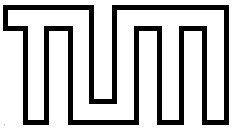


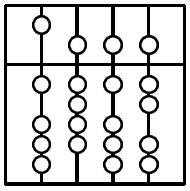
INSTITUT FÜR INFORMATIK

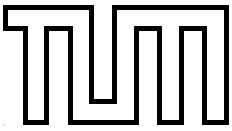
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Concept and Implementation of an Emulation Environment for Distributed Service Discovery

Bearbeiter:	Andriy Golovko
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering, Prof. Dr. Claudia Linnhoff-Popien
Betreuer:	Mike Heidrich (Fraunhofer ESK) Michael Krause Michael Schiffers





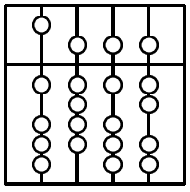
INSTITUT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Concept and Implementation of an Emulation Environment for Distributed Service Discovery

Bearbeiter:	Andriy Golovko
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering, Prof. Dr. Claudia Linnhoff-Popien
Betreuer:	Mike Heidrich (Fraunhofer ESK) Michael Krause Michael Schiffers
Abgabetermin:	15. August 2005



Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. August 2005

. . . . . .

(Unterschrift des Kandidaten)

Abstract

The number of services offered by different devices to the end users has significantly increased. There are different approaches and ways in which the services can be brought to the end user. Services used with as minimal overhead as possible are usually preferred. There are several aspects that must be considered in order to achieve this. The way in which services are discovered by the users is one of these aspects.

Services operate in various environments: home environment, office environment etc. These usually put some specific requirements to the service discovery. For example, an environment with many mobile devices may emphasise power saving characteristics of the service discovery approach as a tradeoff to the service discovery speed. The question which therefore arises is whether it is possible to meet the requirement of an environment in advance.

There are two extremes among the approches used for investigating such kind of questions. A modeling of the service discovery protocol characteristics followed by a simulation is used most often. Another extreme is the practical evaluation of the service discovery protocol in usually large scale environments with hundreds services and terminal devices. This requires high implementation and integration effort. It also worth mentioning that they are often intractable.

The objective of the current work is to design and implement an emulation framework which fills the gap between theoretical simulations and physical implementations. It reuses existing service discovery prtocol implementations and imitates services' and users' behaviour. The framework proposed in this thesis extends a real network environment through virtual emulated services. Thereby the performance of service discovery protocols can be explored with real terminal devices. It may be user-defined by the size of the enhanced service environment. Furthermore, the framework delivers statistical information about the service discovery protocol under consideration.

The framework is also deployed to evaluate the design and implementation of the service environments before their complete practical realization of the environment.

Contents

Chapter 1

Introduction

1.1 Motivation

In a service-oriented computing environment functions and applications are also referred to as services. Services are usually provided by devices and used by people. Computing environment can often consist of many different services. A typical example is a lecture room of the Technical University of Munich. A lecturer usually turns on/off the light in the room, regulates the volume of the microphone, and opens/closes the drapes. Some rooms are equipped with a touch screen based panel which proposes intuitive interface to control most of the available devices. Devices like the beamer and the movable blackboards must be controlled separately. Most of the services offered by different devices are used during a lecture. The associated overhead very often increases with the number of devices.

The situation in an office environment can even be worse. Even today it is quite common that users do not have permanent work places. They must accomplish various tasks like giving presentations or printing documents at different locations. There is also a strong tendency to increase the number of services available per user. For example, the number of different devices in companies has increased significantly in the last few years. Therefore, finding the right services in an office environment is more crucial than in other environments.

There are different ways in which services can be discovered and used by the end users. Some of them require significant configuration efforts. For example, to prepare the proper work place for an external freelancer you would have to go through the secretary. However, would not it be nice to have the work place available right away? Would not it be convenient, if your **Personal Digital Assistant** (**PDA**) could find the necessary services in a new room without you having to undertake additional configurations?

The following example illustrates a ”better” way of finding the right services (based on an example from [SLP]):

> ***Traditional scenario:***
> *Newbie: ”Hey Stan, the setup program is asking me for the name of our printer. What should I type in?”*
> *Stan: ”Which printer do you want?”*
> *Newbie: ”The big one by the copier.”*
> *Stan: ”I’ve heard it does not work well with postscript applications. You’ll have to use the one down the hall.”*
> *Newbie: ”Ok. What should I type in.”*
> *Stan: ”Actually, I do not know; I use the one by the copier. You’ll probably have to call the IS help desk.”*

Newbie: <groan> ...

Desired scenario:
A setup program displays the description (including location) of the printers that work with postscript. The user selects one that is close to his office. The service returns all necessary addressing information directly to his device setup application.

Both scenarios above (the University and "printer"-scenario) imply that the *discovery* of services plays a crucial role in bringing them service to the end user. Therefore, we need an adequate service discovery protocol. A number of such protocols have already been designed and developed. The most famous ones are part of technologies like Java Intelligent Network Infrastructure (Jini), Universal Plug-and-Play (UPnP) technology, Universal Description, Discovery and Integration (UDDI) technology, Service Location Protocol and some others. However, there are still open questions.

Assume, that we have an environment where the services are located on devices with limited energy resources. As a result, the environment put specific requirements on the power saving characterisics of the service discovery protocol. Furhter, we assume that a specific service discovery protocol implementation exists. Even if the protocol implementation works correctly, it is not clear whether it meets the requirements of the *environment*. A practical evaluation of the service discovery protocol implies running it in an arbitrarily large environment with arbitrarily many services and terminal devices. The environment is also supposed to have specific characteristics like a network topology, characteristics of network links etc. This is usually associated with many implementation and integration efforts. On the other hand, modeling followed by an eventual simulation which determines the protocol's suitability can be quite complex too.

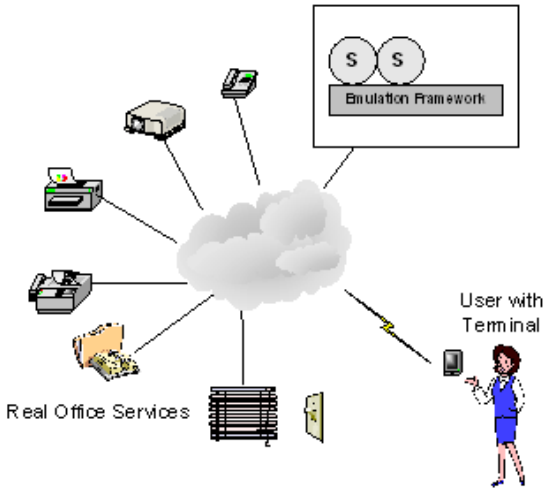


Figure 1.1: The real environment can be extended by an emulated one, which devices and services are discovered in the same way as the real ones.

In this thesis an emulation framework (reffered to as *ESKEm*[1]) is conceptualized and implemented. This must support the researchers in their investigation of the questions mentioned above. The framework fills the gap between theoretical simulations and the physical implementations. It is not our primary goal to develop a new protocol, but rather to explore, whether a specific protocol meets the requirements of a specific environment. Nevertheless, the protocol exploration can still result in an improvement of some concrete service discovery protocols (at least for the specific environment).

Our framework imitates an environment, where existing service discovery protocols are being explored. There are several ideas behind our approach. First, we use existing service discovery protocol implementations in the *ESKEm*. Second, the *ESKEm* can also be seen as an *extention* to an existing environment. As a result, the emulated services are discoverable by the real clients (Figure 1.1). In Chapter 2.3 use cases provide more detailed overview of the *ESKEm* functionality (see Section 2.1). The use cases are used to derive the requirements for the emulation framework (Section 2.3).

[1]"ESK" stands for Fraunhofer Einrichtung für Systeme der Kommunikationstechnik, www.esk.fraunhofer.de; "Em" stands for emulation

1.2 Structure of the Current Work

The thesis is organized as follow. Chapter 2.3 describes the Use Cases which explain the *ESKEm* functionality. They are used to derive the requirements for the *ESKEm* in Section 2.3.

We consider different approaches to the service discovery in Chapter 3. This includes an explanation of the technical background, required for understanding the service discovery operation. We cover in detail some selected service discovery approaches. These are compared among each other. On one hand, the similarities among the service discovery protocols result in the Generic Model for the service discovery (Section 3.5.1). On the other hand, Section 3.5.2 deals with the differences among the service discovery approaches.

Supported by our understanding of the service discovery mechanisms, we first make suggestions about design *ESKEm* in Chapter 4. We analyze related work, namely approaches for reproduction of network environments, for measurement of service discovery performance and for visualization.

We continue with the *ESKEm* design in Chapter 5 and consider different alternatives for the design of particular *ESKEm* modules. We provide detailed description of functionality of modules and the interfaces between them.

In Chapter 6 we introduce the results of the *ESKEm* implementation and validation. We integrate selected service disocvery into the *ESKEm* for validation purposes.

Chapter 7 provides a summury of the thesis and suggestions for future work.

Chapter 2

Emulation Framework Requirements

In Section 1.1 we formulated the task of the current thesis as the creation of the emulation framework *ESKEm*, which could be used for exploration of existing service discovery protocols in a specific user defined environment. We would like to fill the gap between theoretical simulations and physical implementations with the *ESKEm* emulation framework.

We introduce two scenarios that describe our vision of the *ESK* [1] and the way our framework could possibly be used. The specified scenarios are used to derive the requirements to the *ESKEm* in Section 2.3.

A specific service discovery technology is usually designed to operate in some specific environment. In this chapter we also characterize the *office environment* which is of primary concern as a result of the explicit *ESK* interest in the exploration of such types of environment. Thus, an office environment is one of the environments that could be reproduced by the *ESKEm*.

2.1 Use Cases

Providing two goal-oriented Use Cases for the *ESKEm* gives us two points of view regarding our emulation framework. The first Use Case, called ”Exploration”, describes how the emulation framework can be used for research purposes, e.g., how the Researcher can take a look at the service discovery protocol operation in a given environment. The second Use Case, called ”Demonstration”, proposes the way the emulation framework can be used to demonstrate the service discovery operation in an environment with specific characteristics.

2.1.1 Use cases 1: Exploration

The *Researcher* is the main actor in this use case (see Figure 2.1). He selects the service discovery protocol and integrates it into the *ESKEm* for further exploration [2].

The Researcher specifies the emulation scenario in the script files. This includes different commands that control the life cycle of the services and clients. An office environment usually consists of different types of services.

The Researcher also defines the characteristics of the environment where the service discovery (SD) protocol operates. The service discovery in LAN/WLAN is of primary interest for the ESK. The Researcher specifies different characteristics of the network over which the SD protocols operate. Examples are network topology, network links characteristics, packets drop rate etc.

[1] Fraunhofer Einrichtung für Systeme der Kommunikationstechnik, www.esk.fraunhofer.de

[2] The way of the integration will be considered during the Design phase in Chapter 5

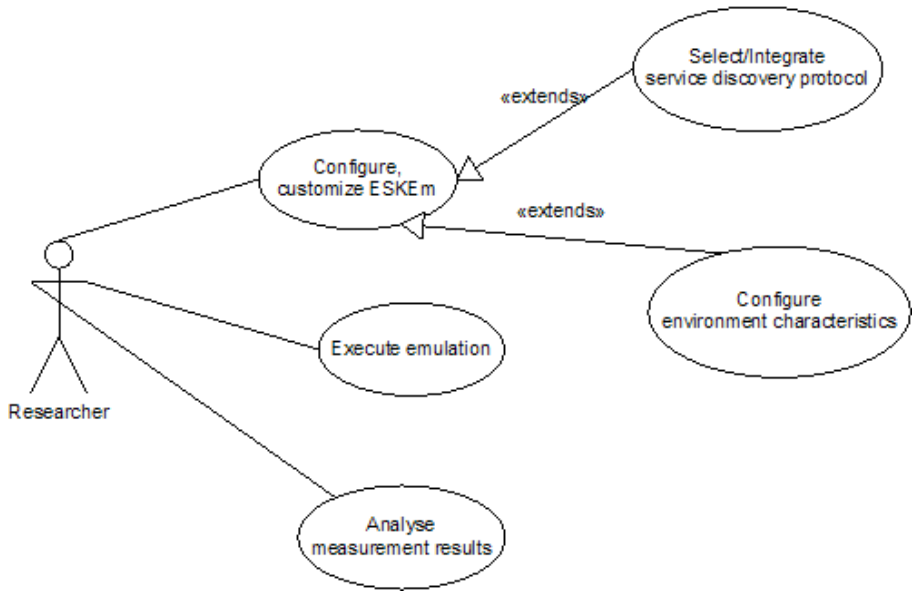


Figure 2.1: Use Cases 1: The ESKEm can be used by the Researcher to explore the service discovery protocol in the specific environment

The Researcher starts the emulation. He explores different metrics measured by the *ESKEm* measurement infrastructure after the emulation is finished. It should be possible to introduce new metrics into the emulation framework. The researcher is supposed to have appropriate programming skills in order to extend our framework for his own needs.

Some service discovery specifications, like UPnP and their implementations provide the means for the service invocation. We are not interested in the service invocation and it is hence enough when the emulated services are only discovered and not invoked.

2.1.2 Use cases 2: Demonstration

This scenario is intended to describe the case in which a usual user can profit from our emulation framework. A usual user is someone with a general understanding of the service discovery operation and without any programming skills. Such a user is going to be called simply the ”User”. The User can execute the emulation and observe the service discovery operation by using the **Graphical User Interface** (**GUI**) provided by the *ESKEm* (see Figure 2.2).

The GUI provided for the User should be intuitive. According to the ESK preferences, the User is confronted with a graph-based view of the *ESKEm* environment, where the services and the clients are represented as nodes. The links between them and the network infrastructure elements like routers are represented as edges. The operation of the service discovery protocol should be clearly visible on the given graph. For example, the message from A to B could be represented as an arrow over along the edges participating in the message transfer.

Additionally, the User can start an external device (e.g. PDA) with a client located on it. With the help of an appropriate configuration of the emulation framework the User should see the list of services that are emulated and were found as a result of the service discovery operation (see Figure 1.1). Both real and emulated clients and services must ”speak” the same service discovery protocol. The external clients are visualized on the GUI too.

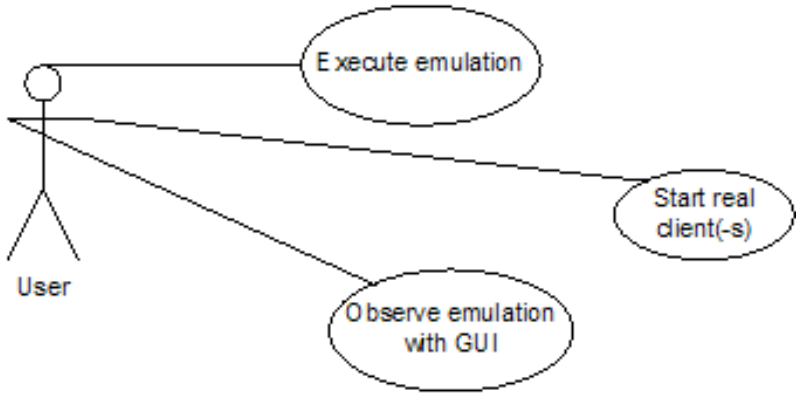


Figure 2.2: Use Cases 2: The ESKEm can also be used by the User to observe the service discovery protocol operation in the specific environment

2.2 Office Environment

The service discovery alone does not make much sense. It always exists in some kind of environment. The environment characteristics play a crucial role in choosing appropriate service discovery middleware in general and a service discovery protocol in particular. For instance, the service discovery latency[3] should be as short as possible in an environment characterized through high level of users or services mobility. As a result, we may be interested in the characteristics of the environment which is to be recreated with the *ESKEm.*

The office environment is of main interest for the ESK. That is why we would like to concentrate on the characteristics of this type of environment. Human environments are usually goal-oriented. This means that the difference in tasks which are to be accomplished in the specific environments, is one of the reasons why environment characteristics differ. For example, the typical home environment offer about 10 services which are not used intensively. On the other hand, the office environment of a company makes available about 100 services that are used very often. But the task of the specific environment characterization is still a vague goal. It is hard to define a typical office environment. That is why we decided to take just the ESK by itself as a prototype and as a live example of the environment that we would like to recreate in the *ESKEm* (see Table 2.1 and related figure).

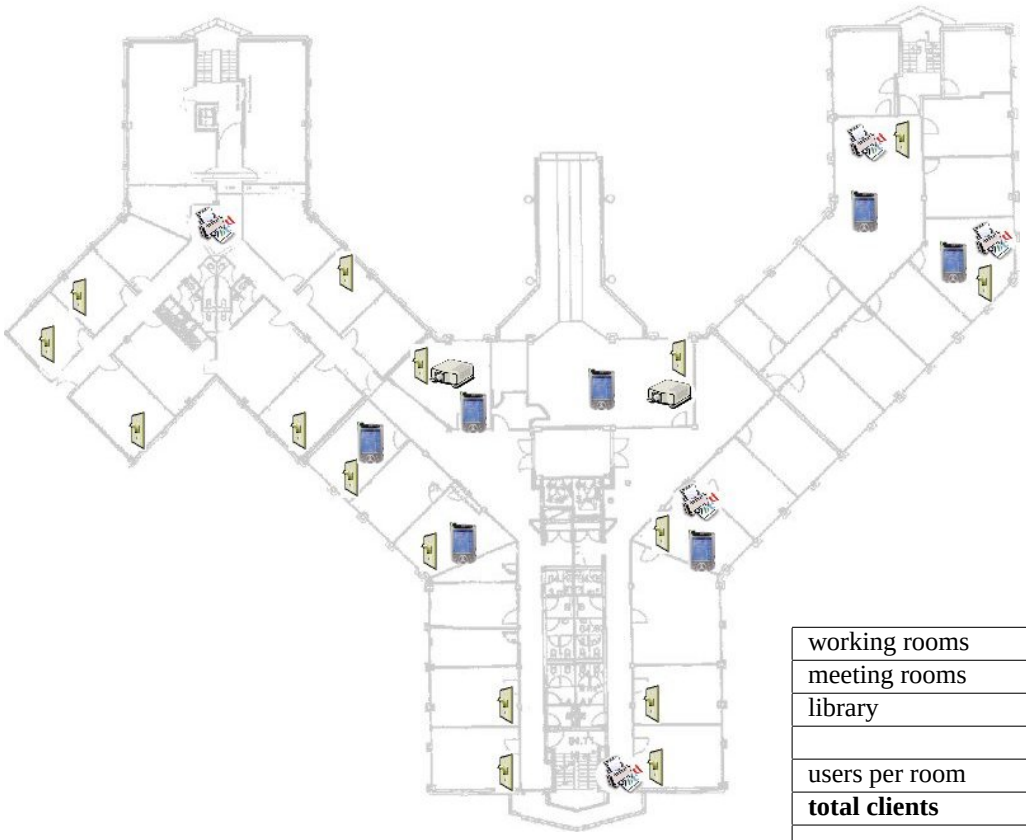
The number of services shown in Table 2.1 was calculated approximately based on the number of rooms and devices in the *ESK*. Very often a device contains several services and the users have several clients. Therefore, the number of the real services and clients can be a multiple of the presented numbers. Nevertheless our example should give a feeling about our vision of a ”typical” office environment and its size.

We also assume that the computational environment on the figure next to Table 2.1 is built mainly over the LAN or WLAN. The availability of the Internet connection is not essential for the service discovery in our case. We are mainly interested in the discovery of services in a single company in a single building. The available services are mainly static. The mobility of users may vary. We are not interested in the exact simulation of users' behavior.

2.3 Requirements to the Emulation Framework

We use the previously described Use Cases to formulate the requirements to the *ESKEm.* The requirements are divided into Functional, Non-Functional and Pseudo-Requirements. The Pseudo-Requirements can be

[3]The time between a client sends a service discovery request to the network and the time of response from a discovered service.



working rooms	26
meeting rooms	3
library	1
users per room	2
total clients	**52**
printers	10
beamers	3
light switchers	36
telephones	56
total services	**105**

Table 2.1: The ground plan of the ESK with a sample of the available devices. The number of services and users is an approximate calculation based on the number of different rooms available.

viewed as a ”wish”, expressed by the ESK. We provide the background for these ”wishes” in the subsequent chapters. The pseudo-requirements either relax the functional requirements (like the Requirement C.4) or constrain them (like the Requirement B.7). Before we start, we would like to give a more abstract characterization of the environments which are of our interest and which should be reproducible by the *ESKEm* for the service discovery operation exploration:

A.1 We are interested in the service discovery in heterogeneous environments

A.2 The considered environments usually consist of a large number of services/clients. There is a tendency for this numbers to increase. This observation results in the Requirement B.8

Following Functional and Non-Functional Requirements were derived from the Use Cases, defined earlier in this chapter:

B.1 **A real implementation of the service discovery protocol.** The requirement means that we want to explore real service discovery implementations with the *ESKEm.* The emulated services and clients should be able to use it for the purpose of announcement/discovery of services respectively.

B.2 **Metrics measurement infrastructure.** The feedback about the service discovery implementation performance is received in form of performance measurement and calculation of the associated met-

rics. The set of interesting for the researcher metrics is specific for each experiment. As a consequence, we are not interested in measuring some specific metrics. Instead, the *ESKEm* should provide an infrastructure which must be flexible enough to define and measure new metrics.

B.3 **The Reproducibility.** We mean the reproducibility of the emulation scenarios. This may be especially important for a comparison of the same scenarios under operation of different service discovery protocols. The emulation scenarios should take into account the specifics of the service discovery mechanisms.

B.4 **Visualization.** The service discovery operation should be ”appropriately” (graphically) visualized. The visualization should support the user of the framework in understanding complex communication patterns between services and clients in the context of the service discovery. There are different ways in which the visualization can be designed in our case. There are a couple of preliminary concerns about the visualization in the *ESKEm*:

(a) **The graph-based visualization**. A network with clients and services can be intuitively represented as a graph. We should consider the graph-based visualization of the service discovery protocol operation as the primary alternative for the visualization. We think, that the other forms of visualization are relevant too.

(b) **Separate deployment.** It seem to be reasonable to develop the visualization facility as a separate module because of the performance concerns. Graphical visualization is usually resource consuming. In other words, the visualization module should be deployed on the other host as the set of modules, which are responsible namely for the emulation/simulation of the service discovery protocol.

(c) **Real-time.** The visualization in real time should be supported. This also applies to the services and clients which are ”real” (see Requirement B.6).

B.5 **Heterogeneity.** As we have seen from the office environment above, the service discovery protocols are supposed to operate over the networks with different characteristics. The services/clients are very often located on the devices with different network protocols stacks. The mentioned differences are *one of the* most important sources for the heterogeneity of an environment in which a service discovery protocol operates. Users’ behavior would be another such source. And we are supposed to reproduce the heterogeneity of the networking environment (see also Requirement A.1).

B.6 **Emulation of Services.** The services emulated by the *ESKEm* should be discoverable by the real clients. This ”emulation” aspect is one of the reasons, why our framework is called an *emulation* framework. In other words, the real clients should not be able to distinguish between real services and the services imitated by the *ESKEm* framework. It should be noted that the requirement applies to the service discovery only. This means that the result of the service invocation on the emulated service (e.g., invocation of the *print* function on an emulated *printing* service) is undefined. The Requirement B.3 ensures that an *initial* emulated environment is always the same for the real clients. The operation of the real clients in the emulated environment is likely to be in contradiction to the reproducibility of the emulation scenarios.

B.7 **Validation.** Our expectations of the *ESKEm* should be validated by the exploration of the *UPnP Cybergarage*[4]. The protocol operation in the environment should be demonstrated using relevant examples.

B.8 **Scalability.** The emulation framework should be designed with the *scalability* in the number of emulated services in mind. The scalability is usually achieved by enabling the distributed mode of the framework operation. This meanemulation of services on multiple hosts.

The Pseudo-Requirements were formulated by the ESK as following. Some of them were already mentioned indirectly but we would like to formulate them explicitly:

[4]This was first introduced in Chapter 3

C.1 **Low cost of the solution.** The developed emulation framework should be available at affordable costs.

C.2 **Office Environment.** The Service Discovery in the office environment is of primary interest.

C.3 **Java-based.** The platform independence of the *ESKEm* should be achieved with Java.

C.4 **LAN.** Our primary interest lies in the exploration of the service discovery protocols over LAN-based environments. Despite Requirement B.5, it was allowed by the *Fraunhoffer ESK* to limit the *ESKEm* to the exploration of the service discovery in the LANs. Nevertheless, the *ESKEm* provides the means for introduction of the other types of networks. Thus, this requirement relaxes the Requirement B.5. The discovery over a network with different topologies and network links characteristics should be possible.

Chapter 3

Service Discovery

We has already mentioned in Chapter 1 that an efficient discovery of services is an important task in the service-oriented computing environment. It is obvious that the services should be discovered before they can be used. The services exist in an environment, which in turn has very often some specific characteristics. For example, a typical home environment differs from an office environment by the type and number of services etc. The selection of the service discovery protocol should be based on the environment's characteristics, in which the service discovery will operate. We have already given our view onto the *office environment*. Now we would like to consider different **Service Discovery** (**SD**) approaches in more details. Among other things we are interested in the answers to questions like What are the different SD protocols? What are the design goals of different SD protocols? What are the characteristics of the environment, for which the specific SD protocol was designed? Which network transport do SD protocols usually use? We need the understanding of the service discovery as a component, which we are going to explored with the *ESKEm.*

Based on the office environment characteristics, we will select several SD approaches (UPnP, SLP, Salutation, Jini), which are either used or could be used in this type of environment. The comparison of the selected SD approaches will be done for two reasons. First, we will derive the Generic Model of the service discovery based on the similarities between selected approaches. Second, we will interpret the differences between selected SD approaches as the parameters, which influence the service discovery performance. We may wish to explore the influence of these various parameters on the service discovery performance in a given environment. The Generic Model will be used in Chapter 5 dedicated to the design of the *ESKEm.*

Both specification and the reference implementation of the two mentioned in previous paragraph SD protocols (UPnP, SLP) will be considered in more details. There are several reasons for doing this. First, the ESK has explicitly expressed interest in the exploration of the UPnP and SLP technologies in the office environment with the *ESKEm.* Second, we wish to get deeper insight into the service discovery middleware implementations for better understanding of the technical problems, which may arise when plugging them into the *ESKEm.* Third, we wish to consider the *UPnP Cybergarage* reference implementation, which will be used later to validate our approach in accordance with the Requirements B.7 and B.7.

But we would like to provide the adequate Technical Background first.

3.1 Technical Background

The service discovery protocols operate at the application level in terms of the OSI model. They use the protocols from the underlying network layers for the messages transfer. The keyword ”message” raises two important question: What kind of messages are transferred during the service discovery operation and

How are they transferred? The answer to the first question is usually specific for each service discovery protocols. A short answer to the second question we would like to provide in the current section.

The way in which the messages are sent during service discovery can be characterized among other things as following:

Unicasting is one-to-one communication from a source to a destination on the network.

Broadcasting is one-to-everyone communication from a source to the rest of hosts on the network. Broadcasting wastes network bandwidth by flooding the network when only a small number of users need the information.

Multicasting - the host sends out only one copy of the information, and only the destination hosts that need the information receive it. It is specified in

Anycasting is a network addressing and routing scheme whereby data is routed to the ”nearest” or ”best” destination as viewed by the routing topology.

Multicasting We will mostly refer to unicasting and multicasting. The multicasting require some additional explanations.

The multicasting is supported by the Internet Group Management Protocol (IGMP), which is referenced in RFC 1112 [Deer]. In the addressing scheme in the IP protocol, class A, B, and C addresses identify a host in a TCP/IP network. A class D address represents an IP multicast group in which many hosts participate in the same IP multicasting application. The first four bits of the 32-bit class D addresses are 1110, and the remaining 24 bits can range from all 0s to all 1s. Therefore, IP multicast addresses can range from 224.0.0.0 to 239.255.255. 255. The protocol reserves address 224.0.0.0 for operating systems, and the protocol cannot assign that address to applications. For example, the protocol permanently assigns 224.0.0.1 to the all-hosts group, which includes all computers and routers participating in IP multicasting in a local network.

UPnP uses the address 239.255.255.250 for multicasting.

There are three levels, at which multicasting can be supported by OS:

Level 0 has no support for multicasting

Level 1 supports sending but not receiving multicast IP data

Level 2 fully supports sending and receiving multicast IP data

Both MS Windows and Linux support Level 2 of multicasting.

For obvious reasons, network routers filter almost all broadcast traffic. This means that broadcast that is generated on one subnet will not be forwarded, or ”route” to any of the other subnets connected to the router (from the router’s perspective a subnet is all machines connected to one of its ports).

Multicast messages, on the other hand, are forwarded by routers. Multicast traffic from a given group is forwarded by routers to all subnets that have at least one machine that is interested in receiving the multicast for that group. A non-multicasting routers can multicast into own network, but it does not forward the messages further to the other routers.

HTTPU and HTTPMU Both HTTPMU and HTTPU were designed to support UPnP and are not W3C [W3C] standards. HTTPU (and HTTPMU) are variants of HTTP defined to deliver messages on top of UDP/IP instead of TCP/IP.

HTTPMU is sending of packets with HTTP content over multicast UDP to a ”group”. HTTPMU is expected to use the syntax of HTTP without changing its semantics. HTTPMU allows the possibility of *new* request types (such as NOTIFY in UPnP) beside of the standard HTTP requests of GET, POST, HEAD etc.

There is no response to these multicast requests (as in the case of usual HTTP protocol request) since the message may be received by any number of clients, and this is not a request-response situation.

HTTPU sends a single datagram over UDP to a host. The recipient may send a reply back to the host and port that sent the datagram. Again, it is not expected that a standard HTTP requests is sent, only that the syntax should be obeyed.

Routing Routing specifies the algorithm, by which the messages from a sender A can find a way to the receiver B over the network. The message is said to pass the routers, or hops. A hop count or so called TTL (Time To Live) determines how many gateways or IP-routers the packet may pass in order to reach the destination. If the hop count reaches zero (every hop decreases the value) the current gateway replies with an error and its IP address as source address.

To ping the hosts on local network (even if it is localhost), the TTL should be set at least to 1 (”ping -t 0 localhost” breaks with the error).

DHCP The Dynamic Host Configuration Protocol (DHCP) is an Internet protocol for automating the configuration of computers that use TCP/IP. DHCP can be used to automatically assign IP addresses or other configuration parameters like the subnet mask and default router, and to provide other configuration information such as the addresses for printer, time and news servers [DHCP].

3.2 Service Discovery definition

We start with a simple generic example before considering service discovery specifications in more details. The example is based on the scenario described at [ZhMuNi 02].

> Bob visits a University. He turns on his PDA. The PDA is a *device*, which in turn contains the *client* used by Bob. Bob searches the Web and finds an online map. He wants to print the map. He tries to find a printer for printing the map with the client. We define this process as the *service discovery*. The printer is a *device*, which provides the printing *service*. After available printers are being found and one of them is selected Bob may print the map, or we could also say that Bob may *invoke* the printing service.

A service is usually defined as *”some computing resource used by users, user programs, or other services”* (for example, in [ZhMuNi 02]). Any function proposed by the devices in a computing environment is said to be a ”service” like printing service, storage service etc. A device usually provides multiple services (for example, a fax device could provide also the ”printing service”, ”copy service” etc. beyond the ”faxing service”). The service usually has a name, the list of searchable attributes or an associated service description file and associated user privileges. A list of service attributes usually varies among different service discovery protocols in respect to the syntax, semantics and the level of details.

The service are called to be used by the *service clients* or simply by the *clients*. In the scenario with Bob, the PDA played the role of the client. Bob can also be called the *service user*. The clients act as the proxy entities, which mediate user’s interaction with services. We assume, that both services and clients are usually located on the *devices*. The services are also said to be *provided by the device*.

The main goal of the service discovery process is to guarantee that a service client discover necessary service(-s) (if any available). Some authors (like [Gold 02]) split the service discovery into two complementary activities: service *discovery* (initiated by the client) and the service *advertisement* (done by the devices or by the services itself). Both activities are supported in the service discovery protocols which we are going to consider (an example of the SD protocol without the service advertisement would be the Bluetooth Simple Service Discovery Protocol, where only the service discovery is supported). We will refer to both the service discovery and the advertisement by the term *service discovery* as long as nothing else is mentioned.

We will often use the terms like"technology", "framework" etc. in the connection with the term of "service discovery", which should be understood as a set of means, which are necessary to guarantee connectivity between clients and services (we mean under *connectivity* the ability of the clients and services to link, or to communicate with each other). The service discovery *protocol* is very often only a part (though important) of this set. The *protocol* is defined in [HeAbNe 99] as following:

> The protocols are precise specifications (in respect to the syntax, functions and semantics) of steps and rules for exchange of the information between two or more parties on the same functional level in a communication system.

The service discovery technologies are usually described by the *specification* (or *reference*), which is usually an informal and abstract description of the service discovery approach. On the other hand, the reference implementation is an executable code. We mean primarily the *specification* when using the terms like *approach*, *technology* or *framework*, whereas the reference implementation is meant when using the term *middleware*. For example, the *UPnP* is a specification, which implies existence of different reference implementations. In general, the middleware is defined as software that is used to tie an application to a network, thus the "middle" terminology. In the context of our terminology, it mediates between the services or clients and the network transport layer (see Figure 3.1).

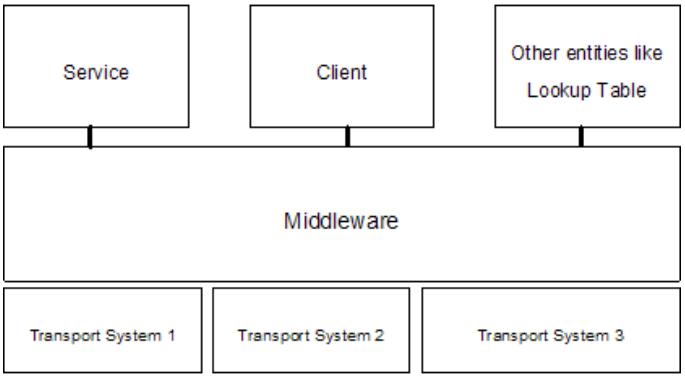


Figure 3.1: The middleware is defined as a software that is used to tie an application to a network, thus the "middle" terminology

Why do we require service discovery specification and a resulting service discovery middleware? The service discovery middleware encapsulates the tasks, which are common for different services and clients and are associated with the discovery or advertisement of the services. A new client should not know the service directly, he just should "speak" the appropriate protocol. The middleware functionality is accessed by the clients and services through the set of interfaces. On the other hand, the middleware also uses the interfaces provided by the lower network layers for communication over the network. We call this both types of interfaces the *middleware entry points*. They connect a service discovery middleware to the surrounding computational environment. This fact motivates us to investigate the middleware entry points precisely in order to plug the service discovery middleware of our interest into the environment, provided by the *ESKEm.*

3.3 SD protocols popularity

There are a number of the service discovery protocols developed till now. Very often they are developed with different goals in mind, and for the specific environment. For example, the GSD protocol (Table 3.4) primary goal was to minimize power consumption during the service discovery. Which service discovery protocols should we consider? On one side, the UPnP and SLP are the technologies, which are interesting for ESK (see the Requirement B.7). And we should obviously analyze them. On the other hand, we would not like to design the *ESKEm* specifically for integration of this two protocols and that is the reason why we would be also interested in the overview over the other service discovery approaches.

Which protocols could be considered as being popular? Our approach to the question is rather simple. First, we base our SD protocols selection on the [Bett 00], which provides the analysis of the SD technologies popularity expressed by the number of companies, involved in the development of the particular

SD technology (see Figure 3.2). Our selection criterion here is pragmatic: why not to concentrate on the protocols, which seems to be accepted by the industry (”follow-the-money” principle).

Second, we have analyzed the frequencies, with which different service discovery protocols are referenced in the scientific papers (see Table 3.1). This was accomplished by keyword search over on-line search interfaces such as [Google Scholar] (searches among other things also in the ACM Digital Library [ACM]) and [Citeseer]. In the cases, where the search results would provide too many matches because of the multiple meaning of the SD protocol abbreviation, we used either full name of the protocol. For example, we used ”service location protocol” to refer to SLP or added additional search words to specify the context of our search. Thus we used ”salutation service” to search in the [Google Scholar] instead of using ”salutation”[1]). Despite of the differences in the searched sources and search algorithm, the correlation between the columns is about 0.97 (0 - no correlation; 1 - the maximally correlation score). So the frequency of references can be considered as consistant at least among this two databases.

We use the frequencies in Table 3.1 as an index of relative popularity of the service discovery technologies . Some of the presented technologies will be covered in more depth in the rest of this chapter. Not all frequently referenced technologies from the next section were included into the Table 3.1. For example, the UDDI (see the next section) technology is the most often referenced technology after Jini (referenced in 7 260 documents available to [Google Scholar]). But we excluded it (and also other technologies like *Jini*), because it was not required by the *Fraunhofer ESK* to consider this technology.

Technology	Keywords	Citeseer [2]	Google/Scholar [3]
Jini	”jini”	654	11 200
UDDI	”uddi”	428	7 260
UPnP	”upnp”	106	1 480
Salutation	”salutation”/”salutation service”[4]	80	1 400
SLP	”service location protocol”	19	1 090

Table 3.1: The popularity of different Service Discovery technologies by references in the research papers: June, 2005

[2]search for exact pairing of words (order is not important)
[3]The search option ”with all of the words” was used
[4]in Citeseer and Google(Schoolar) respectively

We analyzed all four protocols from Table 3.1 in respect to their similarities and differences (Section 3.5).

3.4 Service Discovery Protocol Examples

An overview over different service discovery protocols was motivated by several interrelated reasons. First, we are interested in the architecture of different service discovery middleware, especially in the middleware entry points, which connect the service discovery protocol with the surrounding environment. Second, we would like to compare those of the service discovery protocols, which would be interesting for the office environment as defined in Section 2.2. Third, we are also interested in the service discovery design characteristics, on which the service discovery performance depends. We may wish to control them in the *ESKEm* to explore their influence on the protocols’ performance.

In this section we will also give detailed insight into the UPnP and SLP service discovery protocols because of the Requirement B.7. We also consider their reference implementations to understand better, how they can be plugged into the *ESKEm.*

[1]the [Citeseer] searches for exact pair of words, so we searched for the word ”salutation” without putting it into the context. We did not check manually, whether the ”salutation” was really mentioned in the service discovery context.

Abbreviation	Full Name	Origins	Specific Features
UPnP	Universal Plug-and-Play	Microsoft	The UPnP is an industry initiative designed to enable simple and robust connectivity among stand-alone devices and PCs from many different vendors.
SLP	Service Location Protocol	IETF	Uses *scope* for logical grouping of services. Services are pointed with a URL and are described with the list of attributes
Jini		Sun Microsystems	The executable code for service invocation is downloadable from the central lookup table
Salutation	-	the Salutation Consortium, Inc.	Describes an API, a Protocol and optional data pipes and job control to assist in *discovery* and *use* of devices and services. The architecture is based on a model called the Salutation Manager.
OSGi	The Open Services Gateway Initiative	OSGi Alliance	An open standard for connecting consumer and small-business appliances with commercial Internet services. The specification will provide a common foundation for ISPs, network operators, and equipment manufacturers to deliver a wide range of e-services via gateway servers running in the home or remote office.
UDDI	Universal Description, Discovery and Integration		The discovery of Web services both within and between enterprises
JXTA	”juxtapose”	Sun	JXTA technology is a set of open, generalized peer-to-peer protocols that allow any connected device (cell phone, to PDA, PC to server) on the network to communicate and collaborate.
Bluetooth SDP	Bluetooth Service Discovery Protocol	Bluetooth Special Interest Group	Only the discovery of services is supported (service advertisement is not possible). Based on the Bluetooth short-range, 1-Mb/s[a] wireless radio system (about 10 meters).
GSD	Group-based Service Discovery protocol		decrease average energy consumption during SD in a group of devices

Table 3.2: Overview over Service Discovery technologies referenced in research papers.

[a] 1-megabitpersecond

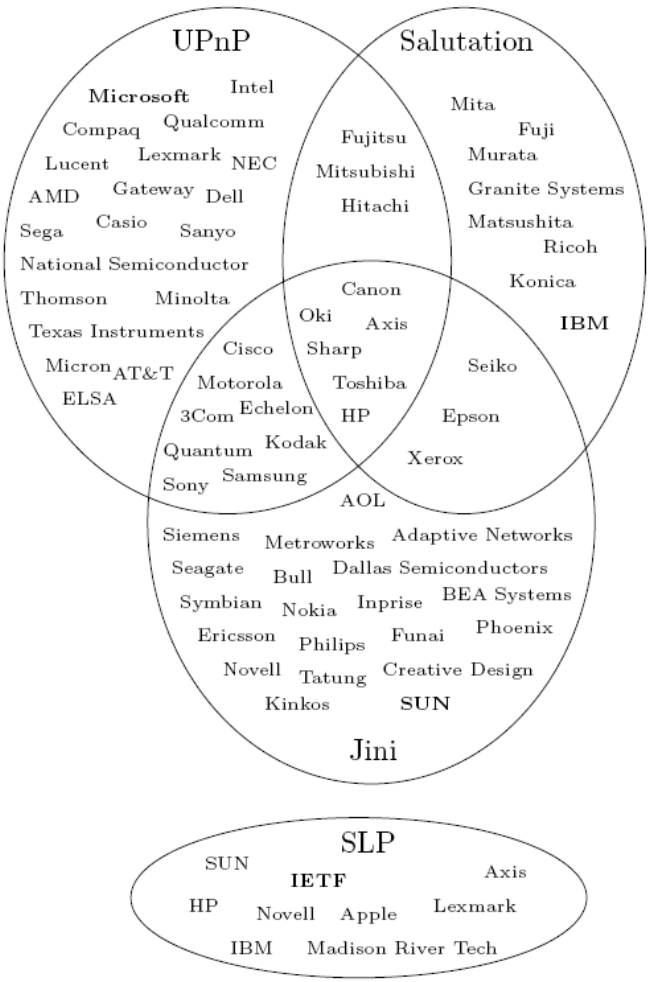


Figure 3.2: Companies involved in the development of Jini, Salutation, UPnP and SLP ([Bett 00])

Jini Jini technology [Jini] is an extension of the programming language Java which has been developed by the Sun Microsystems. It addresses the issue of how the devices connect with each other in order to form a simple ad hoc network (a Jini "community"), and how these devices provide services to other devices in such networks. The Jini architecture principles are similar to that of the SLP. Devices and applications register itself with a Jini network using a process called *Discovery* and *Join*. To join a Jini network, a device or application places itself into the *Lookup Table* on a lookup server, which is a database for all services on the network. Besides pointers to services, the Lookup Table in Jini can also store java-based program code like device drivers that help the user to access the service. The Jini java object returned to the client as the result of the search requests offers direct access to the service over the known to the client interface.

Salutation Salutation is another approach to service discovery [Salutation]. The Salutation architecture is being developed by an open industry consortium, called the Salutation consortium. The Salutation architecture consists of *Salutation Mangers (SLMs)* which provides the service brokers functionality. Services register themselves with an SLM, clients query the SLM when they need specific service. After discovering the desired service, clients are able to utilize the service through the SLM.

UDDI UDDI is short for Universal Description, Discovery and Integration. It can be defined as

> *... a standard for a platform-independent, open framework for describing service on the Internet.*
>
> *... a XML-based protocol that provide a distributed directory (or lookup service) that enables businesses to list themselves on the Internet and discover other services. Similar to a telephone number, businesses can list themselves by name, product, location, or the Web services they offer.*

The UDDI is a complementary to other technologies used for integration of web-based applications (or web services). For example, SOAP is used to transfer the data, WSDL is used for describing available service, and UDDI is used correspondingly for listing available services.

The mentioned concept of the ”web service” is the term for group of loosely related web-based components and resources that may be used by other Web applications (or services) over HTTP. We assume, that a ”usually” office environment services (like printing service) could be used only over HTTP too. So in principle, the UDDI is applicable for the purpose of service discovery in the office environment. Whereas actual strength of the UDDI is in discovery on the large (”whole world”) scale.

We will not be considering the UDDI deeper because of the Requirement B.7.

In the next two sections we will consider the UPnP and SLP service discovery in more details.

3.4.1 UPnP

The UPnP is the technology which enables connectivity among stand-alone device and PCs from many different vendors. It is developed by the different vendors organized into the UPnP Forum, headed by Microsoft [UPnP Forum].

The devices of the UPnP are build hierarchically (see Figure 3.3). In a compound device (for example, a VCR/TV combo), a client (called a *control point*)) can address the individual sub-devices (for example, a tuner) independently as soon as the *root device* was discovered. Virtual Web servers in the device act as an entry points for interacting and controlling of the device. Devices that do not speak UPnP directly are called bridged devices. A bridge maps between UPnP and a device-native protocols.

The UPnP specification describe device addressing, service advertisement and discovery, device control, propagation of events, presentation. Several protocols support these functions:

- AutoIP, a simple protocol that allows devices to dynamically claim IP addresses in the absence of a DHCP service;
- Simple service discovery protocol (SSDP), the UPnP mechanism for service discovery and advertisement;
- Simple object access protocol (SOAP), a protocol for remote procedure calls based on XML and HTTP that is used for device control after discovery; and
- Generic Event Notification Architecture (GENA), a UPnP subscription-based event notification service based on HTTP.

Each device must have a **Dynamic Host Configuration Protocol** (**DHCP**) client. If DHCP server is available on the sub network, the device must use the IP address assigned to it. If no DHCP server is available, the device must use Auto IP to get an address.

A **Simple Service Discovery Protocol** (**SSDP**) is used as a mechanism for service discovery and advertisement. Figure 3.4 shows the protocol stack for discovery search. SSDP is built upon HTTPU and HTTPMU and defines methods both for a control point to locate resources of interest on the network, and for devices to announce their availability on the network. The protocol operates as following:

Control Point *upon booting up*, sends an SSDP search request over HTTPMU to discover devices and services available on the network.

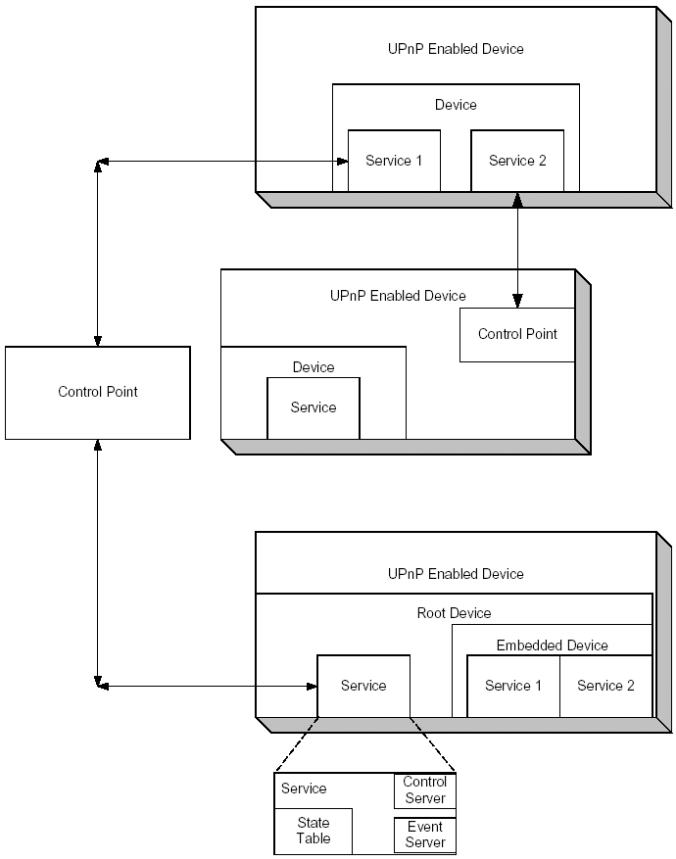


Figure 3.3: The UPnP's device model is hierarchical.

Device listen on the multicast port. *Upon receiving of a search request* device examines the search criteria to determine if they match. If they match, a unicast SSDP (over HTTPU) response is sent to the control point. This response contains URLs, each pointing to an XML *description document* that describes a service. Similarly, a device, *upon being plugged* into the network, will send out multiple SSDP presence announcements advertising the services it supports. SSDP also provides a way for a device and associated service(s) to gracefully **leave the network** (bye-bye notification) and includes cache timeouts to purge stale information for self healing.

Mentioned above XML description document contains among other things following information:

- *A presentation URL* allows entry to a device's root page, which provides a GUI for device control.
- *A control* URL is the entry point to the device's control server, which accepts device-specific commands to control the device.
- An *event subscription URL* can be used by clients to subscribe to the device's event service. The client provides an *event sink URL* in the subscription request. Significant state changes in the device result in a notification to he client's event sink URL.
- A *service control protocol definition* describes the protocol for interaction with the device.

SSDP is similar to the Internet Engineering Task Force's **Service Location Protocol** (**SLP**), but it lacks a query facility which would support a search for service by an attributes.

UPnP specification is programming language independent. We have considered and selected Java implementation of the specification [Konn 04] for validation purposes of our approach.

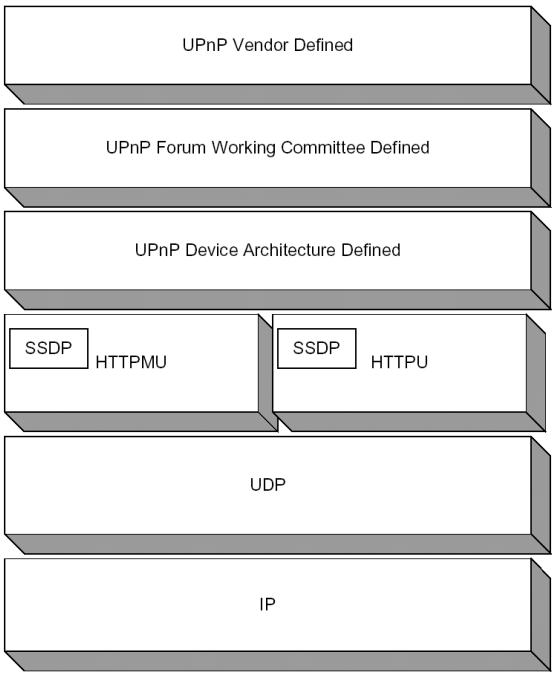


Figure 3.4: UPnP Protocol Stack for Discovery Search

UPnP Cybergarage: the Reference Implementation Example We have taken *UPnP Cybergarage* [Konn 04] reference implementation of the UPnP specification. It was selected in accordance with the Requirement B.7.

The most important classes, which participate on the service discovery, are introduced on the Class Diagram at Figure 3.6. To avoid drawing the Collaboration Diagram, we also introduced them on the class diagram. We have also shortened some dependencies for sake of the diagram simplicity. For example, instead of using the expression like ”A calls B calls C” we just say that ”A calls B”.

During the investigation of the reference implementation, we are mainly interested in the way how the middleware is connected to and interacts with the environment to accomplish service discovery tasks (see Figure 3.5). Or in the other words, we need to know, (a) how the middleware is used by the services and clients, which API is proposed to them; (b) how does the middleware depends on the transport system.

As can be seen on Figure 3.6, the `HTTPUSocket` and the `HTTPMUSocket` classes use the `DatagramSocket` and the `MulticastSocket` classes respectively to send the messages over the transport system, which is represented by the `java.net` **Java Foundation Classes** (**JFC**) library of the Java. The services and control points are already represented in the implementation and we may wish to extend them for customization.

We are also interested in the types of messages, which are sent in the process of the service discovery. As will be seen later sections, this can be required when measuring statistics of the service discovery operation.

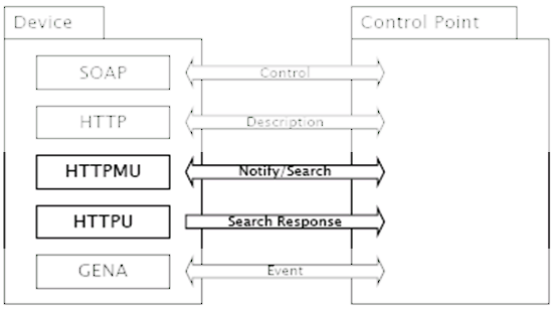


Figure 3.5: We are interested in the ”Notify/Search” and ”Search Response” communication channels during consideration of the service discovery in the UPnP ([Konn 04])

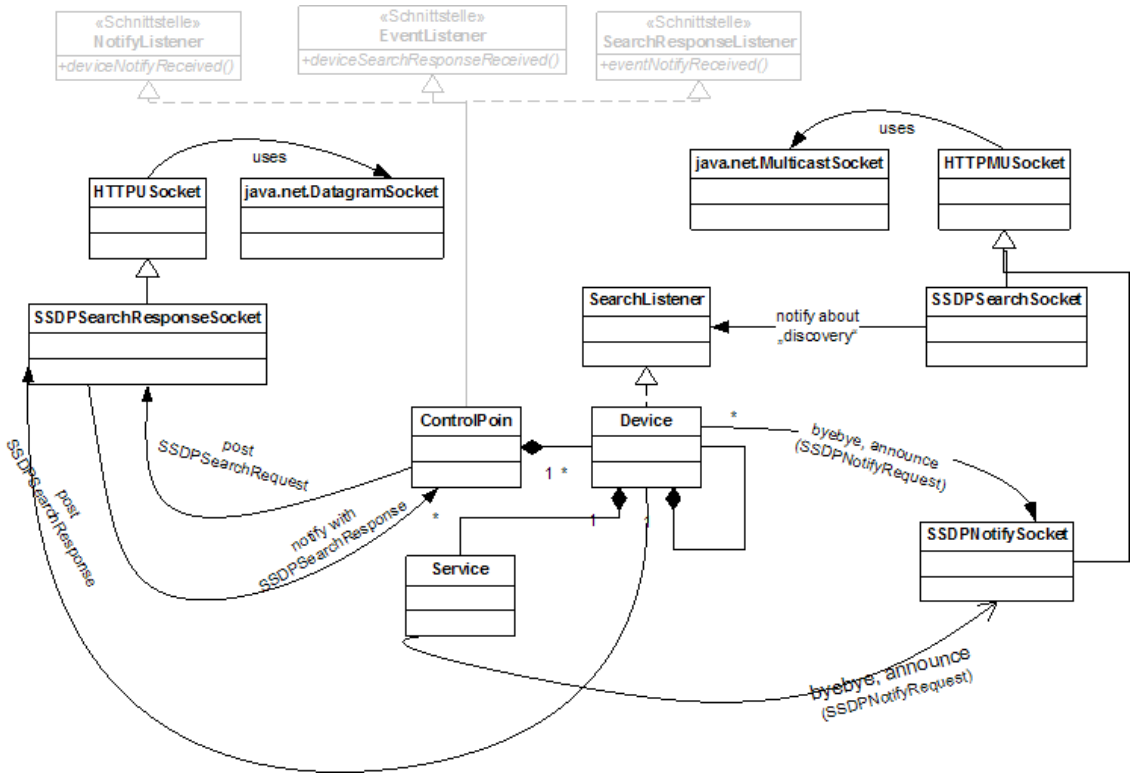


Figure 3.6: UPnP Cybergarage classes, which participate on the service discovery. The `java.net.DatagramSocket` and the `java.net.MulticastSocket` are the classes, which provide the interface to the transport system of the Java Virtual Machine (JVM).

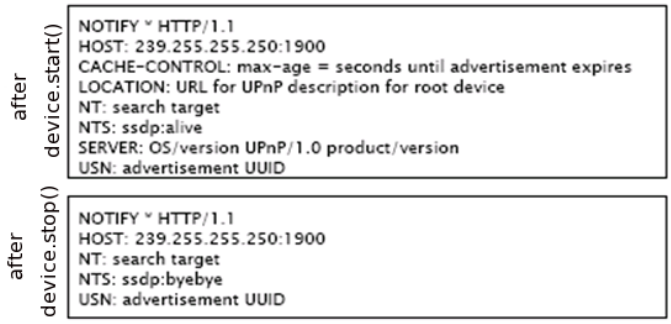


Figure 3.7: There are two types of the multicast *NOTIFY* messages (”announce” and ”byebye” shown on Figure), one multicast message of the type *M-SEARCH* and one unicast *RESPONSE* messages

The *UPnP Cybergarage* parameters like HTTP port or maximum wait for a device searching results before sending one more search request is configured by the changing appropriate variables and recompiling the code.

We will mention the technical problems, which we have accounted with the *UPnP Cybergarage* during validation phase in Section 6.

3.4.2 SLP

The service and clients (called ”user”) in SLP aware environment have Service and User Agents respectively. Additionally, the Directory Agent should exist on the network. Agents act on behalf of services/users. SLP not-aware services/users can be represented by ”SLP proxy”, which perform the same function as a corresponding agent.

The Directory Agent is a structure, not present on UPnP platform, which is responsible for the organization of the services into directories. The directories introduce centralization to the service discovery. In small installations, there may be no Directory Agents.

Network sources, under a common administrative control, are good candidates for inclusion into administrative domains called ”scopes”. Each user agent includes its configured scope in service requests. Scopes may also indicate geographic proximity or network topology.

Following tasks are accomplished by SLP (first two functions are main, the rest are auxiliary):

- user agent obtains service handles (**main**)
- directory of advertised services maintenance (**main**)
- discovering available service attributes (**auxiliary**)
- discovering available Directory Agents (**auxiliary**)
- discovering available types of Service Agents (**auxiliary**)

These tasks define the flow of messages between different entities on SLP-aware platform.

All the SLP messages are prefixed by a common header. The header contains the type, length, and other related information about the message. The communication flows are shown on 3.8 and explained in 3.3.

Message Type	Abbreviation	Usage
Service Type Request	SrvTypeRqst	UAs send this message to SAs and DAs to request the types of services available.
Service Type Reply	SrvTypeRply	SAs and DAs send these messages to the UAs in response to their requests. These contain the service types requested by the client.
DA Advertisement	DAAdvert	DAs send this message to the SAs and UAs to make them aware of their whereabouts.
SA Advertisement	SAAdvert	SAs send this message to the UAs to make them aware of their whereabouts.
Service Acknowledge	SrvAck	DAs send an acknowledge message to SAs in response to their SrvReg and SrvDeReg messages.
Attribute Request	AttrRqst	UAs send this message to SAs and DAs to request the attributes of a service.
Attribute Reply	AttrRply	SAs and DAs send this message to UAs in response to their AttrRqst message. This contains the list of attributes of a requested service.
Service Registration	SrvReg	SAs send this message to register their services with DAs.
Service Deregistration	SrvDeReg	SAs send this message to DAs if they no longer want to make a service available, causing the advertisement to be removed from the DA immediately.
Service Request	SrvRqst	The UA and SA initiates active discovery by multicasting a service request (SrvRqst) with service type `service:directory-agent`. The DA responds by unicasting a DA advertisement (DAAdvert) with its URL and the list of scopes that it supports. The SAs respond to UA too.

Table 3.3: SLP: Message types

`DAAdvert` can be sent both as multicast and unicast, `SrvRqst` are sent as multicast. There are messages, which require acknowledgement about request success. Such transaction can be built on Transport Layer protocol like TCP.

There are three ways to find Directory Agents:

Active Discovery	UAs and SAs send a multicast `SrvRqst` to the network. DAs reply with unicast `DAAdvert`.
Passive Discovery	DAs multicast periodic unsolicited `DAAdvert` to the network.
DHCP Options	A host that uses DHCP may use it to obtain a DA's IP address.

The SLP only define a way to locate a service and leaves open the interaction between clients and services after the service have been discovered.

OpenSLP: the Reference Implementation Example In SLP, an *agent* is a software entity that processes SLP protocol messages.

The Advertiser is the SA interface [RFC 2614],[RFC 2608],[SLP], allowing clients to register new service instances with SLP with call to `register(...)` and `deregister(...)`.

The locator is the UA interface, allowing clients to query the SLP framework about existing service types with `findservicetypes(...)`, `findattributes(...)` and `findattributes(...)`.

The ServiceLocationManager class contains methods that return instances of objects implementing SA and UA capability. it manages access to the service location framework with`getRefreshinterval(...)`, `getLocator(...)` and `getAdvertiser(...)`.

The ServiceLocationEnumeration class is the return type for all locator slp operations. The ServiceLocationAttribute class models SLP attributes, and its instances are communicated along with register/deregister requests.

The SLP can send messages 3.8 either as an unicast or as a multicast.

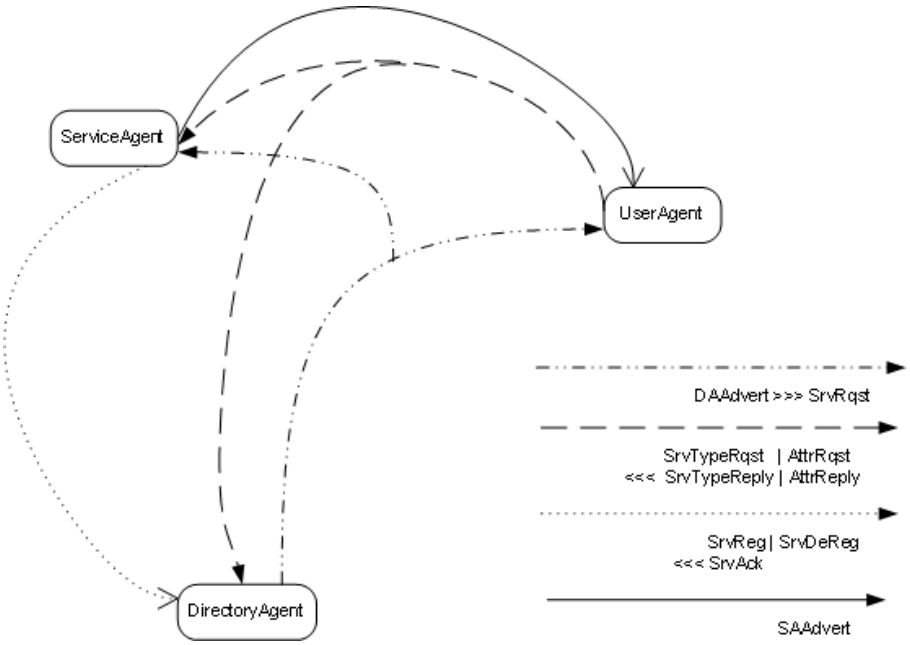


Figure 3.8: slp: message types and flows

OpenSLP implementation details:

- the SrvReg and SrvDeReg messages are sent over TCP.
- the SrvReg and SrvDeReg and Ack for them can be intercepted at NetworkManager: samessage(...), sareadack(...,...) respectively.

- the servicetypereques, servicetyperequest, attributerequest are finally processed in the servicelocationenumerationimpl.transmitdatagram() as udp message. to receive replies as shown in figure 3.8 the servicelocationenumerationimpl.next() is used, which is actually use servicelocationenumerationimpl.readresponse(). the replies are received as udp packages too.

3.5 Service Discovery Comparison

The terminology of different service discovery approaches may be quite different. Nevertheless, some concepts clearly overlap - the function of specific components from different SD frameworks are either similar or comparable. For example, the tasks of the Lookup Table from Jini are comparable with those of the SLP Directory Agent. So we assume that it should be possible to derive a generic model of various service discovery approaches by mapping their concepts between each other. Why do we need a generic model?

First, to derive the uniform terminology. Second, the Generic Model can be viewed as a *domain model* in terms of a UML domain model which ”relate objects in the system domain to each other” [JoFa 99]. Such view onto the service discovery middleware will be required during the design phase.

3.5.1 Generic Model

The concept of the *Service Provider* (or *Service*) and the *Service User* or *Client* is represented in all SD protocols (Table 3.4). The UPnP concept of *Device* can be viewed as a way to group several services, proposed by the device. The *Service Manager* is an entity, which usually plays the role of the proxy to the *Service Provider*. The relationships between service managers and services can differ. In the case of the UPnP, the service and root device are toughly coupled (see Figure 3.3). In the case of the SLP the service is something structurally independent from the service manager.

The *Service Description* is a way to describe the services in different SD approaches. The service description is used to describe the service and provide the information to the client. But the syntax, semantics and the set of possible operations on the service description are quite different. Nevertheless, the service description assumes, that each service has unique name, or *Identity* like ”printing service A” vs. ”printing service B”. The services are also identified by the *Type*, which syntax can still differ from the protocol to the protocol (e.g., ”print” service in UPnP vs. ”printing” service SLP). The device characteristics are described with the set of *Attributes*. As the result of successful service discovery the *Client* receives the *User Interface* or/and *Program Interface*, which can be used for further interactions with the *Service*.

The *Service Cache Manager* (*SCM*) is not represented in all service discovery technologies. The term *cache* here means that there is some repository where the information about services available in the environment can be inquired without need for discovering available services directly. The information about the services is said to be ”cached” in such repository. The *SCM* can be either a central repository (like in *Jini*), but it also possible that each client has own *SCM* as it is the case in Salutation. A Salutation client can still profit from the *SCM*, because instead of conducting new service discovery it can try to find the service in the locally available *SCM*. So the idea behind the *Service Cache Manager* component is to reduce the service discovery complexity associated with the increased number of clients and service available in the environment.

This findings of the current section will be used in Chapter 5, which deals with the *ESKEm* design. Here is just a short example of how the Generic Model could be used during design specification. The Services are represented to the Clients over the Service Manager. So it is important for us to have control over the Service Manager for imitation of the service life cycle. An example from the SLP specification can explain this consideration better: the Service Manager interacts on behalf of the Service with the Service Cache Manager, the Service is hidden behind the Service Manager, and so we do not require availability of the

Generic Model	**UPnP**	**SLP**	**Salutation**	**Jini**
Service Provider or Service	Device or Service	Service	Service Record, consists of Functional Units	Service
Service User or Client	Control Point	User Agent	Collection of Functional Units	Client
Service Manager	Root Device	Service Agent	Service Record	Service or Device Proxy
Service Description:	Device/Service Description	Service Registration	Functional Unit Description	Service Item
– Identity	Universal Unique ID	Service URL		Service ID
– Type	Device/Service Type	Service Type		Service Type
– Attributes	Device/Service Schema	Service Attribute	attribute records (name, value)	Attribute Set
– User Interface	Presentation URL	Template URL		Service Applet
– Program Interface	Control/Event URL	Template URL		Service Proxy
Service Cache Manager (SCM)	not applicable	Directory Agent (optional)	Registry of Salutation Managers	Lookup Service/Table

Table 3.4: Mapping concepts among various discovery protocols (motivated and based on work in [DaMi 01])

real service. It means, that if we want to emulate the services in the *ESKEm*, we just need to emulate the Service Manager part.

3.5.2 Differences

In this subsection we would like to concentrate on the differences between service discovery approaches. The differences between them usually results in the differences in the service discovery performance in terms of speed, number of messages sent etc.

What is the nature of the differences between two service discovery approaches? *First* source of the differences is the service discovery specification. One of the typical examples of such kind of differences would be the absence or availability of the Service Cache Manager. The *second* source of differences is the implementation differences. For example, the implementation language (very often C, C++ or Java) can have impact onto the performance of the service discovery. We are interest in the specification-determined differences in this section.

Why do we need the analysis of differences in the current work? One of their applications is to use them to derive the set of parameters, which may influence the service discovery performance. As an example, the availability of the Service Cache Manager (SCM) in the SD specification can be considered as one of such parameters. The SCM is a ”must” in some SD specifications (e.g. in Jini), other specifications do not use SCM structure at all (like UPnP). Some specifications (like SLP) specify the SCM as an optional component. We could consider the ”availability of the SCM” as the *parameter*: ”a variable, measurable property whose *value* is a determinant of the characteristics of a system” [AES]. The parameter’s values are usually referred to as the *levels* of the parameter. In our example, there are two levels of the parameter: ”the SCM is available” and ”the SCM is absent”. In the case of the SLP, the parameter can be *controlled* by the researcher (compare Use Case ”Exploration” from Section 2.3). In the case of the UPnP or Jini from the example above the level of this parameter is fixed and can not be changed by the researcher (but we can

		UPnP	SLP	Salutation	Jini
Misc	Service description	XML-based description file	service attributes, service URL	service attributes	service attributes
	Network transport	TCP/IP	TCP/IP	independent	independent
	OS and platform	dependent	dependent	dependent	independent
	Srv attributes searchable	no	yes	yes	yes
	Context-aware	N/A	Administrative domain	N/A	Location and administrative domain
	Leasing concept	yes	yes	no	yes
	Central cache repository	no	yes	optional using SLP	optional
Directory	Centralized vs. Distributed	N/A	Centralized	Either	Distributed
	Number of Service Information Copies	N/A	Multiple Copies	Multiple Copies	Multiple Copies
	Flat vs. hierarchical	N/A	N/A	Flat	Flat or hierarchical
	Number of directory hierarchies	N/A	N/A	N/A	Single hierarchy
Announcement and lookup	Query vs. announcement	Both	Both	Both	Both
	Directory-based vs. non-directory-based	Non-directory-based	Either	Directory-based	Directory-based
	Communication	Unicast and multicast	Multicast	Unicast and broadcast	Unicast and multicast

Table 3.5: The differences between Service Discovery Protocols. Motivated by [ZhMuNi 02] and [Bett 00]

still manipulate different characteristics of the SCM in Jini if we want to explore their impact onto the SD performance).

On the other side the difference between SD protocols should be taken into account during design of the *ESKEm.* So, the difference ”Directory address” implies that the way of finding the SCM differs among different SD technologies. We may wish to take into account and provide the way to configure the SCM address in the *ESKEm.*

Network transport Very often both of them can advertise it is availability as well as send requests to the network to discover other entities. The service discovery protocol resides at application layer as defined by OSI Model. Both TCP and UDP Transport Layer protocols are commonly used to transfer messages, which relates to Service Discovery.

Directory availability The availability of a directory is considered by some authors as the most important characteristic, which distinguishes between two classes of service discovery protocols: directory-based and non-directory-based. We prefer to use the terms ”mediator-based” and ”direct or peer-based” respectively as proposed in [Schi 04].

In mediator-based approaches, service discovery is performed using one or multiple mediators, called lookup services. Direct or peer-based approaches do not rely on the presence of a mediator.

Directories cache service information and answer clients’ lookup requests. Thus, the overhead of handling unrelated requests for services and the communication between clients and unrelated services are removed.

More importantly, this facilitates large-scale service discovery. Directory architectures, service information cache strategies, and hierarchies are different depending on the environments.

Centralike vs. Distributed Directories A central directory stores all the services' information in a central location. The directory is can potentially be a bottleneck and the single point of failure, which may cause the whole system's failure. In large service discovery domains, it is inefficient to go through a centralized directory all the time. Most of the service discovery protocols use distributed directories, which store services' information within their own domains. Service information is distributed among directories. A directory failure only affects part of the system. With less information in each directory, service lookup within a directory is more efficient. On the other hand, a service lookup may go through several directories. In contrast, a service lookup in a centralized directory only goes to one directory with less network communication overhead and latency.

Number of Service Information Copies For each service the service information may be a single copy, multiple copies, or fully replicated in directories. Many protocols have a single copy of services in its domain. A directory failure will affect the domain for which it is responsible. In Jini and SLP service discovery environments, multiple directories may coexist. Therefore, multiple copies of service information may exist. It is more reliable with multiple copies of service information in several directories, but the greater the number of directories, the greater the overhead. Some discovery protocols (like INS in [Wang 04] and [ZhMuNi 02], not in the table) implements fully replicated copies within a sub domain. The advantage of fully replicated directories is that a service search only goes to the directory to which a client is attached. Multiple copies or fully replicated copies of service information should be consistent in directories. Otherwise, querying different directories may result in different service information and may cause problems.

Flat vs. Hierarchical Directory Structure In a flat directory structure, directories have peer-to-peer relationships. In one type of flat directory structure, directories connect to each other and exchange information. These information exchanges generate much communication traffic, and therefore it is not scalable.

Which in a hierarchical directory structure, directories have parent and child relationships. Domain Name System (DNS) is an example of a hierarchical directory structure. Searching through the directory hierarchy is necessary. Many service discovery protocols use tree-like hierarchical structures to provide scalable solutions. Nevertheless, it is difficult to make directories both scalable and efficient.

Directory Address Conventional directories listen on well-known ports and clients are manually configured with the directory addresses. Moving to different networks, clients need different directory addresses. With DHCP servers, manual configuration is not necessary, but DHCP servers need to be deployed. To avoid directory address configuration or support from DHCP servers, some service discovery protocols use multicast addresses. Directories listen to and talk on a multicast address, such as in SLP. Clients query for services by sending requests to a multicast address or by listening on a multicast address. If multiple directories reply, clients can pick one directory to further communicate. Using multicast addresses, directories also provide fault tolerance features. A directory failure will not cause any problems, since other directories listening to the same address may provide the same services. Number of directory Hierarchies. A single hierarchy directory has a tree structure, while a multiple hierarchy structure could be a forest or many trees sharing a set of leaf directories. Multiple hierarchies index service information on different keys. Like a database index, service information search based on a key may greatly speed up the search. Extra computing resources and house keeping jobs are obvious overhead for multiple hierarchy directories.

Service Attributes A service usually has many attributes. Service attributes also have standard naming conventions as service names to avoid conflicts. A request of the client is matched against services'

attributes. More precise query requirements client results in fewer selected services. As a result, less network traffic is generated and fewer services are involved. If a query is too strict, no services may be matched and then the client needs to query again with fewer constraints. In addition to search functionality, almost all service discovery protocols provide wild card searches, which let clients examine all the available services.

Query vs. Announcement The two methods for clients to learn which service are available are query and announcement, also known as active and passive (or based on pull or push model). As announcements go to all the clients or directories, interested clients or directories do not need to ask separately for the same service. Nevertheless, clients or directories have to handle all the announcements, regardless of whether they are interested. When asking actively, a client or a directory will receive an immediate response. While listening to service announcements, a client or a directory may wait up to the interval of service announcement.

Service Selection If many similar services are available to a client, which service should the client use? It is challenging to find services for users efficiently and accurately. Service discovery protocols may select services for a user. But for most service discovery protocols, client programs or users choose from a matched list of services. The advantage of protocol selection is that it simplifies client programs or little user involvement is needed. On the other hand, protocol selection may not reflect the actual user's will. Predefined selection criteria may not apply to all cases. Alternatively, too much user involvement causes inconvenience. For example, it may be tedious for a user to examine many printers and compare them. A balance between protocol selection and user selection is preferred.

Service Matching Some service discovery protocols match one of the services for a client. In Matchmaking, a classified advertisement matchmaking framework, client requests are matched to services and one of the matched services is selected for user [5]. In INS, the service discovery protocol matches the best service based on application defined metrics. Most protocols match all the services and let the user choose.

Scope-awareness To support a large amount of service, defining and grouping services in scopes facilitates service search. Location information is helpful in many service discovery cases. Nevertheless not all service discovery protocols integrate location information or propose it as optional feature. For example, Jini uses location information for scope definition and it is an optional attribute for services. We say this is a scope searched by physical location. The administrative domain is another kind of scope. For example, two different departments A and B of the same company may wish to define and administer own domains for their services so that users from department A do not find printers from department B. Other types of a scope search can be defined like search by a network topology, geographical location. Multiple hierarchies may be built based on scope categories; so different service searches may utilize different hierarchies. As usual, the trade-off between speed of search and resources usage should be found.

Leasing Most of the protocols we have considered till now includes the concept of *leasing*. Each time a device joins the network and its services become available on the network, it registers itself only for a certain period of time in the lookup table (by mediator). This process is called a *lease* and a life time that defines how long a lookup table (mediator) will store a service registration is called a *leasing*. This is especially useful for very dynamic ad-hoc network scenarios.

3.6 Summary

In this chapter we have selected the *UPnP*, *SLP*, *Jini* and *Salutation* service discovery protocols for comparison between each other. It resulted in the derivation of the Generic Model, which should provide uniform

terminology among different SD technologies.

We also analyzed the differences between mentioned above service discovery technologies. It is a good starting point for derivation of parameters, which influence service discovery performance (see Section 4.3).

Because of the ESK interest in the UPnP and SLP middleware (see Requirement B.7) we have considered the mentioned protocols in depth. We also provided ad-hoc motivation for the selection of the UPnP and SLP for detailed consideration. They are oriented onto the intra-organization service discovery. The UPnP and SLP are not implementation-dependant (in contrast to Jini, which relies on Java specific features). Additionally, the UPnP is supported by a number of companies involved in its development (the SLP has the lowest number of companies involved in its development). The UPnP and SLP technologies seem to be widely accepted by the industry and they are going to be developed further (also compare to Table 3.4).

The *UPnP Cybergarage* and *OpenSLP* were selected as the reference implementations of the respective specifications. According to the Requirements B.7 and B.7, we will use the *UPnP Cybergarage* for validation of our approach in Chapter 6.

As a result of the service discovery overview, we are ready to make the first suggestion about the *ESKEm* design. This in turn results in the set of problems, which will be considered in Chapter 4.

We want to explore the service discovery technology in the specific environment. We want to explore existing service discovery reference implementations in the *ESKEm* emulative environment (see the Requirement B.1). It means, that we should recreate the environment, in which a service discovery middleware will exist. In this chapter we have considered the characteristics of such environment on the example of the office environment (according to the Requirement C.2). There are several findings, which should be taken into account in the rest of our work.

First, the service discovery protocols use a transport layers to accomplish their tasks. As the consequence, we should consider different approaches to the recreation of the networking environment (”Network Protocol” on Figure 1.1), over which the service discovery protocols operate.

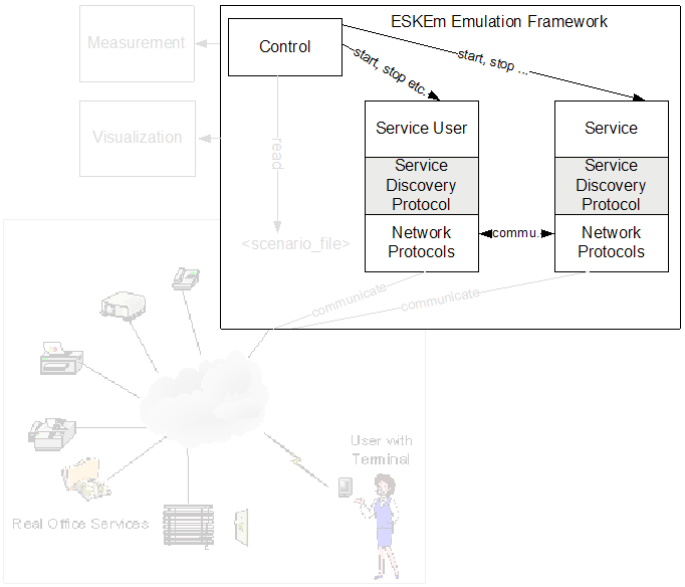


Figure 3.9: The overview over emulation framework design (see also Figure 1.1). The components in grey will be introduced in the subsequent chapters.

Second, we may require to provide components like ”Service User” or ”Service”, which would be responsible for simulation of user’s/service’s behavior respectively. What kind of service discovery related behavior should we simulated? The services/clients can be *started* or *stopped*, client can send a *search*

request, service can *advertise* itself to the network. The actions behind this activities still varies among different protocols. For example, the service manager (see Table 3.4) of the UPnP specification announce its availability to the complete network over multicasting. On the other hand, the SLP service manager register itself in the directory if it is available. The ”Control” module should be responsible for controlling simulated behavior and for reproducibility of this behavior (see Requirement B.3).

We continue the discussion of our ideas about the *ESKEm* design in the next chapter. We will try to derive there the related questions, which should be considered before we can start with more precise design specification in Chapter 5.

Chapter 4

Related Work

We have considered different Service Discovery (SD) technologies in the previous chapter. The service discovery technology is a *middleware* which sits in the *middle* between services or clients and the underlying transport system. It hides the details of the SD from the services and clients. We have considered SD in the previous chapter and identified a ”Network Module” (see Figure 1.1). In this chapter we identify additional modules required for the *ESKEm.* We also will analyze different approaches to the realization of the identified modules in detail.

4.1 Overview

Our first observation from the previous chapter was about the presence of the networking layers used by the clients and services during the service discovery. The networking infrastructure is one of the sources for the heterogeneity of the environment. The Requirement B.5 implies that we should be able to put the service discovery middleware (SDM) onto the transport systems with different characteristics. It can be quite costly if we decide to use real software and hardware to reproduce a network. The Requirement B.6 further implies that we should allow the communication between real clients and emulated services. Precisely speaking, the emulated services should be discoverable by the real clients. This concerns should be taken into account when considering different approaches used to *mimic* the behavior of a real transport system. These approaches are a testbed, a network emulation and a network simulation.

Our second observation was that the environment characteristics may put specific requirements onto the SDM in general and onto the service discovery protocols in particular. We can speak about effectiveness of the service discovery in a given environment. But how do we know that the SDM is effective in the environment? Or how do we know that the SD middleware A performs in the environment X better than the SD middleware B? These questions were actually the motivation for the Requirement B.2. It is not the goal of this work to enable measuring of some specific metrics. Indeed, the set of metrics used in the experiment depends on the particular research goals. In fact, we require a *measurement* infrastructure which would allow to add the measurement of new metrics easily. We make an overview for the various metrics used by researches during investigation of different service discovery protocols. This will help us to formulate requirement to the measurement infrastructure more precisely before we enter the design phase.

Further, we must define the *workload* for the SDM in order to measure its performance characteristics. The load on the protocol is ensured by the appropriate activity of the services and clients which use provided by the SDM functionality. Which activity of clients and services is relevant for exploration of the SDM? How is it possible to ”conserve” this activity so that it can be repeated in the *ESKEm* several times (see Requirement B.3).

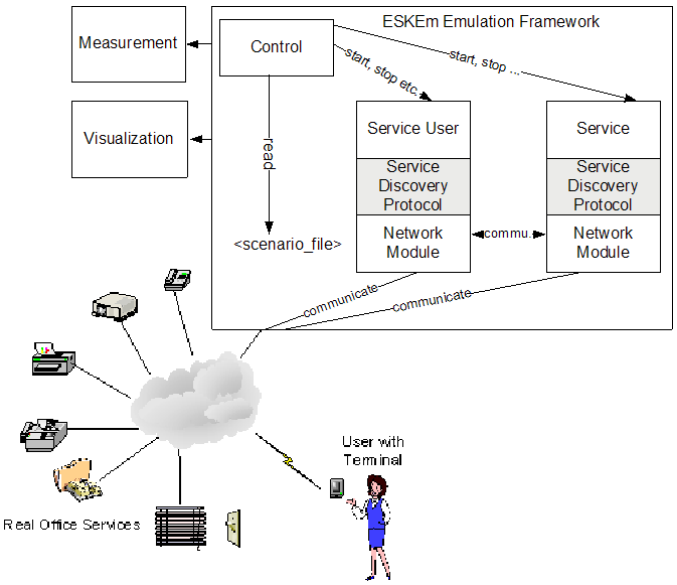


Figure 4.1: The overview over the design of the *ESKEm* emulation framework (compare with the Figure 1.1).

And last, we are interested in the approaches which could contribute to the visualization of service discovery related activity. These may be approaches for the visualization of the network-based activity, visualization of graphs etc.

4.2 Network Layers Simulation

The networking infrastructure is one of the most important components, in which service discovery middleware (SDM) operates. The properties of the networking infrastructure have great impact on the SDM performance. This motivates us to search for a way to reproduce a networking infrastructure and its characteristics in the *ESKEm*. The module with these tasks is labeled as ”Networking Module” on Figure 4.1.

What are the different approaches which are used by researchers to reproduce and explore the most important characteristics of a network? Which of the approaches are potentially relevant for our purposes? These are the questions which we consider in this section.

Different approaches are usually used to reproduce the network environments. Basically, they differ according to the level of abstractness (see Figure 4.2). Very often it would be perfect to have a real networking system available for a research. In this case no reproduction of the environment is required as far as it is already available. The approach is represented on the left of Figure 4.2. A real system consists of the real software and the real hardware components. As can be assumed, researches very often do not have access to a real system for a different reasons: too expensive, the real system with required characteristics does not exists etc.

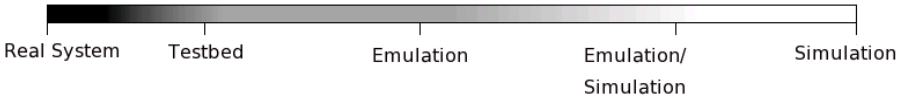


Figure 4.2: The range of approaches used to reproduce the networking infrastructure.

The simulation lies on the other side of the scale. The simulation approach is very often used when the networking system with specific characteristics does not exist and its creation is not possible for some reasons. In this case the model of the network system is created and used for the network simulation.

There are also other approaches which lay somewhere in-between of these two extremes. They vary in the amount of real components used to reproduce a network with desired characteristics. We provide a short introduction to all of the mentioned approaches for the reason of completeness. In fact, we are interested in the simulation and emulation approaches. These two will be considered in more detail.

Simulation. Simulation is defined in the [Chun 04] as:

> *Experimentation with a simplified imitation (on a computer) of a system as it progresses through time, for the purpose of better understanding and/or improving that system.*

The network simulation is a way to explore the network systems by imitating their characteristics. The network simulation is usually written in a special simulation language and is executed by a simulator. Both hardware and software components can be described by a simulation language and simulated. The implementation of a network protocol for the purpose of the simulation is usually totally different from a real implementation of the same protocol written to be executed in an operational system. The difference is especially clear for the real and simulated hardware components. Additionally, researchers are not expected to simulate all characteristics of a real component. They may concentrate on those of them which are assumed to be the most important for the purposes of a research.

What are the advantages of the simulation or When is it good to use a simulation approach? The network system simulation is probably the best way to study the behavior of a component or a system which does not exist. For example, we may wish to explore the protocol which exists as specification only and the protocol implementation is not available. Or we may wish to explore a system with a lot of users (e.g., 1000 users) who are navigating the web. In both cases it may be more appropriate to create a model of the network and use it for the simulation of the interesting network usage scenarios.

However, the typical problem with the network simulators - these are the programs which can execute the simulation - is that they do not support direct execution of a real protocol implementation. First, a researcher must implement the simulated protocol in the appropriate simulation language. Second, the researcher should take a decision about protocol characteristics which should be taken into account during protocol simulation. This latter task is not trivial. Implementation of an existing protocol in the simulation language is essentially a transcoding activity based on the idea of reducing the level of simulation abstraction. In other words, population of the simulation model with more details has as a consequence a decreased level of simulation abstraction. Hence, the obtained simulative code may fail to reproduce the behavior of the protocol due to either possible transcoding bugs or model simplifications that may lead to inconsistent results. The researcher should always find an appropriate level of details taken into account for the simulation. As a consequence, the simulation is not always appropriate. There is a seminal work by Brakmo and Peterson (Brakmo et al., 1996) showing that even if an abstract specification of the TCP transcode can be mostly useful for rapid experimentation, however running the actual TCP code must be preferred if errors have to be avoided. The same concerns are valid to the other protocols.

The costs of the network simulation are usually lower compared to to the costs for the other approaches for the network experiments. Among other things, we do not require any special hardware components.

Emulation In most general sense, an emulation duplicates the functions of a real system (RS) by substituting it with an emulated system (ES), thus the latter appears to behave like the first system [Wikipedia]. A third system - an emulation user (EU) - should not notice the differences between RS and ES. What is the difference between ”emulation” and ”simulation”?

In the definition above the equivalence of the RS and ES in respect to EU is the most important characteristics of the emulation. In other words, it is important that an interface provided by the ES to the EU to be the same as the interface of the emulated RS. There are a lot of examples where an emulation is used

in the computing environment. For example, an absent CD-ROM drive can be emulated in respect to the installation software which is expected to be installed from a CD-ROM only. Or we may wish to emulated multiple network adapters on the same host. The emulated network adapter propose the same interface to be used by the applications. In contrast to the emulation, a simulated network adapter can not be used by a real application.

Unlike the network simulation approach, the network emulation usually does make use of a real software like network protocol implementation or hardware like network routers to mimic network characteristics. Researchers very often reuse the real components like network drivers to develop ES.

Further examples of the emulation will be provided in the subsequent sections. But is a network emulation approach better than simulation? An emulation is usually associated with a higher accuracy of the imitation in comparison to the simulation approach. The problem of the simulation accuracy was already mentioned shortly in the previous paragraph: a researcher must find a tradeoff between the level of the simulation details and the simulation complexity. In contrast, an emulation approach does not force the researcher to search for tradeoff like the mentioned one. Indeed, all implementation details were already accounted in the software or hardware components reused for the emulation purposes. As a consequence, the emulation approach has inherently higher accuracy as the simulation approach. But it has its price. An emulation is not as fast as the simulation approach. The simulation of the complex networks is very often easier as its emulation. The emulation usually consumes significantly more resources as the simulation for imitation of the network with the same characteristics.

Further, it is obvious that it is *not* possible to emulate a host or a link which is faster than its hardware. Further, the real time behavior of the software is tied to the available specific hardware. Thus a hardware performance has an effect onto performance of the investigated network protocol. The example about the host with four was motivated by the [ZhNi 02] and [ZhNi 03], where a single workstation was equipped with several **network interface card** (**NIC**) for experimenting with routing protocols. But there are still the cases, when we need as many workstations as the instances of the protocol or application to be emulated. This problem can be partially overcome with application of the virtualization software like *VMWare*. But this approach still can only be applied for the emulation in small networks.

Testbed The testbed is even closer to the real system than an emulated system. The idea of a testbed is based on two basic assumptions. First, we have a real implementation of the protocol with which we would like to experiment. Second, a real network infrastructure exists where the protocol may be explored under the full control of a researcher. The main problem with such experimental environment is that it may still produce incomplete results. In other words, a testbed may be bound to some given network conditions during the test. For example, the traffic dynamics in a testbed may be different from that available in the real systems. Further, a testbed hardly supplies a fully controllable environment where experiments may be reproduced on a large scale at an affordable cost.

Real System The research in a real networking environment is the goal which is often hard to achieve. Usually, real systems are used in a productive environment, where every associated with the experimentation instability is unacceptable. Reasonably large environments may put constraints onto experimentation. It may be either hard or impossible to control specific networking parameters to study their influence on the network system performance.

To sum up, each approach has its pros and cons. The best approach does not exist. The best approach can only be defined in the context of the research goals. Even the real system can not be considered as the best approach for all cases. If we want to investigate the robustness of some protocol, it could be better to use either formal model checking or analytical modeling with the subsequent simulation. The investigation of the protocol in a real environment may be ineffective for such problem statement.

Which approach to reproduce a networking environment and mimic its characteristics would be the most appropriate in our case? There are actually several requirements which influence our selection. We use the requirements to the *ESKEm* already at this point to exclude from deeper consideration the approaches

which are inappropriate in current context. Requirement B.8 (scalability) assumes that we may need to reproduce reasonably big environments (compare with our description of the office environment in the Section 2.2). As a result, we are likely to fail trying to meet the low-costs Requirement C.1 with either the *testbed* or *real system* approaches.

The rest of approaches which are the *emulation*, *simulation* and *hybrid* (where both simulation and emulation is used), will be considered in more details in the rest of the chapter.

4.2.1 Simulation

A lot of network simulators have been developed until now. They posses very different characteristics. Some of them are targeted at research of networks with specific characteristics. For example, the GloMoSim is used for simulation in global mobile networks. Other simulators can be used to simulate a wide range of protocols. One of them is a widely excepted NS2 simulator. Some simulators use a general purpose language to implement the simulations, other provide its own language in which simulation must be written. Simulators also have different characteristics in respect to the accuracy, performance and scaling of simulation.

The topic of the simulation is quite extensive and there is a lot of different simulators. For this reason we selected only several network simulation frameworks to introduce in this section: NS2, DIANEmu and *JiST/SWANS* . It is not our intention to give complete insight into these frameworks but rather to analyze those features which are relevant in the current work. The most attention will be paid to JiST/SWANS framework which we have selected for simulation of the networking system in the *ESKEm.*

Analysis of the NS2 is interesting while it is a *de facto* standard for network simulations. The simulator also has a modification which enables a hybrid approach combining simulation and emulation to explor a network environment. DIANEmu is an environment for development of the application-level protocols in the ad-hoc networks.

NS2 NS2 is a discrete event simulator used to research of the networks [NS2]. It began as a variant of the REAL network simulator in 1989 and has evolved substantially over the past few years.

The NS2 is one of the most widely used general-purpose network simulators [Bajaj et al. 99a], [AlJi 03], [NS2]. NS2 is a recent version of the NS simulator. The modeling language of the NS2 is extensive. NS2 provides substantial support for simulation of networks with different characteristics. It concentrates on the simulation of the protocols at the Transport and lower layers (see Appendix C). Nevertheless, an approach in described in the paper [BiHe et al. 04] where the NS2 simulator is usd to analyze the impact of the application level **peer-to-peer** (**P2P**) traffic on the **internet service provider** (**ISP**) subnet. The **application level simulation** (**ALS**) was transformed and fed into the NS2 to add packet level details crucial for evaluating the influence of application behavior on the underlying network. The service discovery protocols are actually the application level protocol. For this reason the approach described in the paper was taken into account when designing the *ESKEm.*

Further, we were attracted by the NS2 emulation interface [NS2Emu] which is the modification of the ”pure” NS2 simulator. The NS2 emulation interface allows the simulator to interact with the real network nodes like hosts or their applications. This interface can be useful to evaluate the characteristics of the protocols and their *implementations* in end-systems. The NS2 emulation facility is placed as an intermediate or end node along an end-to-end network path. Further, the emulation interface can either capture or inject a real network traffic from/into real network. The simulator behind the emulation interface is responsible for the manipulation of the captured or generated traffic. The emulation facility differentiates between two modes of operations in respect to the modification of the traffic:

opaque mode real-world protocol fields are not directly manipulated by the simulator. But live data packets may be dropped, delayed, re-ordered, or duplicated. The protocol-specific traffic manipulation scenarios may not be performed. For example, it is not possible to drop the TCP segment containing

a retransmission of sequence number 23045. It is also no possible to modify the packet so that CRC error is introduced.

protocol mode the simulator is able to interpret and/or generate live network traffic containing arbitrary field assignments. The operations mentioned in the examples to the opaque mode are allowed.

As soon as the live traffic is captured by the emulation interface, the effects of the simulated network are applied to it. The real-time scheduler of the simulator synchronizes event execution within the simulator to real time. Provided sufficient CPU horsepower is available to keep up with arriving packets, the simulator virtual time should closely track real-time. If the simulator becomes too slow to keep up with the elapsing real time and the skew exceeds a pre-specified constant ”slop factor” (currently 10ms), a warning is continually produced.

We were also interested in the NS2 visualization tool (Nam) that allows researchers to see the complex behavior in a network simulation in the intuitive way. The visualization shows the dynamics of the network system and may allow a deeper understanding of the protocol behavior.

The simulation approaches can differ in respect to the set of network characteristics which can be taken into account during simulation. Usually it is possible to specify network topologies, specify network links characteristics like link failure probability, generation of complex traffic patterns etc. Just mentioned characteristics can be described as a scenario and executed multiple times. NS2 proposes two ways of the scenario creation: by configuring each single network component or by adopting the automated generation of scenarios. The second approaches provides an automated generation of scenarios. This in turn allows researchers to explore more complex scenarios than with a manual approach.

NS2 adopts two different levels of programming. A higher level uses the Tcl scripts to define simulation scenarios and to configure modules of the simulator. The low level programming allows the C++ implementation of new modules. For example, introduction of the new simulated protocols is associated with the creation of new modules in C++. This modules can be configured and used from within a Tcl script.

There are studies comparing performance and accuracy of the NS2 simulation with other framework. The NS2 is claimed to have pure performance in comparison to the other simulators (see [JSim 03], [SWANS], [SiBrUn 00]). The use of the Tcl for simulation scripts is claimed to be among factors which negatively influence NS2 performance. NS2 incurs also a substantial memory overhead and increases the complexity of the simulation code. NS2 was extended by researches to parallelize its event loop [RiAm 03], but this technique has proved primarily beneficial for distributing NS2’s considerable memory requirements. According to a number of published results, it is not easy to scale NS2 beyond a few hundred simulated nodes. The comparison of accuracy was found in [Bjoe 05]. NS2 accuracy varies from scenario to scenario. The tuning of the simulator can improve its accuracy significantly in comparison to testbeds.

Most efforts to improve the NS2 performance are concentrated around tuning its implementation and providing multiple levels of the protocol abstraction. The idea is to remove unnecessary details without violating the validity of the model. NS2 provides different levels of abstraction each of which sacrifices some details to preserve computer resources. A user can tune simulator performance versus accuracy by adjusting the simulation abstraction level. This means that one can increase the level of abstraction to improve the performance of the simulation or decrease it to supply a greater accuracy.

DIANEmu The DIANEmu framework was developed at the Technical University of Karlsruhe as a part of DIANE (*from germ. ”Dienste in Ad-hoc-Netzen”: ”Services in the Ad-hoc Networks”*) project [DIANEmu]. The DIANEmu was defined by the authors as a ”program for simulating ad hoc networks on a high level”. We were attracted to the framework because of the following goals of the DIANEmu stated in the [DIANEmu 04]:

- GOAL 1: Simulate ad hoc network protocols on a high level. It was attractive for us because of the fact that the service discovery protocols are the application level protocols.

- GOAL 3: Protocols can be transferred to real devices without change. It was attractive, because we also expected that an opposite direction would be true too. In other words, we expected that a real service discovery implementations can be explored in the DIANEmu.

Additionally, the design of the framework was likely to meet some of our requirements. The framework was only partially documented at the moment of writing current thesis. So it was not clear, how well our requirements are met. As a consequence, we decided to explore the framework. Our experience is introduced here.

The DIANEmu architecture is quite intuitive. It defines the following layers:

Meta layer - the documentation for the layer was not available, we did not explore the layer.

Behavior layer - define behavior of the service users.

Protocol layer - define protocols which are used by the users. This also include the protocol implementation.

Network layer - is responsible for delivering of the messages between nodes. The layer is rather rudimentary.

Connectivity layer - contains the topology of the network and its characteristics.

Additionally, the DIANEmu were expected to work in a distributed mode which could help us to fulfill the requirement of scalability.

Unfortunately, we were not able to reuse DIANEmu in the *ESKEm.* We analyze here the reasons, why the DIANEmu failed to match our requirements.

Development, testing and debugging of the new protocols are the most important goal of the DIANEmu. These goals put special requirements on the DIEANmu design. One of them is that the protocol execution must be deterministic.

In contrast, the performance and the scalability of the framework is not the primary goal. Several modes of the DIANEmu operation were mentioned in the manual: Event-, Thread- and Distributed modes. At the time of writing this paper only Event-based mode was supported (version 1.0 of the DIANEmu). The Distributed mode which would enable running the simulation on several workstations was still under development. The Thread-based mode was developed but not actually used because of being *non*-deterministic: scheduling of the threads was accomplished by the JVM (Java Virtual Machine) which led too often to the racing conditions between single nodes. The are also a number of implementation related decisions which negatively influence the performance of the framework.

We also failed to find a direct way how existing protocols can be integrated into the DIANEmu. Nevertheless, we used an approach similar to that described in the NS2 emulation interface (see 4.2.1). The real packets from the *UPnP Cybergarage* were attached to the objects representing messages in the DIANEmu. We also modified the scheduler of the framework. We emulated at total 30 clients, the clients started one after another with interval of 1 second. As a result, the last user was expected to start on 30th second. Each user sent a multicast ”search request” message as specified in UPnP reference. The results are presented in the Figure 4.3. The graph shows nominal vs. actual start of the users. The delay of the user start becomes visible very fast as the number of users increases. We attribute the difference between nominal and actual start time of the device to the design characteristics of the DIANEmu framework. Such conclusion is the result of investigation of the framework performance in the described scenario with a profiling tool. According to our suggestion, the difference is because of the large number of multicast messages which can not be processed by the DIANEmu at appropriate speed.

Proprietary *Visualization* module of the DIANEmu was unsatisfactory for our goals too. The visualization of the messages flow may be appropriate for small simulations or for the debugging purposes. But with increasing number of the simulated nodes the visualization is unlikely to be supportive. Neither code nor design of the visualization framework could be of use for our work.

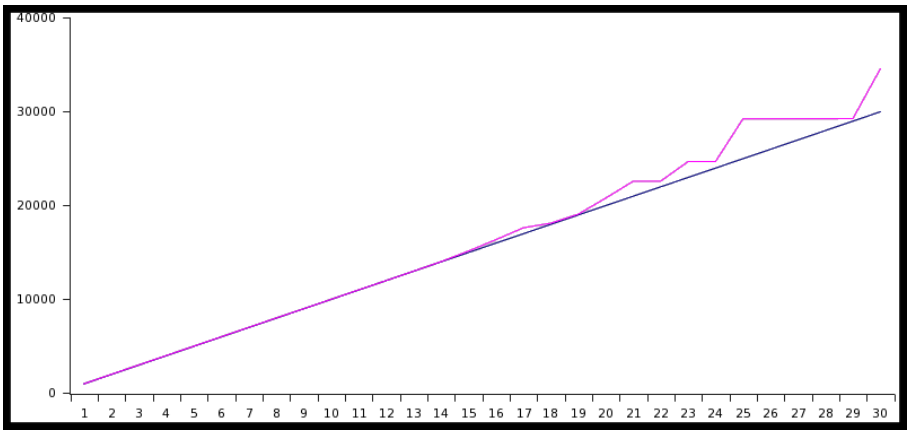


Figure 4.3: Load Test of DIANEmu combined with *UPnP Cybergarage*

Nevertheless, we were interested in the measuring infrastructure, provided by the DIANEmu. It consists of the *Gauges* which are registered to receive specific *Announcement*s like a `MessageSentAnnouncement`. The metrics are calculated as the result of processing and analyzing the announcements in the gauge.

As a conclusion, we attribute the differences between our expectation and the DIANEmu features mainly to the differences in the goals of our projects. Framework performance and scalability characteristics are unsatisfactory for our purposes. There is no support for integration of already developed service discovery protocols. As a result, we refused to use the DIANEmu framework.

JiST/SWANS The **Scalable Wireless Ad hoc Network Simulator** (**SWANS**) built atop **Java in Simulation Time** (**JiST**) platform, a general-purpose discrete event simulation engine [SWANS].

The *JiST* was developed at Cornelt University by Rimon Barr in May 2004. The JiST is a *discrete event-based* simulation. The JiST is rather complex to be covered here completely. The framework design can be found by interested reader in the PhD Thesis at [JIST]. Here, we would like to concentrate on principles of event-based simulation in general and on JiST-related concepts and mechanisms in particular.

Is such overview really required? Is not it possible to use the *JiST* simulation as a black box? Unfortunately not. According to our requirements (at least, the Requirement B.6), the *ESKEm* should be able to run in real time. The roots of this problem lay in the problem of coupling the simulation and emulation. We come back in Chapter 5 to this question and suggestions about its solution. Here we would like to provide the general background required for understanding the problem. We proceed as following. We shortly introduce the concept of the discrete *event-based* simulation and related terms. Next, we introduce the most important design aspects of *JiST/SWANS* which are directly related to the event-based simulation. The parts will be touched upon which are required later in the Chapter 5 for understanding the *ESKEm* design.

Discrete event-based simulation There are several approaches to the simulation in respect to the modeling of time progress. The time-slicing, continuous simulation or discrete event simulation are some of the examples mentioned in the [Robi 03]). The discrete-event simulation is used by the *JiST* framework. There are a number of mechanisms which were proposed for carrying out discrete-event simulation. Among them are the event-based, activity-based, process-based and three-phase approaches[1]. The JiST uses *event-based* approach.

[1] You may wish to check the [Robi 03] for references to the sources, where this approaches are described in details. The [Robi 03] gives a detailed example of the three-phase mechanism.

There are also other approaches. For example, with the time slicing approach the time is advanced at the fixed intervals, the time of the model during discrete-event simulation is advanced to the time of the next significant event. The event-based algorithm of the discrete-event simulation is explained in [Pidd 97]. We give here an informal explanation, based on the example with the booking clerks which has only two tasks to perform:

- the attend to people who arrive in person and queue for service
- they also answer the phone when it rings

An event-based approach simulation model would have the following event routines:

1 Personal inquirer arrives: if a clerk is idle, a service begins and its end must be scheduled via the event list. Otherwise, the arrival joins a queue. The next personal arrival event is scheduled on the event list.

2 Phone call arrives: if a clerk is idle, a phone conversation begins and its end must be scheduled via the event list. Otherwise the call joins a queue. The next phone call event is scheduled on the event list.

3 End of personal service: if a personal queue exists, a new service starts and its end must be scheduled via the event list; or, if a phone queue exists, a new phone conversation starts and it send must be scheduled via the event list. Otherwise, the clerk becomes idle.

4 End of phone conversation: if a personal queue exists, a new service starts and its end must be scheduled via the event list; or, if a phone queue exists, a new phone conversation starts and its end must be scheduled via the event list. Otherwise, the clerk becomes idle.

There are several observations. First, each event leads to the scheduling of another event. Second, the so called *simulation executive* controls the progress of the simulation by looking at the event list to find the time of the next event. It then executes the event routine(s) due at that time. Third, event execution results in more entries on the event list unless the event is one that terminates the simulation. When all events due at this current time are complete, the executive repeats itself.

Time There are three kinds of time which can be met in the literature about simulation: real executive time (or simply real time), executive time and simulative. The *real executive time* is the time of the real system which we want to simulate. The *executive time* is defined as the run time of the simulator. Finally, the *simulative time* is the representation of time in the simulative model. It is the wall clock of the simulation. For example, the simulation time is advanced above as the result of picking up the next event from the events list.

JiST design The overall design of the *JiST* is rather complex to be described here. We are going to consider the most relevant part of it which will be used as a reference in Chapter 5. We are interested in the exact way of the simulation in the JiST as far as it is a basis for the simulation in SWANS, a network simulator. Precisely speaking, we are interested in the simulation kernel. The complete *JiST/SWANS* design description can be found in [JIST].

JiST extends the traditional programming model with the notion of simulation *entities*, defined syntactically as instances of classes that implement the empty `Entity` interface.

entities Every simulation object must be logically contained within an entity, where object containment within an entity is defined in terms of its reachability: the state of an entity is the combined state of all objects reachable from it. Each entity has its own simulation time and may progress through simulation time independently. To be consistent, the entity cannot share its state with any other entry. Entities are components of a simulation and represent the granularity at which the *JiST* kernel manages a running simulation.

simulation events All instructions and operations *within* an entity follow the regular Java control flow and semantics. In contrast, invocations on entities correspond to simulation events. The execution semantics are that method invocations on entities are non-blocking. *They are merely queued at their point of invocation. The invocation is actually performed on the callee (or target) entity only when it reaches the same simulation time as the calling (or source) entity, see Figure 4.4. In other words, cross-entity method invocations act as synchronization points in simulation time.*

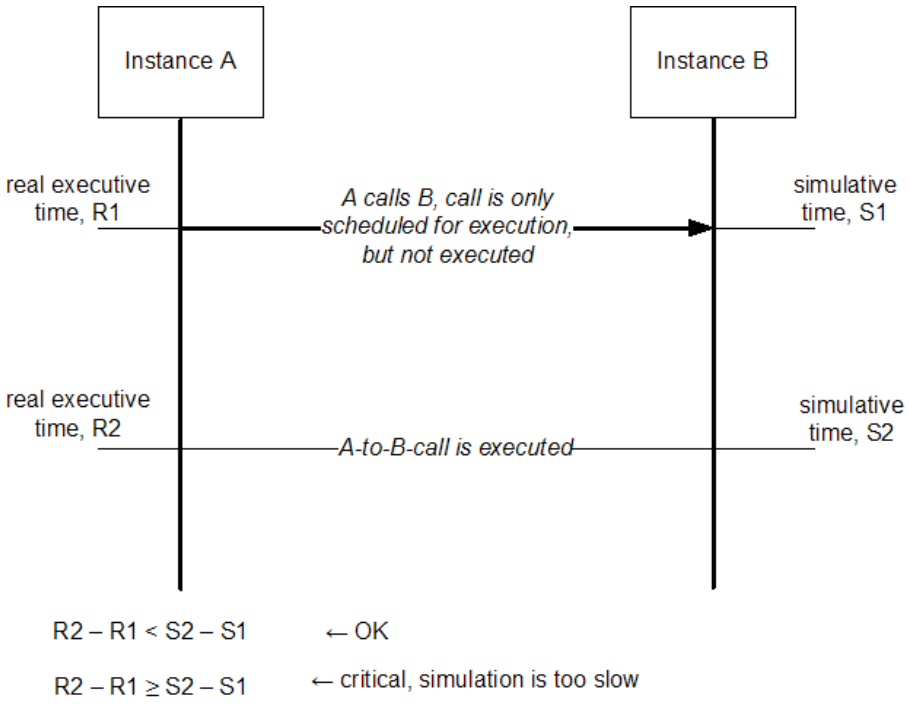


Figure 4.4: JiST/SWANS: simulation events are scheduled for the execution first; the execution is postponed to the time, when B reaches the same simulation time as A.

The node functionality is partitioned into individual, fine-grained entities. Here, an example for the node could be a device on the network or just an application which runs over the networking protocols stack. Partitioning of the node in entities would mean that the networking protocols on the protocols stack could be simulated as the single entities (e.g. one entity pro protocol). An alternative would be that a complete protocols stack is simulated as a single entity. *The partitioning into individual entities provides an additional degree of flexibility for distributed simulations.* The distribution of entities across physical hosts running the simulation can be changed dynamically in response to simulation communication patterns and it does not need to be homogenous.

In the next paragraph we introduce the SWANS framework (also called *JiST/SWANS*) which build on the top of the *JiST* simulator.

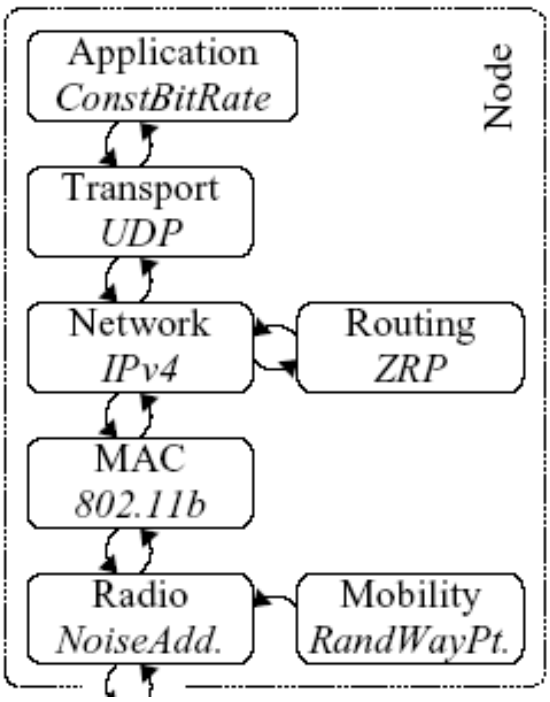


Figure 4.5: SWANS Design highlights, [SWANS]

SWANS design According to the authors, SWANS was created primarily because existing wireless network simulation tools are not sufficient for current research needs. Most published ad hoc network results are based on simulations of few nodes only (usually fewer than 500 nodes), for a short duration, and over a small geographical area. Larger simulations usually compromise on simulation detail. For example, some existing simulators simulate only at the packet level without considering the effects of signal interference. Others reduce the complexity of the simulation by curtailing the simulation duration, reducing the node density, or restricting mobility. The SWANS was targeted at both goals: high simulation speed and without the need to decrease the level of details.

The capabilities of SWANS are similar to NS2 and GloMoSim [GloMoSim]. The components of the SWANS implement different types of applications; networking, routing and media access protocols; radio transmission, reception and noise models; signal propagation and fading models; and node mobility models. The components can be easily interchanged with appropriate alternate implementations of the common interfaces and for each simulated node to be *independently configured*. Finally, SWANS also confines the simulation communication pattern. For example, `Application` or `Routing` components of different nodes cannot communicate directly. The messages can only be passed along their own node stacks.

The SWANS simulator consists of event-driven components (Figure 4.5) that can be configured and composed to form the desired wireless network simulation. The components of the simulator are clearly divided into the layers:

- Physical
- Link
- Network
- Routing
- Transport
- Application

As was already mentioned, the components can be replaced by the alternate interface implementation.

JiST/SWANS advantages As was mentioned at the beginning of the current paragraph, the motivation behind SWANS was to reach high performance without decrease the level of details.

> ”It is important to note that, in JiST, communication among entities is very efficient. The design incurs no serialization, copy, or context-switching cost among co-located entities, since the Java objects contained within events are passed along by reference via the simulation time kernel. Simulated network packets are actually a chain of nested objects that mimic the chain of packet headers added by the network stack. Moreover, since the packets are timeless by design, a single broadcasted packet can be safely shared among all the receiving nodes and

the very same object sent by an `Application` entity on one node will be received at the `Application` entity of another node. Similarly, if we use TCP in our node stack, then the same object will be referenced in the sending node's TCP retransmit buffer. This design conserves memory which in turn allows for simulation of larger network models.

Which performance is high enough? The performance of the *JiST/SWANS* was measured by the developers of the SWANS against similar frameworks by simulating an ad hoc network of nodes running a UDP-based beaconing **node discovery protocol** (**NDP**) application[2]. The scenario was also implemented in NS2 and GloMoSim for comparison.

The SWANS performs better in proposed scenario (a scenario for a wireless network), as can be seen from the Table 4.6 and Figures 4.7 and 4.8 as its counterparts - GloMoSim and NS2 ("hier" (hierarchical binning) and "scan" denotes different algorithms for radio-signal propagation, see [JIST] for more details).

nodes	simulator	time	memory
500	**SWANS**	**54 s**	**700 KB**
	GloMoSim	82 s	5759 KB
	ns2	7136 s	58761 KB
	SWANS-hier	*43 s*	*1101 KB*
5,000	**SWANS**	**3250 s**	**4887 KB**
	GloMoSim	6191 s	27570 KB
	SWANS-hier	*430 s*	*5284 KB*
50,000	**SWANS**	**312019 s**	**47717 KB**
	SWANS-hier	*4377 s*	*49262 KB*

Figure 4.6: Data points showing SWANS time and memory performance, relative to GloMoSim and NS22, running node disocvery protocol simulations [JIST].

The code of the *JiST* is also claimed to be compact. According to authors, components in JiST takes less than half of the code (in uncommented line counts) of comparable components in GloMoSim which are already smaller than their counterpart implementation in NS2.

Beside high simulation throughput, other advantage of the SWANS is its ability to run standard Java applications over the simulated network without modification, thus allowing for the inclusion into simulation of existing Java-based software, including application-level protocols. These applications do not merely send packets to the simulator from other process. They operate in simulation time within the same *JiST* process space, allowing far greater scalability (the overhead for context change by multithreading is high).

Further, the components of the SWANS can be easily interchanged with an appropriate alternate implementations of the common interfaces and can be independently configured for each simulated node or application. Among other things it means that nodes can be configured with different network protocols stacks.

[2]NDP utilizes the entire network stack and transmits over every link in the network every few seconds.

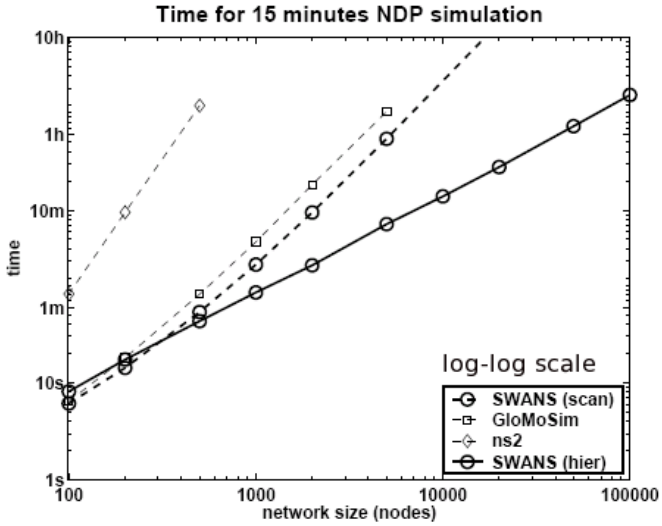


Figure 4.7: SWANS outperform both NS2 and GloMoSim in simulations of the node discovery protocol, log-log scale [JIST].

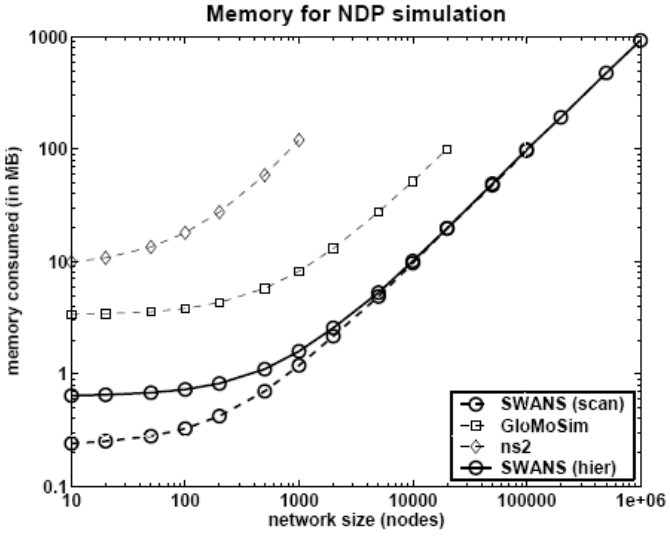


Figure 4.8: SWANS can simulate larger network models due to its more efficient use of memory.

As was mentioned, the SWANS performed better as NS2 and GloMoSim for the wireless networks. What about simulation of LAN? Or put it in other words, what are the sources of performance? Will they be still there, if we decide to introduce additional protocols like those for the LAN into the *JiST/SWANS* ?

According to the *JiST* documentation, two source of the performance can be derived. One of them is the overall architecture of the JiST which reduces threading overhead, synchronization costs, and message copying during event dispatch. Another source of performance is specific to the simulation of wireless network. It is a so called binning algorithm (see JiST design documentation for further explanations). So we can still expect to profit from the design of the JiST to achieve higher performance as with NS2 and GloMoSim.

Unfortunately, we do not aware about any reports about accuracy of the network simulation with SWANS. *JiST* is also very young - it appeared only in May 2004 (in contrast to NS2 which is *de facto* standard for

the network simulations, and widely known GloMoSim with its specialization on mobile networks). We did not found any *JiST/SWANS* related papers besides those published on the home page of the framework [SWANS].

4.2.2 Emulation

We cannot consider emulation as an alternative to simulation, but rather as a complementary approach. The main idea behind emulation approach is to use real hardware (e.g. workstations, routers, network adapters etc.) and real software (e.g. network drivers, routing protocols software etc.) to reproduce the real network. The main advantage of this approach is that an emulator runs real protocols. This means that the behavior of a real protocol and all its implications (like processing overheads for example) must not be simulated.

The emulation approaches exploits the resources of real powerful component for reproduction of *multiple* less powerful components. For example, the EMPOWER framework described in the [ZhNi 02] integrates multiple network adapters on the single workstation. Each network adapter is responsible either for a router or end host. The CPU, memory etc. are the resources of the host which are used to run router code. Another example would the slower links and larger propagation delays which can be simulated by means of links with reasonable higher bandwidth.

The word of ”simulation” may disappoint the reader. Are not we speaking about emulation right now? Aren’t we done with ”faked” protocols? The reason to use the term of ”simulation” in this context may be not obvious for the first time. The term of ”emulation” is applied to describe some inherent characteristic of the real component. For example, it could be a failure to send every 10th package because of the bug in the protocol implementation. On the other hand, the ”simulation” denotes some kind of the behavior which is **not** attributed to the protocol by itself. For example, we may wish that our protocol implementation from previous example fails to send every 5th package. This kind of behavior is simulated. In accordance to the previous notes, it is not possible that the protocol fails to send every 20th package.

For the explanation of the concepts in this section we use the concept map 4.9. It should provide an overview over all important concepts used in this section. We introduce the term first, and than put them into the context of the service discovery exploration.

As to network emulation, it promotes the integration of the real network into an emulative environment, leading to an emulative scenario. Special interfaces called *emulation interfaces* are provided to introduce *live network traffic* into the emulative environment and to inject *simulated traffic* from the emulative environment back into the live network.

The main aim of any network emulator is to reproduce the behavior of a *network scenario* in order to study the performance of the integration between this scenario and the software communications over it which is accomplished by the *system under study (SUT)*. Therefore, each emulator needs a *network model* to mimic a specific network scenario and to characterize its properties.

The *emulation interface* is a combination of both software and hardware that provides an *entry point* to the SUT within the implemented network model (see also Figure 4.10). The emulation interface is in charge of integrating the emulated network and its live traffic coming from the SUT, operating either in opaque or in protocol mode (compare the NS2 emulation facility mentioned in 4.2.1).

The SUT can be located at different layers. This means that the emulation interface must supply entry points at appropriate for the SUT layer which is named the *emulation abstraction layer*. The emulation interface can have several such layers. The choice of an emulator for a specific scenario depends on the emulation abstraction layer it supplies for the study of the SUT. Which layers are usually considered as adequate to be an emulation abstraction layer? Here they are:

Transport Layer, here emulation reproduces the features of the communication channel at the transport level (of the stack ISO-OSI) like TCP or UDP. The entry point of this kind of emulators is set at transport layer, so they promote the measurement of the performances of the applications on their emulated communication channels.

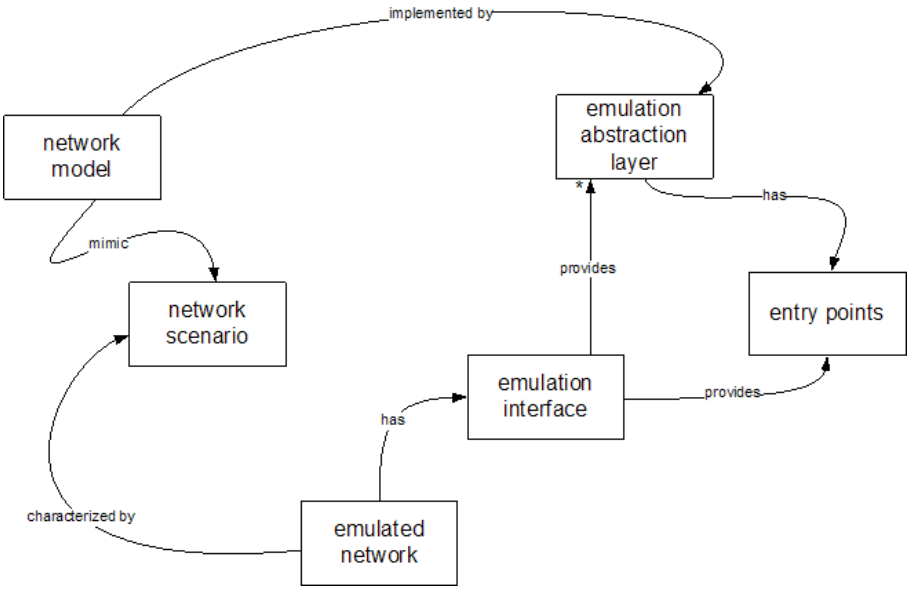


Figure 4.9: The emulation approach concept map. See explanations in text. Read it like ”emulated network *has* emulation interface”.

Network Layer, here emulation mimics the end-to-end behavior of a network connecting hosts at the network level, by reproducing the performance of the network protocol due to packet delays, congestion losses, etc. The entry point of this kind of emulators is set at network layer, so they promote the measurement of the performances of the transport protocols and the applications on their emulated end-to-end behavior of a network.

Link Layer, here emulation replicates the behavior of a single network links at the data-link level due to limited bandwidth, frame delay, etc. The entry point of this kind of emulators is set at Data-Link layer, in addition to the measurement of the performances of the network protocols, they promote the transport protocols and the applications on their emulated links.

As in the case of the simulation, we still have the problem of accuracy with the emulation approach. Or put it in other words, how good does the emulated network reproduce the characteristics of the real network? The problem is that each aspect of the network model should reproduce physical effects, hardware design, and protocol issues of a real network. For example, network layer emulation should reproduce (”fake”) the effects of the network level, like routing, queuing, discarding etc. Instead, link layer emulation approach can exploit the actual implementations of network protocols to create these effects. Hence, these considerations show that it is more difficult to realize realistic emulation on a higher abstraction layer than on a lower. This is due to a great number of factors that one should consider. Lower emulation abstraction layer will carry out more realistic results. According to [HeRo 02], the emulation on link layer is the lowest possible emulation abstraction which is possible without using special hardware.

The emulation approaches show some limitations due to the hardware on which emulation runs: i) to emulate a network with ten hosts one needs ten workstations, ii) it is not possible to emulate a host or a link which is faster than its hardware and iii) the real time scheduling of the software is constrained to the frequency of the specific used hardware. For example, if an emulator has a time accuracy of 10 ms, it cannot emulate the exact delay involved in sending packets smaller than 20 Kb on a 2Mb/s link, because it takes exactly 10 ms to send this amount of bits.

It is also obvious that the approach requires availability of the real components in appropriate amount. Thus high bandwidth link can be used for simulation of lower bandwidth links, it still has the capacity limits: it is not possible to simulate more than 10 of the 1Mb/s links with a single 10Mb/s link.

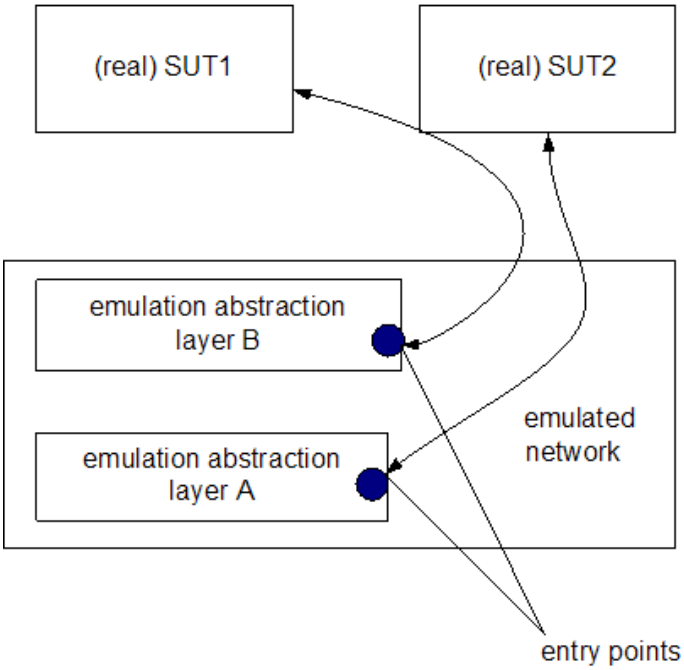


Figure 4.10: A generic emulation framework architecture

EMPOWER emulation framework EMPOWER is a framework developed at the Department of Computer Science and Engineering of Michigan State University. The EMPOWER is and emulation environment which is used for investigation of the routing protocols in wireline and wireless IP networks. This could be just what we need according to our vision of the office environment (Section 2.2). The emulative environment consists of several workstations (*emulation nodes*), each equipped with multiple network adapters to emulate the routers (*virtual routers*). The real network routers topology is mapped onto this environment. to an emulation configuration in a private local area network.

Our primary expectation was to put the service discovery middleware behind virtual routers. But as will be seen, this solution can not match our requirements. The application of the EMPOWER approach would resulted thus in higher accuracy but in pure scalability of *ESKEm.*

The problem of EMPOWER performance was investigated in [ZhNi 03]. When more than one NIC (network interface card) are plugged into a workstation, the maximum throughput of each network port may not achieve its link bandwidth (100Mbit/s). For the purpose of the experiment, the NICs on the single workstation were chained (see Figure 4.11), the input and output throughput were compared and related to the load of the usage of the system resources like CPU. Among other things, the experiment derived a list of factor which influence the maximum end-to-end throughput of the routers: CPU speed, PCI bus bandwidth, system bus bandwidth, buffer size of the network interface card.

We were mainly interested in specific experiment results. So, it was found that the resulting throughput of the host with 4 chained NICs reached 60 Mbit/s (from 100 Mbit/s maximally available). There are two reasons for such decrease in performance: resources competition and pure performance of software-based routers.

The authors tried to improve EMPOWER performance. As a result of these modifications the maximum network throughput was increased by 23%, but nevertheless it was not possible to achieve an end-to-end throughput of greater than 75Mbits/s in case of a virtual router chain of 4 virtual routers.

The paper mentions also several possible approaches for increasing performance in similar to EPOWER emulative environments. They are modifications to the interrupt-driven mechanisms and techniques of

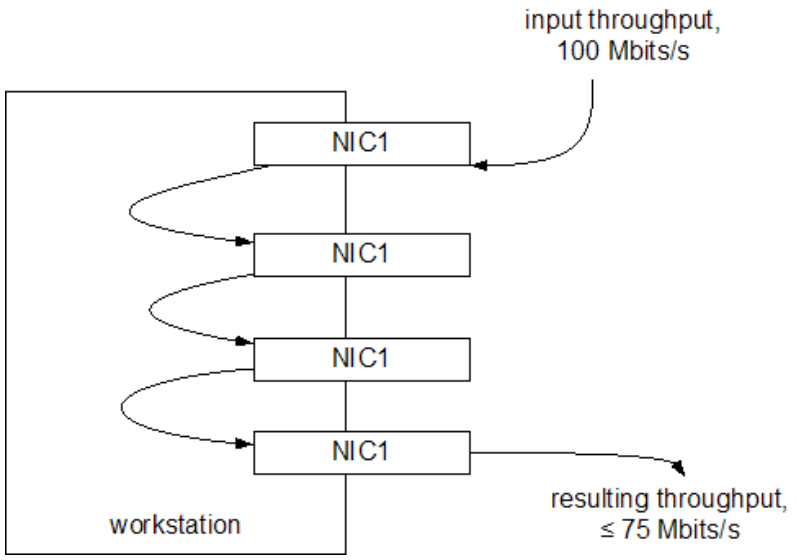


Figure 4.11: EMPOWER: performance degradation because of the resources competition and pure performance of software-based routers.

interrupt mitigation (see [ZhNi 03] for further references). To reduce the number of memory copies of a packet in a host-based router, peer-DMA is used to directly copy packets between two network interface devices. This approaches can be further investigate, if the problem of the performance arise during progress of the current work. The approaches mentioned seems to be more promising as that used in EMPOWER. But we assume that an increase in performance or scalability which would be acceptable in our case, can not be achieved. The reason for the conclusion is that emulation is tightly coupled with the execution of real code on real hardware. There is no way in context of pure emulation approach to abstract some resources-consuming operations and replace them with a simulation. Thus winning in the accuracy of the emulated network characteristics, we are likely to loose in the scalability (Requirement B.8) with the EMPOWER.

Dummynet, NISTNet One of the most well-known network emulators is Dummynet [Rizz 97]. It was developed by Luigi Rizzo at the University of Pisa. Dummynet is a simple, but flexible and accurate network emulator that was built as a result of modifications to the communication protocol stack.

Dummynet runs on a standalone system by intercepting communications of the protocol layer under test and by introducing them into an environment where the effects of bandwidth limitations, communication delays and packet losses are exploited. In other words, the idea is to insert a simplified network model characteristics into an operational protocol stack and to conduct trails on a standalone system.

Dummynet is actually a hybrid between the simulation and emulation. It has control over parameters of emulation, simplicity, capacity to use real traffic generators and real protocol implementations. A set of unmodified real world applications like FTP, SSH and HTTP on a workstation can be used in the experiments with network protocols. Dummynet is inserted between the interfaces of the network layer and the transport layer of the protocol stack of the computer on which is executed (Figure 4.12). Therefore, it reproduces the network infrastructures between two hosts including two different types of elements in the stream of the communication: routers and links. In order to simulate their presence, respectively a router is represented as limited queue and adopted queuing policy, while links are represented as bandwidth limitations, communication delays and packet loss.

The basic configuration of a trail consists of one or two routers and one link. These elements are modeled by means of a couple of queues (namely, router-queue and link-queue) that intercepts the communications between the protocol layer under observation and its lower layer. The queues can be configured separately for each direction of the communication. The router-queue is characterized by a maximum size, while

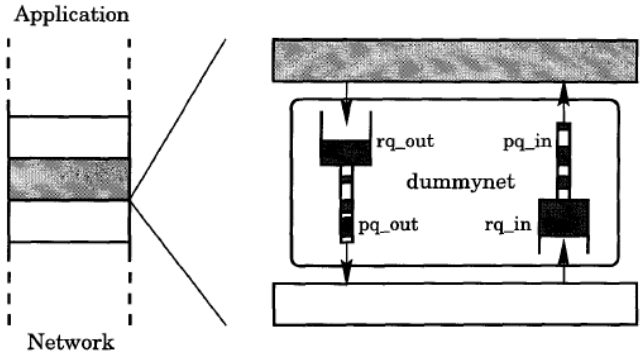


Figure 4.12: The principle of operation of dummynet, [Rizz 97].

the link-queue by a bandwidth and a communication delay. The packets are processed in the router-queue and than in the link-queue before being sent to the protocol layer under observation. For example, the packets in the router-queue could be eventually reordered. Next, the link queue could be used to simulate the bandwidth limitation by holding the packets for a while before sending them further.

Dummynet like the other network simulators and emulators show some limitations to reproduce the behavior a real system. The main limitations come from the granularity and the precision of the system clock. The granularity limits the resolution in all timing related measurements. This is due to the modeling of high-speed networks where the packet delay becomes comparable with the time quantum of the system clock (see also our ”20Kb on 2Mb/s link” example). Another limitation is that the overload of a time-driven system might deliberately delay some packets or might miss one or more timer ticks. Finally, we conclude by highlighting that only a prototype implementing this approach between IP and TCP exists. This is certainly in contradiction to the Requirement B.5.

The NISTNet network emulator is a general-purpose, Linux kernel-based tool for emulating performance dynamics in IP networks [CaSa],[NISTNet]. It has a fixed queue size, but adds delay variation, packet reordering, and packet duplication. NISTNet is a tool working within the Linux kernel. As Dummynet, it can introduce bandwidth limitation, delay, and packet loss, and then it adds delay variation, packet reordering, and packet duplication. NISTNet operates at network emulation layer and therefore can only be configured on the basis of IP layer addresses. By using up a number of connected nodes running an emulation tool, a comprehensive network scenario can be set up.

4.2.3 Summary

The *performance* and *accuracy* are the most important criteria which matter when selecting either simulation or emulation framework to mimic networking environment. We consider the scalability (Requirement B.8) as one of the performance characteristics. The scalability of the *ESKEm* has higher priority for us as the accuracy. This is the first reason, why we selected to apply simulation approach to reproduce networking infrastructure in *ESKEm.*

Other consideration is that with an emulation approach it can be problematic to achieve the heterogeneity (Requirement B.5 of the reproduced networking environment. Emulative environments usually specialized on the reproduction of specific network environments. Additionally, the emulation is usually associated with higher investment compared to the simulation approach, and this is not desirable in our case (Requirement C.1).

So we would like to consider different simulation approach in respect to the performance and eventually accuracy. In general, the simulation seems to be more liable to inaccuracies as emulation because of its

model-based nature. Surprisingly, few papers and studies can be found dealing with the topic of network simulator comparison. The main reason for this is that most developers are only familiar with one simulator, and a daily comparison just does not take place [Bjoe 05].

The network simulator design, implementation language and levels of the simulation details are mentioned in the papers as having the most important impact on the simulators' scalability and performance. It is not obvious that the lower-level languages guarantee higher performance. As was shown above, written in C++ NS2 performs worse as written in java *JiST/SWANS* (and some other not represented in this chapter java based simulators like JSim [JSim 03]). This can be attributed both to improvements in **java virtual machine** (**JVM**) and architecture of the simulator itself.

The researcher may also have more flexibility in determining the level of detail with simulation approach in contrast to emulation. The level of details has actually strong impact on both of performance and accuracy of the simulation. What are the examples for differences in the level of details? One of them would be the timescale of the simulator (we mentioned it when considering the NS2). An illustrative example about influence of the level of details on the accuracy is provided in [Heid 00d].

Our finding from the overview over simulation and emulation frameworks can are short summarized in the Table 4.1.

Require-ments	*JiST/SWANS*	NS2	DIANEmu	Dummynet	EMPOWER
B.8	+	-	-	-/+	-
B.5	+	+	+/-, pure support for transport and lower layers	-	-
B.1	-	-	-	-	-
B.6	-	-, + for NS2 emulation interface	-	-	-
C.1	+	+	+	+/-	-
C.4	+	+	+	+	+

Table 4.1: Comparison of selected simulation and emulation effort in respect to *ESKEm* requirements.

4.3 Measurement

The concept of *measurement* is widely used. We want to use this section to define precisely, what we understand under the *measurement* in the Requirement B.2. We will also considered related concepts like *metrics*, *workload* and *monitors*. Next, we will consider the measurement in the context of the service discovery researches. We are mainly interested in the question which metrics are of interest for different researcher goals.

Informally speaking, the measurement should be considered as a way to get feedback about system performance. The term *metrics* refers to the criteria used to evaluate the performance of the system. For example, the time required for the service discovery (is searched service is available on the network) could be used as a metric to compare two different service discovery approaches. The concept of the *workload* is used in our case to characterize the client and service activity which "load", or provoke the service discovery operation. The metrics which we will consider are measured under certain workload conditions. The workload characteristic will be considered in the Chapter 5 about *ESKEm* design.

In the *ESKEm* we are interested in the measurement infrastructure which would support us in measurement of different metrics which would reflect the operation of the service discovery protocol under consideration. We are not interested in the measurement of some specific metrics. Nevertheless we would like to

consider which metrics are used by researcher to capture different performance characteristics of the service discovery protocols. We will come back to the measurement infrastructure characteristics in Chapter 5.

We summarized the metrics used in different papers in the Table 4.2. We aware that the set of the metrics is strongly depend on the goal of the experiment. But we the goals of the single experiments are not of primary interest for us.

Category	Metric	References	Explanation
speed	average waiting time	[Hong 04]	the minimum time period in seconds, averaged over all the clients, starting from the sending of a service query message and ending with the receiving of a service reply message
	discovery speed	[Junw 01]	total number of requests during a certain period divided through
	update responsiveness	[DME 02]	How much latency is required to propagate changes
reliability	update effectiveness	[DME 02]	What is the probability that a node receives a change?
	number of transmitted messages	[Hong 04], [GoBa 00] (analytical model)	the number of messages transmitted in each round by all the nodes in the network, including search discovery, reply and search announcement messages.
	service discoverability	[Chak 02]	expresses the chances of a protocol discovering a service
	success rate	[Hong 04], [Bjoe 05], [Junw 01]	the ratio of the number of clients which successfully locate the services, over the total client number.
power and resources consumption	update efficiency	[DME 02]	How many messages must be sent to propagate a change throughout the topology
	load balancing	[Junw 01], [Chak 02]	express the degree, to which the load among different network components (like, devices) is fairly distributed
	energy efficiency	[Schi 04], [Chak 02]	amount of time that idle nodes spend in sleep mode

Table 4.2: The categories of service discovery metrics as found in research papers

We identified several three groups of metrics (see Table 4.2). This can be interpreted in the way that the researchers exploring different service discovery approaches are usually interested in service discovery *speed*, *reliability* and *power/resources consumption*.

Further, we differentiate between component under study (CUS) which is the service discovery protocol and system under test (SUT) which is the environment of the CUS. The metrics are measured on the level of the SUT. As we have mentioned earlier, the service discovery can be considered as a middleware. It means among other things that there are components of the SUT which lay above and under the service discovery component (compare also with the Figure 4.1). As a consequence, the metrics are measured either at the upper or at the lower level of the SUT in respect to the CUS. For example, *number of transmitted messages* is the metric at the Network Module level (see Figure 4.1). The *success rate* can only be measured on the service user level as far as only client can decide, whether the discovery was successful or not. The metrics like *load balance* are measured even at higher level like the level of the device, on which client or service is located.

Found metrics could also be characterized as either *individual* metrics (reflect the utility of each user) or

global metrics (reflect the systemwide utility). Mentioned in the Table 4.2 ”load balancing” metric is good example of global metrics. The difference is that the global metrics does not make sense for a single node instance. For example, the fairness metrics make sense only with more than one node. Other examples of global metrics would be reliability, and availability. As a consequence, in general a single node can not decide, whether it is reliable or not. As a result, the global metrics can be usually calculated based on the information from the whole system. On the other hand, individual metrics like name *lookup speed, number of transmitted messages* or other may be measured for each individual as well as globally for the system. Sometimes the decision that optimizes some individual metrics is different from the decision that optimizes the global metric. For example, in the network where the total number of packets allowed is kept constant, increasing the number of packets from one source may lead to increasing its throughput, but it may also decrease someone else's throughput. Thus, both the systemwide throughput and its distribution among individual users must be studied. In general, using only the individual metrics or only the global metrics may lead to unfair results.

4.3.1 Workload

The service discovery protocols are used for discovery of services. They are used by services and clients. The service discovery protocols dont make too much sense without them. The measurement of the service discovery characteristics can only be revealed under appropriate activity of services and clients. We call this activity as the *workload*. The workload is applied at the level of the SUT (system under study).

What should be used as workload in our case? We did not find any workload description or characterization which would be relevant for the service discovery. But we can still define it base on the overview of different service discovery protocol from Chapter 3. On this place we just say that it should be some activity on the side of service and clients. We will come to the problem of workload in the Chapter 5. The workload is something which affects the performance of the SDM under consideration. Other factors affecting the SDM performance are located at the system level parameters.

Both system and workload parameters should be taken into account during comparison of different service discovery protocols. As a researcher, we want to use *ESKEm* framework to explore given SDM in a given environment. Put it in other words, we want to study the influence of different parameters onto performance of the SDM. This groups of parameters we have just defined. As a result, there should be the way how the researcher (see also Section 2.1 with scenarios) could control mentioned parameters in the *ESKEm* framework. So there are two questions which are interesting for us in this context. *Which* parameters should be controlled and *How* should they be controlled. In this section we would like to address first question, we will come to the second question in Chapter 5.

To the system counts the parameters like speed of CPUs of devices where the service/client and other participating in the service discovery components are located, different parameters of the network system, the speed of I/O operations on devices and many others. In general, we assume that the parameters are interesting which significantly affect (either increase of decrease) the SDM performance. It is hard to predict which system parameters may be of interest for a researcher. So we decided in first turn to concentrate on the parameters of the networking layers.

We selected the simulation approach to mimic network layers. This has an additional advantage that the parameters of the simulated network layers can be changed easily. But which parameters should we be able to manipulate? As a result of the analysis from the Section 4.2.2 we can conclude that decision depends on the *abstraction layer* which will be used as an entry point to the simulation facility. For example, if the entry point into the simulation is at the Link Layer, the parameters affecting performance are bandwidth limitation (including frame serialization delay emerging from a limited bandwidth), propagation delay (usually fixed), medium access delay (may very from frame to frame), frame loss (corruption loss) according to [Herr 02]. [CaSa] mention also duplication of packets. In general, it is easier to control the parameters at the lower network layers because less of them should be taken into account to achieve realistic simulation (the same apply also to the emulation).

Which *workload* parameters should we able to control? In the Section 3.6 we have concluded with the set of services and clients activities which is a workload for the service discovery protocols. Both services and clients can be *started* or *stopped*, the client can additionally send the *search requests*, the service can *advertise* themselves. Which workload parameters should we be able to control? Again, the answer to the question depends on the research goals. Here are the examples of them:

- the time between successive service starts (or client start, or time between successive discovery requests etc.)
- location of the services/clients on the network
- number and sizes of the service discovery request parameters
- number and size of the service attributes (compare with the row ”service description” in the Table 3.5) - increased number of attributes may result in the increased load onto the SDM.

There is also third set of parameters which we may wish to control. This are the characteristics of the service discovery middleware by itself. For example, the SLP can work either with DA (directory agent which is responsible for lookup services) or without it. This may affect performance of the discovery protocol significantly.

4.4 Visualization

It is not so simple to analyze the behavior of a complex system involved in large scale networks providing only tables of summarize performance numbers or some plots. From the simulation packages considered in the previous parts of the current chapter the NS2 and DIANEmu simulators provide the visualization interfaces.

The motivation for the section is the following. First, there is explicit Requirement for a visualization facility in the *ESKEm* (see Requirement B.4). Second, we will consider some visualization related approaches/frameworks here which do not relate to already mentioned network simulators mentioned earlier. So we decided to consider in the current section also the approaches to the visualization which are parts of the network simulators mentioned earlier in this chapter like visualization for the NS2.

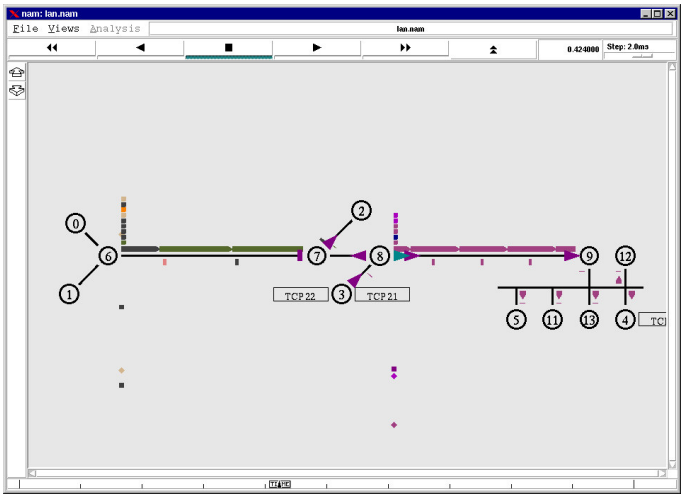


Figure 4.13: Nam: Network Animator.

NS2/Nam Nam is a Tcl/TK based animation tool for viewing network simulation traces and real world packet trace data [NS2Nam]. It supports topology layout, packet level animation, and various data inspection tools. The first step to use *nam* is to produce the trace file. In the next step this file is used by *nam* for visualization. It is not possible to control simulation parameters with *nam*. But it is possible to come back in the animation, or control the animation speed. Only visualization of transport and lower network layers specific information is possible without extending *nam*.

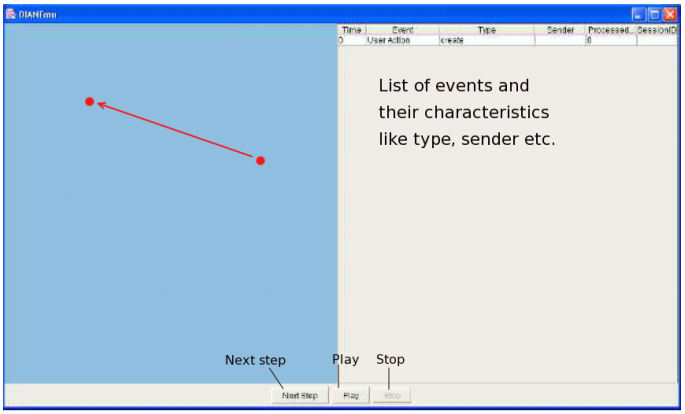


Figure 4.14: Visualization in DIANEmu. The arrow could mean a ping message from one node to another.

DIANEmu As a NS2/Nam, the DIANEmu propose the graph-based view onto the simulation. The service or clients are represented as nodes, the graph is not connected in initial state. The DIANEmu visualization interface is tightly coupled with other modules and components of the DIANEmu like event scheduler. We assume that the DIANEmu visualization of larger networks (for example, more than 20 nodes) will be rather complex to the user .

Left side of the DIANEmu visualization frame contains nodes. The nodes are not connected as far as DIANEmu is intended for simulation of ad-hoc networks. The communication between nodes is visualized as the arrow which disappear once showed. On the left side the list of scheduled events is represented. The event is discarded from the list as soon as it is accomplished. It is possible to control the simulation like to let the simulation progression step-by-step.

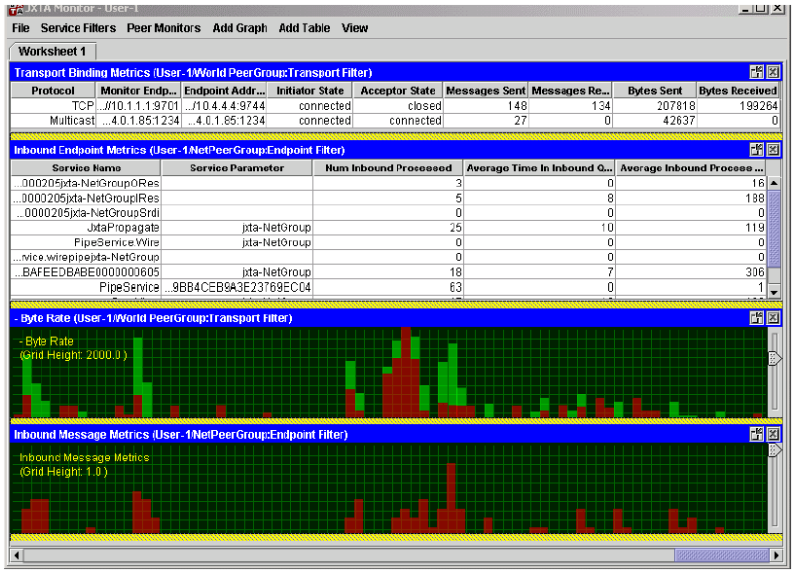


Figure 4.15: Visualization in JXTA peer-to-peer platform.

Visualization in JXTA We decided to mention the JXTA platform because of its visualization features. The JXTA is a platform which enable development of peer-to-peer (P2P) networks [JXTA b]. The P2P network should be considered as The JXTA-Monitor project goal was to develop the monitoring tool, ”that captures messages between JXTA Peers” [JXTA d].

It is intuitive to visualize the network as a graph, where the graph nodes are the network components and the edges represent either connection or communication between network components. Different metrics can than be associated with edges or nodes.

Let us consider the visualization of the communication. The edge could be marked with a value which should represent the number of bytes sent over the link. We could call such kind of visualization as the ”point-in-time visualization”. The *history* of the communication between nodes would be lost in this case. The alternative approach to visualization would be the ”over-time visualization”. The JXTA visualization tool on the Figure 4.15 can serve as one of the over-time visualization examples as we mean it. The user can select a communication link between two peers in upper part of the GUI. The lower part of the GUI visualizes the distribution of the sent bytes for the link over time.

Visualization of graphs The simulator specific visualization tools which we have found and considered did not match our criteria to the visualization tool. The DIANEmu visualization seems to be only good for small networks. Additionally, it is strongly coupled with the rest of the DIANEmu modules. The Requirement B.4b to the visualization module is very likely to fail. The NS2Nam is targeted to visualization of the processes at transport and lower network layers.

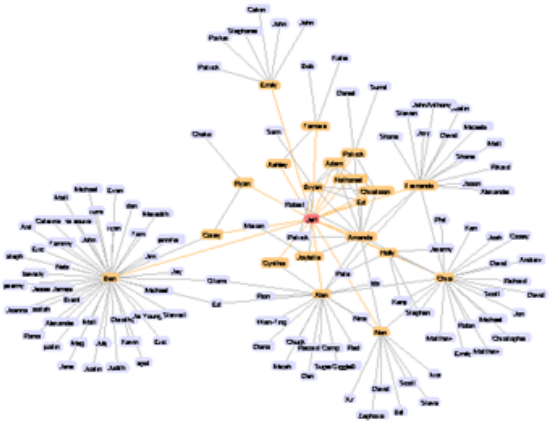


Figure 4.16: Prefuse: a user interface toolkit for building highly interactive visualizations of structured and unstructured data, [Prefuse].

This considerations motivated to develop own visualization module. We decided to based it on the framework which enable visualization of graph-like structures. After consideration of several frameworks for this purpose we stopped on the *Prefuse*[Prefuse]. The authors of the Prefuse define it as ”a user interface toolkit for building highly interactive visualizations of structured and unstructured data”. We were interested in the support of Prefuse for visualization different kinds of networks. The nodes and edges of the visualized by the Prefuse graphs can be either read from an XML file or be created dynamically. The Prefuse is built using Java2D graphics library and claims to use architectural techniques for scalability. The user is also free to customize different aspects of the Prefuse appearance or functionality.

4.5 Summary

In the previous chapter we suggested different modules that make up the *ESKEm* framework. We identified the networking, measurement, control and visualization modules. We made an overview of the work which is related to one of the mentioned modules.

First, we considered different approaches to mimic a networking system and its characteristics. We concentrated on the emulation and simulation approaches as the most appropriate candidates to be used in the *ESKEm.* As a result, we decided to build our framework over the *JiST/SWANS* network simulator. Our decision is summarized in the Section 4.2.3.

Second, we introduced the concept of *measurement*. We identified different categories of metrics which are usually used during service discovery researches. These are *speed*, *reliability* and *power/resource consumption*. The metrics could also be grouped based on the criteria of being *individual* or *group* metric.

Third, we considered two sets of parameters which the researcher might wish to control when exploring a service discovery protocol. These two sets of parameters influence the performance of the service discovery middleware. We gave examples of the networking parameters which are expected to be interesting for the researcher. We decided to provide the means to control for these parameters in the *ESKEm.* Another set of parameters are the workload parameters. For example, the location of the services on the network is one of the workload parameters which the researcher may wish to control in order to explore its influence on the service discovery performance.

Finally, we shortly considered several approaches to the visualization of the network simulation. The visualization modules of DIANEmu and NS2Nam failed to meet our requirements. We therefore decided to explore frameworks which would be suitable for visualization of network structures. We identified the *Prefuse* as a promising tool and we will use it later in the *ESKEm.* The *Prefuse* should be suitable for visualization of large networked structures and has been characterized by authors as scalable.

We will use the experience from the previous chapter during the design of the *ESKEm* in the next chapter.

Chapter 5

Design of the *ESKEm* framework

We implemented the *ESKEm* based on the design considerations from the previous chapter. In the current chapter we are ready to consider the design of the *ESKEm* framework in more detail. In fact, we have already started with the design aspects of the framework in the previous chapters. The identification of the most important modules for the *ESKEm* was introduced in Chapter 3 as a result of analyzing different service discovery protocols. It was continued in Chapter 4 where we identified the complete set of modules, that are required to build up the *ESKEm* (see Figure 4.1). Furthermore, we selected the frameworks which could be reused in the *ESKEm.*

But even if the realization of some modules can be supported by the reuse of the existing code, there are still open question. Not all requirements are met by the identified frameworks. How can existing frameworks be extended or modified to meet our requirements to the *ESKEm*? What are the additional components, which must be developed for the *ESKEm* and how should they be designed? What are the interfaces between different modules and how can the modules be integrated together in the *ESKEm*? These are the most important questions which we would like to consider in this chapter.

The chapter is structured that as follows. We start with an overview of the *ESKEm* modules and components in Section 5.1. This overview concentrates on the identification of questions which should be addressed by the design. The questions will be addressed in the subsequent sections.

5.1 Design Overview

For the purpose of the current work we divide the design related questions into two groups. The first group of questions deals with internal structure of the modules and their tasks. The second one addresses the questions, which are related to the interfaces between different modules. We mapped these questions on Figure 5.1, representing the overall design of the *ESKEm.* The questions will be introduced and addressed in the subsequent sections.

The components on Figure 5.1 are drawn either without or with the use of a specific filling pattern. The area filled with a pattern reflects the fact that the component was modified. The components filled completely with the pattern, like Measurement and Control, were designed and developed from scratch.

The motivation for the design as represented on Figure 5.1 was to great extent already given in Chapter 3 where different service discovery protocols were introduced. The service discovery protocol can be considered as a component under study (CUS). It is referred to as the system under test (SUT).

There are different components which belong to the SUT. One of them is the Service Manager and/or Service User components used to generate workload that forces the CUS to do some useful work, namely service discovery. As a result, we are able to measure the performance of the CUS with the support of

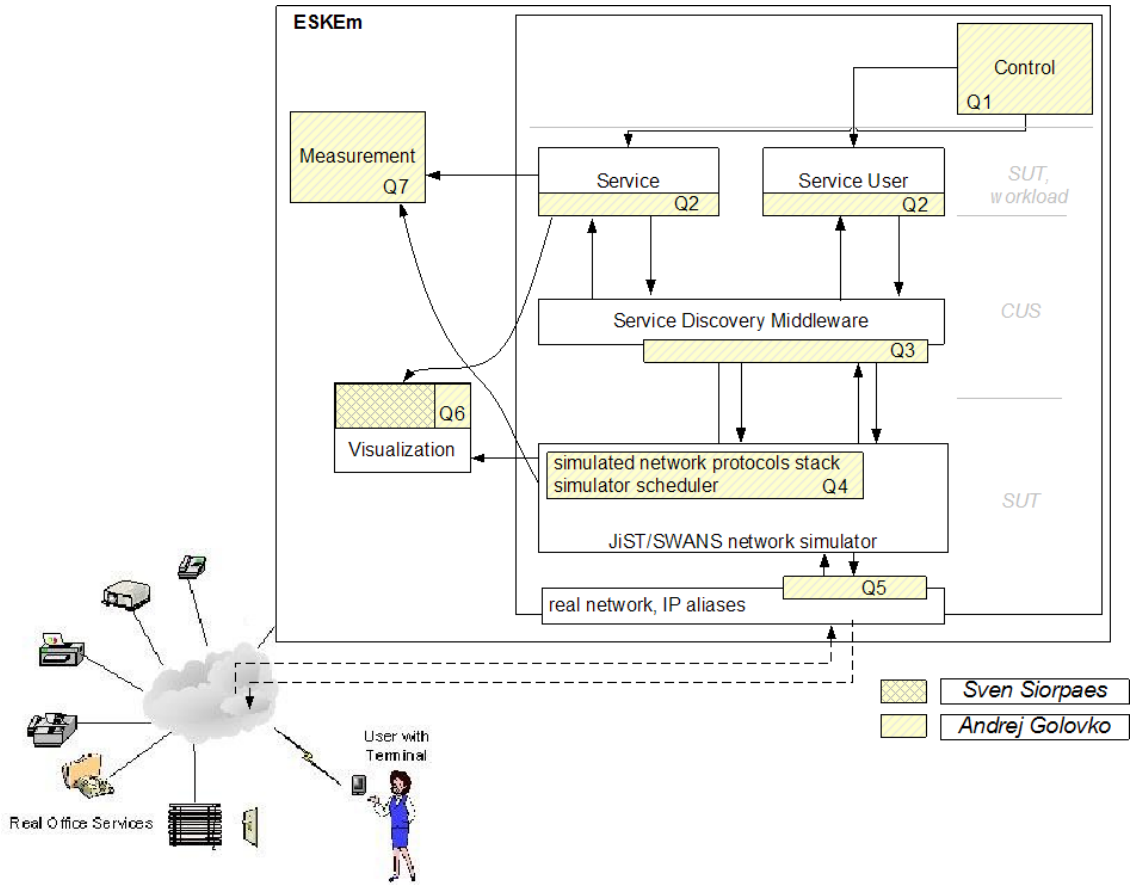


Figure 5.1: The *ESKEm* design and related questions (see the text for the explanation).

the appropriate Measurement infrastructure. Note, that both workload and measurement are applied at the level of SUT. We introduce the Control module, which cares about the management of the emulated Service Managers and Users. Closer discussion about this set of components of the *ESKEm* and questions *Q1* and *Q2* is given in Section 5.2.

The network system is another component of the SUT. The service discovery protocols[1] are usually the application layer protocol in terms of the OSI Reference Model. The network characteristics are an important factor influencing the performance of the service discovery protocol. We decided to introduce it into the *ESKEm*. We selected a network simulator used to mimic the network system. What should be modified in the network simulator (question *Q4*)? How do we communicate with real external clients or services (question *Q5*)? How can a real Service Discovery Middleware communicate over the simulated network (question *Q3*)? These are key problems considered in Section 5.3.

A discussion about the design of the Measurement and Visualization infrastructures is offered in Section 5.4 and 5.5 respectively.

5.2 Workload Components

The Service Manager and Service User are the components which generate the workload resulting in the activity of the service discovery middleware. We also often refer to the Service Manager as to the *service*

[1]The service discovery protocol (SDP) of the Bluetooth is not an application layer protocol.

or *device*. A device is often associated with several services. The Service User is referred to as *client*. There are several questions, associated with these components (questions Q2 on Figure 5.1). For reasons of simplicity we will usually mean both services and clients[2], when using either term: ”service” or ”client”.

Q2.1 How is it possible to emulate an (internal) *ESKEm* service in a way that it is discovered by an external real client as if it were a real service?

As a result of considering different service discovery protocols in Chapter 3, the Generic Model of the service discovery was derived. The *Service Manager* (SM) was identified as a component, which acts on behalf of service. The same applies to the *Service User* (SU). Some service discovery protocols require only the existence of the *SM* for the service to be discoverable. This means that the service might not even exist or might be down but still be discoverable by a potential client. The question which arises is how can we make the client think that a service is there.

The information about services is delivered to the client in the form of messages. On the other hand, the messages like search requests from the clients should be answered by the services or lookup tables appropriately. We are therefore only expected to generate appropriate messages and answers to messages from clients in order to mimic the existence of a service. It is important to note, that as a consequence of the *ESKEm* requirements we are only imitating the service in respect to the *service discovery* and not in respect to the *service invocation* or something else. The behavior of the system is undefined if an external client decides to invoke some functionality of an emulated service.

As a consequence, we are not expected to provide any implementation of the service functionality. It may significantly save the consumption of resources when emulating the service managers.

The next consequence is that we must implement the complete service discovery protocol in order to emulate the services.

Q2.2 What kind of client/service behavior should be reproduced in order to generate a workload on the service discovery middleware?

According to Chapter 3, there are different activities which relate to the process of the service discovery. Services may send advertisements. Clients may send service discovery requests either to the lookup table or to all devices available in the network. The type of messages sent is the service discovery protocol dependant. The same applies to the number of the messages sent, the interval between messages etc.

Nevertheless, the behavior of both clients and services can be abstracted to the *start* and *stop* of the devices, on which the clients/services are located (see also Section 3.6). For example, the start of the device with services is associated with the advertisement messages sent to the network.

As a result, we find it reasonable to associate the service discovery activities of both clients and services with the *start* and *stop* activities of the device.

Q2.3 Should the Service Mangers/Users be developed from scratch? How would it be possible to integrate the existing services into the *ESKEm* for the purpose of services emulation?

In the worst case, we should implement the behavior of the Service Manager so that it appears to an external client as if it were a real service. But as we saw in Chapter 3 (see UPnP specification and especially its reference implementation), basic implementation of services is often available. For example, these are abstract classes which can be extended with an additional functionality in order to develop customized services. The abstract classes already know how to use the specific service discovery protocol.

We could reuse an existing service implementation. We believe that the resulting Service Manger and Service User component can be created with less effort and with a higher accuracy of the emulation. We *adapt* the real services implementation.

[2] the same applies when referring to a user as *he* (or him) - we actually mean ”he/she”(or ”his/her”).

How could we reuse the existing services into *ESKEm*? There are two problems that we would like to address in this context.

First, we possess the means for instantiation of services and for controlling its behavior. As a consequence from Question 2.2, *start* and *stop* of the device are the most important aspects of the device's life cycle, which we would like to control. The technical side of the question will be covered in the next chapter.

Second, we wish to control the messages transfer between the adapted services. There are several reasons, why this may be beneficial for us. For example, it would be possible to block the message from/to the service to imitate. This would save the resources required for true stopping of the service. The messages could also be analyzed for the purpose of the metrics measurement on their way to other devices.

How can we intercept the messages passed between different devices? A similar question was addressed in Section 4.2.1, where we talked about NS2 emulation interface. Section 4.2.2, dedicated to the network emulation approaches also discussed this question. We decided to intercept the messages related to the service discovery protocol before they enter the transport layer, the 4th Layer. We return to the question of message capturing in Section 5.3.

We assume, that the Service Manager or Service User should be responsible for the adaptation of services. The classes are also called *adapters* of a service implementation.

Q1.1 How can the reproducibility of the emulation scenarios be achieved (see also Requirement B.3)?

The emulation scenario can be described and made persistent in the so called *emulation scenario scripts* or simply *emulation scripts*. The approach is usually used in the simulators. For example, the scripts in the NS2 network simulator guarantee reproducibility of the network simulations. The scripts are written in the Tcl/Tk scripting language. The scripting language of the NS2 and most of the other network simulators concentrates mainly on the first four network layers. The scripting language must be extended by additional commands, if we want to describe the simulation at the level of the application layer. The selected network simulator *JiST/SWANS* does not have its own scripting language. As a consequence, any selected scripting language would result in the overhead associated with the interpretation of the language. As a result, we decided to describe the emulation scenarios as a list of commands with a set of associated parameters. Each such command leads either to the start of the service/client, or to its stop. The commands have at least two parameters. First is the ID of the emulation service or client, to which the command should be applied. Second is the time of the command invocation relative to the beginning of the emulation. The emulation script should be generic and independent of the service discovery protocols.

Additional files may be needed for the configuration and customization of the emulation scenarios. These files should reflect the differences between different service discovery protocols. For example, the UPnP Service Manager needs to know, where the service description file is located (see also Chapter 3). On the other hand, the Service Manager of the SLP requires a list of attributes to describe (and eventually advertise) the service.

To sum up, there are at least two types of script files required to describe a single emulation scenario:

general file defines commands and their parameters independent of the service discover technology. The command name, the IP address of the service device are candidates for this file.

middleware specific file contains parameters for each service, which are service discovery middleware specific.

The commands read from the emulation scripts are instantiated to the *actions*. Each action results in some kind of a Service Manager or Service User activity. The *actions* are queued into the scheduler. The main characteristic of this scheduler is that it is a real time scheduler. This means that the scheduled actions are executed in the real time. Our scheduler runs during the duration of the emulation and is ready to react to the *actions*, delivered to it over a network.

We decided to execute the *actions* in concurrent threads. This results in faster processing of the queued in the scheduler actions, because no time is ”wasted” by the scheduler’s thread for the complete execution of the action. The second consequence is that it may result in the non-deterministic behavior of the emulation. The order of the started services can be different for the same emulation script. Nevertheless, we decided to parallelize the execution of the actions because the performance of the emulator is more important for us than its determinisms.

5.3 Network Simulation Component

The selected in Chapter 4 *JiST/SWANS* network simulator (see also Section 4.2.1) can not be taken ”as it is” without any changes for the purposes of the *ESKEm.* There are a number of fundamental questions (see questions Q3, Q4 and Q5 on Figure 5.1), which must be addressed before building the *ESKEm* over the *JiST/SWANS.* The Q3 questions are related to the problem of the interfaces between a real service discovery protocol implementation and a network simulator. The Q4 represents the questions which deal with the modification of the network simulator and the reasons for the modification. The questions marked as Q5 consider the problem of integrating the *ESKEm* emulation environment into an environment with real clients.

Q3.1 How can the messages be captured on their way from some arbitrary service A to the service B (see Figure 5.1)?

This question provokes a critical reader to ask other questions. Why do we need the message capturing in the case of the *ESKEm* framework? The motivation for the message capturing is twofold.

First, it is motivated by the consideration, which is the result of the discussion question Q2.3. The service discovery protocol operates at the level of the application layer in terms of both the OSI and TCP/IP Reference Models. The messages from the service A are passed through the other network layers to it counterpart on the side of the client B. The message capturing makes only sense in the case when we want to have control over the communication between adapted services/clients. Otherwise, the Service Manager or Service Client imitate the service discovery behavior as a result of our efforts and the message capturing is not actually necessary as we already have full control over the Service Manager or Service Client.

Second, we consider the *message capturing* as a way of providing the interface between a real service discovery code and the ”faked” network transport layers, imitated by the network simulator. The problem is not new and was mentioned in the papers, which try to integrate a network simulator in a real environment[3], or where network emulation use capturing to modify the traffic characteristics (see Section 5.3).

Section 4.2.2 introduces the terminology (see Figures 4.9 and 4.10) related to the emulation of the network environments (the ”emulated network” on Figure 4.9). Despite our the decision to use a simulation approach to reproduce a network, the terminology just referenced will be reused. Indeed, in the case of the network simulators we have a ”simulated network” (compare with Figure 4.9), which has a ”simulation interface” providing the ”simulation abstraction layers” with ”entry points” (compare also with Figure 4.10).

The approaches to the message capturing differ mainly in respect to the simulation abstraction layer, at which the messages are captured. This difference influences the technical aspects of the message capturing realization. We should also actually use different terminology, when speaking about capturing at different simulation abstraction layers. So, the *messages* capturing relates to the interception of messages at the level of the transport (4th) layer. The *packets* emphasize that the capturing is done at the network (3d) layer.

The examples of different approaches to the packets capturing were introduced in Chapter 4. Mentioned in Section 4.2.1 NS2 emulation interface provides a collection of objects including so called *tap agents* and *network objects* for transition of packets between the simulator and live network. Tap agents embed live

[3] see NS2 emulation interface as example, Section 4.2.1

network data into objects used for the packets representation in the simulation and vice-versa. Network objects are installed in tap agents and provide an entry point at a particular protocol layer for the sending and receipt of live data [NS2Emu]. The capturing is done with use of the LBNL packet capture library (libpcap), so the packets are captured at the system level in the kernel mode of the Unix-based OS. There are also other approaches (compare Dummynet tool [Rizz 97] and how IPTABLES module is used to capture packets). An interesting approach, called *virtual router method*, is described in [Russ 00]. The main idea is to enable communication between hosts A and B over the host E with installed emulator, which captures the packets and modify appropriately before sending them further to B.

The observation of this and other examples has lead us to the several observations, which are useful in the context of the *ESKEm* design. With the word ”layer” we refer to the ”network layer”.

First (as was already discussed in Section 4.2.2), the simulation at the lower layers (with the real implementation taken for the upper layers) is usually associated with the higher accuracy of the simulation as far as less factors should be taken into account during simulation.

Second, the simulation gives us more flexibility in reproducing different network protocols stacks for different devices. According to our design, no real implementation of the protocol is required at the *simulation abstraction layer* and lower levels. On the other hand, we assume that the real implementations are used for the protocols at the layers above it (especially service discovery protocol by itself). As a conclusion, the *entry points* at the lower layers results in less flexibility of the network simulation. This also results in less flexibility to reproduce heterogeneous networks. So we conclude further, that providing *entry points* at the lower *simulation abstraction layer* is in contradiction with the Requirement B.5.

Third, a real protocol implementation use real resources like memory, CPU resources. It may also require availability of the specific hardware resources. As the result, the costs for the emulation will grow significantly with moderate increase in the size of the emulated network (see [ZhNi 03] for the performance measurement of the EMPOWER emulation framework). The simulation is much more appropriate for experimentation with large and complex networks. As a consequence we assume that providing the *entry points* into simulation facility at the higher layers (and simulating at as much networking layers as possible) is more beneficial for us.

And forth, providing the *entry points* at the 3d (network) and lower layers are usually associated with the modification of the system libraries (compare to Dummynet, NISTNet in Section). This is in contradiction to the Requirement C.3.

As a result of this consideration we decided to intercept the messages related to the service discovery protocol before they enter the transport layer (the 4th Layer). The technical realization of this design choice belongs to the implementation. Nevertheless, we would like to consider some general suggestions already in the design section. So the next question.

Q3.2 How can be the message capturing at the 4th Layer realized?

Two alternatives were considered to address the question. We assume, that these alternatives are independent of the specific Transport (4th) Layer implementation. Nevertheless we will discuss the alternatives in the context of Sun’s Java platform. It should simplify the explanation.

The bottom four layers can be seen as the transport service provider, whereas the upper layer(s) are the transport service user(s). The transport services are provided in the form of function calls. For example, the `java.net` package contains classes like `DatagramSocket`, which can be used by the application to send the messages over the network.

One approach would be to replace the calls to the `java.net` classes with the calls to custom classes, which take care of the captured messages. The result is that the *Service Discovery Middleware* component on Figure 5.1 must be modified in order to use custom functions calls. For example, the calls to `(DatagramSocket)dgmSocket.send(myMessage)` would be changed to the calls like `NetAdapter.send(myMessage)`.

Another approach would be to change the semantic of the transport layer functions. For implementation in Java it would mean, that the semantic of appropriate `java.net` classes should be changed. For example, the changed classes may collaborate with appropriate *Service Manager* classes to decide, whether the message should be sent further or blocked. As a consequence, the signature of the methods used by the *Service Discovery Middleware* is left intact. No modification of the *Service Discovery Middleware* is required.

We chose the first approach, as the `java.net` need not be modified. The modification of the original `java.net` could have unexpected side effects. Further, we used the source code (Java language) of the *Service Discovery Middleware* as a reference implementation. The original calls to the transport layer are replaced with the calls to the class(es), which we name the `NetAdapter`.

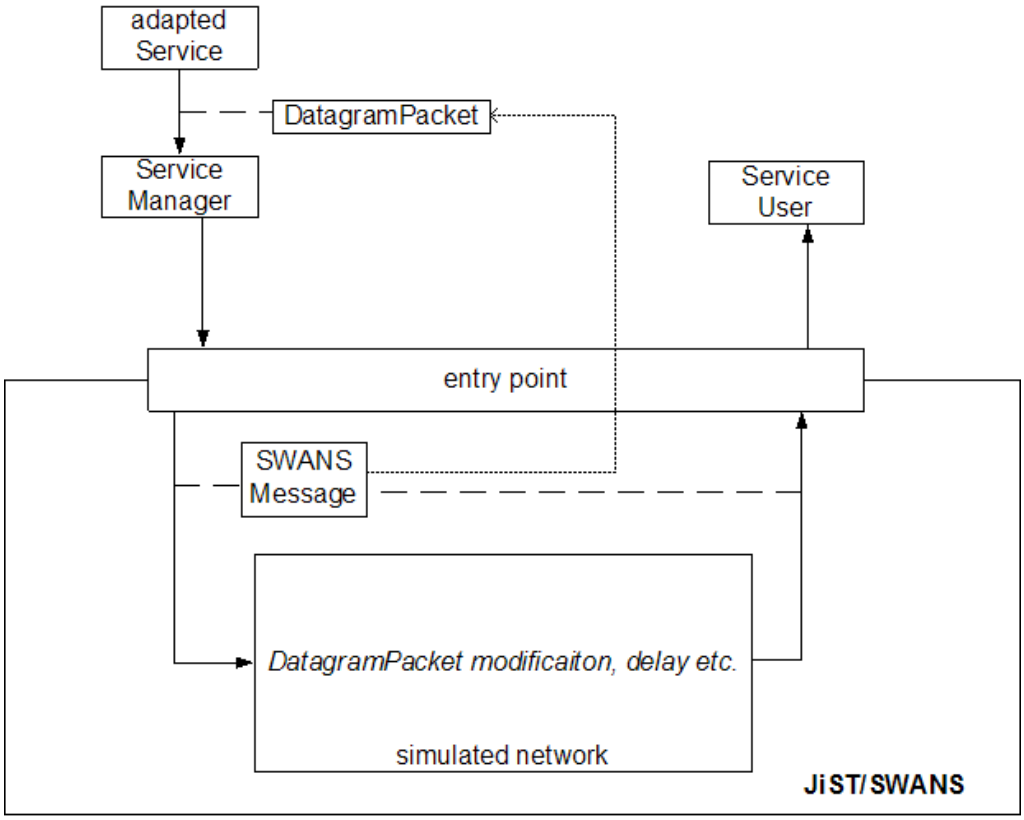


Figure 5.2: Real packets (here, Sun's Java `DatagramPacket`) are passed through *JiST/SWANS* over `NetAdapter` interface.

The captured service discovery messages of the adapted service are passed to the Service Manager. How should we process the message further? The idea is to let the message flow from source A to destination B over the simulated network. An example of the approach is given on Figure 5.2. The messages of the adapter service (encapsulated into the Sun's Java `DatagramPacket` if sent over the connection-less transport) are captured and passed to the *Service Manager*. They are further passed to the entry point of the network simulator (*JiST/SWANS* in our case). For every `DatagramPacket` the *SWANS Message* is created as its representative in the simulated network. The Figure 5.2 shows that the *SWANS Message* knows its `DatagramPacket`. The *SWANS Message* is received by the target *Service User* after the effect of the network characteristics had been calculated for the message transfer. The *entry point* detaches the *DatagramPacket* and passes it further to the *Service User*.

There are two modes in which the *DatagramPacket* is passed through the simulated network (compare the modes used by the NS2 emulation interface, Section 4.2.1). In the *opaque* mode the *DatagramPacket* is

not modified and only its transfer characteristic (like, packet delay) are changed. In the *protocol* mode the packet may be modified. For example, the specific flags in the message header can be set or an error can be intentionally introduced in the packet.

Curious readers may already asked how the calculations of the *SWANS Message* delay are applied to the real packets. Usually, the *calculation* of the delay is significantly less, than the *delay* itself. Technical details like this one will be covered in the context of the implementation (see Section 6).

Q4.1 The *SWANS* is a simulator for the ad hoc networks. How can we extend it so that LANs can be simulated too? (see Section 2.2 for a motivation behind this question).

JiST is a general purpose simulation framework and SWANS was initially designed for the simulation of the wireless mobile ad hoc networks. This results into the fact that the simulation of the LANs is missed in the SWANS. Our need for the LAN simulation is a consequence of the office environment definition given in Section 2.2.

The design of the *SWANS* implies, that new simulated network protocols can be added as a result of implementing appropriate interfaces, provided by the *SWANS* (see also Section 4.2.1). Figure 5.3 represents the network layers at which LAN specific protocols should be introduced. You may wish to compare it with Figure 4.5 where original SWANS protocols stack is introduced.

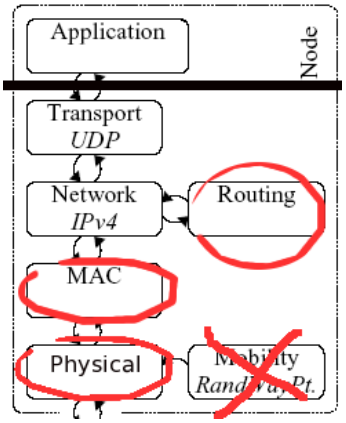


Figure 5.3: The modification of the original protocols stack of the SWANS.

It is usually easier to simulate a network protocol than to implement it. But an accurate protocol simulation is still not a trivial task (it is good analyzed in the [Bjoe 05]). The focus of the current work is not on the simulation of the network protocols at the lower layers, we therefor decided to oversimplify the simulation of protocols as shown on Figure 5.3. Only the most interesting aspects of the protocol functionality are taken into account.

Which parameters of the LAN should the user be able to manipulate during the simulation? We approach this question according to the steps defined in a generic model of the *network experiment* defined in the [BiHe et al. 04] (see also Figure 5.4). We have already considered some phases of the *network experiment*, shown on Figure. We already discussed the question (for example, question Q1.1) of the *Application Level Traffic Generation*. The *Topology Generation* and *Node & Link Properties* are the characteristics of the network, which should be addressed by the network simulator.

The *Topology Generation* assumes that a (physical) topology of the network, on which the Routing protocols will operate, should be specified, . Precisely speaking, we are interested in the topology of the routers, to which other devices (with services and/or clients) are connected. The topology can be naturally expressed as a graph. The word *generation* implies that such a graph can be generated by some external tool. To make things easier, we do not consider different topology generation tools. Instead, we specify the

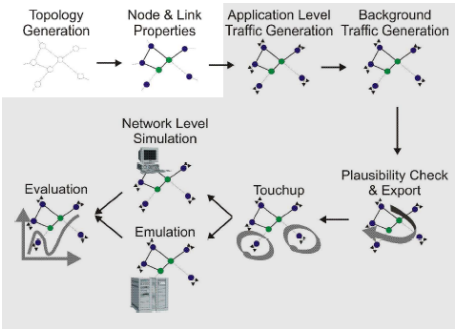


Figure 5.4: The steps of a generic network experiment, as found in [BiHe et al. 04]

topology of the routers in the *flat file* manually. It is important for us that the specific characteristics of the router can be specified too. We are mainly interested whether the router is a multicasting router or not (see Technical Background in Section 3.1). With the evolution of the *ESKEm* it may be required to add new router characteristics. The network topology is specified in a flat file together with its characteristics. This topology is an *initial topology* which may change over time. For example, the router may be turned off etc. We account for the number of hops passed by the message on its way between devices A and B.

Further, we are interested in the *Link Properties* of a simulated network. We account for packet propagation delay and packets loss on the link. The simulation of the characteristics like bandwidth limitation or medium access delay (see Section 4.3.1) were not taken into account for the sake of simplicity.

As a consequence, the simulation of the network system is rather trivial in the case of the *ESKEm.* Further development of the emulation framework may require improvements of the LAN's simulation. It may also be interesting for the users of the framework to introduce the simulation of other protocol stacks. For example, Bluetooth protocol stack is quite common in the modern environments.

Q4.2 How should the simulation kernel be modified in order to work with the emulation?

The question is actually of a generic nature and is not specific to the *ESKEm.* For example, it was addressed in the design of the NS2 emulation interface. It can be reformulated as following: "How can the simulation and emulation work together?" We explain the problem by using the example of the *JiST/SWANS.*

As already mentioned in Section 4.2.1, the *JiST* is a discrete event-based simulation framework. Events are processed as soon as the effects of all prior events are evaluated. The simulation runs in a so called *simulative time*. The emulation, on the other hand, runs in a *real executive time* (see Section 4.2.1). As a result, we need to synchronize these times, if an emulation and simulation are to work together.

An example should help clarify this problem. The simulators can work with network models, and the simulation kernels can compute the effects which specified network properties have on the traffic, traversing the model. Assume that such effect should be calculated for a message transferred from A to B. Let us assume, that the message delay was calculated to be 900ms. Let's further assume, that the calculation by itself was accomplished in 300ms. How should the packet be "delayed" before delivered to B? The *Dummynet* solves the problem by introduction of *queues*, where the messages are delayed artificially. Thus B is guaranteed to receive the packet with a 900ms delay. But the *Dummynet* deals with a single link between two nodes. In the case of the *ESKEm*, the communication over multiple links is simulated concurrently. Our solution is rather technical and it will be explained in Section dealing with implementation details.

Further, what if higher bandwidth were simulated? Indeed, if the simulation in previous example takes 300ms, but the calculated by the simulation delay is 100ms, it means, that the simulation "took too much time" and we are too late delivering the message to the destination. The approaches to this problem differ. The NS2 emulation [NS2Emu] just inform the user about a failure to be in pace with real time. The user is provided with statistical data so that the user can decide whether the simulation was really "too much"

behind the real executive time. The other approaches aim explicitly at the low bandwidth links emulation [DBCF 95]. But the parallelization of the discrete event simulator can be a solution to the problem to some extent too [SiUn]. According to the performance measurement of the *JiST/SWANS* and its support to run the simulation in the distributed mode (see Section 4.2.1) we believe that the framework can address the mentioned problem well.

And now the last but not least we would like to mention the following problem here. There are two schedulers in the *ESKEm*. A real time scheduler was introduced when discussing Question Q1.1. It is responsible for scheduling and execution of the *emulation commands* from the emulation scenario script in a *real executive time*. The timestamp of each command defines when it should be invoked relative to the beginning of the emulation. Another scheduler belongs to the simulation kernel of *JiST* and executes the events in a *simulative time*. The problem is explained through an example. Let us assume that the last event was executed at the simulative time equal to 10 second. What if a new service A is started by the real time queue at the 20th second? The simulation of the message transfer should be started on the 20th second of the simulative time, not on the 10th second. We approach the problem by *advancing the simulative time* from within the real time queue as soon as new *action* is executed in the real executive time. Both times should be synchronized. Additionally, we modify the *JiST* scheduler so that it can run as long as the emulation runs and is ready to start the simulation of the packet transfer (or similar events) at any time.

Q5 How can the emulated services communicate with the real clients?

According to the Requirement B.6, the emulated services should be discoverable by the external clients, without modifying the external clients' implementation. As we already mentioned, the appropriate messages about availability of the emulated service must be deliverable to the external clients. On the other hand, the external clients may require contacting (emulated) services. For example, the UPnP clients should be able to contact the services to read their description (`description.xml` is disposed to the clients with a *web server*, available on the service's device).

The address of the *Service Manager* plays a crucial role in the service discovery. The communication over the network is accomplished over NIC (network interface card). The number of NICs on a single host is limited and in the case of the *ESKEm* the number of the emulated devices will be significantly larger than the number of NICs available on the host with the *ESKEm*. A naive approach to the problem would be to emulate multiple NICs on the host with a single NIC, so that each emulated device can be *bound* to its own NIC. Indeed, the approach would work and the *NISTNet* could be used for the network cards emulation. However, it would require additional effort for the configuration of the emulated network cards. We also assume that the emulation of NICs would consume more system resources as the approach, which we actually selected.

Our consideration was that the IP address of the device is important to distinguish services on the different devices. Each device is assigned at least one IP address. In other words, each device must posses an IP address. We found that it is much easier to define an *alias interface* with assigned IP address. As a consequence, the devices are bound to the alias IP address (see Figure 5.5), the packets from the device are stamped with the correct source IP address and external clients can contact the emulated services by sending messages to the alias IP address.

The IP aliasing is supported at least by Linux and Windows XP operational system. In Linux environment an aliased IP addresses can be assigned to the network adapter with `ifconfig` command like `ifconfig eth:3 192.168.153.21 up`.

To summarize, the problem was to guarantee that each device has at least one IP address, which would distinguish it from another devices. We considered two alternatives and selected one of them which meets our requirement and is easier to implement. More details involved into emulation of the services to the external client in respect to service discovery are considered in the subsequent chapter dealing with implementation.

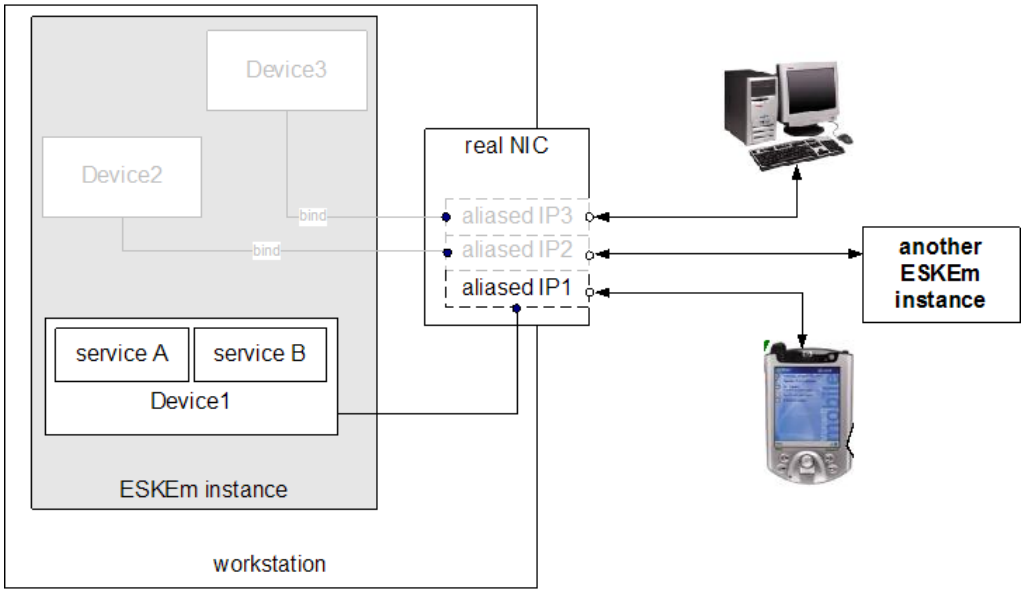


Figure 5.5: With aliases every emulated device (and as a consequence, service) has own IP address, which can be used by external clients and services.

5.4 Measurement Component

The monitoring and measurement of emulation framework activity is accomplished with `Announcements` and `Gauges`. The names for these components were motivated by the [DIANEmu 04].

The approach with gauges and announcements works as following. `Gauge` is a component, which can listen for interesting events and processes them, also for metrics calculation. ”interesting events” are delivered to the `Gauge` in form of the `Announcements`. For example, the `MessageSentAnnouncement` and `MessageReceivedAnnouncement` are announcements, which can be used to calculate the ratio of successfully sent messages, average delay for the messages etc. The configuration file should be used to register specific announcements to be listened by specific gauges. The `Gauger` is the component, which dispatches announcements to the gauges interested in them.

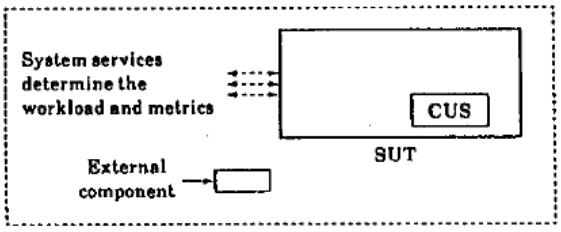


Figure 5.6: The System Under Test (SUT) and The Component Under Study (CUS) representation as defined in [Jain 91].

We already introduced the concept of the SUT (System Under Test) and CUS (Component Under Study). In our case, the CUS is the service discovery middleware (SDM), which is explored in an environment reproduced with the *ESKEm* (see Figure 5.7). The measurement is accomplished at the level of SUT [Jain 91]. The SDM is in the ”middle” between SU/SM and networking layers. Both of them are considered as the layers, at which we may wish to measure the metrics. It is up to the user of the *ESKEm* to decide about measured metrics and the level where it should be done. The introduced concept of gauges and announcements (GaA) should be applicable to the measurements at both levels.

The concept of GaA can be extended even further. One of the directions for the extension is to enable gauges to run on the workstation, which is different from the workstation where the main emulation takes place. This extension would be especially reasonable to meet Requirement B.8, if we decide to calculate metrics in real-time.

The concept of GaA is very flexible. We provide a simple example. Assume the gauge G1 is interested in announcements of types A1 and B1. These are used to calculate some metric M1. Then assume, that there is the gauge G2, which would be interested in using the metric M1 to calculate some additional metric M2. Instead of registering the gauge G2 for the announcements of A1 and B1, we can create a new announcement of type AM1, on which the G2 could be registered. G1 is then the component, which is suggested to generate the announcements of type AM1.

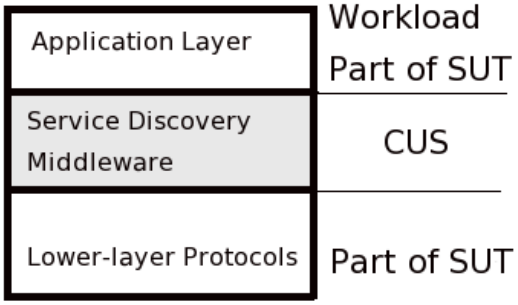


Figure 5.7: Identifying the SUT and CUS is important for the workload and metrics selection.

The described infrastructure of GaA plays actually the role of a monitor. According to [Jain 91], the monitors are used to observe system performance, collect performance statistics, analyze the data, and display results. Precisely speaking, the role of the monitor is played by the *Gauge* component. It can be classified as a software monitor by the level of its implementation. A hardware and hybrid monitors are the other variants. By a triggering mechanism our software monitor can be classified as an event-driven, where the `Announcements` are representing the events. A sampling monitors is another variant. According to the ability to display results the Gauge is rather like a batch monitor, which collects data that can be analyzed later using a separate analysis program. Our Gauges are usually expected to pre-calculate some metrics, which can be used by the researcher as soon as the emulation is finished. Another type of monitor is the on-line monitor, which displays the system state continuously or at frequent intervals.

5.5 Visualization

According to our decision in Section 4.4, the *Prefuse* is taken as a basis for our visualization efforts. How should we extend *Prefuse* to meet our requirements?

As already said, we have the following requirements to the visualization. The services and clients and their connectivity should be visualized. We wanted to deploy the visualization module on the workstation, which is different from the workstation with the rest of the *ESKEm* (”other-host” requirement). Our visualization should be done in real time and should provide meaningful visualization keys.

First, we would like to consider, what we are going to visualize. The network can be intuitively represented as a graph. This was the reason why we searched for something like *Prefuse.* There are at least three types of vertices which should be available in our graph: clients, services and routers. The connectivity in wired networks seems to be better visualized as a link between vertices. In a wireless environment it could be either a circle or a link. The communication over the link could be visualized as a value assigned to the link. Besides the link value we decided to visualize the traffic flow over the link by its thickness. As far as a number of messages processed by the vertex on the graph matters, we visualized them as the size of the vertex. Sending the messages over the link results in its increased size.

We identified two types of visualization in Section 4.4. The current version of *ESKEm* supports only the ”point-in-time” visualization. As a consequence, the thickness of the link between two vertices shows only an accumulated number of sent message. The information about distribution of messages transferred is lost. The same applies to other aspects of visualization.

The visualization of the aspects above is achieved by the set of the commands, which are sent to the visualization module. The commands have usually the set of parameters. At least the following visualization commands should be supported:

- the command which creates/deletes a vertex. The vertex type - namely client, service, and router - and associated IP address used as ID in the visualization module should be passed as the parameters.
- the command which creates/deletes the link between two vertices to visualize the connectivity between them. The IP address of the two vertices is passed as the parameter.
- the command which visualizes the transfer of the message over the specific path. The pairs of IP addresses are passed as a parameter. Each pair is responsible for a single link.

As far as we should be able to deploy the visualization module on another workstation, we need some means to communicate our commands to it. Among different alternatives (i.e. RMI, SOAP etc.) we selected the simplest approach - we use UDP-based messages. The visualization commands are generated either in the application-level code (like in SU/SM, Figure 5.1) on in the network layers (Network Module on Figure 5.1). The commands and its parameters are than sent and interpreted on the other side in visualization module. They are passed as an usual text with separators, i.e. `''command=create;type=client;ip=10.150.61.33''`.

Chapter 6

Implementation

We implemented the *ESKEm* emulation framework according to the design ideas elaborated in the previous chapter. The *UPnP Cybergarage* was used to validate our approach (Requirement B.7 and B.7). The framework is implemented in Sun's Java programming language (Requirement C.3).

In this chapter we consider different implementation details of the *ESKEm* framework. We start with the description of the framework's *installation* details such as required libraries and their versions. Further, we analyze different components of the *ESKEm* starting from its *main* class. The overview of the complete framework's implementation is given first. Specific technical details will be provided in the consecutive paragraphs.

We generate several emulation scenario scripts and use them for the emulation in the *ESKEm.* The results are used for the validation of the framework.

6.1 Installation and Execution

```
andrej@koleso:~/workspace-esk> java -version
java version "1.5.0_01"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_01-b08)
Java HotSpot(TM) Client VM (build 1.5.0_01-b08, mixed mode, sharing)
andrej@koleso:~/workspace-esk> uname -a
Linux koleso 2.6.8-24-default #1 Wed Oct 6 09:16:23 UTC 2004 i686 i686 i386 GNU/Linux
andrej@koleso:~/workspace-esk>
```

Figure 6.1: The *ESKEm* was developed under a Linux operational system with the Sun's JDK 5.0 (Java Development Kit).

In this section we introduce an environment used for the development and test of the *ESKEm.* We present further the instructions explaining how framework should be started. The *Eclipe 3.1* platform was used for the IDE (integrated development environment) for development of the *ESKEm.* The JDK5.0 (see Figure 6.1) or later should be installed in order to compile and run the framework.

Directory structure The workspace of the *ESKEm* consists of several subprojects:

./Em the *ESKEm* subproject

./jist-swans-1.0.6 the modified version of the *JiST/SWANS* framework.

./upnpDirJist the modified *UPnP Cybergarage* framework.

./upnpDirOriginal the original version of the *UPnP Cybergarage* framework, which should be used to run on the external devices. The framework was slightly modified. An explanation for this is given later in the chapter.

./EmulGUI the visualization module of the *ESKEm*

The user of the framework will mainly work with the `./Em` directory, where the emulation scenario scripts and relevant configuration files are placed.

Run The shell scripts mentioned here are assumed to be located under `./Em` directory, if nothing else is specified. The explanation and screenshot are related to the Linux operational system.

Before the emulation framework is started with the `./run.sh` script, several configuration steps need to be accomplished.

First, the emulation scenario should be defined in a file like `./etc/script<number>.em`. The number of the script is used to differentiate between different emulation scripts. Additionally, the topology of the simulated network should be specified in the appropriate `./etc/rnet<number>.xml` file.

```
koleso:/home/andrej/workspace-esk # ifconfig
eth0      Link encap:Ethernet  HWaddr 00:02:3F:14:43:4E
          inet addr:10.150.61.32  Bcast:255.255.255.255  Mask:255.255.255.0
          inet6 addr: fe80::202:3fff:fe14:434e/64 Scope:Link
          UP BROADCAST NOTRAILERS MULTICAST  MTU:1500  Metric:1
          RX packets:1229146 errors:100 dropped:192 overruns:98 frame:0
          TX packets:2540582 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:177522530 (169.2 Mb)  TX bytes:2674880570 (2550.9 Mb)
          Interrupt:10 Base address:0xc000

eth0:1    Link encap:Ethernet  HWaddr 00:02:3F:14:43:4E
          inet addr:10.150.151.21  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST NOTRAILERS MULTICAST  MTU:1500  Metric:1
          Interrupt:10 Base address:0xc000

eth0:10   Link encap:Ethernet  HWaddr 00:02:3F:14:43:4E
          inet addr:10.150.152.29  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST NOTRAILERS MULTICAST  MTU:1500  Metric:1
          Interrupt:10 Base address:0xc000

eth0:11   Link encap:Ethernet  HWaddr 00:02:3F:14:43:4E
          inet addr:10.150.152.30  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST NOTRAILERS MULTICAST  MTU:1500  Metric:1
          Interrupt:10 Base address:0xc000

eth0:12   Link encap:Ethernet  HWaddr 00:02:3F:14:43:4E
          inet addr:10.150.152.31  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST NOTRAILERS MULTICAST  MTU:1500  Metric:1
          Interrupt:10 Base address:0xc000
```

Figure 6.2: The figure above shows the results from executing the shell script `./etc/config4.sh`.

Second, the IP aliases should be configured with the call to the appropriate `./etc/config<number>.sh` shell script. The script uses the calls for the `ifconfig` and `route` commands, which require the privileges of the superuser. Under Linux this would be the *root* account.

The routing table is changed too. The example of the routing table is shown on Figure 6.3. The important routes are shown in bold.

```
koleso:/home/andrej/workspace-esk # route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.1.0     *               255.255.255.0   U     0      0        0 vmnet8
10.150.61.0     *               255.255.255.0   U     0      0        0 eth0
10.0.0.0        *               255.0.0.0       U     0      0        0 eth0
loopback        *               255.0.0.0       U     0      0        0 lo
224.0.0.0       *               240.0.0.0       U     0      0        0 eth0
default         10.150.61.254   0.0.0.0         UG    0      0        0 eth0
```

Figure 6.3: The routing table after invocation of the `./etc/config4.sh` shell script.

Now the `./run.sh` shell script is ready to be run. The `java` call is the most important part of the script:

java -Demulguiserver=10.150.61.35 -Dmode=emu -Dduration=30 -classpath $CLASSPATH jist.runtime.Main org.esk.Main –test 4

The `org.esk.Main` main class of the *ESKEm* is executed from within the *JiST*'s main class `jist.runtime.Main`. The parameters of the *ESKEm* are passed either as the java system property parameters (i.e., as a "-D<name>=<value>" pairs) or as command line parameter like `--test 4` here. The system property value is accessed in the Java with the invocation of the `System.getProperty(`<name>`)` method. The following parameters can be passed to the *ESKEm*:

-Dmode	the *emu* and *simu* are possible values. They specify the mode of operation for the *JiST* simulation kernel.
-Demulguiserver	the IP address of the host, on which the *EmulGUI* is started.
-Dduration	specifies duration of the emulation in seconds. The emulation is finished after the specified time and the measured statistics are printed.
–test	the number of the emulation script, which is to be executed. For example, the "4" means, that the routers' topology from the `./etc/rnet4.xml` will be read and the emulation scenario from the `./etc/script4.em` will be executed,
–platform	Only *upnp* value is currently supported; the value specifies the service discovery protocol implementation, which should be used during emulation. The *upnp* here assumes a specific reference implementation of the UPnP specification like the *UPnP Cybergarage* in our case. Another version of the UPnP could be specified with a value like *upnpA*.

The visualization framework *EmulGUI* can be started either on the same or on the separate host with a call like:

java -classpath $CLASSPATH esk.ubicom.emulator.visualization.VisualizationFrame

6.2 Implementation Overview

Before covering the implementation in details, we provide a short overview of the main logic of the *ESKEm* execution. Instead of phrase *"the subclasses implementing the interface `Interface` call ..."* we just say *"the `Interface` calls ..."* as it is easier to read. Further, calls like `(QueueE)q.start()` should be read as *"the instance **q** of type `QueueE`"*. On the other hand, calls like `RoutersNet.getInstance()` are calls to the static methods.

The execution of the *ESKEm* is started in the `org.esk.Main` main class. The framework is configured before the emulation starts:

- the initial routers' topology is configured with a call to the `RoutersNet` class.

- the `ScriptReader` reads the *emulation commands* from the script file, instantiates the `IAction`s from commands and schedules them in the `QueueE` real time scheduler.

The `Waiter` is executed in a separate thread. It has the task to finish the emulation as soon as the time for it elapses. This is the time specified by the "-Dduratino" parameter. The `Waiter` also initiates the output of the calculation for the emulation scenario statistics.

The emulation is started with a call to the `(QueueE)q.start()`. The `QueueE` is a real time scheduler of the *ESKEm*. The scheduler can execute the `IAction` classes, which are specified as the command in the emulation script. Each `IAction` class is associated with the `EskUnit` class, which is either the `Service Manager` or `Service User`. The `IAction` is executed on the associated `EskUnit`.

The `EskUnit` can send the messages over the simulated network with the `SimAdaptee` instance. The `SimAdaptee` is the `entry point` into the network simulation facility. It can also receive the messages from the simulated network.

The implementation is covered in more detail in the next few paragraphs.

6.3 Emulation Scenario

Here we introduce the emulation scenario script format and related Java classes.

6.3.1 Emulation Scenario Scripts

The *emulation scenario script* (or simply *emulation script*) contains different commands influencing the life cycle of the services and clients. This results in a service discovery related activity. For example, the start of a device may lead to the fact that its services are registered at a lookup table. Here is an example of how a simple emulation script looks like:

start dev2 device 192.168.151.21 1000
start clnt1 client 192.168.153.23 5000
stop clnt1 9000
start dev3 device 192.168.152.22 12000
stop dev2

The syntax is rather simple. Nonetheless, it meets our current requirements. Every line of the script file consists of the script command and the command's arguments (or parameters). The *start* and *stop* are two types of commands which we currently support. All consecutive positions are arguments to the command.

Let us consider the first line from the example above. The first argument to the *start* command is the symbolic name of the device to start (here, *dev2*). The second argument says whether the device contains services or clients. The argument *device* here means that we start a device with services (or Service Manager, according to the terminology from Section 3). The argument *client* means, that we start a device with a client on it (or simply Client). The third argument to the *start* command is the IP address of the started node. The last argument is the timestamp for the command invocation relative to the beginning of the emulation. The *stop* command only takes a symbolic name of the node and the timestamp of the invocation as the arguments.

Each command of that kind results in the instantiation of appropriate subclass of the type `IAction`, which is scheduled to the `QueueE`. The `IAction` is executed in the real time.

Further we introduce the classes responsible for the interpretation of the emulation scrips.

6.3.2 Classes

The emulation scripts are read by the `ScriptReader` in the `Main` class (see Figure 6.4). The `IAction` is created by the call to the `ActionFactory`, which is passed to the emulation command and its parameters. The command name is used by the factory to decide, which `IAction` subclass should be instantiated. For example, the *start* command causes the `ActionFactory` to instantiate the object of the type `StartUnitAtion`. The `IAction` object is instantiated by passing the emulation command and its arguments to the class's constructor. It is up to the `IAction` subclass, how the command's parameters should be interpreted.

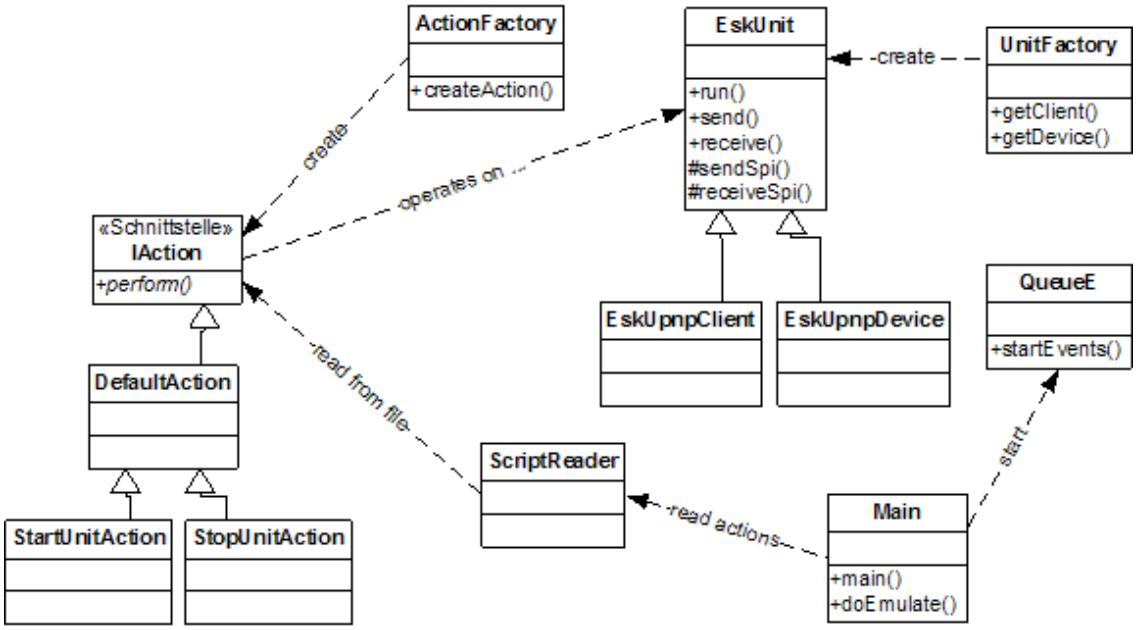


Figure 6.4: The *ESKEm* architecture: the classes participating in the execution of the emulation script scenarios and their most important methods.

There are two `IAction` classes supported by the *ESKEm* now: the `StartUnitAction` and `StoptUnitAction`. The `StartUnitAction` is the class where the `EskUnit` is instantiated and parameterized. We will come back to the `EskUnit` later. The `StartUnitAction` makes a call to the `register(adapter,thread,ip,name)` method at the end of its initialization. The call is important and creates the mappings between passed parameters. For example, the mapping between *thread* and the *IP* address facilitates the finding of the device's thread by its IP address. We use the "mapping" technique rather often.

Listing 6.1: There are several loops in the `QueueE`. Loop [2] waits until the actions queue is non-empty. The next loop [3] waits until the first action on the stack is executed. The outer loop [1] is executed until the queue thread is not stopped for some reason.

```
1  // irrelevant pieces of the code were left out
2
3  public class QueueE {
4
5     public int startEvents() {
6
7        // 1. loop until the emulation should stop
8        do {
9              // 2. wait here until the actions stack is not empty
10              while (isEmpty()) {
```

```
11                  // 2.1. exit current loop because of the stop
12                  if (stop) break;
13                  Thread.sleep(SLEEP);
14
15                  if (System.currentTimeMillis()-startedSimAt>duration){
16                      stop=true;
17                  }
18
19              }
20
21              // 3. exit current loop because of the stop
22              if (stop) break;
23
24              // 3.1 the actions stack is not empty.
25              //      But wait in the loop until the action should be executed.
26              do {
27                  // 3.1. get next action
28                  event = (IAction) this.get(0);
29
30                  // 3.2 wait, if the action should not be execute now
31                  if (!bSimulate){
32                      Thread.sleep(SLEEP);
33                  } else {
34                      // 3.3. progress simulation time before executing the action
35                      JistAPI.sleep(exeTime-JistAPI.getTime());
36                  }
37              } while (!bSimulate);
38
39          // 4. execute action in a separate thread
40          Thread t = new Thread((Runnable)event,event.getClass().toString());
41          t.start();
42
43          // 5. remove action from the stack
44          removeEvent(event);
45
46
47      } while (!stop);
48
49  }
50 }
```

The `IAction` is executed by calling its `perform()` method. The `IAction` instances are executed in the `QueueE`. The `QueueE` is a real time scheduler. The essential parts of the scheduler's loop are represented on the Listing 6.1. To sum up, the scheduler works as following. If the events are on the stack, they are executed provided it is time for the action to be executed. If either the actions' stack is empty or it is too early to execute a current event, the scheduler sleeps for a specified number of a millisecond (e.g., 20 millisecond) and tries either to execute the action again or check the stack for new events.

The `QueueE` also synchronizes the simulative and real executive times. When the action is executed, the `QueueE` compare the real executive time and the simulative time. The real executive time in this case is the time elapsed from the beginning of the emulation. The simulative time is advanced if it is behind the real executive time (see also Sections 4.2.1 and 5.3).

6.4 EskUnit

The `EskUnit` is an abstract class, which should be extended by the specific to the service discovery protocol *Service Managers* and *Service Users*. As already mentioned in the previous chapters, the *Service Manager* should behave "as if the service were there" in respect to the service discovery protocol. For example, our *Service Manager* is not supposed to react in any way when a client tries to invoke it. On the other hand, the *Service Managers* should reply to the service discovery messages from the client according to the protocol. The same applies to the emulated *Service Users*. Each additional service discovery protocol which is to be explored in the *ESKEm* is expected to implement its own subclasses of the `EskUnit`.

The methods call of the `EskUnit` class are designed according to the *Template* Pattern [GoF 95]. The class should be extended by the implementation of its SPI (service provider interface) methods like `receiveSpi()`.

The service discovery related behavior of the *Service Manager* can be either implemented from scratch or *adapted* from existing services. In the case of the *UPnP Cybergarage*, the templates of the services are already available. They already "speaks" the UPnP's SSDP (Simple Service Discovery Protocol). Our approach is explained on the example of the `EskUpnpDevice`. The `EskUpnpDevice` (see Figure 6.4) *adapts* the `org.cybergarage.upnp.Device`, which is specific to the *UPnP Cybergarage* and is a superclass of devices with services like `LightDevice` (compare to the *Adapter* Pattern from [GoF 95]). The `EskUpnpDevice` can instantiate, start and stop the adapted `Device`. The start (or stop) of the `Device` results in the service discover related messages. The messages flow over the `EskUpnpDevice` and can be either passed further to the network or blocked. The blocking of the messages can be used to imitate the stop of the adapter device instead of actually stopping the device. The stop and start of `Device` is resource consuming and the just mentioned imitation is intended to save resources. The technical details are explained in the next section.

On the other hand, the `EskUnit` adapts also the classes, which provide the *entry points* into the simulated by the *JiST/SWANS* network. Precisely speaking, this is the `SimAdaptee` class, which can be used both to send and receive message to/from the simulated network.

The `EskUnit` runs in a separate thread concurrent to the other `EskUnit`s and in particular to the *Jist-Controller* thread responsible for the network simulation. The `EskUnit`'s thread sleeps most of the time to save system resources. The events can be passed to the `EskUnit` by calling an appropriate method with parameters (like `receive(...)` (see Listing 6.2). The call occurs out of the thread *T*, which is different from the `EskUnit`'s thread. The event and its parameters are encapsulated into the private help class and put onto the `EskUnit` stack. Finally, the `interrupt()` on the unit is thread is called. This results in the interruption of the unit's sleep. The unit takes the event(s) from the stack and processes it/them in its own thread and not in the thread *T*.

Listing 6.2: The `EskUnit` sleeps [2] until an interesting event arrives. The call to the `receive(...)` in [5] results in the thread interruption. The code then continues in [3.1] by processing the passed events [3.2].

```
1  // see EskUnit.java for complete code
2
3     public void run() {
4        // 1. execute subclass-specific code
5        startSpi();
6
7        while (started) {
8           // 2. wait as much as possible until interrupted
9           Thread.sleep(1000000000);
10
11          // 3.1. iterate over events passed when sleeping and ...
12          int i=0;
13          for (; i < events.size(); i++) {
```

```
14              action = (String) events.get(i);
15              if (action.equals("receive") && started && data!=null){
16                  // 3.2. ... execute them, in this case in SPI
17                  receiveSpi(data);
18              }
19          }
20
21          // 4. delete processed events
22          synchronized(events){
23              while(i>0){
24                  events.remove(0);
25                  i--;
26              }
27          }
28      }
29  }
30
31  // 5. the code in other thread invokes the methods like this one
32  public void receive(...){
33      ...
34
35      thread.interrupt();
36  }
```

Each instance of the `EskUnit` represents either an internal or external service. For example, if an emulated service A responses to the external client B, the messages from A are sent over the simulated network to the *proxy* of the client B on the host with *ESKEm* first. The role of a *proxy* is accomplished by the `EskUnit`. For example, it is the instance of the `EskUpnpClient` for the external clients. The `EskUnit` unit for an external client is created as soon as appropriate new information about an external environment arrives in the *ESKEm.* In our case, the `EskUpnpClient` proxy for the external real client is created as soon as the *search request* message arrives in the *ESKEm.* The communication with external clients over their proxies will be covered in more detail in the next subsection.

6.5 Communication over the Simulated Network

Here we explain, how the messages are passed between different `EskUnit`s over the simulated network. In the next section we describe the classes which participate in the network simulation.

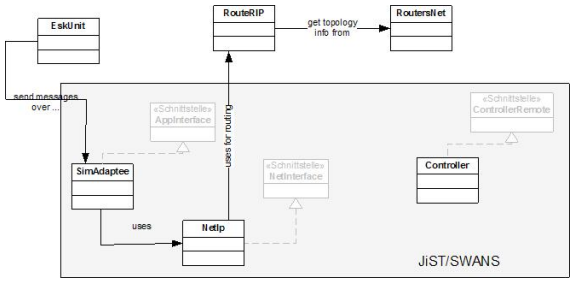


Figure 6.5: The classes related to the network simulation in the *ESKEm.*

The message flows between two emulated `EskUnit`s as shown on Figure 6.6. According to our inspection of the `UPnP Cybergarage` source code, the `HTTPMUSocket` and `HTTPUSocket` are the classes, used by `Device`s for the service discovery related communication. Further, these classes use the `java.net.MulticastSocket` and `java.net.DatagramSocket` Java classes respectively

in order to communicate with their counterpats over the real network. We replaced calls to this classes like `(DatagramSocket)socket.send(DatagramPacket dgmPacket)` with the calls to the `NetAdapter.send(DatagramPacket dgmPacket)`. The calls receiving the messages from the real network are left without changes. The `DatagramPacket` passed as parameter is forwarded to the appropriate `EskUnit` instance (step [1] in Figure 6.6).

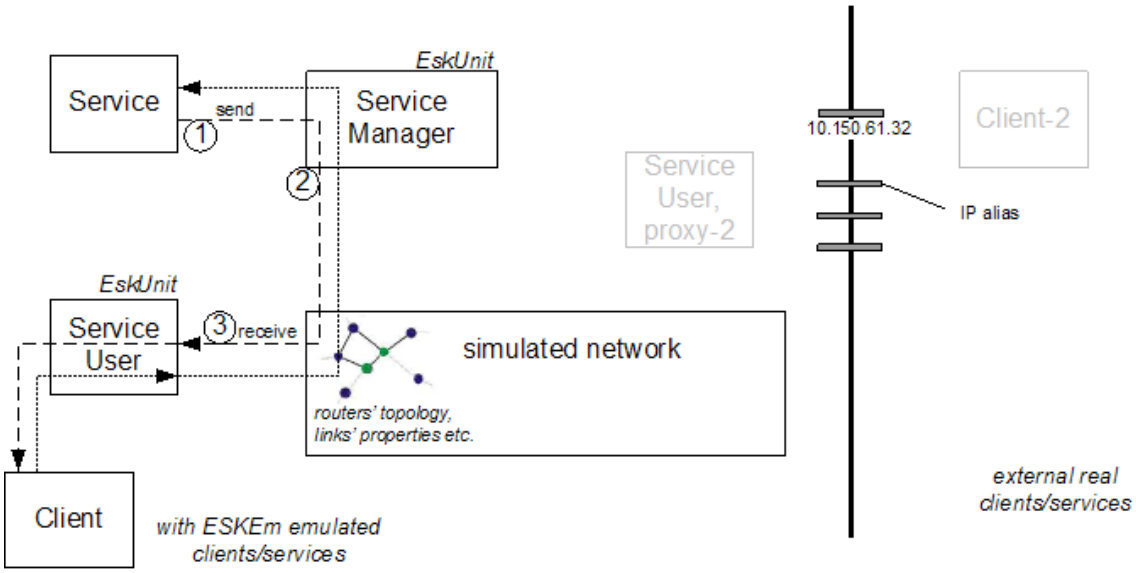


Figure 6.6: An example for communication between an *emulated* service and a client over a simulated network.

As already mentioned, each instance of the `EskUnit` owns the `SimAdaptee` instance (see Figure 6.5). It is used by the `EskUnit` to pass message to and receive them from the simulated network (see step [2] on Figure 6.6). The `SimAdaptee` knows how to attach the `java.net.DatagramPacket` to the `EskIp`, which is the object representing the packet in the *JiST/SWANS* network simulator. The simulation is applied to the `EskIp` messages. The instance contains the values calculated by the simulation. An example is a delay of a message.

The `EskIp` packet is received by the `SimAdaptee` related to the (emulated) client (see step [3] on Figure 6.6). The `SimAdaptee` detaches the `DatagramPacket` from the `EskIp` and pass it further to the `EskUnit`. It was already described in the previous section how the events and their parameters are passed to the `EskUnit`. The `EskUnit` may also wish to pass the message further to the adapter service or client.

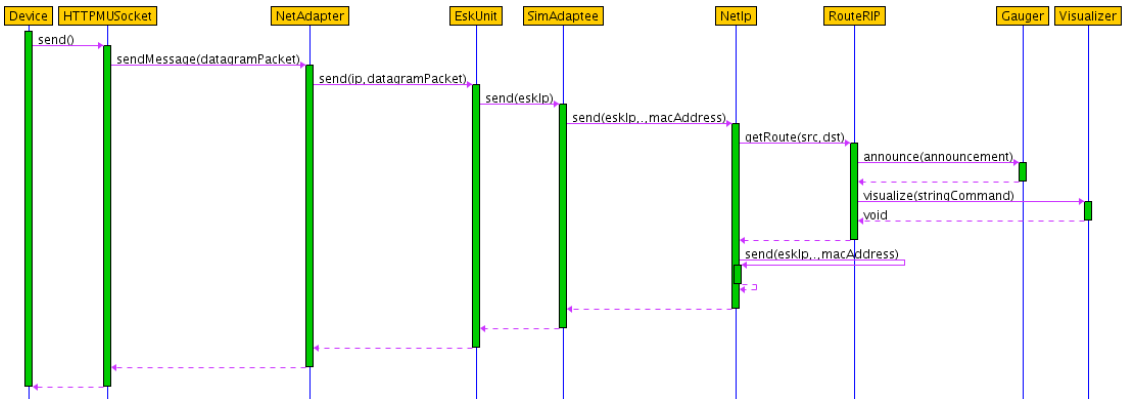


Figure 6.7: The *ESKEm* sequence: Sending Messages

This is the complete path traversed by the message from the sender to the receiver. There are a couple of technical notes left to be made for a complete understanding of the just described process.

Simulated network protocols stack The *JiST/SWANS* defines the set of interfaces which can be implemented by the developer in order to introduce new network protocols into the simulator. For example, the `NetIp` implements the `NetInterface` (both are delivered with the `JiST/SWANS`) and thus is responsible for the simulation of the network characteristics at the 3d (Network) layer. The simulated protocols stack for the `EskUnit` is constructed in the `UnitFactory` class and is bound to the `SimAdaptee` (see Listing 6.3). Indeed, this is the `SimAdaptee` which finally make calls to the simulated protocol on the top of the stack. As a result, the `NetIp` is the first network layer which receives the messages from the `SimAdaptee`.

Listing 6.3: An example of the initialization of the simulated network protocols stack (`UnitFactory`). The developer may wish to extend the class and to provide a different network protocols stack for different `EskUnit`s.

```
1  // see UnitFactory.java for complete code
2
3     private static EskUnit initProtocolStack(EskUnit unit){
4
5        ...
6
7        MacEthernet mac = new MacEthernet(new MacAddress(nodeNum));
8        NetAddress ip = new NetAddress(InetAddress.getByName(unit.ip));
9        NetIp net = new NetIp(ip, NetAdapter.protMap, NetAdapter.pl,NetAdapter.pl);
10       SimAdaptee app = new SimAdaptee(nodeNum, unit);
11
12       ...
13
14       unit.setApp(app);
15       app.setNetEntity(net.getProxy());
16       net.setProtocolHandler(Constants.NET_PROTOCOL_HEARTBEAT, app.getNetProxy());
17       net.setRouting((new RouteRIP(net)).getProxy());
18       mac.setNetEntity(net.getProxy(), intId);
19
20       ...
21
22       return unit;
23    }
```

JistController and EskUnit threads The network simulation of the *JiST/SWANS* is accomplished in a separate *JistController* thread. The attached to the `EskIp` `DatagramPacket` is passed to the `EskUnit` after the effect of the network characteristics are calculated for the `EskIp`. The `DatagramPaket` should be processed further either in the `EskUnit` or adapted classes. And the *JistController* is not a right place for this processing. So we decided to pass messages from the simulated network to the `EskUnit`'s thread as describe in the previous section (see also Listing 6.2). As a consequence, the `SimAdaptee` runs in the *JisController* thread and calls the `receive(DatagramPacket dgm)` method on the `EskUnit`. In the next step, the `dgm` message is processed in the `EskUnit` in a concurrent to the *JisController* thread.

The `EskUnit` is actually a good place for the application of effects. These effects were calculated by the simulation kernel for the transfer of the `EskIp`. Indeed, if the simulation took 300ms to calculate that the message should be delayed for 900ms, how should the effect of the simulation be applied? The *JistController* thread is a wrong place for waiting 600ms before transfering the packet to the `EskUnit`. The delay can be applied in the `EskUnit` itself without any sideeffects.

Communication with real (external) clients According to Requirement B.6, the emulated services should be discoverable by the real clients. Up until now we have only considered only the communication over the simulated network. A question which remains unanswered is how the discovery of an emulated service is accomplished by a real client. The idea is also represented on Figure 6.8.

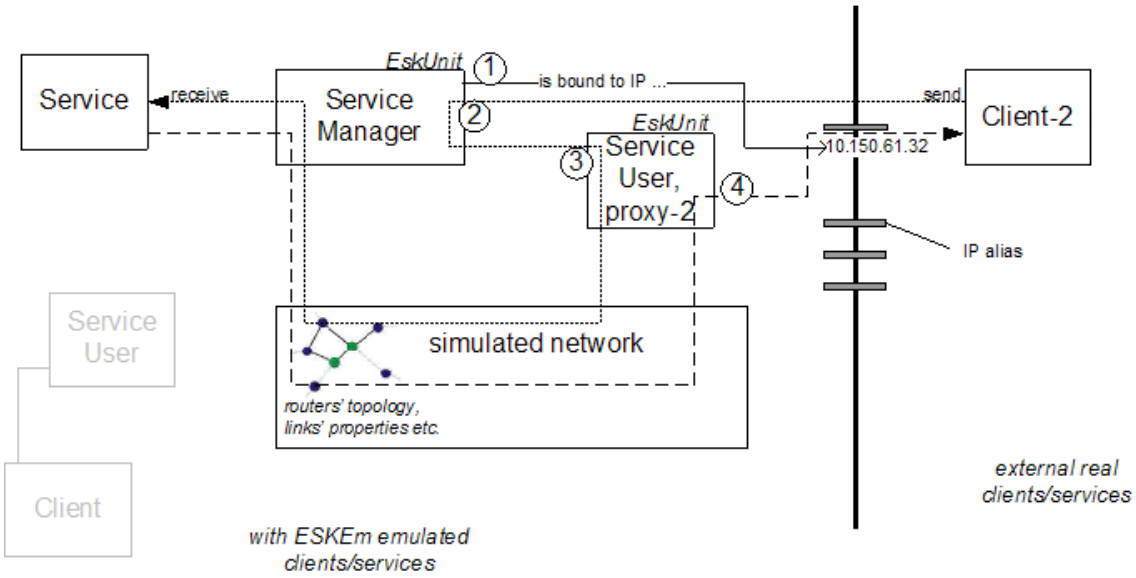


Figure 6.8: An example for the commnication between *emulated* service and *real* external client.

Each emulated `Device` connects with the Java's `DatagramSocket` to one of the *alias interface* of the host (each *alias interface* has an *alias IP address*), where the *ESKEm* runs (see Figure 6.8). The message from an external client arrives at one of the alias interfaces and at the end in the `EskUnit` (step [1] on Figure 6.8). If the IP address of the message is not associated with any (responsible for an external client) `EskUnit`, a new `EskUnit` is created, and is marked as being responsible for an external real client (step [2] on Figure 6.8). This means that the communication between an external client and an emulated (internal) device is accomplished through this newly created `EskUnit`. This was actually the reason why we earlier referred to this instance of the `EskUnit` as a *proxy*.

The messages from the external client is passed to the just created (or picked from the hash table) instance of the `EskUnit` by the call of a `send(DatagramPacket dgmPkt)` method (step [3] on Figure 6.8). Further, the message is passed through the simulated network as described earlier.

The messages from an emulated service are passed to the `EskUnit` of the external client in a uniform way. The `EskUnit` then decides whether the message received through the simulated network should be passed further to the real client through the real network. The proxy sends the messages to the real client over the Java's `DatagramSocket`. It is bound to the IP interface of the service which originally send the message (step [4] on Figure 6.8). In this way the real client receives the message as if it were send from the IP address of the original emulated service.

Finally, we would like to stress the fact that the topology of the simulated network is taken into account during transfer of the message from the real client to the emulated service. Further, the routing tables of the real client should be configured appropriately so that the client messages appear on the *ESKEm* alias interfaces.

The scheme for the communication between emulated and real service or clients works well for our purposes. But there is a problem with the proposed approach which is explain here. The multicasting packets from the emulated services are not influenced by the topology defined for the network simulation as soon as the packet is sent over the alias interface. For example, device *A* sends the multicasting packet. Assume, that there are two real clients *B* and *C* on the local subnet. The client *C* receives the multicasting packet

even if it is not reachable from *A* according to the topology of the routers. This problem will not occur in the case of a unicast message as far as the simulation can decide whether the unicast message can be sent exactly to *C* or not. The workaround for the problem is possible. For example, when the real routers between *A* and *C* are configured in the appropriate way.

6.6 Simulated Network and *JiST/SWANS*

The previous section explained how the simulated network is used in the *ESKEm*. The internal mechanisms of the network simulator were only partially covered. This section intends to consider the parts of the network simulator which we extended.

The `NetIp` is the SWANS's implementations of the OSI Network Layer according to the [RFC 791]. The routing protocol should be bound to it in order to account for its effect during the sending of packets over the simulated network. We name the class responsible for the routing as `RouteRIP`. This is *not* the simulation of the *RIP* protocol according to the [RFC 1058]. Our approach was rather straightforward as far as the exact simulation of the network layers responsible for the transport service was not the focus of the current work. We describe below how the routing is taken into account in the *ESKEm* (refer to Figure 6.5).

Routers' topology The information about the routers' topology is made persistent in the `rnet<number>.xml`, where the *number* is the number of the emulation scenario. Here is an example of the file:

<graph directed="0" graphic="1">
<node id="1" label="10.150.61.254"></node>
<node id="2" label="10.150.62.254"></node>
<edge id="1_2" label="1" source="1" target="2"></edge>
</graph>

In the example above two routers are defined with the IP addresses *10.150.61.254* and *10.150.62.254* respectively. The link between the routers defined with the *edge* XML tag. The `RoutersNet` class reads the initial topology of the network from a file like above. In the current implementation of the *ESKEm* we assume that all routers are multicasting. During the initialization phase the `Router` help classes in the `RoutersNet` are used to hold information about routers. It is among other thing the list of the devices which are connected to the router. We assume the *netmask* of `255.255.255.0` and as a result the device with the IP `10.150.61.x` will be connected to the first router etc.

The `RouteRIP` uses the `RoutersNet` to calculate the route between sender and receiver of the packet. The `RouteRIP` is also used to calculate all reachable through the multicasting receivers packets. The class also accounts for the packets transfer characteristics such as the number of passed hops and associated delay. The `RouteRIP` makes calls to the `Gauger.announce(...)` for the purpose of the metrics measurements. It also makes calls to the `Visualizer.visualize(...)` to visualize the packets transfer over the network. The details of the measurement and visualization modules are explained in the consecutive sections.

The routing between sender and receiver is calculated in advance as follows. Assume that the node A that is a device with a service sends the message to the node B which represents a device with a client located on it. The devices are connected to the different routers like the A-Router and the B-Router. The shortest path between the A-Router and the B-Router is taken from the matrix produced by the Floyd-Warshall algorithm [Floyd-Warshall] (use the reference to find pseudo-algorithm):

The *Floyd-Warshall algorithm* is an algorithm used to solve the problem of finding all pairs of shortest paths in a weighted, directed graph. The edges may have negative weights, but no negative weight cycles. The pseudo-algorithm of the Floyd-Warshall algorithm can be found under [Floyd-Warshall] (see also D for the pseudo-algorithm of the Floyd-Warshall algorithm).

```
Floyd-Warshall Matrix:     each cell contains
 thenumber of the next node
    on the way from A to B
0       2       2       2
1       0       3       3
2       2       0       4
3       3       3       0
******************************
```

Figure 6.9: The Floyd-Warshal matrix is printed out just before the *ESKEm* emulation starts.

Every entry in such a matrix is the next router on the way from the A-source-Router to the B-destination-Router. The A-Router is located on the X-axis while the B-Router is located on the Y-axis of the matrix. The Floyd-Warshal matrix is printed by the *ESKEm* just before the emulation starts (Figure 6.9.

Controller The `Controller` is a *JiST/SWANS* class responsible for the execution of the scheduled events. This is the heart of the JiST's simulation kernel. We modified the `eventLoop()` of the `Controller` by allowing an endless loop. In its original implementation, the simulation exited as soon as the queue with scheduled events was empty. In the current implementation, the queue of the events is checked every 10ms for new events.

6.6.1 Measuring module

The `Gauge`s listen to `Announcement`s, which are generated either by services/clients or by the network simulation layers. We use configuration files to register specific announcements that must be sent to specific gauges as follows:

[*org.esk.gauging.specificgauges.MessageGauge*]
org.esk.simulator.networklayer.MessageSentAnnouncement
org.esk.simulator.networklayer.MessageReceivedAnnouncement

In the example above we register the `MessageGauge` gauge to listen for `MessageSentAnnouncement` and `MessageReceivedAnnouncement` announcements. The announcements are sent by an invocation of an appropriate method on the `Gauger` instance. It is the `Gauger` who meets the decision about sending the announcement to the specific `Gauge`.

The users of the `ESKEm` are encouraged to implement own gauges and announcements. The information encapsulated into the announcement and the way of its processing in the gauge can be quite different. The announcements from the example above contain the information about the messages, which were either sent or received. The `MessageGauge` accumulates them. The announced message is accounted as "received", if the gauge is able to find the matching `MessageReceivedAnnouncement` to existing `MessageSentAnnouncement`. The `MessageGauge` can be used further to output the information about the messages, which participated in the service discovery (see Figure 6.10). Another interesting example of the gauge is the `SearchGauge`. It counts how many services a client has discovered. It can be further compared to the total number of services available in the environment.

```
[response, recv]        =>      2
[byebye, sent]          =>      9
[search, sent]          =>      2
[search, recv]          =>      4
[byebye, recv]          =>      13
```

Figure 6.11: The gauges can also calculate basic statistics based on the data represented in Figure 6.10. Here, we show a count of different UPnP messages by type.

The information processing in the gauges can be of arbitrary complexity. The example on Figure 6.11 shows how the information from Figure 6.10 can be grouped. The messages captured here correspond to different types of the UPnP messages (compare with Section 3.4.1).

	ip1	ip2	commDirection	timestamp	msgType	hops	delay	size
0	192.168.151.21	239.255.255.250	sent	1000	multicast	.	.	154
1	192.168.151.21	239.255.255.250	sent	1000	multicast	.	.	179
2	192.168.151.21	239.255.255.250	sent	1000	multicast	.	.	181
3	192.168.152.22	239.255.255.250	sent	5000	multicast	.	.	154
4	192.168.151.21	192.168.152.22	recv	5010	multicast	0	10	154
5	192.168.152.22	239.255.255.250	sent	5010	multicast	.	.	179
6	192.168.151.21	192.168.152.22	recv	5020	multicast	0	10	179
7	192.168.152.22	239.255.255.250	sent	5020	multicast	.	.	181
8	192.168.151.21	192.168.152.22	recv	5030	multicast	0	10	181
9	192.168.153.23	239.255.255.250	sent	1000	multicast	.	.	101
10	192.168.152.22	192.168.153.23	recv	1010	multicast	0	10	101
11	192.168.151.21	192.168.153.23	recv	1020	unicast	0	20	101
12	192.168.154.24	239.255.255.250	sent	1020	multicast	.	.	101
13	192.168.152.22	192.168.154.24	recv	1030	multicast	0	10	101
14	192.168.151.21	192.168.154.24	recv	1040	unicast	0	20	101
15	192.168.152.23	239.255.255.250	sent	1040	multicast	.	.	154
16	192.168.152.22	192.168.152.23	recv	1050	multicast	0	10	154
17	192.168.151.21	192.168.152.23	recv	1060	unicast	0	20	154
18	192.168.152.23	239.255.255.250	sent	1060	multicast	.	.	179
19	192.168.152.22	192.168.152.23	recv	1070	multicast	0	10	179
20	192.168.153.23	192.168.152.23	recv	1080	unicast	0	20	179
21	192.168.154.24	192.168.152.23	recv	1090	unicast	0	30	179
22	192.168.151.21	192.168.152.23	recv	1100	unicast	0	40	179
23	192.168.153.23	192.168.152.22	recv	1110	unicast	2	50	383
24	192.168.153.23	192.168.151.21	recv	1120	unicast	3	30	383
25	192.168.152.23	239.255.255.250	sent	1120	multicast	.	.	181
26	192.168.152.22	192.168.152.23	recv	1130	multicast	0	10	181
27	192.168.153.23	192.168.152.23	recv	1140	unicast	0	20	181
28	192.168.154.24	192.168.152.23	recv	1150	unicast	0	30	181
29	192.168.151.21	192.168.152.23	recv	1160	unicast	0	40	181

Figure 6.10: An example of the output of a gauge. All messages which were transferred as the result of the service discovery processes are shown.

6.6.2 Visualization

Visualization is based on the *Prefuse* framework, which is aimed at visualizing different structures. We are interested in the visualization of graph structures. We therefore used *Prefuse* to build the *EmulGUI* visualization module (see Figure 6.12), which meets our requirements.

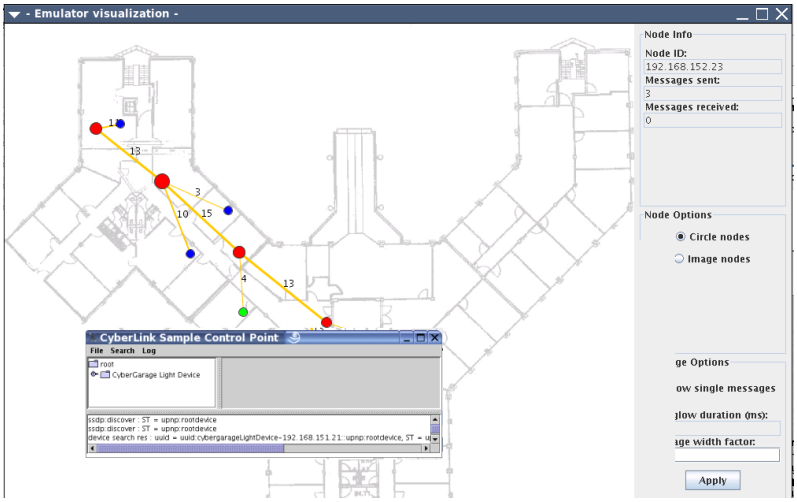


Figure 6.12: Visualization of the emulation scenario. Foreground: *UPnP Cybergarage* client. Background: emulated network infrastructure, where red node are routers, blue - devices/services, green - clients.

The *EmulGUI* can be run on a separate workstation. It receives the so called visualization command from the core components of the *ESKEm.* The UDP transport is used to deliver the visualization commands to the visualization module. The commands are sent as plain text. Here is an example of visualization commands:

command=create;unitip=192.168.153.254;unittype=router;xLocation=300.0;yLocation=300.0
command=create;unitip=192.168.153.23;unittype=device;xLocation=305.0;yLocation=290.0
command=link;from=192.168.152.254;to=192.168.152.23
*command=path;192.168.153.23=192.168.153.254;(<ip1>=<ip2>;)**
command=stop;unitip=192.168.152.23

First *create* command creates a vertex on the graph of the specified type *unittype* and at a specified location (*xLocation*,*yLocation*). The first two commands create a router and device nodes respectively. The link command creates a link between two nodes. The *path* command is used to visualize the traffic between nodes. This command is the result of sending a message (multicast of unicast) from the node of the network. Each pair of the form <ip1>=<ip2> in the *path* command defines one of the links, which were passed during the message transfer. The *EmulGUI* accounts for these links and uses them for a visualization of the graph edges. The last command *stop*s the node with the ID equals to the 192.168.152.23 IP address.

We use custom layout in *EmulGUI* to specify a position of the vertices. As a consequence, new commands like *move* of the vertices can be implemented without significant effort.

6.7 Validation

In this section we introduce the results of the *ESKEm* validation. We used the *UPnP Cybergarage* reference implementation to validate our approach. First, we consider a simple example of an emulation scenario. We validate the accuracy of the example by means of analyzing the trace of the gauges and the graph drawn with the *EmulGUI* module. Second, we consider more complicated examples. On one side, they are used to demonstrate performance of the framework. On the other side, they demonstrate metrics measurement and visualization for the *UPnP Cybergarage*. The explanation in this section assume the reader has basic knowledge of the Linux-like operational systems. Most of the output for the introduced scenarios can be found in Appendix B and A.

6.7.1 Accuracy validation

The following emulation scenario is a simple example of the *ESKEm* functionality. But it is easy to trace and thus may be better for the demonstration purposes than more complex examples.

Out emulation scenario consists of 3 emulated services, 1 emulated client and 1 real client. We validate the number and distribution of messages sent by the services and clients, their visualization and discoverability of the emulated services by a real client. The emulation script looks as follows:

start dev1 device 192.168.151.21 1000
start dev2 device 192.168.152.22 5000
start dev3 device 192.168.152.23 9000
start clnt1 client 192.168.153.23 20000

The *ESKEm* host must be configured for this scenario with the alias IP interfaces as follows:

ifconfig eth0:3 10.150.61.21 netmask 255.255.0.0 up
ifconfig eth0:4 10.150.62.22 netmask 255.255.0.0 up
ifconfig eth0:5 10.150.62.23 netmask 255.255.0.0 up
ifconfig eth0:6 10.150.63.23 netmask 255.255.0.0 up
route add -net 224.0.0.0 netmask 240.0.0.0 dev eth0

The last call to the `route` command is required for the multicast packets be routed to the `eth0` network adapter.

The metrics measured with the *ESKEm* measurement infrastructure were compared with those measured for the real services and clients. The messages from the real devices were captured with the *Ethereal* (*tcpdump*-like GUI-based tool used to capture network traffic).

Both textual and graphical output from the *ESKEm* are represented in Appendix A. The textual output is manually commented to associate it with the UPnP-based service discovery. The trace of the messages is expected to be used in conjunction with an external packet for the statistical analysis. Usually, this output is rather detailed to be analyzed manually.

Results All in all, the emulation works as expected. But careful consideration for the graphical and textual output will reveal a discrepancy to the number of messages which would be captured in a real environment. Namely, the *search* messages from the external clients are not accounted accurately in the current version of the *ESKEm* because of a technical problem as explained in the next paragraph.

As you can see on Figure A.1, the number of messages on the link between external client (marked with an arrow) and its node is equal to 6. But it must be 4 according to our expectations: 1 multicast search request and 3 unicast search responses. The 6 messages were accounted because 1 multicast message was calculated as 3 unicast search request messages.

The roots of the problem are the following. We have already explained how the messages from the external clients are sent over the simulated network. As a consequence of our technique, the multicast packets from the external client are received by all alias IP interfaces (see Figure 6.13). In turn, the messages are passed to the appropriate `EskUnit` instance which plays the role of the *proxy* for an external real client. The proxy is responsible for the external client and send the messages on its behalf over the simulated network. Further, the message received on the different IP interfaces ([2]) result in the `DatagraphPacket` of Java. The problem is that any `DatagramPacket` passed to the proxy ([3]) is considered to be a new packet. In other words, the proxy can not recognize whether two packets are actually the same packets received on two different alias IP interfaces or not. The `DatagramPacket` does not even have the timestamp when it was created on the source host - namely, it could help us to detect identity of the messages. Why is it a problem? It is explained in the next paragraph.

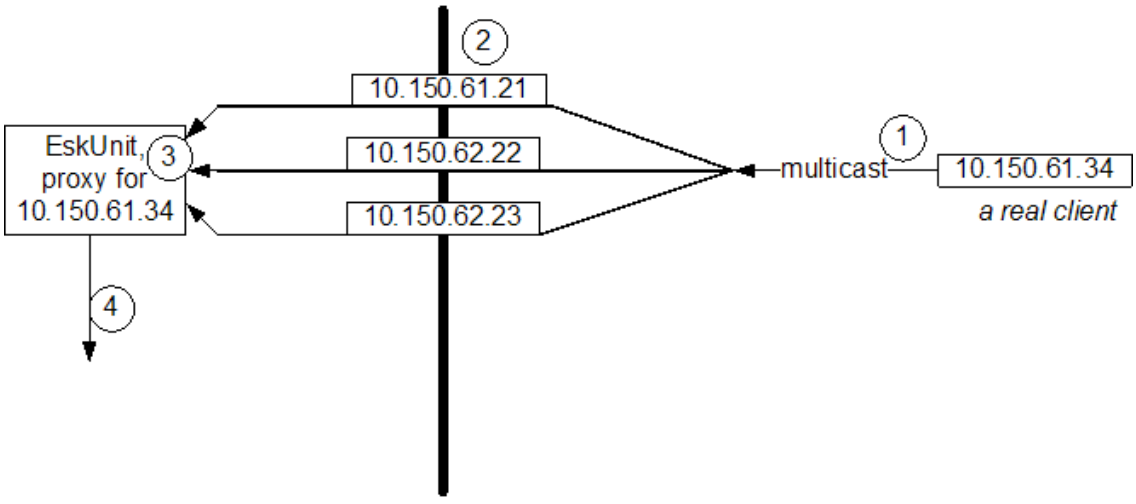


Figure 6.13: The problem with the multicasting messages from the real external devices. See also comments in text.

The proxy should send the packets to the emulated devices on behalf of the external client ([4]). We can not send all passed to the proxy packets in step [4] because it would result in too many messages received

by the emulated services. It is also not possible to detect whether two messages are actually the same or not according to the explanation in the previous paragraph. Thus we decided to sent the packets as a unicast according to the already explained technique used for communication between real and emulated services/clients. As a consequence, the multicast messages from the real clients are accounted incorrectly.

6.7.2 Performance validation

In this section we consider performance characteristics of the *ESKEm*. In this section more complex emulation scenarios are used as in the previous section.

ScriptGenerator We implemented the `ScriptGenerator` for the generation of emulation scripts. It generates randomly the commands for start and stop of the services/clients according to parameters specified by the user. The `rnet.xml` files are created manually.

In short, the `ScriptGenerator` works as follows. Here are the most important parameters which a user can specify:

- the number of devices (both clients and services) in the emulation scenario;
- the probability that a device contains a service and not a client;
- the probability that one of the previously started devices/clients should be stopped;
- the pool of the IP addresses which can be used by the services/clients. For example, the template ”10.150.61.x” specifies the pool of IP addresses where the last ”x” should be replace with a valid number in the range ”0-253” (the addresses like ”x.x.x.254” are reserved for router in the *ESKEm*).
- `MAX_DIFFTIME`, the maximal time difference between two consecutive commands. Lower values of the variable results in more commands for the unit of time.
- the number of the emulation scenario.

The script generator saves both emulation script and commands to configure alias IP interfaces into `./etc/script<num>/script<num>.xml` and `./etc/script<num>/config<num>.xml` files respectively.

Environment The *ESKEm* performance was measured in the hardware and software environment with the following characteristics:

workstation Acer TravelMate 291LMi notebook with Intel Pentium M 1.4Ghz (Centrino) processor, 512Mb RAM

OS GNU/Linux, kernel release 2.6.8-24, machine hardware name: i686

JVM Java 2 Runtime Environment, build 1.5.0_01-b08

The Java Management Extensions (JMX) technology [JMX] was used to measure performance of the Java platform during emulation scripts execution. The JMX support influences the performance of the emulation. As a consequence, the JVM was stared with the JMX support only for the measurement of the memory consumption and number of threads in the system. Other metrics like the number of captured messages were measured with the disabled JMX.

Factors selection

In short, the *independent factors* are the factors which we control. For example, the number of the devices in the emulation scenario. We are interested in the influence of independents factors on *dependant* factors. For example, the memory consumption by the java virtual machine during the emulation is one of such

factors. Here we select both independent and dependant factors which we consider during our experiments. Different value of the independent factor are also referred to as *levels* of the factor.

There are several dependant factors which might be interesting for us. We list all of them here:

real time nominal/actual execution of the real-time emulation commands in the `QueueE` which are read from an emulation script.

the size of the messages queue the size of the messages queue in the `NoDropMessageQueue` of the *JiST/SWANS* network simulator. The `EskIp` packets explained earlier are queued here before they are sent over the simulated network. Many such queues are instantiated during the network simulation.

the size of the simulation events queue the size of the simulation events queue in the `Controller` of the *JiST*.

heap size the memory heap size used by the java virtual machine (JVM) during emulation

CPU consumption of the CPU resources during emulation

messages distribution distribution of the *UPnP Cybergarage* service discovery protocol messages by type

The reason for the monitoring of the queues was the associated technical problems. The size of a queue reflects the needs for the memory during the emulation.

We identified two independent factors which most obviously should influence performance of the emulation:

the burstness of the emulation scenario The concept introduce the idea that different emulation scenarios may specify different number of the emulation commands which are to be executed per unit of time (for example, per minute). We manipulate this factor with the `MAX_DIFFTIME` variable of the `ScriptGenerator`. The variable defines the maximal time between two successive commands in the emulation script. The generator select randomly the time between current and the next command in the range between 0 and `MAX_DIFFTIME`

the number of devices the number of devices (both with services and clients) participating in the emulation scenario

Measurement . As was already mentioned, the JVM performance was measured with the support of the JMX (Java Management Extensions). We used the monitoring facility of this technology. The JVM must be started with a set of parameters in order turn on the JMX support:

java -Dcom.sun.management.jmxremote.port=5001 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false <rest skipped>

Additionally, the `jconsole` must be started which connect to the JVM instance. The `jconsole` is available in the JDK5.0 distribution. The screenshots are provided in the Appendix B. The `jconsole` provides an overview over the JVM parameters like version, loaded libraries, heap size, number of threads, consumed CPU resources etc (see Table B.1).

Additionally, we monitored the size of two different types of queues of the `JiST/SWANS` network simulator which we mentioned when considering the dependant factors. The monitoring was accomplished by a thread running concurrently in respect to the thread with the queue instance. The size of the queue is sampled every 50ms. We are interested in the dynamic of the queue size and the peak values of the queues.

Experiments

There are two independant factors which influence on the performance of the emulation we would like to analyse. As a consequence, we generated different emulation scripts as represented on Figure 5.4. The scripts from 7 to 10 differ in the burstness characteristic of the emulation scenarios. The scripts from 15 to 16 differs in respect to the number of emulation devices. The distribution for the UPnP service discovery messages is provided in Table B.2.

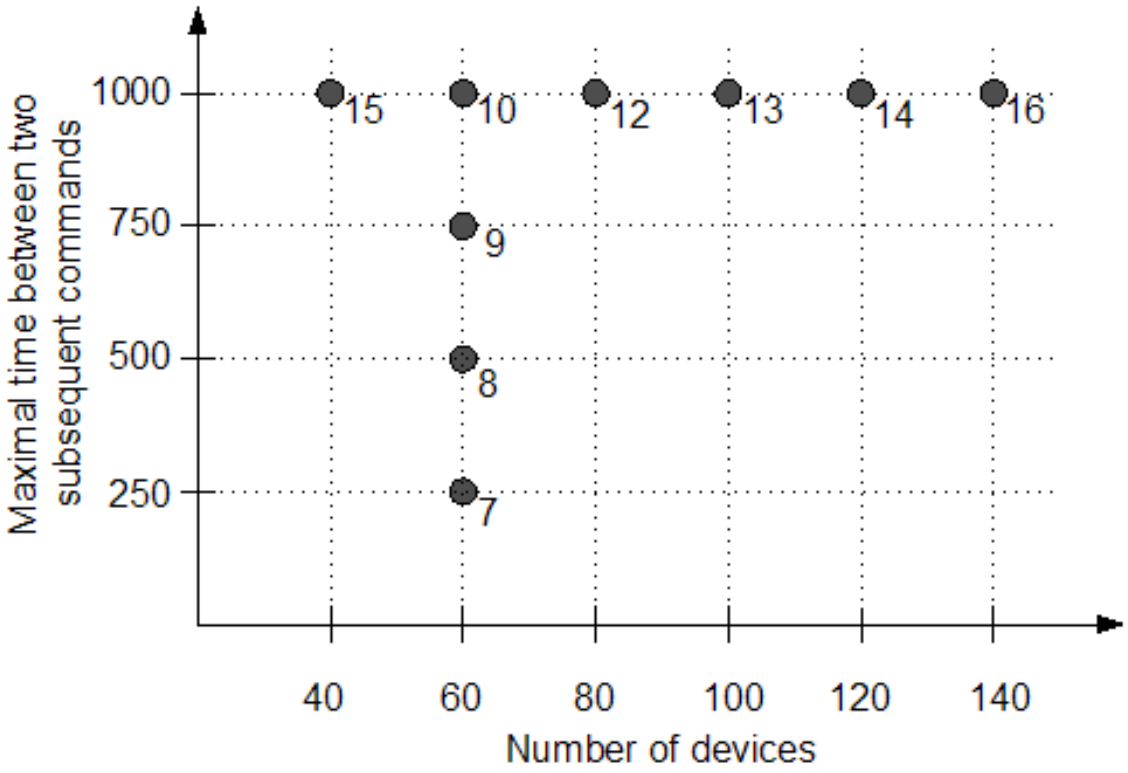


Figure 6.14: The emulation scripts and their characteristics in respect to the number of devices and burstness of the emulation scenario. We refer to the emulation scripts by the numbers like 15, 10 etc.

The results of the experiments are represented compactly in Table B.2 of Appendix B.

Plots But before we start, the plots like those represented on Figure B.1 need to be explanation. We use the plots basically to represent the dynamic of the size of the queues. Each such queue is monitored by a separate thread. The queue size is sampled every 50ms. As a consequence, a textual file with a single very long column is the result of monitoring. The plots which we use is the result of visualization such files.

As was already mentioned, there are two types of queues which we monitor. There is a single instance of the simulation events queue and a number of the `NoDropMessageQueue` instances. Whereas each instance of the latter queue is monitored by from a separate thread. This has the consequence on how the plot looks and how it is interpreted. Special explanation is required for the plots of the message queues. We take Figure B.1 as a typical example. A separate plot like the plot for the script 7 consists of dots organized into the line patterns. Many of such virtual lines overlap. Each such virtual line represents the changes in the state of the message queue.

Burstness of the emulation First, we considered the influence of the *emulation burstness* onto the dependant factors. The independent factor has different levels in the 10th, 9th, 8th and 7th scripts as can be

seen on Figure 5.4. For example, the script 10 has the burstness equal to 1000ms. The number of devices is fixed through the considered scripts. The results of the experiments are represented in Table B.2 (a).

According to Figure B.1 in Appendix B the burstness of the emulation has a moderate influence on the size of the messages queue. But the influence of the factor on the size of the simulation events queue is considerable. As can be seen on Figure B.2, the size of the queue increases with the decreased time interval between consecutive commands of an emulation script. Considerable is also the difference in the size of the queue for the 8th and 9th emulation scripts. We interpret it after explaining some important for understanding details.

The feature of the *ESKEm* design is that the implementation of the real services and clients is used when possible. This results in higher accuracy of the emulation but also in increased resources consumption. The life cycle of the real device implementation can be split into *instantiation* of the device, *start* of the device and *stop* of the device. The design of the *ESKEm* implies that the devices are instantiated before the emulation is start. For this reason the instantiation of the device has no influence on the emulation performance. On the other hand, the start and stop of the devices is the main part of the emulation. We measured the time spent by the devices in three mentioned phases on the sample of 100 devices (samples were done separately for each device). The average time required for the start is about 500ms with the distribution skewed to the left, for the stop of the device 162ms is required.

As a consequence, we propose the following explanation for the differences in the queue size on Figure B.2. The maximal time between two consecutive commands is 500ms, 750ms and 1000ms for the scripts 8, 9 and 10 respectively. This is either equal or greater than 500ms and 162ms required for the start and stop of the device respectively. In the case of the scripts 8, 9 and 10, the simulator has enough resources to manage calculation of the simulation for all messages resulted from the start or stop of the devices. This is accomplished with the moderate size of the events queue. This is not the case when the time between commands is allowed to be lower than 500ms. The computing resources used by the network simulator must be shared with the adapted devices. This results in the significantly increased size of the simulation events queue. As a consequence, the emulation scripts with higher burstness may require more memory to be executed.

As can be seen from Table B.2 (a), the burstness of the emulation script has also significant influence on the real-time characteristics of the emulation. First, we analyzed the nominal and actual execution time of the emulation commands in the real-time `QueueE`. The descriptive statistics for the difference between them is shown in Table B.2 under the column *"Average delay of the emulation commands execution"*. As a consequence of the `QueueE` design, we expect delay for the execution of the emulation commands to be in average about 20ms. Even without careful statistical analysis it is clear that the delay for the script 7 is too high.

Further, we considered the difference between nominal time for the end of the emulation and actual time of processing the last simulation event in the *JiST/SWANS* simulator. The nominal time for the emulation end is associated with the time for the last emulation command specified in the emulation script. The difference between this two times is introduced in the columns *"End of the emulation"* of the table. A negative difference indicates that the simulation was finished to late to be in real time. On the other hand, a positive difference may be an indication that the simulation was fast enough to send the messages over the simulated network in real time or faster.

Number of emulated devices The number of the emulated devices is the next independent factor which we consider in this section. The scripts 15th, 10th, 12th, 13th, 14th and 16th have different levels of this factor as shown on Figure 6.14. The results of the experiments are represented in Table B.2 (b).

As can be seen in the table, the number of emulated devices influences both the size of the simulation events queue and the size of the `NoDropMessageQueue` instances. The dynamic of the first simulation events queue size is shown on Figure B.3. The maximal simulation events queue size is increased significantly with the jump from 80 to 100 of the emulated devices. The same applies to the average delay for the execution of the emulation commands: it is twice as high as the expected delay of 20ms. The end of the

processing the last simulation event in the network simulator lies behind the emulation end as taken from the emulation scenarios with 100, 120 and 140 devices. This is expressed in the negative differences in Table B.2 (b).

Stuck messages Further, we have observed different side effects during the emulation scenarios for the script 13, 14 and 16. The side effects are shortly introduce in the table. The first of them can be called as the *stuck messages* side effect.

The messages in some of the `NoDropMessageQueue` instances stuck in the queue for some time without being processed. This can be observed when zoomed into the plot for the message queues for the script 14 which is represented on Figure B.4. The patterns of the dots grouped in the parallel to the X-axis lines are clearly observable. The pattern means that the size of the messages queue is unchanged for some time. We interpret this pattern as a side effect that the messages are stuck in the queue. The pattern appears seldom for the emulation scenarios with lower number of devices.

The observed phenomenon is likely to be the consequence of the increase load onto the host with the *ESKEm* during emulation. Closer investigation of the problem showed that the queue instances run not in the separate threads but rather in the `JistController` thread which is the thread responsible for the network simulation. What if the message queues would run in the concurrently? In this case the problem would occur too. But we assume, that it would be easier to improve the performance of the network simulator by running it on the host with multiple processors. Whereas at the expense of the network simulation becoming non-deterministic.

Runtime exceptions A specific exception occur with a high probability during execution of an emulations with 120 and more UPnP devices. The exception originates from the *UPnP Cybergarage* implementation:

```
1 Exception in thread "Thread-433" java.lang.ArrayIndexOutOfBoundsException:
2 Array index out of range: 42
3 at java.util.Vector.get(Vector.java:710)
4 at org.cybergarage.xml.NodeList.getNode(NodeList.java:28)
5 at org.cybergarage.upnp.ControlPoint.getDevice(ControlPoint.java:326)
6 at org.cybergarage.upnp.ControlPoint.addDevice(ControlPoint.java:241)
7 at org.cybergarage.upnp.ControlPoint.notifyReceived(ControlPoint.java:517)
8 at org.cybergarage.upnp.ssdp.SSDPNotifySocket.run(SSDPNotifySocket.java:103)
9 at java.lang.Thread.run(Thread.java:595)
```

The exception found with the support of the *ESKEm* demonstrates that our framework can be used to validate the existing service discovery protocol implementations in an emulated environment.

Unprocessed messages The next observation is realted to the execution of the script 16. As can be seen on the plot (a) in Table B.3, the messages are stucked in some of the `NoDropMessageQueue` instances for the rest of the emulation. The messages from the queue results in the simulation events. We could suggest that the simulation events queue is busy but it is not the case - it is empty after certain period of time as shown on the plot (b) in Table B.3. We do not have any suggestion for the problem at the moment. The problem requires more closer analysis.

6.8 Summary

This section provided the implementation details of *ESKEm.* Further, we analyzed different emulation scenario scripts in order to validate our framework. We selected two independent factors and investigated their influence on the *ESKEm* performance.

We considered the steps required to install and run the framework and then introduced the framework implementation details. The description was concentrated around the modules of the framework considered in Chapter 5 about design. We explained in detail the most important classes of the framework like `EskUnit` and `QueueE`. Also the modifications introduced to the reused *JiST/SWANS* network simulator were described.

Several experiments were conducted to validate the framework. First, we used the simple emulation scripts to demonstrate the basic functionality of the *ESKEm*. We mentioned the problem associated with the visualization of the communication between emulated services and external real clients. Further, we used generated emulation scripts to considered the influence of two different independent factors on the performance of the framework.

The increase in the burstness of the emulation scenario is associated with the increase of the size of the simulation events queue, decreased speed of the network simulation and higher delay of the emulation commands execution. The influence of the burstness on the size of the message queues is not considerable according to our experiments. We also associated the burstness of the emulation scenario and the time required for the start and stop of the adapted devices.

The increase number of the emulated devices influence the maximal size of the message queues, maximal size of the events queue. We described different side effects observed for the emulation scripts with 100 and more emulated devices.

As a result of the experiments we conclude that it is possible to emulated about 80 devices (with either services or clients) without undesired side effects. Whereas the emulation commands should be executed with the maximal interval of 500ms or higher. The statement may differ for the environments with other characteristics of the software and hardware than ours. An additional instance of the *ESKEm* can be started on the separate host in order to emulate more devices.

Chapter 7

Summary and Future Work

In this thesis we conceptualized and implemented the emulation framework *ESKEm*. A researcher may use it to explore various implementations of the real service discovery protocols in an environment with the specific characteristics.

In short, the *ESKEm* is designed as follows. We take an existing real service discovery protocol implementation and place it into the *ESKEm* framework. The emulation framework proposes various emulated services and clients which may generate the load on the explored protocol. The framework has a network simulator which proposes the transport layer functionality for the service discovery protocol under consideration. Additionally, the emulation framework provides the means for the emulated services to be discoverable by the real clients.

The rest of this chapter compares the characteristics of the *ESKEm* to the requirements formulated in Section 2.3 and highlight the most important design choices of our emulation framework. We also provide the suggestions for future development of the `ESKEm`.

7.1 Summary

In the following table we provide the summary of the *ESKEm* features as a comparison to the requirement for the framework formulated at the beginning of the diploma thesis in Section 2.3. Please, refer to the list requirements while reading this section. The number of the requirements can be found in the first column of the table.

Require-ment	Summary
B.1	As was already mentioned, a real implementation of the service discovery protocol can be explored with the *ESKEm*. It is also possible to adapt the already existing implementation of services for generation of load for the protocol under consideration. Current design of the framework assumes that both the protocol and the service implementations need to be slightly modified in order to run in the *ESKEm*. Suggestions how the adoption can be accomplished automatically were given.

B.6, B.5	Service discovery operates over the network with specific characteristics. We considered different approaches which could be used to recreate a heterogenouse network environment (Requirement B.5). Different reasons motivated us to adopt the network simulation approach. We adopted the *JiST/SWANS* network simulator for the purpose of simulation the characteristics of the transport and lower network layers. As a consequence, the messages sent from the device A to B are passed through the simulated network. Nevertheless, the communication with the external real devices is ensured with our framework too (Requirement B.6). The messages are pushed to the real network interface of the emulation framework host after the effect of the simulated network is calculated for the message transfer characteristics.
B.8	A researcher may wish to emulate an environment with an arbitrary large number of services. This results in the scalability requirement for the *ESKEm*. We addressed the problem of scalability by using several design choices. The most important of them are mentioned here. First, the scalability requirement was taken into account when selecting an approach to imitate the network system used by the service discovery protocol. We selected the simulation approach. Our expectation is to gain imitation of significantly larger networks than with other approaches like network emulators assumed that the same amount of computing resources is available. Second, the number of emulated services in a real subnet can be increased by starting an additional *ESKEm* instance on a separate host. Third, used for the metrics calculation *announcements* are only accumulated during the emulation phase. The metrics calculation is resources consuming task and it is accomplished *after* the emulation is finished. Fourth, the visualization module can be started on a separate from the *ESKEm* host. The visualization consumes a significant amount of resources. The statements about *ESKEm* performance are done in Section 6.8. According to our experiments, current version of the emulation framework can execute scenarios with about 80 various emulated services and clients without undesirable side effects. The maximal interval between two successive emulation commands should be in average not lower than 500ms. The mentioned constraints apply to hardware and software environment with characteristics as described in Section 6.7. Further, we considered only the `UPnP Cybergarage` version of the service discovery protocol in our experiments.
C.2, C.4	We considered an *office environment* as an example of an environment which the *ESKEm* can reproduce. The network environment used in ”typical” office is reproduced with the network simulator. We have selected the *JiST/SWANS* network simulator and extended it with a primitive simulation of the LAN. This simulation takes into account the topology of the network and the network links characteristics. Our simulation is rather simple but it was not the focus of this work to develop an accurate simulation of the network transport layers.
B.3	Emulation scenarios are described with the *emulation scripts*. These ensure the reproducibility of the emulation scenarios. The emulation script consists of various commands which results in the service discovery related activity. For example, the start of the service usually results in a number of *service announcement* messages .

B.2	We provided a flexible measurement infrastructure in our emulation framework. Various gauges and announcements make up the core of the infrastructure. The announcements can be instantiated in virtually any part of the *ESKEm.* A specific gauge listen for the set of specific announcements. The gauge decides which metrics should be calculated from the received announcements. The announcements are accumulated in the gauges during the emulation. Various metrics are calculated by the gauges after the emulation is finished. Thus the calculation of the metrics does not consume valuable resources during the emulation. Researchers will usually needs to implement custom gauges and announcements in order to measure specific metrics.
B.4	The visualization for the *ESKEm* is conceptualized and implemented as a separate *EmulGUI* module. Such design gives additional advantages in terms of the framework scalability. The *EmulGUI* is based on the *Prefuse* framework specialized on the visualization of the graph-based structures. The simulated network topology, start/stop of service or clients and communication among them is visualized as the result of processing the visualization commands sent from the *ESKEm* to the *EmulGUI* as text in the UDP packets. The visualization is accomplished in real time. Both emulated services/clients and real clients are visualized with the *EmulGUI.*
C.3	One of the requirements was to implement the framework in the Java programming language. Further, for the purpose of the framework validation the Java-based implementation of the `UPnP` protocol was used. All this together with the java-based network simulator allowed us to make specific to the *ESKEm* design choices. For example, the service discovery messages are captured at the *transport* network layer before they are fed into the simulated network. As a consequence, it is easy to adopt an existing java-based service discovery protocol in the `ESKEm`. The adoption of the protocols implemented in other programming languages may be a technically challenging task.
B.7	We have validated our approach with the *UPnP Cybergarage* version of the UPnP service discovery protocol. This implementation is used to demonstrated in practice how our approach works. The statement about validation were already done when considering previous requirements from current table.

Table 7.1: Summary: comparison of the requirements with the features of the *ESKEm*.

7.2 Future Work

The future development of the `ESKEm` is likely to be motivated by the needs of researchers which will use the framework. Nevertheless, there are still general suggestions which we want to introduce in this section. We concentrate on those of them which are related to design of the emulation framework. Only the most significant of our suggestions will be introduced.

Scripts generation Manual creation of the emulation scripts becomes complicated very fast with the number of emulated services. Currently, we have wrote a primitive script generator `ScriptGenerator` to create emulation scenario scripts. The automatic generation of scripts is much more convenient. But this approach can be improved significantly. For example, different probability distributions could be used to describe the duration of a service being available. The idea is to provide an *abstract* description language

to describe the emulation scenarios. The descriptions in this language could be further used to generate the emulation scripts in their current form.

Connection-oriented transport Used for the framework validation version of the UPnP protocol is built over the UDP transport. But there are other protocols which implementation is based on the connection-oriented transport. The *ESKEm* must be extended and validated for the protocols using such type of the network transport.

Distributed mode The *ESKEm* can run on the separate hosts. In this way an arbitrarily large number of services can be emulated. But the scripts executed on the separate hosts are independent from each other. We assume that a *Script Manager* should be developed for the coordinated execution of a single emulation script in the various instances of the *ESKEm.* The real time scheduler of the *ESKEm* was developed to facilitate development of such manager. Actually, the challenge is to provide the algorithms which would split the original emulation script so that the load on the *ESKEm* instances is minimized.

Appendix A

Emulation scenario example.

In this Appendix the output for scenario 3 is provides. We provide here both commented textual output and graphical output.

Emulation script for scenario 3:

start dev1 device 10.150.61.21 1000
start dev2 device 10.150.62.22 5000
start dev3 device 10.150.62.23 9000
start clnt1 client 10.150.63.23 11000

Commented output generated by the *ESKEm* for the Scenario 3. The output is produced by the `MessagesGauge` described earlier. The semantic of the columns is as follows.

1 order, in which messages were registered in the `MessageGauge`

2 IP address of the node, which processed the message. In other words, it contains the IP of the source if the message is marked as ”sent”. Or it contains the IP of the destination if the message is marked as ”recv” (see column 4).

3 IP address of the counterpart node (see explanation to the previous column). The IP address here can be either multicast (like ”239.255.255.250”) or unicast.

4 says, whether the message was ”sent” or ”recv” (received).

5 simulation time of the message processing.

6 number of passed hops/routers if applicable. We defined the number of hops only for the unicast messages.

7 unicast message delay on the way from the source to the destination node. The difference is calculated in simulation time. Compare this with our notes about different definitions of time in Section 4.2.1.

8 message size (in bytes)

9 UPnP message type (see UPnP description in Section 3.4.1);

```
=> device dev1 is started, but the other node are unavailable, so nobody receives
   multicast messages. After being started the "byebye" messages (one message
   for root device and for every available services) are sent followed by the
   "announce" messages.
0 |10.150.61.21 |239.255.255.250|sent|1000 |multicast|.|. |152 |byebye
1 |10.150.61.21 |239.255.255.250|sent|1000 |multicast|.|. |179 |byebye
2 |10.150.61.21 |239.255.255.250|sent|1000 |multicast|.|. |181 |byebye
3 |10.150.61.21 |239.255.255.250|sent|1000 |multicast|.|. |287 |announce
4 |10.150.61.21 |239.255.255.250|sent|1000 |multicast|.|. |314 |announce
5 |10.150.61.21 |239.255.255.250|sent|1000 |multicast|.|. |316 |announce

=> device dev2 is started, its byebye and announce messages are received by
   already available dev1
6 |10.150.62.22 |239.255.255.250|sent|5000 |multicast|.|. |152 |byebye
7 |10.150.61.21 |10.150.62.22   |recv|5010 |multicast|0|10|152 |byebye
8 |10.150.62.22 |239.255.255.250|sent|5010 |multicast|.|. |179 |byebye
9 |10.150.61.21 |10.150.62.22   |recv|5020 |multicast|0|10|179 |byebye
10|10.150.62.22 |239.255.255.250|sent|5020 |multicast|.|. |181 |byebye
11|10.150.61.21 |10.150.62.22   |recv|5030 |multicast|0|10|181 |byebye
12|10.150.62.22 |239.255.255.250|sent|5030 |multicast|.|. |287 |announce
13|10.150.61.21 |10.150.62.22   |recv|5040 |multicast|0|10|287 |announce
14|10.150.62.22 |239.255.255.250|sent|5040 |multicast|.|. |314 |announce
15|10.150.61.21 |10.150.62.22   |recv|5050 |multicast|0|10|314 |announce
16|10.150.62.22 |239.255.255.250|sent|5050 |multicast|.|. |316 |announce
17|10.150.61.21 |10.150.62.22   |recv|5060 |multicast|0|10|316 |announce

=> device dev3 is started, its byebye and announce messages are
   received by already available dev2 and dev1
18|10.150.62.23 |239.255.255.250|sent|9000 |multicast|.|. |152 |byebye
19|10.150.62.22 |10.150.62.23   |recv|9010 |multicast|0|10|152 |byebye
20|10.150.61.21 |10.150.62.23   |recv|9020 |unicast  |0|20|152 |byebye
21|10.150.62.23 |239.255.255.250|sent|9020 |multicast|.|. |179 |byebye
22|10.150.62.22 |10.150.62.23   |recv|9030 |multicast|0|10|179 |byebye
23|10.150.61.21 |10.150.62.23   |recv|9040 |unicast  |0|20|179 |byebye
24|10.150.62.23 |239.255.255.250|sent|9040 |multicast|.|. |181 |byebye
25|10.150.62.22 |10.150.62.23   |recv|9050 |multicast|0|10|181 |byebye
26|10.150.61.21 |10.150.62.23   |recv|9060 |unicast  |0|20|181 |byebye
27|10.150.62.23 |239.255.255.250|sent|9060 |multicast|.|. |287 |announce
28|10.150.62.22 |10.150.62.23   |recv|9070 |multicast|0|10|287 |announce
29|10.150.61.21 |10.150.62.23   |recv|9080 |unicast  |0|20|287 |announce
30|10.150.62.23 |239.255.255.250|sent|9080 |multicast|.|. |314 |announce
31|10.150.62.22 |10.150.62.23   |recv|9090 |multicast|0|10|314 |announce
32|10.150.61.21 |10.150.62.23   |recv|9100 |unicast  |0|20|314 |announce
33|10.150.62.23 |239.255.255.250|sent|9100 |multicast|.|. |316 |announce
34|10.150.62.22 |10.150.62.23   |recv|9110 |multicast|0|10|316 |announce
35|10.150.61.21 |10.150.62.23   |recv|9120 |unicast  |0|20|316 |announce

=> client clnt1 is started. It sends the search request message of type
   "search" to discover available devices and services.
   The search request is received by the dev1, dev2 and dev3
36|10.150.63.23 |239.255.255.250|sent|11000|multicast|.|. |101 |search
37|10.150.61.21 |10.150.63.23   |recv|11010|multicast|0|10|101 |search
38|10.150.62.23 |10.150.63.23   |recv|11020|unicast  |0|20|101 |search
39|10.150.62.22 |10.150.63.23   |recv|11030|unicast  |0|30|101 |search

   the devices response with appropriate messages
40|10.150.63.23 |10.150.62.22   |recv|11050|unicast  |2|20|379 |response
41|10.150.63.23 |10.150.61.21   |recv|11080|unicast  |3|30|379 |response
42|10.150.63.23 |10.150.62.23   |recv|11100|unicast  |2|20|379 |response

=> a search message from an external real client is received.
43|10.150.62.23 |10.150.61.34   |recv|11120|unicast  |2|20|1024|search
44|10.150.61.21 |10.150.61.34   |recv|11130|unicast  |1|10|1024|search
45|10.150.62.22 |10.150.61.34   |recv|11150|unicast  |2|50|1024|search

=> the emulated dev1, dev2 and dev3 send their reponces to the real client
46|10.150.61.34 |10.150.62.22   |recv|11170|unicast  |2|20|379 |response
47|10.150.61.34 |10.150.62.23   |recv|11190|unicast  |2|20|379 |response
48|10.150.61.34 |10.150.61.21   |recv|11200|unicast  |1|10|379 |response
```

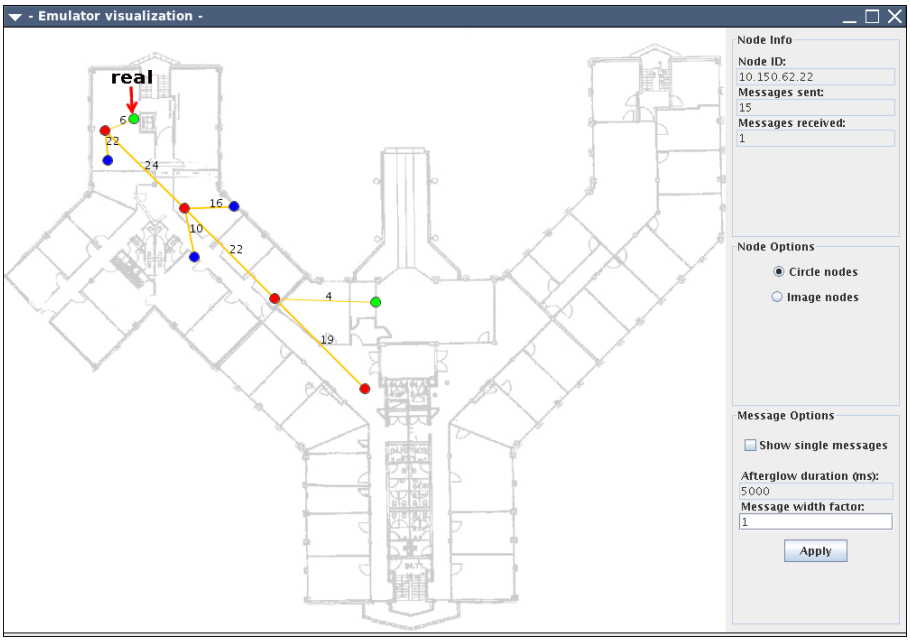


Figure A.1: The graphical representation of the emulation scenario with the *EmulGUI*.

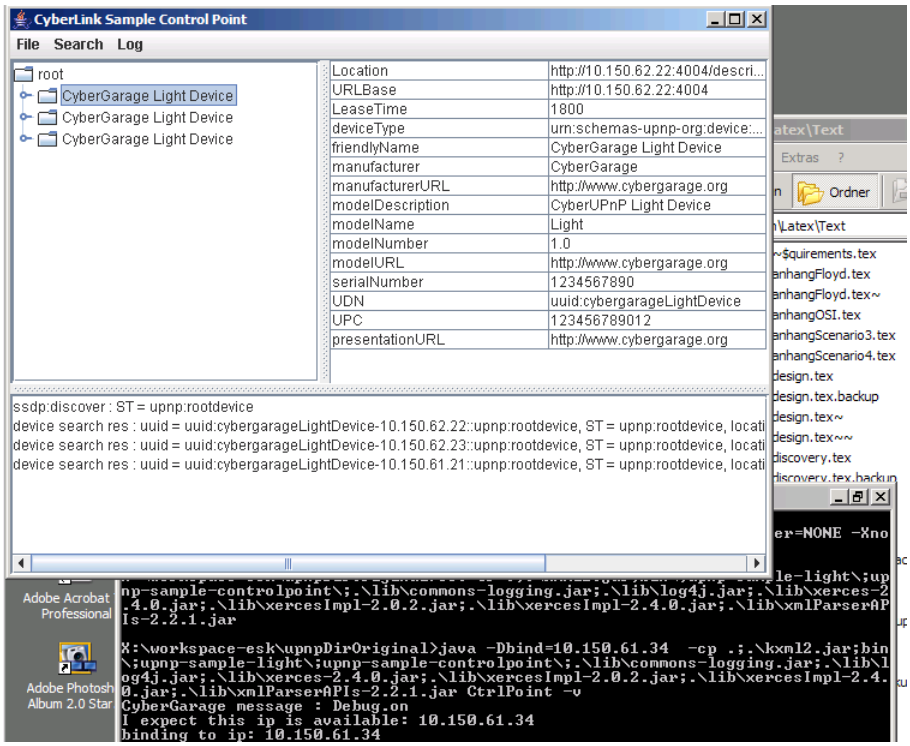


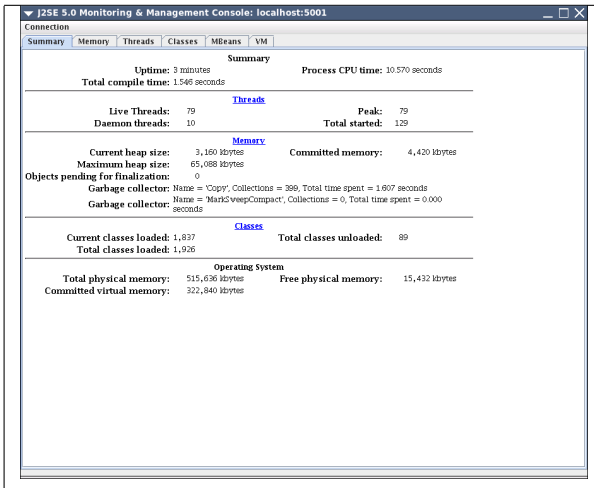
Figure A.2: The client started on the separate WinXP-based host. The client is marked with the arrow on Figure A.1.

Short form of the output, printed by the `SearchingGauge`:

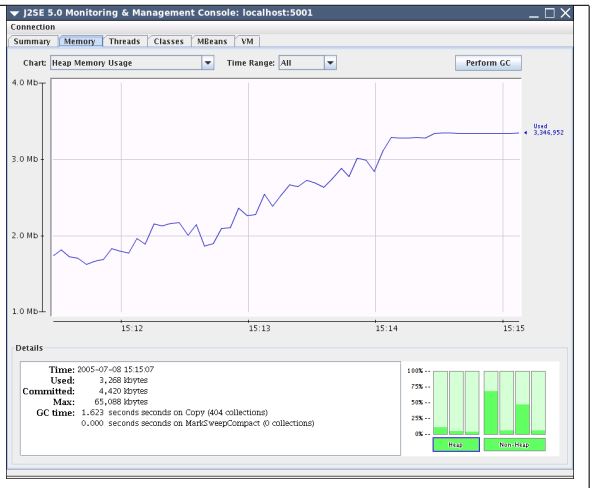
```
=> statistics for the emulated clients.
   Note: we can't calculate the metric
         for a real client, because not
         all information about available
         knowledge of the client is available

found statistics: [10.150.63.23 = 3]

----------------------
=> distribution of messages
   after the emulation

announce, sent   =>  9
announce, recv   =>  9
response, recv   =>  6
byebye, sent     =>  9
search, sent     =>  1
search, recv     =>  6
byebye, recv     =>  9
```

Appendix B

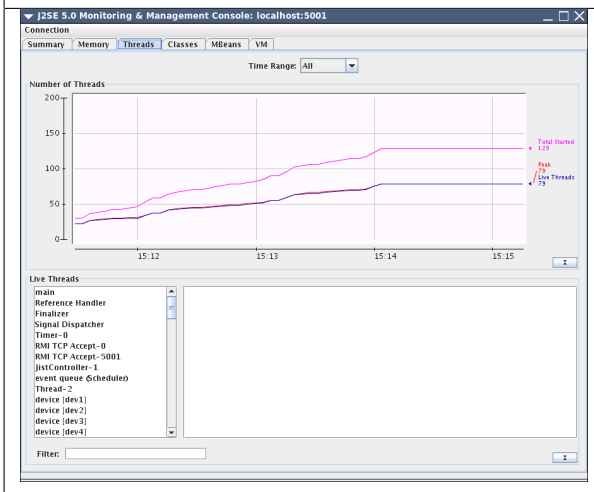
ESKEm performance measurements.



(a) overview over different performance characteristics



(b) memory monitoring



(c) monitoring of the threads

Table B.1: The Java Management Extensions (JMX) and the *jconsole* tool to monitor performance of the Java Virtual Machine (JVM).

Script	Factor levels (burstness)	Number of events in simulator: max	End of the emulation (sec.): Actual	Nominal	Difference	Average delay of the emulation commands execution: mean	max	mode	Messages queue: max
7	250	2080	15.8	9.7	-6.1	225	1595	16	300
8	500	560	18.3	20.6	2.3	26	211	7	260
9	750	440	29.0	30.8	1.8	36	290	10	280
10	1000	140	34.5	42.9	8.4	20	150	2	250

(a)

Script	Factor levels (burstness)	Number of events in simulator queue: max	End of the emulation (sec.): Actual	Nominal	Difference	Average delay of the emulation commands execution: mean	max	mode	Messages queue: max	
15	40	180.00	23.8	27.2	3.4	19	112	9	200	
10	60	140.00	34.5	42.9	8.4	20	150	2	250	
12	80	340.00	45.9	54.8	8.9	26	293	5	330	
13	100	3160.00	80.0	65.4	-14.6	51	580	22	480	*
14	120	4200.00	95.0	84.0	-11.0	148	1384	18	600	**
16	140	8900.00	130.0	99.7	-30.3	55	727	20	680	***

** the messages hang in the messages queue for a long time*

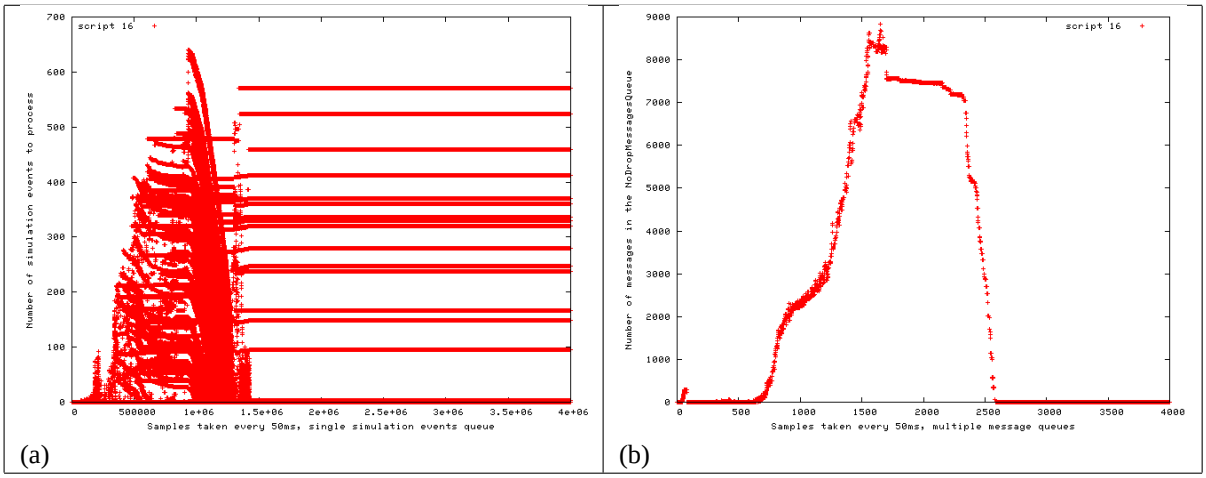
*** the probability for the exceptions in the simulator and in the UpnP Cybergarage is high*

**** the messages hang in the queue even if the events queue is empty*

You will find more explanations in the text

(b)

Table B.2: Summary table of experiments conducted for the *UPnP Cybergarage* with the *ESKEm*. See also Figure 6.14.



(a) (b)

Table B.3: Comparison of the simulator message queues size (a) and simulation events queue size (b) for the emulation script 16.

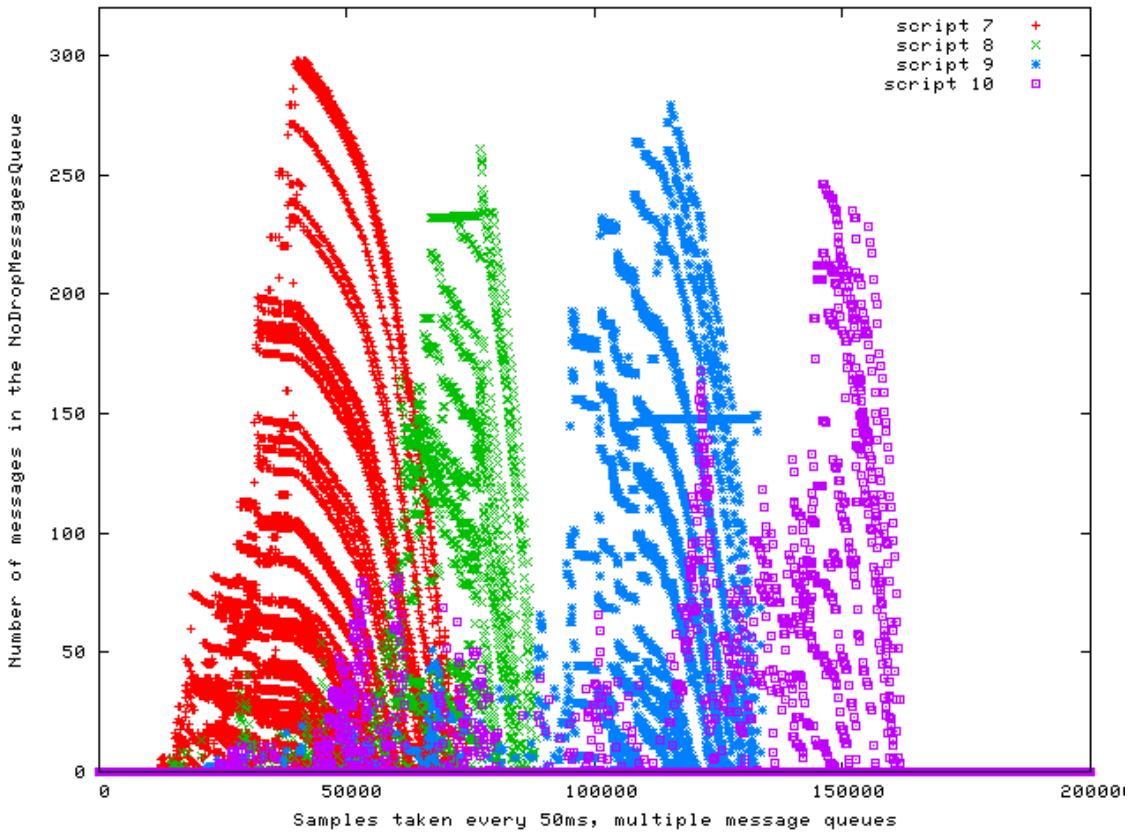


Figure B.1: Dynamic of the `NoDropMessagesQueue` size for different levels of the *"burstness of the emulation"* factor.

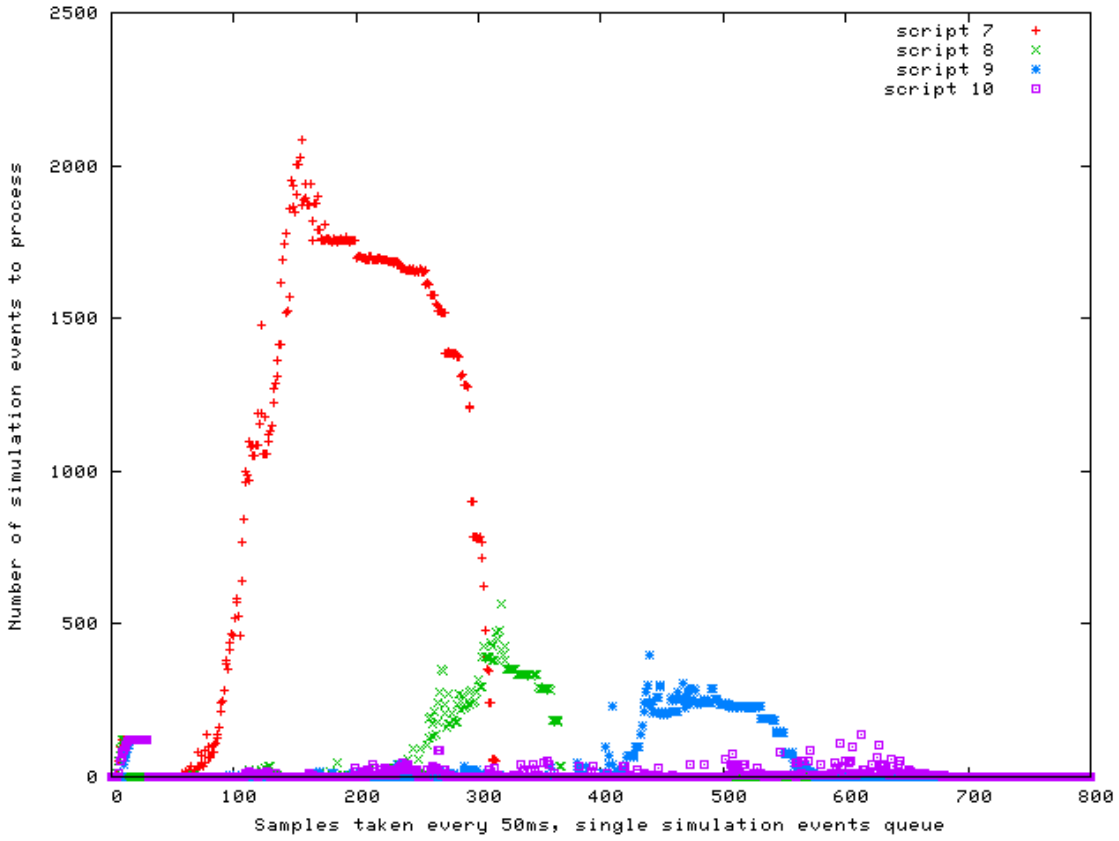


Figure B.2: Dynamic of the simulation events queue size for different levels of the *"burstness of the emulation"* factor.

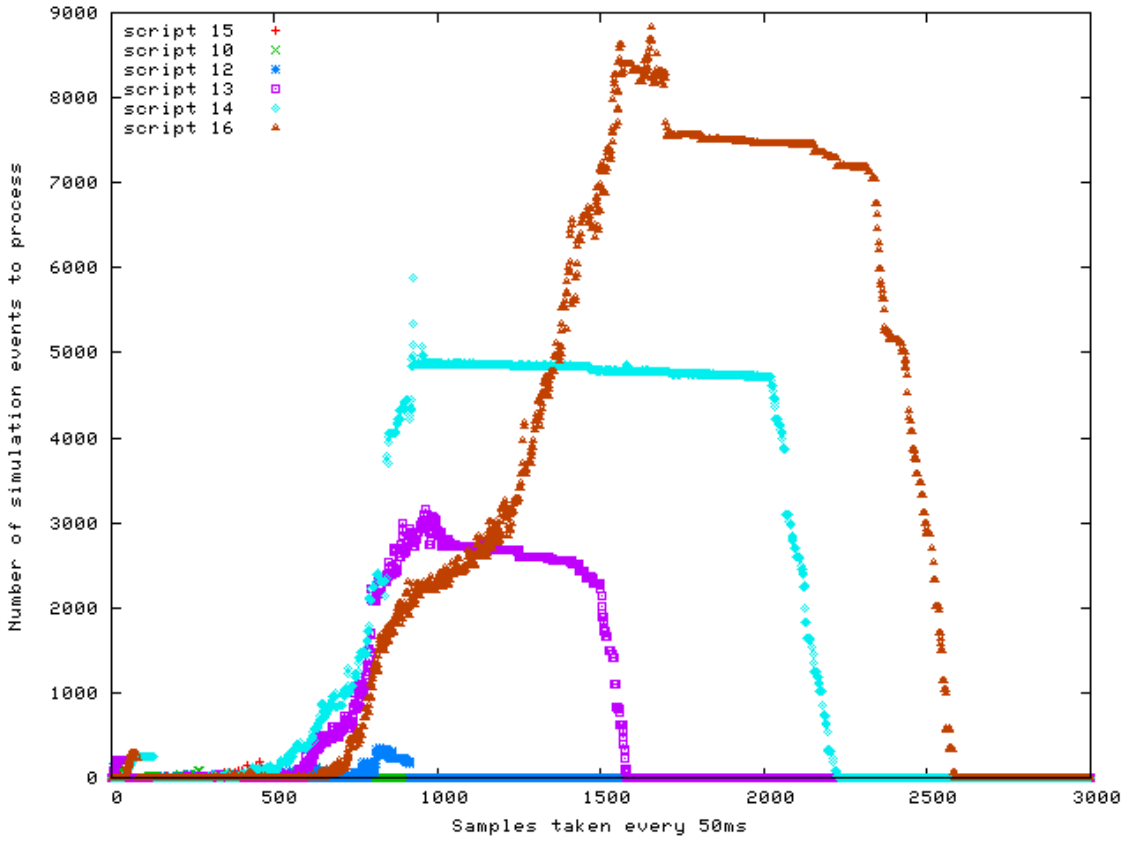


Figure B.3: Dynamic of the simulation events queue size for different levels of the factor *"number of emulated devices"*.

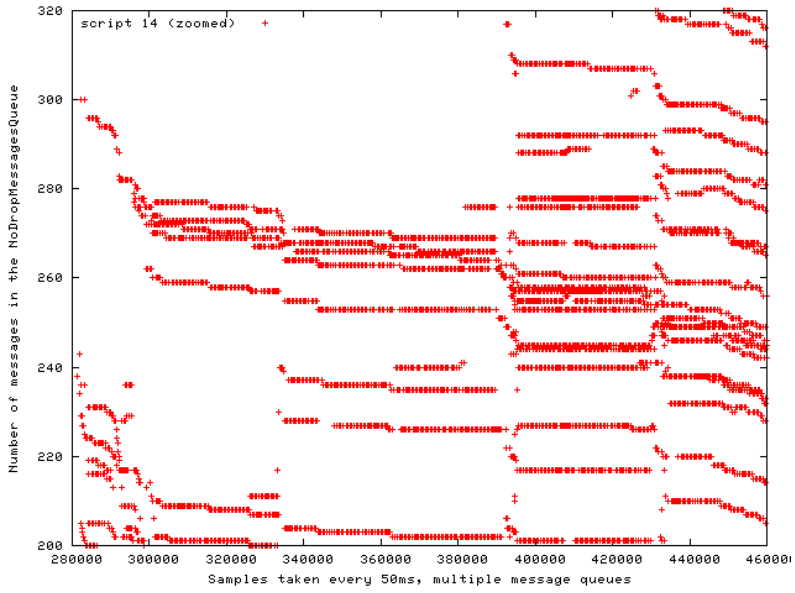


Figure B.4: Dynamic of the `NoDropMessagesQueue` size for the factor *"number of emulated devices"* in the script 14.

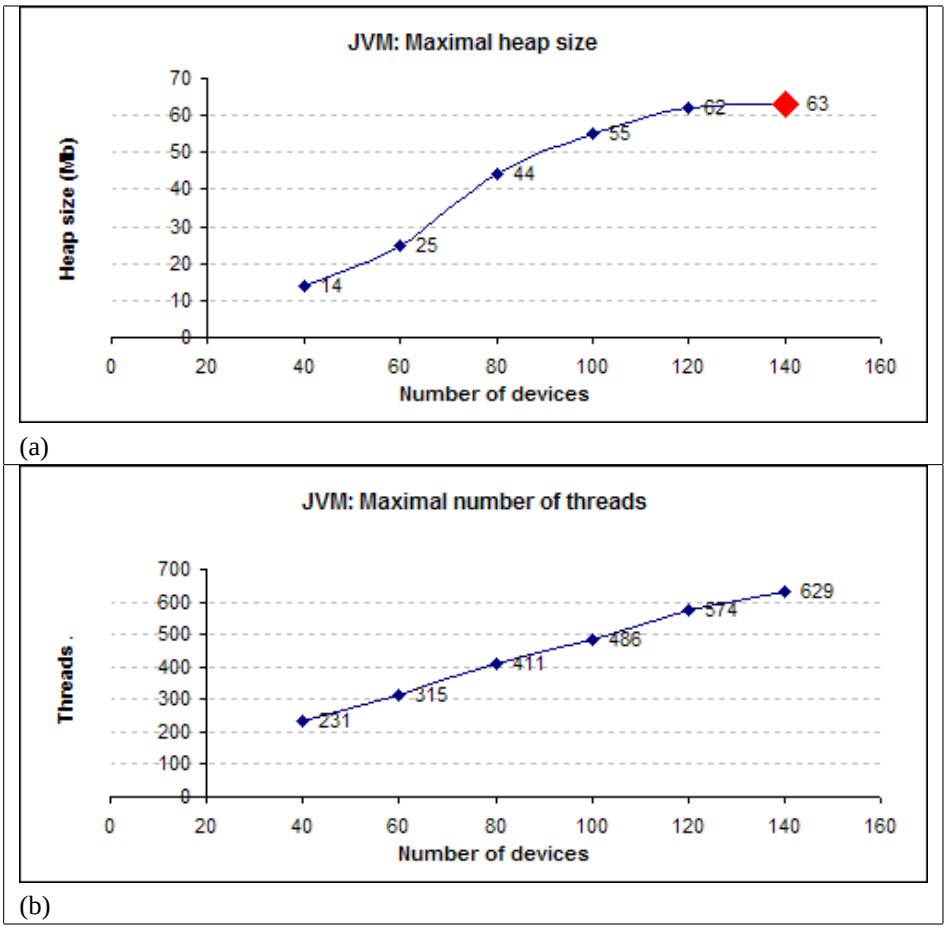
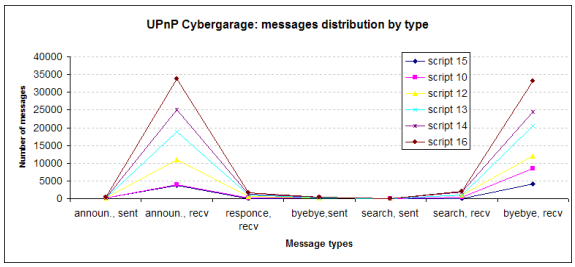


Table B.4: JMX: java heap size (a) and the number of threads (b) for the emulation scripts with different number of devices.

Script	announ., sent	announ., recv	responce, recv	byebye, sent	search, sent	search, recv	byebye, recv
script 15	108	3687	43	120	4	51	4159
script 10	153	3923	281	182	9	341	8433
script 12	180	10898	708	210	20	1118	12119
script 13	246	18881	949	276	18	1132	20451
script 14	288	25126	1250	328	24	1767	24442
script 16	342	33776	1624	387	26	2051	33159
script 7	138	3588	200	144	13	314	3256
script 8	138	3159	288	161	14	369	3619
script 9	138	3064	394	153	14	481	3564
script 10	153	3923	156	177	9	185	4534

(a)



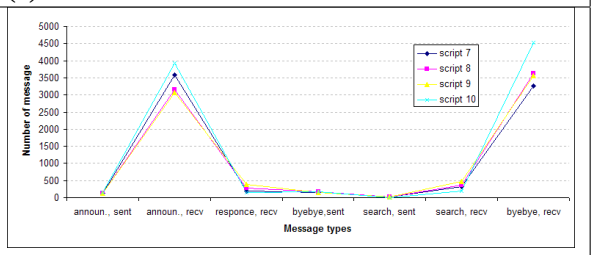


Table B.5: The *UPnP Cybergarage* messages distribution for the scripts from 15 to 16 (b) and from 7 to 10 (b).

Appendix C

The OSI reference model.

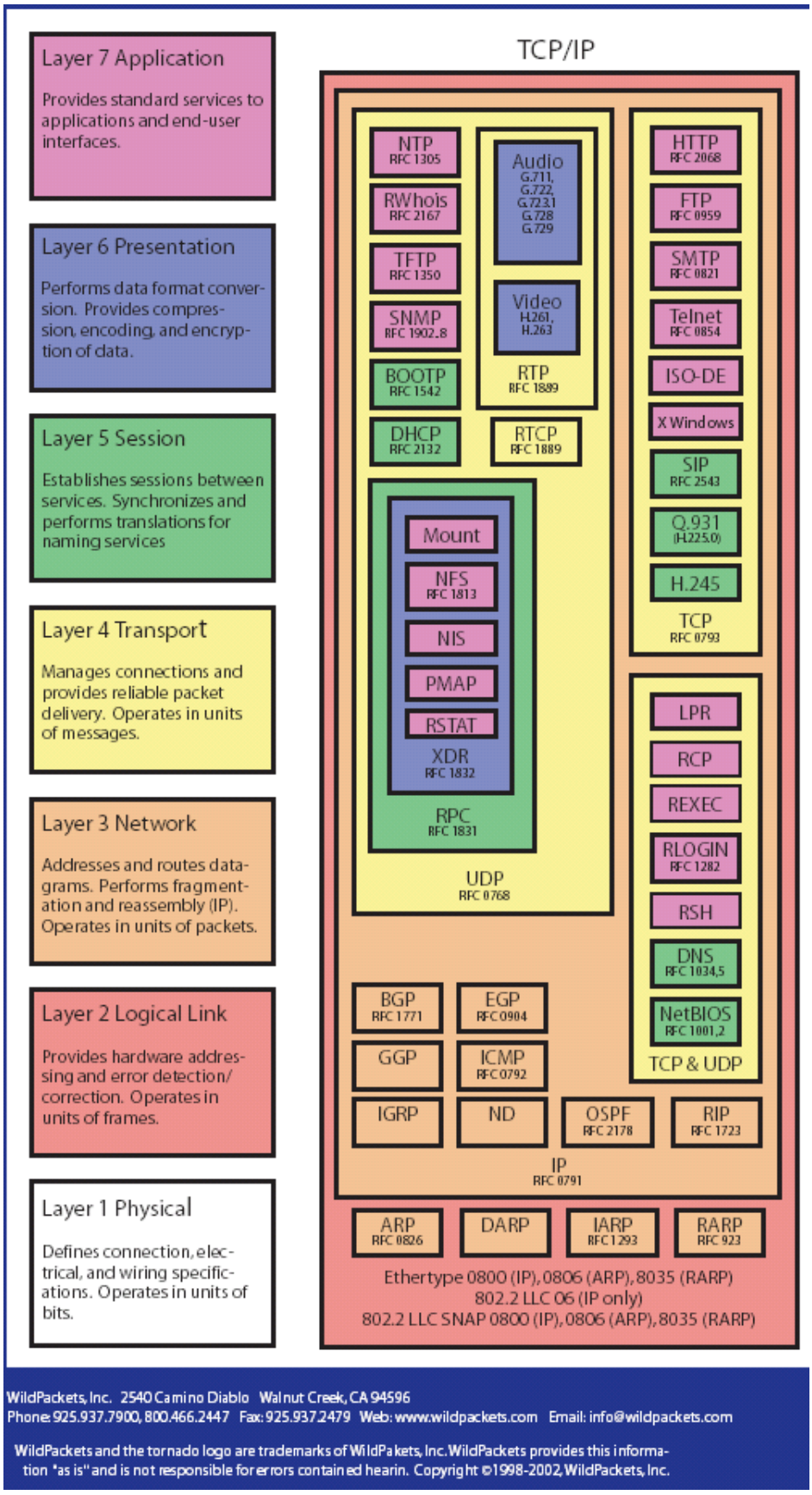


Figure C.1: OSI Layers

Appendix D

The Floyd-Warshall algorithm.

The pseudo-algorithm of the Floyd-Warshall algorithm:

```
function fw(int[0..n,0..n] graph) {
    // Initialization
    var int[0..n,0..n] dist := graph
    var int[0..n,0..n] pred
    for i from 0 to n
        for j from 0 to n
            if dist[i,j] > 0
                pred[i,j] := i
    // Main loop of the algorithm
    for k from 0 to n
        for i from 0 to n
            for j from 0 to n
                if dist[i,j] > dist[i,k] + dist[k,j]
                    dist[i,j] = dist[i,k] + dist[k,j]
                    pred[i,j] = pred[k,j]
    return dist
}
```

Glossary

ALS application level simulation

DHCP Dynamic Host Configuration Protocol

GUI Graphical User Interface

ISP internet service provider

JFC Java Foundation Classes

JiST Java in Simulation Time

JVM java virtual machine, see http://java.sun.com

NDP node discovery protocol

NIC network interface card

P2P peer-to-peer network

PDA Personal Digital Assistant

SD Service Discovery

SLP Service Location Protocol

SSDP Simple Service Discovery Protocol

SWANS Scalable Wireless Ad hoc Network Simulator

Bibliography

[Abow 99] ABOWD, GREGORY D.: *Software engineering issues for ubiquitous computing*. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 75–84, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[ACM] *The ACM (Association for Computing Machinery) Digital Library*, `http://portal.acm.org/` .

[AES] *Advanced Environmental Systems*, `http://www.aesinc.com/revamp/inf/got_p_rv.htm` .

[AlJi 03] EITAN ALTMAN AND TANIA JIMENEZ: *NS Simulator for beginners*, December 2003, "http://www-sop.inria.fr/maestro/personnel/Eitan.Altman/COURS-NS/n3.pdf" . Lecture notes, 2003-2004.

[Bajaj et al. 99a] SANDEEP BAJAJ, LEE BRESLAU, DEBORAH ESTIN, KEVIN FALL, SALLY FLOYD, PADMA HUANG, SATISH KUMAR, STEVEN MCCANNE, REZA REJAIE, PUNEET SHARMA, KANNAN VARADHAN, YA XU, HAOBO YU, DANIEL ZAPPALA: *Improving Simulation for Network Research.* 2003.

[Barb 00] BARBEAU, MICHEL: *Bandwidth Usage Analysis of Service Location Protocol*. School of Computer Science, Carleton University.

[BCEL] *BCEL: The Byte Code Engineering Library*, http://jakarta.apache.org/bcel/ .

[Bett 00] CHRISTIAN BETTSTETTER AND CHRISTOPH RENNER: *A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol*, 2000.

[BiHe et al. 04] HANNES BIRCK, OLIBER HECKMANN, ANDREAS MAUTHE, RALF STEINMETZ: *Analysis of Overlay Networks at Message- and Packet-Level*, March 2004.

[Bjoe 05] SCHILLING, BJÖRN: *Modeling and Simulation of Computer Systems: Qualitative Comparison of Network Simulation Tools*. Institute of Parallel and Distributed Systems, University of Stuttgart.

[Bres 00a] LEE BRESLAU AND DEBORAH ESTRIN AND KEVIN FALL AND SALLY FLOYD AND JOHN HEIDEMANN AND AHMED HELMY AND POLLY HUANG AND STEVEN MCCANNE AND KANNAN VARADHAN AND YA XU AND HAOBO YU: *Advances in Network Simulation*. IEEE Computer, 33(5):59–67, May 2000, http://www.isi.edu/ johnh/PAPERS/Breslau00a.html . Expanded version available as USC TR 99-702b at `http://www.isi.edu/~johnh/PAPERS/Bajaj99a.html`.

[Brey 03] BREYER, TOBIAS: *Modellierung der Bewegung und des Verhaltens von Dienstnutzern in mobilen Ad-hoc-Netzen*, 2003. Diplomarbeit.

[CaSa] MARK CARSON, DARRIN SANTAY: *NIST Net - A Linux-based Network Emulation Tool*. Washington, DC, USA. .

[CaSaSch 02] D. CAVIN, Y. SASSON, A. SCHIPER: *On the Accuracy of MANET Simulators.* 2002.

[Chak 02] D. CHAKRABORTY AND A. JOSHI AND T. FININ AND Y. YESHA: *GSD:A novel groupbased service discovery protocol for MANETS*, 2002, citeseer.ist.psu.edu/chakraborty02gsd.html .

[Chun 04] CHUNG, CHRISTOPHER A.: *Simulation Modeling Handbook: A Practical Approach*. CRC Press LLC, ISBN 0-8493-1241-8, 2004, http://... . 2nd Edition.

[Citeseer] *Citeseer*, http://sherry.ifi.unizh.ch .

[DaMi 01] *Analyzing Properties and Behavior of Service Discovery Protocols Using and Architecture-Based Approach*, 2001. Proceedings of Working Conference on Complex and Dynamic Systems Architecture.

[DBCF 95] DAVIES, NIGEL, GORDON S. BLAIR, KEITH CHEVERST and ADRIAN FRIDAY: *A Network Emulator to Support the Development of Adaptive Applications*. In *MLICS '95: Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, pages 47–56, Berkeley, CA, USA, 1995. USENIX Association.

[Deer] DEERING, STEVE: *The Internet Group Management Protocol (IGMP)*, `http://www.cis.ohio-state.edu/htbin/rfc/rfc1112.html` .

[DHCP] *Dynamic Host Configuration Protocol (DHCP)*, http://www.dhcp.org/ .

[DIANEmu] *DIANEmu - Dienste in Ad-hoc-Netzen*, http://www.ipd.uka.de/ kleinm/diane/ .

[DIANEmu 04] MICHAEL KLEIN, DANIEL MATHEIS, MARKUS HOFFMAN, MICHAEL MÜSSIG: *DIANEmu Manual: A High-level Ad hoc Network Simulator*, 2004.

[DME 02] DABROWSKI, CHRISTOPHER, KEVIN MILLS and JESSE ELDER: *Understanding consistency maintenance in service discovery architectures during communication failure*. In *WOSP '02: Proceedings of the third international workshop on Software and performance*, pages 168–178, New York, NY, USA, 2002. ACM Press.

[EHAB 99] ESLER, MIKE, JEFFREY HIGHTOWER, TOM ANDERSON and GAETANO BORRIELLO: *Next century challenges: data-centric networking for invisible computing: the Portolano project at the University of Washington*. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 256–262, New York, NY, USA, 1999. ACM Press.

[EnBuMa 05] *A Survey of Software Infrastructures and Frameworks for Ubiquitous Computing*, January 2005.

[Ethereal] *Ethereal: A Network Protocol Analyzer*, http://www.ethereal.com/ .

[Floyd-Warshall] *Floyd-Warshall algorithm*, `http://www.aesinc.com/revamp/inf/got_p_rv.htm` .

[GloMoSim] *GloMoSim Manual, v1.2*, http://pcl.cs.ucla.edu/projects/glomosim/GloMoSimManual.html .

[GoBa 00] JAVIER GOVEA, MICHEL BARBEAU: *Comparison of Bandwidth Usage: Service Location Protocol and Jini*, 2000, citeseer.ist.psu.edu/govea00comparison.html .

[Goet 02] GOETZ, BRIAN: *Java theory and practice: Thread pools and work queues*, http://www-106.ibm.com/developerworks/java/library/j-jtp0730.html .

[GoF 95] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON, JOHN VLISSIDES: *Design Patterns*. Addison-Wesley Professional; 1st edition, ISBN 0-201-63361-2, 1995. 395 p.

[Gold 02] GOLDEN R. RICHARD III: *Service Advertisement and Discovery: Enabling Universal Device Cooperation*. University of New Orleans, 2002. IEEE Internet Computing.

[Google Scholar] *Google Scholar*, http://scholar.google.com .

[HaDe 03] EMIR HALEPOVIC, RALPH DETERS: *The Costs of Using JXTA*.

[HeAbNe 99] HEGERING, H.-G., ABECK, S.: *Integrated Management of Networked Systems – Concepts, Architectures and their Operational Application*. Morgan Kaufmann Publishers, ISBN 1-55860-571-1, 1999. 651 p.

[Heid 00d] USC/INFORMATION SCIENCES INSTITUTE: *Effects of Detail in Wireless Network Simulation*, September 2000, http://www.isi.edu/ johnh/PAPERS/Heidemann00d.html . Submitted to SCS Communication Networks and Distributed Systems Modeling and Simulation Conference.

[HeRo 02] DANIEL HERRSCHER, KURT ROTHERMEL: *A Dynamic Network Scenario Emulation Tool*. IPVR, University of Stuttgart.

[Herr 02] *Modeling Computer Networks for Emulation.*, 2002.

[Hong 04] HONGHUI LUO AND MICHEL BARBEAU: *Performance Evaluation of Service Discovery Strategies in Ad Hoc Networks*. In *CNSR '04: Proceedings of the Second Annual Conference on Communication Networks and Services Research (CNSR'04)*, pages 61–68, Washington, DC, USA, 2004. IEEE Computer Society.

[Jain 91] JAIN, R.: *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley- Interscience, New York, NY, 1991, http://www.cse.ohio-state.edu/ jain/books/perfbook.htm .

[JDO] *JDO: Java Data Objects*, http://java.sun.com/products/jdo/index.jsp .

[Jini] *Jini Reference Implementation*, http://www.sun.com/jini .

[JIST] BARR, RIMON: *JiST - Java in Simulation Time (User Guide)*, 2003-2004, http://jist.ece.cornell.edu/docs.html .

[JMeter] *Apache JMeter - load test functional behavior and performance measurement.*, http://jakarta.apache.org/jmeter/ .

[JMX] *Java Management Extensions (JMX)*, `http://java.sun.com/products/JavaManagement/` .

[JoFa 99] "MOHAMED E. FAYAD, RALPH E. JOHNSON": *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley & Sons; 1 edition; ISBN: 0471248754, 1999, http://www.wiley.com/legacy/compbooks/catalog/24875-4.htm .

[JSim 03] *Evaluation of J-Sim*, http://www.j-sim.org/comparison.html .

[Junw 01] JUNWEI CAO, DARREN J. KERBYSON AND GRAHAM R. NUDD: *High Performance Service Discovery in Large-scale Multi-agent and Mobile-agent Systems*. Software Engineering and Knowledge Engineering, International Journal of, 2001.

[JXTA a] *Project JXTA v2.3: Java Programmer's Guide*, May 2003, `http://www.jxta.org/docs/JxtaProgGuide_v2.3.pdf` .

[JXTA b] *JXTA - Java-based peer-to-peer platform*, http://www.jxta.org/ .

[JXTA c] *JXTA Benchmark Project*, http://bench.jxta.org/ .

[JXTA d] *JXTA Monitoring Tool Project*, http://jxta-monitor.jxta.org/ .

[Konn 04] KONNO, SATOSHI: *CyberLink: Development Package for UPnP devices*, http://www.cybergarage.org/net/upnp/java/ .

[Lin et al. 05] LIN, SHIDING: *WiDS: an Integrated Toolkit for Distributed System Development*. June 2005, http://research.microsoft.com/research/pubs/view.aspx?pubid=1398 .

[LoadRunner] *Mercurry LoadRunner - performance and load testing tool*, http://www.mercury.com/us/products/performance-center/loadrunner/ .

[Lucio 03] G.F.LUCIO, M.PAREDES-FARRERA, E.JAMMELN, M.FEURY, M.J.REED: *OPNET Modeler and NS-2: Comparing the Accuracy of Network Simulators for Packet-Level Analysis using a Network Testbed*. 2003.

[Maie 00] MAIER, STEFFEN: *Emulationskonzepte für Netze mit gemeinsamem Medium*. Diplomarbeit.

[Mari 98] GOYENECHE, JUAN-MARIANO DE: *Multicast-HOWTO*, http://www.linuxrx.com/HOWTO/sunsite-sources/Multicast-HOWTO.html .

[Milojicic et al. 02] *Peer-to-Peer Computing*, March 2002, http://www.hpl.hp.com/techreports/2002/HPL-2002-57.html .

[NISTNet] *NIST Net Home Page*, http://www-x.antd.nist.gov/nistnet/ .

[NS2] *NS2: The network simulator*, http://www.isi.edo/nsnam/ns .

[NS2Emu] *Network Emulation with the NS Simulator*, http://www.isi.edu/nsnam/ns/ns-emulation.html .

[NS2Nam] *Nam: Network Animator*, http://www.isi.edu/nsnam/nam/ .

[ObDu 03] JOHANN OBERLEITNER, SCHAHRAM DUSTDAR: *Constructing Web Services out of generic Component Compositions*, 2003, citeseer.ist.psu.edu/708795.html .

[OPNET] *OPNET Modeler*, http://www.opnet.com/products/modeler/home.html . OPNET: Making Networks and Applications Perform.

[PaFl 01] "S. FLOYD, V. PAXSON": *Difficulties in simulating the Internet*, http://www.aciri.org/floyd/papers.html . IEEE/ACM Transactions on Networking, Vol.9, No.4, pp.392-403, August, 2001.

[PaFl 97] *Why We Don't Know How To Simulate the Internet*, 1997.

[Pidd 97] PIDD, MICHAEL: *Computer Simulation in Management Science*. John Wiley and Sons Ltd, ISBN: 0471979317, 1997, http://www.cse.ohio-state.edu/ jain/books/perf-book.htm .

[Prefuse] *Prefuse - toolkit for building visualizations of data*, http://prefuse.sourceforge.net/ .

[RaGr 02] RAKOTONIRAINY, ANDRY and GREG GROVES: *Resource Discovery for Pervasive Environments*. In *On the Move to Meaningful Internet Systems, 2002 - DOA/-CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 866–883, London, UK, 2002. Springer-Verlag.

[RFC 1058] *RFC 1058 - Routing Information Protocol*, http://www.faqs.org/rfcs/rfc1058.html .

[RFC 2608] *RFC 2608 - Service Location Protocol, Version 2*, http://www.faqs.org/rfcs/rfc2608.html .

[RFC 2614] *RFC 2614 - An API for Service Location*, http://www.faqs.org/rfcs/rfc2614.html .

[RFC 791] *RFC 791 - Internet Protocol*, http://www.faqs.org/rfcs/rfc791.html .

[RiAm 03] G. RILEY AND M. AMMAR: *A generic framework for parallelization of network simulations.*, 2002. In MASCOTS, Mar. 1999.

[Rizz 97] RIZZO, LUIGI: *Dummynet: a simple approach to the evaluation of network protocols*. ACM Computer Communication Review, 27(1):31–41, 1997, citeseer.ist.psu.edu/rizzo97dummynet.html .

[Robi 03] ROBINSON, STEWART: *Simulation: The Practice of Model Development and Use*. Wiley- Interscience, New York, NY, ISBN 0-470-84772-7, 2003, http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470847727.html .

[Russ 00] RUSSELL BRADFORD, ROB SIMMONDS, BRIAN UNGER: *A Parallel Discrete Event IP Network Emulator*. In *MASCOTS*, pages 315–, 2000, citeseer.ist.psu.edu/bradford00parallel.html .

[Salutation] *The Salutation Consortium, Inc.*, http://www.salutation.org/ .

[Sato 03] SATOH, ICHIRO: *Software Testing for Ubiquitous Computing Devices*, 2003, citeseer.ist.psu.edu/576758.html . National Institute of Informatics.

[Scharioth et al. 04] DR. JOACHIM SCHARIOTH, DR. MARGIT HUBER, KATINKA SCHULZ, MARTINA PALLAS: *Horizons2020: Ein Szenario als Denkanstoss für die Zukunft*. Eine Untersuchungsbericht der TNS Infratest Wirtschaftsforschung, München.

[Schi 04] SCHIELE, G. AND BECKER, C. AND ROTHERMEL, K.: *Energy-Efficient Cluster-based Service Discovery for Ubiquitous Computing*. .

[SiBrUn 00] *Applying parallel discrete event simulation to network emulation*. IEEE Computer Society, 2000. Washington, DC, USA.

[SiUn] *Towards Scalable Network Emulation*, 2001.

[SLP] *An Introduction to the Service Location Protocol (SLP)*, http://www.openslp.org/doc/html/IntroductionToSLP/ .

[SWANS] BARR, RIMON: *SWANS - Scalable Wireless Ad hoc Network Simulator (User Guide)*, 2004, http://jist.ece.cornell.edu/docs.html .

[Timm 01] TIMMERMANS, HARRY: *Models of activity scheduling behaviour*. Stadt Region Lang - Heft 71.

[UPnP Forum] *The UPnP Forum*, http://www.upnp.org .

[ViBa 02] VIKRAM VIJAYRAGHAVAN, JOHN J. BARTON: *WISE – A Simulator Toolkit for Ubiquitous Computing Scenarios*, 2002, http://citeseer.ist.psu.edu/473651.html .

[VMWare] *VMware Workstation: Powerful Virtual Machine Software*, http://www.vmware.com/products/desktop/ws_features.html .

[W3C] *World Wide Web Consortium (W3C)*, http://www.w3.org/ .

[Wang 04] WANG, JUE: *A Thesis: Performance Evaluation of a Resource Discovery Service*, 2004. A Master Thesis.

[Weis 93] WEISER, MARK: *Some computer science issues in ubiquitous computing*. Commun. ACM, 36(7):75–84, 1993.

[Wikipedia] *Wikipedia - the free encyclopedia*, http://en.wikipedia.org .

[Zhen 02] *Expriences in Building a Scalable Distributed Network Emulation Systemn*, September 2002.

[ZhMuNi 02] FENG ZHU AND MATT MUTKA AND LIONEL NI: *Classification of Service Discovery in Pervasive Computing Environments*, 2002, http://www.cse.msu.edu/ zhufeng/ServiceDiscoverySurvey.pdf . Michigan State University, EastLansing.

[ZhNi 02] P. ZHENG, L. M. NI: *EMPOWER: A scalable framework for network emulation*. In Proceedings of the 2002., 2002.

[ZhNi 03] P. ZHENG, L. M. NI: *EMPOWER: A Network Emulator for Wireline and Wireless Networks*. In Proceedings of the 2003., 2003.