MASTER

Evaluation of UPnP in the context of service oriented architectures

Beckers, Remco J.J.

Award date:
2006

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER'S THESIS

Evaluation of UPnP in the context of Service Oriented Architectures

by

R.J.J. Beckers

Graduation Committee:	
Supervisor:	Dr. J.J. Lukkien, TU/e
	Dr. ir. I. Radovanovic, TU/e
	Dr. J.M.T. Romijn, TU/e

Eindhoven, december 2005

Abstract

Service Oriented Architectures (SOAs) are not a very new concept. They are used in many business applications. However SOAs are usually associated with web-services supporting a business environment. In this paper we use services to define a Distributed Storage Network in which services have to interact directly with service-users. Moreover a service-user can be a normal user, but a service user can also be a service which uses or monitors (the state of) the functionality of the first service.

The implementations of these services in our case use Universal Plug and Play (UPnP) as the protocol for discovery, advertisement and communication. UPnP is a relatively new standard which is only recently being used in products, mainly to configure routers and other network devices. Since UPnP is relatively new only little research is done on it, which introduces the situation that a technology is being used in an increasing number of devices without a decent knowledge of its properties and characteristics.

We implement a Distributed Storage Network using UPnP and based on this implementation we provide design practices for developing SOAs using UPnP. Moreover we evaluate UPnP on our implementation and usage experiences and we perform a detailed performance analysis to support the design practices and determine the suitability of UPnP as a basis for SOAs interacting directly with users.

Acknowledgment

I would like to thank, among others and in no particular order: The people at SAN, especially my room mates, which provided me a pleasant working environment and with some interesting discussions on services and components. Further I want to thank Johan Lukkien for his supervision, guidance and support and Igor Radovanović and Judi Romijn for taking place in my examination committee.

Contents

Chapter 1

Introduction

1.1 Goal

The goal of this project is to gain insight in the implementation technology necessary for Service Oriented Architectures (SOA). This means we design an application completely based on services which, combined in a specific way and using each others functionality, will form the application. These services can be implemented using components which will use a universal way (UPnP in this thesis) to expose the services to the network. These services can be combined to form new services or to form a (new) application, on the other hand a user can use the service directly. When combining services to form new services a certain cooperation structure is created, deliberately or not. This structure will be discussed more elaborate in section 3.2.2.

This form of creating applications based on services which in turn can be based on services again will be evaluated on several points during this project:

methodology Is there a single best strategy for combining components to create an application and what is it? Moreover, is it possible and is the SOA powerful enough to use existing services and combine them to create an application or service which has better characteristics than the services composing it? For example, is it possible to create a reliable application from unreliable components or a secure application from insecure components.

claims vs. facts During this project we make certain claims about SOA's, but can we find facts to support those claims?

performance & usability Is it possible to combine the components in such a way that a user doesn't really notice he is using a SOA based application? Depending on how fine grained services will be made and in which way services are combined, performance problems can arise. For example because of excessive network communication introduced which can flood the network. Considering usability should help us determine the extra work for a user (or administrator) introduced in using a SOA instead of a normal architecture.

All our service implementations will be based on the UPnP standard, which will be discussed in more detail in chapter 4. This means we will not only be evaluating SOA's on the aforementioned points, but more specific we will be evaluating UPnP on its performance, usability (for a developer) and how well it is suited to implement Service Oriented Architectures.

1.2 Structure of this thesis

In the second chapter we will discuss the specifications to which we want to adhere when designing a service oriented architecture as a system of cooperating services. In chapter 3 a framework is introduced which can be used as a basis for this system. Here we'll also discuss which parts of the specification(s) will be addressed in the framework and which parts will be made explicit in the further work.

As an example of a SOA we will be building a Distributed Storage Network (DSN) during the project, we will describe the design in chapter 4. This DSN will be used for examples and for testing concepts and claims made in this document. The DSN's task will be to offer data storage space and to offer services to users to store and retrieve data from it. The DSN will be setup as a SOA and will therefor abide to all the requirements of a SOA.

Using the design and implementation of the DSN the next chapter will discuss performance tests done. These performance tests will then be used in the next chapters to suggest improvements for the UPnP implementation and design rules for designing new services and communication behaviors. Finally we will show how to add extra functionality to an existing distributed system by designing a new service which implements and uses interfaces already discussed in the specification and framework chapters.

Chapter 2

Definitions

Service Oriented Architecture A Service Oriented Architecture (SOA) is a collection of communicating services. These services use message passing as their means of communication. The communication can be simply between two services, between a service-user and a service or it can be multiple services coordinating an activity (for example a third service orchestrates the communication between two other services).

Service A service is a contractually specified functionality, for which an interface, an access point, functionality and quality of service are specified.

Component A component is the implementation of specific functionality in a reusable unit of deployment.

UPnP Service A UPnP service is used to implement a service interface using a component to implement the functionality. The interface of a UPnP service is specified in a XML description using actions, responses and events.

UPnP Device A container for one or more UPnP services which makes those UPnP services accessible over a network. A device (and its services) is dynamically advertised and discovered and has a location.

UPnP Control Point Searches for UPnP devices and UPnP services and makes the service interface (consisting of actions, responses and events) accessible as functions in the software (application, component etc.) interested in using services.

UPnP Statevariable A UPnP service has statevariables. All these variables together represent the state of the component implementing a service and make this state visible to control points. Statevariables are declared in the XML specification of a service.

UPnP Action Actions form, together with events, the interface of a UPnP Service. Actions and there parameters are declared in the XML description of a service. Every action implements a part of the functionality of a service and returns a response in the form of output parameters and/or changes the state of a service.

UPnP Event A control point can subscribe to events of a service. Every statevariable of a service has an associated event. When a statevariable changes all control points subscribed to the service receive an event which contains the name-value pare of the changed statevariable.

Chapter 3

Services

A Service Oriented Architecture (SOA) is a collection of communicating services. These services use message passing as their means of communication. The communication can be simply between two services or it can be multiple services coordinating an activity (for example a third service orchestrates the communication between two other services). So the only way to define what a SOA is to define what a service is. That is exactly what we will do in this chapter. First we will determine what a service is and discuss the differences between services and components. After that we will continue with a discussion of additional properties of a Service Oriented Architecture, because they are desirable and often necessary or useful because of the use of services.

3.1 Services and components

The main differences between components and services is that a component is a deployment unit and a service is specification unit. Advantages and disadvantages of using components and services are discussed in appendix C.

The properties of a component are:

Interface A component has a well-defined interface (for provided and required functionality) and depends only on its interface and the platform it runs on. This interface is often defined using a contract, description or Interface Definition Language (IDL).

Encapsulation A component encapsulates or hides its inner workings from the outside and thus effectively abstracts away from implementation details.

Reusable A component is reusable in the sense that it is already designed with multiple uses in mind and as flexible as possible.

Composable A component can be used together with other components to cooperate and form new components again.

Deployable A component is a unit of deployment, meaning that it can be deployed and used independently.

The implementation of a component can be done using one or more objects. However a component does not have to be implemented as (an) object(s). It can just as well be implemented as a library, module or application. As long as it can be used as a unit of deployment. It is even very easy to turn an object into a component. To make a component out of an object we have to make the object independently deployable and we have to specify both its interfaces (provided and required, the latter is not always well documented in objects whilst the first are the public methods of an object).

Now we define what a service is. A service is a contractually specified overall functionality and a service has:

Service Access Points Access Points are where the functionality of a service is provided. A service can have two types of access points, one for provided functionality and one for required functionality.

Service Interface The functionality of a service is made available to a service user through the service interface in the form of actions, responses and events (i.e. autonomous responses)

Specification A service has a specification describing its functional properties, i.e. the effect of an action on the output parameters and the state variables of a service. The specification also describes the behavioral properties of a service, i.e. how and when actions should be called and how responses are generated.

Quality of Service Additional to the specification there is a specification of the quality of the service, for example in terms of performance or reliability.

The main differences between a service and the functionality of a component are:

- A service is a higher abstraction level than a component.
- The quality is separated from the functionality and thus can be handled separately.
- A service is (just like a component, but unlike an object) rather isolated functionality which can be defined separate from its use and users, a service is unaware of the way it is used.
- In Service Oriented Architectures a service is independent of platform.

The functionality of a service can be implemented using one or more objects or components. However this requires a way of mapping the component interface to the service-interface as they are not necessarily the same or don't even work in the same way (function calls for the component vs. message passing for the service for example).

Since we intend to use services in a Service Oriented Architecture build using a network for communication we have to add a few properties to the services:

Accessible over a network Making a service accessible over a network makes it operating system and language independent.

Dynamic advertising To be able for service users to use a service on the network services have to advertise their functionality.

Dynamic discovery To enable service users to use a service they have to be able to dynamically discover services on the network.

Figure 3.1 shows how a service user can find and use service functionality. The service-user in this figure can actually be several different ”users”. Of course the service-user can be a normal user who is using the functionality of the service. However it can also be another service, which uses the functionality of the service in order to be able to provide its own functionality or to monitor the correct functioning of the service. The first introduces relations between the services and is essentially responsible for the layering we discuss in the next section.

In our implementation we will be using components to implement the functionality of a service. The components will be using Universal Plug and Play (UPnP [7]) to make the service accessible over a network and implement the service interface, the dynamic advertising and the dynamic discovery properties. The service has now as a main access point the network address the UPnP device is running on combined with the port number x where x is the port number on which the UPnP device is accepting action requests. In chapter 4 we will discuss the working of UPnP in more detail.

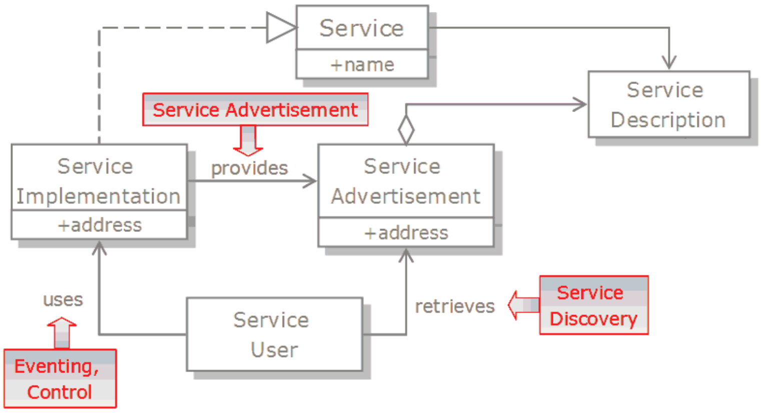

Figure 3.1: The conceptual modal of a service. Image used from [16].

3.2 Extra-functional properties

A service can have many more properties which have to be specified and which we can gather under extra-functional properties. Many of these properties are important or useful in Service Oriented Architectures. For example the possibility to compose services is necessary for service oriented architectures in which we try to compose several services in such a way together that they form a complete and functional application.

The properties we will discuss and will investigate in this thesis are distribution, composability and performance. Other properties which will be discussed, however in much less detail, are reliability and consistency. These last two are also closely related in a distributed environment we want to create in Service Oriented Architectures. There are even more properties which can be of importance, like security for example, which we will not discuss in this chapter nor in the rest of the thesis.

The properties are separated of the functionality specification because they are mostly part of the quality specification. We can create separate services which have their own interface and provide functionality to control or handle for example performance or reliability of a service. However both services can be implemented by the same component.

3.2.1 Composable

Services can be composed, meaning that one service can use other services to provide its functionality. A service which uses other services (thus forming a composition of services) to provide its functionality has a required service access point which connects to other services provided service access points. What is also needed is that the service interfaces of the required and provided functionality have to match. Moreover is it important that the required services match their specification, because if the required services don't match their specification, the provided services probably will not match their specification either.

Since we are implementing services as components and UPnP is used to implement the service interfaces we can make the aforementioned composition requisites more explicit. Connecting required and provided service access points is a task delegated to the UPnP protocol. This uses dynamic advertisement and discovery to find services with the required functionality and allows the service access points to connect.

Matching interfaces in terms of services means in our implementation that the component interfaces have to match. Using UPnP this means that a provided service description (which actually lists actions, their responses and events) has to match with the action calls and event subscriptions of the service user. Unfortunately UPnP doesn't offer an easy way of checking matching interfaces, because it does not have a specification for the required service interface.

Finally we need to make sure that components implement the functionality of a service to the service specification. There is no way of guaranteeing this using UPnP, we do not have a specification associated with every service which can be checked before using the service (provided by a component). The only guarantee we can get is that a component developer has followed the service specification of a specific type and version of a service (which is provided during the advertisement and discovery of a UPnP service). When a service user needs to use a service he can check the type and version of the service implementation and just has to hope the component adheres to the specification.

3.2.2 Distributed

Since we are using components connected to a network as an implementation of services we can deploy a service anywhere on a network as long as a network node is running the correct operating system or virtual machine. Services can still be composed because they can be accessed over a network. This makes it very easy to distribute services over different network nodes. Compared to using normal components running on one device this offers greater flexibility in composition and the different dependencies of a component. A component can have four relations:

1. it depends on the platform it is running on
2. it can expose its services for end-usage to end-users or other systems
3. it can expose its services on the network where its functionality can be used by other components
4. it can use services over the network to fulfill its own task (for end-usage or other components).

Of these relations only the first holds for every component, most current components have or use the first and second relation. Current service oriented applications are usually designed with a specific purpose in mind and don't directly serve a user or don't use services on the network.

This is in contrast to our intentions with Service Oriented Architectures. We can design non-dedicated services which we can implement and combine using the relations mentioned to build components with greater functionality than its parts. In a SOA we try to make better use of the possible relations. A component implementing a service in a SOA offers services, uses services, uses the platform it runs on and some components (also) offer functionality to end-users.

We can determine an ordering for services using their relations. A service which uses functionality of another service with order x gets an order which is at least $x + 1$. Like this we can assign numbers to every service depending on how they fit in the composition of services. If we place all services with the same order in one layer we get a very clear layered structure. A layer can consist of only one service, however there will probably be many instantiations of this service during run-time. When we implement services as components we get the picture in figure 3.2.

This picture suggests however that communication within a layer is not possible. This does not have to be true, for replication for example it could be even necessary that components within the same layer communicate to synchronize states. Neither is it necessary that every layer communicates with end-users, limiting the number of layers communicating with end-users can also help simplifying the design. For example to be able to offer a simple user interface at one location it may be useful to limit the user interaction to the layer with the highest order.

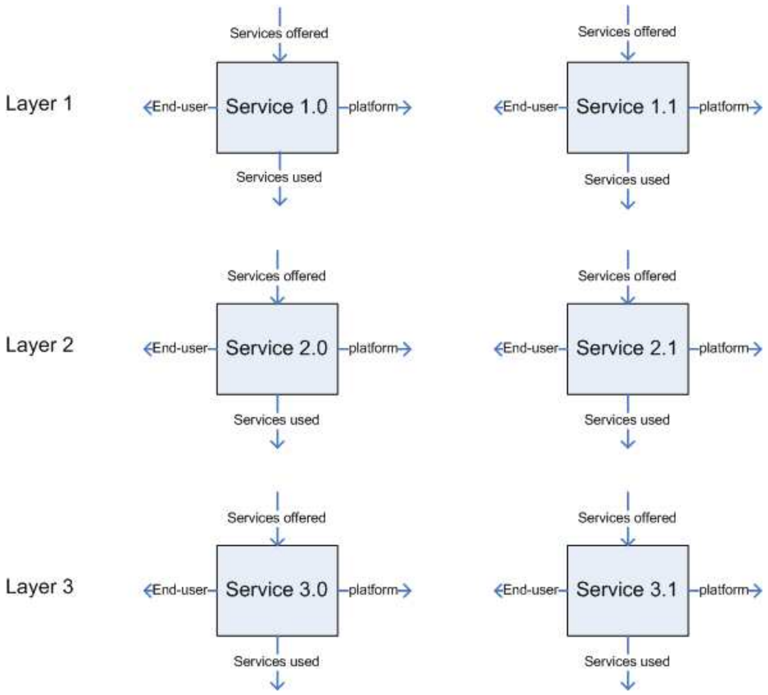

Figure 3.2: The ordering of services implement as components. Layer 1 has two instantiations of service 1, layer 2 has two instantiations of layer 2 etc..

This layered structure on its own isn't new, in lots of applications layering is used for abstraction from the task at hand to more simple components. What we do now is we specify the different components as services with explicit specifications and interfaces intended to function over a network. The services have to be dynamically composable so the functionality of a service can be enhanced even after the implementation (as a component) by simply replacing one of its required services with a more advanced similar service. The functionality of an entire application can be enhanced by developing new services and adding them to the network where they can immediately fit in the already existing composition of services.

Designing this layered structure on the services beforehand can be useful to shape the application, separate functionality, determine bottlenecks in a design and determine load balancing or replication strategies. For example we can see in a layering that in one layer we have only one instantiation of a service which can be a major bottleneck for all other services. Recognizing the layering after an application is created can be useful to comprehend and understand it better and again visualize possible flaws or improvements. For example in the Distributed Storage Network (DSN) we are going to design in chapter 5, we can determine several tasks beforehand:

- Data has to be stored somewhere somehow. Data can be stored in many ways (persistent or volatile), but we actually do not care how it is stored. We just create a service, the data storage service, which stores a predefined maximum amount of data and offers actions to retrieve and store this data.
- A service is needed which provides access to a directory structure listing all files with their locations and providing access to meta data on those files and directories. This is needed to be able to retrieve files once they are stored. We will call this service the DirectoryService.
- Finally we need a service an end-user can understand. We call this the file service. The function of this service is to offer the user an interface for storing and retrieving files, given a filename and file path (in the directory tree). This service takes care of creating and finding files within the directory service and storing the data of the files in an available data storage service.

In this example the first layer consists of file services, this is the layer a user interfaces with. The data storage services form the second layer, they are only used by the file services and should not be accessed by an end-user. The directory service is a bit more difficult to match into a layer. It is used by the file service, however it can also be accessed by an end-user to retrieve a directory listing (maybe this functionality should also be provided by the file service to avoid this ambiguity). So actually it is somewhere between the first and second layer and could join the data storage services in the second layer if some functionality would be added to the file service.

Implementing these services would give us three components we can deploy anywhere we want to depending on the implementation we choose. For example implementing the components in java would allows us to run them on every platform a java virtual machine is available for and has a network connection to connect the components. Only the end-user service needs to fulfill the end-user dependency and therefore needs a way of communicating with the end-user, by means of an end-user terminal for example.

Introducing layering in the design of an application using services as the very simple units of construction shows great flexibility. We could even have continued adding layers after the data storage services, for example to store single blocks of data with a constant size. However oversimplifying can cause unnecessary communication and delays in performing a task or solving a problem. So in the DSN we stopped at the data storage service level. It can be compared in developing a normal application from functions and statements. Up to a certain point new functions are defined for a group of statements, but at some point creating a new function is not very useful anymore and only generates extra overhead.

3.2.3 Performance

Required performance of a service can be specified when designing a service. This can be in terms of expected average response time or average through-put. However it can also be given by boundaries between which the different performance parameters are expected to be.

We implement services using components which can be deployed on different devices with different system parameters resulting in different performance. Demanding a specific throughput in a performance specification of a service may prove to be impossible to reach on some devices the component implementing the service will be deployed on. So actually performance specifications should always incorporate the fact that different hardware gives different performance, this can be done by for example giving minimal system parameters a service should be implemented and run on.

To make matters worse even if in a normal situation the performance of a service implementation on specific hardware is as specified, the performance may degrade significantly when other software is running on the same hardware at the same time. The performance of the service may drop below the specification and therefore can be of little or even no use anymore. We prefer the situation in which we can predict the performance of a service to be predictable between specific bounds to the situation in which we sometimes gain a faster performance but we can not guarantee a minimal performance.

The protocol used for discovery and for communicating between the services (the interfaces) is also of importance for the performance of the service implementations and can be of big influence in meeting the performance specification. We are using UPnP as the main protocol and in discussing its properties in chapters 4 and 5 we will also refer to the performance impact every part of the protocol can have.

Other factors which can be of great influence on the performance can be the network used to communicate with other services and the usage (i.e. load) of the network. If for example their is a lot of data traffic on the network it may slow down the communications of UPnP to the speed at which performance specifications cannot be met any more.

In chapter 6 we will measure the performance op UPnP under different situations because we not only want to determine the normal performance of the protocol, but we also want to know the impact other factors, like high network load or other software running on the same hardware, can have on its performance.

3.2.4 Reliability & Consistency

In a normal architecture an application would probably fail if one component of the application would fail. In a SOA we want it to be possible for only one or several components to fail without the entire application failing (partial failure). When identifying the different failures a component can suffer from we find (from [21] chapter 7):

- Crash failure: a component halts, but is working correctly up until then
- Omission failure
  - Receive omission: a component fails to receive incoming messages
  - Send omission: a component fails to send messages
- Timing failure: A component's response is to early or to late according to a specified time interval
- Response failure
  - Value failure: The value of the response of a component is incorrect
  - State transition failure: The component does not follow the specified program flow
- Arbitrary failure (or Byzantine failure): A component may produce arbitrary responses to request at arbitrary times

Most, if not all, of these failures can be handled by having multiple components running which implement the same or similar services. If one service implementation fails, a service-user can easily switch to another one. Moreover if the responses of a service implementation are not to be trusted the responses can be requested from multiple implementations to compare them.

Detecting the state of a component from the outside is difficult. This functionality could be implemented as a separate reliability service which runs together with every component. It should monitor if the components implementing a service still operate according to specification and send a report out on a regular basis so an external running service or service-user can determine that the components are still functioning correctly.

Solving reliability issues using replication does not only solve the reliability problem, it also introduces a consistency problem. If we have multiple components which implement the same service, but also maintain an internal state, they somehow have to cooperate in maintaining the same state. This is something which can be easy in a service oriented architecture as it is quite easy to extend an application with additional services. It would for example be possible to implement a service which takes care of the cooperation between the different components implementing the same service.

If this would not be done, failure of one component would result in loss of the state of that component. Moreover component A and component B can produce different action responses (the response can depend on their state) to service-users. This is unexpected and unwanted behavior since from a service-user perspective the different implementations of a service are the same and it should not matter which is used.

An example of this can be the directory service in the DSN, which maintains the location of files (or parts of files). It maintains a list of all files (and directories) with the location of the data of those files. When introducing two components implementing the directory services on a network they won't be maintaining the same list of files automatically and thus not representing the same state. Some work has to be done so both directory components represent the same state and a client can make a choice between them instead of using both of them hoping together they represent the complete directory/file listing of the DSN. Failure of one of the components isn't such a big issue anymore either then because another component maintains the same state.

When maintaining the same state among a number of services there are several models of consistency possible. However from every point of view the state of all the services should be the same at any time. Sequential consistency is just what we need for this. However because sequential consistency is rather strict with respect to the timing of writes - the state should change in all services simultaneously - we relax this sequential demand a bit:

> A state change in a group of services should have spread from its origin to all other services in that group within a certain time $\Delta, \Delta > 0$ and all state changes are executed in some sequential order. Only the state changes from the same origin have to be in the original order.

In figure 3.3 we can see what exactly sequentially consistent means. Here R(x)1 means a read from variable x returns value 1 and W(x)1 means write value 1 to variable x. S1 and S2 are two separate services which should have a consistent state.

Relaxing the consistency demand from instantly to a time frame Δ does have some disadvantages. For one, during constant change the state of all services of a group is never entirely consistent. A consistent state is only reached after stopping all changes to the state and waiting time Δ.

In the mean time, when the state of the services is not completely synchronized, the services have to make sure these small inconsistencies don't introduce errors which a user could notice. Preferably the services take care that during time Δ after a state change both versions (the old state and the new state) can be used.

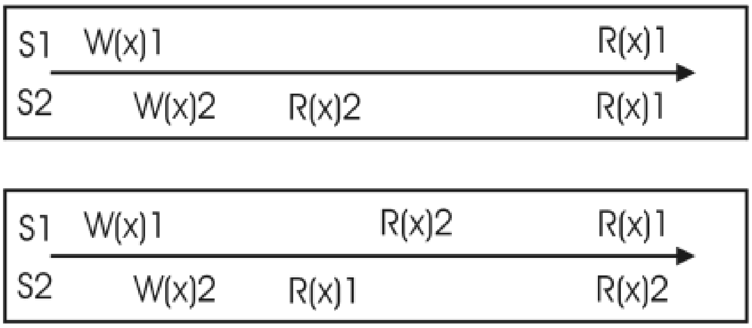

Figure 3.3: The top interleaving of reads/writes is sequential consistent, the bottom one is not

For maintaining this consistency demand an interface will have to be specified. Services which have to comply with this demand will have to implement this interface. Plugging in an extra component managing consistency between services can utilize this interface and guarantee the demand is met for a certain group of services. Even better would be if no extra interface would be needed and the service guaranteeing consistency only uses the already existing interface.

Chapter 4

Framework for Distributed UPnP based Systems

In this chapter we propose a systematic way of implementing services as components and using UPnP as the protocol to provide the service access points and the service interface. UPnP gives functionality to discover service access points and implements the necessary message passing interface for services. Also the framework given in this chapter can, combined with the results of the performance tests, be of great help in making Service Oriented Architectures perform best.

We based our framework on the Java programming environment and the Cyberlink UPnP API. We used Java because than the software could run on different operating systems (our test environment runs on Linux while the development computer runs on Windows for example) and we were already a little bit familiar with this programming language. However choosing for Java limited our choice in API to only two free usable API's: the Siemens AG UPnP API and the Cyberlink open source API. We tried them both to some extent, however the Cyberlink API is more flexible when testing, we would be able to look in to and modify the source code if necessary.

4.1 UPnP

UPnP is a fairly simple protocol which is intended as an easy-to-use device architecture, taking care of joining a (TCP/IP) network, finding service access points on that network and offering a service specification and an interface in the form of actions and events. Its main advantage over other service oriented architectures is that UPnP uses a form of peer-to-peer networking. Connections can be made and destroyed ad-hoc whenever a service joins or leaves the network. UPnP uses a UDP and multicast based discovery protocol to support this ad-hoc nature and make discovery of services possible without the need of a central server.

When using the UPnP architecture we need an API. There are several API's available, however only one is freely (usable) available (the Cyberlink API, which uses a BSD style license). Others have multiple restrictions and often they have to be paid for (see [23] for a list of available API's).

UPnP dictates a specific structure. The main "container" is a device, every device can contain several services which will be offered through the network interface a device is connected to. This device can be used or in UPnP terms controlled by a control point. So the control point is actually the service-user part of the UPnP standard while the services in a device are of course the service. This rather strange construction of devices containing services originates from the fact that UPnP was designed with the main focus on controlling (parts of) hardware devices whose functionality would be divided in multiple services. First thing to do when using UPnP is to identify which side of the communication becomes a service and which one should be the control point (service-user).

As an example we will show the two possibilities in defining the communication between a remote control and a television set. With UPnP we could make the television the device offering services and the remote control the control point controlling this device by using actions of the television like "switch channel" or "volume up", the television set could then sent events like "channel switched to 2" or "volume set to 10".

The other possibility is to make the remote control the device offering services and the television set the control point. At first this sounds strange, how could the remote *control* not be the *control* point, i.e. the service-user? But it just requires a slightly different way of communicating. Now when a user presses a button on the remote control this can be seen as an event generated on the device (the remote control). Now the television set subscribes to the events of the remote control so the television can adapt its state depending on the events received from the remote control. Events now take the form "button 2 pressed" or "volume up key pressed". These type of events surprisingly reflect a "normal" remote control more accurately than the previously mentioned implementation.

First he has to make the service interface and specification explicit in a XML description file using a syntax specified by the UPnP standard. Assuming there is already a component which implements the complete functionality the developer then has to map the component interface using the UPnP API function calls to implement the mapping between the component and the service interface.

Now there is a service which can be accessed and used, however for a service-user to be able to use the service the service-user needs a control point. The UPnP control point provides the functionality to find the service access points and use the services functionality through the service interface.

We can see that the UPnP standard is actually the glue between the interface of a component implementing the service functionality and the service interface. Of course there is more to it than described in this section, but we covered the general idea of UPnP. For more and more accurate information about the UPnP protocol we refer to the specification [7].

4.2 Distribute using UPnP

As already discussed in section 3.2.2 we compose a distributed system or application out of services. Every service offers some functionality over a network to a user or to other services. So every service has 4 dependencies (see section 3.2.2.

However UPnP is divided into two parts, a service part which offers functionality and a control point part which uses functionality. So to create services which can use functionality from other services we have to combine them with a control point. This is something UPnP was not originally intended to do, so there is also no ready to go solution for this. Luckily we could easily combine a service and a control point in one application so the service can use that control point to use the functionality of other services. However this introduces some problems and questions:

- There is a lot, too much, freedom for implementers of API's in the UPnP specification. Something left completely open is the concurrency of the different parts of the specification. For example the Cyberlink API deadlocks when a service uses a control point to call an action of another service in the same device (or an action of itself). This happens because the handling of action requests happens completely in one thread. Maybe it works in other API's, but it is still very unclear for a developer which behavior he should expect and he can only find this out by experimenting.

- A service sometimes needs functionality it provides self. Using this functionality via the control point would allow for greater flexibility and uniformity, however it would also slow down the functionality significantly compared to just calling a function internally. For example the DirectoryService of the DSN (chapter 5) can return a complete directory tree. This can be implemented by a function which returns the listing of one directory which is called recursively. However it makes a big difference if this function is called directly in java or if it is done through the service interface (which would even deadlock with the current implementation).
- UPnP Actions should not take long to complete according to the specification (page 48 of [7]) and of course also because they should not keep a service occupied to long. However there could be actions which take a long time to complete and UPnP does not offer a call-back mechanism besides eventing which has its own problems (see next bullet).
- Events are very limited in their possibilities. A control point can only receive all events or no events of a service. This makes events almost useless for even a poor man's call-back mechanism although this is suggested in [7] on page 48. For example take a service which has for every action an event which is used to send the results of that action to a control point. Every control point would get the results of every other control point, so the events should have some sort of identification. Moreover it is possible the result for one control point is overwritten by the result for another control point before it gets send. So this system is unreliable, some control points could get no result at all and would have to reissue their request and hope they would get a result within a limited number of attempts. This was already discussed in [16].

The question raised in the second point can be answered easily with the help of the first point. As long as the UPnP specification is unclear about the notion of concurrency we can not make any assumptions about concurrency and a service should never try to use its own functionality via its control point. It not only slows function calls down, it can even lead to deadlock. Although it is possible to make sure a service does not use itself for this functionality it would only add complexity to the code to determine this and would still lead to a slow down.

Solutions for the other problems mentioned will be given in the next section.

4.3 Framework

4.3.1 Abstraction

First of all we want to be able to use the UPnP functionality as simple as possible. However when writing more than one service or device it requires us to write almost exactly the same code for every service and/or device. Therefore we create a generalized device and service. This does not only make it easier to use UPnP, it also makes it easier to switch API when considered necessary.

We call the device GenericDevice and the service GenericService. The device is actually very simple, it just reads in a device description XML file (see [7]) and determines which services it should provide. Then it checks if it can provide these services (i.e. if it has classes implementing the service type found in the XML file) and if so creates the appropriate instances and starts the services. The GenericDevice also maintains a list of services it provides associated with the GenericService children implementing these services.

Actions are handled by the device by propagating them to the service they were intended for (if that service is offered by the device of course). Instead of a very long list of control statements in the service or device source code (as can be found in some example source code provided with the API) the service maintains a list of the actions it provides linked with action processors, which contain the code that needs to be executed by an action and produce result (output) parameters. If the service receives an action via the device it searches for the appropriate action processor, if found the action processor produces a result and returns this. If there is no action processor listed for the specific action (although it might have been advertised) an error code is returned. This is a much easier and transparent way of handling the actions, instead of needing to extend an already complicated control statement we only need to create a new action processor and link it with the action a service should provide.

The action processor for every available action should be a child of the ActionProcessor class which provides useful functionality for retrieving the input parameters of an action and for setting the output parameters. Further it contains one abstract function which should be overridden by its children, the processAction function. This is called by the GenericService when an action of the type of the action processor is received. When a new service class is derived of the GenericService it can add its actions associated with an actionProcessor object to a list maintained by the GenericService class which it uses to find an action processor when an action is received.

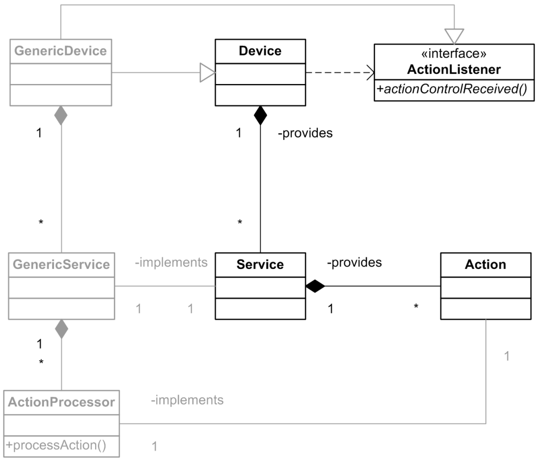

Figure 4.1: Simplified UML design of the UPnP API (black) and the abstraction classes added (gray) for devices and services.

For the control point we did something similar, there is a significant amount of code necessary to get a working control point even before it has any functionality. As can be seen in figure 4.2 a normal control point needs to implement at least two interfaces and also needs to extend the Controlpoint class. The GenericControlpoint takes care of all this and provides functionality for administering in a simple and accessible way the interesting services currently available on the network. Interesting services means in this context services the control point at some point in time might need to use, because there can be much more services of all kinds of types online which are of no interest to this control point. However they still report their presence every now and then over the network, so the control point wrapper hides them from the rest of the application.

What we actually did here was abstracting from the advertisement and discovery protocol used in UPnP. Using the GenericControlpoint we only indicate which service types we are interested in, the GenericControlpoint then listens to all advertisements and filters out the ones we are interested in. Moreover when the GenericControlpoint is started it does a discovery search for all services a control point is interested in. The GenericControlpoint actually tries to maintain and offer (to control points) the best possible view on the current available services on the network a control point is interested in.

A child of the GenericControlpoint can use functions to indicate which types of services it is interested (for using its functionality and which types of services it wants to subscribe to for events, usually the latter is a subset of the first) in. It also offers, to its children, functionality to retrieve a list of services (of one specific type or all) and to retrieve one service of one type (random or a specified number). A child of the GenericControlpoint can now implement public functions which are implemented by means of calls to actions of services, these functions should actually form the interface between the UPnP parts and the other parts of the application. Like this all UPnP communication is 'hidden' in one java class which can be used as a normal class without even knowing the result it produces are coming from UPnP services.

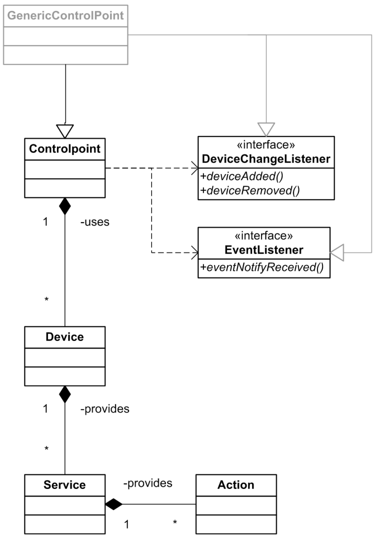

Figure 4.2: Simplified UML design of the UPnP API (black) and the abstraction classes added (gray) for control points.

4.3.2 Services and UPnP services

Now we can combine the services and control points to the components we already discussed in chapter 3. To avoid confusion from now on we will refer to the UPnP service as UPnP service and to the service from chapter 3 as service.

We can now divide applications into four types of 'implementation':

1. Components which have end-user, platform and services-used dependencies can be implemented by a control point and a user interface.
2. Components which have services-offered and platform dependencies can be implemented by a UPnP service.
3. Components which have services-used, services-offered and platform dependencies can be implemented by a combination of a control point and a UPnP service.
4. Components which have services-used, services-offered, platform and end-user dependencies can be implemented by a combination of a control point, a UPnP service and a user interface.

Of course we can think up more categories like components with end-user and platform dependencies, but this leads to a normal component just like components with services-offered and platform-dependencies leads to a traditional service component.

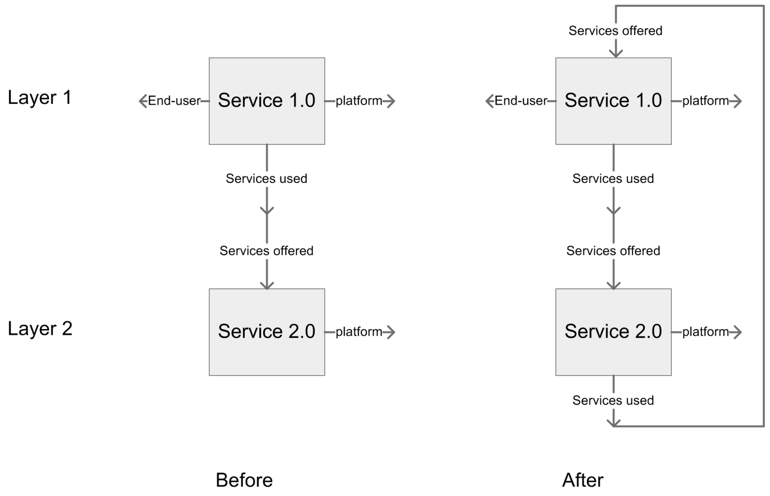

Figure 4.3: On the left the situation before a call-back was possible (first layer contains type 1 service, second layer contains type 2 service), on the right the situation afterwards.

Now we can solve the lack of a decent call-back function in UPnP, which is much more versatile than the eventing system of UPnP. We can use call-back functions to return results specifically to one 'subscriber' whom also issued the request, without spamming all other subscribers and without difficult encoding of the result value(s).

Suppose we have an application consisting of only a control point and another application consisting only of a UPnP service, they have dependencies like in the left hand side of figure 4.3. To make call-back functions work we have to add a UPnP service to the control point and add a control point to the UPnP service so we get the situation in the right hand of figure 4.3. Now the original action call has to be modified by adding an argument to specify the call-back-action the service application should use to return the results to the control point. Of course both, service and control point application, should agree on the syntax of the call-back-action. Just as they agreed on the syntax of the original action.

4.3.3 Design Practices

As with all development projects, also when using UPnP to develop an application it is necessary to have a specification of the functionality of a service and a (detailed) specification of the interface offering the functionality. With UPnP this is even more important than usual since the specification of the interface is made explicit in an XML file and necessary for the correct functioning of the entire application. The difference with designing a normal application is that when designing an application as a Service Oriented Architecture one should think in terms of tasks instead of components. Components only come in a later stage when the tasks are defined and are mapped to services which have to be implemented by components. A service has to fulfill a task, but as long as the service adheres to a specification it does not matter which component it internally uses.

Since components are now embedded in services we can use any suitable component to implement the service, we can even switch components without from the rest of the application noticing it and spread the separate services over different devices and even have multiple instances of one service, so if one services crashes your entire application can still function properly possibly with reduced functionality, however your application won't crash entirely. Downside of this flexibility is that a price is payed in terms of speed, a task executed on a remote service has a slower response time than when a task is executed locally using one or multiple components.

So there is a constant deliberation between slower response times and more network traffic on one hand and a better spread of load and distribution of risk on the other hand. Many tasks could be turned into services so the SOA gets more fine-grained, however at a certain point this isn't good practice anymore because the price paid in terms of response time for completing a relatively simple task outweighs the flexibility gained by introducing an extra service for it. Moreover adding a separate service for every task will also introduce a lot of extra communication and thus put extra load on the communication interfaces and network. With the experiments in chapter 6 we hope to gain more insight in how to make choices regarding these problems.

Sometimes however an action can be less frequent and very complex which could be an argument to split it up into more simple services which may require and/or allow communication between these simple services or even replication of data. This in turn can give problems in maintaining consistency between the state of different services or when an action has to be executed atomically (thus locking up many services). There are solutions for these problems which don't require locking up all services to make an action atomic and which can keep data consistent over multiple services, see for example [9], [5] or [14], however they still introduce unnecessary complexity to the services and add extra communications between services.

Some of these problems arise when implementing the DataStorageService. To store the data in a safe way we would like every file part to be stored on multiple DataStorageServices. On the one hand this requires additional communication (for example between the DataStorageServices), however because the DataStorageService fulfills such a simple function (compared to for example the FileService) it has become easier to implement replication in a way that the rest of the applications is unaware of it.

Using layers can be of help when structuring a UPnP based distributed system. Allowing a service to only communicate with the layers above and below it self (like for example in a network stack every service communicates only with its closest neighbors) avoids complicated entanglement of services. The result is that it is easier to see the separation between the provided and required services and to see the service hierarchy from complex functionality to simple functionality.

Also a problem arises when considering the coarse or fine grainedness of a service interface. Suppose we have a ”media storage device” which stores for example recorded movies and meta-information about those movies (like title, lead actors, other actors, summary, poster etc..). Sometimes we only want to retrieve the title (in a listing of movies for example), some other time we want to see all information except for the other actors. Some other time we only want to see the poster.

One solution would be to create a separate function for every part of information even up to each individual actor. However when we need (almost) all information we need to do many function calls which will be very inefficient. We could also implement this with only one function delivering always all information or even with one complex function which takes arguments to indicate which information is needed, thus selecting the information already at the service. The first will generate relative very much data traffic whilst the latter will be complex to use and implement. In figure 4.4 we give two extreme examples.

It is the task of the designer and implementer to find at least an acceptable balance between many functions, complex functions and high data traffic within the target environment. This balance often comes down to experience and experimentation with a prototype or partial implementation of the service and also depends significantly on the environment (speed of the device a service is

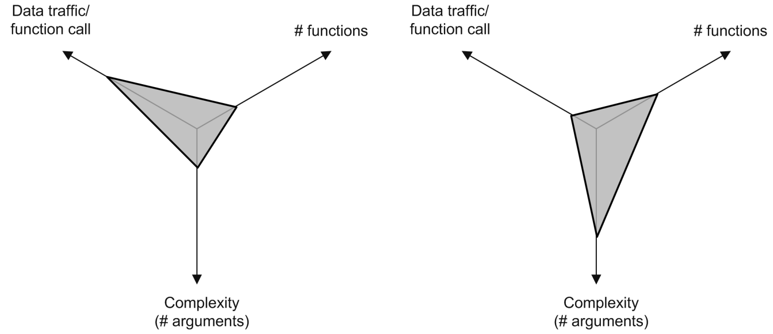

Figure 4.4: The grayed out area represents the overall cost of the service interface design. The axes go from the center (little functions/arguments/traffic) to the outside (many functions/arguments/traffic). The gray triangle intersects the axes reflecting the choices made. The left picture could be the situation with a fast (LAN) network, the right situation could be on a wireless and/or slow network.

running on, speed, latency and packet-loss of the network used etc.) the device and service are intended to operate in.

Next thing to keep in mind is event propagation. If an event is started at one service, but this services has many subscribers which each have many subscribers etc. the time for sending one event increases exponentially. This of course is a bad thing, it could even be that if a service is used by 100 users eventing can get pretty slow because of the peculiar design of the eventing protocol in UPnP.

UPnP uses the GENA eventing protocol which is based on TCP and when a UPnP service notifies a subscriber about an event the subscriber has to confirm the reception before closing the TCP connection. In most implementations only after the confirmation the next subscriber is notified. When however the subscriber also has to notify some subscribers in a second layer it would do this immediately after receiving a notification. However the confirmation is only sent after completing the notification so before the original UPnP service sending the event has to wait for its subscriber to complete its notifications (see figure 4.5).

So to avoid this cascade of notifications a subscriber which is notified should start notifying his subscribers in a separate thread so control can immediately be returned to the service which can continue sending notifications instead of blocking for a long period of time.

Lastly useful tools for the creation of the device and the service XML files are not included in the Cyberlink API/toolkit. Intel provides, only for Windows, a toolkit [11] which provides software (Device Author) to create these XML files. This toolkit can be used separately from the Intel UPnP stack and provides a more friendly interface for creating device and service specifications.

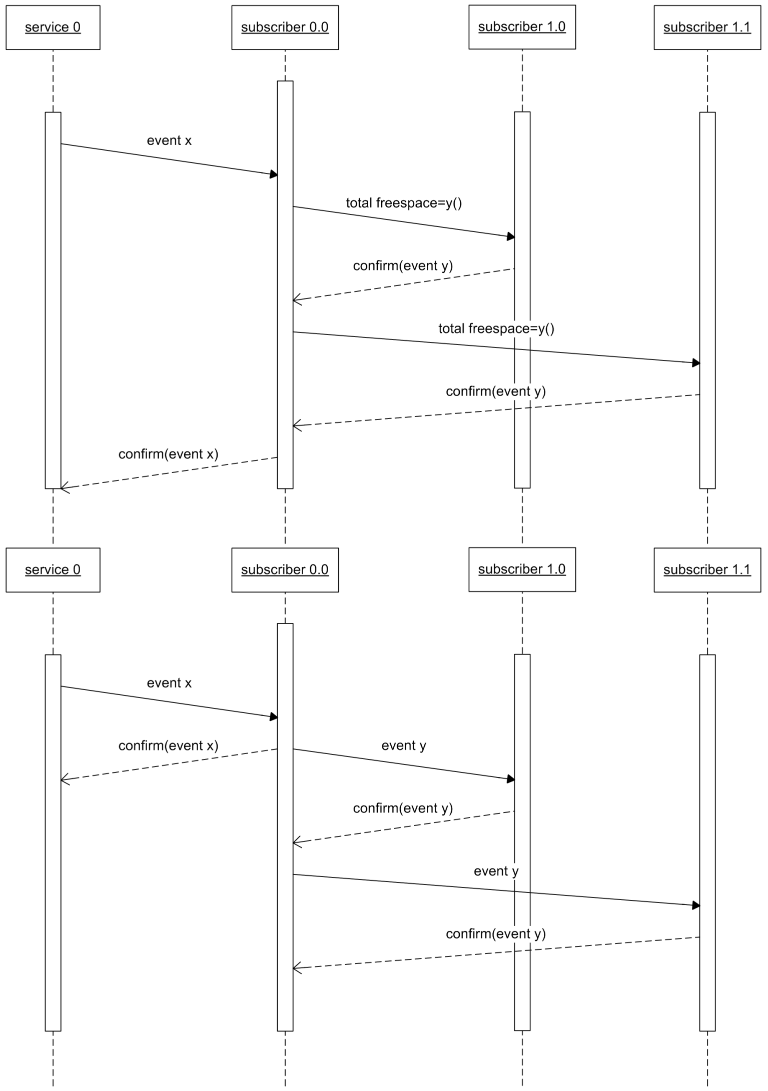

Figure 4.5: The above schema is what happens when using the naive approach, while the lower schema happens when we use a separate thread for event propagation starting immediately after the original event was received. Obviously service 0 can continue earlier with notifying other subscribers in the second case. Subscriber 1.0 depicts subscriber 0 of service 1 just as subscriber 0.0 is subscriber 0 of service 0.

Chapter 5

Distributed Storage Network design

As an example of the possibilities of Service Oriented Architectures and UPnP we developed a Distributed Storage Network. It offers us a play-ground to experiment on and has some nice properties which can be tested and demonstrated:

- It offers a storage space which can be easily extended by just adding another memory device.
- It can make files available where there is no (or small) permanent storage available, for example in sensor networks.
- It can be a ubiquitous storage, providing general access with (possibly) location dependent context.
- It can be used to store data in a transparent way because the interface of a service can be discovered automatically.
- It can be constructed to be fault tolerant offering safer storage than a normal hard disk.

Instead of storing data on your hard drive you can store it in the Distributed Storage Network which will provide a directory and file structure for retrieving your data and provides space for storing the data.

When a user starts his client he doesn't have to do anything, except wait a few seconds before a service is discovered. After that he can use it just like an ftp program. This solution could even be integrated in the operating system to look just like another network drive.

The goal is to create a distributed file system with the same functionality as a file system used for hard drives e.o. This means storing data in a structured, retrievable way and offering functionality to store, locate and retrieve the data. Additionally it should be possible to store meta information on data (i.e. file name, date stored, readable/writable, user permissions etc.). Something not specifically considered in the initial design, however kept in the back of our heads is that we might want to be able to extend the architecture with fault tolerance and security in the future (as discussed in chapter 3).

5.1 File System

Actually we designed the file system from two directions toward each other. On the one hand we had the inspiration from original old style file systems for hard disk, floppy disks etc. File system design is discussed extensively in [8] and [4]. It will not always be explicitly noted in this section, but many ideas are borrowed from these two sources.

On the other hand we started with the functionality we want to provide and divided it into several logical coherent parts. This was actually the starting point, however to solve problems we encountered we used the information on file system design we already had.

Splitting the functionality we want into several areas gives us this list:

- Store data (in a file)
- Retrieve data (from a file)
- Locate data (find a file)
- Alter meta information (of a file)
- Retrieve meta information (of a file)

We can already organize the first three in two different services, we need services to store data (and to retrieve it from) and a service to locate the data. To fit in with other file systems and for convenience we use a directory structure to locate the data, so we have a root directory which can contain directories and files. In the section 5.3 we will come back to all services introduced here and provide a more detailed description of their interfaces and interactions.

Each directory can contain other directories and files. The files are actually data structures which store a list of references to the locations of the data. The data itself will be stored on dedicated services. This directory structure can also be used to store meta information about the directories and files. Moreover directory listings can be enhanced with date/time information etc. We introduce the DirectoryService to maintain and store the directory structure and meta information.

Since we introduced a state for the DirectoryService it now is difficult to exchange one DirectoryService for another or to have multiple DirectoryServices running in one DSN (this would require some form of synchronization). In section 5.6 we suggest an alternative implementation of DirectoryService which avoids this problem.

We call the service which stores the data the DataStorageService. Just like a physical hard drive a DataStorageService doesn't store files, but blocks of data. Some blocks together form a group and every group gets a unique identifier. One group can be a complete file, however a group can also be a part of a file with another part on a completely different DataStorageService or in another part of the same DataStorageService. Like this all space on a DataStorageService can be used, even if a complete file wouldn't fit anymore. Moreover a file can be split up over various DataStorageServices if it would be to big to fit on one DataStorageService.

The references in the directory structure stored by the DirectoryService actually are a combination of two things which uniquely identifies a datagroup. The reference contains the datagroup identification number and the unique service identifier (which identifies the DataStorageService on which the datagroup is stored and does not change when a service is restarted see [7]).

This is illustrated in figure 5.1. A file can now be reconstructed from this list of references and it can be stored by finding free space on DataStorageServices, retrieving a datagroup and DataStorageService id and storing these id's in as references in a file in the directory structure. If there are multiple DirectoryServices in the DSN storing a file requires updating all DirectoryServices, so all DirectoryServices still represent the same directory structure.

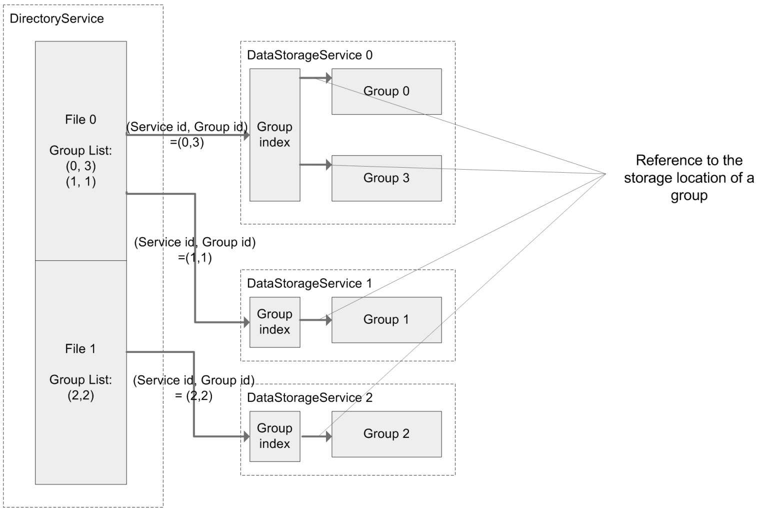

Figure 5.1: The links between the DirectoryService and DataStorageServices for two files in the DirectoryService.

Compared with a file system the reference in the DirectoryService to a group of data can be seen as a link to an indirect block in a file system. This indirect block in a file system then contains the links to the actual file data blocks. Just like our DataStorageService has a list of data blocks belonging to one datagroup.

We now have created a system which offers a very rudimentary interface for storing data. Actually we only created two separate services which cannot cooperate yet. We need an easy to use interface which will act as glue between the user, the DirectoryService and the DataStorageService. This service receives an entire file, distributes it to the DataStorageServices and updates the DirectoryService accordingly. The same service, which we'll call the FileService, will also be used to do exactly the reverse: retrieve all file parts and merge them together to create a new file which can then be transported over the network and returned to a user.

For users we introduce a user interface which acts as a service-user. It is implemented using the UPnP control points. This is a graphical user interface which can be used to 'upload' and 'download' files from and to the DSN. For this purpose it offers a graphical, browsable view of the directory structure in the DirectoryService like for example the directory tree in the windows explorer. However this could also be a daemon integrated into an operating system to behave as a file system which is accessible for all programs running on the computer, just like in Unix operating systems it is possible to use NFS for remote file systems.

5.2 Difficulties

Now we have a very coarse sketch of the system, however we can notice two difficulties with this system. The first is that we have services which have state. We already suggested this can be solved for the DirectoryService, but for the DataStorageService it is impossible to avoid storing data. This means we have to differentiate between the DataStorageServices (as we already do by storing their unique identification in the files in the directory structure). This also means we have a more complicated system than would be usual in a normal Service Oriented Architecture where usually every service instantiation can be replaced by another instantiation.

Secondly we cannot use UPnP for data transfers of files. The reason for this is rather obvious, UPnP wraps all its communication in SOAP messages (an XML based messaging framework). This of course takes time and can take a lot of time when we need to send lots of data. Moreover on a slow network or when sending much data the transfer can take a long time, which will block the communication interface of the UPnP service and will exceed the UPnP timeout of 30 seconds. In such a situation the UPnP documentation suggests we use another method for transferring our data.

The transportation of the actual data between the services will be handled by a simple TCP socket connection, which of course could be replaced by another arbitrary means of transport like the FTP protocol. Building facilities for this type of large data communications into UPnP might be a good idea. This situation does not only occur in a storage network, but for example also when using UPnP for distributed computing there is a large amount of data to be transfered. In chapter 7 we will come back to this shortly.

5.3 Service specifications

Now we have determined the functionality of every service we will have to specify this functionality more accurate in the form of action with parameters and an expected behavior. We will first discuss the interface and specification of the DataStorageService and the DirectoryService. Next we will see in the FileService how they all interoperate and connect with the FileService.

Something of great importance for the FileService and for a service-user is to be able to discover other services, more precise the FileService needs to find a DirectoryService and as many DataStorageServices as possible. The first it needs for finding and storing information on the directories and files, the second it needs to store the actual file data. For this we can use the UPnP build-in discovery and advertisement system. Using the specification of the functionality of the services in this chapter we were able to create the XML description files the advertisement system of UPnP uses to specify the interface of a service. An example of such an XML file can be found in appendix B.

If a file service is started it will broadcast its presence using the XML advertisement, but it will also try to discover all DirectoryServices and DataStorageServices currently online. It does this by using the discovery mechanism of UPnP, which broadcasts a search request over the network to which other services have to respond, if they fit the search description.

5.3.1 DataStorageService

The DataStorageService is, considering its interface, the most simple service. It has two functions and one event:

storeData The storeData action is used to store data in a datagroup. Its parameters are the size of the data which has to be stored, a reference to an already existing datagroup (if this is empty a new one is created) and an address in the datagroup where the data has to be stored (0 if it is a new datagroup or if the entire datagroup is to be overwritten). It reserves storage space for the data if a datagroup is extended or created (if possible of course) and assigns a reference id to a newly created datagroup and a reference id for the communication. Both, the datagroup reference id and the communication reference id are returned as a response indicating a successfully completed action.

retrieveData The retrieveData action is used to retrieve a datagroup. It uses similar parameters as the storeData action, however the reference id of the datagroup has to refer to an existing datagroup. The action checks if the datagroup is available and if it is it returns a reference id for the communication.

getHostInfo This action is used to retrieve information on the host a service is running on. It takes no parameters, but responds with the address and port of the TCP socket accepting connections for transporting large amounts of data.

freeSpace This is an event which reports the space available on the DataStorageService.

Storing data happens by first calling the storeData action with the appropriate parameters. Than the getHostInfo function has to be called (only once, when a DataStorageService comes online actually) to retrieve the address and port number for transferring the actual data. Finally, if all was successful, a TCP communication can be set up and the data can be send for the datagroup, however prefix with the communication reference id. The DataStorageService uses this id to match the data transfer with a successful storeData or retrieveData action call using the parameters provided with these action calls to store the data in the correct (location in the) datagroup.

Retrieving data works similar. First a retrieveData action is called, again with appropriate parameters. The address and port of the TCP socket for transferring data is again retrieved using the getHostInfo action if it is not yet known. Finally we open a TCP connection over which we send the communication reference id. Once the DataStorageService receives this id it determines this belongs to a specific retrieveData action and starts sending the data requested in the retrieveData action.

For an illustration of how this service is used we refer to section 5.3.3 where we discuss the general operations for storing and retrieving data using the FileService.

5.3.2 DirectoryService

The DirectoryService provides actions for all basic file system operations: createFile, createDirectory, deleteFile, deleteDirectory. These operations work on a directory structure which is similar to those found in *nix file systems. There is one root directory, directories and files can be created in this root directory or in other directories. Like this we can create a tree structure with directories as nodes and files or directories as leaves. There should not be two directories or two files with the same name in the same directory.

Every action operating on files and directories requires a path to the directory in which the action should be executed. This path consists of directories separated by the "/" character and uniquely identifies a path along the directory tree. The other parameter required for every action is the file or directory on which the action should operate. For example for creating a new file ("test.txt") in the root directory we would call createFile(path, file) as follows: createFile("/", "test.txt"), when we would want to create the new file in the sub directory "tmp" of the root directory we would use: createFile("/tmp", "test.txt"). All actions respond with a boolean value indicating success or failure of the action.

Other actions the DirectoryService offers on its interface are:

getListing This action requires only one parameter, the path of the directory of which the listing is requested. It will respond with a XML encoded directory listing of the directory path provided and all its subdirectories (i.e. the entire tree below the directory indicated by the path is returned), including all information on directories and files like creation date, file size etc. It is very simple to use as is illustrated in figure 5.2.

getFileInfo This action requires a path and a file name. If the file indicated by the file name exists in the path provided, all information on that file will be sent as a XML encoded response. This information includes: file name, file size, creation date, modification date and the list of references to file parts with their exact addresses within the file and length.

updateFilePart This action also requires a path and a file name. If the path/file combination is valid (i.e. the file can be found in the directory structure) the other required parameters of the action are used: a reference to the file part which has to be added or updated, consisting of a DataStorageService id (*serivceID*) and a datagroup id (*datagroupID*). These two parameters are used to map the file parts in the directory structure to the datagroups which store the data on the DataStorageServices.

However a file part has more properties, just as a datagroup it has a size (*length*), which is also a required parameter, and it has a starting address (*address*) within a file (which a datagroup does not have) to allow reconstruction of the entire file from its parts. The effect of the action is that either a. a new file part is created for this file, the start address of the file part in the file is *address*, the size is *length* and the data can be found at service *serviceID* in datagroup *datagroupID* or b. an existing file part (identified by the *serviceID* and *datagroupID*) is altered to have the starting address *address* and size *length*.

updateFileSize This action requires, just as the previous two actions, a path and a file name to operate on. Further it requires an integer value representing the new file size. If the combination of path and file exists the file size will be updated and a response will be sent indicating success.

The getListing and getFileInfo action can return a big amount of information in some cases, for example if we request the listing of a root directory in a very big directory structure. We still choose to specify their functionality in this way because we do have a very fast network which minimizes the communication time necessary for large amounts of data. Moreover we are mainly interested in testing if and how well our DSN works, maybe in a later stage when large directory structures will be used (which we probably won't be doing in our tests) we will decide to limit the getListing action to return only the directory tree with a depth of one directory instead of the entire subtree. Also if the getListing action shows to be a significant performance bottleneck in our tests we can still easily alter or split it (as suggested in chapter 4 to improve the performance.

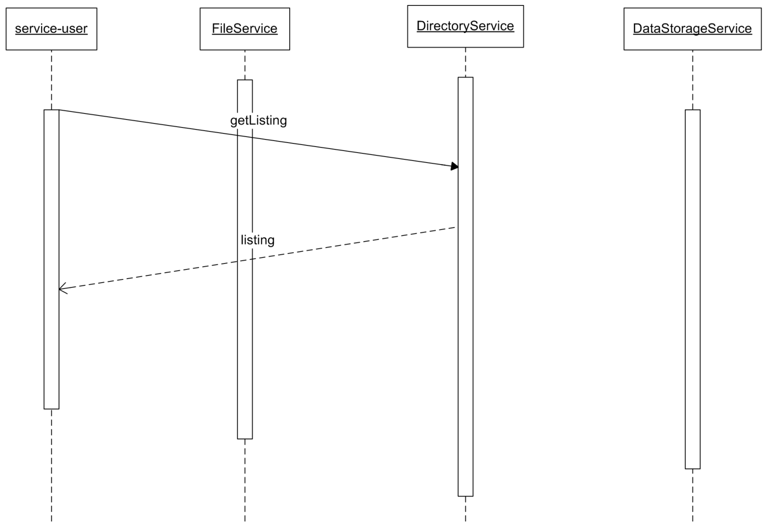

Figure 5.2: The getListing action only requires access to a DirectoryService.

The getFileInfo function returns all available file information including the file parts, thus it has a partial overlap with the getListing function. However since we know we will need the file information in situations where we will be using only one of the two functions we will include the information in both of them.

Extending the interface of the DirectoryService with more actions can be useful. We could add an action to add special attributes to a file/directory or we could create actions which allow a service-user to change the modification date of a file/directory. We are currently only interested in the basic functionality so therefore we don't include these additional features.

5.3.3 FileService

This service is where file parts and directory and file information get 'connected', files are split in parts for storage and parts are merged to create files upon retrieval. The communication for retrieving and storing files will go through this service. It uses DataStorageServices to store file parts (storeData action of the DataStorageService) and saves the location of these parts in a DirectoryService (updateFilePart action). Vice-versa it uses the DirectoryService to retrieve the location of file parts (getFileInfo action) and the DataStorageServices to retrieve the actual data of the files (retrieveData) which will be returned to the original control point.

So the FileService handles all file functionality (at least the functionality which involves the handling of data). Thus it needs a function to read from a file, to write to a file and to create a file. Eventually we would also require an action to delete a file, but for our testing purposes we will not implement this yet.

Since we are creating a file system we want to give the read and write actions of the FileService an interface which is as flexible as possible. We introduce two actions, the storeData action and the retrieveData action. The first can store data in a file, the second can retrieve data from a file. However instead of first retrieving a file handle by opening a file and then using this file handle to write to or read from a file, we provide both actions with the path and name of a file.

This avoids the creation of a direct link between a service-user and a specific instantiation of the FileService, which would exist when we would use a file descriptor. On the other hand we now have to use a full path and file name every time we want to read something from or write something to a file. However on the service-user side we could build a component which has a normal file system API interface, but internally translates those function calls to action calls using the FileService interface.

The actions are now defined as follows:

getHostInfo This action has exactly the same functionality as in the DataStorageService.

createFile For now this action has the same interface as the createFile action of the DirectoryService. It forwards the action call to the DirectoryService, receive a response and returns the response to the service-user. It is useful for testing actions traversing through multiple services and it offers convenience to a service-user who does not have to communicate with to different services to create a file and store data in the file. It is especially a good test-case since we are interested in the (performance) effects of services using other services.

storeData The following parameters are required: *path* and *filename* to identify the file in which data is going to be stored, *startAddress* to indicate at which address in the file the data has to be stored and *length* to indicate the size of the data which will be stored. The exact working of the storeData action will be discussed in the next paragraphs, but eventually when everything was successful the action will return a reference id for the data communication.

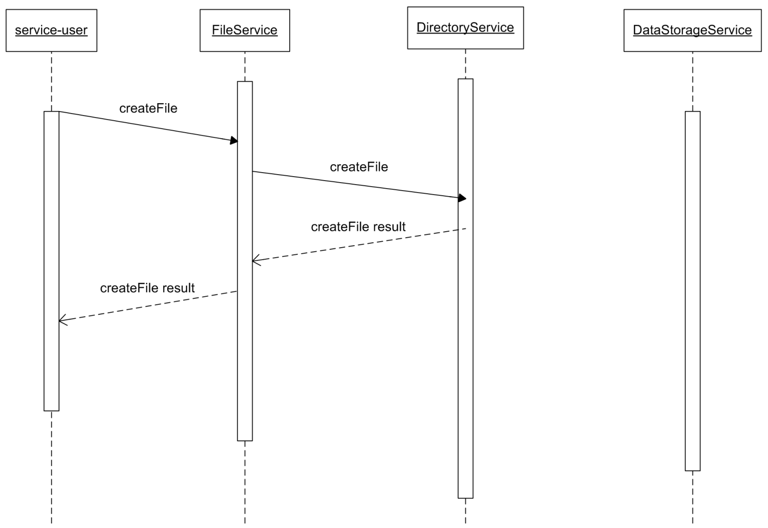

Figure 5.3: The createFile action just relays all its parameters to the createFile action of the DirectoryService.

retrieveData The retrieveData action has the same parameters as the storeData action, however now they are used to indicate which data will be retrieved from the file. It also will return, when everything was successful, a reference id for the data communication.

freeSpace The freeSpace event is the accumulation of the freeSpace events of all DataStorageServices. This means that it represents the total available space on all DataStorageServices currently discovered by the FileService. When the freeSpace of a DataStorageService changes the FileService receives this event and changes its total freeSpace and sends out its own freeSpace event.

In figure 5.4 we can see how the interactions of a storeData action work and in figure 5.5 we see how a retrieveData action works when a user calls one of these actions. Starting with the storeData action we see that after a FileService has received a storeData action request it almost immediately starts calling other services.

First the information of the already existing file (which was created using createFile of the DirectoryService or the FileService) is retrieved from the DirectoryService. If the file does not exist the storeData action immediately terminates and reports the failure to the service-user (at the first dotted line with the ”storeData failure” response). Now the file information is retrieved the file size is known and all existing file parts are known (including their size and location). If the address at which data has to be written falls outside of the file we again report failure (also at the first dotted line).

However if all these checks are passed the next check is if we are overwriting data or if we are adding data. If data is being overwritten the already existing file parts with their datagroups will be reused, otherwise the FileService checks for freeSpace on the DataStorageServices. In both cases the storeData action of the DataStorageServices we want to send the data to are called to reserve the space and receive a communication reference id for sending the data later, all changes which have to be made and all reference ids are stored so they can be used when transferring the actual data.

Immediately after this the file parts are also updated in the directory structure to reflect the new situation and eventually the file size is updated. All of these actions can fail for many

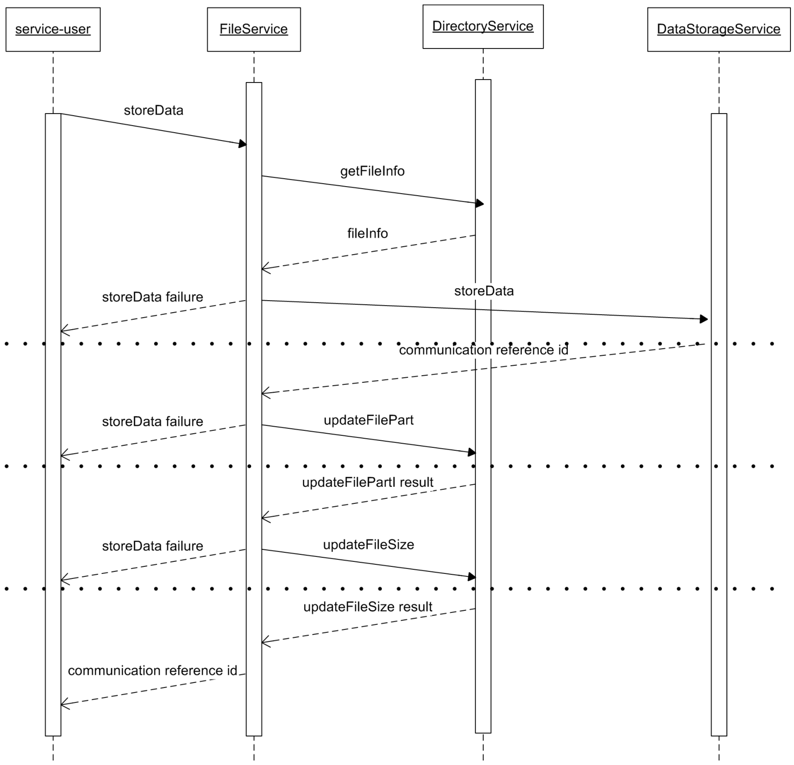

Figure 5.4: The storeData action execution. The file consist of only one datagroup, if there were more data groups there would be more storeData and updateFilePart calls to multiple DataStorageServices.

reasons, which, depending on the severity of the failure, will result in the failure of the entire storeData action which is reported back to the service-user resulting in an early action response and termination at one of the dotted lines in the figure.

Finally if all actions have completed successfully we have reserved enough space to store all the data and the storeData action can send a response indicating success and also sends a communication reference id. The service-user now can use the host address and port information to open a TCP connection to send the actual data which has to be stored to the FileService (not in figure 5.4 anymore). It will prefix this data with the reference id it received from the storeData action so the FileService can identify what has to be done with the data it is receiving. It will use the information it gathered in the storeData action (communication ids for DataStorageServices, file part addresses, locations and sizes etc.) to immediately sent the data to the appropriate DataStorageServices as explained in the section on DataStorageServices.

The retrieveData action works similar. However because it does not store data it is a little bit easier: it only retrieves the file information (if the file exists) and checks whether the data requested is actually available in the file (i.e. if the sum of the start address and the length does not exceed the file size). It then uses the retrieveData action of the DataStorageService to retrieve communication ids for the data requested and stores this information again. If this all was successful it will return a response indication success and also return a communication id.

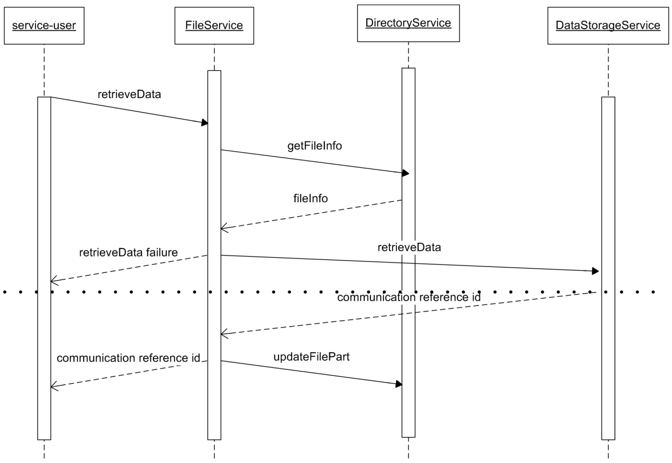

Figure 5.5: The retrieveData action execution. The file consists of only one datagroup because else there would have been more retrieveData calls to multiple DataStorageServices.

Again, not in figure 5.5 anymore, the service user can open a TCP socket to the FileService over which it sends the communication id it received. The FileService uses this id to determine that the service-user wants to receive data and uses the information it earlier stored to retrieve the data from the DataStorageServices.

5.4 Mapping services to UPnP services and control points

At first sight it seems like we have defined our services very nicely so they should map to UPnP in a very trivial way. However when we tried it, it turned out there were some expected and some unexpected difficulties.

To begin with a UPnP service cannot call another UPnP services. UPnP uses control points to connect to a service. In the documentation of UPnP there is no mention of being or not being allowed to combine the two into one application or service. So we just had to try this to see if it would work. So we created descendants of the GenericService class discussed in chapter 4. We created all the needed ActionProcessors and connected them to the actions described in the XML description of the services. Lastly we created one device XML file containing all the services, so we had one test device providing all three services.

To enable the FileService to communicate with the other two services we also created a FileServiceCP, a descendant of the GenericControlPoint, which we introduced in chapter 4. the FileServiceCP is interested in using functionality of DirectoryServices and DataStorageServices, so we only maintain records of the presence of these services (by using the functionality provided in the GenericControlpoint see section 4.3.1). Now we wrap the actions we want to use from the DirectoryService and DataStorageService in normal functions of the FileServiceCP class which are implemented as UPnP action calls and can be called by the FileService to accomplish its tasks.

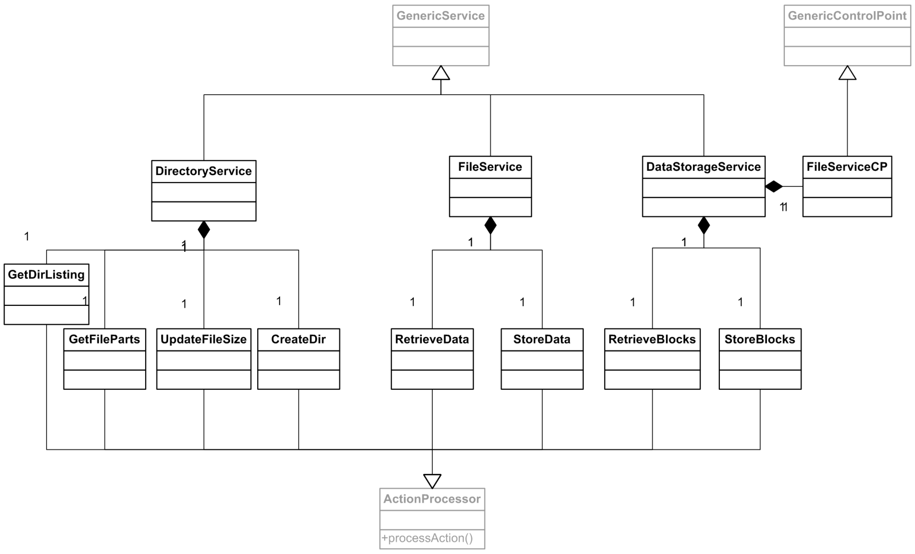

Figure 5.6: The UML diagram of the DSN, with all services, however without many ActionProcessor as they would only clutter the view. The gray parts are the framework classes. The ActionProcessor implement an action of a service, i.e. they are a separate classes used for implementing a function in a service.

All this results in the UML diagram in figure 5.6. The GenericDevice, which is not in the diagram, can use any of the descendants of the GenericService, depending on the XML description we provide for the device.

However while running and testing the newly created device with a very simple control point, just trying to store one very small file via the FileService, already deadlocked the whole device but all services were working separately as specified. The cause of this turned out to be that the UPnP API uses only one thread per device to handle actions. So if an action of the FileService is issuing an action request to the DirectoryService in the same device it will deadlock because the new action has to wait for the FileService request to complete, which in turn is waiting for the new action to complete.

So we concluded we could combine a UPnP service with a control point to allow a UPnP service to connect to other UPnP services, however those services should not reside in the same device as the calling service. After scattering the three created UPnP services over three different devices (not necessarily physical devices, just in the UPnP terminology) we could use the functionality as intended.

Lastly we created another control point, the BrowserCP, which would be used by our browser. The browser (which is the actual service-user) offers a user interface, which gets updated by events of the BrowserCP (which in turn can receive events from the UPnP services) and by calling actions of the FileService and DirectoryService via the BrowserCP. These actions are again implemented as normal functions of the BrowserCP enabling us to separate the UPnP parts from the user interface parts.

With the BrowserCP we created an observer pattern so if there is a change in for example the directory listing the user interface can immediately be notified of this. After some experimenting it turned out this would probably generate an overkill of events (at least if many users would be using the DSN) because it is impossible to generate only an event when an interesting part of the directory listing changes for example (see chapter 7 and section 4.2 for more details on this). So we changed the user interface to query for new directory listings while a user was browsing through it, and only updating the user interface if something actually changed (using the same observer pattern).

5.5 UPnP as a platform for SOA's

Using UPnP, with the added wrapper, turned out to be rather simple. Most difficulty was found in transporting large amounts of data since we had to implement our own system for this. This was something we had not expected initially, however building a standard solution which we could use in both the FileService and the DataStorageService proved to be not to difficult. Without the wrapper around the existing UPnP API things would have been a bit more complicated, however depending on the API one uses experiences may vary greatly.

It actually comes down to making the UPnP protocol as transparent as possible not requiring much detailed knowledge about the protocol and its internal functionality before being able to use it in a useful way. Compared to the Cybergarage API Intel has done a better job at this. They provide numerous tools with their API to make it easy for a developer to use UPnP. There is no need to write some sort of wrapper like we did. On the other hand the Intel toolkit is not as freely usable as the Cybergarage one and we would not have been able to execute the extensive tests we did because Intel does not provide the source code of its API.

Something all UPnP API's will 'suffer' is that it is not possible to hide the loose coupling of services, so when writing control points it is always necessary to take this loose coupling into account. One always has to consider that some service is used, over a network, and can thus give performance problems.

Something which, in our case at least, turned out to be very easy to use was the automatic discovery. In general it doesn't even take a few seconds for a control point to find some devices/services it needs, so there is not really a startup time, but there is the benefit of not having to bother with annoying ip-addresses or hostnames to find your information.

The most important problems and difficulties we encountered in developing our DSN were:

- Designing a service architecture with a specific performance is not very easy (as we discussed in chapter 4
- Mapping of services to UPnP services and control points is not as trivial as one would expect because of the limitations of UPnP (see chapter 4 and earlier in this chapter).
- Moreover it is quite easy to do things in a clumsy way with UPnP causing unnecessary bad performance or errors. For example the way events are handles discussed in section 4.3.3.

- Reasoning using services takes some getting used to. Also because there are so many different opinions on what a service is and what the relation between a service and a component is.

Adding functionality to an existing system of UPnP services *can* be very easy if it is not too intrusive, we present an example in chapter 8. We have to put emphasis on *can* in the previous sentence because it depends very much on the existing interfaces of the services and how they already interoperate. For example putting a service in between the communications of two already existing and working services is quite difficult without modifying one of the two since they will both completely ignore the existence of the new service. However an orchestrator can avoid this problem. The orchestrator decides how a composition is made out of two passive services, instead of one service actively using another service (the approach we choose). The latter limits the possible extensions, while an orchestrator can decide to compose services in a slightly different way allowing a service to fit in between the already existing services.

What is possible though is to add a new UPnP service with a control point to the network which communicates with existing services through the already existing interfaces. This new service could then provide extra functionality for the information already present in the SOA. The new service can be used as a completely new and separate view on the original application, but it can also be a management service taking care of some administrative tasks which enhance the capabilities of the application. The example in chapter 8 can be placed in the last category since we introduce replication for the sake of fault tolerance.

Extending an existing application with additional service will in most cases put an extra load on existing services. This is something which has to be taken into account; the extra load can slow down the entire application and require faster hardware or extra services (and extra hardware for the services to run on) to regain its original performance. Eventually the added load may even cause failures in the application which can be noticed by the user if no additional measures are taken.

5.6 Alternatives

In the previous sections we discussed the design actually used in the implementation. However choices made there are rather arbitrarily and could have gone in some other direction. In this section we will look at some possibilities different from the one used.

Instead of storing the data of the directory structure in the DirectoryService we could also store it using the DataStorageService (more similar to a real file system). In the introduction we noted that we wanted to be able to extend this system easily and we wanted it to be fault tolerant (or at least it should be easy to make it fault tolerant). Storing the directory structure in DataStorageServices makes meeting both requirements easier. Since a DirectoryService will not be storing any data it self anymore we can easily extend the DSN with additional DirectoryServices.

Extending the DSN with multiple DirectoryServices makes it immediately fault tolerant because now one DirectoryService can fail without having major consequences for the entire DSN. We intend to use replication on DataStorageServices (as will be discussed in chapter 8) to make the data storage more fault tolerant. When storing the directory structure using the DataStorageService we get replication of the directory structure for free.

Downside of this solution is that there will be added complexity in retrieving a directory listing or changing meta information of a file or directory. This could require multiple requests to DataStorageServices and slow down the retrieval of a directory listing significantly. Further needs to be decided where to store the directory listing (and meta information) data. First and maybe best idea is on the same service or 'close to' the location of the file parts. Because when we loose a directory listing because of a service crash, we can't reach the files anymore, but when we loose file parts the directory listing of them also becomes relatively useless.

So we might just as well loose them (file parts and listing of those file parts) together instead of loosing some directory listings and some file parts, of other files than in the listing, thus loosing in total more information. 'Close to' should thus be considered on for example the same connection or reachable through the same router. A file would by default be stored on the same DataStorageService as the directory information, however if it fills up the DataStorageIdentification field can be used again to refer to file parts on other DataStorageServices (close to the current one).

The directory information can be treated as normal files using the normal functionality of the DataStorageServices and FileService(s) to store and retrieve them. This can be XML files for convenience, but more space efficient would probably be some binary format. And to solve the probably significant slow-down caused by retrieving directory information from DataStorageServices we can cache the directory information in the DirectoryService. Then on the first request of a directory listing the DirectoryService retrieves the entire directory structure which it stores in its local cache. Changes to its structure are stored in cache, but are also written to the DataStorageServices again so they don't get lost when the DirectoryService crashes.

From time to time it checks if the cached version is still consistent with the stored version, however this can be done even in the background so it won't slow down user requests. This would also prevent the loss of the directory structure when a DataStorageService which stores (a part of) the directory structure would go offline, the DirectoryService could simply write the information to a different DataStorageService.

Only if a user request cannot be completed with help of the cache the directory information is fetched again from the stored version because it is possible that the cache isn't up-to-date anymore because of changes by for example other DirectoryServices.

When the DirectoryService is stripped down a bit for example like illustrated before it could be combined with the FileService thus reducing the number of services and reducing the amount of communication necessary. This would enable us to think in a different way of the system. The FileService+DirectoryService now together form an InterfaceService to the data stored on the DataStorageServices. This InterfaceService provides users with the information they need (access to files and directory listings) much like an operating systems provides a similar interface to applications running on it.

Chapter 6

Performance Analysis

In this chapter we will be searching for an answer to several performance related questions. Not only will we try to determine if UPnP based systems are fast and responsive enough to be used instead of normal applications. Actually we will be doing this performance analysis with the following 5 questions in mind:

- How fast does UPnP actually perform? Is this usable in real-world applications?
- Is there a model which can predict the performance of a UPnP based application? What is this model?
- What is the effect of interference from other applications (or other services/control points) on the performance of UPnP?
- What can we say about the performance of the toolkit used and can these results be used to evaluate other toolkits?
- Can we find results supporting or assumptions, advices and design practices from chapters 4 and 5?
- Can we improve UPnP or the toolkit? If so, where can we improve it?

The performance test will be executed on the current version of the Distributed Storage Network and will focus on the control actions of UPnP. The values we are going to measure and our expectations will be discussed in the first sections. In the following sections the problems and results of the measurements will be presented and explained.

6.1 System Parameters

As already mentioned the tests will be performed on the Distributed Storage Network presented in chapter 5, it will be running on a small Linux cluster in the computing science faculty of the Technical University of Einhoven. We will be using three machines directly connected to each other by switches, one of these will act as the host machine and has a second network interface connected to the university network enabling ssh-login. From the host we can login on the other to machines to setup the test configuration there. The specifications of all machines are:

Memory	256 MB (host 512 MB)
Node processor	PIII (Coppermine) 533 MHz
Operating System	Gentoo Linux 2.4.26
Quantum size (of process scheduler)	100 ms

A rather peculiar parameter in this list is the Quantum size of the process scheduler. However we will need it to be able to explain some of the test results, we will be returning to this issue later in this chapter when analyzing the results. The network parameters are:

Network bandwidth	100 Mbit
Network latency (ping round-trip), during ideal circumstances	0.1-0.2 ms

Factors we will be varying and measuring in order to determine its influence on the performance are:

UDP Network traffic The effect of heavy UDP network traffic additional to the traffic caused by the DSN

TCP Network traffic The effect of heavy TCP network traffic additional to the traffic caused by the DSN

CPU Load The effect of introducing a high CPU load additional to the load caused by the DSN

6.2 Performance Parameters

For our test we focus on the control actions and events because these will be the parts mostly used in an operational SOA. Services leaving and joining the SOA will be marginal compared to users or services using the SOA to perform certain activities. Moreover the current discovery protocol of UPnP is not very scalable ([12]) and already newer and better protocols are proposed (for a comparison of discovery protocols see [2] alternative discovery protocols: [15] or a similar, but improved, version as used in UPnP: [6]). The responsiveness and speed of control actions also are the main contributors to the (degradation of) the user experience, a user using the functionality of the DSN for example. So we will focus with our performance testing on these actions and are not interested in the performance of the discovery protocol.

Since we are using UPnP in an RPC like manner here we will be executing our measurements in a somewhat similar way as in [18] and [13] which also did performance measurements of Firefly RPC and DCE RPC. Just like they did we will also split the response time of an action or event into several parts. Measuring the different parts of the execution of a control action provides us with more information on what part of an action execution might cause a slow response time and can help us in fixing this bottleneck (some examples encountered during our tests are discussed in section 6.5.1).The different parameters we split an action response time (i.e. the time from initiating a request by the client until the time of receiving the response by the client) in can be found in figure 6.1.

In this figure we introduce a notation which uses functions to represent a few parameters while we consider the rest to be almost constant. The functions map from the action domain to the time domain. We define the set of control actions A, for every supported action a we have $a\epsilon A$.

Other parameters we consider constant because the influence of the different actions on these parameters is only in the size of all the action parameters together. Since most of the size differences we encounter cannot be really measured because of the interval of the timers used and the high speed of our network (100 Mbit LAN) and we are not really interested in the performance compared to the message size (necessary for an action) we do not introduce functions for these

$$\begin{aligned}
\text{Initiate Request} &: init_{req} \\
\text{Send Request to service} &: send_{req} \\
\text{Receive Request from control point} &: recv_{req} \\
\text{Process Request} &: process_{req} \\
\text{Calculation for Action} &: calc(a) \\
\text{Initiate Response} &: init_{resp} \\
\text{Send Response to control point} &: send_{resp} \\
\text{Receive Response from service} &: recv_{resp} \\
\text{Process Response} &: process_{resp} \\
\text{Total Response Time} &: total(a) \\
\text{with} &: a \, \epsilon \, A
\end{aligned}$$

Figure 6.1: Parameters measured during the test with their associated function(s)

parameters but consider them constant. In wireless networks or otherwise slow networks with low throughput it would be necessary to differentiate over the size of the messages necessary for an action though.

We have omitted the messages size (for requests and responses) from all functions and equations because some preliminary tests showed the influence of the size was marginal. This can easily be explained by the fast network used for communications (100 Mbit Ethernet LAN) and the relative small sizes of all messages; except for the directory listing, which can vary in size significantly, all messages are less than 2 KB whilst to be measurable they should be more than 10 KB which would take approximately 1 millisecond or more to transfer.

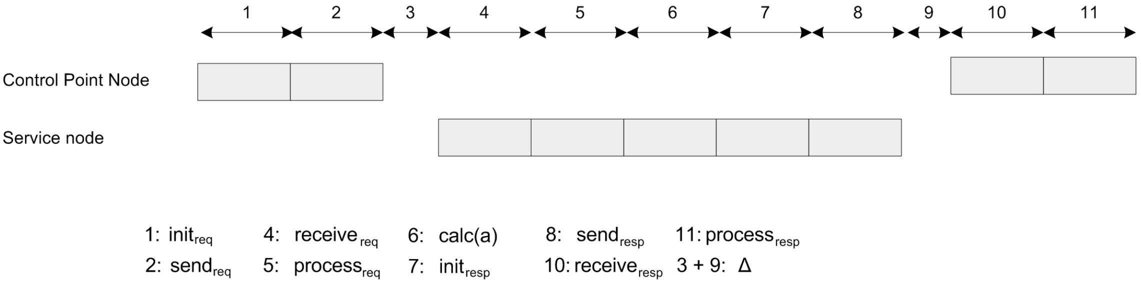

Figure 6.2: The order in which the performance parameters occur during execution of an action.

Now we will continue with our theoretical performance model for which we can see a graphical representation in figure 6.2, which depicts exactly how the different performance parameters can be combined to get a total response time of an action. Initiate Request and Initiate Response both represent the time needed by the UPnP API to collect all data necessary, marshall it in a SOAP XML structure and prepare it for transmission. Similar Process Request and Processing Response both represent the time necessary for the UPnP API to umnarshall the SOAP XML structure into the separate variables, store them into the API's internal structure and pass control and data to the service or control point implementation.

Send Request, Receive Request, Send Response and Receive Response all represent the time needed for sending, respectively receiving the SOAP XML structure. Additionally Send Request also contains the time needed to setup the TCP connection over which the rest of the communication takes place. The Calculation for Action represent the time needed by the service implementation to produce a response for the specific requested action and its parameters, this includes the calling of actions of other services.

Finally we have the total response time as observed by a control point calling an action, this is the sum of all other parameters together with a Δ, which represents the time we could not measure in a single machine, but only derive by combining other measurements. This Δ can be caused for example by delays in the network communications or by the scheduler of the operating system giving processor time to other processes. A big Δ is not desirable because it would mean there is a large external delay. In theory Δ could even be slightly negative since it is possible for the send and receive actions to occur practically simultaneously. The parameters in this sum can all vary independently to a certain extent, they are all influenced by external occurrences for example other processes running on the hosting machine. These external factors influence all parameters, however some are influenced more by some event than others. For example a lot of TCP traffic will probably have more effect on the sent/receive parameters than on the others.

6.3 Test design

We are going to measure the variables in the previous section by introducing a very simple and artificial workload to the system. In one test run we will be measuring all performance parameters of one service and for one specific control action. From the available services and control actions of the DSN we have selected the following:

- File Service
  - storeData
  - createFile
- Directory Service
  - getListing
  - createFile
  - addFilePart
  - getFileInfo
  - updateFileSize
- Storage Service
  - storeBlocks

These control actions together cover all possible types of actions, from very simple actions to the more complex actions and from actions only operating locally on a service to actions using one or multiple other services for completing their task. For a reference base we also create a new service offering only one control action, the EmptyAction. This EmptyAction will do no calculation, it will only respond with a empty result. Like this we can eliminate a possible influence of different calculations for different actions.

To make measuring easy for ourselves we created an automated control point which takes input from a text file. This text file describes which action of which services should be called with which arguments and how often it should be called. For every action call the control point samples the time in milliseconds at the moment of initiating the request and right after receiving the request. The difference is the total response time of an action.

This automated control point will run on one node of the test cluster. We always tested with only one service of every type so we would be certain about which service would be used for the test. Only the necessary services were running during a test (so only a DirectoryService was running when doing getListing tests, but for the StoreData tests we had all three services running because they are all three necessary). The service being tested was running on another node and if any other additional services were necessary they would be running on a third node of the cluster. The resulting time measurements were saved in a text file for later evaluation using a spreadsheet.

After measuring the total time tests we inserted time sampling statements in the code of the UPnP API and our applications code. These time samples also have a resolution of milliseconds, were also written to a text file for later evaluation and are situated at the points in the code where they can represent the borders of the different performance parameters. We actually measured these intervals (in a separate test run than we measured the total time), with their associated performance parameters:

$$\begin{aligned}
&\textbf{Control point measurements} & \\
\text{Initiate Request} &: init_{req} \\
\text{CP Sending} &: send_{req} \\
\text{Receiving Response} &: recv_{resp} \\
\text{Processing Response} &: process_{resp} \\
\text{Waiting for Response} &: - \\
&\textbf{Service measurements} & \\
\text{Service Receiving} &: recv_{req} \\
\text{Processing Request} &: process_{req} + calc(a) + init_{resp} \\
\text{Sending Response} &: send_{resp} \\
&\textbf{Other measurements} & \\
\text{Rest} &: \Delta \\
\text{CP Total} &: - \\
\text{Total Time} &: total(a) \\
\text{with} &: a \,\epsilon\, A
\end{aligned}$$

The measurements "Initiate Request" and "Processing Response" both measure the UPnP API performance, the first measures the time between invocation and the actual sending of the result, the latter the time between response reception and producing the result to the user program.

The "CP Sending" measurement includes the time necessary to setup a TCP connection to the service and sending the request. The "Receiving Response", "Receiving Request" and "Sending Response" times only include the time used for sending/receiving a message because the same TCP connection is used for all communications. Up until here all measurements can be directly mapped to the performance parameters we are interested in. However "Waiting for response" does not have an associated performance parameter, it contains the time the control point spends idling after sending a request and before receiving a response (see figure 6.3). This measurement will be used to determine Δ or "Rest".

The "Processing Request" measurement is the time the services spends on executing the UPnP API code and the action specific code (which should produce the response), which is clearly visible in figure 6.3. We didn't measure the three separate performance parameters enclosed in this one value because it is actually quite difficult to determine where the action code starts and where the API code ends. For example the code for executing an action contains code to extract parameters from the action request, but the time spent in extracting parameters is actually time used up by the API. Measuring this specific time is impossible because it is mixed with the code for executing the action itself. Some other API's use a clearer separation of API and application code by already providing the UPnP parameters as function parameters which would make it possible to give an accurate measure of the performance parameters.

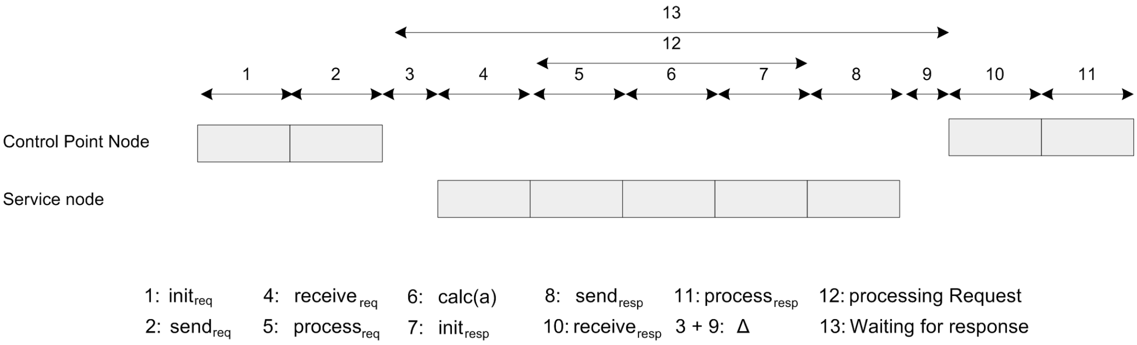

Figure 6.3: All performance parameters are mapped exactly to the measured values, except $process_{req}$, $calc(a)$ and $init_{resp}$ which are merged together as indicated by the "Processing Request" arrow (number 12).

The "Rest" measurement is actually not a real measurement, it" is the "Waiting for Response" measurement minus the sum of all service side measurements. This is the time spent somewhere on the control point and/or service machine which is out of the control of the test program (for example the time the operating system needs to free the network interface for sending the response), which we already referred to as Δ as a performance parameter. The "CP total" measurement also is not a real measurement, it is the sum of all control point side measurements (subtracted from the "Total Time". This measurement is only used as a consistency check, because not all measurements were done in one run (specifically the total time was measured separately). This might cause a small error between the total time and the sum of the other times, which is made visible using this measurement.

Because we did much measurements for every action we could easily determine where the slow parts of our current implementation were and we even fixed some of these parts. This also gave us insights for improving the UPnP performance since we can determine which parts take up a relative big part of the time necessary for an action and improve upon those parts. Some successful improvements we made to UPnP API with the help of these measurements are discussed in section 6.5.1, suggestions for even more improvements can be found in chapter 7.

For every control action we will repeat the measurements 500 times (in one run), afterward we will dismiss the first 100 to ignore an initial slow start noticed during a few initial tests. We will vary the previously mentioned interference factors (TCP traffic, UDP traffic and CPU load) between the maximum level (full load/traffic) to the minimum level (only the DSN load/traffic). This will help us in determining the effects these types of interference can have on the performance of UPnP and will help us in deriving a (simple) model for calculating the worst case performance of UPnP.

However generating UDP/TCP traffic from the same host system as the service and/or control point is running on will also increase the CPU load, so we will have to take this into account when evaluating the results. For this purpose we will also measure the CPU load and the network throughput on both the control point and the service side. For generating the CPU load or network traffic we created a java test application which contained the control point for one side of the communication and contained a service for the other side of the communication.

Before a test is started the test application it first starts a generator generating one of the interference types (for example TCP traffic between two nodes or a high CPU load by starting a CPU intensive calculation) and on both sides the test application recorded the CPU load and network throughput to a text file by means of sampling with 50ms intervals. Measuring the network throughput and CPU load only generates a very small overhead (between 1% and 2% CPU load) as we tested by running these measurements separately from the tests.

The TCP and UDP cross-traffic was generated so that every node which would participate in communication would have exactly one outgoing and one incoming data stream. This means if we have two nodes participating in the tests (the control point node and one service node) there would be two TCP or UDP streams simultaneously be sending data, one from the control point node to the service node and one vice-versa. If there were more nodes participating in the tests because we needed more services the TCP or UDP streams would create a ring structure so that every node still had exactly one incoming and one outgoing stream and all nodes would participate in the cross-traffic.

6.4 Expectations

We can pose a few expectations or hypotheses on what will we the results of our tests. In the next sections we can than find support for these hypotheses or we can disprove them. First we will discuss them all shortly in this section.

Performance during interference will degrade, but there may be a way to predict it

When introducing interference in the form of cross-traffic on the network interface or a high CPU load we can expect it to have a significant impact on the performance of UPnP. Especially in the cases we are testing with maximum interference possible.

Although this interference is dependent on the hardware setup and many more location specific circumstances we think it may be possible to predict the behavior of the different actions by just testing one action and using that as a basis to extrapolate to other actions. This can be done since we expect a lot of the effect to be noticeable in the Δ performance parameter and a little bit of the effect in the other parameters. However most of the performance parameters are constant, even most of the code executed is constant, irrespective of which control action is actually used, so if we can determine the influence on one action it should be not too difficult to determine it for other actions.

These results may prove to be very useful since high TCP traffic can be easily caused, even in our DSN, by transferring a lot of data between a control point and a service node (for example when a user is storing a big file in the DSN the FileService uses two TCP connections to transport the data at the maximum possible speed at the moment).

If we can predict the effect of the interference we can try to counter it by granting specific threads of the UPnP action handling mechanism a higher priority than for example the threads causing high TCP or UDP traffic.

Layered actions performance can be predicted

The performance of layered actions, i.e. actions which use actions of other services to complete, can also be predicted as long as the time necessary to complete the extra actions is known. Suppose b and c are actions needed to calculate the response of a, $calc(a_x), 0 \leq x < 3$ represent the possible calculations between the calls to these actions. We then expect the total calculation time for the request to be

$$calc(a) = calc(a_0) + total(b) + calc(a_1) + total(c) + calc(a_2)$$

with the functions as defined in section 6.2. This also suggests that we will have a bigger Δ in total because we have the $\Delta_b + \Delta_c + \Delta_a$ with Δ_a being the Δ of the original action request. Combined with the possible prediction of the influence of interference by network traffic or high CPU load we could also predict the influence of this interference on layered actions.

Comparable to existing RPC

We don't expect UPnP to do very well in comparison to existing RPCs. UPnP was not created with the intention of using it for RPC. Moreover it uses XML to communicate which is inherently slower because compared to binary formats it needs more bytes to transfer the same information and it needs an XML parser to extract the relevant data from the XML data.

Design Practices can be supported

We expect we can use the results to quantify the design practices suggested in chapter 4. Based on test of a few very simple actions we want to be able to determine with the help of a few simple formula's if a certain layering or service interface of actions is feasible given the fact that the actions will be used in a certain pattern (frequently, infrequently, automated, every x seconds etc..).

This can be done by determining the time necessary to complete one entire action on the hardware platform the eventual application is intended for. Than we can try and extrapolate this performance by using the predictions for layered actions and interference. With those results we can eventually determine if we think the resulting delay is still usable. For example an action which will take in total of 200ms but will be called 20 times a second will not be usable, but a redesign could fix this problem easy depending on where the slow performance originates.

6.5 Results

The averages of the measurements we did are represented in the tables in appendix A and in the graph 6.4. The averages are calculated over the 201 until the 500 measurement to avoid startup effects in the averages. The tables contain all measurements explained in section 6.3 and one extra column: the 95% confidence column. In the "95% confidence" column we calculated the 95% confidence interval for the "Total time" measurement. With this column and the total time column we can determine an interval in which in 95% of the times a similar test would be performed the result would occur in. This interval can be calculated by subtracting and adding the 95% confidence column to the "total time" (this would result in an interval of 8.0 - 8.8 for EmptyAction in normal circumstances).

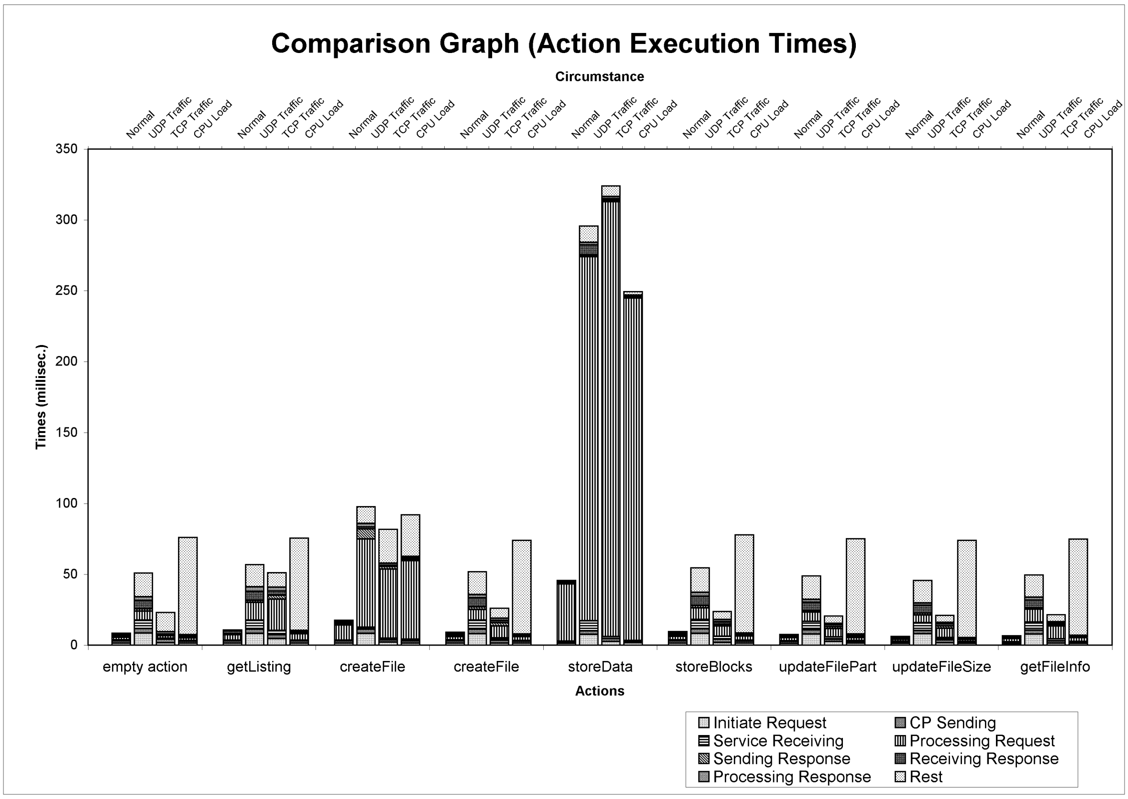

Figure 6.4: Comparison graph of all actions measured

In the remainder of this chapter we will continue first with some improvements we made to the software as a result of our test efforts. After that we will continue with some initial observations and a comparison of UPnP with some RPC protocols. We will continue with a more detailed discussion of the results which includes the prediction of the performance of layered actions, the effects of interference on UPnP and throughput predictions.

The graph in figure 6.4 is the graphical representation of our test results in appendix A. The results and this graph will be used and discussed throughout the rest of this chapter.

6.5.1 Improvements during testing

While running the tests we initially got very disappointing results (approximately 180ms for the execution of a single action). Since this was very slow we used our initial test results to research the cause of these disappointing results. Using the elaborate division in performance parameters we presented in the previous section helped us gain insight in the slow and in the fast parts of executing an action and revealed some unexpected performance parameters which we later eliminated again.

The first performance parameter we didn't account for was the unexpected long time necessary for receiving the in XML marshalled action calls from the network. Since this took a rather unexpected long time so we investigated it. It turned out the API created an array of 512KB every time data had to be received over an TCP socket. On our test platform this took a significant amount of time, so we reduced the size of this array (which doesn't effect the functionality at all) to 2KB which reduced the execution time of an action to only 18 to 20 milliseconds and the additional performance parameter for allocating the big array was removed since we could not measure the time necessary for the allocation anymore (the measurement resolution was to low).

In the next runs we still found the results not to be very good and discovered that the XML parser used by the UPnP API was a performance parameter of its own, so we replaced it with a more lightweight XML parser (kXML parser instead of Xerces). Simultaneously we replaced the XML parser used in our own service code with the same lightweight parser, because also there it showed up as a performance parameter we would have to account for. This again saved in total more than 10ms bringing the execution time of an action (in normal circumstances) down to a mere 6ms. Again XML parser got now fast enough that we could not separate the the time needed by the parser from the other parts we measured in the same interval, thus also eliminating this unexpected performance parameter.

These improvements were all rather straight-forward, easy to implement adaptations of the API and our own code. We could only track down the locations which needed improvements because of the design of our experiment using many performance parameters for the different parts of an action and deriving additional performance parameters where needed by the implementation. With the help of this type of testing more improvements can be made though they will require more complex changes to the implementation of the API and implementing them fell outside the scope of our research. However we will suggest possible improvements we found using our tests in chapter 7.

6.5.2 Initial Observations

When looking at the data and at the graph we can see the 95% confidence interval we calculated is relative small indicating these results are reliable. This is the confidence interval calculated over the total time. We didn't calculate it for the rest of the values since these are so small it would give no usable values. Moreover we have confidence in our results since the different measurements are over parts of code which to our estimate will need the time measured now for performing their task.

We also notice that the Δ or "Rest" values are all approximately 1ms for the normal case, although we stated this should be about 0 in ideal circumstances. However the service side of an action communication needs to create a socket before being able to receive data, this is something also contained in this 1ms because we only start measuring when our UPnP service starts reading data from the created socket. From experience with the control point side creating sockets and measurements on this we know this can take up to 1ms of time so for a big part we can contribute the Δ found here to the socket creation and of course also a small part is necessary for the actual data transfer between the two machines.

6.5.3 UPnP vs. RPC

From the graphs and tables we can immediately see that there are no actions performing faster than 6ms to completion. We could compare this result with some existing RPC systems, which are closest related to what we are measuring for UPnP here. Compared to dedicated RPC systems which usually don't use XML this is a rather poor result, they often reach results around or less than 1ms per action, however this is not only dedicated software but often also on dedicated hardware (which in numbers (processor speed, memory size, network bandwidth) might be slower than our setup).

For example the Firefly RPC system which is analyzed in [18] performed in 1990 already faster

	EmptyAction vs. NULL RPC			GetFileInfo vs. smallest RPC[3]		
	UPnP	DCE RPC	Firefly RPC	UPnP	DCE RPC	Firefly RPC
Communicate request	2650	2470	954	1700	4309	954
Communicate response	1450	2516	954	1200	2516	4964
Client time	2400	866	303[2]	1700	924	303[2]
Server time[1]	1900	614	303[2]	2300	614	303[2]
Total	8400	6466	2514	6900	8363	6524

Table 6.1: Comparison of DCE RPC, Firefly RPC and UPnP for executing a NULL RPC or EmptyAction, times are in μs.

[1]: Server time includes the time to execute the actual action/RPC provided which in the RPCs cases was a dummy function, but in the GetFileInfo was a real action executing code.

[2]: For Firefly RPC we only had a total calculation time, for comparison we split it equally between client and server

[3]: smallest RPC, but not the NULL RPC.

than our implementation on slower hardware. However it uses multiple client threads to reach its maximum throughput (using only one thread it performed equally fast as our UPnP implementation), something we would have to adapt UPnP for since our UPnP services only use a single thread to handle requests. It also runs on different hardware and is optimized for speed on that specific hardware handling many tasks usually handled by the operating system just to improve the speed. Another example encountered in literature is DCE RPC which reached in 1995 already speeds of up to 1000 RPCs in little more than 3 seconds [13], though using only bigger RPC packets caused it to run even slower than our UPnP implementation.

So although we cannot compare them very easily using absolute numbers, we can compare how the performance is divided among the different parts of executing a RPC (or action for UPnP). We can compare the Firefly and DCE RPC implementations with UPnP because both of the articles discussing their performance also split the execution time of a RPC in many different small parts, similar to what we did. Because the different measurements done for UPnP and the two RPCs are very different though, we will merge them in a few general categories so we can compare them more easily. For reference we will simply use the EmptyAction or NULL RPC since these should be comparable because they only execute the UPnP or RPC code and do not execute any real action. We will also use one action with a small data size for request and response for the comparison: for DCE RPC we used a data size of 1392 bytes of request data and a response with no additional data, for Firefly RPC we used the smallest tested version which is a 1514 byte return packet and a 74 byte request packet. For UPnP we use the GetFileInfo message with 571 bytes of request data and 1008 bytes of return data.

The categories we will be using are: communicate request (send + receive), communicate response (send + receive), client (control point) time, server (service) time. The client and server times are the time necessary for an RPC or action call without the actual communication necessary for that call (although the communication also uses time on the server and client that time is contained in the communication time) For UPnP we can calculate them, for Firefly RPC they are given in table VIII in [18] and for DCE RPC we can calculate them using table 2 from [13].

In this comparison it seems that UPnP doesn't perform to bad, it actually comes close to the performance of both RPC protocols. However we did not account for the fact that we tested on a (much) faster hardware platform, so we will have to scale all our results respective to the CPU power. Improvements in the speed of the network do not seem to be of big influence, as was also concluded during the Firefly RPC tests [18].

We will normalize all times, for DCE and Firefly RPC and for UPnP, to the number of instructions they execute using the speed of the processor in MIPS. Although this is by no means an exact approximation it will give us better comparable numbers than the current absolute times. Even

	EmptyAction vs. NULL RPC			GetFileInfo vs. smallest RPC[3]		
	UPnP	DCE RPC	Firefly RPC	UPnP	DCE RPC	Firefly RPC
Communicate request	1404.5	79.0	1.0	901.0	137.6	1.0
Communicate response	768.5	80.5	1.0	636.0	80.5	4.9
Client instructions	1272	27.7	0.3[2]	901.0	29.7	0.3[2]
Server instructions[1]	1007	19.7	0.3[2]	1219.0	19.7	0.3[2]
Total	4452	206.9	2.6	3657	267.5	6.5

Table 6.2: Comparison of number of instructions (in thousand instructions) executed by the CPU for DCE RPC, Firefly RPC and UPnP for executing a NULL RPC or EmptyAction.
[1]: Server time includes the time to execute the actual action/RPC provided which in the RPC's cases was a dummy function, but in the GetFileInfo was a real action executing code.
[2]: For Firefly RPC we only had a total calculation time, for comparison we split it equally between client and server
[3]: smallest RPC, but not the NULL RPC.

for the communication times this is still reasonable since most part of the time there is spent using the CPU. Doing these calculations gives us the results in table 6.5.3. The system parameters for the systems were: IBM RISC System/6000 model 520 and 530 (3̃2 MIPS [1]) for DCE, Micro VAX II CPUs (1̃ MIPS [1])for Firefly and a Pentium III 533MHz (5̃30 MIPS [1]) for UPnP.

As we can see both DCE and Firefly RPC are much faster in the execution of client and server side code, already in absolute time, but when we look at the number of instructions executed to complete an action (or RPC) the UPnP implementation needs more than a factor 20 more instructions than DCE RPC, not to mention Firefly RPC which needs only 2600 instructions.

.Even if we assume that the measurements are not that accurate (at least the UPnP measurement are absolute time, not CPU time used) the difference is so big we can see that UPnP is rather inefficient. For this huge difference in number of instructions needed we can find multiple causes. For one we know that both RPC protocols, especially Firefly, are optimized for a short as possible call-path. UPnP is not. Moreover UPnP uses XML as a means of communication, which requires XML parsing, which in turn requires extra memory, extra instructions and extra data copies. We will elaborate more on this in chapter 7.

Finally comparing our results with XML-RPC [25] could also be useful because it uses XML (like UPnP does) instead of a binary format to communicate. There has been done a performance analysis of XML-RPC on similar hardware as ours [1] by Mark Allman, however he only explores actions which transport a lot of data, either as a request, a response or both, something UPnP should not be used for. The one action tested which doesn't need a lot of data uses a lot of calculation time (determine whether a given number is prime) which also makes it impossible to find the speed of the protocol (implementation) apart from the actual service provided. So although this may have been a very interesting comparison it should be considered future work because with the current information and measurements on XML-RPC we have no real ground for our comparison.

The hypothesis of section 6.4 that UPnP will perform poorer than dedicated RPC protocols can now be confirmed and denied. Yes UPnP performs much slower, it performs much slower when we take the current advances in hardware into account. No UPnP performs almost as fast, if we do a very naive comparison based on absolute times. The real conclusion we have to make is that there is much room for improvement of the speed of UPnP. The RPC protocols show that a similar system can be many times faster on current hardware. However this will require a completely different approach of implementing UPnP and will even require to alter the UPnP specification to not use XML for action communication for example. As already mentioned, these optimizations

[1] The MIPS used here are VAX MIPS, which seems to differ from the normal MIPS used in some applications

will be discussed in chapter 7.

6.5.4 Normal performance

In ideal circumstances the performance of UPnP is acceptable, a user interfacing with a UPnP services through a control point will usually not notice any difference between the UPnP application and a application running locally.

Problems can occur though when services or automated control points make frequent use of some specific service. So when eventually in some layer there need to be done many action calls for completing one user request or when for one user request many layers have to be traversed the delays will add up to an interval noticeable to the user. As already mentioned in chapter 4 and 5 it is necessary to determine a suitable grainedness of the interfaces and an appropriate layering of the functionality.

For example if we would have created another services for the DSN which would store a single block of data there would have been a huge increase in function calls (depending on the block size of course). Every DataStorageService than would have to call a new action for every block it would have to store, quickly causing a huge amount of communication and thus a significant and noticeable slow down.

Assume we use blocks of size 4KB. Storing a file of 128KB on one DataStorageService normally would take about 5 action calls. However when we have to store every block using an action, this results in 37 actions; an increase by a factor 7. Assuming all actions are comparable to our fastest measured actions (which they aren't, most are slower) this will give us a lower bound of 30ms in the current case compared to 222ms. The first is not very noticeable to a user however the second becomes already significant. Also compared to the size of the file we are storing now the time needed for transferring of the actual data becomes insignificant compared to the UPnP communication necessary for it.

Using the results found in this chapter and appendix A should be of help when making decisions about the fine- or coarse-grainedness of the service interfaces, about introducing new services for every distinguishable task and about the arrangement of these services in layers. Even when they perform more complicated tasks which take more time these test results can be used as a reference.

Another indication that we are gaining maximum performance for our setup is that the Δ (or "rest" in the tables) is almost 0. This implies that there is minimal interference of other applications running on the system and thus can make the UPnP application optimal use of the processor and network time.

6.5.5 Incorrect results?

Something that might seem as incorrect is that the empty action, which we expected to be the fastest action since it does not perform any real task, actually takes more time to complete than three of the other actions (updateFilePart, updateFileSize, getFileInfo). These actions though cannot be tested without running first some other actions since they need files to operate on which have to be created by the createFile action. Immediately after file creation these tests were executed on these created files without restarting the Directory Service or the test control point. Since all actions have much code in common, (all UPnP code used for the communications is completely equal, they only differ in the action performed and (the number of) parameters, therefor the java virtual machine can optimize the code easily and the code can be run from faster memory sources, RAM instead of a page file or cache instead of RAM.

Some further investigation shows that all performance parameters give better results except for the execution of the action itself, which was to be expected as the EmptyAction does not do anything while the other actions do perform a real task. This can be easily seen when comparing in appendix A the rows for the EmptyAction and the three actions (UpdateFilePart, UpdateFileSize and GetFileInfo). The only measurement which is slower is the ”performing request” measurement which includes the execution of the action. Since similar differences appear not only in the normal test situation but also during interference, we know the speed increase has to be in the code which is executed for every action (and is equal for every action). So this definitely points toward some speed optimizations occurring after many runs of the same parts of code.

6.5.6 Layered actions

When one action uses one or more other actions, like the StoreData action does, we assumed in section 6.4 that we could determine the total calculation time for the action by adding up all action response times of all actions called and add some extra time for the calculations necessary for the execution of the action. With the results we have now we can check this. Since the CreateFile action of the FileService does nothing but relay its arguments to the CreateFile action of the DirectoryService this should be the fastest possible layered action.

Comparing the ”request processing” time of the FileService, 10.7ms, with the total execution time of the action for the DirectoryService, 8.9ms, results in an overhead of about 1.8ms which we can attribute to the $process_{req}$ and $init_{resp}$ parameters. So we can indeed attribute the exact amount of total response time of the DirectoryService to $calc(a)$, namely the 8.9ms. We can safely assume the 1.8ms overhead are in the two other times since even the EmptyAction action has a ”process request” time of 1.8ms which we can only attribute to these two parameters because there is no further calculation done in the EmptyAction thus $calc(a)$ should be approximately 0 for the EmptyAction.

So for this simple layered action the formula presented in section 6.4 holds, however for the more complex action StoreData it is more difficult to show it holds. It can easily be shown it is a lower bound, since we know all actions performed by the StoreData actions and we have tested their total response times:

$$\begin{array}{ll}
total(”StoreBlocks”) = & 9.2 \\
total(”GetFileInfo”) = & 6.7 \\
total(”UpdateFileSize”) = & 5.9 \\
total(”UpdateFilePart”) = & 7.3+ \\
 & \text{---} \\
 & 29
\end{array}$$

So we can see the sum, 29ms, is less than the 40.3ms needed for completion of the StoreData action. It would be alarming if the actions would have performed faster than they already did when we tested them one by one.

Just as in the previous case we can probably attribute about 1.8ms to the $process_{req}$ and $init_{resp}$ parameters, which still leaves us with a gap of 9.5ms. In section 6.5.5 we gave an explanation why the total response times of the last three actions are even faster than the one of the EmptyAction. It is possible the explanation given there doesn’t hold in the StoreData case because the FileService performs several different requests and will be earlier preempted by the operating system scheduler, because it has to wait for the actions it calls to complete. This means the actions GetFileInfo, UpdateFileSize and UpdateFilePart ’under perform’ compared to the results we got when testing them (which were influenced by specific circumstances as discussed earlier).

As it turns out the simple formula presented in the expectations section 6.4 does not work always very well. However it is possible we can still get an accurate estimation of the total response time of a layered action on a specific platform without even measuring all actions called by the layered action and in our case avoiding the use of the suspicious response times of the separate actions. So this is exactly what we will do next.

We introduce the default action time DAT. We use this DAT as an approximation of the original action call time (for example the StoreData action) and as a basis for the time an action at least needs to complete (because of the UPnP API overhead). We can get a usable value for DAT by measuring the total time an EmptyAction takes on the platform the application under development is intended for. It gives an indication of the performance reachable by UPnP on that specific platform. Now we can split the total response time of StoreData in two parts as in figure 6.5.B. Next we introduce $calc'(a)$ which is the expected (or estimated) calculation time for an action on the machine the action is being executed on, thus without the time used by waiting for action call(s) to complete. We can also split the rest of the original calculation time of an action a into the total times of the separate actions called by a. Doing all this for the StoreData action results in the situation in figure 6.5.C.

For simplicity we assume that no actions are called in parallel, but that they are all called in sequence. The total times we need for these actions can again be obtained in two ways. We can use values measured in tests or we can estimate the values again using the default action time DAT and an estimation of the calculation time $calc(b)$ they need.

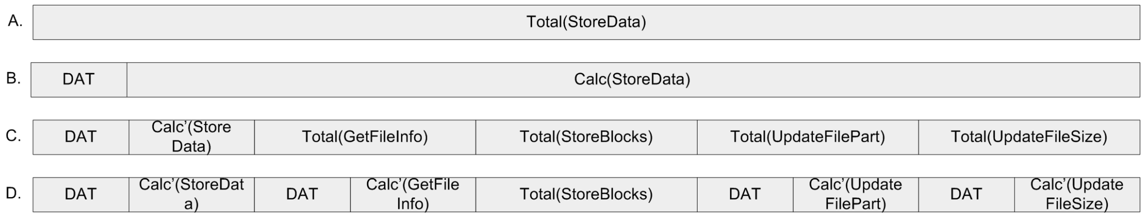

Figure 6.5: A visual representation of how we split the StoreData action in separate parts for an estimation of its total response time. To keep all text readable the block size does not represent the time an action takes. Moreover the blocks do not necessarily represent the order in which the actions related to the blocks occur.
This figure is only a visual representation of the way we divided the total time in figure A to smaller portions we can approximate in figure D.

So as a model for layered action we now have: if we call an action a which in turn calls actions $b_x, 0 \leq x < N$, with N the number of actions called by a, we can calculate an estimate of its total time using the exact known values for the b_x actions:

$$total(a) = DAT + calc'(a) + \sum_{x=0}^{N-1} total(b_x)$$

or we can also estimate the $total(b_x)$ times using $DAT + calc'(b_x)$ as an estimate:

$$total(a) = DAT + calc'(a) + N * DAT + \sum_{x=0}^{N-1} calc'(b_x)$$

Of course we can estimate some of the b_x actions while we use exact values for others. Using estimates for all action except for the StoreBlocks action results in the situation in figure 6.5.D.

Calculating the total time for the StoreData action according to the situations in figures 6.5.C and 6.5.D and applying the formula on the CreateFile action (using the measured values for the CreateFile action of the DirectoryService) results in:

$$\begin{aligned}
StoreData(fig.\ 6.5.C) : & \ DAT + calc'(StoreData) + total(GetFileInfo) + total(StoreBlocks) \\
& \quad + total(UpdateFileSize) + total(UpdateFilePart) \\
& = 8.4 + 1.5 + 9.2 + 6.7 + 5.9 + 7.3 = 39.0 \text{ (measured: 44.9)} \\
StoreData(fig.\ 6.5.D) : & \ DAT + calc'(StoreData) + total(StoreBlocks) + calc'(GetFileInfo) \\
& \quad + calc'(UpdateFilePart) + calc'(UpdateFileSize) + 3 * DAT \\
& = 8.4 + 1.5 + 9.2 + 0.5 + 0.5 + 0.2 + 3 * 8.4 = 45.5 \text{ (measured: 44.9)} \\
CreateFile : & \ DAT + calc'(CreateFile_{DirectoryService}) + total(CreateFile_{FileService}) \\
& = 8.4 + 0.1 + 8.9 = 17.4 \text{ (measured: 16.8)}
\end{aligned}$$

The first result is not very accurate, but this is mainly caused by the fact that we are using total response times for most actions which are influenced by the effects described in section 6.5.5. The other two calculations are however well within a 5% deviation of the measured values, which is even within the confidence intervals we calculated for our measurements (see beginning of section 6.5) and thus is a very acceptable result.

This shows this model can work and can be used quite easily in determining response times of actions without having to implement all separate actions and test them all during the design time. We based our model in this section on the expectations expressed in section 6.4, however we omitted the Δ mentioned there for now. The Δ only plays a significant role (compared to all the estimates we are doing) when there is interference from other processes. We will return to this issue in the next sections discussing the effects of interference on the performance of UPnP.

Combining the model for the performance of an action with an expected call-pattern (automatically and with default intervals or user initiated etc..) for an action we can now verify if a specific layering in a design is feasible. For example expecting to be able to call a StoreData action 100 times per second in our DSN would not be possible since it takes about 40 to 50 milliseconds to complete and thus we can only do about 20 of these actions in a best case scenario. If we would want to be able to call it 100 times per second we would have to redesign this part of the SOA or use multiple instances of the same service type to reach this performance. So this method gives us simple yet usable means for verifying the feasibility of a specific design with specific expectations and it can give, with some experimenting, guidance in designing a SOA. Moreover it can help in optimizing the performance of a design.

6.5.7 Normal performance vs. performance with interference

The tests with UDP or TCP traffic and the test with high CPU Load give the same differences and distribution between the actions as we saw with the normal (as ideal as possible) circumstances. However they performed, as expected, a lot slower, for many of the tests we saw especially the Δ performance parameter increase significantly.

The interesting part of the measurements is where the real performance loss happens, because this is not for all types of measurements the same. Especially the cases with some form of network traffic and the high CPU load differ significantly. We will start with the discussion of the CPU load interference and after that will continue with the network traffic interference.

Increased CPU load

In the tests with a high CPU load, on both the service and the control point node, we can only notice that the control point has to wait much longer for a response from the service than the actual time spent on an action by the service. This extra time (combined with the transmission time of the request and response) can be found in the Δ performance parameter (numbers 3 and 9 in figure 6.2). All other parts of the execution path work just as fast as in the normal situation, implying the UPnP processes are not preempted more or less than they were in the normal situation. We can notice the preemption of an action because if a UPnP process is preempted during a specific part of the action execution, we get an extraordinary long time measurement for that part of the action, usually between 5 and 10 milliseconds slower.

The cause of the slower performance is that the control point process can be preempted after it has sent its request and has to wait for its turn again before being able to receive the response, since the CPU load generator will start running its calculations and will continue until it blocks (which never happens) or it is preempted (for example because of an interrupt or it exhausts its quantum). For the service something similar happens; after completing an action request it will be preempted and has to wait for the CPU generator to be preempted by the operating system before it can handle a new request.

Since the time quantum every process in Linux receives is by default approximately 100ms this would mean that on the control point side and on the service side the maximum time the UPnP process has to wait is 100ms. So in total the maximum response time is 200ms plus the time needed for an action by the service and the control point, assuming the UPnP processes only get preempted twice in total (our measurements show this assumption holds with the exception of a few outliers). Real-time scheduling is a possible solution to avoid these big effects on performance (see chapter 7.

Luckily the Linux scheduler works better than this. Once the operating system handles an interrupt for the reception of data over the network interface it also runs the scheduler to determine which process is allowed to run next. It determines the best candidate process by (among other things) examining the time a process has still left of its current quantum. The process with the most time left will be allowed to run. Because the CPU load generator will be running most of the time it will use up its quantum quite fast and next time the UPnP process is waiting for a request it will have more chance to run because it probably has more time left of its quantum than the CPU generator currently has.

Also the Linux scheduler has a preference for I/O bound processes, like the UPnP processes for the control point and the service, over CPU bound processes, like the CPU load generator. Therefor the chances for the UPnP processes to get time to perform their task increase even more. Moreover the chance a UPnP process gets preempted is rather small since every action can be handled easily within the 100ms default quantum it receives.

So the total response time for an action lies approximately between x and $200 + x$ milliseconds, with x the total processing time necessary for the action on both control point and service side. However because of the specifics of the schedulers behavior we see in our test results that almost all response times are lower than $100 + x$ ms. Note its specific distribution in figure 6.6 where we have the total response time intervals depicted against the number of times a total response time occurs in that interval.

In this figure we used the EmptyAction action as an example, the other actions resulted in almost exactly similar graphs. For the normal case as expected almost all times fall into the first or second interval, with a few a little bit slower. For CPU we see that the times are not equally divided in the interval, not only are almost all values below $100 + x$ ms (with x approximately 8ms as we know from the tests without interference), we see two maximums, one at approximately 100ms and

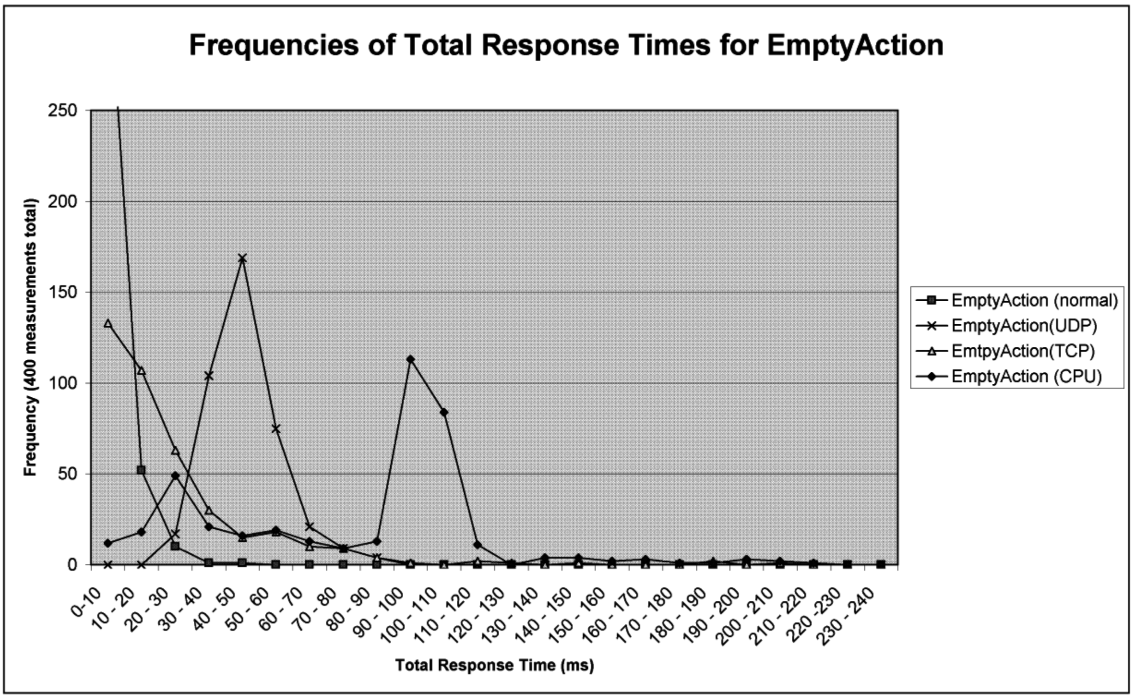

Figure 6.6: The frequency of total response times measured in intervals of 10 milliseconds. Note the two maximums for the CPU graph at 20-30 milliseconds and at 90-110 ms.

one at 25ms. These are probably caused by the behavior of the operating system scheduler and the call pattern we used (we called the actions immediately after each other without any delays).

We can now extend the model from the previous sections where we approximated the total response time of an action by using a default action time DAT and an estimation of the time needed to complete an action itself. We can now add the Δ to the equation we already mentioned in the expected performance model. For this Δ we can state that during high CPU loads on an operating system using Linux with a 2.4 kernel the following holds $0 \leq \Delta < q$ where q is the quantum time of the operating system running on, in our case $q = 100$ holds. We have to keep in mind that Δ is, according to our tests, almost half of the time very close to its maximum value as can been seen from the CPU graph in figure 6.6. Also for other operating systems it is difficult to state something similar because the scheduler of them probably operates different than the one for Linux 2.4.

This results in the following formula for a non-layered action:

$$total(a) = DAT + calc'(a) + \Delta \qquad\qquad \text{with } 0 \leq \Delta < 100$$

For layered actions we can extend the formula given in section 6.5.6:

$$total(a) = DAT + calc'(a) + \Delta + N * DAT + \sum_{x=0}^{N-1} calc'(b_x) + N * \Delta \qquad \text{with } 0 \leq \Delta < 100$$

We have to add one remark to this formula for layered actions. From our test results we can see that the statement that $0 \leq \Delta < 100$ holds, however the distribution of the total response times is different. There are more fast response times compared to the graph in figure 6.6, actually the two maximums are almost equal at about 80. This results in a better average performance than one would expect and is caused by the specific timing of actions and the scheduler algorithm.

Network traffic interference

Although the increased CPU load is of influence in case of network traffic interference we cannot contribute the significant slow down in communications to this alone.

For both network test cases we have TCP or UDP traffic (in both directions between the control point and the service) being generated by a java application. Since these generators also need to use the network interface, memory and CPU these resources are shared between the UPnP services/control points and the generators. When looking at the CPU load during the entire test period we can see it is almost continually at 100%, while running only the UPnP services/control points caused a much lower CPU load. We can also see huge amounts of data being sent between the two network nodes (running the control point and the service).

Here we also see the main difference between TCP and UDP traffic interference and our main reason for testing both separately. Contrary to UDP TCP uses control algorithms to avoid network congestion. In practice this means a TCP connection will not use all bandwidth pressing out other data communication (for example other TCP streams) over the same network (interface). This means the TCP socket is not sending data continuously and will regularly block. When the socket blocks, this also blocks the generator and the operating system scheduler will allow other processes like UPnP processes run.

The result of this is that there are much more process switches than when using the CPU load generator, because the TCP socket will block a process switch will occur much earlier than the 100ms of the quantum thus the UPnP processes have more opportunities to run and perform their task (which only takes a short time allowing again for a fast process switch). The effect of the congestion control can be seen clearly in the TCP traffic graph in figure 6.7 where we see the throughput rise and drop repeatedly; rise when there is no congestion detected, but drop to a lower rate (or a start from 0 again) when there is some data congestion detected.

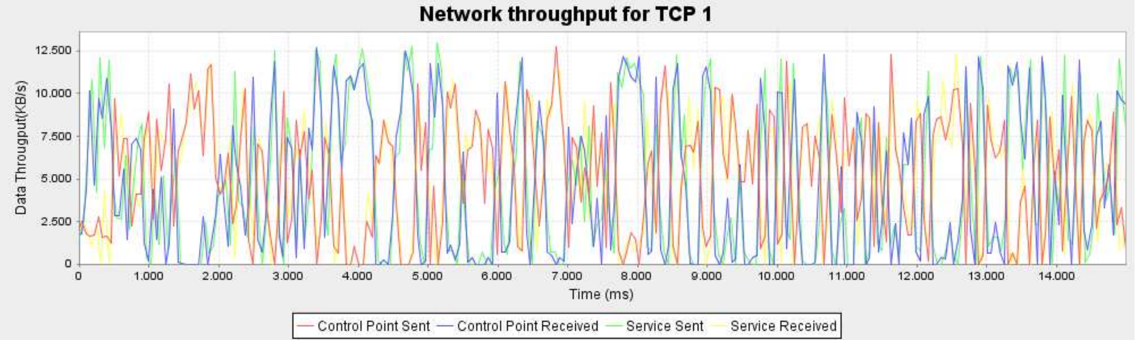

Figure 6.7: The TCP generators throughput for sending and receiving data from control point to service and service to control point. These are the measurements during the EmptyAction tests (other tests produced similar results). The amount of data sent by the control point should equal the amount received by the service and vice-versa (which is the case since we used TCP which guarantees delivery).

As we expected from the previous reasoning and as we can see in figure 6.6 introducing TCP traffic causes a nice gradual degradation of the performance of UPnP because of this congestion control which prevents the generators from using the entire bandwidth. The degradation is not dramatically as we can see by comparing the curve of TCP with the curve of the normal situation in the graph. The TCP curve is a more ’flat’ version of the normal curve. We still have most actions completing within 10ms, however much less than in the normal situation.

We actually tested with the TCP traffic generators sending an unrestricted amount of data, thus (in total) we used the entire available bandwidth. So we tested the worst case scenario of continuous TCP streams at maximum throughput. In real life this will not be the case continuously, so the performance will usually be better than the results of the TCP traffic interference suggest.

What happens with UDP traffic is that the UDP traffic generators start sending out UDP traffic to the other node as fast as they can (as can be seen in figure 6.8, especially when compared with the TCP graph), because of packet-loss on the network most of the data is never received by the other node. The side effect of this unlimited sending of data is that all resources are continuously occupied by the UDP traffic generators and, unlike the TCP sockets, UDP sockets do not block while sending offering less moments for the scheduler to preempt the generator.

Luckily the continuous reception of data (from the UDP traffic which is received and from the UPnP requests) generates interrupts which allow the scheduler to preempt the UDP traffic generator and switch to another thread. Often the data received will be for the receiving process of the UDP traffic generator so the UPnP processes will not start, however when a UPnP request has been received recently the UPnP processes can handle it (depending again on the quantum left for all processes as explained in section 6.5.7 on CPU load interference). These process switches will also occur more often than the quantum of 100ms because the interrupts will occur much more than once during 100ms. Moreover the interrupts will occur more often than when only the UPnP processes were using the network interface, offering more opportunities for the UPnP process to start running again which allows for faster response times of an action than during the CPU load interference.

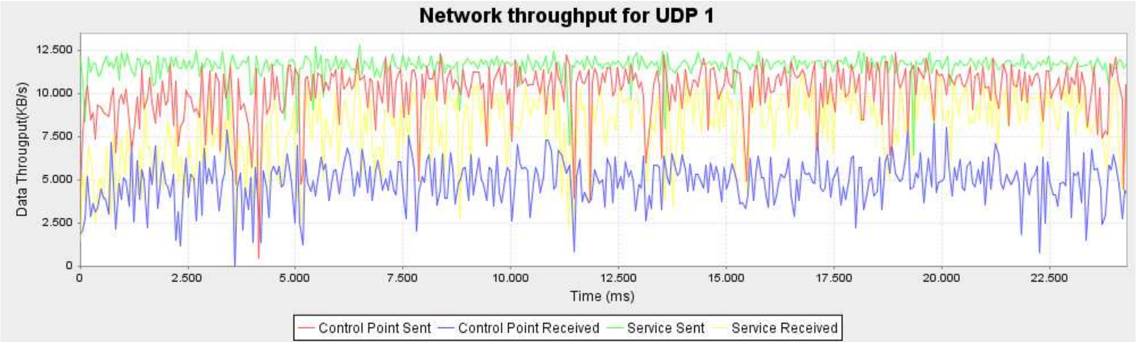

Figure 6.8: The UDP generators throughput for sending and receiving data from control point to service and service to control point. These are the measurements during the EmptyAction tests (other tests produced similar results). The amount of data sent by the control point should equal the amount received by the service and vice-versa (which is not the case since we flooded the network).

When we look at the spread of the extra time spent completing an action request we can see (in appendix A for both, TCP and UDP interference, that a large part of the time still is spent ”Rest” or Δ as we already explained for the CPU load interference. However contrary to the CPU load interference we see now that all the average times of the measured parts of the action execution are slower than in the normal case. The cause for this is very simple: sometimes the UPnP processes get preempted somewhere during the execution of a part of an action because of the big number of preemptions caused by all the interrupts generated by the big amount of network traffic.

We examined the log files of our tests and indeed discovered that only a few out of the 500 measurements provided slower results than we found with the tests without interference. If there are only a few measurements which give different results we can only explain this by some special event, like a preemption of the UPnP process right during that measurement, which supports our theory from the previous paragraph.

The effects of UDP are bigger than the effects of TCP because the only way for the UDP generator to be "stopped" is by preemption because of interrupts or because it has used up its quantum. The TCP generator will also be preempted because of these reasons, but additionally it is preempted because its TCP socket blocks because of the TCP traffic control. The result of this blocking is that the TCP generator is much less "greedy" for processor time than the UDP generator since it voluntarily releases control and then does not immediately start running again because the generator has to wait for the TCP socket to un-block. For the socket to un-block it first has to empty its buffer which can take a relative long time for the TCP socket compared to the UDP socket because it has to wait for all the data to be received and confirmed by the other side.

We can now extend our model from the previous sections with information on the effects of TCP and UDP traffic interference. We will again use the Δ performance parameter to capture the effects of this type of interference. Although sometimes the effect of the interference shows up in other performance parameters than this one it does not really matter for our purpose. We just want to capture the effect of TCP and UDP traffic interference on the total response time of an action and since the effects the interference has on all performance parameters have the same cause, namely processes preemption and switches caused by interrupts or blocking of processes, we can safely accumulate the effects in one single performance parameter.

If we look at the graph for the UDP interference in figure 6.6 we see that this graph looks quite similar to a graph of a normal distribution. Moreover if we look at the graph for TCP interference we can see in figure 6.6 that it actually is a graph with the same shape as the normal case, however with a (much) flatter slope. When we look a little closer at the graphs of both the normal situation and the TCP interference situation we see that they too look quite similar to the right-hand side of a normal distribution.

To test if this is true we will use the same formula for the total response time as we used in the section on CPU load interference (section 6.5.7), but we will be using a different Δ for which we will test a function which follows a normal distribution.

First for UDP interference traffic we see in the graph of figure 6.6 that all the UDP traffic is centered around the averages of our measurements for UDP traffic (40 to 50 milliseconds). Further we find that the standard deviation is for almost all UDP interference tests 10 milliseconds, only for a few actions with more data communication (like the GetListing action) we know that they are approximately 15 milliseconds.

Using the formula from section 6.5.7 we find that the formula in equation 6.1 describes the behavior of the UPnP actions quite accurately during UDP traffic interference. The mean (μ) used is between 30 and 40 milliseconds instead of 40 and 50 because we know a typical action requires a little less than 10 milliseconds to complete, the standard deviation (σ) is exactly equal to the values we measured during our tests. In figure 6.5.7 we see an example of how accurately we can match the real behavior of Δ with this model for the EmptyAction. We calculated the graph of the normal distribution using $\mu = 35, \sigma = 10$ and we created the graph for the Δ of the EmptyAction by subtracting 8.5ms ($DAT + calc'(a)$) from the total response time of the EmptyAction for UDP traffic interference.

(6.1) $total(a) = DAT + calc'(a) + \Delta$
with Δ a function with its results distributed according to $N(\mu, \sigma)$
and $30 \leq \mu < 40, 10 \leq \sigma < 15$

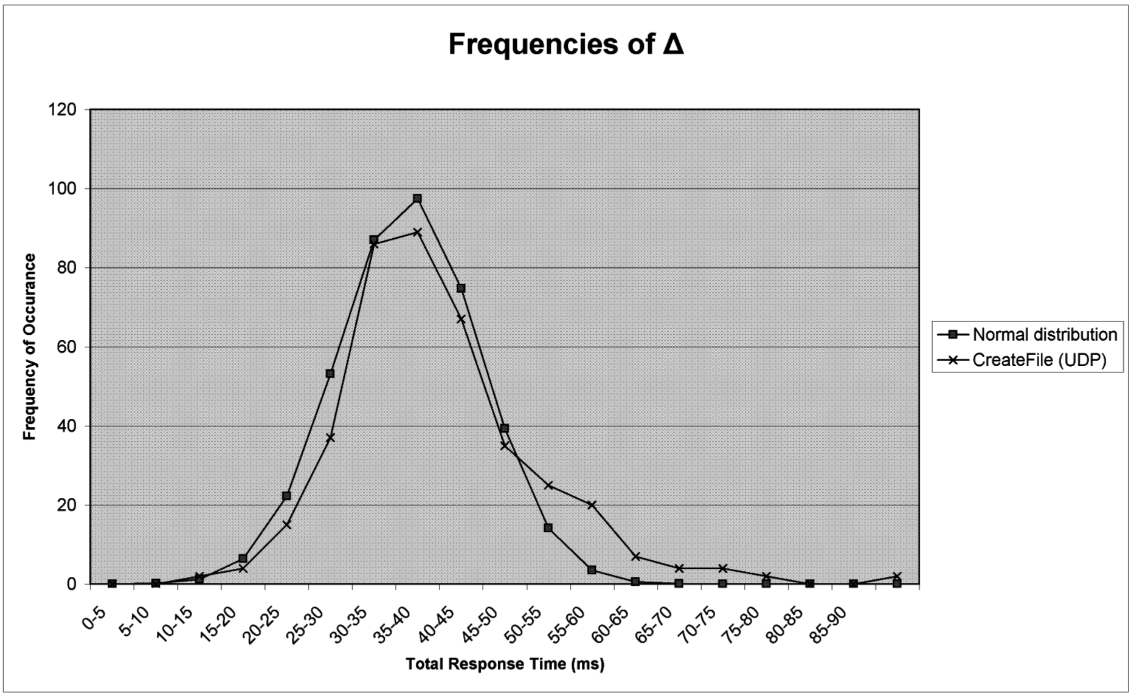

Figure 6.9: The graph of the frequencies in which values of Δ occur for the (non-layered) CreateFile action during UDP traffic interference combined with the graph of an estimation of these frequencies using a normal distribution with $\mu = 37$ and $\sigma = 8$.

Next we want to try if we can do something similar for the normal situation and the situation with TCP traffic interference. One problem we have to solve first is that we only have one side (the right side) of the normal distribution, because we cannot have negative values for Δ (a negative value would imply that by adding an interfering process we would improve the performance of UPnP). Also we will not be able to use the mean (of the total response times or the "Rest" measurement) we measured since this is nowhere near the place we expect the mean of the normal distribution. Looking at the graph of figure 6.6 we notice that the mean gives an inaccurate generalization of the test results. We would expect the mean of a normal distribution, with a right-hand side similar to the measured results, to be somewhere between 0 and 1 milliseconds.

To work around the fact that we only are using half of the normal distribution we actually add the chance that a negative time $-x$ occurs to the chance that the positive x occurs. Looking at the "Rest" or Δ values for the normal situation we see that they usually are about 1 milliseconds which looks as a good average to use for our experimentation. So using 1 millisecond as a mean and a modified normal distribution (N')with only a right-hand part of the distribution, which is the sum of the right- and the left-hand chances, instead of the normal distribution (N) we can try some comparison graphs for the normal and the TCP interference situation using the formula:

(6.2) $total(a) = DAT + calc'(a) + \Delta$
with Δ a function with its results distributed according to $N'(\mu, \sigma)$with $\mu = 1$

To avoid any confusion because some actions have longer total response times we again use a default value for $DAT + calc'(a)$ which is in this case different for every action, but can be calculated by taking the total response time in the normal situation and subtracting 1 millisecond which we attribute (on average) to Δ. With this we can draw the graph for an action with only the Δ we measured for that action to compare it with the graph we create using the new estimation for Δ. Some experimentation shows us that we can find values for σ for both the normal (an example in figure 6.5.7) and the TCP interference (an example in figure 6.5.7 situation which describe a distribution for Δ.

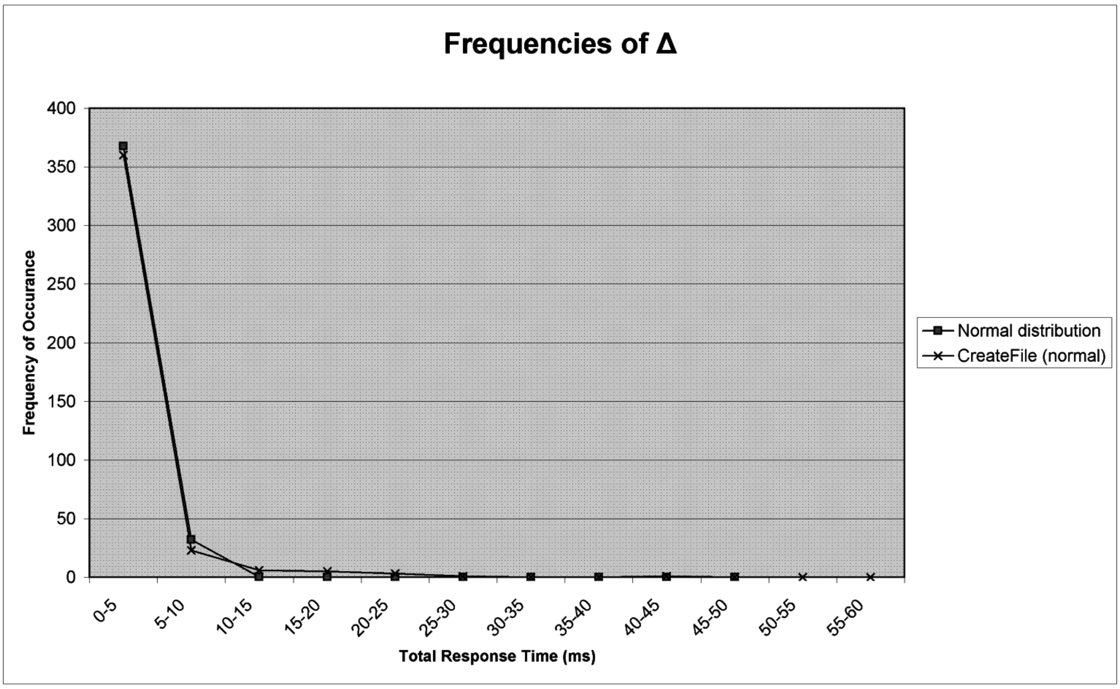

Figure 6.10: The graph of the frequencies in which values of Δ occur for the (non-layered) CreateFile action in the normal situation combined with the graph of an estimation of these frequencies using the adapted normal distribution (N') with $\mu = 1$ and $\sigma = 3$.

The values for σ and μ we found to give the most accurate approximation of our tests results can be found in table 6.5.7. In all cases we were able to create a normal distribution very similar to the measured performance indicating that we can indeed describe the interference effects of other processes using a normal distribution for Δ. What we can see in the table is that there is no direct link between the mean of our measurements and the mean we use for the normal distribution for the TCP interference and the normal measurements. The cause of this is that the results of our measurements are all based on half of the normal distribution because it is cut off at a Δ of 0. So the mean we calculate from those test results is the mean of a continuously decreasing function instead of a bell-shaped function like the normal distribution.

That we can find some influence of other running processes even in the normal situation is not very strange. Even when we are testing in the normal situation other processes are running on

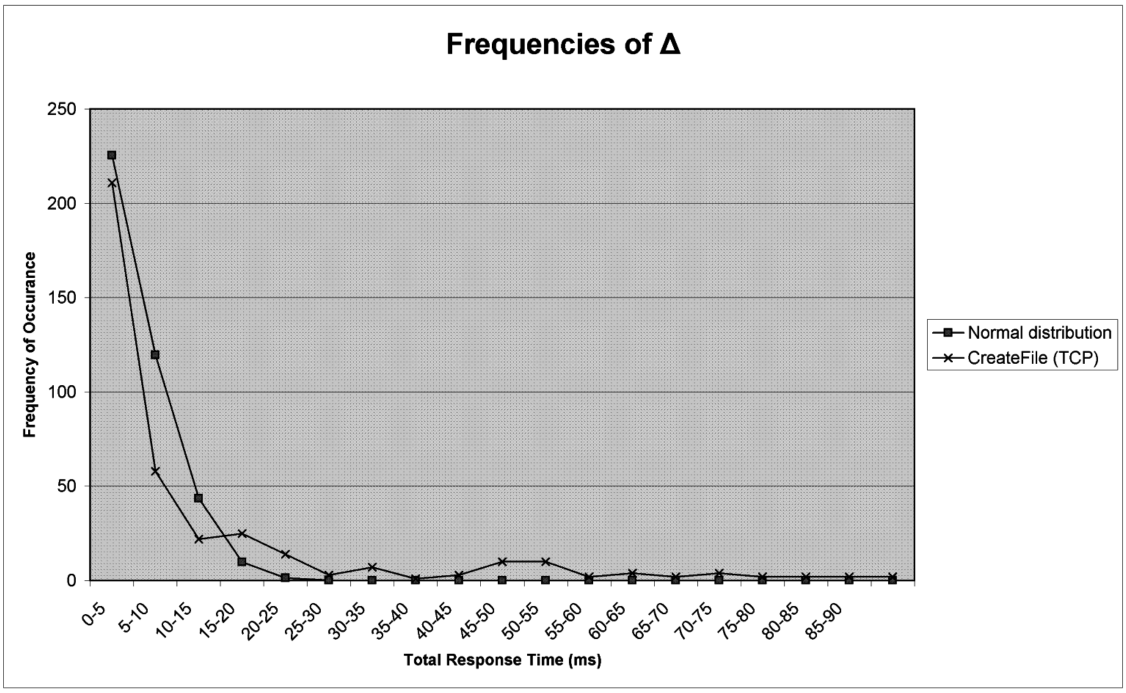

Figure 6.11: The graph of the frequencies in which values of Δ occur for the (non-layered) CreateFile action during TCP traffic interference combined with the graph of an estimation of these frequencies using the adapted normal distribution (N') with $\mu = 1$ and $\sigma = 7$.

Situation	μ (ms)	σ (ms)
Normal Situation	1	3
TCP Traffic Interference	1	$6 \leq \sigma < 8$ [7]
UDP Traffic Interference	$30 \leq \mu < 40$ [35]	$10 \leq \sigma < 15$ [10]

Table 6.3: Values for μ and σ for the normal distribution of equation 6.1 and 6.2 under the possible different circumstances. Between square brackets we give the value which gives in most cases the best result.

the nodes (operating system daemons etc..) which occasionally will cause the UPnP processes to be preempted. However because those other processes are not very active the influence of the preemption is very small and UPnP processes only get preempted occasionally and for short periods (as can be seen from the frequency graph).

Introducing a TCP traffic generating process which runs continuously adds a significant load to a node and fierce competition for processing and resource time. As already explained this leads to a significant increase of the number of preemptions of the UPnP processes in all locations (during the execution of parts of an action, but specifically when the UPnP processes are blocked waiting for requests or responses received via the network interface).

The reason that the graph of the response times frequencies is shifted to the right for the UDP traffic interference is the already explained fact that the UDP generator process can always start running again (contrary to the TCP generator process which is blocked regularly because of the traffic control algorithms). This causes the UDP generator to be almost always running when a request or response arrives, therefor UPnP processes will have to wait for the UDP generator to be preempted by the operating system scheduler before they can process the received data.

So we conclude that we can describe the influence of other processes which are running simultaneously with the UPnP processes can be described best with a normal distribution for Δ using the formula presented in equations 6.1 and 6.2. For the the μ and σ parameters we suggested the values from table 6.5.7 for the various different situations. Only with a high CPU load and no additional communication the behavior of UPnP actions cannot be explained using the normal distribution for Δ.

There are two downsides on these two formulas, the first is that the mean and the standard deviation we propose here are probably specific for our operating system. We suspect our results depend on the quantum size (all actions complete, on average, just under half the quantum size), the scheduling algorithm used and the priorities assigned to the processes. Further research could be used to find a more exact correlation between these parameters and the resulting distribution. On the other hand, the intervals for the mean and standard deviation hold for all normal actions we tested. So if one wants to use UPnP with different parameters than we tested (100ms quantum size, default priority for all processes and Linux 2.4 scheduler) it is possible to only test one action under the different circumstances (no interference or TCP/UDP traffic interference) and use the results from that test to estimate the response times for all other actions one wants to implement.

A second problem with this result is that we cannot extend it to estimate the performance of the layered actions, like we were able to for the CPU interference and the normal situation. For example the results for the StoreData action during UDP traffic interference do not show a normal distribution any more. They just show the results spread over a rather large interval (between 200 and 500 milliseconds) with two local maximums at 260 and 360 milliseconds. Some more investigation into the specific locations (which service, during outgoing or incoming traffic etc.) where the delays in performing an action occur during interference traffic may be able to explain the big difference in response time we are experiencing. The fact that the service executing an action is calling itself an action of another service is probably of effect here and causes different behavior than in the normal case.

Interference effects summarized and possible solutions

In short we can say all types of interference tested have a big effect on the performance of UPnP. However we tested all types of interference on the maximum possible interference. In a normal environment the extremes tested here will not occur that often. For example when a machine hosting a UPnP service has a continuous processor load of 100% there should be added more machines to handle all the tasks that machine is fulfilling. If the load is mainly caused by the UPnP service, it might be a good idea to add some more services of the same type to the network to distribute the load over multiple nodes, if this is possible with the specific service type.

Also if it can be foreseen that a service will generate so much load or that handling of an action for that one service will take very long, it is a good idea to revisit the design of the service and its actions and split it into smaller services and/or actions. If there is no other way a partial solution is to grant the main UPnP thread a higher priority than the other threads (and applications) running. This should give the main thread, which handles the action requests, an advantage in competing for processor time and will at least make the behavior of the service more predictable so statements about the expected (slower) performance can be made more accurately. Not surprisingly UPnP services and control points are effected significantly by a (continued) high CPU load, as are most applications.

When there is much UDP traffic on the network interface the performance of UPnP suffers significantly. If the cause of the UDP traffic is external to the UPnP services running, the problem might be solved by placing the UPnP services on a separate machine which doesn't have that much UDP traffic although any application creating so much UDP traffic should be thoroughly examined for flaws.

It is also possible that there are so many UPnP services on the network that they together generate a huge amount of UDP (broadcast) traffic for announcing their presence and sending a keep alive signal every now and then. The only way around this is reducing the number of UPnP services on the network which again can collide with the entire design of an application. For example the UPnP service for storing blocks introduced in section 6.5.4 might not only be a bad idea for reasons given there, there would be also the need for a huge amount of these services introducing a huge amount of UDP traffic received by all control points on the network.

In the UPnP documentation is suggested that the fix for such a problem should be to increase the keep alive time so the services broadcast their keep alive signals with longer intervals. Unfortunately this is for many applications not a solutions as they need to have an up-to-date view of the presence of services like in our case we don't want to be waiting for a time out when we are trying to store a block to an offline service. Moreover when a control point does a search for all available services it will still get replies from all services generating a huge amount of UDP traffic to the control point. The best way to circumvent these problems is to fix the UPnP protocol. Suggestions on how to do this can be found in 7.

Finally a lot of TCP traffic does not have to be a very big problem. In our DSN implementation the FileService for example will not only be handling a significant amount of UPnP action request, but for every request there will also be a certain amount of TCP data traffic. Of course if we detect that a FileService cannot handle the load anymore we could always add a second or third FileService to help out. In general though we can say it will take much longer for UPnP to experience a real performance impact due to high TCP traffic than it takes for the other two cases, actually the maximum performance degradation we measured under maximum TCP traffic wasn't that bad at all and showed a gradual degradation of the performance of a service instead of a sudden leap to much slower response times. Moreover high TCP traffic on a machine will probably also result in a high CPU load suggesting adding services of the same type to handle the workload.

6.5.8 Adding services for improved performance

If a specific service type in the application has trouble handling its current load for some reason, it can be useful, as we already suggested in section 6.5.7, to add additional services of the same type so the load can be distributed over these services. However the results of this depend on a few simple factors.

Prerequisites for the service of which we want to start multiple instances are:

- The service should not have an internal state which cannot be derived from external sources. If the service type does have an internal state all instances of the service type will have to synchronize their state, unless of course this state can be derived from other external sources like the state of other services (of which they can be noticed by events). Avoiding that a service maintains a state and eventually having multiple instances of the same service type with a synchronized state is discussed in chapter 8 where we discuss replication of services.
- Control points which use this type of service should not all choose the same service. This can be accomplished by letting control points choose the service they use at random.

If these prerequisites are met it is possible to just start extra instances of the service we want to reduce the load and automatically the load should be spread over both (or even more) services. This however does not guarantee that the response time of the service improves.

For example if control points have to connect to both services through a network router this network router can still be a bottle neck because it cannot handle the total traffic. Placing the extra service closer (in the network topology that is) to the control points which experience slow response times can solve the problem more adequate. Although this can also just move the problem from the communication between the control point and the service to the communication between the original service and the services it uses if those are behind the router which was the bottleneck in the network.

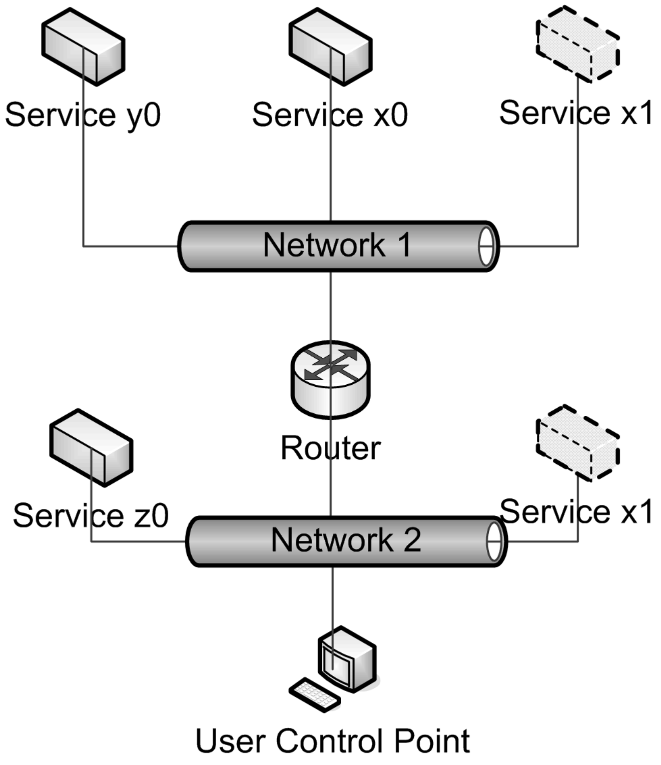

Figure 6.12: In this figure we assume that the user uses a service of type x, which can use a service of type y and a service of type z. We also assume we cannot move the current services.

As an example we use figure 6.5.8. Here we assume that all services with number 0 are working and have a fixed location. But service $x0$ has poor response times, this can have many different causes. However if we assume we cannot add multiple y and z services without extra overhead (something we can do with x) we have only two options to consider. Placing $x1$ in network 1 or placing $x1$ in network 2.

A reason for placing $x1$ in network 2 can be that many users are connecting to x services from network 1 through the router. However the router cannot handle that much extra data traffic (there probably is already data traffic between network 1 and 2 caused by other applications or services) and thus causes the slow response times. However if $x1$ makes extensive use of y services this still does not fix the problem, because now the calls to service $y0$ are going to be very slow. If on the other hand x services make extensive use of z services this placement of $x1$ delivers double profit because communication between $x1$ and $z0$ does not have to travel to the router anymore (in contrast with the communication between $x0$ and $z0$).

There are many more cases in which we can argue to place $x1$ in network 1 or network 2, depending on the actual situation. However, there are also some situations in which we cannot create any speedup. For example if $y0$ or $z0$ are the main cause of the slow response times, because they have slow response times. A more complex situation is when we have to place $x1$ in network 2 to avoid to much data traffic through the router, but this introduces a similar but new amount of data traffic through the router between $x1$ and $y0$ which effectively does not solve the problem.

So starting additional services of the same type can solve slow response times and high loads of that service. However adding the service in an arbitrary position in the network will not always work, a careful analysis of the problem has to be made to determine where the actual problem is in the network (it does not necessarily have to be the service itself which causes the slow response times). Moreover not all service are suitable for replication without any additional measures like a synchronization of states of the different services. So although in many situations we can improve performance by adding an extra service to the network there sometimes is no direct and easy solution as demonstrated in the example; we sometimes just move the problem from one service to another.

6.5.9 Throughput predictions

With the current implementation of our UPnP API it is not possible for two actions to be executing simultaneously because there is only one thread running for handling action requests. This fact makes it easy for us to predict the workload a service can handle (or the throughput a service can reach); we can just calculate the number of actions/second by the time it takes for one action to complete.

There is just one gotcha in there, one service offers usually more than one action. We can only say something of the throughput of a service, so if we take a certain interval in which only the fastest action is executed by the service, we would find a throughput which shows us a to positive view of the possible throughput. Vice-versa, if we choose another interval with only the slowest action, we get a too negative view of the throughput. These are actually approximately the lower- and upper-bounds of the throughput, in reality it will be somewhere in between depending on the frequency every action of a service is used. For example a read action will probably occur more often than a write action, thus the throughput will be more dependent on the time a read action takes than the time a write action takes.

We can also, with help of our tests, only calculate the throughput a service can handle in the four tested cases. As might be obvious the worst case scenario is the upper lower under 100% CPU load. The best case scenario is upper bound in the normal scenario. Some examples for our DSN for the lower and upper bounds in the best case (normal) scenario:

$$\begin{aligned} DirectoryService(lowerbound): \quad & \frac{1}{total("getListing")} = \frac{1}{0.0108} = \quad 96\ actions/second \\ DirectoryService(upperbound): \quad & \frac{1}{total("updateFileSize")} = \frac{1}{0.0059} = \quad 170\ actions/second \end{aligned}$$

One mistake which can be made here is to assume that these values will always hold, but they won't even hold when we request a bigger listing. In that case the parameters, of the performance model, values will be bigger and the calculation time for getListing will be longer resulting in a significant slower response time and thus in a lower number of actions/second.

If we take the lower and upper bounds for the worst case scenario we can see to which crawl a

SOA based on UPnP services can be slowed down if not designed correctly:

$$
\begin{aligned}
DirectoryService(lowerbound): \quad & \frac{1}{total("updateFilePart")} = \frac{1}{0.0756} = & 13.2\ actions/second \\
DirectoryService(upperbound): \quad & \frac{1}{total("getFileInfo")} = \frac{1}{0.0750} = & 13.3\ actions/second
\end{aligned}
$$

The result is not only a service which can handle only 13 actions/second instead of at least 100. The difference between the lower and upper bound has disappeared because all actions are performed almost equally slow because of the high CPU load.

To get back to the issue of one thread per device for handling action requests. This is something that can be fixed in the UPnP API, but the effects this will have are difficult to predict for the moment. We could also fix our action handling for actions which take a long time. For those actions we can initiate a new thread upon receiving an action request and return an action response with a message saying the action will be executed after which the result will be returned with the use of a callback action. In this scenario the workload which can be handled by one UPnP service will probably depend mainly on the calculation time necessary for an action and its behavior when competing with other similar actions for processor time. Thus it will be different for every service and even for different actions.

6.6 Conclusion

The results obtained and presented in this chapter are not the result of one single successful test run. These are results obtained after several iterations improving the test design and even improving the implementation of the services to iron out a few 'bugs' slowing down the communications significantly (see also section 6.5.1 and chapter 7. Also the results in this chapter have been gathered on a network using only a single switch and no routers between the different nodes. When traffic has to travel through nodes and multiple hubs or switches the results may be effected, probably not for the better.

We were able to reach the goals we set out at the beginning of this chapter for a big part.For almost all situations tested we have been able to determine the behavior of UPnP and explore and understand it. From this understanding and the absolute performance numbers we have been able to make improvements in the toolkit used and we will be suggesting many possible improvements in chapter 7 based on our performance tests. Finally we have been able to find a model which can be used to predict the performance of a UPnP API under different circumstances and with a few simple parameters at its basis.

The performance model can be of significant use to people developing SOAs based on UPnP, since UPnP's biggest limitation is not its usability, but its performance. For example we found that avoiding high CPU load on nodes running services requires splitting of services or running multiple instances of one type of service. On the other hand avoiding of to much UDP traffic (and also avoiding too many event subscribers might come into play here) requires merging different services or at least making sure there are less services. These are obviously contradictory demands which have to be met, so as we already stated in the chapter on the UPnP framework and in the chapter on DSN design we can repeat again that it is not trivial to design a good working Service Oriented Architecture which also scales well. It almost always will come down to some experimenting in which initially the performance model can be used to give direction at the design and the experimenting.

Something else we can support with our test results is that it is of great importance to consider very carefully which actions and/or services should be split up in new actions or services introducing one or more additional communications necessary for completing the same action. In section 6.5.6 we showed we can do predictions about the performance of an actions (also during specific types of interference), these results can also be used to calculate the impact of splitting one action into multiple actions on the performance.

As a last note it may be interesting to compare the different UPnP API implementations and judge them on their speed and ease of use for the developer. We actually did a preliminary and very simple comparison test between the Cyberlink and the Intel UPnP APIs by comparing the performance of the getStatus action of the Intel API based "light bulb.exe" device (provided with the Intel toolkit) with the EmptyAction action we implemented for the Cyberlink API. After all improvements suggested in 6.5.1 the results were (on different hardware than the other tests) an average of 3.5ms for the Intel API and an average of 3.8ms for the Cyberlink API.

We conclude from this that the Intel API probably has similar performance problems as the Cyberlink API, although before we can be certain about this conclusion more tests will have to be done. This also supports our opinion that UPnP in its current specification can only be a little bit faster than the version we used. For example by using a C(++) based API instead of the Java version. However more improvements in performance will have to be found in a performance oriented implementation of the API and will probably be at cost of usability and readability of the source code of the API. Also a real improvement in speed would need adaptations to the protocol and would have to drop XML as the method for action communication. Possible improvements for this API and probably also for other APIs (like Intel's) and the UPnP protocol itself are proposed in the next chapter (chapter 7).

Chapter 7

Evaluation and improvements

In this chapter we will discuss some weak points of UPnP and, if possible, propose a better solution for it.We will start in the first section with discussing the weaknesses and improvements for UPnP in general. In the second section we will come to specific UPnP API issues which may be of help for people implementing their own API or for adapting an existing UPnP API to be more usable/faster. Some suggestions are for performance improvements, while others are for the usability of UPnP or the API. When referring to the current version of UPnP we are referring to version 1.0.1 from December 2003.

7.1 UPnP in general

7.1.1 UPnP specification unclarity's

There are some unclarity's in the UPnP documentation which leave quite a lot room for implementers of a UPnP API. Usually this is not a problem and might even be a good thing, however for some parts of the UPnP standard it makes a significant difference how it is implemented. The ones we came across will be discussed here.

Multiple simultaneous actions

The UPnP API we used can handle actions only sequentially. This means that if an action x of a service in device A calls an action y of a service which also resides in the same device the device is still waiting for action x to complete and thus cannot handle action y which effectively causes a deadlock situation (left part of figure 7.1).

In the UPnP specification there is no mention of the way a device should handle actions (concurrently or sequential), which can have a significant effect on its behavior and on its performance (handling actions concurrently can make better use of processor time). In the UPnP specification one would expect something like this to be mentioned, so designers of UPnP services and control points know which behavior they can expect.

Further it seems logical to us that a device (or service), which in fact is a server-like application, should be able to handle multiple actions concurrently. A device, and the services within the device, should handle actions without blocking other actions and the execution of an action should not block other actions from being executed. This to avoid deadlock situations like in our example.

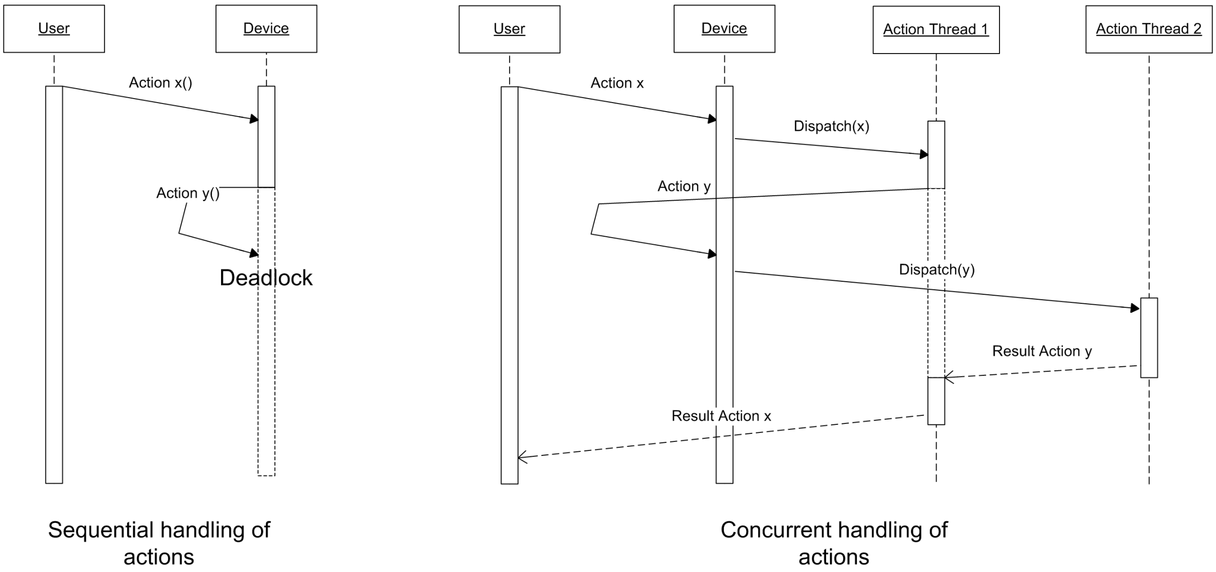

Figure 7.1: Execution of action x which in turn calls action y on the same device for a device handling actions sequential and a device handling actions concurrently using threads. The dotted blocks indicate the execution of the object is blocked (i.e. waiting for the result of an action).

An example of the way how a device should handle actions is given in the right part figure 7.1, here the separate actions each are executed in their own threads instead of in the main device thread which would block the reception of the second action request (as can be seen in the left part of the figure).

Event handling

Receiving events by a control point is often implemented in a way that the event is received, the control is given to some event handling code and only after that the API returns an acknowledgment of the event to the service. It would be better to first acknowledge the event reception and then let the event be handled by the rest of the application. A visual representation of this was already given in chapter 4 in figure 4.5. Using the latter approach we avoid a significant delay which could occur while a service is sending an event, because it has to wait every time for the control point to complete its event handling code completely. Nowhere in the UPnP documentation is there a note on the significant difference this small implementation detail can have on the performance of a UPnP service with a large number of subscribers. This is something that could have been noted and would have avoided a bad performing eventing system. A work-around for this is suggested in chapter 4.3.3.

7.1.2 UPnP changes/additions

We have encountered several points on which we think UPnP can be improved, mainly to make it more usable for our SOA oriented purposes, something UPnP initially wasn't designed for. So it is no surprise we found some problems with UPnP which could have been fixed if they were considered while designing UPnP. Changes we will suggest in this section can be to make UPnP easier in use, perform faster or scale better.

XML communication

For the sake of speed it would be beneficial to not use XML as the syntax for sending control actions and responses. Not only are the messages bigger than in a binary protocol, but it is also necessary to use a XML parser at the receiving side and some way to create the XML structure on the sending side. The parsing of the XML tree will usually take longer than just reading out a binary record. Although this would be better for a fast protocol, it would compromise the interoperability of the protocol. See also 7.2.1 for some more details on XML parsing in the UPnP toolkit.

To keep interoperability and also offer a faster way of communicating then is possible with XML an option would be to offer a standard service with an action which can be used to determine if a specified binary protocol is supported by the device. If so, the control point and the device could switch from the XML version to the binary version and speed up communication. If the binary protocol is not supported or incompatible (for example because of a different hardware platform which stores integers big-endian instead of little-endian or vice-versa) they keep using the XML format.

If UPnP is to use XML despite the performance penalty paid it should at least use it to its full potential. Currently UPnP supports only basic types, although XML is very well equipped for encoding complex types like arrays and records. If one needs an array parameter for an action it is up to the developer to encode the array in a string parameter (using for example XML). This should not be necessary. SOAP [19], which is the basis for the UPnP communication syntax, has support for it and an implementing it is rather straightforward. Even the much simpler and less bloated (compared to SOAP) XML-RPC [25] protocol offers support for arrays and structured types, XML-RPC would also be a good replacement to simplify (and reduce the overhead involved in) the action communication syntax.

Communication protocols

From the point of view of optimizing for speed it is a lot of work to setup an entire TCP connection to send about 1000 bytes of data over and receive a similar sized result back. Since we often send this kind of amounts of data back and forth it using TCP is actually a waste of time and only beneficial for the delivery guarantee it gives.

It would be beneficial to switch to the UDP protocol with a simple form of resending and reception confirmation. Something similar has already been demonstrated to work for the eventing protocol used by UPnP in [12]. Another option would be to use T/TCP (TCP for transactions [3]) though there are many discussion on the Internet about its security and it is doubted that it will ever be usable.

Finally it is possible to use persistent connections. Meaning that if a control point connects to a device once, the connection is kept open after wards for future requests to that device. An advantage is that after a while a control point has TCP connections to all services it uses, a disadvantage is that some controlpoints can use a huge amount of services and a service can be used by a huge amount of control points. This will result in a huge amount of open TCP connections using up resources on both sides of the TCP connection. This could however give the best performance gain since we can completely skip the socket creation and connecting part when requesting a new action, whilst with UDP or T/TCP we would still have to create a new socket every time a control point uses an action.

Eventing

Instead of using the current UPnP eventing protocol (GENA) we prefer to adopt the UDP protocol based on the GENA protocol suggested in [12]. This is beneficial for the performance when sending large amounts of events, which could occur on a regular basis when designing SOAs based on UPnP. For example in the DSN we can let the FileService notify users of the available space in the DSN by use of an event. However when there are many users using the FileService there are probably many changes in the available space of the DSN and every change is evented to all users again. This causes a significant amount of communication which in the current situation can be a pretty slow processes as can be seen from the tests done in [12]. To keep a better up-to-date view of the current situation events should be delivered as fast as possible which can be achieved better by the new protocol.

A second useful way of improving the eventing functionality is by allowing a subscriber to subscribe to a single event of a service instead of to all events of a service. Allowing this type of event subscription will not limit the number of events, but it will limit the number of subscribers an event gets sent to and as a result limit the data traffic necessary.

Services need to be more in control of the eventing system in two areas. Events should not be connected to state variables, this limits the use of events as the service can only sent events indirectly by modifying a state variable. A more flexible and direct way of implementing events is to use a function which can sent an event using two parameters (the event and its value, for example for a DataStorageService: freeSpace,12304).

Secondly extending eventing to differentiate between the different subscribers can be useful when eventing is used to notify a specific subscriber of a specific state change. This would require control points to identify themselves in a unique way just as services have to identify themselves uniquely. For services to be able to differentiate between the different subscribers it is also necessary that they can determine to whom they will be sending an event (to all subscribers or only to a specific subscriber which initiated the action which is causing the event). This can be accomplished by extending the function used for sending events from the previous paragraph with a subscriber parameter.

The enhanced eventing system would avoid constructions as proposed in 5.4 for implementing call-back actions. It also avoids the unnecessary amount of communication when an event is only intended for one specific service although other services are interested in similar events. The current solution in UPnP for subscriber specific events would be to encode the subscriber for whom the event is intended in the event value. This causes the event to be sent to all subscribers, although for most subscribers the event is of no interest. The extended eventing system is an elegant solution for this.

An alternative approach for offering a simple call-back system would be to equip a control point not only with an event URL, but also with a call-back URL which can be used by services to report they have finished processing some task and sent their result in a by UPnP specified syntax. Though this might look like a different solution than adapting the eventing protocol it essentially comes down to the same mechanism: using an URL of the control point for reporting a state-change in the service of importance for the control point. Depending on the exact implementation we have an added flexibility of this solution in that a service can sent more information back than only a single value when using a flexible call-back system which allows multiple parameters.

Discovery

In the current version of UPnP the discovery protocol can be a big bottleneck when scaling to an architecture with many services and/or many control points. The discovery protocol is the only part binding UPnP to a local network and is also the only real limitation to its scalability, if there would be a Google for UPnP devices and services UPnP could we used worldwide. The current discovery protocol requires a broadcast of an advertisement of every device and every service in a device when the device comes online. Moreover it has to sent a keep-alive advertisement every once in a while. On the other side we have control points searching for devices and services they are interested in by also sending a broadcast of a discovery message, to which devices and services have to respond again.

In total this amounts to a significant amount of traffic even for small architectures with only a few services and control points. A solution would be to set the duration for an advertisement to a very long period of for example a day. This however is not very usable for most applications because control points will usually need a relative fresh view on the state of the services implying a relative short duration of the advertisement close to the minimum of 1800 seconds. Setting a long advertisement duration would also only work in case of a few control points since the control point search mechanism is not affected by this duration.

To summarize we can say an improved version of the discovery protocol is needed for UPnP to be able to scale to large implementations with many devices, services and/or control points. An alternative could be to introduce a discovery proxy which can be used instead of the multicast messages. A solution using this idea is WS-Discovery [6]. Another solution is to create a UPnP service which acts as a discovery proxy and allows control points to use UPnP actions to discover other services.

XML description for control point

When a control point finds a specific service it wants to use it currently determines if it really is the service it was looking for by checking the type of the service. However sometimes a service can have the correct type but just not (yet) implement some functionality which is required by the control point. Strictly speaking this should not occur, but in real life situations it is not unthinkable that this may happen.

A solution for this is for the control point to also store an (XML) description of the service it needs with all the actions it requires to be functional. A quick comparison of the two descriptions, one of the service and one of the control point, done by the control point can warrant that a service provides all actions required by the control point.

A control point could also use this description of its required actions to find service types which are newer than the control point (and thus the control point has no knowledge of) but offer the functionality the control point requires with some additional (newer) actions. Actions could even be given version numbers so the control point can check the version of every action of a service which can in turn specify with which versions its action implementations are compatible. A similar approach is studied formally in the Space4U project, however instead of using services remotely they download components and run them locally and instead of a description they use a model of the component [20].

Data transfers

Transferring large amounts of data as parameters of a UPnP action is currently very problematic. A developer does not only risk exceeding the total action time-out of 30 seconds, but one action call also blocks a device and its services from executing other actions (see section 7.1.1). Additionally transferring a big amount of data as a parameter requires that a service or control point first loads this data into a program variable after which it can be marshalled into a parameter of the action in XML. This requires a big amount of memory to store the data temporarily and, depending on the exact implementation of UPnP, the data can be copied even multiple times between different program variables which is also very inefficient especially with large amounts of data.

A solution for transferring large amounts of data is presented in 5.4 and 5.3, however this solution is not very transparent and eliminates the abstraction from the nodes a service is running on. Disadvantages are that we still are involved with programming sockets (or possibly a different dedicated data transfer protocol), retrieving host addresses (an argument for using UPnP is that the UPnP API takes care of the correct addressing of the actions and events, simplifying the task of the developer) and we introduced our own solution of doing things which makes interoperability with other services or control points difficult. A preferred way would be to use the UPnP API (or a service) to transport bulk data so we can also make use of the nice properties of UPnP (specifically automatic discovery and abstraction from the location of devices and services) for this type of communication.

Two solutions can be adopted to implement this in the UPnP standard. The first requires an extension of the standard, the second adheres more to the standard ways of doing things in UPnP and uses a new service type which handles data transfers. This could be a standardized service (as there are already a significant amount specified on UPnP.org [22] similar to the ConnectionManager service which is used for managing media streaming between different devices (MediaRenderDevice and the MediaServer device). It still leaves the exact protocol used up to the developer whom can choose the protocol best suited for his needs (FTP, or a simple TCP connection or maybe a secure FTP connection), but can also use the benefits of UPnP for finding and connecting the communication source and target.

7.2 UPnP API

7.2.1 Improvements for performance

Keep TCP connections open

Setting up a TCP connection is costly. First a socket has to be created and than that socket has to be connected to a server socket on the other side of the connection which in turn also has to create a socket for the server to be able to receive the data. Setting up the TCP connection also requires the TCP handshake which already needs a significant amount of time given the fact we typically communicate a total of about 1000 bytes of data over this TCP connection.

Some control points typically use only a very limited number of services, for example a control point of our DSN typically uses only one DirectoryService and one FileService as long as they don't go offline. It would be very well possible for this control point to keep open its TCP connections with these two services for future use.

However the problem in this example is that the DirectoryService and the FileService also have to keep the connection open. These services can be used by a big number of control points. Keeping all TCP connections open for future use will result eventually in a huge number of open TCP connections on the service side, all using resources and slowing down the node on which the service is running. Also the service now has to check for action requests on all of these sockets, something which can be done rather efficiently, but still will probably slow down the communication again when there are many connections and thus undoing the gain we made by keeping the connections open.

These problems can be solved by keeping a connection only open within a specific interval after opening it. We can also just keep the socket and close the TCP connection, the socket can then be reused for new connections eliminating the need to create a new socket every time. It will require some testing to find out which of these options gives the biggest performance improvement. It could very well be that in some cases one option is the best while in other cases another option is the best. This solution stays within the boundaries of the specification of UPnP, for a better solution which brakes with the specification we refer to the next section 7.2.1

Do not use TCP

The RPCs we compared with in section 6.5.3 use UDP instead of TCP as a transport protocol. As we already noted in the previous section, setting up a TCP connection is costly considering we only transfer small amounts of data. So it is a good idea to trade in TCP for UDP.

This introduces the problem that UDP is an unreliable protocol compared to TCP, so the UPnP API will have to make the data transfer reliable by using checksums and retransmissions. Similar solutions have been adopted by existing RPC protocols, for example the Firefly and DCE RPC discussed in section 6.5.3. Despite slower hardware and slower network communications in those tests they perform at an almost similar level as our UPnP communication at the moment indicating we could gain a significant amount of speed by adopting a similar protocol for communication.

Moreover in [12] TCP is replaced in the UPnP eventing protocol by UDP, extended with delivery guarantee. The goal of this replacement also was to get a performance boost, but this time for the eventing system. The results presented there show a significant performance improvement motivating a similar approach to improving the performance of the UPnP control action protocol.

If only one API uses this change a device using it will not be able to communicate with devices using an other API anymore. For some applications this does not have to be a problem because the devices and control points will all be using the same API. This can be in an experimental environment or for example in a university or test lab. For all other applications this adaptation of the protocol will also have to be made in the UPnP specification so all control points and devices using different API's will still be able to communicate with each other.

Use a simple, lightweight and fast XML parser

As long as UPnP uses XML to communicate control actions we can still make handling this XML as fast as possible by using a very simple and fast XML parser. In our DSN example we replaced the Xerces parser with the kXML parser which resulted in a significant performance improvement (see section 6.5.1.

Since the XML code for actions (and events) is rather simple it might even be beneficial to write a dedicated XML parser which can only handle these two XML types and in one run not only parses the XML but also immediately creates the necessary structures (objects) for the API to use. This could be a very simple parser whose code is generated on-the-fly using the XML description of services and their actions and can only parse one specific action.

As already suggested in section 7.1.2 the best solution for speed improvements still is to remove XML completely from the communications for actions. In figure 7.2 we can see that during an action call XML code is parsed on two occasions (the request XML and the response XML) and in both cases, according to the profiler (JProfiler) we used, this still takes a significant amount of time (more than 20% of the total time a control point or service spends on an action) which also increases when we increase the request or response data size.

Simplify API code

We noted in section 6.5.3 on RPC that compared to the RPC protocols our UPnP API is not optimized for speed. Call and return paths are rather long (they have to pass through numerous functions, objects and lookups before reaching their target). The API is created in a very structured way which makes it very suitable for reading and maintenance.

However this structured API with many functions and objects using each other and inheriting from each other causes long call-paths internally in the API. Rewriting parts, or even the whole, API with speed optimization in mind, i.e. get the data as fast as possible at the code executing an action and get the response as fast as possible back to the user using the control point, will probably give an improvement in the speed.

For example in the current API all information on devices, services, actions and statevariables are only stored in a tree structure exactly similar to the XML description tree structure of the device. When getting information on a specific service (like its name, type or a list of its actions) we use the getService function, however this function first builds a list of all services by searching (every time again) over the nodes of the XML tree. Only when the list of all services is created the function searches again in this list to find the service requested. A similar approach is used when searching for actions of a service etc. This is only one example of many things which can be done in a more time efficient way. Another example is in figure 7.2 the ActionRequest.getActionName() function (in the lower left corner), it uses in total 4 additional functions and the XML parser to retrieve the action name, nobody could argue this is a fast way of implementing this.

Apart from this complicated internal API structure we can also observe the call-path through which an action call traverses from a control point through several layers, over the network and through several service side layers. In figure 7.2 we can see that there are many methods on the call-path which of course contradicts the "handle as fast as possible" idea. Partly we see that this incredible number of methods is caused by the complex structure of the API, for some simple information (like requesting an action name) many functions have to be traversed eventually ending in a function parsing the XML and searching for this value, this seems rather expensive and could be handled simpler.

The other part is caused by the structured and layered setup of the entire API. All functionality is broken down into small parts which all are implemented in a specific object (often using many other objects) causing many method calls. The ideal case, as it is with many RPC protocols, is that from action call to action result we only have a few functions which are accessed, minimizing the total delay. This would result in a figure complete different from figure 7.2, we would only have a few methods performing a task as fast as possible.

Even if we would have a fast as possible API we would still have the layers which are situated below the API. From the API messages (data) is transfered (copied) to the operating system which in turn has to place the data in packages and copy it to the network interface buffer. Vice-versa on the other side the data travels through the buffer of the network interface to the operating system and only then to the API and the application. Integrating these parts better and avoiding the copying of data between the different layers would be necessary to achieve maximum performance, but doing this will be a difficult task.

Priority assignments

When the load of the node a device or service is running on is very high our test results have shown that the response time for an action increases dramatically. A fix for this is to use a real-time approach in designing the API. Using this we can make a service behave predictable in all circumstances, even under high load.

For this to work we will have to do a real-time analysis of the API and the services. This means we have to determine all the different tasks a service has to fulfill and assign deadlines to them. We should also determine a scheduling algorithm which will schedule all these tasks. Eventually this will provide us with services with a predictable (performance) behavior under all loads, in contrast with the erratic behavior we observed during our different tests with interference.

7.2.2 Improvements for usability

Improve the interface of the API

The current interface for creating a service, but even more so for creating control points, is rather tedious. After creating two control points it turns out that they have very much in common, actually they have so much in common we created an extra layer between the API and our application in which we place all the 'default' control point functions. For services and actions we noticed something similar and also made a wrapper round the API, this is already discussed elaborately in chapter 4.

Instead of having to create the additional wrapper the API should offer a much more simple interface in which a developer only has to implement a few simple functions (or classes) to implement the actions of a service. A developer should be able to call functions from a control point to call actions from a service and should not have to bother with building a list of the services currently online. A control point class should offer functionality to get the current available services and select a specific service for an action request. It should actually offer out-of-the-box similar functionality as we now use the wrapper introduced in chapter 4. All administration of (online) devices and services should initially be hidden from the developer, but if necessary it should be accessible.

Of course it does not have to be exactly the same as the wrapper we created, moreover it should still allow much of the flexibility we now have. For example we still want to be able to detect when a service comes online or goes offline. The Intel UPnP toolkit [24] offers a simple interface for UPnP, however in some cases it is to restrictive for what we want to do with it. That toolkit is very simple, yet restrictive, because it uses a few simple utility application. One tool is used to create a device and service description. The other tool generates from this description a service and a control point. This makes it for example difficult to create a control point which uses more than one service type, something which can be quite common in SOA's.

A different and maybe interesting approach for creating UPnP services is to use the class definition. While developing a service we do not bother about the exact service and action descriptions, we implement actions as normal methods of a normal class (which we derive of a service class to implement all the service specific code) mixed with the ordinary methods of the the class (which we do not want to be available as actions). Next we provide a list with - or set a specific variable for - the methods we want to be exported as actions of the service (or in Java this could be done by providing an interface which requires all the methods which should be exported). Finally we can build a very simple device which just creates service objects using the service classes and when the device is started it derives its own and the service and action descriptions from the implemented variables and some service variables we set in the service class (like the service type and the service version). Like this we can even transform existing components very easily into services.

We conclude that there should be a very simple interface for creating control points and services and that there are multiple approaches for doing this. This interface should offer all functionality necessary for creating a normal functioning control point and service. Further it should not be difficult to do more complicated things with services and control points.

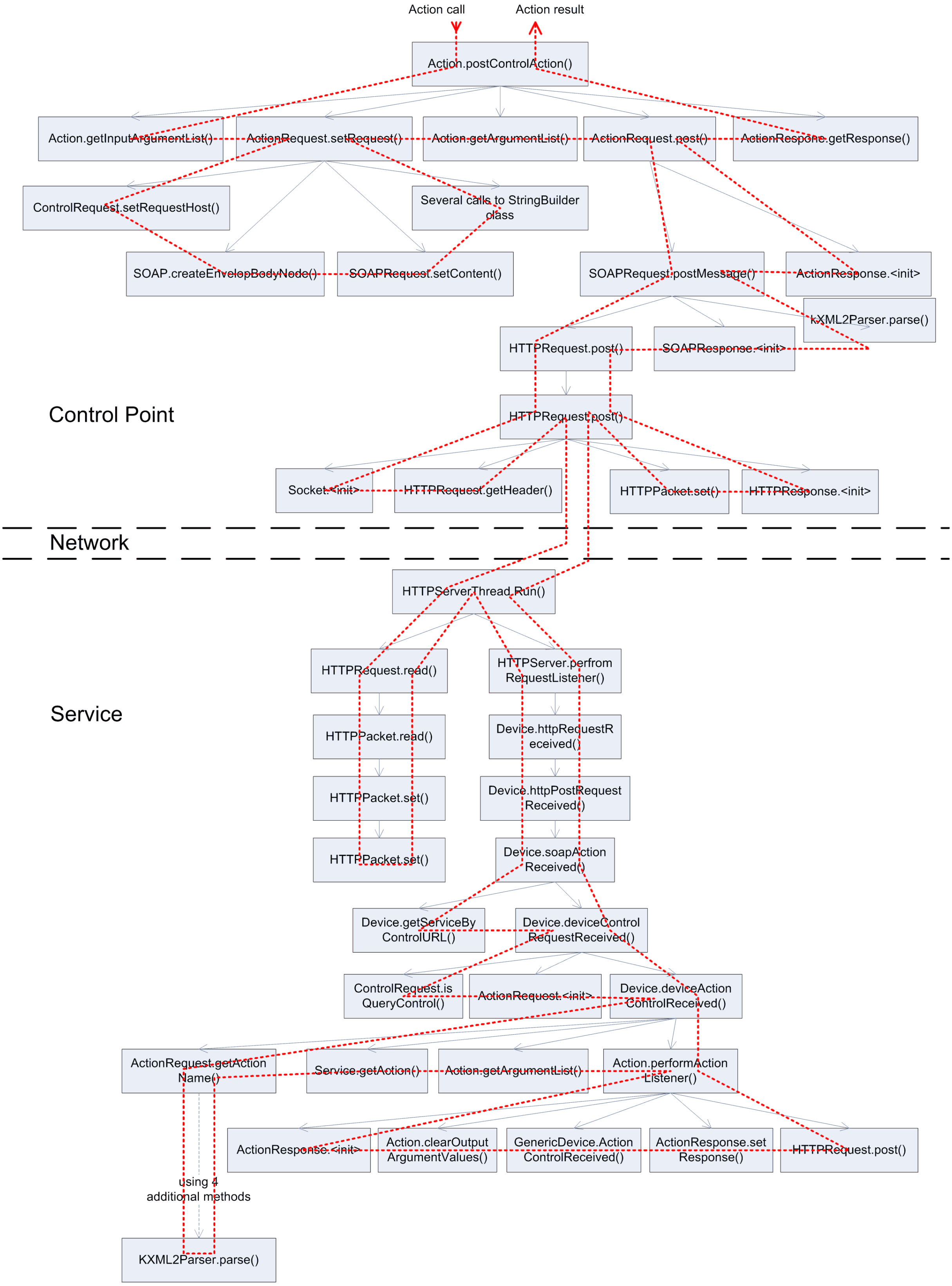
Action call
Action result
Action.postControlAction()
Action.getInputArgumentList()
ActionRequest.setRequest()
Action.getArgumentList()
ActionRequest.post()
ActionRespone.getResponse()
ControlRequest.setRequestHost()
Several calls to StringBuilder class
SOAP.createEnvelopBodyNode()
SOAPRequest.setContent()
SOAPRequest.postMessage()
ActionResponse.<init>
kXML2Parser.parse()
HTTPRequest.post()
SOAPResponse.<init>
Control Point
HTTPRequest.post()
Socket.<init>
HTTPRequest.getHeader()
HTTPPacket.set()
HTTPResponse.<init>
Network
HTTPServerThread.Run()
HTTPRequest.read()
HTTPServer.perfrom RequestListener()
Service
HTTPPacket.read()
Device.httpRequestReceived()
HTTPPacket.set()
Device.httpPostRequest Received()
HTTPPacket.set()
Device.soapAction Received()
Device.getServiceBy ControlURL()
Device.deviceControl RequestReceived()
ControlRequest.is QueryControl()
ActionRequest.<init>
Device.deviceAction ControlReceived()
ActionRequest.getAction Name()
Service.getAction()
Action.getArgumentList()
Action.performAction Listener()
ActionResponse.<init>
Action.clearOutput ArgumentValues()
GenericDevice.Action ControlReceived()
ActionResponse.set Response()
HTTPRequest.post()
using 4 additional methods
KXML2Parser.parse()

Figure 7.2: The path through the Cyberlink UPnP API an action call traverses is rather complex. This representation lists only the methods which are most time consuming (according to JProfiler, a Java profiler). The actual execution of the action is performed in GenericDevice.ActionControlReceived. The arrows represent function calls while the dotted line represents (approximately) the flow of control.

Chapter 8

Extending functionality of a SOA

When an application is built upon several cooperating services it could simplify the process of adding new functionality to the application. Taking the application and adding new functionality, which was not taken into account during the initial design, can be quite difficult and can introduce many new errors and bugs.

With a Service Oriented Architecture we can add functionality to an application by simply creating a new service which operates in conjunction with the existing services to provide the new required functionality. Adding fault tolerance to account for failing services or network connections is a good example of a functionality which can be added in this way, so we'll discuss the addition of a fault tolerant service as an example. We will not go into all kinds of implementation details, this is just a sketch to show it can be done with only a few simple changes.

8.1 Fault tolerance

We already considered fault tolerance (by replication) very shortly when we introduced the alternative directory storage presented in section 5.6. However adding fault tolerance may seem difficult at first. It usually requires replication of information stored at only one location. In our DSN example all information we collect is stored in only one location (in the DirectoryService and in the DataStorageService(s)), so we would have to introduce replicas of all services except the FileService(s).

There are several ways of introducing fault tolerance to the system. One way is to extend the DirectoryService and the DataStorageService with functionality to synchronize states with other implementations of DirectoryServices and DataStorageServices. This means we have to specifically design the DirectoryService and the DataStorageService with fault tolerance in mind which will require significant adaptations to the implementation and probably a drop in the performance of the services.

Another way to add fault tolerance is to add an extra service to the network which will facilitate the replication of information. It will not bother the DirectoryService or the DataStorageService with this, but will, as transparently as possible, organize fault tolerance by replicating data from the DirectoryService and the DataStorageService. It could do this even by just using the default available functions of a DirectoryService and a DataStorageService.

8.1.1 Fault tolerance for the DirectoryService

If an extra service, which we will call FaultTolerantService from now on, is added, replication of instances of DirectoryServices will have to be done in both directions. Directories added or removed at DirectoryService 1 will have to be added or removed at DirectoryService 2 and vice versa, because control points using DirectoryServices don't care which one they use, they just take the first one available. So this makes replication of the directory structure more complicated, but it works very transparently because replicated versions are used automatically by control points.

This still leaves us with the difficult task of maintaining multiple DirectoryServices consistent with each other. For example if at DirectoryService 1 a file is removed, it still remains in DirectoryService 2. When comparing these 2 DirectoryServices contents it may look like the file has been added to DirectoryService 2 and thus will be added to number 1, this is of course undesirable behavior.

A very simple but effective solutions for this could be to let a DirectoryService generate an event for every change in the directory structure it contains, if an event for every change turns out to be a to frequent occurrence events can be merged together in batches. This event than can contain only a reference to the file which was changed, but it can also encode the change. Eventually what has changed can be determined by the FaultTolerantService and propagated to the other DirectoryServices in the application.

When considering the alternative design proposed in section 5.6 with the directory information stored as files we can omit synchronization completely and just take care that a DirectoryService updates its directory cache often enough so it has almost always the most recent version which is stored on the DataStorageServices. The stored version of the directory information can then be updated by all DirectoryServices and fault tolerance of this information is taken care of by making the implementations of the DataStorageServices fault tolerant. This is the approach we will be using in the rest of this chapter.

8.1.2 Fault tolerance for the DataStorageService

To provide fault tolerance for the DataStorageService we have to replicate the data stored in one instance of a DataStorageService. Replicating the data will preserve the data if one instance of the DataStorageService crashes, because the data will be present on other instances. This provides us also with "free" replication of the directory structure if we store the directory structure information on DataStorageServices, as we proposed in the previous section.

Replication strategies

When the FaultTolerantService replicates the information of DataStorageServices we can choose out of several alternatives:

- the FaultTolerantService traverses the directory structure of the DirectoryService(s) and replicates every file part found in there (which is not yet replicated) to different DataStorageServices than already listed in the list of file parts. After this it keeps the copies up-to-date.

- the FaultTolerantService copies the entire contents of a DataStorageService to another DataStorageService. After this it keeps the copies up-to-date.

The second alternative requires extra functionality in the DataStorageService (to copy its entire contents) and will introduce extra problems if we have DataStorageServices with different storage capacity, something which seems very probable. On the other hand the first alternative can use the already existing functionality of the DirectoryService and the DataStorageService and thus has our preference. When a FaultTolerantService starts it will traverse the entire directory tree in search for not replicated file parts. When found it will replicate them.

Accessing replica's

Now all file parts have replica's we only need a way for our DSN to access them. Again we can discern two approaches:

- We make a few changes in the directory structure so it can store alternative file parts on alternative locations. Moreover we adapt the FileService to use the alternative locations of files if one location fails and to not always use the same locations to balance the load between the DataStorageServices storing the same file(s).
- The FaultTolerantService checks the status of every DataStorageService, if one fails it replaces the entry in the directory structure of the files on that DataStorageService with the replicas.

The second approach has again some disadvantages. Space is taken up on the DataStorageServices, however this space cannot be traced back to some file occupying it because only the FaultTolerantService knows what is stored there. Moreover now we have to store a second list of file (part) locations at the FaultTolerantService which is very undesirable because now we would also need to replicate the FaultTolerantService to avoid loosing this information which will only introduce new problems.

The other method does have its disadvantages too, we will have to adapt two services (DirectoryService and FileService). Luckily these adaptations aren't very big and could have been foreseen while designing the original application. If we implement this rather simple extra future (support for multiple storage locations for a single file part) in these two services they can operate in the same way they did before. However adding a FaultTolerantService to the application will be easy and will introduce fault tolerant behavior by means of replication of the data on the DataStorageServices.

Keeping up-to-date

So far it is not that difficult, however when all files are replicated after a certain startup time of the FaultTolerantService, it has to keep the files up-to-date with each other. This means if one file is changed in one location, all its replicas will have to be changed and if a new file is created it will have to be replicated immediately (or at least as fast as possible).

We conclude we need a simple yet effective way of responding as quickly as possible to changes in data or the addition of new data. Since we can operate on the level of file parts we can choose where we want to monitor the changes since they can be observed on all three services, FileService, DirectoryService and DataStorageService.

Again we can use eventing to get the job done. If a file part, or multiple file parts, are updated or added to a file an event is generated by the DirectoryService or by the FileService. If these events would be generated by the DataStorageService it would be difficult to merge several updates into one bigger batch update. For example storing a new file can create many file parts which would generate many events on different DataStorageService instances, but can be handled by one event, indicating there is a new file which has to be replicated, if the event is generated on a DirectoryService or FileService. A FaultTolerantService should then subscribe to these events and when it receives one immediately take action and replicate the file parts or update the existing replica(s).

Scaling

In case of a big application with many service instances cooperating it can be necessary to have multiple FaultTolerantServices. Of course it would be a waste of CPU-time, network bandwidth etc. if they all respond to the same event and do the same work more than once possibly introducing inconsistencies. Therefor there needs to be a mechanism to determine if a certain event is already being taken care of by a FaultTolerantService or not. So when a FaultTolerantService starts acting upon an event it can check if it still has to do something or can safely skip this event because some other FaultTolerantService is already acting upon it.

To accomplish this there are several methods which can be followed. A simple solution is based on the fact that all actions are handled synchronously by a service. If a FaultTolerantService wants to replicate a specific file part because it received an event it calls an action of the DirectoryService with as a parameter the event on which it is responding. If no FaultTolerantService is responding yet upon the event the response to this event is stored with the file being replicated and the return value of the action is true, so the FaultTolerantService proceeds with replicating the file. However if there was already a response to this event stored with the file, nothing is done and the return value is false, signaling the FaultTolerantService to not replicate the file because another service is already handling this.

Chapter 9

Conclusion(s)

9.1 Conclusion

When starting this project the first goal was to explore the possibilities of implementing services using UPnP and using these services to build an application. First we explored the possibilities of UPnP for building services in chapter 4. Based on experience we got from building our distributed storage network we presented design practices for building such a service oriented architecture.

In chapter 5 we explored the possibilities and impossibilities of UPnP even further using our distributed storage network as an example of problems we encountered and solutions for these problems. Many of the difficulties and impossibilities we encountered can recur in other applications using UPnP to build services. The solutions presented in this chapter can easily be applied in other situations making this chapter a practical addition to chapter 4.

The performance analysis in chapter 6 provides us a deep insight in the workings of a UPnP implementation. Moreover we have been able to create a model which can be used to estimate the performance of a service without actually implementing it. We have also shown that the effects of a high processor load, UDP traffic and TCP traffic on the same network node which is running a UPnP service can have a big influence on the performance. This effect can also be predicted, however is dependent on the operating system and hardware used.

With the performance measurements we have been able to support the design practices in chapter 4. We can use the model for example to predict the performance of different service compositions and using that we can show that some compositions will perform much better than other compositions. The performance analysis also shows that the performance of UPnP is quite poor compared to the performance of remote procedure calls. This shows that much improvement can be made on the performance of UPnP.

During this project we learned a great deal about UPnP, its strengths and its weaknesses. Especially in the performance department we gained much insight. This lead to chapter 7 where we discussed issues we encountered and possible improvements for UPnP and UPnP implementations. This chapter is not only an evaluation of UPnP, it can also be used as a guideline for implementing a new UPnP API or toolkit. Moreover it can be used to learn from the "mistakes" made in designing and implementing UPnP when designing and implementing a new standard for applications we explored in this project. For example making a new version of UPnP which is optimized for performance would be useful and could bring near RPC performance to UPnP (although this would probably require sacrifices in other parts of the UPnP standard).

9.2 Future work

Most answers found in this project also raise new questions which can be answered my more research. Future research in this topic can focus on many different points, but we will list here a few main points which can be of interest:

- Implement an extension for the Distributed Storage Network (or another service based architecture), for example the service illustrated in chapter 8. Try different approaches for extending an existing application build on services to find design practices for this task.
- How can security be defined on network accessible services like the UPnP services? There should probably be user and service authentication, so a user can trust a service and vice-versa. How should this be added to an existing service oriented architecture or how can we incorporate this in UPnP for example?
- Define a general reliability service which can be used to determine the status of various service implementations (components). Combined with this can be to find a way of improving the reliability of service implementations.
- Define and implement an improved version of UPnP using some, many or all suggestions in chapter 7. This can be a version which specifically focuses on eliminating the most obvious weaknesses of UPnP or for example a real-time scheduled implementation allowing prediction of response times. An important improvement can also be to define a different, more scalable, discovery protocol for UPnP. This could make it possible to make a service discoverable over an entire network instead of only over a relative small part.
- Helpful with the previous point can be to do a more accurate comparison between RPC protocols and UPnP. This can provide a list of important difference and similarities between the two approaches.
- Execute more elaborate performance measurements. This can be to find a better model for layered actions under high CPU load or high network traffic. This can also be measurements on a bigger network where traffic has to travel through a router for example. This can add additional cross-traffic interfering with the UPnP communications.

Deliberation between abstracting away from network communication for convenience and having to continually consider communication is over a network because it is slower and bring benefits in multiple forms (replication etc..).

Appendix A

Test results

A.1 Normal situation

Measurements		Total		Control Point node					Service node			Differences	
	Name	Total	95% confidence	Init. req.	CP send	Wait for resp.	Recv. resp.	Process resp.	Recv. req.	Perf. req.	Send resp.	Rest	CP total
1	empty action	8.4	0.4	1.4	1.6	3.9	0.5	1.0	0.6	1.9	0.5	0.9	0.0
2	getListing	10.8	0.6	1.3	1.6	5.8	0.7	1.4	0.6	3.8	0.4	1.0	-0.1
3	createFile[1]	16.8	0.7	1.3	1.7	13.0	0.5	1.0	0.6	10.7	0.6	1.1	-0.6
4	createFile[2]	8.9	0.4	1.3	1.6	4.5	0.5	1.0	0.6	2.3	0.5	1.0	0.0
5	storeData	44.9	1.4	1.2	1.1	42.0	0.2	0.7	0.5	40.3	0.3	0.9	-0.3
6	storeBlocks	9.2	0.5	1.4	1.6	4.7	0.5	1.1	0.6	2.6	0.4	1.1	-0.2
7	updateFilePart	7.3	0.4	1.2	1.3	3.9	0.3	0.8	0.4	2.3	0.3	1.0	-0.4
8	updateFileSize	5.9	0.3	1.1	1.3	2.9	0.4	0.6	0.3	1.4	0.3	0.9	-0.3
9	getFileInfo	6.7	0.2	1.0	0.9	3.8	0.5	0.7	0.4	2.3	0.3	0.9	-0.2

[1] createFile of the file service (a layered action)

[2] createFile of the directory service (a non-layered action)

A.2 UDP traffic

Measurements		Total		Control Point node					Service node			Differences	
	Name	Total	95% confidence	Init. req.	CP send	Wait for resp.	Recv. resp.	Process resp.	Recv. req.	Perf. req.	Send resp.	Rest	CP total
1	empty action	46.1	3.2	8.5	3.2	30.8	1.3	2.4	6.0	6.5	1.5	16.9	-0.1
2	getListing	53.2	1.5	8.2	3.0	36.2	1.6	3.1	6.4	12.5	1.4	15.8	1.2
3	createFile[1]	95.6	1.4	8.2	3.1	82.7	0.9	2.4	1.3	62.2	7.2	11.9	-1.6
4	createFile[2]	48.3	1.1	8.1	3.1	31.9	1.4	2.3	6.4	7.6	1.7	16.2	1.5
5	storeData	328.8	9.0	7.5	2.6	276.6	1.2	1.9	7.0	256.9	1.1	11.6	39.0
6	storeBlocks	48.6	1.0	8.3	3.1	33.7	1.5	2.6	6.7	8.2	1.6	17.3	-0.7
7	updateFilePart	44.2	1.1	7.7	3.1	30.2	1.0	2.0	5.7	6.6	1.3	16.5	0.2
8	updateFileSize	41.3	0.9	7.9	2.4	28.3	1.0	1.6	5.5	5.4	1.4	16.1	0.0
9	getFileInfo	44.1	0.7	7.7	2.9	31.3	1.3	2.0	5.6	9.1	0.9	15.7	-1.1

A.3 TCP traffic

Measurements		Total		Control Point node					Service node			Differences	
	Name	Total	95% confidence	Init. req.	CP send	Wait for resp.	Recv. resp.	Process resp.	Recv. req.	Perf. req.	Send resp.	Rest	CP total
1	empty action	26.2	3.2	1.7	2.5	16.8	0.8	1.4	0.6	1.9	0.5	13.8	3.0
2	getListing	29.2	5.9	4.5	2.8	38.1	1.4	2.8	2.9	22.2	2.7	10.3	-20.4
3	createFile [1]	81.4	4.8	2.1	1.6	75.8	0.6	1.2	1.1	48.8	1.8	24.1	0.1
4	createFile [2]	24.6	3.1	1.4	2.1	19.3	0.5	1.7	1.5	8.5	2.4	7.0	-0.3
5	storeData	323.3	9.8	2.5	2.1	316.4	0.7	1.5	1.4	306.8	0.7	7.5	0.1
6	storeBlocks	25.7	2.5	1.8	2.3	16.0	0.8	1.6	1.9	7.3	1.1	5.6	3.2
7	updateFilePart	25.4	3.1	2.5	1.6	14.1	0.4	0.8	1.5	6.1	1.2	5.3	6.0
8	updateFileSize	23.3	2.8	1.6	2.0	15.0	0.4	0.9	1.5	6.7	1.7	5.2	3.4
9	getFileInfo	20.4	2.8	1.5	1.5	16.3	1.1	0.9	1.4	9.2	0.7	5.1	-0.8

[1] createFile of the file service (a layered action)

[2] createFile of the directory service (a non-layered action)

A.4 CPU Load

Measurements		Total		Control Point node					Service node			Differences	
	Name	Total	95% confidence	Init. req.	CP send	Wait for resp.	Recv. resp.	Process resp.	Recv. req.	Perf. req.	Send resp.	Rest	CP total
1	empty action	78.8	3.9	1.4	1.4	71.7	0.3	1.1	0.4	1.9	0.6	68.9	2.9
2	getListing	74.8	3.6	1.3	1.6	70.7	0.6	1.2	0.6	4.5	0.5	65.1	-0.7
3	createFile [1]	91.2	6.2	1.4	1.5	86.8	0.5	1.0	1.2	55.4	0.7	29.4	0.1
4	createFile [2]	75.3	3.6	1.3	1.5	69.7	0.3	1.0	0.5	2.7	0.4	66.1	1.5
5	storeData	247.9	7.0	1.5	1.1	245.4	0.2	0.8	0.5	241.8	0.8	2.4	-1.2
6	storeBlocks	77.4	3.8	1.3	1.5	73.4	0.5	1.1	0.5	3.0	0.5	69.4	-0.4
7	updateFilePart	75.6	3.7	1.3	1.2	70.9	0.5	0.8	0.9	1.9	0.8	67.4	0.9
8	updateFileSize	75.2	3.5	1.2	1.0	70.8	0.3	0.6	0.2	1.5	0.4	68.6	1.3
9	getFileInfo	75.0	3.7	1.2	1.0	71.3	0.4	1.0	0.3	2.7	0.3	68.0	0.1

[1] createFile of the file service (a layered action)
[2] createFile of the directory service (a non-layered action)

Appendix B

UPnP XML Descriptions

As an example this is the UPnP XML description of the DirectoryService. Since these description are quite long we will only give one example.

```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <actionList>
    <action>
      <name>getListing</name>
      <argumentList>
        <argument>
          <name>path</name>
          <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
          <direction>in</direction>
        </argument>

        <argument>
          <name>listing</name>
          <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
          <direction>out</direction>
        </argument>

        <argument>
          <name>result</name>
          <relatedStateVariable>A_ARG_TYPE_BOOLEAN</relatedStateVariable>
          <direction>out</direction>
          <retval />
        </argument>

      </argumentList>
    </action>

    <action>
  <name>getFileInfo</name>
  <argumentList>
    <argument>
      <name>path</name>
      <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
      <direction>in</direction>
    </argument>

    <argument>
      <name>file</name>
      <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
      <direction>in</direction>
    </argument>

    <argument>
      <name>fileInfo</name>
      <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
      <direction>out</direction>
    </argument>

    <argument>
      <name>result</name>
      <relatedStateVariable>A_ARG_TYPE_BOOLEAN</relatedStateVariable>
      <direction>out</direction>
      <retval />
    </argument>

  </argumentList>
</action>

<action>
  <name>createDir</name>
  <argumentList>
    <argument>
      <name>path</name>
      <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
      <direction>in</direction>
    </argument>

    <argument>
      <name>newDir</name>
      <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
      <direction>in</direction>
    </argument>

    <argument>
      <name>result</name>
      <relatedStateVariable>A_ARG_TYPE_BOOLEAN</relatedStateVariable>
      <direction>out</direction>
      <retval />
    </argument>

  </argumentList>
</action>
```

```
<action>
  <name>createFile</name>
  <argumentList>
    <argument>
      <name>path</name>
      <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
      <direction>in</direction>
    </argument>

    <argument>
      <name>newFile</name>
      <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
      <direction>in</direction>
    </argument>

    <argument>
      <name>result</name>
      <relatedStateVariable>A_ARG_TYPE_BOOLEAN</relatedStateVariable>
      <direction>out</direction>
      <retval />
    </argument>

  </argumentList>
</action>

<action>
  <name>deleteDir</name>
  <argumentList>
    <argument>
      <name>path</name>
      <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
      <direction>in</direction>
    </argument>

    <argument>
      <name>dir</name>
      <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
      <direction>in</direction>
    </argument>

    <argument>
      <name>result</name>
      <relatedStateVariable>A_ARG_TYPE_BOOLEAN</relatedStateVariable>
      <direction>out</direction>
      <retval />
    </argument>

  </argumentList>
</action>

<action>
  <name>deleteFile</name>
  <argumentList>
```

```
    <argument>
      <name>path</name>
      <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
      <direction>in</direction>
    </argument>

    <argument>
      <name>file</name>
      <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
      <direction>in</direction>
    </argument>

    <argument>
      <name>result</name>
      <relatedStateVariable>A_ARG_TYPE_BOOLEAN</relatedStateVariable>
      <direction>out</direction>
      <retval />
    </argument>

  </argumentList>
</action>

<action>
  <name>updateFilePart</name>
  <argumentList>
    <argument>
      <name>path</name>
      <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
      <direction>in</direction>
    </argument>

    <argument>
      <name>file</name>
      <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
      <direction>in</direction>
    </argument>

    <argument>
      <name>storageServiceID</name>
      <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
      <direction>in</direction>
    </argument>

    <argument>
      <name>filePartID</name>
      <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
      <direction>in</direction>
    </argument>

    <argument>
      <name>length</name>
      <relatedStateVariable>A_ARG_TYPE_INTEGER</relatedStateVariable>
      <direction>in</direction>
    </argument>
```

```
        <argument>
          <name>startAddr</name>
          <relatedStateVariable>A_ARG_TYPE_INTEGER</relatedStateVariable>
          <direction>in</direction>
        </argument>

        <argument>
          <name>result</name>
          <relatedStateVariable>A_ARG_TYPE_BOOLEAN</relatedStateVariable>
          <direction>out</direction>
          <retval />
        </argument>

      </argumentList>
    </action>

    <action>
      <name>updateFileSize</name>
      <argumentList>
        <argument>
          <name>path</name>
          <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
          <direction>in</direction>
        </argument>

        <argument>
          <name>file</name>
          <relatedStateVariable>A_ARG_TYPE_STRING</relatedStateVariable>
          <direction>in</direction>
        </argument>

        <argument>
          <name>size</name>
          <relatedStateVariable>A_ARG_TYPE_INTEGER</relatedStateVariable>
          <direction>in</direction>
        </argument>

        <argument>
          <name>result</name>
          <relatedStateVariable>A_ARG_TYPE_BOOLEAN</relatedStateVariable>
          <direction>out</direction>
          <retval />
        </argument>

      </argumentList>
    </action>

  </actionList>
  <serviceStateTable>
    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_INTEGER</name>
      <dataType>int</dataType>
      <defaultValue>0</defaultValue>
    </stateVariable>

    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_FLOAT</name>
      <dataType>r4</dataType>
      <defaultValue>0</defaultValue>
    </stateVariable>

    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_BOOLEAN</name>
      <dataType>boolean</dataType>
      <defaultValue>0</defaultValue>
    </stateVariable>

    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_STRING</name>
      <dataType>string</dataType>
      <defaultValue>0</defaultValue>
    </stateVariable>

  </serviceStateTable>
</scpd>
```

Appendix C

(Dis)Advantages of components and services

C.1 Components

Advantages of using and developing components:

- Components are reusable because of their well-defined interface.
- Components can be composed easily, to form applications or new components.
- Components encapsulate and protect the (difficult) implementation of specific functionality.
- Components can easily be exchanged for other components with a similar interface and they can be easy deployed.

Disadvantages for developing components:

- Components require a lot of testing. Because they are going to be used in possibly very different unknown situations they have to be tested very thoroughly.
- Components require excellent documentation to be useful to developers.
- Because components can be used in many unforeseen situations they have to be able to handle every possible situation gracefully by either producing a result or producing a useful error-message.

Disadvantages of using components:

- Components will do approximately what is needed, but not exactly. Adaptation is needed when using them in real applications.
- Components can often be provided by third-parties, thus this requires trusting the third party.

C.2 Services

The advantages and disadvantages of services are not only based on the general notion of a service, but on the notion of service introduced in the final two paragraphs of section 3.1: services implemented on networks.

Advantages of using services:

- A service offers a general way of specifying functionality for very different situations. For example it can be used to describe the functionality of a layer of the network stack, but also to describe the service the railways can provide.
- Similar services can have many different interfaces, depending on the situation. Vice-versa, different services can provide a similar interface.
- Services can be used very easy in compositions. They can use other services (through their required SAP) to be able to provide their services (on their provided SAP). It is even possible to compose services during run-time using predefined compositions or an orchestrator which composes the services.
- Services can be independent of (programming) language and operating systems because they are available on a network.
- Quality of service can be easily checked against the required quality of service (which was specified) and can be guaranteed or negotiated by an external quality control service.
- Services are reusable.
- Services allow multiple different implementations, as long as they adhere to the service specification and interface.
- Services can be used very flexible in compositions, it is possible to connect services at run-time instead of doing this during design or implementation.

Disadvantages of services:

- Services are used over a network connection and thus respond slower than calling a local function.
- Quality of service specifications and requirements can give the illusion of a fail-safe system. However if the network goes down services will never be able to offer any required quality anymore.

Bibliography

[1] Mark Allman, *An evaluation of XML-RPC*, SIGMETRICS Performance Evaluation Review **30** (2003), no. 4, 2–11.

[2] Christian Bettstetter and Christoph Renner, *A comparison of service discovery protocols and implementation of the service location protocol*, August 28 2000.

[3] R. Braden, *RFC 1644: T/TCP — TCP extensions for transactions functional specification*, July 1994.

[4] Rémy Card, Theodore Ts'o, and Stephen Tweedie, *Design and implementation of the second extended filesystem*, Proceedings of the First Dutch International Symposium on Linux, 1994.

[5] E. Cooper, *Replicated distributed programs*, Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP), vol. 19, 1985, pp. 63–78.

[6] Jeffrey Schlimmer et al., *Web Services Dynamic Discovery (WS-Discovery)*, http://specs.xmlsoap.org/ws/2005/04/discovery/ws-discovery.pdf, april 2005.

[7] UPnP Forum, *UPnP Device Architecture*, http://www.upnp.org /resources/documents/CleanUPnPDA101-20031202s.pdf, 2003.

[8] Dominic Giampaolo, *Practical file system design with the BE file system*, Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999, Includes comparison with Apple Macintosh, Linux, and Microsoft Windows file systems.

[9] David K. Gifford, *Weighted voting for replicated data*, SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles, ACM Press, 1979, pp. 150–162.

[10] U. Gl, s Gurevich, and M. Veanes, *Universal plug and play machine models*, 2001.

[11] Intel, *Intel tools for upnp technologies*, http://www.intel.com/cd/ids/developer/asmona/eng/downloads/upnp/overview/index.htm#anchor_2.

[12] Y. Mazuryk J.J. Lukkien, *Improved eventing protocol for universal plugn play*, Proceedings of the 5th workshop on Embedded systems, STW, 2004.

[13] Abu Masud Khandker, Peter Honeyman, and Toby Teorey, *Performance of DCE RPC*, Tech. Report citi-tr-95-2, University of Michigan Center for IT??, January 1995.

[14] B. Lampson, *Designing a global name service*, Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing (PODC'86) (Calgary, Alberta, Canada), August 1986, http://research.microsoft.com/∼lampson/36-globalnames/Acrobat.pdf, pp. 1–10.

[15] Sergio Marti and Venky Krishnan, *Carmen: A dynamic service discovery architecture*, Tech. Report HPL-2002-257, Hewlett Packard Laboratories, September 23 2002.

[16] Y. Mazuryk, *Service oriented architectures in heterogeneous environments*, presentation at the SAN group meeting, november 2003.

[17] James McGovern, Sameer Tyagi, Michael Stevens, and Sunil Mathew, *Java Web services architecture*, Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 2003.

[18] Michael D. Schroeder and Michael Burrows, *Performance of firefly RPC*, SOSP, 1989, pp. 83–90.

[19] SOAP, *Simple object access protocol (SOAP 1.1)*, http://www.w3.org/TR/SOAP.

[20] Space4U, *Space4u technical university of eindhoven homepage*, http://www.win.tue.nl/space4u/.

[21] Andrew S. Tanenbaum and Maarten van Steen, *Distributed systems: Principles and paradigms*, Prentice Hall, Upper Saddle River, NJ, 2002.

[22] UPnP, *Upnp device and service standards*, http://www.upnp.org/standardizeddcps/default.aspp.

[23] ———, *Upnp software development kits*, http://www.upnp.org/resources/sdks.asp.

[24] Intel UPnP, *Intel upnp homepage*, http://www.intel.com/cd/ids/developer/asmo-na/eng/downloads/upnp/tools/index.htm.

[25] XML-RPC, *Internet remote procedure call*, http://www.xmlrpc.com/spec.