Eindhoven University of Technology

MASTER

Themis : load sharing in an in-home network

Heijmans, J.

Award date:
2002

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER'S THESIS

Themis
Load sharing in an in-home network

by
J. Heijmans

Supervisor: prof. dr. ir. J.F. Groote
dr. ir. M.A. Reniers
ir. H. van Woerkom

Advisors: dr. C. Huizing

Eindhoven, October 2002

Abstract

In-home networks connect household devices with each other, enabling cooperation between devices and allowing for new functionality. The possibilities and usefulness of in-home networks are still being explored; research focuses on enabling technologies as well as development environments and applications.

In this document, we take a look at the possibility of adding a load sharing system to an in-home network. A load sharing system would enable devices to have parts of their work executed by other devices in the network. Such a system could not only improve performance, and may also enable new kinds of functionality.

Using properties and approaches of known load sharing systems, we discuss the approach of a load sharing system for an in-home network. With these properties, we develop an architecture for such a system to be added to an in-home network prototype implemented using Java and Jini. One of the key parts of this architecture is the component that determines which of the devices are capable of executing work from other devices, and which of these devices will execute the work.

This document concludes with an evaluation of the developed prototype and a discussion of the practical application of a load balancing system in in-home networks.

Contents

Preface

This document is my master's thesis of the study *Technische Informatica* at the Technische Universiteit Eindhoven. All research and work for this thesis was conducted at Philips Technical Application Software Services (TASS) in Eindhoven from December 2001 to October 2002. Philips TASS is an autonomous business unit of the Philips concern that develops software for products as well as for production support.

During the project, I worked together with Melissa Tjiong, a student of *Telematica* at Universiteit Twente in Enschede. Most of the initial research and design was done together, but thereafter we divided most of the work such that we both had our own part in the final result. In this report, I will refer to Melissa's master's thesis for those parts of the project that were primarily conducted by her.

Acknowledgements

First of all, I would like to thank Melissa Tjiong, with whom I worked throughout the project. Without her, I would not have been able to deliver the same result.

Of the people at Philips TASS, I first wish to thank Harald van Woerkom. Every week, he helped solving our problems with the project, whether they were theoretical, technical or organisational, and he always answered the many questions we fired at him by e-mail. As the manager of our project, Tiny Henst helped us with organisational problems we encountered and taught us to look at things from a non-technical perspective as well. Dragos Manolache, one of the developers of the TASS In-Home Network Simulation Framework II, helped us with technical details of Jini and the framework, and critically read our initial technical documentation.

I thank my university supervisors Jan Friso Groote and Michel Reniers, who were initially sceptic of the originally vague problem description, but greatly supported me with their critique and useful advice, especially during the final period, when I wrote this report.

Finally, I thank everybody who in any way supported me during my graduation work, especially my family, friends and house-mates.

Jeroen Heijmans, October 2002

Chapter 1

Introduction

Before presenting the problem description, we briefly introduce the concepts of *in-home networks* and *load balancing*. Understanding these two terms is required to comprehend the problem description.

1.1 In-home networks

An in-home network is a network of household devices, ranging from televisions and audio systems to personal computers and even microwave ovens.

This network enables communication between the devices, which allows for new functionality. Cooperation between the devices becomes much easier, for example using your audio system for the sound of your television, or muting all sound when you pick up the telephone. As all devices are interconnected, it is possible to provide a unified interface; all devices can be controlled using a single remote control.

Completely new services are also possible, especially as household devices become digital. An example of this is the so-called "follow-me behaviour". This allows a user to copy the state of one device to another. For example, a user may set the television in the bedroom to continue playing the CD the user was listening to in the living room.

Currently, in-home network implementations and applications are still in an early development phase. Several technologies facilitating the creation of in-home networks exist, but no standard has yet emerged. Actual implementation of in-home networks will require an increase in the processing capabilities of most current household devices, as most of them are not yet capable of performing operations other than their primary functions.

1.2 Load balancing

In distributed computer systems, load balancing denotes the problem of spreading tasks over the network in such a way that each of the devices in the network has an approximately equal load. Definitions of load differ per application and environment; examples are the number of tasks executed by a device, or the number of instructions performed per time unit.

As load balancing solutions are very much dependent on the nature of the tasks, the processors and the definition of processor load, there is no single solution for the load balancing problem.

A major division in load balancing solutions can be made between *static* and *dynamic* load balancing. Static load balancing can be performed if all properties of the tasks and the processors are known beforehand. As such a situation rarely

occurs, dynamic load balancing, where tasks are allocated to servers taking current and past information into account, is the most used form of load balancing in practice.

An example of (dynamic) load balancing can be found with large websites. Requests for pages are scheduled to one of the servers that can handle them such that all servers are handling approximately the same amount of requests.

1.3 Problem description

1.3.1 Overview

At Philips TASS, a prototype in-home network has been developed, running on one or more PCs rather than actual household devices. It appears to be rather CPU-intensive and it experiences slowdowns when using some of the applications. This observation has lead to the suggestion from TASS that a load balancing system may be useful for an in-home network.

The main objective of the project was to investigate whether addition of a load balancing system to TASS' in-home network is technically feasible. Pending the results of this investigation, an implementation of a load balancing system should be added to the current in-home network. Furthermore, a study of the benefits and drawbacks of this system should be conducted.

The project also contained a part in which the aforementioned performance problems were investigated, which served mostly as an introduction to TASS' in-home networks and related technologies. Although this investigation showed the performance problems were caused by the used software and therefore load balancing would not solve these particular problems, it was decided to continue the project on the topic of load balancing, as it was not only meant to solve the particular performance problems [9].

The name *Themis*[1] was given to the prototype that was developed during the project. In the remainder of the document, "Themis project" will be used to refer to the entire project.

1.3.2 Constraints

The concepts of in-home networks and load balancing both have no clearly defined boundaries. For the Themis project, we impose the following constraints:

Use TASS in-home network Although we attempt to discuss properties of in-home networks in general, we restrict ourselves to the Tass in-home network if necessary. We attempt to re-use available functionality of the current network if possible, and may give priority to solutions not involving new components.

No external access We assume that no tasks or servers outside the "home" will participate, and that security measures against external access have been taken.

Singular tasks The execution of each task can be performed by one single device.

No execution guarantee Although the system should strive to be reliable and have all tasks executed, it is impossible to guarantee that a task moved to a processor is actually executed.

---

[1]In Greek mythology, Themis was the goddess of justice, wisdom and council. She is usually depicted with scales in her hand.

1.4 Document overview

The specifics of the in-home network at Philips TASS, and the technologies used for its implementation are described in Chapter 2. In Chapter 3, we show that pure load balancing cannot be applied to in-home networks, but that it can be replaced by a similar technique called *load sharing*. A discussion follows of how load sharing could be applied to an in-home network, and what the approach in Themis will be. The next chapter gives more details on the architecture of Themis. Chapters 5 and 6 take a closer look at two sub-problems of load sharing. An evaluation of the Themis prototype, and a discussion of load sharing in in-home networks are presented in Chapter 7.

Appendix A contains a list of object-oriented terms and definitions as used throughout this document.

Chapter 2

The TASS In-Home Network

Within Philips TASS, a number of projects concerning in-home networking have been carried out. The developed in-home network prototypes at TASS are implemented in the Java programming language, while making use of Jini and the related technologies of RMI and JavaSpaces for the network infrastructure.

In Section 2.1, we give a short overview of the in-home network projects that have been conducted at Philips TASS, and focus on the In-Home Network Simulation Framework II (Section 2.2), which will be used as the starting point of the Themis project. The remaining sections of this chapter give a brief overview of the most important technologies used: Jini (Section 2.3), Remote Method Invocation (2.4) and JavaSpaces (2.5). These three technologies are discussed in more depth in [6].

2.1 Overview

Currently, an in-home network framework is available at Philips TASS. This framework has been developed in a number of projects.

The first project, the In-Home Network Simulation Framework [3], resulted in a generic framework for an in-home network, also containing several simulated devices. These devices can operate and interact, running on one or several PCs. The framework features libraries to enable easier implementation of devices and services.

The In-Home Network Simulation Framework II expanded this framework to incorporate time-based media[1] into the simulations, using the Java Media Framework (JMF). Additional projects have dealt with attaching real devices (a television tuner and a remote control), interactive games and a home heating system.

2.2 In-Home Network Simulation Framework II

The In-Home Network Simulation Framework II [12] forms the starting point of the Themis project. The In-Home Network Simulation Framework II is itself an extension of the In-Home Network Simulation Framework, which offers a framework for devices participating in the in-home network. Common operations, such as discovery of look-up servers and the looking up of services (see Section 2.3) are implemented by the classes of the framework, which can be used by the devices.

The In-Home Network Simulation Framework II provides additional functionality in the form of support for state management and audio/video stream management. In addition, it provides several devices demonstrating these functions.

---

[1]Time-based media are any media that change meaningfully in time, such as video or audio.
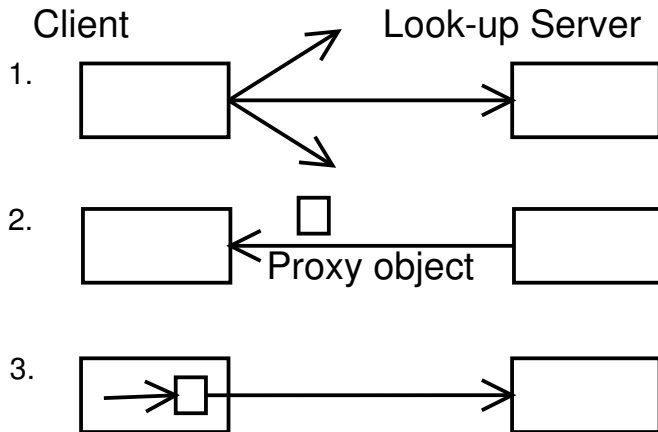
Figure 2.1: Three steps of discovery of a look-up server in a Jini network.

Note that while Themis uses the In-Home Network Simulation Framework II software, the additional functionality with respect to the In-Home Network Simulation Framework is not used.

2.3 Jini

Jini [2] is an infrastructure that enables the use of *services* in a network. Almost anything can comprise a service, ranging from small tasks such as performing a simple computation to bigger jobs such as printing a document. With Jini, all programs in the network can use services without having prior knowledge of their original location or implementation. Jini is written in Java, and intended primarily for use by programs written in that language.

The aim of Jini is to make interaction between networked programs easier. Jini automatically discovers all programs that wish to participate in a Jini network. Therefore, programs do not have to search for other programs, but can concentrate on looking for services instead. To use a service, a program need only know the interface to that service; it does not matter whether the service is implemented in hardware or software or if it is available locally or remotely.

2.3.1 Architecture

A Jini system is built around one or more *look-up servers*, which offer the *look-up service*. The look-up service can be used by a program to register services and to find (or look up) services it wants to use.

Before the look-up service can be used, programs will first need to find it. This is done through the *discovery* process, which is depicted in Figure 2.1. A program that wants to find the available look-up servers in the network will send out a multi-cast message to the network (1). If a look-up server receives such a message, it contacts the sender by transmitting a *proxy object* (2). This proxy object, which implements a known *interface*, can then be used to submit a new service; the proxy object, using RMI (see Section 2.4), contacts the remotely located look-up server (3).

Services can be added in a similar way. Services are registered by sending a proxy object to the look-up server. This proxy object implements one or more interfaces through which the users of the service can access it. A client program can *look-up* a service by submitting the interface(s) the service should implement

to a look-up server. If a matching service has been registered at the look-up server, it will send a copy of the proxy object for that service to the client program.

A proxy object can be used by the client program, since it knows the interface implemented by the proxy. The proxy object may then perform the actions of the service locally, or connect with a remote program, like the proxy for the look-up server. The way in which a proxy connects with a remote proxy is unspecified, but RMI is most commonly used.

2.3.2   Other concepts

Jini makes use of three other important concepts that are also available to the programs using Jini: leasing, remote events and transactions.

The concept of *leasing* is based on the real-world concept: clients are allowed to make use of an object for a pre-specified amount of time. A lease can be renewed before it expires, or it terminates and the client is no longer allowed to use the object. Jini uses this principle in a number of situations, for example with the registration of services. A program is granted a lease when it registers a service with the look-up server. If this lease expires, the look-up server will discard the service registration. This enables automatic discarding of service registration from devices that have crashed or disconnected from the network, which makes the Jini network more robust, as services that are no longer available cannot be requested.

Like Java, Jini offers the possibility of using *events* for asynchronous communication. An object can register interest in certain events with an event generator. If such an event occurs, the event generator will call a specified event listener object — not necessarily the same object as the one that registered — and pass information about the event that occurred. In Java, events are, among others, used for I/O operations; mouse clicks or keystrokes generate events. Jini's remote events are used in the same way. An example of its use can be found in the look-up service: a program can subscribe with the look-up server to receive an event when a service implementing a specified interface registers with that look-up server.

Jini also enables the use of *transactions* to deal with partial failure of network operations. Programs can use the services of a transaction manager, which coordinates transactions using the *two-phase commit* protocol: each participant in the transaction provisionally performs its operations. The transaction manager then issues a "pre-commit" message, to ask if all participants have completed their part. If all participants acknowledge, the transaction manager issues the "commit" command, and each participant will make the changes final. If one of the participants does not acknowledge (within a given period of time), all participants will receive an "abort" signal. This ensures that an operation is always performed completely, or not at all, assuming reliable communication. The service of the transaction manager is included in Jini by default.

2.4   Remote Method Invocation

Remote Method Invocation (RMI) is a Java technology that allows objects to perform method calls on objects in a different Java Virtual Machine[2] (JVM). The calling object only needs to know the interface of the remote object; its implementation need not be available locally.

Jini uses RMI for communication between look-up servers and applications. Most custom-built Jini services that connect to a remote process also apply RMI.

---

[2]The Java Virtual Machine is an interpreter for Java byte code. This allows for the same Java code to be executed on multiple platforms.
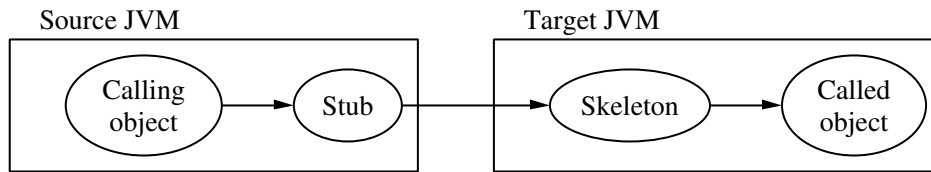
Figure 2.2: The steps of a method call to a remote object using RMI.

2.4.1 Stubs and skeletons

Because ordinary Java objects can only call on local objects (that is, objects created in the same JVM) and only be called by local objects, RMI provides two intermediary steps to allow for object calls between JVMs (see also Figure 2.2). The server uses a *stub* object to call on, which is created at compile-time, and made available to the user at run-time. Using the HTTP protocol, the stub transmits the method calls to an object in the target JVM, the *skeleton*, which is created during run-time and subsequently calls the target object, running locally.

All parameter and return objects can be transported using Java *serialisation*, an operation that transforms Java objects into byte streams and vice versa. In case the class of a parameter or return value is not known in a JVM, the class implementation can be downloaded using the HTTP protocol. The place where the implementation of an object is located is called *codebase*, and this information is attached to an object when it is transmitted using RMI, allowing for the dynamic loading of code.

2.4.2 Activation

Remotely accessible objects usually are "alive", that is, permanently waiting to be called upon. In case there are many such objects running on a single machine, resource comsumption may be unnecessarily big. The *RMI Activation Framework* makes it possible to de-activate such objects, and activate them only when they are called upon.

The activation framework makes use of an *activation daemon*, a program to which *activatable* objects can be registered. Whenever one of the de-activated objects is called, the activation daemon activates that object in a new JVM[3]. Note that the activation daemon itself is permanently alive and therefore only beneficial if multiple activatable objects are present.

An additional benefit of the Activation Framework is that objects can still be activated after the machine on which they were running has crashed and recovered.

The standard Jini services, such as the Sun implementations of the look-up server and transaction manager are implemented as activatable objects.

2.5 JavaSpaces

JavaSpaces is an implementation of a *tuple space*, offered as a Jini service [1].

A tuple space is a collection of *tuples*, terms consisting of a key and zero or more arguments. Tuples can be read from the space by specifying the key and the arguments (possibly as wild-cards) with which the result should match. If there are multiple matches, the result is non-deterministically chosen from the matches.

---

[3]For each activatable object, it is possible to define an *activation group*. Objects having the same activation group will be activated in the same JVM.

In JavaSpaces, the tuples are objects, and their keys are the interfaces implemented by the objects. Because an object can implement multiple interfaces, a JavaSpaces tuple can have several keys if the object implements multiple interfaces. The public variables of the object correspond with the arguments of the tuple space. The JavaSpaces service essentially offers three operations: writing an object, reading an object and taking an object (reading and then removing an object).

The JavaSpaces service provides a type of shared memory to a network. Applications can exchange objects with each other without knowing the other application exists — in fact even without having to exist at the same point in time — and objects can be found by associative look-up (type of object, properties of the object). Since the objects are Java objects, they not only contain data, but can also provide executable content.

The method of storage used by JavaSpaces is not defined by the specification. Sun's default implementation stores objects in a file for persistent storage, and in memory for non-persisent storage.

The In-Home Network Simulation Framework II uses JavaSpaces for general services and state management.

Chapter 3

Approach

Because of the properties of in-home networks, it appears to be impossible to implement strict load balancing in an in-home network, as we will show in Section 3.1. Instead, a "weaker" form of load distribution, load sharing, will be applied.

Using a classification of load distribution algorithms, we attempt to discover the properties that best fit with load sharing in in-home networks in Section 3.2. Using these properties, we can take a look at the behaviour of the system, which is done by discussing four *policies* that together make up a general load sharing algorithm (Section 3.3). We conclude this chapter with an overview of the properties of a load sharing system in an in-home network.

3.1   Load balancing in an in-home network

As mentioned in Section 1.2, the method of load balancing depends very much on the network, the processors in it and the tasks that are subject to load balancing. If we examine in-home networks, we find a number of properties that make implementing load balancing very difficult.

By nature, an in-home network is *heterogeneous*. Devices do not only have different capacities with respect to memory and processing power — which makes it difficult to define a good measure for device load — but many tasks can simply not be performed by all devices in the network: an audio system can play sound, but a microwave oven cannot[1].

While not all tasks can be executed at all nodes for technical reasons, practical reasons also prevent tasks to be performed just anywhere in the in-home network. The physical location of a device is an important property, and it is unacceptable to, for example, move a task playing audio to a device in another room.

Furthermore, an in-home network, especially when implemented using Jini, is likely to be prone to many changes in the network. Devices can be connected, disconnected and moved to a different location (mobile devices) at any time.

The above observations indicate that strict load balancing is not possible within an in-home network. The heterogeneity of the network, importance of location of execution and the dynamic nature of an in-home network make it impossible in most situations to achieve a balance in the network load. It must be concluded that it is not useful to try to apply a load balancing solution to an in-home network.

However, when looking at the reason for the load balancing requirement, we observe that its goal was not to achieve absolute equal load, but rather have the

---

[1]Note that the simulated devices in the TASS In-Home Network are *homogeneous* with respect to their operating environments, Java and Jini.
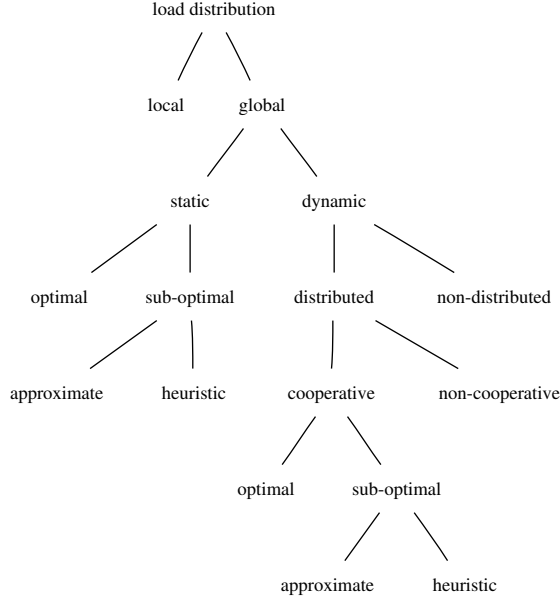
Figure 3.1: Hierarchical classification of load distribution algorithms from Casavant and Kuhl [4]

opportunity to spread the load more equally over the network. A system that can do that is commonly called a *load sharing* system.

Decision 1 (Load sharing) *Because the objective of a balanced load is probably impossible to achieve for in-home networks, we decide to investigate the application of a load sharing system instead.*

3.2   Classification of load sharing approaches

As mentioned in Section 1.2, there are many approaches to load balancing, and this is no different for load sharing. Both load balancing and load sharing can be seen as a load distribution algorithm. Casavant and Kuhl present a classification scheme of load distribution algorithms in [4]. We use this classification to investigate which properties of a load sharing solution are most suitable for application in an in-home network.

The first part of the classification has a hierarchic structure, as presented in Figure 3.1. We discuss the branches of this tree from top to bottom, following the choices made on the previous level in Sections 3.2.1 to 3.2.4. Casavant and Kuhl also discuss a number of properties outside of their hierarchic classification. Of those, we will discuss adaptiveness and re-assignment in Sections 3.2.5 and 3.2.6, respectively.

3.2.1   Local or global

The first distinction made is between local and global distribution. Local load distribution concerns the assignment of tasks to the time-slices of individual processors; global load distribution deals with the distribution of tasks among multiple processors. While local load distribution may be of use for the individual devices

in an in-home network, we are only interested in global distribution as we need to deal with scheduling of tasks between devices.

Decision 2 (Local or global) *Because an in-home network consists of multiple devices, we need to apply global load distribution.*

3.2.2 Static or dynamic

As earlier explained in Section 1.2 for load balancing, static and dynamic load distribution refer to the point in time where the distribution decisions are made. With static distribution, all relevant information is available at compile-time, and the decisions can be made at that time. Dynamic load distribution takes place at run-time, as new tasks — or processors — arrive.

Clearly, most tasks in the in-home network cannot be predicted beforehand — users may switch on devices at any time — and the dynamic form of load balancing is therefore the most suitable for in-home networks.

However, a number of tasks may be known some time before their execution, such as the task of a videocassette recorder to tape a certain television program. For Themis, such knowledge is not employed.

Decision 3 (Static or dynamic) *In Themis, we apply dynamic load sharing, as a static approach is impossible.*

3.2.3 Distributed or non-distributed

In a non-distributed approach, the job of distributing tasks lies with a single device, while in a distributed approach, this responsibility is shared. It is claimed that, in general, non-distributed approaches give a better performance than distributed ones [5].

A non-distributed approach requires one device to take care of all load sharing activities. Since no assumptions can be made on which devices are available in the network, this would have to be a new device[2]. This device will require information from the other devices in the network in order to be able to make load distribution decisions. Given the heterogeneity of the network, assembling and applying this information is not a trivial task; it is hard to decide whether a particular device could benefit from a task, or if a device could accept another task for execution. Moreover, a non-distributed approach is more scalable and less dependable: the size of the network has no influence on the working of the system, and failure of one or a few devices does not lead to loss of functionality.

Decision 4 (Distributed or non-distributed) *We think that a distributed load sharing approach is more suitable for in-home networks than a non-distributed one.*

3.2.4 Cooperative or non-cooperative

In the case of distributed load distribution, the system may work in a cooperative or non-cooperative way. In the non-cooperative case, each processor can take decisions completely autonomously, while in the cooperative approach, processors work together towards a common goal. This division is not black and white; processors may still have a large extent of autonomy while cooperating with other processors. For an in-home network, fully cooperative load distribution seems too strict, given the dynamic and heterogeneous properties of the network.

---

[2]The In-Home Network Simulation Framework already requires the presence of an additional device for running the default Jini services and these might be combined.

Decision 5 (Cooperative or non-cooperative) *The devices in the network are best capable to make load sharing decisions themselves, and should therefore receive the required degree of autonomy.*

3.2.5 Adaptive or non-adaptive

As a characteristic outside of the hierarchic classification, a load distribution approach can be adaptive or not. An adaptive solution can change its scheduling behaviour based on the past or current system behaviour, while a non-adaptive system cannot.

Decision 6 (Adaptive or non-adaptive) *An adaptive load sharing approach is probably most suitable for in-home networks, as the configuration of the network will differ between networks, and can change over time within a network.*

3.2.6 One-time assignment or dynamic re-assignment

For dynamic load distribution, another characteristic concerns the re-assignment of tasks. After a task has been assigned to a processor, it may be necessary or beneficial to move the task to yet another processor, rather than leaving it at the processor until it has finished (one-time assignment).

While dynamic re-assignment may pose a lot of overhead, the situation in an in-home network could change very quickly — devices may become unavailable for other tasks when for example the device is suddenly switched on by a user.

Decision 7 (One-time assignment or dynamic re-assignment) *Given the dynamic nature of the network, a load sharing solution for an in-home network should enable dynamic re-assignment.*

3.2.7 Summary

Summarising the decisions of the preceding sections, a load sharing solution in an in-home network should have the following properties:

- Load sharing takes place during execution of the system, *dynamically*.

- There is no central device concerned with load sharing, all devices take part in a *distributed* solution.

- Devices act mostly autonomously, with a low degree of *cooperation*.

- Used policies can be changed as the system is running; it is *adaptive*.

- Assigned tasks may be *dynamically re-assigned*.

3.3 Policies of load sharing

Knowing some of the properties a load sharing approach in an in-home network should have, we can now look at its behaviour. We do this by taking a look at four phases through which a load distribution solution — and therefore also load sharing — will go, as presented in [14]. These four phases, or *policies*, as they are usually called, are the *transfer policy*, the *selection policy*, the *location policy* and the *information policy*. These four policies do not concern the actual moving and execution of a task by a server device, as this is specific to the system.

3.3.1 Transfer and selection policies

The transfer policy decides when a device is ready to either move a task to a different device or receive a task from another device. Most implementations of the transfer policy are based on *thresholds*; once a function for the load of the device exceeds or falls below a certain limit, the next phase — the selection policy — is commenced.

In case the device wishes to move tasks elsewhere, it will have to select this task or tasks. Similarly, if a device is willing to execute tasks from other devices, it will have to indicate which of its resources it wants to make available, and how much of them.

As the devices in an in-home network are all different, they should act autonomously. A device can judge best whether a situation calls for moving a task or allows for the execution of other devices' tasks.

Decision 8 (Transfer and selection policies) *Each device implements its own transfer and selection policy, and these policies will not form part of Themis.*

3.3.2 Location policy

Once a task has been selected for migration, a destination will have to be selected for the task, while in the case resources are made available by a device, requests for execution on that server will have to be answered; the location policy determines how this destination is selected.

A simple location policy for distributed load sharing is *polling*, where each device is asked whether it can execute a task (or has a task to be executed). This approach is however not suitable for an in-home network, since devices are not aware of (all) other devices in the network.

An improved version of polling can make an announcement to all other devices without knowing them, using a broadcast message; this also allows devices to react only when interested. This approach, which generates a lot of communication, has the disadvantage that it cannot cope well with changes in the network. To be able to react on new devices or existing devices that have become available (and, analogously, new tasks), devices will constantly have to try and offer their task or services. Apart from the large amount of communication, this also poses overhead on the devices themselves, which is not desirable as they were already trying to move tasks away.

The technique of *bidding* deals with the changing network by putting announcements for tasks in one or more known central locations. Other devices can be notified when new announcements arrive, or periodically check the central location for new announcements. In case a device finds a task it wants to execute, it can make a "bid" to the source of the task, after which the client can choose to whom it will offer the task.

This approach still requires devices willing to execute foreign tasks to check the announcements regularly. This can be prevented by mirroring the solution to the server side. If a device is willing to act as a server, it will place an announcement in the same central location. It then checks if there are any tasks that it could execute. Client devices will act similarly and check for matching servers after announcing of a task. Because a check of the stored tasks and servers is performed after every addition, matching task/server pairs are always found as soon as possible.

Once a client has found a number of candidate servers or a server has found a number of candidate tasks, a decision must be made about which server or task is chosen. Algorithms for making such decisions can range from very simple, taking the first match found, to complex ones that take into account a number of variables. Which method is best depends very much on the type of network; many solutions for specific situations are available in literature (see Chapter 6).

Decision 9 (Location policy) *We use an extended version of bidding as the location policy for Themis, since it allows for a distributed solution where devices can work together without much cooperation, as was desirable (see Section 3.2.7). In order to accommodate adaptiveness, we do not specify an algorithm to choose matches from the candidates found.*

3.3.3 Information policy

The information policy describes the way information is collected to support the decision made by the location policy regarding the choice of the matches. Three problems have to be addressed by this policy: what information is collected, where it is collected, and when.

The kind of information collected completely depends on the location policy. If no additional information is required by this policy, no information policy is required. Because the location policy is executed by the device that was the last to add a task or a server, and all devices may submit a task or register as a server, the information needs to be available to all devices. Collecting it in the same central location therefore seems the most appropriate solution.

Load data can be collected on demand, delivered periodically, or delivered when the data has changed significantly with respect to previous information. If the information would be collected in the same central location as used in the location policy, submitting the required information at that same time is probably the best solution, but any change of information would need to be reflected. If the data is subject to a lot of change, this may not be an appropriate solution.

As the location policy does not specify which information is needed, we cannot specify the information policy in more detail. However, if possible, it will be attempted to collect the information in the same location and at the same time as devices announce tasks and availability.

3.4 Overview

We have explored the properties and policies of a load sharing application in the in-home network, and now combine them in an abstract overview of Themis, summarising the decisions made in this chapter.

Our in-home network consists of a set of *devices*, $\mathcal{D}$. These devices may perform their own private operations, but are also capable of executing zero or more *tasks*, which are contained in $\mathcal{T}$. The sets are not static; devices and tasks may be added or removed at any time.

A device cannot execute every possible task, depending on the requirements of the task and the possibilities of the device. The function $match : \mathcal{T} \times \mathcal{D} \rightarrow \mathbb{B}$ expresses this relationship.

Devices that want to submit new tasks or become a server and execute tasks, can announce this at any time at a message board, $\mathcal{C}$. At $\mathcal{C}$, two collections are kept, one of submitted tasks, and one of registered servers. Each time a task or server is added to one of these collections, it is checked whether there are now any tasks at $\mathcal{C}$ that can be executed by a registered server. This can be verified using the *match* function. If there are multiple possibilities, one of the options is chosen. The algorithm that makes this choice is not fixed; it can be replaced or change while the network is active.

If a server has been found and selected, the task is submitted to that server, and executed. Servers executing tasks can interrupt the execution at any time and try to re-assign the task in the same manner as the original client that submitted the task.

Chapter 4

Architecture overview

The previous chapter discussed the requirements and properties of a load balancing system for an in-home network. In this chapter, we present an architecture for a prototype of the load balancing system Themis in the TASS In-Home Network.

We first discuss how the services of Themis will be made available to the devices in the network, in Section 4.1. Section 4.2 gives a brief overview of how the system can be divided into components. Next, we discuss the six most important actions performed by Themis.

After submission of a task (Section 4.3) and registration of a resource (Section 4.4), have been discussed, Section 4.5 presents the implementation of the central location required for bidding. The location policy is addressed in Section 4.6, while Sections 4.7 and 4.8 deal with two topics not addressed in Chapter 3, the moving and execution of tasks, and returning the results produced by a task. Section 4.9 introduces a special kind of task, the composite task.

For more details on the architecture of the Themis prototype, we refer to the project documentation: [10] and [8].

4.1   Jini service

It is possible to include the Themis load sharing system in the standard Java or Jini libraries, but a more flexible solution is to make it available as one or more Jini services. Devices can look-up these services using only the interfaces and use the returned service objects to perform the desired operations (see also Section 2.3.1). As no back-end process is contacted, these service objects are not proxies; all operations are performed locally.

Making Themis available as a Jini service poses us with the problem of which device offers the service, as we would like the service to be generally available, and not be dependent on the presence of a specific device in the network. This is taken care of by a program called *Hermes*, which is capable of uploading services to current and future look-up servers. The program runs as a part of the device that also runs the Jini services and programs (such as the look-up server). Hermes was also developed during the Themis project, as it was considered to be essential in order to be able to use Themis. More detailed information on the working, design and implementation of Hermes is presented in [15].

Themis will not be made available as a single Jini service, but as a collection of such services. This decreases the size of services to download, and also makes it easier to decompose the system into smaller parts that can communicate with each other through defined interfaces.
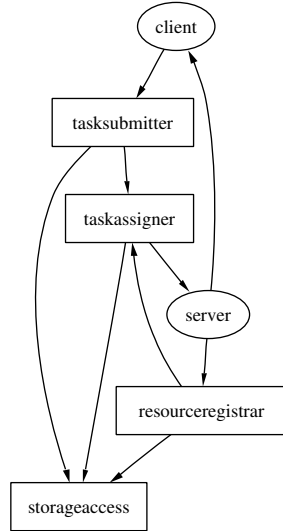
Figure 4.1: An overview of the components in Themis.

4.2 Components

Themis has three types of users, most importantly *clients*, devices that submit tasks and *servers*, devices available to execute tasks[1]. The third type is the *monitor*, a device that is able to display the current status of the Themis system and retrieve the past actions performed by Themis. As the monitor only influences the architecture with respect to a number of details, we ignore this type of user for the remainder of this chapter.

The submission of a task by a client and the registration of a server will be handled by two different Jini services, the Themis components TASK SUBMITTER and RESOURCE REGISTRAR. Both of these components need to store the information received from their users. To do this, they use the STORAGE ACCESS service, which takes care the information is properly stored and that it can be retrieved again. After storing the information on tasks or servers, the TASK ASSIGNER component checks whether there are any tasks and servers that match. If that is the case, the task will be started on the matching server.

In order to execute a task on a server, the server and the client involved will also need to cooperate with Themis. As they cannot become components of Themis, they have to be addressed by means of interfaces. The server will implement the `TaskServer` interface, which allows Themis to pass a task to the server such that it can be executed. Once the task has been completed, the result of the task can be returned to the client, making use of the `TaskClient` interface, which is implemented by the client.

Figure 4.1 displays the components and users of Themis. Arrows are drawn from users or components to the other components or users they activate. Figure 4.2 shows part of the interaction between the components. In this situation, a client submits a task to Themis, which is then stored. A matching server is found, and the task is executed and the result returned.

---

[1] Note that a device can be different types of users; one and the same device can both submit tasks and execute tasks submitted by other devices.
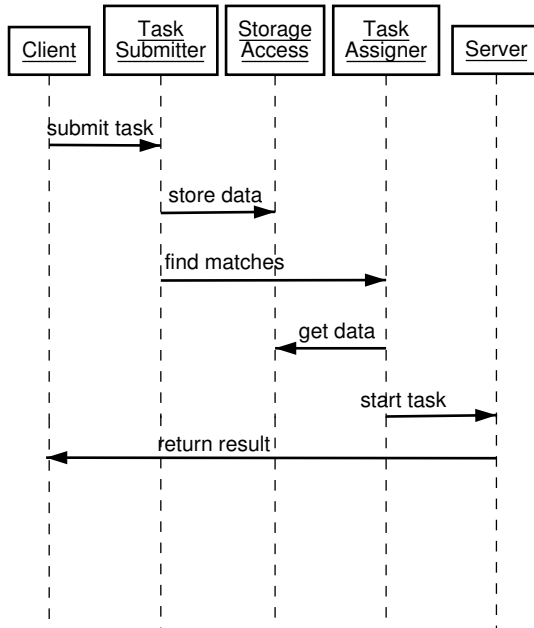
Figure 4.2: Synchronous message sequence diagram of a client submitting a task, execution of the task and returning of the result.

4.3 Submitting a task

4.3.1 Overview

A client wanting to submit a task will do this using the TASK SUBMITTER service, through the `submitTask()` method. In order to handle the task, Themis will require information from the client: the task in question and the object to which the result of the task should be returned. The latter, an object implementing the `TaskClient` interface will be passed as an argument of the `submitTask()` method (see also Section 4.8).

The task itself is also passed as an argument. The object or objects representing the task must be recognisable as a task by Themis, and must be executable by the server without having to know the specifics of the task in question. This is accomplished by hiding the implementation of a task by means of an interface (`Task`)

This is however not yet sufficient information for Themis. Not every task can be executed by every processor; the function *match* from the previous chapter indicates whether that is the case or not. Themis requires information from the task in order to compute *match* for the task and a server.

The required information should express what a server should offer in order to be able to execute that task. We do this through *resources*. A resource can be anything a task could possibly require of a device on which it will be executed, for example a processor with at least 10 MHz, or a Random Access Memory of 256 kB or the capability to output audio in MP3 format.

As suggested by the information policy discussion (Section 3.3.3), the resource information is sent together with the submission of a task. Themis can obtain these resources through the method `getResources()` in the `Task` interface.

The TASK SUBMITTER itself performs only two actions. First, it stores the submitted task and the corresponding client interface by using the STORAGE ACCESS service. Next, it will activate the TASK ASSIGNER component.

20

4.3.2 Resources

From the explanation in the previous section, it becomes clear that a resource has two parts: a description of the type of resource (processor, Random Access Memory) and attributes for these types of resources (processor speed, memory capacity).

Description

Descriptions of resources could be implemented using strings to describe the resources or through a class-like system where resources can inherit from each other. The first method, which does not impose limits on how to describe a resource, has the disadvantage that relationships between resources can not easily be expressed. The second solution, which is similar to the way in which services are found in Jini, uses classes to represent resources, which allows us to, for example, express that Random Access Memory is a sub-class of a general memory resource.

Decision 10 (Resource descriptions) *Resources in Themis will be represented by classes, as this allows for an easy way of specifying inheritance relationships.*

All resource classes in Themis will extend the base class `Resource`. Since tasks will often require more than one resource, we also introduce the class `ResourceSet`, a container class for resource objects.

Attributes

Attributes provide more detailed information about a resource. A straightforward solution to include attributes in a resource class is to make them class variables. These variables can be inherited or overridden by sub-classes of a resource.

Another option is to adopt the way in which Jini handles service attributes, since the description method also resembles the Jini solution. For Jini services, each attribute is a separate object, although they all implement a common interface, `Entry`. The attributes are submitted together with the service that is registered (or searched for). Using the same technique in Themis would mean we could apply an already known technique, with which devices are already familiar. However, the existing Jini attribute mechanism only has the possibility of comparing attributes based on equality; greater-than, smaller-than, or more advanced options are not available. Although it is possible to adapt Jini's attribute mechanism to our wishes, this would be more difficult to develop than our own system, and the users would not be familiar with the system.

Decision 11 (Resource attributes) *As adapting the Jini mechanism to enable more advanced comparisons causes that option to lose its benefits, we therefore decide to implement attributes by means of class variables.*

Example

As an illustration, we give a possible representation of a printer resource. We define three types of printer resources: `PrinterResource`, a generic printer resource, `ColourPrinterResource`, for colour printers, and `BlackWhitePrinterResource`, for black/white printers. There are two ways in which we could place these three resources in a hierarchy: the black/white and colour printer could both inherit direct from `PrinterResource`, or `ColourPrinterResource` could be a sub-resource of `BlackWhitePrinterResource`, as colour printers can also print in black and white — the latter has our preference.

Printer attributes, such as the ability to print in duplex, or the maximum number of pages printed per minute, can be represented by integer or boolean variables.

4.4   Registering a server

Registration of a server to Themis is very similar to the submission of a task discussed in the previous section. A server will use the Resource Registrar service for that purpose, calling the method `registerResources()`, which takes two parameters, a set of `Resources` and an object implementing the `TaskServer` interface.

As with the task, we submit the required information — resources — together with the registration. However, unlike for a task, the available resources of a server are subject to change through time. Reacting to such changes forms part of the transfer and selection policies, which are not part of Themis, and the server device itself is responsible for keeping the resource information of its registration up to date. Because we assume that such changes do not occur very often, updating this information is done by re-registering the server — the previous registration is discarded.

Besides the available resources, we also need to be able to contact the server to start execution of tasks. The `TaskServer` specifies an interface for objects that can receive tasks from Themis. Whenever a task has been assigned to the server, Themis will call a method from this interface.

The actions taken by the Resource Registrar when a server registers a task are the same as those of the Task Submitter: the server information is stored using the Storage Access service and the Task Assigner is activated next.

4.5   Storing data

In order for the Task Assigner service to find all submitted tasks and registered servers, the Task Submitter and Resource Registrar components store the data using the Storage Access service. This service handles all actions regarding the storage and retrieval of data needed by Themis.

4.5.1   Physical storage

One of main reasons for the existence of the Storage Access component is to hide the actual storage from the rest of Themis. There are many ways to store data, but for Themis we require the storage to have a number of properties. The method of storage must enable:

replication to prevent the data from being concentrated in one point

persistence to keep data stored even if a crash occurs

safe storage to store data completely and accurately.

Decision 12 (Storage) *In Themis, JavaSpaces will be used to store data. As this service is already required by the In-Home Network Simulation Framework II, this does not impose any new requirements on the network. Furthermore, JavaSpaces allows direct storage of Java objects.*

JavaSpaces has all the required properties: it can be run in persistent mode, using the RMI Activation Framework to recover data after crashes. Transactions can be used in combination with JavaSpaces to guarantee data is stored correctly and JavaSpaces enables replication of stored data[2].

---

[2]Replication is supported by JavaSpaces, although not implemented in the default version provided by Sun. Third-party implementations that support replication are available, although our prototype will use the default implementation from Sun.

4.5.2 Concurrent access

Besides taking care of storing the objects in JavaSpaces, the main concern of the STORAGE ACCESS component is to handle concurrent access to the storage. Problems may occur if tasks or servers are added (or removed) while the TASK ASSIGNER is busy trying to find matching tasks and servers.

Using transactions can solve this problem. While using the JavaSpaces read, write and take operations under a transaction, other processes trying to access the same JavaSpaces instance are automatically blocked until the transaction has been completed. This allows for a solution where the STORAGE ACCESS performs all its operations under transactions, while hiding this from its users. More details on this and other parts of the STORAGE ACCESS component are discussed in [15].

4.6 Assigning a task to a server

After a submitted task or a server registration has been stored, the TASK ASSIGNER component verifies whether there are tasks that can be executed by an available server[3]. This process takes place on three levels:

1. All matching task/server combinations have to be found

2. To find out whether a task and a server match, their resource sets have to be checked.

3. To determine whether two resource sets match, the individual resources of that set have to be compared.

Chapter 5 deals with this problem in detail.

After all matching tasks and servers have been found, it is possible that some tasks match with multiple servers, or vice versa. In that case, it is necessary to make a choice between the alternatives. This choice is also made by the TASK ASSIGNER component. Details regarding this aspect of the TASK ASSIGNER are discussed in Chapter 6.

4.7 Executing a task

When the TASK ASSIGNER has determined that a task is to be executed by a server, it will contact the server through the `TaskServer` interface that the server provided at registration, calling the method `startTask()`. As the actual object is located at the server, RMI is applied (see Section 4.7.1). If this method completes successfully, the TASK ASSIGNER can assume the task has been correctly started and removes the information regarding the task and server from the storage. This implies that if the server is capable of and willing to execute more tasks, it will have to re-register.

In order to execute tasks concurrently with the processes of the server, we have `Task` extend the Java class `Thread`. This allows the task to be run in a separate execution thread within the server's JVM.

4.7.1 Remote Method Invocation

The `TaskServer` object that is passed to Themis upon registration of a server is the RMI stub object (see Section 2.4). This object can be created at compile-time,

---

[3]Note that it would also be possible to have the server pose requirements on the task it wants to execute. As the objective of load sharing is to execute tasks, rather than to have servers working, we do no consider this option for Themis.

by the developer of the server program, or at run-time by Themis. The latter option was found to be more beneficial, as implementation details were hidden from the user. However, it was impossible to implement the run-time solution, as this required serialisation of un-serialisable objects. Therefore, the stub objects now have to be created at compile-time.

The same problem of serialisation prevented us from making the `TaskServer` activatable. This does not influence performance greatly, in fact, as the servers only have one `TaskServer` object, having an activation daemon could even be more demanding to the server.

The mechanism to return results of a task to the client (see Section 4.8) works in much the same way as starting a task, and RMI has been applied in the same way, with the same restrictions.

4.7.2 Resource usage

In order to prevent tasks from using more or other resources than those they requested, it would be desirable if a server could in some way monitor the use of resources by tasks.

This goal could be achieved by letting the tasks access resources through objects. The server could then pass these objects to the task when it is started; this is difficult to implement, however. Resources like memory and processing power are already granted by starting execution of the task thread and cannot be controlled through such objects. For other resources it may be equally difficult to regulate access in this manner.

Decision 13 (Resource access) *As a system for secure access of the resources of the server is difficult to implement such that it works for all resources, we decide to relegate the responsibility for secure access to the servers themselves[4].*

4.7.3 Pre-emption and states

As described in Section 3.2.6, Themis should allow dynamic re-assignment of tasks. In case a server cannot execute a task completely, it can stop execution of the task completely and submit the task again, following the same procedure as described above and using the `TaskClient` object of the original client. However, in many cases, moving a task will result in the loss of the data computed during execution.

Although the migration of Java objects — including all data used in memory — is technically feasible [13], such methods are subject to severe limitations and require a lot of anticipation in the implementation of the objects. Instead, Themis will collect information on the internal state of the task in a `State` object. This object is created by the task itself when a server stops execution of that task (by calling the method `stopTask()` of the `Task` interface). The same object can be passed to the task when it is restarted, allowing it to resume execution from approximately the same point at which it was stopped.

We decide to also apply this method for tasks that have not been pre-empted: at submission of a task, clients supply a `State` object that is passed to the task when started. This allows tasks to be created independently of the input data, allowing for re-use. Moreover, this allows for more complex constructions of tasks (see Section 4.9).

As the task has to be able to react to interruption, additional handling code is required from the task, which may influence performance of the task. We expect this influence to be negligible, and judge the ability to pre-empt tasks more important.

---

[4]Note that using RMI already requires the presence of a RMI Security Manager. This manager monitors the objects that access the JVM from outside, and can disallow actions such as writing to disk or starting new threads.

It is of course possible for tasks not to implement the `stopTask()` method. As we cannot verify this without running the task, a system in which tasks (or their clients) are authenticated may be necessary. For Themis, however, we assume that servers are capable of stopping tasks non-programmatically if required, and not consider authentication of tasks or clients.

4.8 Returning the result of a task

Once a server has completed execution of a task, it is necessary to return the result of this task to the original client[5]. The task signals its completion to the server by calling a method of the `TaskServer` interface, and passes a result object; this result object is of the same type as the `State` object introduced in the previous section. This leaves several possibilities for returning the result to the client:

- Return the result directly to the client.

- Place the result in storage, and notify the client of this

- Place the result in storage, and have the client periodically check

Directly returning the result has the benefits of being fast and simple. If the client device has disappeared from the network, the result can immediately be discarded.

Placing the result in storage first means results can be retrieved at any time, even after crashes of the client device. A disadvantage is that some clients may never pick up results, cluttering the storage. This can be solved, but imposes additional overhead. Both methods also increase the number of messages sent, especially the option where a client device checks the storage periodically. In Section 1.3.2, we decided not to deal with tasks of disappeared clients, which eliminates the benefits of those storage options[6].

Decision 14 (Result returning) *As we do not consider the tasks of disappeared clients in Themis, we choose to return results of tasks directly to the client by the server that executed the task.*

The server will forward this result to the client using the `TaskClient` object it received together with the task. That object is an RMI stub for the actual object located at the client device (see also Section 4.7.1).

4.9 Composite tasks

The Themis prototype also enables the use of a special type of task to be submitted, the composite task. A composite task, like an ordinary task, is submitted by a client device and a result is eventually returned to it. However, composite tasks are divided into a number of task objects. This allows clients to handle a collection of tasks as one, even if not all tasks can be executed on a single server. As these composite tasks are themselves tasks, it is possible to construct composite tasks using both simple and composite tasks as the building blocks.

Two different types of composite tasks are defined, a parallel composite task and a sequential composite task. Upon submission of a parallel composite task, all contained tasks are submitted to the storage at the same time by Themis, allowing for concurrent execution. The end result is returned when the results of all these

---

[5]While not all tasks will yield a real result, as in the sense of the result of a computation, we assume all tasks at least signal their completion to the original client.

[6]Moreover, an attempt to implement the second option using Jini's distributed event mechanism failed because of the internal working of that mechanism.

tasks have returned. In the case of a sequential composite task, a new task is only submitted to the storage if the result of the previous task has returned.

Like `Task`, composite tasks are defined as interfaces, `ParallelTaskCollection` and `SequentialTaskCollection`. These interfaces specify methods to obtain the sub-tasks and the corresponding states, and to obtain the final result from the results of the sub-tasks. As the clients implement the interfaces, very flexible solutions are allowed. For example, sequential composite tasks can base the next task to be submitted on the results of the previous task.

Composite tasks also require adaptation of the TASK SUBMITTER component, since the intermediary results have to be received and handled before the end result can be returned. For this purpose, two new classes implementing the `TaskClient` interface are used, one for parallel and one for sequential composite tasks. Upon submission of a composite task, a new instance of one of these classes is created to handle the composite task. This new object then extracts the tasks, submits them, receives the results and eventually returns the end result to the `TaskClient` defined by the client device.

Chapter 5

Finding matching tasks and servers

As discussed in Section 4.6, the first task of the TASK ASSIGNER service is to identify matches within the sets of submitted tasks and registered servers. The identification of matches takes place at three different levels: the individual resource level (Section 5.1), the resource set level (Section 5.2) and the system level (Section 5.3).

5.1   Matching resources

In order to verify whether resource objects (see Section 4.3.2) match with each other, we define a function *matchresource*: $\mathcal{R} \times \mathcal{R} \to \mathbb{B}$, where $\mathcal{R}$ is the domain of resources.

The function *matchresource(r, s)* is true if a task requiring resource r can be executed on a server offering resource s. *matchresource* is therefore transitive and reflexive, but not symmetric.

As a resource consists of a type and a set of attributes, the comparison consists of two parts. The first part of the comparison should inspect the types of the resources; if the types are not compatible, it is not necessary to check the attributes. Two resources r and s can only match if the type of s is the same as that of r, or if the type of s is a sub-type of that of r.

The attributes of a resource are specified for types, and no generic comparison of these can be defined. In fact, every instance of a resource can define its own comparison method. This allows for complex comparisons of attributes of a resource.

Example   Using the printer resources from Section 4.3.2, we take resource r as an instance of `BlackWhitePrinterResource` with duplex printing at 10 pages per minute, and s as an instance of `ColourPrinterResource` with duplex printing at 9 pages per minute. Although r and s do not have the same type, `ColourPrinter-Resource` is a sub-type of `BlackWhitePrinterResource`, and the two types match. r and s also match on the duplex attribute, but r requires a printing speed of 10 pages per minute, which is not matched by s. Therefore, r and s do not match.

5.2   Matching resource sets

Being able to check whether individual resources match is not sufficient to find out if a task can be executed on a server. Tasks usually require more than one resource, and servers will have several resources to offer. These resources are contained in

resource sets[1], and we therefore will have to be able to verify if two resource sets match.

A simple approach to verify if two resource sets match, appears to be to compare each of the elements of the task resource set with each of the elements of the server resource set. If a matching resource is found for each resource required by the task, we have a match. This approach, however, does not guarantee that the sets actually match: if the task requires two resources of the same type, we may find that they both match with the same resource in the server's resource set, but this is not sufficient, as there need to be two such resources to actually execute the task.

5.2.1 Problem description

Given are two sets of resources X (the resources of the task) and Y (of the server). For all resources $r \in X, s \in Y$, a function *matchresource(r, s)* is defined, which equals `true` if and only if a task requiring r could be executed on a server offering resource s (see Section 5.1).

Required is to find whether an injective[2] mapping M of X onto Y exists, where for all $x \in X$, *matchresource(x, M(x))* holds. The predicate *match(X, Y)* indicates if such a mapping exists for resource sets X and Y.

5.2.2 Bipartite graph matching

In graph theory, the problem domain of *matching* deals with problems similar to the resource matching problem. We can represent the two resource sets as the two partitions of a bipartite graph $G = (X \cup Y, E)$, where each resource is a vertex in X or Y, and where there is an edge between two vertices x and y if $x \in X$, $y \in Y$ and *matchresource(x, y)* holds.

The resource matching problem can then be described as finding whether a subset $N \subseteq E$ exists such that all vertices in X are incident to exactly one edge of N and such that all vertices in Y are incident to at most one edge of N. This problem is known as finding whether a bipartite graph matching of X *into* Y exists. *Hall's Matching Condition* [16] provides a necessary and sufficient condition for such a matching:

Hall's Matching Condition A bipartite graph $G = (X \cup Y, E)$ has a matching of X into Y if and only if $|nb(S)| \geq |S|$, for all $S \subseteq X$, where *nb(S)* is the set of neighbours of S, vertices connected to a vertex in S.

A simple algorithm checking all the subsets of X has worst-case complexity $\mathcal{O}(2^{|X|})$. We could improve that algorithm, observing that G can be divided into a number of bipartite sub-graphs with no edges between them. For example, if X contains a memory resource, the corresponding vertex will only have edges to vertices in Y that are also memory resources, as the edges indicate that the two resources match. However, an algorithm exploiting this is still exponential, and we prefer a polynomial one[3].

The most common problem in bipartite graph matching is to find a perfect matching, that is, a subset of the edges such that each of the vertices is incident to exactly one edge of this subset. This problem is solved by repeatedly applying the *Augmenting Path Algorithm* [16], which yields the *maximum* matching, or the

---

[1]Resource sets may contain resources more than onece, and are therefore formally multi-sets.

[2]A function $f : X \to Y$ is injective if $(\forall x, x' : x, x' \in X : (f(x) = f(x')) \Rightarrow (x = x'))$.

[3]In the prototype of Themis, an algorithm based on Hall's Matching Condition is implemented, also using the partition observation. The polynomial algorithm described has not been implemented.
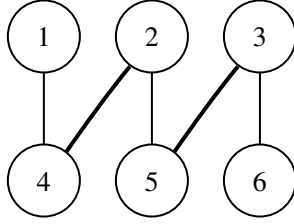
Figure 5.1: An augmenting path.

matching with the greatest number of edges. If this matching covers all vertices, it is also a perfect matching. This *Maximum Matching Algorithm* has worst-case complexity $\mathcal{O}(|V|^3)$, although optimisations may give a performance of $\mathcal{O}(|V|^{5/2})$. We show that this algorithm can be used to solve our resource matching problem, giving a polynomial-time solution.

First, we define more properly what the Maximum Matching Algorithm computes. The outcome of the algorithm is a matching $N \subseteq E$, for which each of the vertices $v \in X \cup Y$ is incident to at most one edge in N, and for which there are no other such matchings with a number of edges greater than $|N|$. Our claim is that *match(X, Y)* holds if and only if all vertices in X are incident to an edge in any maximum matching N found by applying the Maximum Matching Algorithm.

Proof First, we prove sufficiency of the claim. Given a matching N with all vertices in X incident to one of the edges in N, we can directly derive a mapping M from X to Y. To construct M, we define $M(x)$ as the element from Y which is incident to the same edge in N to which x is incident. Since all vertices of X are incident to exactly one edge in N, and all vertices in Y are incident to at most one edge (since N is a matching), mapping M is injective. As an edge between two vertices implies the corresponding resources match, we have found a correct mapping M, and *match(X, Y)* holds.

Now, we assume $match(X, Y)$ holds. If we have a maximal matching N with not all vertices in X covered, it follows that there cannot be an injective mapping from X to Y, since that requires all of the elements of X to be mapped; this contradicts the assumption that *match(X, Y)* holds. □

We now give an outline of the Maximum Matching Algorithm. As mentioned, the Maximum Matching Algorithm is based on the Augmenting Path Algorithm. That algorithm finds, if it exists, and *augmenting path* in G, given a matching N. An augmenting path is a path over the edges of G, for which the edges alternately do and do not belong to N, and for which the first and last vertex reached are not *saturated*[4] by N. In Figure 5.1, where edges contained in N are shown in bold, the path 1–4–2–5–3–6 is an augmenting path.

The Augmenting Path Algorithm finds an augmenting path in the following way. Given G and N, we define three sets: U, the set of all vertices in X not saturated by N; $S \subseteq X$ and $T \subseteq Y$ are sets of vertices that are reached from U (initialised by $S = U$ and $T = \emptyset$). Next, we take an element s from S and consider all vertices in $y \in Y \setminus T$ that can be reached from s through an edge not in N. If y is not saturated by N, we have found an augmenting path P. If y is already saturated by N, add y to T and the vertex connected to y by the edge in N to S. Now repeat

---

[4]A vertex v is saturated by a matching N if there is an edge in N incident to v.

this step until an augmenting path has been found, or until all vertices in S have been checked.

We demonstrate the Augmenting Path Algorithm by finding the augmenting path in Figure 5.1 with $X = \{1, 2, 3\}$ and $Y = \{4, 5, 6\}$ and the matching N indicated in bold. Initially, the set U consists of vertex 1, hence S also contains 1, while T is empty. The only edge from 1 to a vertex in Y is $(1 \sim 4)$, which is already satisfied by N. We therefore add 4 to T and add 2, which is connected with 4 in N, to S. We repeat the procedure for $(2 \sim 5)$, and add 5 to T and 3 to S. In the third iteration, we find that 6 is not yet saturated by N; an augmenting path has been found, which can be reconstructed by tracing back the path followed, in this case 1–4–2–5–3–6.

```
1   P  :=  ∅ ;
2
3   U  :=  {x | x ∈ X ∧ saturates(N, x)} ;
4   S  :=  U ;
5   T  :=  ∅ ;
6   V  :=  ∅ ;
7   W  := ∅ ;
8
9   do  S \ V ≠ ∅ ∧ P = ∅  →
10        forall  s ∈ S \ V  and  while  P = ∅  do
11            forall  y ∈ (Y \ W)  and  while  P = ∅
12                if  s ∼ y ∧ (s ∼ y) ∉ M  →
13                    T  :=  T ∪ {y} ;
14                    W  :=  W ∪ {y} ;
15                    if  saturates(N, y)  →
16                        P  :=  new path ;
17                    □¬saturates(N, y)  →
18                        S  :=  S ∪ {x | (x ∼ y) ∈ M} ;
19                    fi
20                fi
21            od
22            V  :=  V ∪ {s} ;
23        od
24  od
```

Listing 5.1: Augmenting Path Algorithm

The Maximum Matching Algorithm (Listing 5.2) repeatedly applies the Augmenting Path Algorithm, starting with a an empty matching. If the Augmenting Path Algorithm yields an augmenting path P, we can increase N by removing the edges of P that were in N and adding the edges of P that were not in N — in Figure 5.1, the edges removed are $(2 \sim 4)$ and $(3 \sim 5)$; $(1 \sim 4)$, $(2 \sim 5)$ and $(3 \sim 6)$ are added to N. If no augmenting path was found, then N is a maximum matching in G, and the algorithm terminates.

```
1   N  :=  ∅ ;
2   do  AugmentingPathAlgorithm(G, N) ≠ ∅  →
3        P  :=  AugmentingPathAlgorithm(G, N) ;
4        N  :=  (N \ N ∩ P) ∪ (P \ N) ;
5   od
```

Listing 5.2: Maximum Matching Algorithm

5.2.3 Finding matches

The previous section silently assumed that the matches between all the resources of the two resource sets — the edges in the graph — were already known, which is however not the case.

A simple algorithm checks for all resources in X which resources in Y match — worst-case complexity is $\mathcal{O}(|X| \cdot |Y|)$. However, for finding the matches between resources, we may benefit from the same observation made in the previous section: a memory resource will only match with another memory resource. This knowledge makes it unnecessary to search all resources in Y to see if there is a match. Instead, it is sufficient to check all resources in the partition of Y which contains potential matches with the element from X. Although the worst-case complexity is still the same, this algorithm has a better performance for most cases.

```
1  E := emptyset;
2  forall resources x in X do
3      Y_x = partition of Y that can match with x;
4      forall resources y in Y_x do
5          if matchresource(x,y) →
6              E := E ∪ {(x ∼ y)};
7          fi
8      od
9  od
```

Listing 5.3: Resource match finding algorithm

In order to implement this, we need to pay attention to construction and representation of the partitions. For finding resources in Y matching a resource $x \in X$, we need to define the partition Y_x of Y in which to search. As we wish to make as little comparisons as possible, we want to make Y_x as small as possible. We therefore include in the partition all resources y that could match with x based on the resource type. That is, for a resource x of type A, Y_x contains all resources that either have type A, or have a subtype of A.

Since we have introduced the partitions to decrease the number of comparisons made between resources, representation of the partitions should not pose a lot of overhead to the system. For representation, we apply the fast technique of *hashing*, using Java's `Hashtable` data type.

In hashing, a *hash-function* is used to map input values from a large input domain to values from a smaller co-domain. Using a value from the domain — called the *key* — we can store a value associated with that key. In our case, we use the type — or super-type — of the resource and store the partition as the value. In other words, the hash function maps resource types to the power set of resources.

The hashtable data type makes it possible to store and retrieve the resource sets quickly using only the type of a resource.

5.3 Finding matching tasks and servers

Being able to determine whether a task and a server match or not, we can also define an algorithm to find all such matches given all tasks and servers. From storage, the TASK ASSIGNER obtains two sets: a set of submitted tasks $\mathcal{T}$ and a set of registered servers $\mathcal{S}$[5]. While the TASK ASSIGNER is active, no new tasks or servers can be added (see Section 4.5.2). Tasks or servers may only be removed from the sets by the TASK ASSIGNER if either a task is being executed by a server, or if the device that submitted the data has become unavailable.

---

[5]The elements of both $\mathcal{T}$ and $\mathcal{S}$ are sets of resources.

The objective of the Task Assigner is to assure that for none of the tasks in $\mathcal{T}$ there is a matching server in $\mathcal{S}$. This can be formulated as the following post-condition:

$$(\forall t : t \in \mathcal{T} : \neg(\exists s \in \mathcal{S} :: match(t, s)))$$

The function *match* is defined as $\mathcal{T} \times \mathcal{S} \to \mathbb{B}$, and returns whether task t can be executed on server s (see Section 5.2).

A simple algorithm testing all combinations of tasks and servers would have worst-time complexity $\mathcal{O}(|\mathcal{T}| \cdot |\mathcal{S}|)$, not taking into account the complexity of the *match* function. This performance can be improved using the observation that a Task Assigner run is made after each addition of a task or server (see Section 3.4); only these additions[6] made between to runs can therefore "disturb" the post-condition reached by a previous Task Assigner run. Therefore, the following pre-condition holds:

$$(\forall t : t \in \mathcal{T} \setminus \mathcal{T}_{new} : \neg(\exists s \in \mathcal{S} \setminus \mathcal{S}_{new} :: match(t, s))),$$

where $\mathcal{T}_{new}$ and $\mathcal{S}_{new}$ are defined as the subsets of $\mathcal{T}$ and $\mathcal{S}$ that contain the newly added tasks and servers. Note that this pre-condition also holds for the initial situation, where $\mathcal{T} = \emptyset$ and $\mathcal{S} = \emptyset$.

From these conditions follows the program listed in Listing 5.4. The pairs of tasks and servers that match are collected in the set M. The function *match* is discussed in detail in Section 5.2. Again not taking the complexity of *match* into account, worst-time complexity of this algorithm is $\mathcal{O}((|\mathcal{T}_{new}| \cdot |\mathcal{S}|) + (|\mathcal{S}_{new}| \cdot |\mathcal{T} \setminus \mathcal{T}_{new}|))$; for most cases, where only one change is made, this amounts to either $\mathcal{O}(|\mathcal{T}|)$ or $\mathcal{O}(|\mathcal{S}|)$.

```
1   M := ∅;
2   forall t in T_new do
3       forall s in S do
4           if match(X_t, Y_s) →
5               M := M ∪ {(t, s)};
6           fi
7       od
8   od
9
10  forall s in S_new do
11      forall t in T \ T_new do
12          if match(X_t, Y_s) →
13              M := M ∪ {(t, s)};
14          fi
15      od
16  od
```

Listing 5.4: Task/server matching finding algorithm

---

[6]It is possible for multiple additions to take place between two runs, as the storage is not blocked between the storage of a task or server and the start of a Task Assigner.

Chapter 6

Selection of matches

Once all matching tasks and servers have been identified, some tasks or servers may occur in multiple matches. As it is not possible or necessary to assign tasks or servers twice, Themis needs to make a selection of the task/server pairs found. We call this selection the match choosing algorithm.

As decided in Section 3.2.5, Themis must be adaptive. For selection of matches, this implies the manner in which the selection is made must not be fixed. Section 6.1 presents an architecture to accommodate this adaptivity, while Section 6.2 discusses the possibilities for the algorithms themselves. Some example algorithms are presented in the final section of this chapter.

6.1 Architecture

A flexible structure for adaptive match choosing algorithms separates implementation from its use, allowing for change of algorithm during compile-time and run-time.

In object-oriented programming, and especially in Java, *interfaces* are commonly used to separate a specification from an implementation; interfaces define functions, which can be implemented by a class. This technique could also be used for the match choosing algorithms; the TASK ASSIGNER can use any algorithm, as long as it implements a known interface.

An alternative solution is suggested in [7], where it is called the *Strategy pattern*. Instead of using an interface, an abstract class is defined as a skeleton for the algorithm (see also Figure 6.1). Actual implementations of the algorithm extend this class and are obliged to implement the operations defined by the abstract super-class.

Decision 15 (Algorithm architecture) *We choose the Strategy pattern solution*
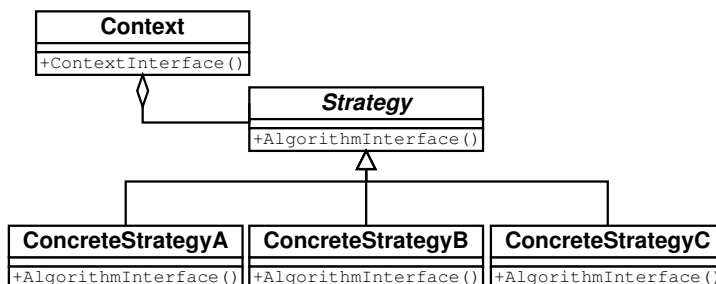

Figure 6.1: The structure of the Strategy pattern, from [7].

to enable the use of different match choosing algorithms, as that approach has the possibility to include code with the abstract class, which can be re-used by the algorithms. This is also beneficial in case algorithms are arranged in a hierarchy.

The abstract class for the match choosing algorithm is called `bestMatchAlgorithm`. The process of selecting matches is commenced by passing the set of all matches found to the concrete implementation.

An algorithm is responsible for the actual starting of the tasks, calling the `startTask()` method from the `TaskAssigner` interface. This leaves open the possibility to choose a different server whenever starting a task fails because the chosen server has become unavailable.

6.2 Match choosing algorithms

Algorithms for choosing matches from all matching task/server pairs are part of the location policy, as discussed in Section 3.3.2. Implementations can range from very simple algorithms that assign each task to the first matching server found, to complex algorithms that consider future scenarios and use lots of additional information on the network. In this section, we discuss some of the options for implementation of a match choosing algorithm.

6.2.1 Algorithm complexity

It is claimed that, for a small numbers of matches, simple algorithms often give the best result in situations where a high performance is demanded. More sophisticated algorithms tend to produce overhead, as they require more information to be collected and take longer to execute.

One could therefore argue that simple algorithms would also be best for Themis, as more complex algorithms merely increase overhead. On the other hand, the overhead posed by Themis itself is already considerable regardless of the algorithm chosen here. Moreover, increase of performance is not the only goal targeted by Themis; it can also be used for adding functionality or providing reliability.

Match choosing algorithms that employ load information to make matching decisions could be useful for use in in-home networks, but are very difficult to implement, given the dynamic and diverse nature of in-home networks. Unfortunately, we are not in a position to study load-related variables in in-home networks.

6.2.2 Using resource information

Although no load information is available, Themis already has access to the desired and offered resources. It is possible to use knowledge about the resources in a match choosing algorithm, since not every server matching the requirements of a task is equally suitable. A server with a faster processor than requested might have preference over a server with only just the required speed, but more memory than required will likely not be considered more beneficial for a task. We could use such information to choose between different servers all matching the same task (or vice versa).

However, given the diversity of devices and resources, it is difficult to compare them and define an order. We can attempt to do this by assigning scores to each match. Given the manner in which resources are matched, it is the easiest to compute this score from the viewpoint of the task — that is, the client defines how the score is computed — although doing it from the server's point of view is also possible. In the remainder of this section, we assume the score is computed from the task's viewpoint.

Unfortunately, having a comparable measure for all matches still leaves open several possibilities for a solution. We could assume that it is best to find a high scoring server for all tasks, as these are then performed the faster, and by the best available servers. An opposite reasoning is also possible, however: if we find a low scoring server that is still sufficient to execute a task, this leaves more high-scoring servers for possibly more demanding future tasks. It is also possible to do this assignment on network level: trying to assign tasks to servers with a minimal/-maximal total score. The problem would even get more complex if we take into account that servers can actually execute more than one task at the same time; this is however outside the scope of this project (see Section 1.3).

6.2.3 Number of different tasks and servers

Another issue to consider is the number of different tasks and servers that occur in the matches. Since the TASK ASSIGNER is activated after each addition of a task or server, the matches found will most probably only concern the newly added task or server. With this in mind, we could greatly reduce the complexity of some algorithms. However, in those few cases where more than one task or server is taken into account, such algorithms may perform badly.

This problem can be solved by selecting the appropriate match choosing algorithm at run-time, which is allowed by the architecture. It could be detected whether the amount of matches found deals with one or many different tasks/servers and the the appropriate algorithm is then chosen.

6.2.4 Summary

Without knowing actual in-home network behaviour and properties, it is very difficult to determine an optimal match choosing algorithm. Even if such knowledge were available, it is very unlikely that this yields one algorithm, if only due to the fact that no two in-home networks are the same and their composition can change dynamically.

It is most likely that the best solution for a match choosing algorithm consists of several algorithms, and it is dynamically decided which one to use, based on variables such as the number of different tasks/servers involved. Care should be taken that the decision on which algorithm to take does not get too complex.

6.3 Algorithm examples

In this section, we present algorithms for selecting matching tasks and servers. The first is First Fit, which simply selects the first match found. Next, we discuss some other "fit" algorithms. The weighted matching algorithm can be used when scores have been assigned to matches, for example those based on the available resources.

6.3.1 First Fit

First Fit is probably the simplest match choosing algorithm. It assigns each task to the first encountered server to which it matches. This algorithm would be most efficient if it were applied directly when a match is found. This, however, circumvents the architecture presented in Section 6.1. Listing 6.1 shows First Fit as it would be without integration in the match finding; M denotes the set of matches, consisting of pairs of tasks and servers.

```
1       T_a := ∅;
2       S_a := ∅;
```

```
3      forall pairs (t,s) in M do
4          if  t ∉ T_a ∧ s ∉ S_a  →
5              if assigning t to s is successful  →;
6                  T_a  :=  T_a ∪ {t};
7                  S_a  :=  S_a ∪ {s};
8              fi
9          fi
10     od
```

Listing 6.1: First Fit algorithm

6.3.2   Other "fit" algorithms

An number of simple algorithms based on the "fit" of a match exist. The fit is determined using a measure, for example the resource scores discussed in the previous section. In that case, we can each task to the server with the best fit, or worst fit, provided the server has not been assigned before. The definitions of best and worst fit vary, but usually best fit is defined as the match with the lowest score that is still high enough. Other variants also exist, such as the Almost Worst Fit (which takes the next to worst fit).

All these algorithms require inspection of all matches for a particular task or sever, which makes them slower than First Fit. Listing 6.2 shows an example of a Best Fit algorithm. For all tasks t for which a match was found, we inspect the set M_t of matching servers. The server with the lowest value for the fit function is eventually chosen. The algorithm does not check whether a task was already assigned to that server, but this can easily be added (see Listing 6.1).

```
1  forall t do
2      forall s ∈ M_t  do
3          best  =  null;
4          if  fit(s) < fit(best)  to
5              best  =  s;
6          fi
7          assign t to best;
8      od
9  od
```

Listing 6.2: Best Fit algorithm

6.3.3   Weighted matching

As described in Section 6.2.2, a way to compare matches is to assign scores to them. One way to select matches after scoring has been applied is to take these matches that give the highest (or lowest) total score.

This problem can be translated into a graph problem known as *weighted bipartite matching*. Given are a bipartite graph $G = (X \cup Y, E)$ and a function $w : E \to \mathbb{N}$ that assigns weights to the edges. Requested is to find a matching — a set of edges $N \subseteq E$ such that each vertex is incident to at most one edge in N — for which the sum of the weights is maximal.

We define the tasks as the vertices in X and the servers as the vertices in Y. A match between a task and a server is indicated by an edge between their vertices; the score of the match is the weight of that edge.

The weighted bipartite matching problem can be solved using the *Hungarian algorithm*, first developed by Kuhn and Munkres (see [16]). It finds the matching

with the maximum sum of weights, but could be adapted to find the minimum sum instead.

The Hungarian algorithm requires the graph to be complete (that is, there are edges between all vertices) and balanced ($|X| = |Y|$). If this is not the case, vertices and edges with weight 0 can be added without loss of generality.

We define a *feasible labeling* l on the vertices of G as a function $l : V \to \mathbb{N}$ such that for all vertices, the sum of the labels of two vertices is greater than or equal to the weight of the edge between them: $l(x) + l(y) \geq w(x \sim y)$[1].

For a labeling l, we define the *equality sub-graph* as the graph $G_l = (X \cup Y, E_l)$, with $E_l = \{(x \sim y) \mid l(x) + l(y) = w(x \sim y)\}$; that is, a graph with the same vertices as G, but only with the edges for which the weight of an edge is equal to the sum of its vertices labels.

It can be proven that if, for a feasible labeling l, the equality sub-graph G_l has a perfect matching M, then M is a maximum weighted matching for G. Note there is always an equality sub-graph with a perfect matching, as graph G is complete.

The Hungarian Algorithm finds the maximum weighted matching in the following way. Given G, we define a feasible labeling l and the corresponding G_l. A feasible labeling always exists: define $l(y) = 0$ for all vertices in Y, and set $l(x)$, for all vertices in X to the maximum weight of the edges it is incident with.

Next we take a maximum matching M in G_l, which can be found by applying the Augmenting Path Algorithm (see Section 5.2.2). If M is perfect, we have found the maximum weighted matching.

Otherwise, take the sets S and T at the end of the Augmenting Path Algorithm. Define ε as the minimum $l(x) + l(y) - w(x \sim y)$, with $x \in S$ and $y \in Y \setminus T$, and update labeling l as follows: $l(x) := l(x) - \varepsilon$ for all $x \in S$ and $l(y) := l(y) + \varepsilon$ for $y \in T$. This labeling is still feasible. Repeat the previous steps for the G_l of the new labeling until a perfect matching M for G_l is found.

The Hungarian Algorithm has a worst-case complexity of $\mathcal{O}(|V|^3)$.

---

[1]$x \sim y$ denotes the edge that connects vertices x and y.

Chapter 7

Evaluation and conclusions

7.1 Prototype evaluation

With a prototype implementation of Themis available, it is possible to evaluate decisions made during its development and its other properties. Before doing so, we first give a brief overview of the prototype. A complete overview of the prototype and its development is given in the Software Requirements Specification ([11]), Software Design Description ([10]), Detailed Design Description ([8]).

7.1.1 Prototype

The Themis prototype implements the architecture as described in Chapter 4, and contains all interfaces and implementations of the four Themis services. All other interfaces concerning the users, tasks and resources have been implemented as well, to be able to demonstrate the prototype. The part of the task assigner that finds matches was implemented differently from what is described in Chapter 5.

The demonstration features a client that can submit a number of tasks. These tasks compute values required for the graphical representation of a *Mandelbrot set*. A dummy resource is required for executing such tasks, and all servers have this type of resource available. These servers register themselves to Themis as available on start-up, and remain registered unless they are executing a task that was assigned to them. The client is also capable of submitting a parallel composite task and a sequential composite task.

A separate monitor device makes it possible to view, at any time, the submitted tasks, registered servers and tasks that are being executed; all actions performed by Themis are also written in a log file, which can be read from the monitor device.

The simulated in-home network has been extended with Hermes. This program is coupled to a look-up server and takes care of the registration of the Themis services. For the storage of tasks and server registrations, the default JavaSpaces implementation is used, which however does not support replication of data. A commercial implementation, called GigaSpaces, does have this feature, and could be used by Themis if required.

7.1.2 Evaluation

Prototype environment

Although the principles of Jini and its related technologies are easy to understand, the combination with the In-Home Network Simulation Framework II and Themis forms a complex union. Although Themis and RMI offer a number of useful features,

which relieved us from implementing them ourselves, their behaviour and properties were not always clear, or slightly different behaviour was required for Themis. Much time and effort was spent to find out how Jini or RMI worked, in combination with our own product.

The In-Home Network Simulation Framework II, which has the purpose to make using Jini easier, was unfortunately hardly used. This was mostly due to the fact that Themis does not behave like most Jini users or services, but insufficient documentation of the framework made it difficult to find out whether the offered functionality could be used our not.

With Themis, yet another addition was made to the already complex combination of different components and layers. This makes the TASS in-home network prototype increasingly difficult to extend, and care should be taken that the work done does not become valueless because of its complexity.

Development decisions

Of the decisions made during research and design of the prototype, the only decision made that was not fully lived up to is the decision to apply distributed load sharing (see 3.2.3).

In Themis, JavaSpaces is used to store data regarding submitted tasks and registered servers. In situations with only one look-up server, the data is concentrated in one point, and even with several look-up servers, the situation is not fully distributed. Therefore, one of the benefits of our decision, non-dependency, does not hold here. One could argue that the presence of a look-up server and the use of JavaSpaces already make the in-home networknetwork dependent on a small number of devices, but that is particular to the current implementation with Jini, while the storage would also be present if Themis would be implemented in a different environment.

On the other hand, a completely distributed solution, is not possible, as it would require all devices to be able to store data on tasks and servers. Clearly, not all devices are capable of this.

It should also be noted that the other reasons why distribution was chosen — easier because of heterogeneity and scalability — are not violated. Devices all decide themselves when and how to participate, and the replication functionality of JavaSpaces allows for a fully scalable solution.

Performance

Use of the demonstration set-up indicates that use of Themis poses an additional overhead to the system. As this overhead is already considerable while using powerful hardware (PC), using actual in-home devices will likely only increase the overhead.

Experiences with in-home network prototypes without Themis, however, show that this overhead is also present in these systems, and is mostly due to the use of Java and Jini, rather than the load sharing system. The performance of Java and Jini is therefore a topic that requires future attention.

7.2   Conclusions

The main objective of the Themis project (see Section 1.3) was to investigate whether it was technically feasible to add a load balancing system to an in-home network. This was shown to be possible — although applying load sharing rather than balancing — by implementing a prototype for Themis. We have already discussed the properties of this prototype in the previous section, but have not yet

answered the question whether we can actually apply such a load sharing system, and if it would be useful.

7.2.1 Technical suitability

Judging by current household appliances, an in-home network typically contains a small number of devices that execute specialised tasks. A load sharing system like Themis is not very beneficial to such a network.

The main advantage of Themis is that it hides the selection of a destination for tasks from a potentially large group of candidates, and that it can do that for large numbers of tasks and servers. For a network with few devices and tasks, the overhead posed by Themis is therefore big; it is probably best to let the selection of destinations be handled by the devices themselves, which is possible in the service-based Jini environment.

The overhead might be reduced if the load sharing system is implemented differently. The combination of Java and Jini already poses quite a large overhead, and adding Themis as an additional layer only increases that. A configuration where load sharing is integrated with the network technology (such as Jini), or even in the primary soft- or hardware, would probably greatly reduce the overhead.

In the current configuration (that is, on top of Jini), Themis would be more useful in environments with more devices and/or more — and therefore less specialised — tasks. As the number of devices in an in-home network is not expected to grow much in the future, the devices would be required to have more general computing capabilities, allowing them to execute more kinds of tasks — a change that will possibly occur in the future. Alternatively, we could consider applying Themis in other networks that already have the desired properties, for example in office networks.

7.2.2 Practical suitability

As for practical application of a load sharing system in an in-home network, we see three areas in which such a system could enhance the network: performance, functionality and reliability.

The most obvious advantage load sharing can offer is a better performance. Devices are able to perform their jobs quicker by letting other devices do parts of it, idle devices could relieve busy ones from their jobs and improve response time, and tasks can be executed with a higher level of quality. However, the currently existing household devices do not have performance problems and they will only become more powerful, as an increase in capacity is already required to become part of an in-home network. Hence, an increase in performance is currently not needed. The increase in performance because of the load sharing system will not make it possible either to weaken the capacity of the devices, as no execution of tasks is currently guaranteed.

The improved performance of the devices does leave additional options for new functionality, besides the possibilities enabled by the fact that devices can execute each other's tasks. Few concrete ideas for such new functionality are in existence however, as development of in-home networks is still in an early stage. The availability of a prototype with the possibility of load sharing may give rise to new ideas and enable further research.

A third use for load sharing in in-home networks could be to improve reliability, as it is now easy to execute tasks in duplex. However, as current household devices hardly suffer from reliability problems, there seems to be no real need for this type of use.

7.2.3   Summary and recommendations

Summarising, we conclude that a load sharing system for an in-home network is technically possible, but such a system is not yet very useful, as its overhead is disproportional with respect to the network, and no applications are yet available.

Nevertheless, we think that further study of the combination of in-home networks and load sharing is useful. Such studies should initially focus on finding applications for load sharing within in-home networks and other network environments. Technical studies will not be of much use until a definite technical environment for in-home networks has been set up, or if no applications for load sharing are found.

When conducting research, it may also be useful to examine some of the issues that were not, or not fully, addressed by Themis, but are interesting for future development. These issues include:

Server capacity The fact that servers can execute more than one task at the same time has not been explored, but could lead to quicker assignment of tasks, which is especially interesting in the case of tasks with a short execution time.

Dividable tasks As an extension of composite tasks, we can create dividable tasks, which can be split up in smaller parts when the available servers do not have sufficient capacity for executing the entire task.

Scheduled tasks The desired period of execution for some tasks is, on some occasions, known beforehand. The possibility of reserving server time in advance could be beneficial for such tasks.

Execution guarantee Although we must assume an in-home network to be unreliable, it might be possible to offer better guarantees for execution of tasks than presently, or at least better exception handling when failure does occur.

Security For Themis, we have made some assumptions regarding the behaviour of tasks with respect to security. Measures need to be taken to enforce this behaviour rather than assuming it.

Transfer and selection policies Although not part of Themis, the transfer and selection policies play an important role in load sharing — implementation of these policies in devices is required.

Appendix A

Object-oriented terminology

In this appendix, we present a few object-oriented terms that have no common well-defined meaning. Since the Themis prototype was developed in Java, names and their meaning depend on their use in that programming language.

abstract class An abstract class is a class of which no instances can be made; it can only be extended.

class A class defines a category, or type. A class can define properties of data and operations — methods — on that data. A class can be instantiated by an object. Classes can extend another class, inheriting the properties and operations, which it can add to or modify. A class extending another class is a *sub-class*, a class that is extended by another class is a *super-class*.

interface An interface specifies how interaction between two objects can take place by defining method names. Classes can *implement* an interface, which allows other objects to communicate with them while only knowing the interface and not the class itself.

method A method is a function defined on a class. Objects can be passed as arguments, and a method can return zero or one objects as a result.

object An object comprises data of a certain type, or class.

thread A thread is a separate process in the Java Virtual Machine. Objects of a class that extends the `Thread` class, are run in separate threads.

Bibliography

[1] Ken Arnold, Eric Freeman, and Susanne Hupfer. *JavaSpaces — Principles, Patterns and Practice*. Addison-Wesley, 1999.

[2] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison-Wesley, 1999.

[3] Natalya Belousova. In-Home Network Simulation Framework. Technical report, 2000.

[4] Thomas L. Casavant and Jon G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. In *IEEE Transactions on Software Engineering*, volume 14, 1988.

[5] Luis Paulo Peixoto dos Santos. Load Distribution: a Survey. Technical report, Universidade do Minho, 1996.

[6] W. Keith Edwards. *Core Jini*. Prentice Hall, 1999.

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

[8] Jeroen Heijmans and Melissa Tjiong. In-home Network: Load Balancing — Detailed Design Document. Technical report, Philips TASS, 2002.

[9] Jeroen Heijmans and Melissa Tjiong. In-home Network: Load Balancing — Investigation Report. Technical report, Philips TASS, 2002.

[10] Jeroen Heijmans and Melissa Tjiong. In-home Network: Load Balancing — Software Design Document. Technical report, Philips TASS, 2002.

[11] Jeroen Heijmans and Melissa Tjiong. In-home Network: Load Balancing — Software Requirements Specification. Technical report, Philips TASS, 2002.

[12] Dragos Manolache and Marton Zelina. In-Home Network Simulation Framework II. Technical report, Stan Ackermans Institute, Technische Universiteit Eindhoven, 2001.

[13] Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. In *Proceedings of Third International Conference on Coordination Models and Languages*, 1999.

[14] Niranjan Shivarati, Philip Krueger, and Mukesh Singhal. Load Distributing for Locally Distributed Systems. *IEEE Computer*, 25(12), 1992.

[15] Melissa Tjiong. Load Sharing in a Jini In-Home Network. Master's thesis, Universiteit Twente, 2002.

[16] Douglas West. *Introduction to Graph Theory.* Prentice Hall, 1996.