaVM, there is a number of open source Java environments. While many of them are experimental projects or implement only a subset of the virtual machine specification, some implementations are mature enough to be considered as an option for embedded systems. The most promising free Java virtual machines are:

- *JamVm* [Lou] is developed by Robert Lougher and provides an interpreter for PowerPC, Intel x86 and the ARM architecture. The ARM architecture is supported in little and big endian versions.
- *Kaffe* [Kaf] is a clean room implementation of the Java virtual machine. It was originally developed as a commercial product but was released to the open source community when the company was closed down in 2002. Kaffe offers a just-in-time compiler and an interpreter for many operating systems and hardware architectures.

For the XSclae big endian architecture, as used in the SmartCam project, only the interpreter is available.

- *SableVm* [Sab] originates from a Ph.D. research project. It offers an interpreter for Intel x86, PowerPC, Alpha, Sparc, ARM and other hardware architectures. A just-in-time compiler, SableJIT, is currently in development.

A complete java environment not only requires a virtual machine, but also an accompanying class library. An open source implementation of this library is provided by the GNU Classpath project [GNU].

2.3 Load Distribution

Load distribution tries to improve the performance of a system by transferring tasks from heavily loaded machines to those that are idle or lightly loaded [SKS92]. The goal is to achieve better response times and resource utilization of the overall system. After discussing the requirements for load distribution in section 2.3.1, the different types and levels of load distribution are presented. Section 2.3.4 provides an outline of the elements of dynamic load distribution systems.

2.3.1 Requirements for Load Distribution

A fundamental requirement for load distribution is the divisibility of a given problem into a number of sub problems, that can be executed in parallel on different execution units. The lowest granularity level is represented by an instruction. Instruction based parallelism can be exploited in tightly coupled systems and involves the scheduler of the operating system. Load distribution in user space typically deals with more coarse mechanisms. A well-known concept is the Remote Procedure Call (RPC) paradigm. It allows a procedure to be executed on a remote machine, thereby reducing the workload of the calling machine. Process parallelism represents the coarsest form of parallelism. It deals with entire processes that are loosely coupled, or independent of one another. These processes can be assigned to different machines for execution. For communication, message passing mechanisms can be employed. The level of granularity for task decomposition depends on the architecture of the system and the problem to be solved. Not every problem is well suited to be divided into smaller portions. In addition to that, load distribution becomes more complicated the smaller the granularity gets. Hence, many load distribution systems use entire processes as smallest distributable units.

2.3.2 Types of Load Distribution

Load distribution mechanisms can be, as show in figure 2.6, divided into two main groups:

- *Static Load Distribution* allocates tasks to specific machines already at design time. It requires a priori knowledge about available system resources, as well as processing

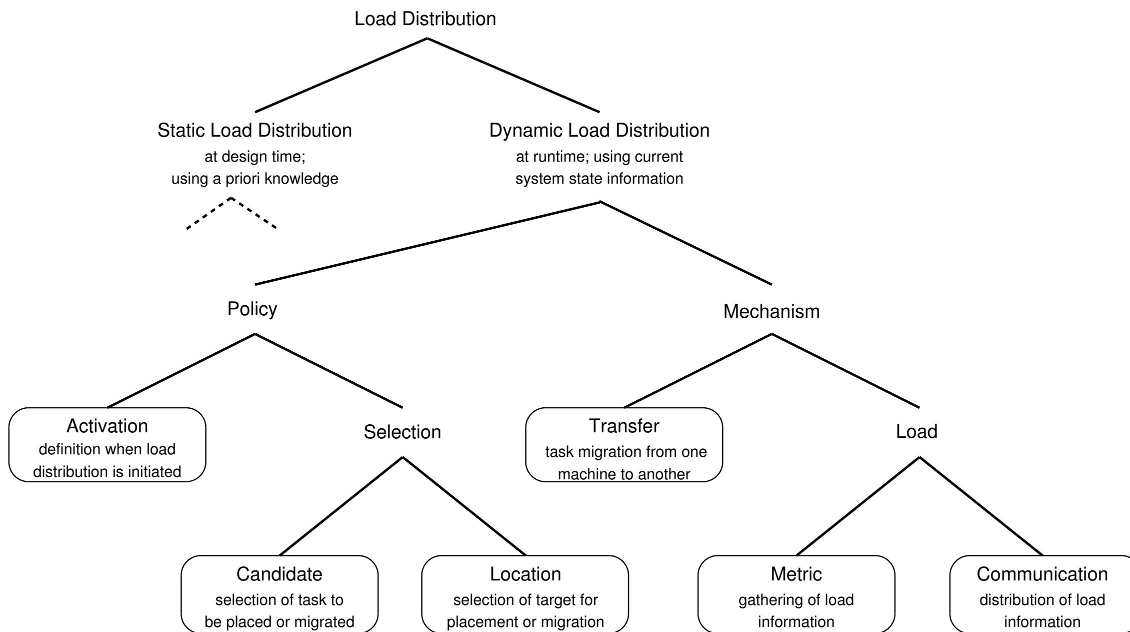


Figure 2.6: Load distribution can be divided into static and dynamic load distribution. Dynamic load distribution relies on a number of mechanisms and policies defining the behavior of the system.

time and resource requirements of all tasks. This information can be obtained by task profiling or analyzing critical paths of execution. Runtime load changes caused by processes which are not under control of the load distribution system or outside events, cannot be dealt with.

- *Dynamic Load Distribution* is done at runtime by monitoring the current state of the system and distributing tasks accordingly. Thus, unpredictable events and load changes are taken into account. This allows a better utilization of the available resources. An extension of dynamic load distribution is *adaptive load distribution*. It provides the mechanisms to modify the distribution policies depending on the current state of the system. A policy that is working well for low and medium loaded systems might not be equally well suited for systems with higher load.

For large systems, it is not practicable to use static load distribution. As soon as the system involves dynamic, unpredictable creation and addition of processes, or processes with requirements not known in advance, dynamic load distribution mechanisms are required. The following sections will focus on aspects related to dynamic load distribution.

2.3.3 Levels of Load Distribution

Load distribution cannot only be divided into static or dynamic load distribution, but also the level of load distribution can vary. A common classification splits load distribution into load sharing and load balancing. Load sharing denotes the distribution of load among

a number of machines. No statement is given about the load levels of individual machines. Load balancing is described as the effort to distribute load evenly among all participating machines.

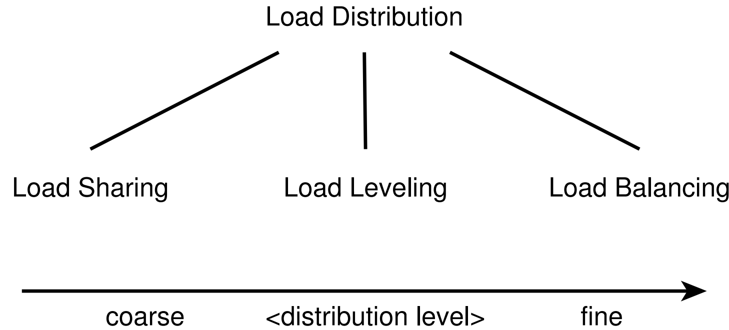


Figure 2.7: Relationship between Load Distribution, Load Sharing and Load Leveling and Load Balancing.

[Bub96] presents more restrictive definitions of load sharing and load balancing and introduces a third type, namely load leveling (see figure 2.7). The following definitions will be used throughout this thesis:

- *Load Sharing* requires that only idle machines are considered in the process of load distribution. If a machine is already executing a task, no other task will be assigned to it.
- *Load Leveling* not only utilizes idle machines for task assignment. The goal is to avoid overload situations on all machines of the cluster, while utilizing the available resources more efficiently than with load sharing. Load leveling is done periodically.
- *Load Balancing* seeks to evenly distribute the load among all machines of the system. It is evident that the load can hardly be totally equal on every machine. A certain range for the load to be in, is defined. In contrast to load leveling, load balancing is done continuously until the load on all machines is within this predefined range.

2.3.4 Dynamic Load Distribution

To decide if a task, running on a particular machine, should be transferred to some other machine of the system, dynamic load distribution considers the current state of the overall system. This allows a better utilization of system resources, since short term load fluctuations can be considered. In addition to that, the number and types of tasks do not have to be known at design time. In contrast do static load distribution, additional overhead is introduced for gathering and analyzing status information of the participating machines.

In literature, several core components of dynamic load distribution systems have been identified [RH00, CWD02, SKS92]. The *information gathering* component is responsible for obtaining and distributing load information. The process of *selecting a suitable task* to

be moved to another machine, and the *choice of an adequate new location* for the task are triggered by an *activation policy*. For the movement of the tasks, some form of *transfer mechanism* is required.

This classification is refined and extended in [Bub96]. The identified building blocks of dynamic load distribution systems are depicted in figure 2.6. Load distribution policies specify (1) how and when load distribution mechanisms are activated, (2) how a suitable candidate to be moved to a remote machine is selected, and (3) how this remote location is determined. Load distribution mechanisms provide facilities that allow the (4) transfer of tasks from one machine to another. To be able to measure load, (5) a load metric has to be implemented and the measured values have to be (6) communicated to all participating machines. These policies and mechanisms will be explained in the following sections.

Activation Policy

With dynamic load distribution, the state of the system is continuously monitored. Based on this information, a decision has to be made when to start the selection of suitable candidates and their new location. A typical implementation of such a mechanism is a threshold value. When the load of a machine reaches this value, it starts to offload work to remote machines. The definition of a lower bound can be used to trigger receiver initiated load distribution where an underloaded machine tries to relieve overloaded machines by taking over some of their tasks. A combination of one or more different thresholds can be used to implement an adaptive system behavior [SKS92].

Computing power of workstations and of personal computers has greatly increased over the past years. These machines typically spend a great deal of their time being idle. A number of efforts, such as [SGB01] and [DH02], try to take advantage of these unused resources. The success of such Networks of Workstations (NOWs) highly depends on the acceptance and participation of the owner of the individual machines. For these reasons, machines only take part in load distribution when they are idle. As soon as the owner of the machine starts to use it again, all foreign work has to be removed again. This policy is know as ownership policy.

Candidate Selection Policy

The selection of a suitable candidate to be executed on a remote machine is a central aspect of load distribution systems. A simple approach, which does not require any knowledge about the behavior of tasks is random selection. This assumes, that all tasks are equally well suited to be executed on a remote machine. In a typical system, there are tasks that cannot be moved to any other machine since they require access to local resources, such as hardware components of the local machine. Other tasks can, but should not be moved to another machine, since this would result in significant additional overhead. An example for this class of tasks are those communicating a lot with local objects. If such a task is moved to a remote machine, the communication is done over the network instead of locally. To choose a suitable candidate for remote execution, knowledge about the task, its requirements and its behavior is required. Such knowledge can be available statically

or gathered dynamically. Static information has to be collected at design time of a task and stored in a database. This information is then used by the system at runtime to select a suitable candidate for remote execution. If providing static information about tasks is not practical, it has to be gathered at runtime. To obtain this type of dynamic information, the behavior and resource requirements of a task have to be monitored while it is executing. For the initial placement of a new task, static information has to be used. If no static information is available, random placement can be used. After the task has been monitored for some time, it can be transferred to a more suitable machine.

Another aspect to be considered is, by whom a suitable candidate is selected. One approach is a central decision maker that is aware of all machines and the tasks running on them. This central station can then decide if a task should be transferred from one machine to another. Having one station with a complete view of the entire system, makes distribution of load easier while introducing a potential single point of failure. To overcome this problem, a more distributed selection mechanism can be used, where every machine decides on its own which task should be transferred. An even further decentralized mechanism allows the tasks themselves to choose if they want to move to another machine.

Location Selection Policy

The location selection policy is responsible for the selection of a suitable new location, for a selected candidate to be moved to. Different types of location policies can be distinguished by the component that initiates the location selection.

- *Sender initiated* location selection is triggered by the activation policy. A machine experiencing a high load situation decides to offload a task to another, lighter loaded machine. After having selected an appropriate task, a suitable target system has to be found. This is based upon the load information from the load communication mechanism. After a new location for the task was found, it is transferred and resumes its execution at the new host.

- *Receiver initiated* location selection is the counterpart to sender initiated location selection. Finding a suitable new location for a node can involve a number of computations as well as remote communication. If a system, as described in sender initiated location selection, is already experiencing a high load situation, the additional load generated by location selection might not be acceptable. Receiver initiated location selection presents an alternative to overcome this problem. In such a setup, an underloaded machine tries to find an overloaded machine. If such a machine is found, the candidate selection policy of the overloaded machine can select a task which is then moved to the underloaded machine.

- *Symmetrically initiated* location selection tries to combine the advantages of both, the sender and receiver initiated location selection. For low load situations of the overall system, the sender initiated way is more effective since the number of nodes with low load is higher. It will not take long to find a suitable machine as a target

for the transfer. For a high overall load, the receiver initiated approach is more successful because there are more machines that potentially want to offload a task.

A possible implementation of a symmetric location selection policy is presented in [SKS92]. Instead of using a single upper threshold for the activation policy, two different thresholds are defined: a lower and an upper threshold. In case the load of the system drops below the lower threshold, the receiver initiated location selection is used. If the upper threshold is exceeded, the sender initiated location selection is activated.

- *Centrally initiated* location selection is based on a central machine that selects an overloaded source and an underloaded destination machine. The source machine then selects a suitable candidate which is moved to the destination machine. The advantage of a central dedicated machine for load selection is its complete knowledge of the overall system state. Such an approach can easily be combined with a centralized load communication mechanism. The disadvantage is that the central machine not only forms a potential performance bottleneck but also a single point of failure.

Transfer Mechanism

Two fundamentally different approaches can be identified depending on when load is distributed: initial placement of tasks and task transfer. Initial placement takes place before the execution of a task is started. A suitable machine for the task is chosen by the location selection policy. Then the task is transferred to this machine and its execution is started. The task remains on the same machine until it is completed. If the system supports the transfer of tasks, it is possible to suspend the execution of a task and move it to another machine where its execution is resumed. This allows the load distribution system to react to changing resource requirements of a task or load changes of the hosting machine. To implement task transfer, a number of existing technologies can be used:

- *Remote Procedure Call* (RPC) represents a mechanism that allows the client machine to execute procedures on a remote machine. The name of the procedure to be called, along with all required parameters is sent over the network. The Java programming language provides a similar mechanism, called Remote Method Invocation (RMI).
- *Web Services* are functions that can be programmatically invoked over the Internet [Rym03]. They can be seen as an advanced version of the RPC paradigm which is designed to be used in web environments. The Simple Object Access Protocol (SOAP) defines a mechanism to send XML messages to remote hosts. These messages contain a web service request. The web service infrastructure also provides a description language for web services (Web Service Description Language – WSDL) and a lookup service (Universal Description, Discovery and Integration – UDDI). The key benefit of web services is their interoperability. Since all protocols make use of XML, they can easily be implemented and deployed regardless of the operating system or the programming language.

- *Common Object Request Broker Architecture* (CORBA) [GT00] not only allows to call remote procedures, but provides a distributed object architecture. In CORBA, a client can obtain a stub of a remote object. It then can interact with this stub as if it would directly interact with the object itself. All calls to the stub are forwarded to the real object over the network. Interfaces of CORBA objects are described using the Interface Description Language (IDL). The object stubs are generated from this interface description using an IDL compiler. This way, clients written in any programming language can access CORBA objects. The only requirement is that an IDL compiler for the client programming language is available.

- *Mobile Agents* can be suspended and migrated to another host. At the destination, the execution is resumed. This makes mobile agents very well suited for dynamic load distribution.

Load Metric Mechanism

The load metric mechanism is one of the most important parts of a load distribution system. Other components, such as candidate or location selection, directly depend on the measured load values. But not only the measurement of load is important, but also the question which system parameters are monitored. A common index for the load of a system is the length of the CPU queue. Depending on the specific setup, other factors such as memory utilization or network usage, can be taken into account.

Load Communication Mechanism

The purpose of load communication is to distribute the load information to all the other machines that require this information for their location policy decisions. It is important, that the information is reliably made available and without great delays. In addition to that, network communication should be kept at a minimum.

- *Polling* is the most straight forward way to get information from a remote system. A client, that is monitoring variables of a remote machine, periodically asks this machine for the latest values. This can be implemented using remote procedure call or simple messages. The drawback of polling is that a host has to repeatedly query its counterpart. This can generate lots of needless network traffic if, for example, the monitored variables rarely change their values.

- *Broadcast* represents the opposite of polling, because in broadcast systems the host under surveillance plays the active role. Whenever a variable that is of interest to other machines, changes its value, the host sends a message out on the network. Due to the nature of broadcasts, every other node located on the same network is able to pick up this message.

- *Publisher/Subscriber* is a mechanism to get specific information without periodic polling or broadcasts. In the context of load monitoring, a host that makes its load information available to the public is called publisher. Any other host interested

in this information can contact the publisher and subscribe to a specific variable or information. The publisher will now send an update to the subscriber whenever this variable changes.

- *Central Collector* is an approach where a specific machine collects the current load information of all other machines. If a task is to be transferred to a remote machine, the machine currently hosting the tasks contacts the central collector. The central collector can now provide the enquirer with the current load information. In case the enquirer also sends information about the transfer candidate to the central collector, it can directly select a suitable destination. This represents a centralized location selection policy.

2.4 Constraint Satisfaction Problems

Constraint Satisfaction Problems (CSPs) have their origin in artificial intelligence, but are also applicable to other fields of computer science [Kum92]. The problem of assigning tasks to computers, as done in load distribution, can be modeled as a constraint satisfaction problem. This section will present the foundations of constraint satisfaction problems followed by a discussion of fundamental strategies to solve them.

2.4.1 Definition of Constraint Satisfaction Problems

A constraint satisfaction problem is defined by three elements [RN03], [EET01]:

- a finite set of variables $X = \{x_1, x_2, \ldots, x_n\}$.
- each variable x_i has a non-empty domain D_i of possible values.
- a set of constraints $C = \{c_1, c_2, \ldots, c_n\}$. A constraint c_j involves one or more variables and defines feasible values for these variables. As an example for two variables x_1 and x_2, a constraint can be given explicitly by enumerating all allowed value combinations such as $c = \{(a, b), (b, c)\}$. Another form of constraint is an implicit constraint where the constraint is represented by a formula like $x_1 < x_2$.

An assignment of values to some or all variables is called a state of the problem. If the assignment does not violate any constraint, the assignment is called consistent. A solution of a constraint satisfaction problem is an assignment involving all variables, also called a complete assignment, which is consistent.

Figure 2.8a shows an example of a constraint satisfaction problem as presented in [EET01]. Four areas should be colored using three different colors. Adjacent areas must not have the same color. The formal definition of this constraint satisfaction problem is presented in equations 2.1 to 2.3.

$$X = \{x_1, x_2, x_3, x_4\} \tag{2.1}$$

$$D_i = \{red, green, blue\} \tag{2.2}$$

$$C = \{x_1 \neq x_2, x_1 \neq x_3, x_1 \neq x_4, x_2 \neq x_3, x_3 \neq x_4\} \tag{2.3}$$

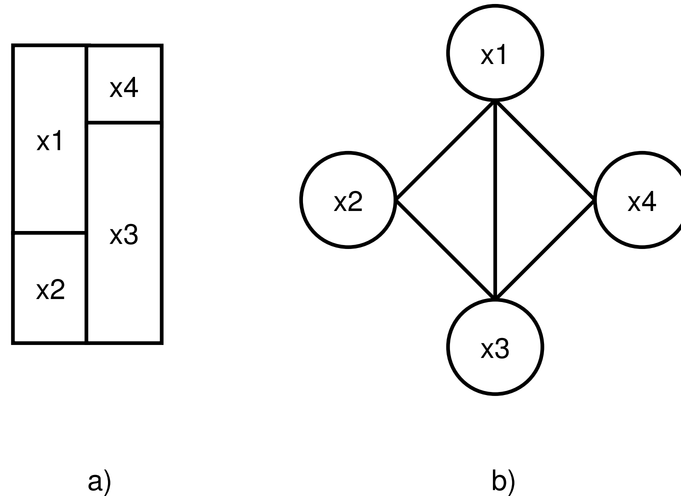


Figure 2.8: (a) Assigning three different colors to x1, x2, x3 and x4 such that no adjacent areas have the same color, is a CSP. (b) In the constraint graph, nodes represent variables and arcs represent constraints.

A constraint satisfaction problem that only includes constraints, involving not more than two variables, is called a binary constraint satisfaction problem. It can be modeled as a graph where variables correspond to the nodes, and constraints are represented by the arcs (figure 2.8b). Higher-order constraints can be reduced to binary constraints [Kum92].

2.4.2 Solving Constraint Satisfaction Problems

For solving a constraint satisfaction problem, a straight forward approach is called *generate-and-test*. Every possible assignment of values to variables is generated systematically. A solution is found, if the current assignment satisfies all constraints. If not all possible solutions for the problem are required, the search can be stopped. Otherwise it has to continue until all remaining combinations have been generated and tested. The number of possible complete assignments is given by $O(d^n)$, where d denotes the maximum domain size and n stands for the number of variables.

Backtracking

Backtracking algorithms provide a more efficient way to find solutions for constraint satisfaction problems than simple generate-and-test. In backtracking, values are assigned to variables sequentially. Whenever all variables involved in a constraint have been assigned

a value, this partial assignment is checked for validity. If the partial assignment violates a constraint, no further yet unassigned, variables will be assigned a value. Instead, the algorithm goes back to the variable most recently assigned that still has alternatives left. For this variable, another value is selected. Whenever backtracking is performed, a subset of the solution-space is discarded, which would have needlessly been scanned, when using generate-and-test. Nevertheless, the run-time complexity of simple backtracking is still exponential for most non-trivial problems [Kum92].

To improve the performance of backtracking, the number of examined branches need to be reduced. One way to reduce the number of visited branches relies on how the next variable to be assigned, is chosen. The *minimum remaining values* heuristic suggests to select that variable that offers the smallest set of remaining values. The idea is, that such a variable will lead to failure faster than a variable with a lot of possible values. That way the search space is reduced faster. Aside from sensibly selecting the next variable, a careful selection of the next value to be assigned, can be benefiting. One way to do this is known as the *least constrained value* heuristic. It selects this value that maximizes the number of possible assignments for the following variables. The idea is that keeping a broader choice of values for the following variables, will more likely result in finding a solution earlier. If, however, all solutions of a CSP are required, the ordering of values does not matter because all of them have to be examined. The same is true if no solutions for the problem exist.

One of the problems of backtracking is thrashing, which means that the same reason leads to failures in different parts of the search space. *Node inconsistency* can be one cause for this type of failures. Node inconsistency results from unary constraints, which are constraints that only involve one variable. To remove node inconsistencies, one has to remove all values from the variable's domain that do not satisfy the constraint. By this preprocessing, also the unary constraint itself becomes obsolete.

Forward checking provides another mechanism to reduce the search space. Every time a value is assigned to a variable x_i, all, yet unassigned, variables that are related to x_i by some constraint, have to be checked. From the domains of these unassigned variables all values are deleted that are in conflict with the value assigned to x_i.

Arc inconsistency represents a form of thrashing that involves two variables x_i and x_j that are related to one another by a binary constraint. x_i is assigned some value $x_i = v$. If a binary constraint between x_i and x_j is defined in such a way that, after v was assigned to x_i, no value of D_j is allowed for x_j, this is called an arc inconsistency. The algorithm will always fail for the same reason whenever x_i is assigned the value v, and x_j should be assigned any value later on, no matter which other variables have been instantiated in between. Arc (in)consistency is a directional property. Having an arc inconsistency from x_i to x_j does not necessarily mean that the arc from x_j to x_i is inconsistent. Arc inconsistencies can be removed in a preprocessing step: One has to check if for every value for x_i from D_i exists a valid x_j from D_j. If no suitable element of D_j is found, the value of x_i is removed from D_i. This deletion does not affect the solution of the CSP since the deleted value could not have been part of a complete solution.

2.4.3 Distributed Constraint Satisfaction Problems

Distributed constraint satisfaction problems (DCSPs) are defined as CPSs, that involve multiple software processes. [LHB93] describes different types of distribution for CSPs. For *variable-based distribution*, each process, that solves a part of a CSP, is in charge of one or more variables and their domains (figure 2.9). The process solves the local CSP. In a next step it has to communicate its partial solution of the CSP to the other processes, whose variables share constraints with the variables of the local process. The communication and synchronization overhead required, can be very high for this type of distribution.

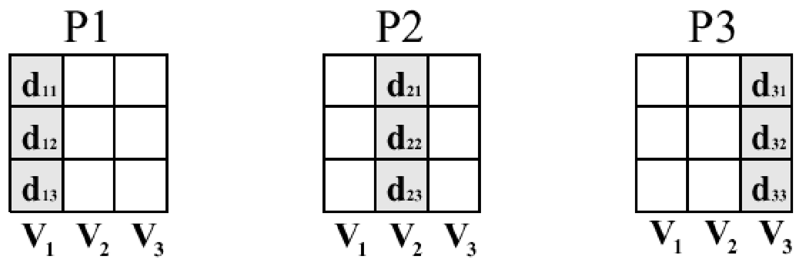


Figure 2.9: Each process is in charge of one variable. (source: [LHB93])

Domain based distribution divides the original problem by its domain. In figure 2.10 the domain of variable V_1 is divided into three sub domains, resulting in three independent search spaces. Each of the three search spaces contains all three variables with V_1 set to a different element of its domain. The value of V_1 is also called the root of the search space. The DCSPs can now be solved independently of one another.

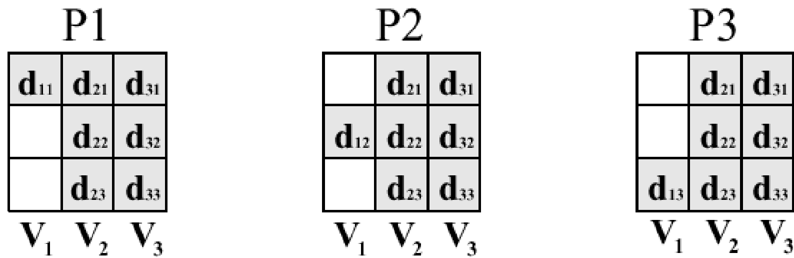


Figure 2.10: The CSP is split into smaller CSPs by decomposing the domain of V_1 into three sub domains. (source: [LHB93])

The approach of distributing a CSP by splitting its domain into sub domains, and thereby generating several CSPs with a reduced search space, is versatile. It is not difficult to implement and it can make use of large multiprocessor networks since only very little communication is required.

Chapter 3

Related Work

This chapter presents a survey of the related work in the field of agent based load distribution. It is followed by a summary of the approaches, where the problem of allocating tasks to specific nodes was modeled as a constraint satisfaction problem.

3.1 Load Distribution

The migration capabilities of mobile agents make them especially useful for the implementation of load distribution systems. Tasks that are modeled as mobile agents can be moved to any host at runtime. Not only the mobility of agents is of interest for load distribution systems, but also inter-agent communication can be used for the monitoring and communication of load information.

3.1.1 Resource Monitoring

An essential component of load distribution systems is the monitoring of the available system resources. This information has to be collected on every machine, and then distributed to all the other nodes of the system. In literature, a number of approaches based on mobile agents as well as approaches that combine agent based on conventional technologies can be found.

Load Monitoring and Communication Using Mobile Agents

[RH00] presents a system for resource monitoring that makes use of stationary agents. On every node of execution, an agent is located that gathers the local system state. When selecting the new location for a task, the load status of all potential targets is required. To get this information, the presented approach proposes, that every load monitoring agent communicates with every other node of the system thru messages. That way, an agent obtains a complete picture of the load status of the overall system. For larger setups the agents can limit their search to hosts in their vicinity in order to reduce network traffic.

The obtained information is stored in a local repository, also known as knowledge base, used for decision making.

[CWD02] proposes another approach that is using mobile agents for load monitoring and load communication. An agent, also called Load Information Agent (LIA), is traveling from host to host. On every node, this agent collects the local load information and adds it to its internal list with a timestamp. In addition to that, the LIA leaves a copy of its list on the current host before it moves on to the next one. This way, every host of the system gets a snapshot of the global load and resource situation. The application of such a LIA is of special interest, since it allows the preprocessing and aggregation of load information before it is transmitted over the network. This can significantly help to reduce the amount of bandwidth required for load communication. A potential disadvantage is that the snapshot of the global load and resource situation might not be accurate if the machine has not been recently visited by the LIA.

Combined Approaches

For network monitoring [LKP99], or the monitoring of network connected systems, it is often favorable to deploy agents in a hierarchical fashion, since this typically fits existing network topologies, such as networks split into sub-nets. To reduce the overall agent count, multiple entities to be monitored, can be assigned to an agent instead of deploying one agent per entity. Depending on the type of device, the agent can either migrate to the device to obtain current load information or poll the device. Using the conventional technology of polling is useful if a given device is not running an agency, but offers other monitoring facilities, such as a Simple Network Management Protocol (SNMP) interface [RB02]. The agent can then aggregate the information of all its monitored objects, and pass it upward when being polled by an agent in an upper level of the hierarchy, or by some kind of subscription based mechanism.

3.1.2 Mobile Agent Based Load Distribution

Not all work in the field of load distribution using agents deals with the load of the machine hosting the agent. [SSCJ98] focuses on the load of an agent, in contrast to the load of the machine the agent is running on. An agent might be overloaded by its duties while the machine it is executed on, still has free resources. Therefore, agent cloning, either locally or remote, is proposed to overcome this problem. If an agent is overloaded, the following two options exist: (1) Agents are seen as computation units, that can perform one or more tasks. To reduce the load of an agent, sub tasks can be assigned to agents with lower load. (2) Creating a clone of the original agent helps to distribute the load. The clone can be created on the local machine or on a remote host. The mechanism of agent migration is described to be a subset of agent cloning: if an agent was cloned on a remote host and the original agent dies, this can be seen as agent migration. Cloning of agents will not help resolving high load situations, if the problem the original agent is working on cannot be decomposed into sub-problems.

Exploiting agent mobility, [CWD02] describes a framework for load sharing in web server groups. Three different types of agents form the core of the proposal:

- The *Server Module Agent (SMA)* is a stationary agent that is executed on every server. It handles incoming requests from clients. In case the server gets overloaded, the server module agent selects one ore more tasks to be re-located to other hosts. To determine if a server is overloaded, a static or dynamic threshold is used. The selection of the new location of a task and its migration is done by the job dispatching agent.
- The *Job Dispatching Agent (JDA)* is a mobile agent that determines the target host for a task using the information provided by the load information agent. The job dispatching agent then migrates to the target host on behalf of the actual task to be migrated. At the target host, the JDA either gets the acknowledgment for the resources the actual task requires, or not. Since distributed systems might show high load fluctuations, it is possible that the target host is no longer able to satisfy the resource requirements of the task. In such a case, the JDA picks a new destination host to move to. This process is repeated until the JDA manages to get an acknowledgment for the resources required by the task to be migrated. At this point, the actual task is finally migrated to the target host.
- The *Load Information Agent (LIA)* was already described earlier in this section. It provides every host with a local snapshot of the global load situation. The JDA uses this local information for its migration decisions, without the need to query a central server or poll all the nodes of the system. For large networks multiple, cooperating LIAs can be deployed.

[DH02] presents a load distribution mechanism using mobile agents where an agent is continually circulating through the network. Similar to the LIA proposed in [CWD02], it collects data about the current load situation, on every machine it visits. Based on this information and the knowledge about previously visited machines, the agent can decide to migrate a task of its current host to another machine. All load distribution mechanisms are modeled within this single agent.

Today's computers spend a lot of their time in idle mode. The *While You're Away* system presented in [SGB01] tries to make use of these resources and introduces the notion of roaming computations. Such roaming computations are mobile agents, which are submitted to a central coordinator machine, which then distributes the agents among currently idle workstations. Whenever the owner of a workstation starts to use it himself, the roaming computation agent is stopped and returns to the coordinator machine. To be able to smoothly stop and resume the execution of agents, strong migration techniques based upon a modified Java virtual machine are used.

3.1.3 Load Distribution as Constraint Satisfaction Problems

A wide range of problems can be modeled as constraint satisfaction problems. The field of applications ranges from computer vision and graph problems to scheduling problems.

Also the problem of allocating tasks to nodes of execution can be modeled as a constraint satisfaction problem. Not much literature can be found on this specific topic.

In [FF99], constraint satisfaction techniques are applied to resource allocation in networks. Networks consist of nodes, such as routers, gateways and servers, and connections between the nodes, the links. Links are characterized by their available bandwidth. The network must fulfill the communication needs of users, called demands. The network must satisfy these demands by allocating a connection to each demand. This problem can be mapped to a CSP as follows: the variables of the CSP are the demands of the users. The domain of each variable is the set of all routes between the endpoints of the demand. The constraints are modeled in such a way that the capacity of a link is not exceeded by the demands. While the formulation of the CSP is relatively simple, measures must be found to deal with the complexity of the problem. The graph of a network can not only be complete, where each node has a link to every other node of the network, but even multiple links between two nodes are not uncommon. Hence, the main part of the work deals with measures that help to reduce the complexity of the problem.

3.2 Mobile Agents in Embedded Environments

The idea of deploying agents on embedded environments is not new. While the autonomy of agents is an important aspect, many of the existing implementations use stationary agents instead of mobile agents [Vrb04, OTM03]. By design, stationary agents are limited to message passing for interaction with other agents. Representatives of this class of agent systems are the "Lightweight Extensible Agent Platform" [LEA05] and CougaarME [Cou02]. Both of them are designed to run with the "Connected Limited Device Configuration" of the Java 2 Micro Edition.

[OTM03] describes the implementation of agents, called Very Lightweight Agents, that run on digital signal processors. In this project, the agents are used for data acquisition and system monitoring in a sensor network. The agents are stationary, they can be reactive or proactive and communicate using messages. The agents are written in C or assembly and are designed to consume as little resources as possible.

Chapter 4

The SmartCam Project

The aim of the SmartCam project is the development of a next generation traffic surveillance platform [RBB+04,BRS04,Bra05]. In contrast to existing video surveillance systems, the SmartCam project tries to do all the data processing directly on the cameras. Surveillance systems of the first and second generation employ analog cameras to capture the observed scene. The video data is transmitted to a central backend system where it is digitized and analyzed. Third generation surveillance systems use digital cameras that feature on-board video compression. The video data is then transmitted digitally to the backend system. Smart cameras, as developed in the SmartCam project, take this concept one step further. They not only capture and transmit the video data digitally, but also do realtime on-board video analysis. Advantages of the distributed approach are reduced network utilization, enhanced fault tolerance by omitting a central point of failure, and greater flexibility.

The following sections present a general view of the SmartCam project, the hardware and software architecture, and the video surveillance tasks.

4.1 Hardware Architecture

The SmartCam system has a multiprocessor architecture. The main processor is responsible for communication with the outside and for system management. For video processing and analysis, additional high speed processors are used.

The prototype platform (figure 4.1) is based on an Intel IXDP425 development board equipped with an XScale CPU running at 533MHz, 256MB of SDRAM memory, 16MB flash memory, two integrated 100Mbit ethernet interfaces and an ADSL interface. An additional GPRS (general packet radio services) module, connected via one of the two available high speed serial ports, allows for wireless communication. The system can carry up to four PCI add-on boards. The prototype is equipped with two *Network Video Development Kits* (NVDK) from ATEME. Both boards are equipped with a TMS320C6416 digital signal processor (DSP) from Texas Instruments, operating at 600MHz and providing 4800 MIPS. Each DSP features 1MB of internal memory and 264MB of local memory organized in two banks. One of the DSPs is connected to a CMOS image sensor and

is therefore responsible for the acquisition of the raw images. Figure 4.2 presents the architecture of the system.

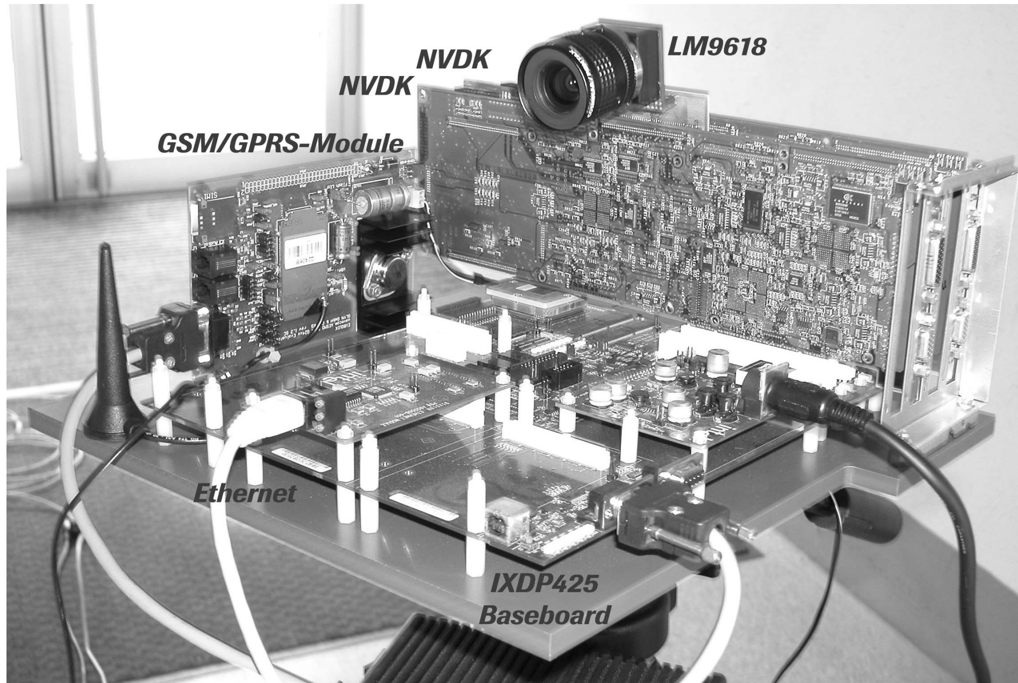


Figure 4.1: The SmartCam prototype.

Video encoding, processing and analysis are entirely done on the digital signal processors. The DSPs are able to communicate with each other as well as with the IXDP425 via the PCI bus. The IXDP425 platform controls the PCI communication of the DSPs.

4.2 Software Architecture

The software architecture of the SmartCam can be divided into two parts: The first part are programs executed on the DSPs, while the second part are applications executed on the XScale platform. For both sides, software frameworks have been developed providing basic functionality. One of the main services of the DSP framework is dynamic loading. It allows software components to be downloaded to, and unloaded from the DSPs at runtime. The DSP framework also manages and monitors the resources of the DSPs. Status information is made available to the SmartCam framework running on the XScale platform.

On the IXDP425 platform, a custom built GNU/Linux distribution [Win04] based on Kernel 2.6.x is used. A kernel module [Wöc04] was developed providing communication facilities for the DSPs using the PCI bus. The kernel module and an additional user space DSP access library (DSPLib), allow applications running under Linux to interact with DSP programs by using messages. Also loading and unloading of DSP applications from user-space is handled by the the DSPlib and the kernel module. For basic interaction with the DSPs, a set of command line applications are available.

On the GNU/Linux operating system, a Java environment is available that runs a mobile agent system. Every task that is executed on one of the DSPs, is represented by a mobile agent on the XScale side. This provides control over the DSP applications and their allocation to DSPs. To be able to communicate with the DSPs, the agent environment relies on the services provided by the SmartCam framework. Figure 4.2 provides an general view of the software layers on the XScale platform.

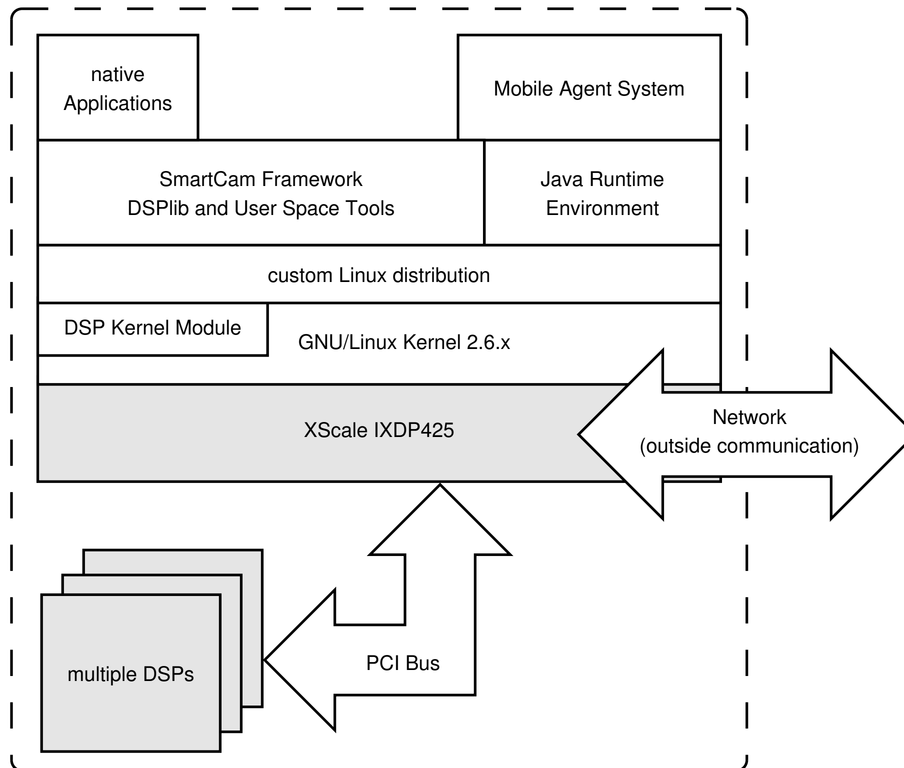


Figure 4.2: The SmartCam prototype is composed of an IXDP425 base board running Linux. Multiple DSPs are attached via the PCI bus. Hardware components are depicted in grey, software components in white.

4.3 Surveillance System Architecture

To be able to monitor a highway segment or a tunnel, multiple cameras are required. As with all embedded systems where low power consumption is important, also the Smart-Cam is limited in its resources. Having multiple cameras, presents the possibility of not executing all tasks on every camera, but to distribute a set of tasks among a group of cameras. Such groups are referred to as clusters. Clusters represent a logical grouping that is not predetermined by network topology or similar constraints. One camera can belong to one or more clusters. Every cluster is assigned a set of tasks. These tasks are

then allocated to the nodes of the cluster without further intervention from the outside. This mechanism is referred to as task allocation or load distribution. At runtime, tasks can be added to, or removed from a cluster. Requirements of agents and available hardware resources can change over time. The load distribution system has to handle these events and (re-)organize the tasks accordingly. An important design goal is that the system is decentralized and operates without a decision maker that has global knowledge.

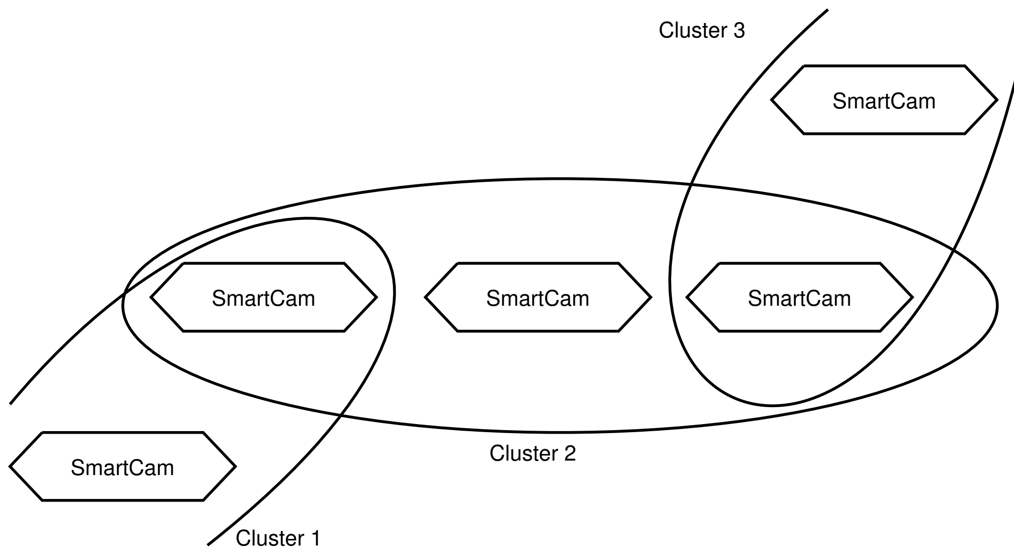


Figure 4.3: SmartCams are deployed in clusters. One camera can belong to one or more clusters.

The clustering of cameras, as depicted in figure 4.3, helps to manage the complexity of the system. Events observed by co-located cameras are causally associated. Examples of such events are wrong way drivers or traffic jams. It is sufficient if such events are detected by one camera of a cluster. Also service tasks, such as traffic statistics gathering, are not required to be executed on every camera.

Chapter 5

Design

This chapter discusses the design of the basic agent infrastructure (section 5.1) and a more detailed explanation of the mechanisms and the design of the load distribution system developed for the SmartCam project (section 5.2).

5.1 Basic Agent Infrastructure

On every camera, also called node, runs an agency. The agency provides an environment for agents to live in. The agents that are hosted by the agencies are divided into two groups:

- *SmartCam agents* are agents that implement control functionality and do not heavily interact with DSPs. Cluster management and the load distribution system are implemented as SmartCam agents.
- *DSP agents* are not only executed on the XScale platform, but also have a DSP part. This DSP part of the agent is downloaded to one of the camera's DSPs for execution. DSP agents typically implement an image processing algorithm, which needs to satisfy realtime constraints. The load distribution system is designed to make sure that all required resources of an agent are available. The requirements of DSP applications are measured in advance by profiling them. This information is stored in the agent.

A standard agency, as shipped with typical agent systems, requires some extensions to support the clustering of nodes. This functionality is provided through a number of agents. The agents provide mechanisms to create clusters, to add nodes to clusters and to allow agents to find out about the other nodes belonging to their cluster. The agents providing this infrastructure are:

- The *Node Information Agent* (NIA) is created on every node at startup. It represents the node to the outside. When a cluster is created, or a node is added to a cluster,

the appropriate functions of the NIA are called. The NIA also provides mechanisms that allow agents to look up the other nodes of a cluster or to get a list of the cluster memberships of a node.

- The *Cluster Manager Agent* (CMA) is not bound to a specific node, but can be located anywhere within a cluster. A CMA is instantiated upon the creation of a cluster. It knows about all the nodes that belong to the cluster. All requests to add or remove a node from a cluster are ultimately routed to the CMA.
- The *Cluster Manager Proxy* (CMP) is a representative of the CMA. The cluster manager proxy is created on a node once the node is added to a cluster. Since a node can belong to multiple clusters, multiple CMPs can exist on one node. The NIA keeps a list of all the CMPs of a node. Incoming requests to the NIA that involve a specific cluster, such as adding a node to the cluster, are passed on to the CMP of this cluster. The CMP then forwards the request to the CMA of the cluster (see figure 5.1). In addition to forwarding requests to the CMA, the CMP also keeps track of all agents of the local node that belong to the cluster represented by the CMP.

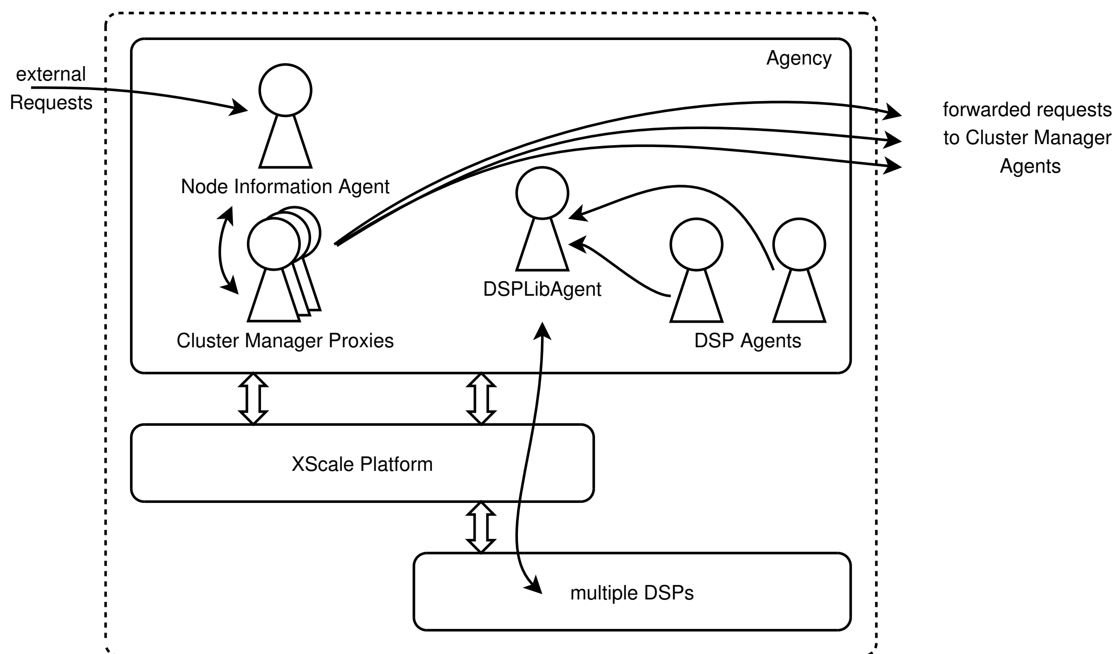


Figure 5.1: Agent Infrastructure of a node: The Node Information Agent handles incoming requests. Every cluster the node belongs to, is represented by a Cluster Manager Proxy. These proxies forward calls to the Cluster Manager Agent. The DSPLibAgent allows DSP agents to communicate with the DSPs.

After startup, every node is running only the NIA. To create a cluster, a call is sent to the NIA on the first node of the cluster. The NIA then creates a CMA that represents the cluster. A CMP is created as a local placeholder for the CMA. The new cluster now

consists of just one node. To add a second node to the cluster, another call is sent to the NIA. This call tells the NIA which other node should be added to the cluster. On this new node, a CMP is created that forwards cluster specific calls to the CMA of the cluster. From the users perspective, all nodes of a cluster are equal. Requests to add additional nodes to a cluster can be sent to any node that is already part of the cluster. The CMA and CMP infrastructure is not seen by the user – all outside communication only takes place with the NIA.

For the DSP agents a facility is required to interact with the DSPs and the applications executed on the DSPs. The DSPlib provides a set of native functions for this purpose. To make this functionality available to agents, a wrapper for the DSPlib has to be provided. This is implemented in the form of an agent called DSPLibAgent (DLA). A DLA is created on every node on startup. DSP agents can use the DLA to download their DSP applications to one of the DSPs of the node. The DLA also makes the publisher/subscriber based messaging interface for DSP communication, as provided by the DSPlib, available to DSP agents.

5.2 Load Distribution

Every cluster consisting of a number of nodes is assigned a set of tasks. It is up to the cluster to find an allocation of those tasks to its nodes. Every task is implemented as a mobile agent. The goal of the load distribution is to find a placement of the agents of a cluster, so that no node is overloaded and all tasks are executed at the best quality possible.

5.2.1 Resource Requirements of Agents

The resource requirements of agents are known in advance and do not have to be measured at runtime. One agent can offer multiple quality of service (QoS) levels. The better the quality delivered by an agent, the higher are its resource requirements. This allows to save system resources if the best quality of an agent, for example an MPEG streaming agent, is not required. The resources considered for an agent are listed in Table 5.1.

Resource Type	XScale	DSP
CPU Load	x	x
internal Memory		x
external Memory	x	x
DMA channels		x
DMA reload tables		x
DMA complete codes		x
Network connectivity	x	
PCI Bus		x

Table 5.1: The considered resources on the XScale platform and on the DSPs.

Since all algorithms that require a lot of computational power are executed on the DSPs, the considered requirements are focused on DSP resources. In addition to CPU and internal memory usage, the memory hierarchy is considered for the DSPs. Not only the fast, but small internal memory, but also a larger and slower external memory is used. To fully exploit the performance of the DSPs, the algorithms make use of direct memory access (DMA) where possible. The available DMA resources are limited and therefore have to be considered as well. For communication between DSPs, the PCI bus is used. For the external communication, which is done by the XScale part of the agent, different types of connections are available such as standard ethernet and gigabit ethernet, wireless LAN or GPRS. The internal bus system as well as the network connections are limited in bandwidth. Hence, these resources also have to be considered when placing agents.

5.2.2 Agent Placement: A Distributed Constraint Satisfaction Problem

When determining the nodes for the execution of the agents of a cluster, the load distribution mechanism has to make sure that all requirements of the agents are met. Finding an allocation of agents to the nodes can be described as a distributed constraint satisfaction problem. The variables of the problem are the agents to be placed in the cluster:

$$A = \{a_1, a_2, a_3, ..., a_n\} \tag{5.1}$$

Every agent a_i can be placed on one of the m nodes of the cluster. It is also possible that agents cannot be assigned to a node due to insufficient resources of the cluster. This is represented by the additional element *notAssigned*. Hence, the domain D of a_i is:

$$D = \{1, 2, 3, ..., m, notAssigned\} \tag{5.2}$$

Without considering any constraints, this results in $(m+1)^n$ possible assignments of the n agents to the m nodes. A specific assignment is referred to as A_j. Due to the limited resources of the nodes, not all of the assignments are applicable in practice. The limited resources of the nodes and the resource requirements of the agents are described by the following constraint C_{A_j} for a specific assignment A_j:

$$C_{A_j} = \{\forall_{a_i \in A_j} : \forall_{d \in D} : \forall_{r \in R} : a_i = d : (\sum_{a_i} required(r, a_i)) \leq available(r, d)\} \tag{5.3}$$

with

$$R = \{CPU, MEM, DMA, \ldots\} \tag{5.4}$$

For a placement of agents A_j, the sum of resources *required* by the agents on every node has to be less than or equal to the amount of *available* resources R for every individual resource type r.

The full list of considered resources is shown in table 5.1. Solving the constraint satisfaction problem given in equations 5.1 to 5.4 on a single machine is not efficient. The problem can be divided into smaller problems in such a way that all nodes of a cluster can participate in solving the problem. To do so, the domain D is split into m sub-domains D_1 to D_m containing only two possible values. The set of agents to be placed remains the same. The constraints are modified to reflect that only a single node is considered. The resulting constraint satisfaction problem to be solved on every node k independently is:

$$A = \{a_1, a_2, a_3, ..., a_n\}$$

$$D_k = \{k, \text{not assigned}\}$$

$$C_{A_j} = \{\forall_{a_i \in A_j} : \forall_{d \in D} : \forall_{r \in R} : a_i = d : \sum_{a_i} required(r, a_i) \leq available(r, k)\}$$

This constraint satisfaction problem is solved in parallel on every node k. The overall goal is to find a placement of agents in such a way that all agents are assigned to a node and can be executed. Such a placement that contains all agents is called a complete solution. For a sub-problem computed on a single node, this will hardly be the case since this would mean that this node has enough resources to host all agents. Therefore, every node also generates all partial solutions. Partial solutions are solutions where not all agents are assigned to the node. The maximum size of the solution set generated on every node is 2^n. To generate all these solutions a backtracking approach is used.

Solving m sub-problems results in m sets of allocations $P = \{P_1, P_2, \ldots, P_m\}$ where P_i is the result set of node i containing up to 2^n possible placements. In the next step those result sets P_i have to be merged to get the complete solutions. When merging allocations, it must be ensured that no agent is assigned to more than one node concurrently. When all solution sets have been merged, all allocations are removed, which do not contain all agents. The resulting set of solutions is called the set of feasible solutions F. Merging the solution sets from the individual nodes can be done in parallel.

5.2.3 Selecting a Solution Based Upon its Costs

The result of solving the distributed constraint satisfaction problem is a set of feasible solutions. Such a feasible solution defines the allocation of the agents to the nodes of the cluster. To select the best solution, a cost scheme with five cost types is defined: Resource costs (C_R), data-transfer costs (C_T), migration costs (C_M), affinity costs (C_A) and quality-of-service costs (C_Q). These costs are calculated on every node and for every agent

Quality of Service Costs

Agents can operate at different quality of service (QoS) levels. The level with the best service quality is defined as level 0, levels with lower quality are labeled with ascending positive numbers. The goal of the load distribution system is to find an agent placement, where all agents operate at the best quality possible. To encourage this, quality of service degradation penalty costs, QoS_{deg}, are introduced:

$$QoS_{deg} = \begin{cases} QoS_{level} * QoS_{DegradationFactor}, & \text{if } QoS_{level} \neq 0 \\ 1, & \text{if } QoS_{level} = 0 \end{cases}$$

The lower the service quality becomes, higher the quality of service degradation costs QoS_{deg} is. If there are multiple agent placements that include all agents, then those with lower quality of service costs are preferred.

The operator of the system can define a desired quality of service level for an agent. This is honored by the load distribution system with the $QoS_{wrongLevel}$ costs. If the agent is not running at its desired quality of service level, $QoS_{wrongLevel}$ will rise:

$$QoS_{wrongLevel} = \begin{cases} QoS_{NotDesiredLevelPenalty}, & \text{if } QoS_{level} \neq QoS_{DesiredLevel} \\ 1, & \text{if } QoS_{level} = QoS_{DesiredLevel} \end{cases}$$

The quality of service costs are used together with the resource and data transfer costs. The lower the service quality of an agent is, the lower are its resource requirements and transfers are. Every quality of service level of an agent is treated separately by the load distribution system. Therefore, agents with many low quality of service levels will significantly increase the number of possible allocations. A large number of quality of service levels will, therefore, yield longer runtimes of the load distribution system to find a solution.

Resource Costs

For the resource types required by the agents (table 5.1), costs are defined. For every individual resource type, the ratio of resources required to resources available is computed. The available resources of a digital signal processor are those resources not yet consumed by agents of other clusters.

$$C_{R_i} = \frac{\text{Required Resources}}{\text{Available Resources}} * K_{R_i}$$

If the requirements of a resource exceed the availability, the cost for this resource C_{R_i} is set to infinity. An agent can support multiple quality of service levels with different resource requirements. Hence, the costs C_{R_i} are computed for every QoS level. The total resource costs for a QoS level are the sum of all individual resource costs:

$$C_R = (\sum_{R_i} C_{R_i}) * QoS_{deg} * QoS_{wrong}$$

The QoS_{deg} costs factor becomes the higher, the lower the quality of an agent is. This encourages the best possible service quality. The QoS_{wrong} costs factor is used to encourage an agent to run at a predefined QoS level.

Data Transfer Costs

For data transfer, two different kinds are distinguished: (1) Internal data transfers and (2) external data transfers. Internal data transfers are transfers via the PCI bus or direct memory transfers. The external data transfers are transfers using a network interface for communication with remote hosts. Different kinds of data transfers offer different performances, resulting in different costs. The amount of data of a specific type, transferred per second by an agent, is multiplied with the corresponding factor from table 5.2:

$$C_{T_i} = \text{Data Transfer Per Second} * \text{Data Transfer Cost Factor}$$

Connection	Bandwidth	Costs per MB
GPRS	28 kbit/s	1171.00
10Mbit Ethernet	10 Mbit/s	61.90
100Mbit Ethernet	100 Mbit/s	19.59
Gigabit Ethernet	1000 Mbit/s	6.19
PCI	1064 Mbit/s	6.00
SDRAM	6400 Mbit/s	2.44
On-Chip Memory	38400 Mbit/s	1.00

Table 5.2: Data transfer costs and bandwidths for different interfaces.

The total data transfer costs of an agent are the sum of all its individual data transfer costs. The amount of data transferred per second depends on the quality of service level the agent is running at. As with resource costs, lower quality results in increased costs through QoS_{deg} costs factor. Not running at the desired QoS level also adds additional costs through QoS_{wrong} costs factor:

$$C_T = (\sum_{T_i} C_{T_i}) * QoS_{deg} * QoS_{wrong}$$

Migration Costs

The implementation of a new agent placement can require agents to be moved from one node of execution to another. Two types of costs are defined to reflect the effort that results from agent migration.

- *Migration Transfer Costs — C_{MT}.* When an agent is moved from one node to another, it first has to save the state of its DSP application. This state is saved inside of the agent, increasing the size of the agent. The agent designer estimates a typical size of this data to be transferred. This size is then multiplied with the appropriate data transfer cost factor from table 5.2. The migration transfer costs will be zero if the agent does not move from one node to another, but only changes its DSP or quality of service level.

$$C_{MT} = \begin{cases} 0, & \text{if no migration is required} \\ \text{Agent Size} * \text{Data Transfer Cost Factor}, & \text{if migration is required} \end{cases}$$

- *Migration Idle Costs — C_{MI}.* Many of the DSP algorithms require a certain amount of time to become operational after they have been started. This is due to learning and initialization done by the video algorithms. The migration idle costs are based on this idle time of an algorithm. The idle time is measured by the agent developer and is stored in the agent. The migration idle costs are not only taken into account when an agent moves from one node to another, but also when the DSP or the QoS level is changed.

$$C_{MI} = \begin{cases} 0, & \text{if no migration, DSP or QoS change is required} \\ \text{Agent Idle Time}, & \text{if migration, DSP or QoS change is required} \end{cases}$$

The migration idle cost and the migration transfer costs are scaled and added to receive the total migration cost C_M of an agent:

$$C_M = k_{MT} * C_{MT} + k_{MI} * C_{MI}$$

Affinity Costs

The agent designer can influence the placement of agents inside the cluster by defining the affinity of the agents. Two different types of affinities are available:

- *Cluster Affinity.* The workload of a cluster can increase over time. Reasons are increased resource requirements of agents or the addition of new agents. Because of the limited resources of a cluster, not all agents might be able to run. The cluster affinity defines a priority scheme where a high cluster affinity makes the allocation of an agent to a camera more likely than a low cluster affinity. From this cluster affinity, which is defined for every agent, the cluster affinity costs C_{AC} are derived:

$$C_{AC} = \begin{cases} 0, & \text{if Cluster Affinity} = 0 \\ \frac{\text{Max Cluster Affinity}}{\text{Cluster Affinity}}, & \text{if Cluster Affinity} \neq 0 \end{cases}$$

  A higher cluster affinity denotes lower cluster affinity costs. Solutions with low overall costs are preferred to those with higher costs.

- *Scene Affinity.* Some agents rely on the video data from a specific camera. An example are MPEG encoder agents, which encode the raw video supplied by the CMOS sensor. To increase the probability that such an agent is running on the desired node, the scene affinity is introduced. If it is not possible to execute the agent on the desired node, the raw video images have to be forwarded to the agent over the network. This additional effort is modeled by a remote penalty factor k_p.

$$C_{AS} = \begin{cases} 0, & \text{if Scene Affinity} = 0 \\ \frac{\text{Max Scene Affinity}}{\text{Scene Affinity}} * k_L, & \text{if Scene Affinity} \neq 0 \end{cases}$$

$$K_L = \begin{cases} 1, & \text{if the agent is executed on the desired node} \\ k_p, & \text{if the agent is not executed on the desired node} \end{cases}$$

  Again, a higher scene affinity results in lower scene affinity costs C_{AS}. Before solving the distributed constraint satisfaction problem, agents with a scene affinity other than zero are asked about their desired node. This allows the agent to take influence on its new location.

An agent can be either cluster affine or scene affine. The total affinity cost is the maximum of the scaled cluster and scene affinity:

$$C_A = max(k_{AC} * C_{AC}, k_{AS} * C_{AS})$$

Total Costs

Since every node can be equipped with different resources, and the load situation on the nodes of a cluster must not be homogeneous, the costs for an agent are calculated individually on each node. The values for resource costs, data transfer costs, migration costs and affinity costs are summed up resulting in the total costs C_{tot} of an agent. Additional scaling factors allow the system operator to emphasize some cost type over another:

$$C_{tot} = k_R * C_R + k_T * C_T + k_M * C_M + k_A * C_A$$

The costs of a solution containing multiple agents is the sum of the individual total costs of all agents present in the solution. When two partial solutions are merged during the merge phase of the distributed constraint satisfaction problem, the total costs of the placements are accumulated.

Complexity of the Problem

Solving the constraint satisfaction problem for n agents and m nodes, a maximum number of m^n complete solutions can be found. Complete solutions are those that contain all agents. For the m local CSPs solved on every node, $1^n = 1$ complete solutions can be found if all agents are equipped with one QoS level. For multiple QoS levels, this

number increases. But when solving the local CSPs, not only complete solutions are of interest, but also partial ones. As already mentioned, this is taken into account by adding the *notAssigned* element to the domains of the local CSPs. The maximum number of placements generated for every DSP on every node increases to:

$$O(2^n)$$

The next operation to be discussed is the merge of two such solution sets. For two sets resulting from the solved CSP of two DSPs, the maximum size of the merged solution set is given by 3^n. The three elements of the domain are the two DSPs and the *notAssigned* element. A more general formulation of the maximum number of placements of a merged solution set is:

$$O((numDSPs1 + numDSPs2 + 1)^n)$$

where $numDSPs1$ and $numDSPs2$ denote the number of DSPs covered by solution sets 1 and 2 respectively. The additional element is, again, the *notAssigned* element.

Adding additional agents to a cluster results in an exponential growth of the solution set size. Adding nodes, or more specific DSPs, results in a polynomial growth. It has to be noted that the numbers presented are worst case scenarios. For real applications, the number of placements is reduced significantly by the resource constraints. An approach to reduce the number of placements further is presented in the next section.

Early Pruning

For a cluster that is hosting many agents with relatively low resource requirements, solving the CSP will result in a large number of possible combinations for the placement of the agents. This leads to an increased running time of the CSP. To limit this effect, a mechanism called early pruning is introduced. When enumerating all possible placements of agents, not only the available resources are considered but also the cost of the placement. If the costs exceed a predefined threshold, the placement will be rejected. The rationale is, that such a placement with high costs will not be the cheapest one that is finally selected and implemented.

The early pruning can be modeled as an additional constraint for every concrete agent placement A_j:

$$C_{A_j} = \{\forall_{a_i \in A_j} : \sum_{a_i} TotalCost(a_i) \leq TotalCostThreshold\}$$

The cost threshold is not only checked when computing the CSP on a node, but also when merging partial solutions of the CSP from different nodes. If a merged solution exceeds the total cost threshold, it is rejected. The early pruning mechanism can help to reduce the size of the solution sets, and therefore decrease the running time of the

computation. But it has to be kept in mind that this is achieved by rejecting otherwise valid solutions.

5.2.4 Using Mobile Agents for Task Allocation

The algorithms presented in sections 5.2.2 and 5.2.3 are implemented using the mobile agent paradigm. Not only is every task represented by a mobile agent but mobile agents are used for solving the constraint satisfaction problems on every node and the subsequent merge operations. This section presents a schematic outline of workflow.

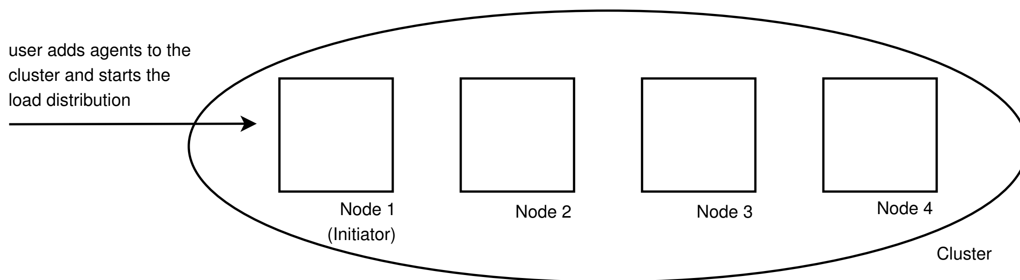


Figure 5.2: Nodes are grouped into clusters. The user adds agents to the cluster. After all required agents have been added, the initial load distribution for the cluster is started.

After having grouped the nodes into clusters, the first step is the assignment of agents to a cluster as shown in figure 5.2. After all required agents have been assigned, the initial load distribution process is started by the user. The node that was selected by the user to start the load distribution, is called the initiator for this load distribution process.

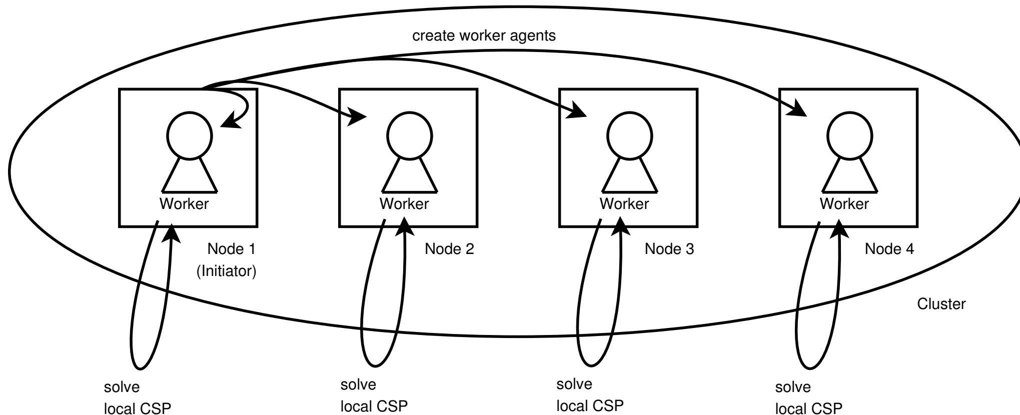


Figure 5.3: The initiator creates a worker agent on every node. The workers solve the local constraint satisfaction problem.

In a first step, the initiator locks all nodes of the cluster. Subsequently, it creates a worker agent on every node (figure 5.3). This worker has a list of all the agents assigned to the cluster. From the DSPLibAgent, the worker agent gets all required information about the available digital signal processors of the node and their resources. Based on

this information each worker agent solves the local constraint satisfaction problem. When finished, every worker sends a message to notify the initiator (figure 5.4).

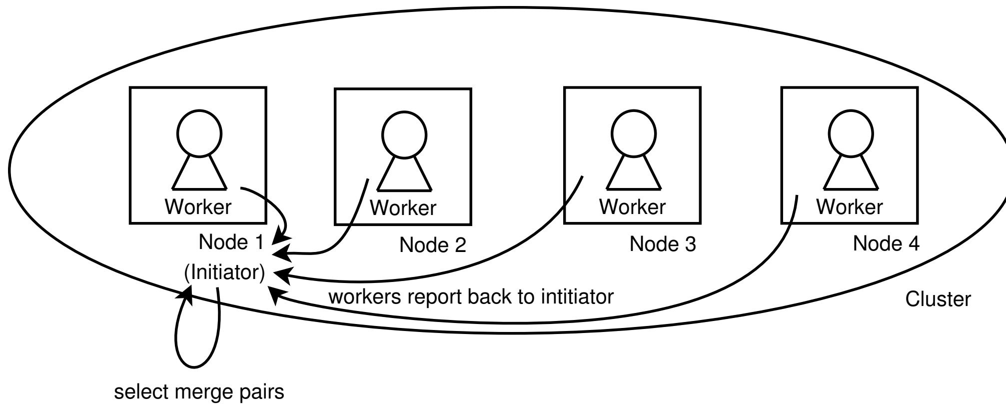


Figure 5.4: After the workers have solved the local constraint satisfaction problem, they report back to the initiator. The initiator selects pairs of workers for the merge process.

The initiator then picks two worker agents for a merge. The first worker, called the master, remains at its current node while the second worker moves to the node of the master (figure 5.5). After it arrives, the two workers merge their solutions. While the master reports back the completion of the merge to the initiator, the second worker is destroyed. The initiator again picks two workers to merge their solutions. This process continues until there is only one remaining worker agent.

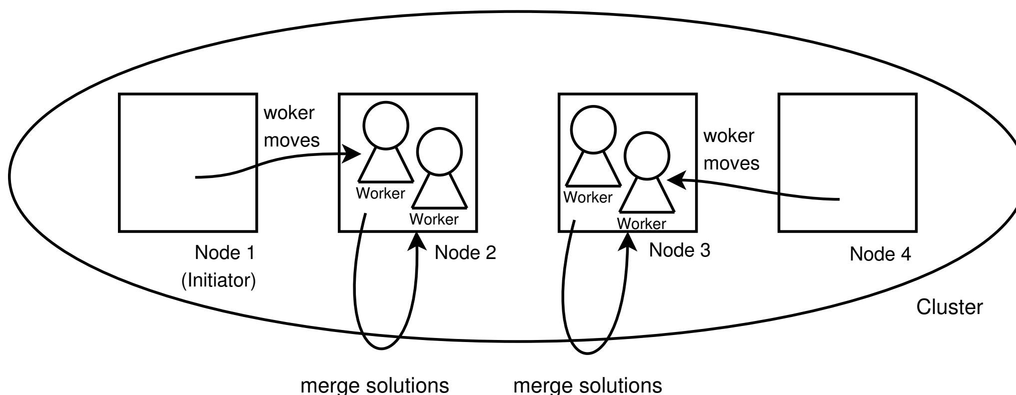


Figure 5.5: Worker agents move to the node of their merge partner where the two workers merge their solutions.

This last worker contains the final set of solutions. The worker extracts the complete solutions, which are those containing all agents, and reports them to the initiator. If, in case of insufficient resources, no complete solutions were found, the worker agent reports the next best solutions. These are the solutions where one or more agents could not be assigned to a node.

The initiator selects the cheapest solution according to the cost functions defined in section 5.2.3. This solution is then implemented by the initiator. It sends messages to every node telling them which agents to start, which DSP they should use and at what quality of service level they should operate. The final set of solutions is sent to all nodes of the cluster. Finally the initiator unlocks all nodes of the cluster.

5.2.5 Inter-Cluster Communication

The process of initially assigning agents to a cluster, as described in section 5.2.4, omits an important detail: A node can be part of multiple clusters. This introduces a number of issues to be considered. To illustrate them, the two clusters depicted in figure 5.6 will be used.

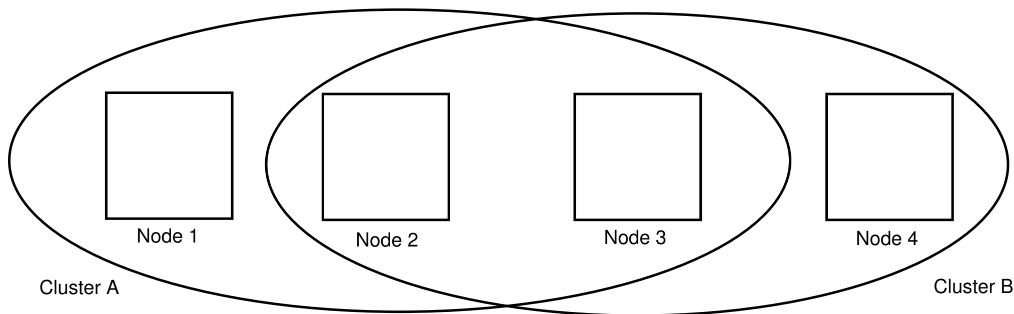


Figure 5.6: Two overlapping clusters.

For cluster A, the initial allocation of agents can be done as described in section 5.2.4. For cluster B, this approach has to be adapted. Nodes 2 and 3 are part of cluster A and cluster B. The worker agents, which solve the local constraint satisfaction problems on nodes 2 and 3 for cluster B, generate all feasible placements of agents. As a basis for their computations, the workers consider the available resources of the node.

An alternative approach is to use the total resources of the node without considering the resources already occupied by cluster A. This ensures that all possible placements are generated. This allows cluster B to take advantage of a lighter load on nodes 2 and 3 in the future. In such a case, the constraint satisfaction problem does not have to be solved again. The load distribution system can fall back to the already calculated set of placements. For the moment, however, the resources already taken by cluster A cannot be ignored. Hence, placements of agents of cluster B that cannot be applied due to the resources already occupied by cluster A are marked as disabled.

For this work, the approach to solve the CSP based on the currently available resources was selected. This allows to reduce the size of the solution sets in case of overlapping clusters compared to the variant where solutions only are disabled.

Cluster B now proceeds as specified in section 5.2.4: It merges the solutions generated on every node, and finally selects the cheapest complete solution and implements it. The start of the agents of cluster B will reduce the remaining available resources on the nodes of cluster B. Node 2 and node 3 are not only part of cluster B, but also belong to cluster

A. As a consequence, cluster A must be notified that the amount of available resources on nodes 2 and 3 has changed. This process is detailed below.

Due to the limited resources of the nodes it cannot be guaranteed that a complete solution is found. In such a case two options exist:

- An incomplete solution is selected and implemented. A solution where not all agents of the cluster are assigned to a node for execution is called incomplete. The decision which agents are not executed is done implicitly by the cluster affinity costs described in section 5.2.3. Tasks with a high importance to the cluster have a high cluster affinity, resulting in low cluster affinity costs. Tasks that are less important have higher cluster affinity costs. Since the solution that is finally selected is the cheapest one, it is more likely that agents with a high importance for the cluster will be assigned a node for execution rather than agents that are less important.
- If some of the nodes are members of multiple clusters, as is the case with node 2 and node 3, the foreign clusters can be asked to reduce the load of the shared nodes. A reduced load on node 2 and 3 might allow cluster B to find a complete solution. The process of load reduction of a node is described later in this section. If, however, a reduced load on node 2 and 3 does not lead to a complete solution for cluster B, the selection and implementation of an incomplete solution cannot be avoided.

Figure 5.7 presents flow charts of the agent placement actions when multiple clusters are involved.

Informing Other Clusters of Load Changes

Whenever the load of a node is changed, all clusters this node belongs to, have to be notified. This gives the clusters the chance to update their solution sets. Two cases can be distinguished: An increased load of a node and a decreased load.

In figure 5.6, nodes 2 and 3 are members of clusters A and B. If the load these nodes increases because of the initial placement of agents from cluster B, the solution set of cluster A needs to be updated. Node 2 and node 3 reduce the solution set of cluster A by disabling all those solutions that are no longer feasible because of the resources consumed by agents of cluster B on these nodes. The sets of disabled solutions are sent to all nodes in cluster A to allow them to keep their local solution sets up to date. The increased load on a node, caused by a another cluster, does not require any further actions by cluster A: The agent placement of cluster A does not need to be modified because cluster B only occupied resources that were available anyway. Cluster A only has to make sure to maintain a correct view of the feasible placements for possible future relocations of its agents.

The opposite event is the reduction of the load of a node. This can result from an event such as the removal of an agent or decreased resource requirements. In such a case, the cluster in which the event occurred gets the chance to revise its solution set. With additional free resources, currently disabled solutions might become applicable. If this is

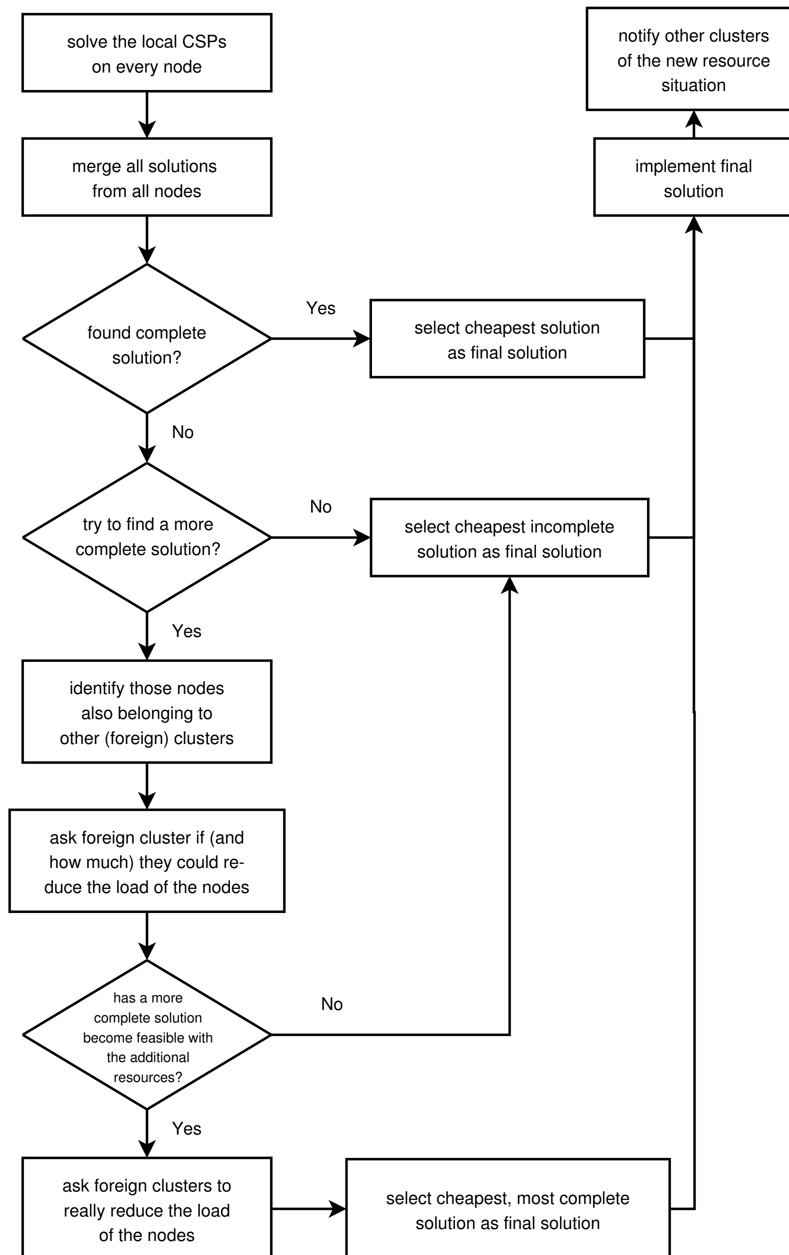


Figure 5.7: Finding the allocation of agents to nodes of a cluster when nodes belong to multiple clusters and insufficient resources are available.

the case, the cluster will check if one of these solutions is cheaper than the one currently executed. If such a solution is found, it is implemented. As a consequence, the available resources of the nodes in this cluster will change. Nodes that are part of multiple clusters must inform the other clusters of the new resource situation. If the available resources of the node were reduced, the other clusters will disable solutions of their solution sets. If the available resources have increased, the other clusters might enable currently disabled

solutions. If such a cluster now discovers a more complete solution, this solution is not implemented. The reason is that the overall behavior of the system becomes difficult to control. If a cluster implements a new solution, this will again result in changed resource availability on its nodes. When the other clusters, present on these nodes, are notified of the resource change, they in turn might start to change their configuration and so on. Because of that, new feasible solutions only are enabled, but no new cheapest solution is selected and implemented. The increased set of enabled solutions can be of use for future requests to the clusters, such as an increase of the service quality of an agent.

Reducing the Load of a Node

A cluster tries to place its agents independently of the other clusters. If the resource availability is low, this strategy might not always result in a complete solution. Instead of choosing an incomplete solution, where one or more agents are not assigned to a node for execution, the cluster can try to free additional resources to be able to find a complete solution. In the example presented in figure 5.6, cluster B might not be able to assign all of its agents to a node. Nodes 2 and 3 are also members of cluster A and might already be executing agents of cluster A. This will reduce the available resources for cluster B. Cluster B now requests cluster A to reduce the consumed resources on nodes 2 and 3. If cluster A is able to do so, it will report the amount of resources that would become available to cluster B. But cluster A does not yet really free these resources. Nodes 2 and 3 only consider quality of service level reductions for agents of cluster A running on node 2 and 3. If relocations of agents in cluster A were taken into account as well, this could lead to unpredictable execution time behavior for the whole process.

Cluster B now rechecks its solution set to see if any complete solution, or a solution containing more running agents, becomes applicable with the additional free resources. If one or more such solutions become applicable, the cheapest one is selected. Cluster B now asks cluster A to really free the resources previously promised. After cluster A has freed the resources, cluster B can implement the selected solution.

If no complete solution could be found, cluster B will select and implement and incomplete solution and the agents of cluster A will not change the quality of service level they are running at. After the solution was implemented, the other clusters are notified of the changed resource situation.

5.2.6 Event-Based Redistribution of Load

After the initial allocation of agents to nodes was computed, events can occur that require a dynamic reconfiguration of the placement of agents of a cluster. The process of solving the local constraint satisfaction problems for every node of the cluster, and the subsequent merges of the solutions, can take a significant amount of time. Some events, however, require a faster handling. The types of events that have been identified can be classified into two groups: The first group are events that lead to an increased requirement of resources or a decreased availability of resources. The second group are the opposite events, resulting in decreased resource requirements of an agent or increased resource

availabilities of a node. Both groups of events, and the measures taken to deal with them, are discussed in the following two sections. Not event every requires a full reconfiguration of a cluster.

Increased Requirements / Decreased Resource Availability

An increase of the resource requirements of a cluster can have numerous reasons. Adding an additional agent to the cluster, or increasing the quality of service level of an agent has this effect. Similar problems arise from decreased resource availability. Reasons for such a decrease could be hardware failures, or reduced performance due to the activation of a power saving profile. The load distribution system must find a new placement of the agents of a cluster that conforms to the new situation. A full reconfiguration, as with the initial placement of agents, can be done. A faster way that is not always applicable is the reduction of the set of solutions generated by the last configuration process. Such a reduction disables all solutions that can no longer be executed because of the changed resource situation. Finally, a new cheapest solution is selected and implemented.

Adding Permanent Agents A cluster affine agent is an agent that typically will be a member of the cluster for a longer time period, and is therefore referred to as a permanent member. When such an agent is added to the cluster, a full reconfiguration, as done at the initial placement of agents, is required. This means that the constraint satisfaction problem has to be solved with the new set of agents. Since resources are limited, the permanent addition of an agent can mean that not all of the agents of the cluster can run at the best quality of service level. The worst case is that not all agents can be allocated to a node. The agents not executed, are those with the lowest cluster affinity.

If suspending agents is not acceptable, resources have to be freed. To do so, the load distribution tries to reduce the load of the cluster nodes. It identifies the nodes that also belong to other clusters. Those other clusters are then asked to reduce the load of the nodes as described in section 5.2.5. If this is not successful, suspending tasks with low cluster affinity is not avoidable. If the other clusters are able to free resources, the cluster that is hosting the new agent will recheck its solution set and enable those solutions that have become feasible.

When finally a placement is found and implemented, this will result in a decrease of the available resources on the cluster nodes. All other clusters that also run on the same nodes have to be notified to be able to update their set of solutions, as described in section 5.2.5.

Adding transient agents Some scene affine agents are not long term members of a node or cluster. This is especially true of tracking agents following some object from node to node. Such agents only stay for a relatively short period of time and therefore require a special handling. Doing a full reconfiguration would take too much time. If a transient agent arrives at a node, the following strategy is applied: If not enough resources are available for the agent, in a first step it is checked if enough resources can be freed by reducing the quality of service level of cluster affine agents. In this process, agents of all

clusters present on this node are considered. If not enough resources can be freed on one of the DSPs of the node, also the service quality of scene affine tasks is reduced. If still not enough resources are available, agents are suspended. The order for suspending agents is defined by their cluster affinity costs. Agents with high cluster affinity are suspended first.

Handling agents with the default mechanisms of the load distribution, such as setting the desired node of a scene affine agent and doing a full reconfiguration, is not practical for transient agents. This process would take too much time. Since transient agents only stay on a node for a short period of time, it can be accepted that some other agents of the node operate at reduced quality or even are suspended. Handling transient agents and special tracking issues are covered in [Qua05].

Increasing the quality of service level Increasing the quality of service level of an agent means that the agent will require additional resources. All, not currently disabled agent placements that are part of the final solution set will be examined. Those solutions that do not support the required quality of service level for the agent are disabled. Of the remaining solutions the one with the lowest cost is selected and applied.

If this reduction of the set of feasible solutions leads to an empty solution set, the current agent placement will not be modified. The load distribution mechanism then tries to reduce the load of the cluster as described in section 5.2.5. The additionally freed resources might enable currently disabled agent placements. The solution set is rechecked if any of these new placements satisfy the increased quality of service requirements of the agent. If multiple such solutions are found, the cheapest one is selected. Before it is implemented, the other clusters are now asked to really free the resources previously promised. If no such solution can be found, the quality of service level change will be aborted unsuccessfully.

Removing a node The removal of a node from a cluster greatly reduces the available resource. It is very unlikely that the solution set contains placements that do not require the removed node. Nevertheless, the first step is to reduce the solution set by disabling all placements that contain the removed node. If the remaining solution set is not empty, the cheapest solution is selected and applied. Otherwise a complete reconfiguration of the shrunk cluster is started. If the removed node was a member of multiple clusters this reconfiguration has to be done for each of them sequentially.

A DSP is disabled or running with reduced performance To save energy, it might be required to reduce the performance of a DSP. This will result in reduced capabilities of the node. From those clusters present on this node, the one is selected causing the highest load. It is asked to reduce its solution set by disabling all placements that have become infeasible due to the reduced resource availability. If the resulting solution set is not empty, the cheapest solution is selected and implemented. If the reduced solution set is empty, the other clusters hosted by the node are asked to reduce their result set the same way. Because of the decreased resource availability, it cannot be guaranteed

that a solution is found where all agents of all affected clusters are assigned to a node for execution. Agents with high cluster affinity costs might be suspended.

Decreased Requirements / Increased Resource Availability

A decrease of requirements can have multiple reasons. Removing agents from a cluster or the reduction of the QoS level an agent is running at, can have this effect. Adding new nodes to a cluster or enabling currently disabled DSPs results in increased resource availability.

Removing a permanent agent When some service is no longer required, the corresponding agent is terminated. At the node the agent was running on, additional resources become available. The allocation of the reduced set of agents to nodes has to be recomputed. This full reconfiguration of the shrunken cluster will lead to a new set of feasible solutions. The cheapest one is selected and implemented. Not finding a complete solution is not likely since the removal of an agent from the cluster has freed additional resources. For the reconfiguration of the cluster no hard time limit exists: The agents of the cluster are currently running. The new solution would, at most, lead to an allocation where the agents are running at a better QoS level. After the new solution is implemented, the other clusters are notified of the changed resource situation as described in section 5.2.5.

Removing a transient agent Some scene affine tasks, such as trackers, that are short term members of a node or cluster require a special handling. When they arrive at a node, resources are freed by reducing the QoS level of agents that are executed on the node. If this does not lead to sufficient available resources, agents are suspended. After the transient agent has left the node, the agents are resumed or their QoS levels are increased again. A more in-depth discussion of the handling of tracking agents and short-term node members can be found in [Qua05].

Decreasing the quality of service level Decreasing the quality of service level of an agent results in a lower resource utilization of the node that executes the agent. Resulting from a previous increase of the QoS level of agents, some solutions might have become disabled. When the quality of service level of an agent is decreased, currently disabled solutions might become enabled again. The set of feasible, enabled solutions increases. This set is now rechecked. If a new cheapest solution is found, it is implemented. As a consequence, other clusters are notified of the changed resource situation as described in section 5.2.5.

A decrease of the QoS level of an agent does not require a full reconfiguration of the cluster, but can be handled by reenabling solutions. The implementation of a new cheapest solution can lead to down-times of agents due to migrations, DSP changes or QoS level changes. To avoid these down-times, it can be considered to only update the solution set, but not to implement the new cheapest solution. The approach taken, depends on what is more important to the system operator: an uninterrupted service or agents running at the best quality possible.

Adding a node When a node is added to a cluster, this means that new resources become available. To take advantage of these resources, a complete reconfiguration of the cluster is required. The reconfiguration of the cluster is done as described in sections 5.2.4 and 5.2.5.

A DSP is reenabled or its performance is increased At the time a DSP was disabled or its performance was reduced, solutions have been disabled. When the DSP is reenabled again or its performance is increased, the inverse approach can be taken. The set of solutions is checked with the additional resources in mind. The new resources might allow to enable currently disabled solutions. This could lead to a new cheapest solution that can be implemented. After the implementation of this solution, the other clusters are notified to be able to update their solution sets (see section 5.2.5).

As with a decrease of the QoS level, it depends on the priorities of the system operator if the new cheapest solution should be implemented or not. Applying the new solution might increase the quality of some agents, but will also lead to a short-term interruption of agents because of migrations, DSP or QoS changes.

5.2.7 Comparison With Other Load Distribution Systems

This section discusses the elements of the proposed load distribution mechanisms in reference to the classification of load distribution systems presented in section 2.3.

The *load metric* is focused on the resource utilization of the digital signal processors. Measuring their load not only involves one specific value such as the CPU utilization, but also takes into account memory, memory hierarchy, and transfers on the PCI bus. *Load communication*, on the other hand, is not such an important component. The load of a node and its digital signal processors is not communicated to other nodes. The information is only required locally for solving the constraint satisfaction problem. The only exception is the update of a solution set due to changed availability of resources: In such a case, the node doing the update queries all nodes with changed resources about their current load status. This polling of nodes takes place relatively seldom.

For the *transfer mechanism*, the mobility of agents is exploited. Every task of the system is implemented as an agent and can therefore be moved to any node easily. The *activation policy* is based on events. Aside from the initial placement, those events can, for example, be added or removed agents, or hardware changes. Events can be either triggered by the system operator or by an agent. The *candidate selection* and *location selection* result from solving the distributed constraint satisfaction problem. Solving the CSP can result in multiple solutions that would allocate the agents of a cluster to its nodes. To be able to pick the best solution, a cost system is used that assigns costs to resources, data transfers, migrations and affinities of an agent. The cheapest solution is selected and applied.

Chapter 6

Implementation Aspects

This chapter presents implementation specific details. Section 6.1 briefly discusses the selected agent platform, and the modifications and extension that have been done. It also describes the communication paths and mechanisms of the system. It is followed by a discussion of the implementation of the constraint satisfaction problem and the merge algorithms in section 6.2.

6.1 Diet Agents Platform

Mobility of agents is an important topic in the SmartCam project. Agents must be able to move to a different node if, for example, the load situation of a cluster changes. Hence, a fundamental requirement for the agent platform is support for agent migration. Conventional agent platforms such as Voyager [Rec] support this feature. These systems require the Java 2 Standard Edition for execution. But the J2SE is not available for the XScale architecture as used in the SmartCam project. Agent systems explicitly targeting embedded environments, such as CougaarME [Cou02] or LEAP [LEA05] are built for the Java 2 Micro Edition. Both systems do not support agent mobility.

The Diet Agents[1] platform originates from a research project funded by the European Union. As a result of the project, the Diet Agents platform was released as open source. It is designed to be conservative with system resources, which makes it interesting for use in embedded systems. Additionally, agent migration is supported. Diet is an abbreviation for *Decentralised Information Ecosystem Technologies*, reflecting the original goals of the Diet project. It is inspired by natural ecosystems where the overall functionality of a system arises from the collective behavior of many, very simple organisms [MKvL+01]. The Diet Agents system is designed to support a large number of simple agents. The actual processing of information is not meant to take place inside the agents but through communication between the agents.

[1] Diet Agents website: http://diet-agents.sourceforge.net

6.1.1 Available Functionality

The Diet Agents framework has a three layer architecture. The core layer, also referred to as the kernel, is designed to be very minimalistic and light-weight. It provides an environment for the agents to live in that supports the basic operations to create, destroy and move agents. Agent communication is also supported, but is limited to connections to agents within the same environment. Remote communication is not supported by the core layer. Agents do not hold direct references to other agents, even if both are executed in the same Java virtual machine and belong to the same environment. All communication is done using messages. To identify agents, they are labeled with a unique binary *name tag* that is randomly generated when the agent is created. In addition to that, a *family tag* is supported that allows to group agents. To send a message to an agent, its name tag must be known. Alternatively messages can also be sent using the family tag of an agent. The family tag is not unique – if multiple agents with the same family tag exist, one of them is randomly selected as the receiver of the message [HWBM02]. Diet Agents provides no lookup or directory service. The reason is, that directory services do not scale well and the Diet Agents system is designed to imitate ecosystems with a large number of agents.

Communication is done asynchronously. Messages sent to an agent are not directly processed by the receiver, but are placed in its incoming-messages buffer. The caller is not blocked, but continues its operation after the message has been sent. In contrast to implementations using proxy mechanisms, the caller is not blocked until the callee sends a response. That means that agents have to implement message handling routines. The Diet Agents supports this by associating a user defined context with every connection. This makes it easier to track the state of a connection. Diet Agents is designed to avoid system overloads. The number of available threads, concurrently active connections and the size of the message buffers can be configured. In situations of high load, messages and connections will be rejected. This allows the caller to adapt its behavior and fits well into the concept of self-adjusting ecosystems.

Remote communication is not a core service of the Diet Agents platform but is located in the Application Reusable Components (ARC) layer. Different types of remote communication are available [Bon04]. For simple, infrequent remote communication, carrier agents, also referred to as carrier pigeon agents, can be used. The carrier agent takes the message, migrates to the host of the receiver and then delivers the message locally. For frequent communication the use of carrier agents is too complex. Mirror agents provide a more comfortable mechanism for remote communication. By using mirrors, an agent in environment A can communicate with an agent in environment B as if both were in the same environment. In the environment of agent A, a mirror agent representing B is created. The same is done for agent A in B's environment. Agent A now interacts with the local mirror agent of B. The mirror agent forwards the requests, using carrier agents or message channels, to the mirror agent of A in the environment of B. The mirror agent of A now finally delivers the message to B. For both, A and B, this mechanism is transparent. Both interact with their local mirrors as if they were communicating with their real partners. A third mechanism for remote communication is the `MessageChannelProvider`.

It is used for remote communication in the SmartCam project and will be discussed in section 6.1.2.

The third layer of the Diet Agents system is the application layer. It contains all application specific data structures, logic and agents.

6.1.2 Extensions for the SmartCam Project

For the SmartCam project, several extensions to the functionality provided by the Diet Agents platform have been implemented. One of the most important aspects is a wrapper that provides simple remote communication. Diet Agents supports a socket based remote communication mechanism, called `MessageChannelProvider`. An agent that wants to be called from remote, first has to send a "listen" message to the `MessageChannelProvider`. In reply, it receives a message containing a channel id representing the address by which the agent can be contacted from the outside. A preferred channel id can be added to the initial "listen" message. This allows to provide services at "well known" channel ids, similar to the concept of ports in TCP/IP. When the `MessageChannelProvider` receives an incoming message channel with this id, it creates a local connection to the agent that is waiting for messages with this channel id. Incoming messages are then forwarded to that agent.

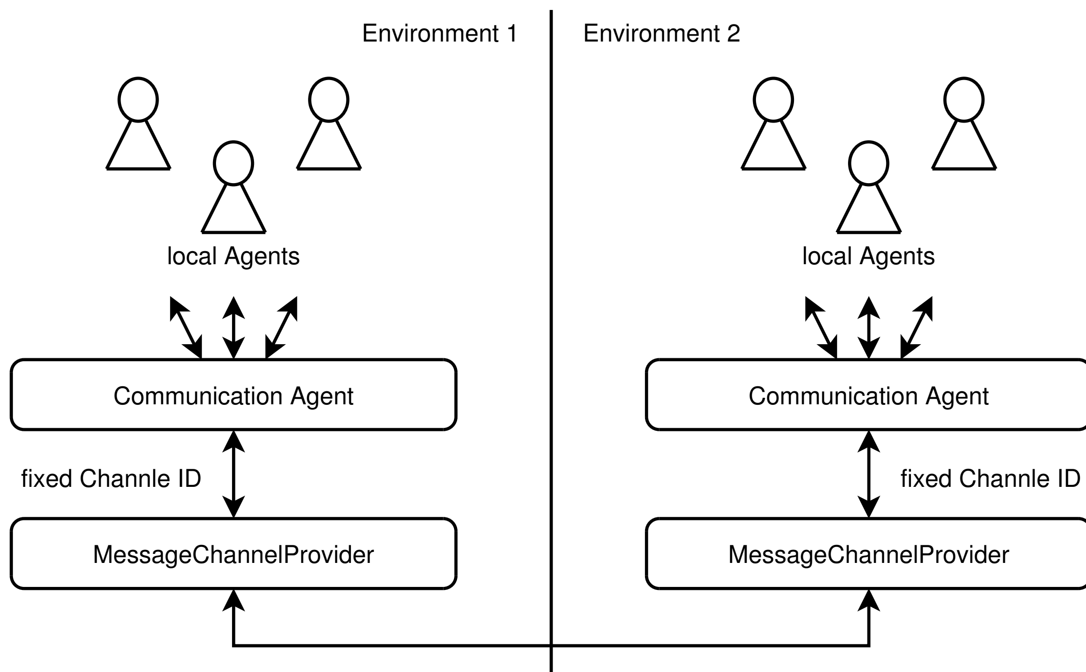


Figure 6.1: Remote communication mechanisms are provided by the Communication-Agent. The messages are transmitted using a `MessageChannelProvider`.

To simplify this process, the SmartCam project introduces a `CommunicationAgent`. The `CommunicationAgent` is the only agent that is directly connected to the `MessageChannelProvider`. All other agents that want to send remote messages only interact with

the `CommunicationAgent`. By setting the name tag length of the `CommunicationAgent` to zero, only one instance per environment is allowed. Hence, other agents can easily connect to the `CommunicationAgent` using its family tag, without the need to know its address. Normal agents do not have to take care about message channels ids or registering themselves as "listening" with the `MessageChannelProvider`. In case a message should be sent to a remote agent, the sender connects to the local `CommunicationAgent` and hands over the message along with the address of the receiver. The `CommunicationAgent` then sends the message, via the `MessageChannelProvider` to the destination host. For the channel id a well-known value that is used by all `CommunicationAgents` when connecting to the `MessageChannelProvider` is used. The message is forwarded to the `CommunicationAgent` on the destination system. It then extracts the address of the recipient agent and opens a local connection to this agent. The message is finally delivered to the recipient. The local connections between the `CommunicationAgent` and the sender agent, and the `CommunicationAgent` and the receiver agent remain open. They can be used to directly send further messages, until one of the agents at the connections ends, closes the connection. The setup used for remote communication, including the `CommunicationAgent` and the `MessageChannelProvider` is shown in figure 6.1.

To make agent development easier, a synchronous mechanism for sending messages, originally not provided by the Diet Agents platform, is introduced. Synchronous calls block the caller until an answer from the callee is received. Avoiding asynchronous handling of response messages helps to reduce the complexity of agents. The implementation of the synchronous call mechanism waits for a corresponding response after a request message has been sent. While waiting, the agent gives up its thread. Because the Diet Agents system relies on message passing, in the meantime other messages not related to the call with the pending response might be received. These messages are then handled in the agent context. Hence, great care must be taken to avoid synchronization problems.

To sum up, the SmartCam project extends the communication infrastructure of the Diet Agents platform in two ways: A simple way for remote communication is added, which avoids that every agent that wants to get in contact with a remote agent has to deal with channel ids and the `MessageChannelProvider`. The second enhancement is the addition of synchronous calls that allow an agent to block until the response to its request is received.

As described before, the Diet Agents platform does not provide a lookup or directory service. This feature, however, is required for the SmartCam project. In addition to that, a mechanism to group nodes into clusters has to be implemented. Both requirements, the directory service and the cluster management, are designed and implemented as described in section 5.1. It supports the grouping of nodes into clusters and keeps track of the agents belonging to nodes and clusters. Additionally, the cluster memberships of a node can be enumerated.

The Diet Agents system provides visualization support for the agents hosted on a node, using an external tool called *Elvis*. Elvis has to be started locally on every node before the agency is started. Although Elvis is an useful tool, especially for debugging, it is not practical for setups involving multiple hosts. For the SmartCam project, Elvis is extended in such a way that it can be attached to a remote node at runtime and visualize the state

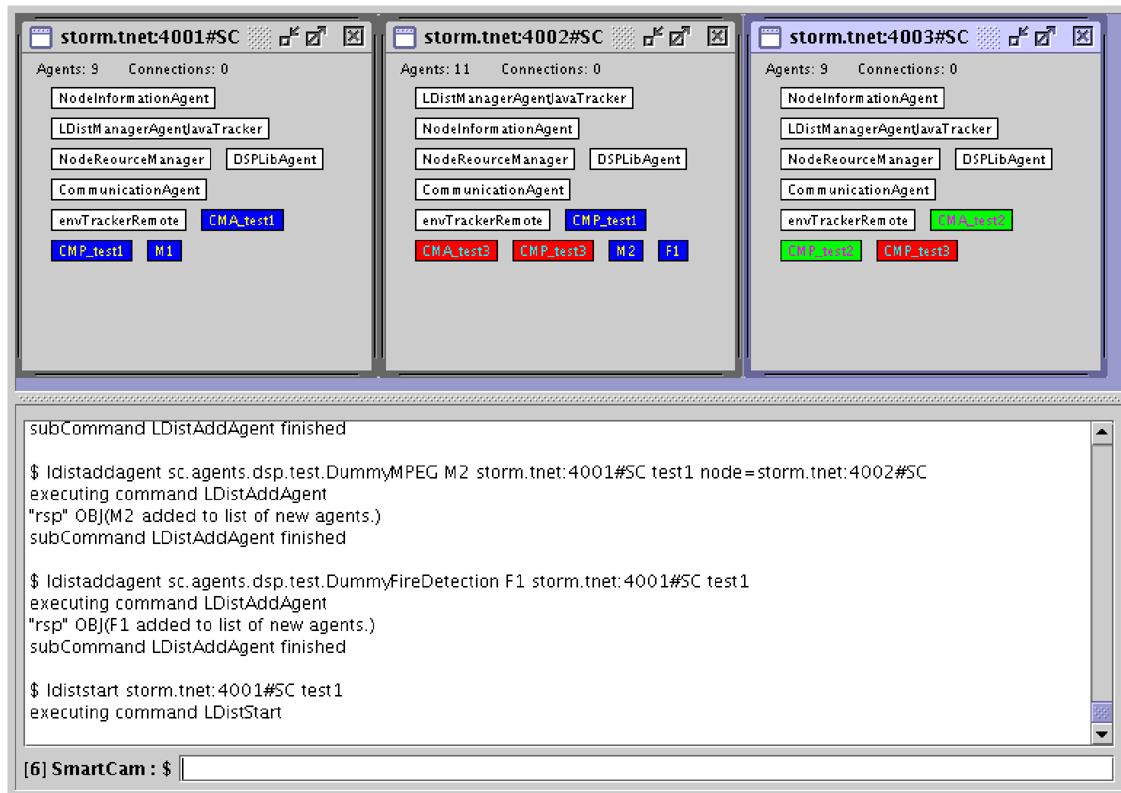


Figure 6.2: The SmartCam Console with three visualization windows showing the agents of three different agencies.

of the remote agency. The visualization shows the agents currently on a node. Cluster memberships are represented by the colors of the agents. In addition to that, Elvis is integrated into the SmartCam console, a graphical user interface (GUI) used for sending commands to remote nodes. A screenshot of the SmartCam console with the command prompt and three visualization windows is shown in figure 6.2. For environments without GUI support, command line management tools are available. A simple scripting language is provided that allows for the aggregation of multiple commands.

For the SmartCam project, the Diet Agents platform is extended in several ways. The most important difference compared to the original intentions of the Diet Agents project is that the SmartCam project does not focus on deploying a very large number of small agents, but on deploying a relatively small number of agents that are equipped with much more internal logic.

6.2 Implementation of the Distributed CSP

This section describes implementation details of the constraint satisfaction problem algorithm. It also discusses the merging of solutions. Some of the higher level considerations

from chapter 5 will be presented in more detail. To illustrate the explanation of this section, the following set of agents is used:

$$V = \{A, B, C\}$$

6.2.1 Computing the Agent Placement for Every Node

For the allocation of agents to a digital signal processor, a constraint satisfaction problem has to be solved. It is unlikely that all agents of a cluster can be executed on one DSP. Hence, the solution of the CSP for a single DSP hardly will contain a complete solution where all agents are executed. This means, that the goal when solving the CSP is not to find a complete solution, but to generate all partial solutions where zero, one or more agents are executed on the specific DSP. Complete solutions typically are found when the partial solutions from all the CSPs are merged.

All possible solutions, including partial ones, can easily be generated by a naive generate-and-test approach. All placements of agents are generated sequentially and then tested for their feasibility. A placement of agents is feasible, if the sum of the required resources of all agents contained in the placement is less then, or equal to the amount of available resources. For n agents, 2^n placements have to be generated and then tested for their feasibility. The generate-and-test algorithm will discard solutions repeatedly for the same reason: If, for example, the accumulated resource requirements of agents A and B already exceed the amount of available resources, it makes no sense to generate-and-test a placement that contains A, B and C.

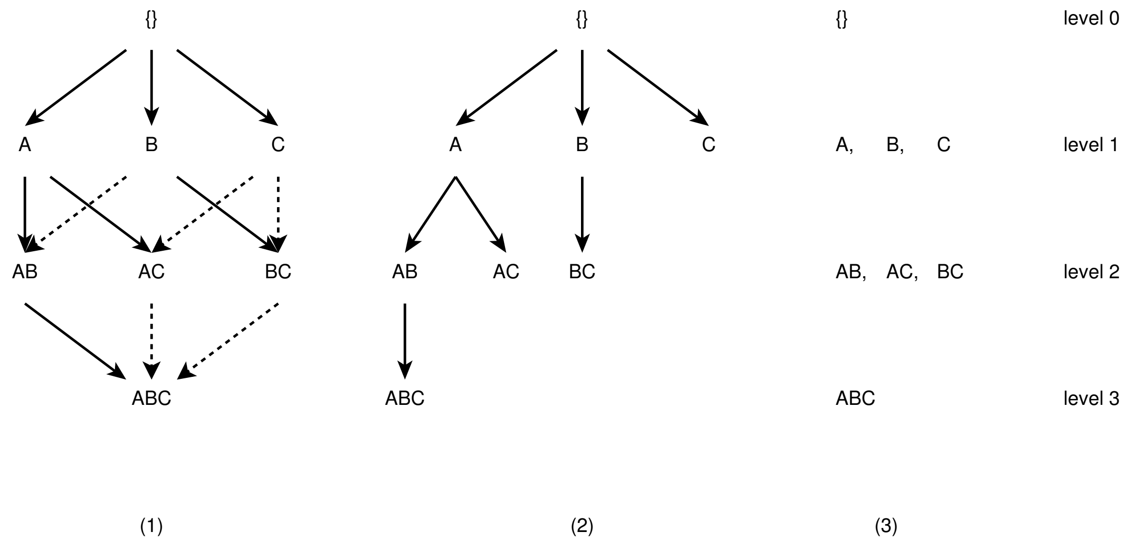


Figure 6.3: (1) shows all the 2^3 possible placements of the agents from V. A placement can be generated in more than one way, represented by the dotted lines. (2) avoids the multiple paths leading to a placement. Finally, (3) does not include any path information between the placements.

Backtracking provides an improvement over the generate-and-test approach. When agents A and B cannot concurrently be executed on a DSP due to resource constraints, other placements also containing A and B will not be generated. Backtracking will go back up and select another value instead of $B - C$ as shown in figure 6.4. Further improvements of backtracking, such as the *minimum remaining values* or the *least constrained value* heuristics, cannot be applied. They only offer a benefit if just one solution of the CSP is required. In the SmartCam project, however, all solutions, including partial ones, must be found.

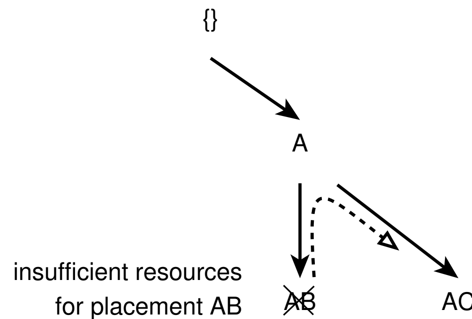


Figure 6.4: The placement AB is not possible due to insufficient resources. Backtracking goes back and tries to instantiate the next variable, in this case C.

The algorithm used in the SmartCam project starts with an empty set. In the next step, the solutions that only contain one agent are added (figure 6.3 (1), level 1). In level 2, to all placements of level 1, agents are added so that level 2 contains all possible placements of two agents. These placements are not unique: The placement AB can be reached via two different paths. The first path reaching a placement is drawn with a solid line, subsequent, redundant, paths are drawn with a dotted line. Figure 6.3 (2) shows the graph that omits the redundant dotted lines. To avoid permutations, which are the reason for the dotted lines, only agents are added to placements that are farther on the right in the set V than the already placed agents. For example, starting from B in level 1, only C is added resulting in BC in level 2. Combinations of B with other elements of V are already part of level 2.

Listing 6.1 shows a pseudo-code version of the algorithm presented above. The backtracking, and the resulting reduction of the search space, is done implicitly by not storing an infeasible solution. The algorithm always operates on the placements of the last level. If an infeasible solution was not added to the level, the sub-tree of this solution will not be examined. Placements that cannot be executed because of the resource consumption of agents from other clusters, are rejected as described in section 5.2.5.

The worst case runtime of $O(2^n)$, where n is the number of agents, to generate all 2^n possible placements is reduced in practice by the resource constraints. Typically, not more than about two to four agents can be placed concurrently on a DSP, resulting in a significant reduction of the search space. Multiple quality of service levels of agents can significantly increase the running time. Every QoS level has to be considered separately when solving the CSP. It has similar effects as the introduction of additional agents. To

```
1  levels[0].add(empty_set)
2
3  // the maximum number of levels is the same as the number of agents
4  // if resoruces permit, the last level holds a complete agent placement
5  for i = 1 to #agents
6
7    // take the placements from the last level ...
8    foreach placement in levels[i-1]
9
10     // ... and extend them by adding an agent on the right
11     //     of already placed agents
12     for j = (current_placement.rightmost_agent + 1) to #agents
13       new_placement = copy_of(current_placement)
14       new_placement.place_agent(j)
15       new_placement.calculate_costs
16
17       // only add new placement if it can be executed
18       if (new_placement.required_resource <= available_resources AND
19           new_placement.costs < early_pruning_threshold)
20
21         levels[i].add(new_placement)
22       end if
23
24     end for
25   end foreach
26 end for
```

Listing 6.1: Pseudo code of the CSP algorithm

limit the additional overhead, the early pruning mechanism (see section 5.2.3) can be used to reduce the search space. If the costs of a placement exceed a predefined threshold, the placement is not added to the current level (see line 19 of listing 6.1).

6.2.2 Merging Solutions

The constraint satisfaction problem is solved individually for every DSP of a node. These solutions then are merged into one single solution of the node. Subsequently, the individual node solutions are merged, resulting in one overall cluster solution. For a merge, not only the elements of two solutions are put into the new solution, but also new agent placements are generated by combining individual placements from the two merged solutions. Figure 6.5 shows solutions 1 and 2 that should be merged. The solutions result from solving the CSP for two different DSPs or nodes with different resource availability. In the merged solution, those placements surrounded by a box are newly generated as combinations of placements from solutions 1 and 2. The only limitation is, that no two placements can be combined that have one or more agents in common. Level n of a solution contains all elements where n agents are placed.

To merge solutions 1 and 2 from figure 6.5, all placements of solution 1 are visited. For the current placement of solution 1, referred to as placement 1, the tree of solution

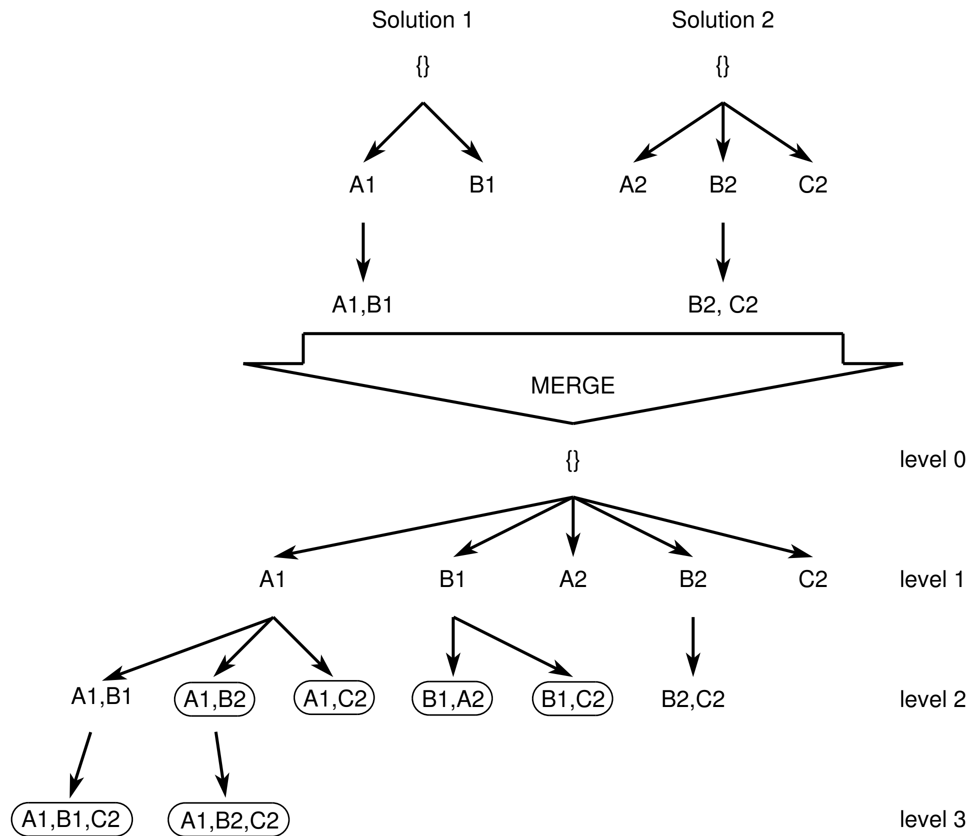


Figure 6.5: Solution 1 and solution 2 are merged. The boxed elements of the merged solution have been generated during the merge.

2 is traversed from top to bottom. If placement 1 does not intersect with the current placement of solution 2, called placement 2, a new placement is created that contains the merged placements 1 and 2. This new placement is then added as a new child below placement 1.

If placements 1 and 2 intersect, which means that they have one or more agents in common, they cannot be merged. As a consequence, placement 2 and all placements below it, are rejected. If placements 1 and 2 intersect, also all placements in the sub-tree below placement 2 will intersect with placement 1. Exploiting this property, many unsuccessful attempts to merge placements of solutions 1 and 2 can be avoided. Not only intersections of two placements result in rejection of merged placements. The early pruning mechanisms from section 5.2.3 is also used when merging solutions. Whenever the costs of a merged placement exceed a predefined threshold, the placement, and its sub-tree, are rejected. If already the costs of placement 1 merged with placement 2 exceed the early pruning threshold, then this will also be the true for all placements below placement 2. Listing 6.2 presents a simplified pseudo-code version of the merging algorithm.

For the merged solution, the number of placements greatly increases. As detailed in section 5.2.3, the number of placements contained in the merged solution is given by

```
1 foreach placement in solution 1
2   traverse solution 2 top to bottom
3     if (placement1 and placement2 do not intersect AND
4         placement1.costs + placement2.costs < early_pruning_threshold)
5
6         merged_placement = merge(placement1, placement2)
7         placement1.addChild(mergedPlacement)
8     else
9         do not go down further in solution 2
10        but try the next element right/above of placement2
11
12        // if placement1 and placement2 intersect (or have too high costs),
13        // all placements below placement2 will intersect with placement1
14        // (or generate too high costs)
15     end if
16   end traversal
17 end foreach
```

Listing 6.2: Pseudo code of the merge algorithm

$O((numDSPs1 + numDSPs2 + 1)^n)$ where $numDSPs1$ and $numDSPs2$ denote the number of DSPs covered by solutions 1 and 2 respectively and n denotes the number of agents. For a node equipped with four DSPs, the maximum number of placements is given by $O(5^n)$. For a cluster containing four such nodes, the size of the final set of placements is given by $O(17^n)$. For these numbers, no resource constraints are considered. As a consequence, the number of placements will be significantly reduced in practice because of the limited resources of the nodes and the early pruning mechanism.

6.2.3 Enhancing Java Object Serialization

During the implementation phase, performance problems related to network communication and object serialization have been discovered. Not all virtual machines are equally affected of the problems. The Java Runtime Environment 1.4.2 does not show any performance decrease when serializing objects. Other virtual machines, such as JamVM, do not perform well. The reason for the problem is, that object serialization relies on the reflection feature of the Java language. Reflections are used to determine the type of the objects to be (de-)serialized. Some virtual machines do not perform well with reflections. This problem was also known for early implementations of the JRE. A method to alleviate the problem of bad reflection performance, is to do the serialization of objects explicitly [MR01]. This is done by implementing the `Externalizable` interface instead of the `Serializable` interface. The methods `writeExternal` and `readExteral` must be implemented such that all attributes of the object are written to, and read from the `ObjectStream`. By doing this explicitly, the use of reflections can be reduced. Depending on the VM, this results in a significant speed-up for object serialization as shown in the evaluation in section 7.1.4.

Chapter 7

Evaluation

This chapter presents the practical results of this work. The load distribution system described in chapters 5 and 6 is evaluated.

Section 7.1 describes the testbed environment for the evaluation, followed by measurements taken with different test scenarios in section 7.2. The results are discussed and compared with the expected system behavior.

7.1 Testbed

The testbed for the evaluation consists of multiple machines presented in table 7.1. Two SmartCam prototypes, as described in chapter 4, are available. To be able to test the system with other cluster sizes, conventional desktop PCs are used. All machines are connected via a switched 100MBit ethernet network.

Name	CPU	Memory	Architecture	Operating Systems
SmartCam1 (SC1)	IXP425, 533MHz	256MB	XScale big endian	Linux 2.6.8, custom
SmartCam2 (SC2)	IXP425, 533MHz	256MB	XScale big endian	Linux 2.6.8, custom
PC1	Pentium III, 1GHz	256MB	x86	Linux 2.6.8, SuSE 9.2
PC2	Pentium III, 1GHz	256MB	x86	Linux 2.6.8, SuSE 9.2
PC3	Pentium III, 1GHz	256MB	x86	Linux 2.6.11, Debian SID

Table 7.1: Machines of the testbed.

7.1.1 Java Runtime Environments

On the XScale big endian platform, not all of the embedded Java options presented in section 2.2 are available. For the tests, JamVM 1.3.0 using GNU Classpath 0.14 is used. On x86 based hardware, the performance of the SUN Java Runtime Environment (JRE)

1.4.2 and JamaicaVM 2.6 is compared with JamVM 1.3.0 for x86. The JRE is of interest because it represents the official implementation of the Java standard. It is, however, not available for the XScale prototype platform. In contrast to that, JamaicaVM 2.6 is already available for little endian XScale based systems, and a port to big endian systems might be available in the future. Table 7.2 summarizes the Java environments considered in the evaluation.

Name	Vendor	Licence	Architecture
JRE 1.4.2	SUN Microsystems	commercial	x86
JamaicaVM 2.6	Aicas GmbH	commercial	x86
JamVM 1.3.0	Robert Lougher	GPL	x86
JamVM 1.3.0	Robert Lougher	GPL	XScale, big endian

Table 7.2: Evaluated Java environments.

For a first comparison of the performance of the virtual machines, they are tested using the Embedded CaffeineMark 3.0 and the jByteMark benchmark programs [Han97]. The Embedded CaffeineMark 3.0 is a subset of the full CaffeineMark 3.0 from Pendragon Software that omits tests of the graphical user interface. On x86 based hardware (PC1 of the testbed), JRE 1.4.2, JamaicaVM 2.6 and JamVM 1.3.0 are tested. The JRE, with its JIT compiler, outperforms JamaicaVM 2.6 with the entire benchmarks compiled to native code by a factor of two in most tests. JamVM 1.3.0, which is only using an interpreter, remains behind the results of JamaicaVM 2.6 (native) by a factor of ten in CaffeineMark, and a factor of five in jByteMark. To compare the performance of the SmartCam prototype and the PCs of the testbed, the benchmarks are also done on SmartCam 1 using JamVM 1.3.0. With the exception of the floating point index of jByteMark, the benchmarks on the SmartCam are slower by a factor of two compared to the results on PC1. The low floating point index results from the missing floating point hardware unit of the XScale platform. The benchmark results are summarized in table 7.3. As with every benchmark, it has to be kept in mind that the measurements can only cover certain aspects of the virtual machines. But they give a first estimate of the expected performance.

Java Environment	Embedded CaffeineMark 3.0	jByteMark (Integer Index)	jByteMark (FP Index)
JRE 1.4.2 (PC1)	20924	120	63.4
JamaicaVM 2.6, interpreter (PC1)	323	1.7	1.4
JamaicaVM 2.6, native (PC1)	10499	52	46
JamVM 1.3.0 (PC1)	930	9.9	9.3
JamVM 1.3.0 (SmartCam1)	542	5.8	1.4

Table 7.3: Benchmark results of different Java environments. Higher numbers denote better performance.

7.1.2 Code Size

For an embedded system, not only computing power but also the amount of available memory is limited. Therefore, the code size of the agent system and the implemented applications on top of it, is of interest. Compared to other agent systems Diet Agents is relatively small. It is shipped as a single Java archive (JAR) without the need for additional, third party libraries. The size of about 500kB can even be further reduced by removing sample applications contained in the archive (see table 7.4). The additional infrastructure required for the SmartCam project including the implementation of the load distribution system requires additional 370kB of memory. The SmartCam console and GUI applications are not included in this number because they are not required on a camera but only on machines that are used for maintenance.

Name	Version	Size
Diet Agents	0.97	496kB
Diet Agents (modified)	0.97 (without samples)	354kB
SmartCam Applications	without console and GUI	363kB
SmartCam Applications	with console and GUI	516kB
JamVM	with 1.3.0 GNU Classpath 0.14	5.7MB

Table 7.4: Code size of the agent system, the SmartCam applications and the Java runtime environment.

The total memory required for the agent system and the SmartCam applications is about 720kB. This does not yet include the virtual machine and the Java class library required for execution. On the XScale platform, JamVM 1.3.0 together with the class library consumes about 5.7MB of space. Hence, the overall required space is 6.4MB. To reduce this size, parts of the class library that are not required, can be removed. To do that, an analysis of the applications, the class library and the inter-dependencies must be done.

7.1.3 DSP Applications

For the test of the load distribution system, the DSP applications and their resource requirements are simulated. To make the results as accurate as possible, the requirements of the MPEG-4 encoder and the stationary vehicle detection (SVD) algorithm were identified by profiling. The requirements of the other algorithms have been estimated based on these results. Many of the algorithms can operate at different quality of service levels. The lower the quality, the lower the resource requirements are. The reduction of the resource requirements results from using input data of lower resolution, or with less frames per second (fps). As a consequence, the quality of service is decreased. "Full frame" denotes that the algorithm operates on input images with a full resolution of 704x576. CIF stands for "Common Intermediate Format" and has a resolution of 352x288 pixels. Other used formats are "Quarter CIF" (QCIF) with 176x144 pixels and 2CIF with a resolution of 704x288. The DSP algorithms used for testing, including their QoS levels and their resource requirements, are presented in table 7.5.

Algorithm	QoS		CPU	RAM		DMA		
	Level	Description	[MIPS]	Int.	Ext.	Chan.	Tables	TCC
MPEG-4	0	Full Frame, 25fps	3840	400kB	0	7	3	5
MPEG-4	1	Full Frame, 12fps	1920	400kB	0	7	3	5
MPEG-4	2	2CIF, 25fps	1920	350kB	0	6	3	4
MPEG-4	3	2CIF, 12fps	960	300kB	0	6	3	4
MPEG-4	4	CIF, 25fps	960	300kB	0	5	2	2
MPEG-4	5	CIF, 12fps	480	300kB	0	5	2	2
SVD	0	CIF, 12fps	3600	500kB	17MB	25	6	25
SVD	1	QCIF, 12fps	900	330kB	4MB	25	6	25
Fire Detec.	0	CIF, 12fps	570	200kB	4MB	2	2	2
Vehicle Cnt.	0	CIF, 12fps	3600	500kB	17MB	25	6	25
Vehicle Cnt.	1	QCIF, 12fps	900	330kB	4MB	25	6	25
Vehicle Clas.	0	CIF, 12fps	4400	500kB	17MB	25	6	25
Wrong-Way	0	CIF, 12fps	4000	550kB	18MB	16	7	26

Table 7.5: Requirements of the DSP algorithms.

The capabilities and the resource supply of the digital signal processor used for evaluating the load distribution system are listed in table 7.6.

Name	CPU	MIPS	RAM			DMA		
			Int.	Ext. A	Ext. B	Chan.	Tables	TCC
NVDK	C6416, 600MHz	4800	1024kB	256MB	8MB	64	17	64

Table 7.6: Resource supply of SmartCam DSP.

7.1.4 Migration Performance

For a mobile agent system, the total time required by an agent to travel from one host to another, is of special interest. It not only includes the time required for transmitting the agent over the network, but also the time required for suspending and resuming the agent. The migration performance depends on the size of the agent and its complexity. Data structures, which are part of the agent have to be serialized before they can be transmitted over the network. At the destination, all data has to be de-serialized again before the agent can be resumed.

To measure the migration times of the Diet Agents system, an agent is used that migrates between two hosts several times. When the agent finishes, the average migration time is calculated. The payload, which is varied for different test runs, is a chunk of random binary data. Every test is repeated ten times. The average migration times are presented in table 7.7. As with the computation-oriented tasks of the benchmarks in section 7.1.1, JamVM is significantly slower than the JRE in communication-oriented tasks.

As described in section 6.2.3, measures are taken to improve the speed of the Java object serialization. This is of interest for agents that contain complex data structures.

Java Version	Hosts	Agent Size					
		0kB	32kB	64kB	128kB	256kB	512kB
JRE 1.4.2	PC1, PC2	60ms	63ms	64ms	70ms	95ms	169ms
JamVM 1.3.0	PC1, PC2	258ms	597ms	957ms	1730ms	3136ms	5984ms
JamVM 1.3.0	SC1, SC2	728ms	2696ms	4670ms	8471ms	16275ms	33239ms

Table 7.7: Migration times between two hosts for different agent sizes.

If such an agent is migrated from one host to another, all the data has to be serialized which can take a significant amount of time. To measure the gained speedup, the payload of the agent used for migration testing is modified. Instead of a binary chunk of data, a complex data structure is used. The data structure is a solution set containing 180 solutions as generated when solving the distributed constraint satisfaction problem. The test is repeated ten times. Table 7.8 presents the average migration time with and without enhanced object serialization. The results for JRE 1.4.2 do not differ in performance. This indicates that the reflection mechanisms of the JRE are highly optimized. In contrast to that, JamVM significantly gains from serialization enhancements.

Java Environment	Standard Serialization	Enhanced Serialization
JRE 1.4.2	470ms	465ms
JamVM 1.3.0, x86	14245ms	5848ms
JamVM 1.3.0, XScale	38256ms	10901ms

Table 7.8: Serialization performance of different Java environments.

7.2 Evaluation Scenarios

Two main scenarios are used to test the load distribution system. The first one consists of a single cluster and is used to test the basic functionality and behavior of the system. The second setup is more complex and involves three overlapping clusters. The goal of this scenario is to test the effects on the load distribution system when multiple clusters are present on the nodes. In addition to that, aspects of inter-cluster communication are discussed and tested.

7.2.1 Single Cluster Scenario

The first test scenario is based on a single cluster containing three nodes as shown in figure 7.1. Every node is equipped with two DSPs.

For all test cases of this scenario the following agents should be allocated to the cluster. Three MPEG-4 encoder agents are assigned to the cluster. All of them are only equipped with a single possible QoS level, the best one. In addition to that, a desired node is set in such a way that one MPEG-4 agent should be placed on every node of the cluster. Two more agents, one stationary vehicle detection agent and one fire detection agent, are

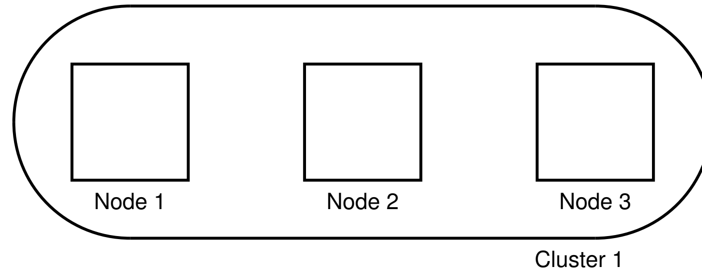


Figure 7.1: A single cluster containing three nodes.

assigned to the cluster. These five agents assigned to the cluster form the base of this scenario. If a test case diverges from this setup, it is explicitly mentioned.

Test Case 1 – Initial Placement

This test case examines the initial placement of the five agents of this scenario without further modifications.

Having a total of six DSPs and five agents, the maximum number of complete solutions is $6^5 = 7776$. With the resource and data transfer constraints of the test scenario, the load distribution system reports a total number of 2160 feasible, complete solutions. This number can be reduced by enabling the early pruning mechanism. Solutions with costs that are above the early pruning threshold are rejected during the computation of the CSPs and the subsequent merge operations. The size of the final set of complete placements is reduced.

pruning threshold	number of solutions	solutions received at	agents started at
5000	96	2461ms	3304ms
6000	144	2517ms	3344ms
7000	192	2853ms	3904ms
8000	672	3147ms	4219ms
not used	2160	3576ms	4662ms

Table 7.9: Results for the allocation of five agents with one QoS level using JRE 1.4.2 on PC1, PC2 and PC3.

Tables 7.9 and 7.10 summarize the running times for the initial placement of the agents for different early pruning cost thresholds. Both tests were conducted using PC1, PC2 and PC3 of the testbed. The lower performance of the JamVM compared to the JRE is clearly demonstrated by the running times. Table 7.11 presents the results when using PC1, SmartCam1 and SmartCam2. *Solutions received at* denotes the time from the start of the load distribution process until the set of final, complete solutions is received at the initiator. The *agents started at* time is the elapsed time from the point the load distribution process is started until all agents are created and started at their target nodes. The *number of solutions* denotes the number of placements contained in the final solution. Figure 7.2

pruning threshold	number of solutions	solutions received at	agents started at
5000	96	4737ms	6445ms
6000	144	4813ms	6433ms
7000	192	5703ms	7462ms
8000	672	7817ms	9468ms
not used	2160	9214ms	10961ms

Table 7.10: Results for the allocation of five agents with one QoS level using JamVM 1.3.0 on PC1, PC2 and PC3.

pruning threshold	number of solutions	solutions received at	agents started at
5000	96	10297ms	12609ms
6000	144	10573ms	12896ms
7000	192	11989ms	14652ms
8000	672	16697ms	19190ms
not used	2160	17660ms	20235ms

Table 7.11: Results for the allocation of five agents with one QoS level using JamVM 1.3.0 on PC1, SmartCam1 and SmartCam2.

presents a graphical comparison of the running times of the different Java environments and hardware platforms, until the final set of complete solutions is received at the initiator.

Agent Name	Node 1	Node 2	Node 3
MPEG Agent 1	DSP 0		
MPEG Agent 2		DSP 0	
MPEG Agent 3			DSP 0
Fire Detection Agent	DSP 1		
SVD Agent			DSP 1

Table 7.12: The cheapest solution for test case 1.

Table 7.12 shows the cheapest placement of the agents as selected by the load distribution system. The resource requirements of all agents are satisfied. The MPEG agents have been allocated to their desired nodes. For placements where the MPEG agents are not located on their desired nodes, the scene affinity costs, and as a result the total costs, are higher.

Test Case 2 – Agents With Multiple QoS Levels

For this test case, the three MPEG-4 encoder agents are now equipped with multiple QoS levels. In a first step, the number of QoS levels is increased to 2. In a subsequent step, three QoS levels are used.

Increasing the number of QoS levels has similar effects as increasing the number of agents. Every quality of service level is treated as if it were an own agent. The only

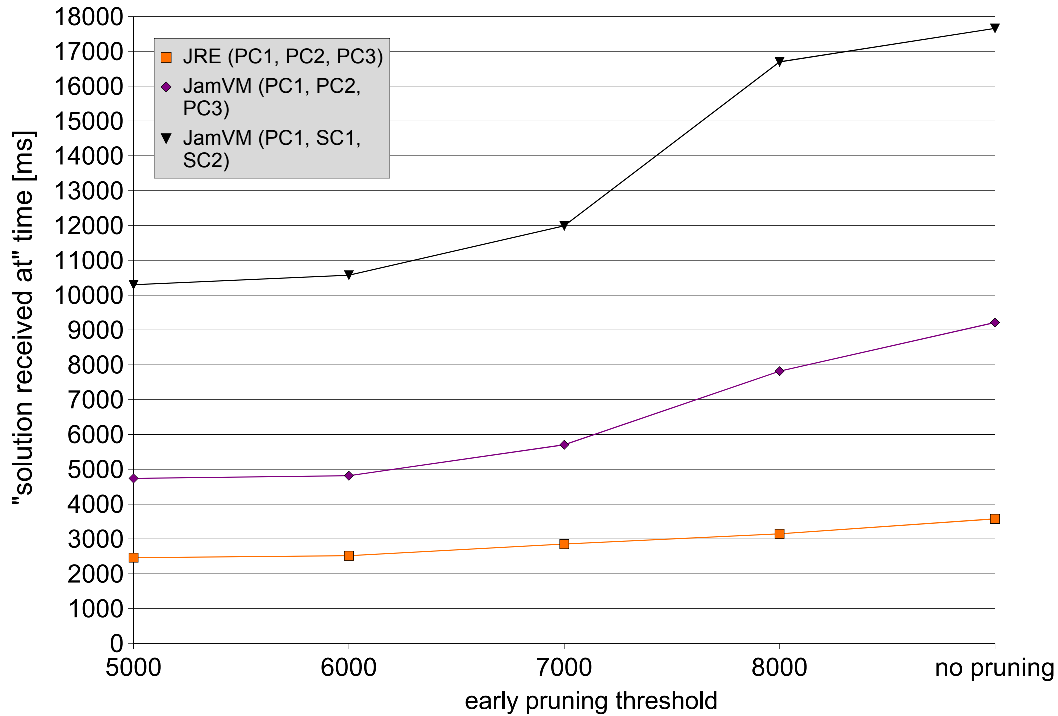


Figure 7.2: Running times until the final solution is received at the initiator using different virtual machines on PCs and SmartCams with several early pruning thresholds.

additional constraint is that one agent can only be contained once in a solution no matter what its QoS level is. Two really independent agents can both be part of a solution at the same time. The new upper bound for the maximum number of complete solutions, when all three MPEG encoder agents are equipped with two QoS levels, is $6^8 = 1.679.616$. Since the additional three QoS levels are not equivalent to three additional agents, the true upper bound is lower than this rough estimation.

Without early pruning, the load distribution system is unable to handle solution sets of this size. Table 7.13 therefore presents the numbers of solutions reported by the load distribution system for different early pruning cost values.

Solutions received at in table 7.13 denotes the elapsed time from the point where the load distribution is started until the final solution is received at the initiator. The final solution is the solution resulting from the last merge of two worker agents. When the final solution is received at the initiator, the cheapest one is selected and the agents are created accordingly. The *agent started at* time denotes the point where all of the agents are started on their correct note. Again, this time is measured from the point where the load distribution is started.

The load distribution process not only requires computation time for solving the CSP and merging the solutions. Additionally, time for workers to move with their solution from

pruning threshold	QoS levels	number of solutions	JRE 1.4.2 PC1, PC2, PC3 solutions received at	JRE 1.4.2 PC1, PC2, PC3 agents started at	JamVM 1.3.0 PC1, SC1, SC1 solutions received at	JamVM 1.3.0 PC1, SC1, SC1 agents started at
4500	0	96	2511ms	3342ms	9575ms	12137ms
	0,1	192	2670ms	3207ms	10196ms	12646ms
	0,1,3	192	2644ms	3485ms	10801ms	13427ms
5000	0	96	2478ms	3265ms	10533ms	12994ms
	0,1	384	2973ms	4029ms	11984ms	14650ms
	0,1,3	576	3293ms	4344ms	13619ms	16131ms
6000	0	114	2535ms	3348ms	10877ms	12277ms
	0,1	960	2942ms	4001ms	12696ms	15154ms
	0,1,3	2160	3641ms	4638ms	14921ms	17811ms

Table 7.13: Results for the allocation of five agents. The three MPEG agents are equipped with multiple QoS levels.

one host to another is required. To estimate how much time is spent with computation, communication and migration, table 7.14 presents the running times for solving the CSP and the subsequent merges. The numbers have been taken from the run with three QoS levels for the MPEG agents and a pruning threshold of 5000. Table 7.15 presents the results for a pruning threshold of 6000. Solving the CSP for the DSPs and the merge of the DSP solutions is done in parallel on every node. The merges of the individual node solutions are done pairwise for two nodes. Tables 7.14 and 7.15 also present the accumulated times for the critical paths of computations.

Task	JRE 1.4.2 PC1, PC2, PC3 Node 1	Node 2	Node 3	JamVM 1.3.0 PC1, SC1, SC1 Node 1	Node 2	Node 3
solving CSP for DSP 0	25ms	24ms	33ms	31ms	45ms	43ms
solving CSP for DSP 1	2ms	3ms	4ms	3ms	12ms	9ms
merging DSP solutions	6ms	8ms	8ms	22ms	41ms	39ms
merging: node 1 and 2	50ms			123ms		
merging: node (1, 2) and 3	674ms			3145ms		
total time (critical path)	759ms			3366ms		

Table 7.14: Running times for solving the CSP and subsequent merges. MPEG agents are equipped with three QoS levels, pruning threshold is 5000.

Table 7.16 shows the cheapest agent placement found by the load distribution system. It is taken from the run with three QoS levels for the MPEG agents and a pruning threshold of 6000 from table 7.13. Because of the quality of service degradation penalty QoS_{deg} all agents are running at the best QoS level 0. Placements with lower service quality are more expensive. All MPEG agents are assigned to their desired node. MPEG agents are scene affine tasks. If they are not running at their desired node, the scene affinity costs

Task	JRE 1.4.2 PC1, PC2, PC3 Node 1	Node 2	Node 3	JamVM 1.3.0 PC1, SC1, SC1 Node 1	Node 2	Node 3
solving CSP for DSP 0	60ms	23ms	34ms	11ms	92ms	43ms
solving CSP for DSP 1	2ms	3ms	2ms	3ms	12ms	10ms
merging DSP solutions	9ms	7ms	9ms	24ms	43ms	44ms
merging: node 1 and 2	89ms			103ms		
merging: node (1, 2) and 3	1121ms			3175ms		
total time (critical path)	1281ms			3425ms		

Table 7.15: Running times for solving the CSP and subsequent merges. MPEG agents are equipped with three QoS levels, pruning threshold is 6000.

are increased by the remote penalty factor. Hence, solutions where the MPEG encoders are assigned to their desired nodes are cheaper. It can be noted that not two agents are running on the same DSP although this would have been possible for the fire detection agent. It could be executed on the same DSP together with an MPEG agent or the SVD agent. This is discouraged because the resource costs of agents are computed as the ratio of required resources to available resources. Therefore, those DSPs are preferred for agent placement that have more available resources.

Agent Name	Node 1	Node 2	Node 3
MPEG Agent 1	DSP 0, QoS 0		
MPEG Agent 2		DSP 0, QoS 0	
MPEG Agent 3			DSP 0, QoS 0
Fire Detection Agent	DSP 1, QoS 0		
SVD Agent			DSP 1, QoS 1

Table 7.16: The cheapest solution for test case 2. All three MPEG agents are equipped with three QoS levels.

Test Case 3 – Adding Agents

For this test case, at first only the three MPEG encoder agents are added to the cluster. Then the load distribution is started. The fire detection agent, the stationary vehicle detection agent and an additional wrong way driver detection agent are added to the cluster. After every of those agents, the load distribution system is started and a new configuration is computed. For this test, early pruning is disabled.

with adding new agents, the set of complete solutions would grow exponentially if there were no resource constraints. Tables 7.17 and 7.18 also present the numbers of complete solutions for the six DSPs of the cluster without considering resource constraints. When compared with the numbers reported by the load distribution system, it can be seen that the growth is significantly limited by the resource constraints.

	3 Agents (MPEG)	4 Agents (+Fire)	5 Agents (+SVD)	6 Agents (+WrongWay)
number of solutions (no constraints)	$2^3 = 216$	$2^4 = 1296$	$2^5 = 7776$	$2^6 = 46656$
early pruning threshold: 6000				
number of solutions (with constraints)	8	288	24	4
solutions received at	2350ms	1882ms	1598ms	1959ms
agent started at	2976ms	2201ms	2094ms	2467ms
early pruning threshold: none				
number of solutions (with constraints)	120	720	2160	4320
solutions received at	2399ms	1929ms	2714ms	5293ms
agent started at	3026ms	2261ms	3225ms	5817ms

Table 7.17: Increase of solution set size for additional agents. JRE 1.4.1 on PC1, PC2 and PC3 with and without early pruning.

	3 Agents (MPEG)	4 Agents (+Fire)	5 Agents (+SVD)	6 Agents (+WrongWay)
number of solutions (no constraints)	$2^3 = 216$	$2^4 = 1296$	$2^5 = 7776$	$2^6 = 46656$
early pruning threshold: 6000				
number of solutions (with constraints)	8	288	24	4
solutions received at	9531ms	8375ms	7982ms	9850ms
agent started at	11535ms	9131ms	9690ms	11124ms
early pruning threshold: none				
number of solutions (with constraints)	120	720	2160	4320
solutions received at	8992ms	9474ms	13938ms	28328ms
agent started at	10853ms	10358ms	15186ms	29597ms

Table 7.18: Increase of solution set size for additional agents. JamVM 1.3.0 on PC1, SmartCam 1 and SmartCam 2 with and without early pruning.

The final placement selected by the load distribution system after all six agents have been added to the cluster is presented in table 7.19. The agents that were sequentially added to the cluster are printed in bold letters. When adding additional agents, the load distribution system tries to avoid migrations of agents that are already part of the cluster. In this test case, the MPEG encoder agents were not moved when the fire detection agent was added to the system, although such solutions are part of the final set of placements. This is achieved by the migration costs described in chapter 5. If, for example, the MPEG encoder agent 3 currently located at node 3 migrated to node 1, the migration costs would rise. If, however, the agent remained at its current location, the migration costs

would be zero. As a consequence, the cheapest solution that is selected at the end of the load distribution process will be one that omits migrations of agents. In this test case, migrations were not done when new agents were added.

Agent Name	Node 1	Node 2	Node 3
MPEG Agent 1	DSP 0		
MPEG Agent 2		DSP 0	
MPEG Agent 3			DSP 0
SVD Agent		**DSP 1**	
Fire Detection Agent	**DSP 1**		
Wrong Way Driver Detection Agent			**DSP 1**

Table 7.19: The final placement of all six agents.

Test Case 4 – Adding DSPs

Adding DSPs to a cluster has similar effects as adding nodes: The size of the set of complete solutions will increase polynomial in the number of nodes if resource constraints are not considered. To demonstrate this effect, only the three MPEG agents of this scenario are assigned to the cluster. Every node is equipped with two DSPs. In subsequent steps, the DSP count on every node is increased to three. After every addition of a DSP, a new configuration for the cluster is computed.

	6 DSPs	7 DSPs	8 DSPs	9 DSPs
number of solutions (without constraints)	$6^3 = 216$	$7^3 = 343$	$8^3 = 512$	$9^3 = 729$
number of solutions (with constraints, no early pruning)	120	210	336	504
solutions received at	2475ms	2493ms	2623ms	2603ms
agents started at	3041ms	3145ms	3348ms	2985ms

Table 7.20: Increase of solution set size for additional DSPs. JRE 1.4.1 on PC1, PC2 and PC3 without early pruning.

Tables 7.20 and 7.21 present the size of the solution sets reported by the load distribution system compared to the theoretical numbers without considering resource constraints. The relatively large solution sets reported by the load distribution system result from the fact that there are a lot of resources for a small number of agents. As a consequence, many different placements of these agents are possible. By enabling early pruning the size of the solution sets can be reduced. In this case, solutions where an MPEG agent is not running at its desired node are omitted due to high costs.

	6 DSPs	7 DSPs	8 DSPs	9 DSPs
number of solutions (without constraints)	$6^3 = 216$	$7^3 = 343$	$8^3 = 512$	$9^3 = 729$
number of solutions (with constraints, no early pruning)	120	210	336	504
solutions received at	9741ms	9914ms	10643ms	10753ms
agents started at	11838ms	11886ms	12510ms	12512ms

Table 7.21: Increase of solution set size for additional DSPs. JamVM 1.3.0 on PC1, SmartCam 1 and SmartCam 2 without early pruning.

Test Case 5 – Increasing the QoS of Agents

For this test case, the MPEG encoder agents are equipped with the two quality of service levels 0 and 1. In addition to that, they are encouraged to run at level 1 by setting the desired QoS level accordingly. For an early pruning cost threshold of 8000, the number of feasible solutions reported is 1248. The cheapest solution that was selected, is shown in table 7.22. The MPEG agents are running at QoS level 1 (higher QoS level numbers denote lower quality). All placements containing MPEG agents not running at their desired QoS level are penalized with the QoS_{wrong} costs. These costs outweigh the quality of service degradation penalty QoS_{deg}. As a consequence, placements where the MPEG encoder agents run at reduced quality become cheaper than those placements with higher service quality.

Agent Name	Node 1	Node 2	Node 3
MPEG Agent 1	DSP 0, QoS 1		
MPEG Agent 2		DSP 0, QoS 1	
MPEG Agent 3			DSP 0, QoS 1
Fire Detection Agent	DSP 1, QoS 0		
SVD Agent			DSP 1, QoS 0

Table 7.22: The cheapest solution for test case 5.

Increasing the QoS level of an agent is done by re-checking the set of feasible solutions from the last full cluster configuration. In this case, this set contains 1248 complete solutions. If a solution does not satisfy the new QoS requirements of the agent, it will be disabled. This results in a reduction of the number of feasible solutions. From this reduced set a new cheapest solution is selected. Increasing the QoS level of one MPEG agent reduces the set of feasible solutions to 656 elements. When also increasing the QoS level of the other two MPEG agents, another 304 and 144 solutions are disabled. The number of remaining, enabled solutions is 144. The results are summarized in table 7.23 The long time required to increase the QoS level of the MPEG agents 1 and 3 in the case of JamVM can be explained as follows: The request to increase the QoS level of an agent can be sent to any node that belongs to the cluster that hosts the agent. In this case, all

requests are sent to node 2. Only MPEG agent 2 resides on node 2. For MPEG agents 1 and 3, remote communication is required to inform them of their QoS change after the re-checking of the solution set on node 2.

Agent Name	disabled solutions	remaining solutions	time required (JRE, PC1, PC2, PC3)	time required (JamVM, PC1, SC1, SC2)
MPEG Agent 1	656	592	811ms (11ms)	3221ms (506ms)
MPEG Agent 2	304	288	713ms (7ms)	2537ms (89ms)
MPEG Agent 3	144	144	881ms (4ms)	2996ms (116ms)

Table 7.23: Running times for increasing the QoS level of agents. The numbers in brackets denote the time required for updating the solution set (disabling placements). The values in front of them denote the total time required for the QoS change.

By only manipulating the set of feasible solutions, a QoS increase can be handled faster than with solving the distributed CSP. When the QoS level of an agent is decreased again, the inverse approach is taken: Previously disabled solutions become enabled again and a new cheapest solution is selected.

7.2.2 Overlapping Clusters Scenario

The second test scenario is based on three overlapping clusters and four nodes, as shown in figure 7.3. Every node is equipped with two DSPs.

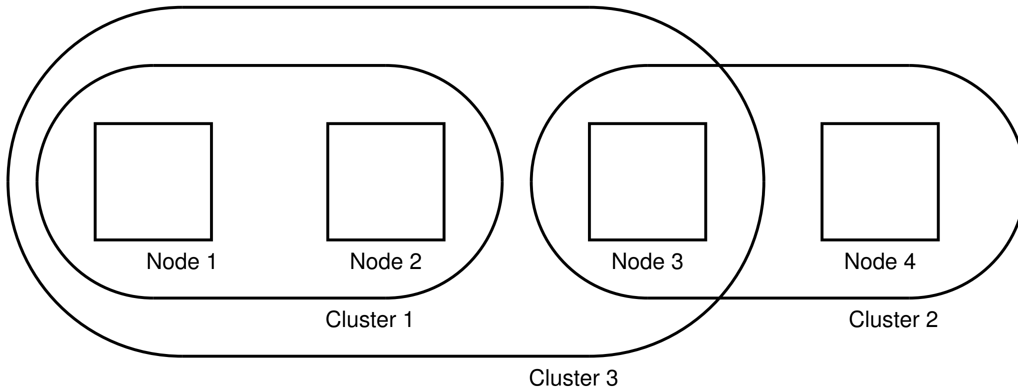


Figure 7.3: Three overlapping clusters on four nodes.

Cluster 1 should host two MPEG-4 encoder agents and one fire detection agent. Cluster 2 should contain the same types of agents with an additional stationary vehicle detection agent. For all MPEG-4 encoder agents the desired node is set upon creation in such a way that the placement of one MPEG-4 encoder is encouraged on every node. One wrong way driver detection agent, one vehicle count agent and one vehicle classify agent are assigned to cluster 3.

Test Case 1 – Effects of Overlapping Clusters

For this test case, the load distribution is done sequentially for clusters 1 to 3 in this order. Cluster 3 overlaps with cluster 1 and 2 which means that not the full amount of resources is available for cluster 3 on nodes 1 to 3. Early pruning is disabled for this test. Table 7.24 shows the cheapest final solutions reported by the load distribution system. Due to the limited amount of available resources, not every DSP application could be placed on an own DSP. The fire detection agents, consuming only little resources, share their DSPs with agents from cluster 3.

Cluster	Agent	Node 1	Node 2	Node 3	Node 4
Cluster 1	MPEG Agent 1	DSP 1			
	MPEG Agent 2		DSP 0		
	Fire Detection Agent		DSP 1		
Cluster 2	MPEG Agent 1			DSP 0	
	MPEG Agent 2				DSP 0
	Fire Detection Agent			DSP 1	
	SVD Agent				DSP 1
Cluster 3	Vehicle Count			DSP 1	
	Vehicle Classify	DSP 0			
	Wrong Way Driver		DSP 1		

Table 7.24: The cheapest solution for test case 1.

Table 7.25 presents the number of complete solutions with and without constraints as well as the running time for every cluster until all the agents of the cluster were started. The number of complete solutions for cluster 3 is of special interest. Of the six DSPs of cluster 3, only 3 are available in practice: DSP 0 on node 1 which is completely free, and both second DSPs on nodes 2 and 3 where only the fire detection agents are executed. This reduces the theoretical limit for the number of complete solutions of cluster 3 to $3^3 = 27$. This number is reduced further when looking at the requirements of the agents of cluster 3 (see table 7.5): The vehicle classification agent cannot be placed concurrently with the fire detection agents because of its CPU requirements. It therefore has to be placed on DSP 0 of node 1. For the other two agents, there are two remaining possible placements resulting in a total number of two complete solutions for cluster 3.

Cluster	number of solutions (no constraints)	number of solutions (with constraints)	solutions received at	agents started at
1	$4^3 = 64$	48	5349ms	6510ms
2	$4^4 = 256$	96	4966ms	5829ms
3	$6^3 = 216$	2	4552ms	6481ms

Table 7.25: Numbers of complete solutions and running times for test case 1. JamVM 1.3.0 on PC1, SmartCam 1, PC2 and SmartCam 2 without early pruning.

To examine the effects of multiple QoS levels, the vehicle counting agent is equipped with an additional, lower QoS level (see table 7.5). Table 7.26 presents the updated

solution set sizes and running times. For cluster 3, the size of the set of complete solutions has increased from two to ten. As in the previous case, the vehicle classification agent can only be assigned to DSP 0 of node 1. With two QoS levels, there are now four possible combinations to place the vehicle counting agent and the wrong way driver detection agent on the second DSPs of node 2 and 3. In addition to that, the vehicle counting agent at reduced QoS, can now also run concurrently on the same DSP with the three MPEG agents of nodes 1 to 3. Together with the two possible placements of the wrong way driver detection agent, this results in six new possible placements. Therefore, the total number of complete solutions increases to ten. The cheapest solution selected by the load distribution system, remains the same as presented in 7.24, because the additional placements where the vehicle counting agent is running at reduced quality result in higher costs.

Cluster	number of solutions (no constraints)	number of solutions (with constraints)	solutions received at	agents started at
1	$4^3 = 64$	48	5181ms	7050ms
2	$4^4 = 256$	96	5018ms	7460ms
3	$6^3 = 216$	10	4107ms	5299ms

Table 7.26: Numbers of complete solutions and running times for test case 1. The vehicle counting agent is now equipped with two QoS levels. JamVM 1.3.0 on PC1, SmartCam 1, PC2 and SmartCam 2 without early pruning.

Test Case 2 – Inter-Cluster Communication

It cannot be guaranteed that in case of overlapping clusters the requirements of all involved clusters can be met. In this case, two options exist. The first one is to select and implement an incomplete solution, where one or more agents are not assigned to a node. The second option, as described in section 5.2.5, is to ask clusters, also present on the same node, if they are able to reduce the load of the node.

Agent	Affinity	
	Cluster	Scene
MPEG-4 Encoder	0	10
Fire Detection	10	0
Stationary Vehicle Detection	8	0
Wrong-Way Driver Detection	6	0
Vehicle Classification	5	0
Vehicle Counting	4	0

Table 7.27: Affinity values of agents.

For this test case, an additional stationary vehicle detection agent is added to cluster 3. The four MPEG encoder agents are equipped with three QoS levels, namely levels 0, 1 and 3. All other agents only have their best QoS level. Early pruning is not enabled. The placement reported by the load distribution system (table 7.28) is almost identical to the placement of the last scenario. For cluster 3, the vehicle counting agent could not

be assigned to a node due to insufficient resources. For cluster 3, an incomplete solution, only containing three out of four agents was selected. The selection which agent of cluster 3 is not executed, is done via the cluster affinity costs of the agents. As demonstrated in table 7.27, the vehicle counting agent has the lowest cluster affinity of the agents of cluster 3 . Since a low cluster affinity results in high cluster affinity costs, the vehicle counting agent is not part of the cheapest solution containing three agents.

Cluster	Agent	Node 1	Node 2	Node 3	Node 4
1	MPEG Agent 1	DSP 1, QoS 0			
	MPEG Agent 2		DSP 0, QoS 0		
	Fire Detection		DSP 1, QoS 0		
2	MPEG Agent 1			DSP 0, QoS 0	
	MPEG Agent 2				DSP 0, QoS 0
	Fire Detection			DSP 1, QoS 0	
	SVD Agent				DSP 1, QoS 0
3	SVD Agent		DSP 1, QoS 0		
	Vehicle Classify	DSP 0, QoS 0			
	Wrong Way Det.			DSP 1, QoS 0	
	Vehicle Counter				

Table 7.28: The cheapest solution for test case 2. The vehicle counting agent could not be placed due to insufficient resources on the nodes of cluster 3.

As shown in table 7.29, the sizes of the solution sets for clusters 1 and 2 have increased. This results from the two additional QoS levels for every MPEG agent. The solution set size of 12 for cluster 3 denotes the number of solutions where three agents could be placed.

Cluster	solution set size	solutions received at	agent start times
Cluster 1	520	7350ms	10056ms
Cluster 2	1344	9952ms	11883ms
Cluster 3	12	4420ms	6551ms

Table 7.29: Numbers of solutions and running times for test case 2. JamVM on PC1, SmartCam 1, PC2 and SmartCam 2 without early pruning. The 12 solutions for cluster 3 do not denote complete solutions but solutions where three agents could be placed.

To allow cluster 3 to allocate all of its agents to nodes, communication with the other clusters present on its nodes is required. When computing the local CSPs, the other clusters on the node are asked if and how much they can reduce the load of the node (see section 5.2.5). With inter-cluster communication enabled, the load distribution system now reports the solution shown in table 7.30. Changes compared to the last solution are highlighted. As a result of the inter-cluster communication, the QoS level of MPEG agent 2 of cluster 1 was reduced. The resources that become available, allow cluster 3 to allocate all four of its agents, including the vehicle counting agent. Table 7.31 presents the sizes of the solution sets and the running times for the load distribution process.

Cluster	Agent	Node 1	Node 2	Node 3	Node 4
1	MPEG Agent 1	DSP 1, QoS 0			
	MPEG Agent 2		DSP 0, **QoS 3**		
	Fire Detection		DSP 1, QoS 0		
2	MPEG Agent 1			DSP 0, QoS 0	
	MPEG Agent 2				DSP 0, QoS 0
	Fire Detection			DSP 1, QoS 0	
	SVD Agent				DSP 1, QoS 0
3	SVD Agent		DSP 1, QoS 0		
	Vehicle Classify	DSP 0, QoS 0			
	Wrong Way Det.			DSP 1, QoS 0	
	Vehicle Counter		**DSP 0, QoS 0**		

Table 7.30: The cheapest solution for test case 2. The QoS level of MPEG agent 2 of cluster 1 was reduced to free resources. As a consequence, the vehicle counting agent of cluster 3 can now be placed.

Cluster	solution set size	solutions received at	agent start times
Cluster 1	520	7613ms	9880ms
Cluster 2	1344	10471ms	12793ms
Cluster 3	24	4836ms	7488ms

Table 7.31: Numbers of complete solutions and running times for test case 2. JamVM on PC1, SmartCam 1, PC2 and SmartCam 2 without early pruning.

7.2.3 Cost Factors

Table 7.32 presents the cost factors used in the evaluation.

Cost type	Factors
Resource Costs	$k_{R_i} = 1$ $k_R = 200$
Data Transfer Costs	$k_T = 1$
Migration Costs	$k_{MT} = 2$ $k_{MI} = 1$ $k_M = 1$

Cost type	Factors
Affinity Costs	$k_{AS} = 10$ $k_{AC} = 100$ $k_A = 4$ $k_p = 100$
QoS Costs	$QoS_{DegradationFactor} = 2$ $QoS_{NotDesiredLevelPenalty} = 10$

Table 7.32: Cost factors used for the evaluation.

Chapter 8

Conclusion and Future Work

The result of this work is a prototype implementation of the proposed load distribution system. The evaluation of the system has proven the general feasibility of the approach. The work has shown that it is practical to solve the constraint satisfaction problem distributedly and merge the partial solutions.

To find an optimal solution the assignment of costs to each solution is proposed. This mechanism allows to select a solution that avoids migrations, runs agents at high quality and takes into account the affinity of agents to clusters and scenes. The evaluation has shown that the results of the solution selection conform with the expected behavior. Care has to be taken when modifying cost factors. The effects this can have on the overall results, especially in combination with early pruning, can be difficult to anticipate.

In terms of performance, the selected development environment turned out to be not a perfect choice for embedded systems. While Java allows for fast development based on its extensive class library, it also introduces a significant performance overhead compared to lower level programming languages such as C or C++. An additional drawback regarding system performance is, that no just-in-time compiler but only an interpreter is available for the XScale platform. Since most mobile agent systems are written in Java, this programming language was the logical choice for the project. In terms of performance, it has proven to be not optimal.

8.1 Future Work

Not only performance improvements are to be examined in the future. Other topics such as efficient inter-cluster communication and optimizations of the cost system and the early pruning mechanism can be explored.

- *Native Implementation.* The mobile agent paradigm is well suited for the implementation of the load distribution system. At the same time, the overhead introduced by the agent infrastructure and the JamVM runtime environment cannot be neglected: in terms of performance, native implementations are by far better suited for embedded environments. Hence, a possible way to improve the overall performance

of the system is to evaluate which features of mobile agents are required, and to re-implement the load distribution system in a programming language such as C or C++.

- *Inter-Cluster Communication.* Inter-cluster communication is an important topic for realistic scenarios with many overlapping clusters. In general, clusters are not aware of each other. As a consequence, the allocation of agents to nodes might fail because agents of foreign clusters are already consuming resources on the nodes. A first step towards inter-cluster communication was taken with the approach presented in section 5.2.5. Decreasing the QoS level of foreign agents is a simple measure to deal with the problem. More advanced approaches such as reconfiguring foreign clusters to reduce the load of certain nodes have to be explored.

- *Optimizing the Cost System.* The cost system offers a wide range of possibilities for future optimizations. Aside from selecting the optimal solution, the costs in combination with early pruning can have a significant impact on the overall performance of the system. If it is possible to reject expensive solutions reliably, early in the load distribution process, the amount of data to be processed and transferred over the network can be reduced. At the same time, it has to be ensured that the final set of complete solutions will not be empty by rejecting too many intermediate solutions. A possible approach is an adaptive early pruning mechanism, that does not operate with fixed thresholds but also takes the results of previous runs of the load distribution system into account.

- *Distributed Merge.* A potential improvement is not only to do the computation of the CSP in parallel on the nodes of a cluster, but also the merging of two solutions. To do so, one of the two solution trees to be merged, has to be split. This can be done by removing the empty set, which is the root of the tree. This results in a set of independent sub-trees with elements of level 1 as their new root elements. These sub-trees can then be merged with the second solution tree in parallel on different nodes. The resulting trees are then combined into one single solution by re-attaching them to the empty set. This approach is similar to the domain based decomposition of CSPs presented in section 2.4.3. It has to be evaluated, if the additional communication required, does not outweigh the potential increase in performance.

Bibliography

[AgD] Wordreference.com: Agent definition. http://www.wordreference.com/definition/agent. last visited: April 2005.

[Aic04] Aicas GmbH. JamaicaVM - User Documentation. http://www.aicas.com/jamaica/doc/html/index.html, 2004. last visited: April 2005.

[Bon04] Erwin Bonsma. *DIET Agents tutorial*, 1.26 edition, July 2004. available online at: http://diet-agents.sourceforge.net/.rsrc/tutorial.html. last visited: April 2005.

[Bra05] Michael Bramberger. *Distributed Dynamic Task Allocation in Clusters of Embedded Smart Cameras*. PhD thesis, Institute for Technical Informatics, Graz University of Technology, May 2005.

[BRS04] Michael Bramberger, Bernhard Rinner, and Helmut Schwabach. An Embedded Smart Camera on a Scalable Heterogeneous Multi-DSP System. In *Proceedings of the European DSP Education and Research Symposium (EDERS2004). Birmingham, United Kingdom*, 2004.

[Bub96] Kristian P. Bubendorfer. Resource Based Policies for Load Distribution. Master's thesis, Victoria University of Wellington, August 1996.

[Cou02] CougaarME Development Team. CougaarME Architecture Guide. http://cougaar.org/docman/view.php/9/4/CougaarMEArchV1.1.pdf, May 2002. last visited: April 2005.

[CWD02] Jiannong Cao, Xianbing Wang, and Sajal K. Das. A Framework of Using Cooperating Mobile Agents to Achieve Load Sharing in Distributed Web Server Groups. In *Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'02)*, pages 118–125, Beijing, China, October 2002. IEEE Computer Society.

[DH02] Sasa Desic and Darko Huljenic. Agents Based Load Balancing with Component Distribution Capability. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)*, pages 327–331, Berlin, Germany, May 2002. IEEE Computer Society.

[EET01] Uew Egly, Thomas Eiter, and Hans Tompits. *Skriptum zur Lehrveranstaltung Wissensbasierte Systeme*. Institut für Informationssysteme, Abteilung Wissensbasierte Systeme, Vienna University of Technology, 2001.

[FF99] Christian Frei and Boi Faltings. Resource Allocation in Networks Using Abstraction and Constraint Satisfaction Techniques. In Joxan Jaffar, editor, *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, volume 1713 of *Lecture Notes in Computer Science*, pages 204 – 218, Alexandria, VA, USA, October 1999. Springer-Verlag GmbH.

[GBH98] Michael Greenberg, Jennifer Byington, and David Harper. Mobile agents and security. *IEEE Communications Magazine*, 36(7):76–85, July 1998.

[Gig02] Eric Giguere. Object Serialization in CLDC-Based Profiles. http://developers.sun.com/techtopics/mobility/midp/ttips/serialization/, February 2002. last visited: April 2005.

[GNU] GNU Classpath Team. GNU Classpath Hacker's Guide. http://www.gnu.org/software/classpath/docs/hacking.html. last visited: April 2005.

[GT00] Volker Gruhn and Andreas Thiel. *Komponentenmodelle*. Addison-Wesley, 2000.

[Han97] Fred Hantelmann. Probierstuben (Java-Benchmark-Suites). *iX*, 5:154, 1997. available online at: http://www.heise.de/ix/artikel/1997/05/154/, last visited: May 2005.

[Hol04] Ralf Holzheu. Untersuchung der Einsetzbarkeit von Echtzeit-Java in der Radarsignalverarbeitung. Master's thesis, Studiengang Technische Informatik, Fachhochschule Ulm, October 2004.

[HWBM02] Cefn Hoile, Fang Wang, Erwin Bonsma, and Paul Marrow. Core specification and experiments in DIET: a decentralised ecosystem-inspired mobile agent system. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 623 – 630, Bologna, Italy, 2002. ACM Press.

[IKWK00] Torsten Illmann, Frank Kargl, Michael Weber, and Tilmann Krüger. Migration of Mobile Agents in Java: Problems, Classification and Solutions. In *Proceedings of the Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce (MAMA'2000) Conference*, University of Wollongong, Australia, December 2000.

[Jen99] Nicholas R. Jennings. Agent-Oriented Software Engineering. In Francisco J. Garijo and Magnus Boman, editors, *Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi-Agent System Engineering (MAAMAW-99)*, volume 1647 of *Lecture Notes in Computer Science*, pages 1–7. Springer-Verlag: Heidelberg, Germany, 1999.

[Kaf] Kaffe Development Team. Kaffe documentation. http://www.kaffe.org/documentation.shtml. last visited: April 2005.

[Kum92] Vipin Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine*, 13(1):32–44, 1992. American Association for Artificial Intelligence.

[LEA05] LEAP Consortium. Lightweight Extensible Agent Platform (LEAP) User Guide. http://jade.tilab.com/doc/LEAPUserGuide.pdf, March 2005. last visited: April 2005.

[LHB93] Qiang Y. Luo, Peter G. Hendry, and John T. Buchanan. Comparison of different approaches for solving distributed constraint satisfaction problems. Technical report, Department of Computer Science, University of Strathclyde, Glasgow G1 1XH, UK, 1993.

[LKP99] Antonio Liotta, Graham Knight, and George Pavlou. On the Performance and Scalability of Decentralized Monitoring Using Mobile Agents. In *Proceedings of the 10th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management: Active Technologies for Network and Service Management*, volume 1700 of *Lecture Notes In Computer Science*, pages 3–18, October 1999.

[LO99] Danny B. Lange and Mitsuru Oshima. Seven Good Reasons for Mobile Agents. *Communications of the ACM*, 42(3):88–89, March 1999.

[Lou] Robert Lougher. JamVM website. http://jamvm.sourceforge.net/. last visited: April 2005.

[MJT+01] Pragnesh J. Modi, Hyuckchul Jung, Milind Tambe, Wei-Min Shen, and Shriniwas Kulkarni. A Dynamic Distributed Constraint Satisfaction Approach to Resource Allocation. In T. Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP 2001)*, volume 2239 of *Lecture Notes in Computer Science*, pages 685–700, Paphos, Cyprus, November 2001.

[MKvL+01] Paul Marrow, Manolis Koubarakis, Rolf H. van Lengen, Francisco Valverde-Albacete, Erwin Bonsma, Jesús Cid-Sueiro, Aníbal R. Figueiras-Vidal, Ascensión Gallardo-Antolín, Cefn Hoile, Thodoros Koutris, Harold Y. Molina-Bulla, Angel Navia-Vázquez, Paraskevi Raftopoulou, Nikolaos Skarmeas, Christos Tryfonopoulos, Fang Wang, and Chryssani Xiruhaki. Agents in Decentralised Information Ecosystems: the DIET Approach. In *Proceedings of the Artificial Intelligence and Simulation Behaviour Convention 2001 (AISB'01), Symposium on Information Agents for Electronic Commerce*, pages 109–117, York, UK, March 2001.

[Mon] Mono Development Team. Mono - UNIX version of the Microsoft .NET platform. http://www.mono-project.com/. last visited: April 2005.

[MR01] Ashok Mathew and Mark Roulo. Accelerate your RMI programming - Speed up performance bottlenecks created by RMI. http://www.javaworld.com/

javaworld/jw-09-2001/jw-0907-rmi_p.html, September 2001. last visited: May 2005.

[Ort04] Enrique Ortiz. A Survey of J2ME Today. http://developers.sun.com/techtopics/mobility/getstart/articles/survey/, October 2004. last visited: April 2005.

[OTM03] Jae C. Oh, Madhura S. Tamhankar, and Daniel Mossé. Design of Very Lightweight Agents for reactive embedded systems. In *Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'03)*, pages 149–158, Huntsville, Alabama, April 2003. IEEE Computer Society.

[PCVM99] Menelaos K. Perdikeas, Fotis G. Chatzipapadopoulos, Iakovos S. Venieris, and Gennaro Marino. Mobile agent standards and available platforms. *Computer Networks*, 31:1999–2016, 1999.

[PPW02] Josef Pichler, Reinhold Plösch, and Rainer Weinreich. MASIF und FIPA: Standards für Agenten. Übersicht und Anwendung. *Informatik Spektrum*, 25(2):91–100, April 2002.

[Qua05] Markus Quaritsch. An agent-based Framework for Object Tracking among multiple Smart Cameras. Master's thesis, Institute for Technical Informatics, Graz University of Technology, 2005.

[RB02] Emmanuel Reuter and Francoise Baude. A mobile-agent and SNMP based management platform built with the Java ProActive library. In *Proceedings of the IEEE Workshop on IP Operations and Management*, pages 140 – 145, 2002.

[RBB+04] Bernhard Rinner, Horst Bischof, Michael Bramberger, Andreas Doblander, Arnold Maier, Roman Pflugfelder, and Helmut Schwabach. Eine intelligente Kamera zur Verkehrsüberwachung. *Bulletin SEV/VSE 11/04*, 95:19–F, 2004.

[Rec] Recursion Software. *Voyager ORB Developer's Guide*. available online at: http://www.recursionsw.com/Voyager/Voyager_User_Guide.pdf. last visited: April 2005.

[RH00] Ashok Rajagopalan and Salim Hariri. An Agent Based Dynamic Load Balancing System. In Xue Zhang, Yongdong Tan, and Baohua Zhang, editors, *Proceedings of the International Workshop on Autonomous Decentralized System*, pages 164–171, Chengdu, China, September 2000. IEEE Computer Society.

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.

[RRF01] Carlo S. Regazzoni, Visvanathan Ramesh, and Gian L. Foresti. Special issue on video communications, processing, and understanding for third generation surveillance systems. In *Proceedings of the IEEE*, volume 89, pages 1355 – 1539, October 2001.

[Rym03] Arthur Ryman. Understanding Web Services. http://www-128.ibm.com/developerworks/websphere/library/techarticles/0307_ryman/ryman.html, July 2003. last visited: April 2005.

[Sab] SableVM Development Team. SableVM documentation. http://sablevm.org/docs.html. last visited: April 2005.

[SGB01] Niranjan Suri, Paul T. Groth, and Jeffrey M. Bradshaw. While You're Away: A System for Load-Balancing and Resource Sharing Based on Mobile Agents. In Rajkumar Buyya, George Mohay, and Paul Roe, editors, *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'01)*, pages 470–475. IEEE Comuter Society, May 2001.

[Sho93] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[SKS92] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load Distributing for Locally Distributed Systems. *IEEE Computer*, 25(12):33–44, December 1992.

[SSCJ98] Onn Shehory, Katia Sycara, Prasad Chalasani, and Somesh Jha. Agent cloning: an approach to agent mobility and resource allocation. *IEEE Communications Magazine*, 36:58–67, 1998.

[Sun01] Sun Microsystems. Connected Device Configuration (CDC) and the Foundation Profile. http://java.sun.com/products/cdc/wp/CDCwp.pdf, 2001. last visited: April 2005.

[Tve00] Amund Tveit. A survey of Agent-Oriented Software Engineering. Technical report, Norwegian University of Science and Technology, May 2000.

[Vig04] Giovanni Vigna. Mobile Agents: Ten Reasons For Failure. In *Proceedings of the 2004 IEEE International Conference on Mobile Data Management (MDM'04)*, pages 298 – 299, 2004.

[Vrb04] Pavel Vrba. JAVA-Based Agent Platform Evaluation. In Vladimir Marík, Duncan McFarlane, and Paul Valckenaers, editors, *Lecture Notes in Computer Science*, volume 2744, pages 47–58. Springer-Verlag GmbH, 2004.

[Win04] Thomas Winkler. XScale Linux Distribution - PCI Board Linux Integration. Technical report, Austrian Research Centers - Seibersdorf Research, 2004.

[WJ94] Michael Wooldridge and Nicholas R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10:115–152, 1994.

[Wöc04] Thomas Wöckinger. Interfacing Linux and TI C64 DSPs. Technical report, Institute for Technical Informatics, Graz University of Technology, 2004.