Magisterarbeit

Replication and Synchronization of Web Services in Ad-hoc Networks

Ausgeführt am

Institut für Informationssysteme
der Technischen Universität Wien

unter der Anleitung von

Univ.Prof.Mag.rer.soc.oec.Dr.rer.soc.oec. Schahram Dustdar

durch

Lukasz Juszczyk, Bakk.techn.
Schweglerstraße 47-49/16, A-1150 Wien
Matr.Nr. 9925140

Vienna, 09. May 2005

To my parents

Abstract

Due to their main advantage of offering standardized, extensible, and interoperable machine-to-machine interaction the popularity and importance of Web services is increasing. As a consequence Web services have became particularly interesting within mobile ad-hoc networks which can be used for building spontaneously an infrastructure for providing desired functionality or applying various business workflows. However, taking the dynamic behavior of ad-hoc networks into consideration, it becomes obvious that this infrastructure will be unreliable since hosts can easily relocate in the network and disappear completely making also their deployed Web services unavailable. In the course of writing this master thesis a system has been developed by means of which replication and synchronization of stateful Web services is performed within a highly dynamic network environment, able to handle all difficulties which unpredictable network topologies can raise.

Zusammenfassung

Web Services ermöglichen eine standardisierte, erweiterbare und vollständig kompatible Interaktion zwischen Maschinen. Diese Interoperabilität hat maßgeblich zur steigenden Akzeptanz von Web Services als Kommunikationsstandard im Internet beigetragen. Besonders dienlich wurden Web Services in mobilen Ad-hoc Netzwerken, die spontan gebildete und höchst dynamische Netzwerk- und Kommunikationsinfrastrukturen ermöglichen. Doch diese dynamischen Netzwerktopologien erschweren das Anbieten von hochverfügbaren Web Service-basierten Architekturen. Im Zuge dieser Diplomarbeit wurde ein System entwickelt, das Verfügbarkeit und Zuverlässigkeit von Web Services mit Hilfe von Replikation und Synchronisation signifikant erhöht und fähig ist, mit der Dynamik und Unberechenbarkeit von Ad-hoc Netzwerken umzugehen.

Contents

List of Figures

List of Tables

1 Introduction

During the last few years there has been a trend towards creating architectures for business processes based on distributed and independent services instead of building single applications which control the entire business logic. This new modularity offers a lot of benefits, including higher flexibility of applications as well as reduced costs of development due to reuse of services. Even though these Service Oriented Architectures (SOAs) can be built using various technologies, such as CORBA or Enterprise JavaBeans (EJB) Web services have become the most popular technology used for their realization. Reasons for this tendency are advantages of Web services such as the application of open standards or interoperability facilitating machine-to-machine communication within heterogeneous environments.

These very qualities are especially desired within mobile ad-hoc networks, which make it possible to build up network infrastructures spontaneously and quickly by using wireless communication and without the need of any preexisting devices or physical structure, such as access points, routers, or cables. Even if this technology not yet used widely nowadays it is expected to gain a wide field of application in the near future. This has been indicated by rising investments in companies and research.

If one regards the characteristics and advantages of these two technologies, it becomes obvious that Web services in ad-hoc networks offer a lot of new possibilities, such as infrastructures for communication and cooperation for groups of engineers, the needs of action forces in emergency situations, spontaneously built business processes, etc. However the dynamic behavior of ad-hoc networks raises the problem of transient service lifetimes and thus unacceptable reliability of whole workflows and processes. They can be brought to a halt if essential services cease to be available due to disconnections or relocations.

This problem can only be solved by replicating Web services and synchronizing stateful ones. Thus a number of backup services are kept to compensate failures and synchronization is used to keep a consistent state between all replicas in order to enable a smooth changeover to a backup service. Although there exist many solutions proposing static replication of services to gain a higher availability, this is not the case with regard to networks with transient topologies.

Consequently the goal of this thesis was to develop a solution for replicating Web services in highly dynamic ad-hoc networks to ensure their constant availability. The implemented solution is based on Apache Axis and performs replication by monitoring the state of the network and by adapting the placement of replicas to it.

1.1 Motivation

So far mobile ad-hoc networks have not yet been used widely but are gaining popularity. A topical example is the 100$ Laptop, which is developed by the One Laptop Per Child (OLPC) organization created by members of the MIT Media Lab. This project's intention is to provide cheap computers for child-education in least developed countries and one of the features provided by these laptops is the ability of building mobile ad-hoc networks via wireless communication. Another indication for a growing popularity and acceptance are a great number of companies which invest in research on new technologies based on ad-hoc networks.

Since ad-hoc networks are most often established in heterogeneous environments, communication within them requires a high level of interoperability and therefore calls for Web services: a communication standard which was designed for almost unlimited interoperability and extensibility.

The flexibility but also unreliability of Web services in ad-hoc networks can be well explained by using a scenario of an emergency situation (e.g. after an earthquake) where numerous action forces arrive and require a quick and highly available infrastructure for communication and for coordination of their squads. It can be easily achieved to setup such an infrastructure by providing Web services which offer the needed functionality for coordinating the activities of the squads, informing about the current situation, requesting reinforcements, etc. Figure 1 illustrates a simple scenario with host H1 providing one of these Web services which must constantly be available to all clients. The availability of this service is mainly endangered by a disconnection of host H6, which is critical for the coherence of the network, or by a failure of host H1, which is providing the Web service. The only way to ensure the availability of the Web service is to distribute replicas within the network which function as backups and are ready to take over the service's work after a failure. This requires a mechanism which reacts quickly to changes in the network and synchronizes the internal states of Web services continuously.

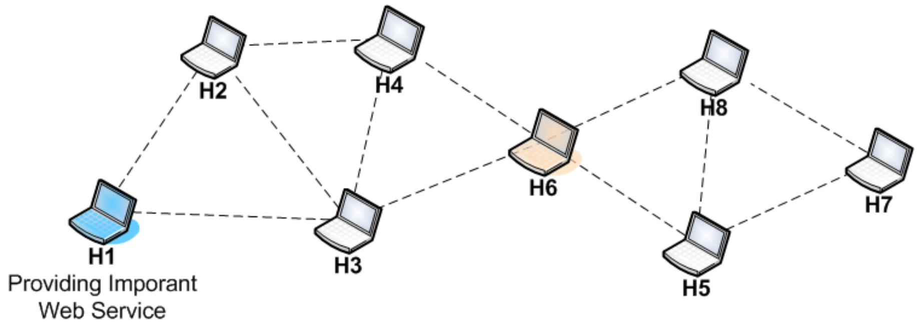

Figure 1: Ad-hoc Network Scenario with a Critical Structure. H6 is a critical spot splitting the network in case of a failure. (Links between hosts indicate direct connections)

1.2 Problem Definition

Replication and synchronization within as highly dynamic environments as mobile ad-hoc networks is far away from being trivial. In fact it is the most tricky environment one could image due to its unpredictable behavior. In the worst case hosts are connecting and disconnecting from the network often without leaving a chance of putting any structure into it. Unfortunately, there is no satisfactory solution which guarantees service availability in such a situation. However as such a worst case scenario happens extremely rarely, it still makes sense to work on a solution for the usual scenarios, which, anyway, has to deal with the following fault situations:

- *Unavailable hosts:*
  - Hardware/operating system crash
  - Shutdown
  - Empty battery
  - Unstable network connection
  - Moving out of wireless network range
- *Delays:*
  - High load on host/Web services
  - High load on network link
  - Unstable network connection
- *Changing network topology:*
  - Relocating hosts and routers
  - Splitting ad-hoc networks caused by router movements

Nonetheless if every host is up and running and the network works perfectly, the availability of Web services may be hampered due to configuration errors or software failures:

- *Unavailable Web services:*
  - Crash/shutdown of Web service container
  - Crash/undeployment of individual Web services
  - Firewalls blocking requests/responses

To overcome these obstacles for providing highly available Web services, it is unavoidable to use the approach of replication. Nevertheless the majority of replication techniques described in scientific papers or used in various software products are either focusing on performance and load balancing (e.g. Grids) or actually aim at increasing availability (e.g. HA-Clusters), but at the same time they use static and centralized resources such as controllers or request dispatchers. These systems are well-functioning in their domains, but considering the following requirements for ad-hoc networks a completely new solution is needed.

- Avoiding centralization
- Monitoring changes in availability & reacting to them
- Monitoring health of hosts & considering this while replicating
- Keeping bandwidth usage low
- Synchronizing service states quickly
- Making invocation of Web services convenient for the clients

To have a solution capable of meeting all the challenges mobile ad-hoc networks pose to replication, means to have a solution flexible enough to meet the challenges of any other network environment as well. This is exactly one of the main goals of this thesis and will be explained in more detail below.

1.3 Organization of this Thesis

The *current chapter* contains the introduction to the thesis, the motivation, and the problem description, which explains some basic requirements this thesis had to meet.

Chapter 2 provides an overview of used technologies and some concise reviews of technical papers which had a strong influence on this thesis' concept.

In *Chapter 3* the concept of the implemented system is explained, including a more detailed description of some basic requirements and an overview of the system's architecture.

Chapter 4 contains the documentation of the replication mechanism with its monitoring, leader elections and the whole replica placement mechanism.

In *Chapter 5* it is explained how synchronization of stateful services is performed.

In *Chapter 6* the evaluation of the system is documented, including ideas for future work.

Finally, in *Chapter 7* a conclusion and a summary of this thesis can be found.

2 State of the Art Review

In order to be able to understand the problem of using Web services within ad-hoc networks, it is necessary to take a look at both technologies and the difficulties their combination raises. In a nutshell the challenge lies in applying a technology which relies on static and centralized locations of registries and services to a network environment which cannot accept centralized solutions due to its highly dynamic structure.

2.1 Web Services

The implementation of Service Oriented Architectures is not bound exclusively to Web services, although they represent the most favored choice. It is perfectly possible to run an SOA while using Sun's EJB [25], Microsoft's Distributed COM [22], CORBA [13], and many others. Important for SOAs is the concept of using loosely-coupled and independent resources provided as application services by nodes on a network. This implies a certain interoperability. Mainly for this reason Web services have become the most popular standard, offering numerous additional benefits.

- Web services are based on open standards, such as XML [28], XSD [29], SOAP [24], WSDL [27], UDDI [26]. No license fees have to be paid for proprietary protocols.
- SOAP-based communication provides almost unlimited interoperability within heterogeneous environments.
- Code libraries for almost all programming languages and operating systems make development of Web services convenient.
- SOAP-messages and RPC-calls can be transported via any possible transport protocol. For instance the usage of HTTP [15] makes it easy to traverse firewalls.

In contrast to all the benefits, Web services have the handicap of poor performance compared to the other technologies. This includes a sometimes extensive communication overhead as well as a slower processing time, which is due to the costly parsing of XML & SOAP while various middleware platforms use binary transmission.

The usual lifecycle of a Web service can be described as follows:

1. service provider registers WSDL description at UDDI registry
2. service requester queries UDDI registry for desired Web service, retrieves WSDL data, extracts information, and adapts client routines
3. service requester invokes Web service

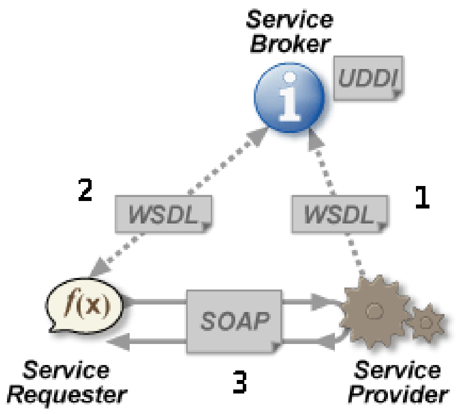

Figure 2: Typical Lifecycle of a Web Service. (Figure modified from [30] © H. Voormann, 25 March 2005)

Furthermore, the World Wide Web Consortium (W3C) and the Organization for the Advancement of Structured Information Standards (OASIS) have certified a number of open standards for extending Web service based communication. The most popular examples include:

- *WS-ReliableMessaging* for reliable delivery of SOAP messages
- *WS-Security & WS-Trust* offering cryptographic security
- *WS-AtomicTransaction & WS-BusinessActivity* for transactions, etc.
- and many more ...

2.1.1 XML & XSD

The Extensible Markup Language (XML) [28] is a standardized meta language, recommended by the World Wide Web Consortium (W3C) and used for defining machine- and human-readable documents. As a meta language, XML only specifies rules for creating valid documents while their content, such as structure and payload, is not subject to any restrictions. This implies unlimited extendability but every software processing an XML document must be aware of its composition. Furthermore, XML does not restrict the preferred way of representation in contrast to HTML [14].

As Listing 1 demonstrates, XML documents consist of a tree structure built of a root element with optional subelements and attributes. This makes it possible to describe almost every kind of data within XML.

Listing 1: "XML Sample Document"

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
    <customer id=12345>
        <name>John Jackson</name>
        <address>Foo-Avenue 5, Testcity</address>
    </customer>
    <ordereditems>
        <item amount=1 id="123-ab-90">
            <name>DVD recorder<name>
            <description> ... </description>
        </item>
        <item amount=5 id="456-xy-78">
            ...
        </item>
    </ordereditems>
</order>
```

Often XML documents contain elements from various modules serving different purposes, such as system-specific and user-defined information. To distinguish them, XML namespaces have been introduced which add a lot of flexibility for parsing and understanding the documents. This extension also allows to have elements with identical names, which are nevertheless distinguishable by the URIs of their namespaces:

Listing 2: ”XML Sample Document with Namespaces”

```
<?xml version="1.0" encoding="UTF-8"?>
<somedocument
        xmlns="http://www.anywhere.org/path"
        xmlns:ns1="http://www.abc.de/path"
        xmlns:ns2="http://www.url.com/path">
    <item attrib="value"/>      <!-- in default namespace -->
    <ns1:item attrib="value"/>
    <ns2:item attrib="value"/>
</somedocument>
```

Before an XML document is processed to extract its data, the document's validity has to be verified first. Validity of an XML document does not only mean that it is syntactically well-formed (e.g. all tags closed, attributes correctly set, etc.) but also implies correct content and structure. There are numerous possibilities to describe valid XML structures, with XML Schema Definition (XSD) [29] and the already out-dated Document Type Definition (DTD) representing the most popular ones.

Listing 3 contains an XSD example, defining a data structure named “person”, with the two mandatory fields “name” and “student”, and the optional field "birthdate". Listing 4 shows a valid document for this schema definition.

Listing 3: "XML Schema Sample Document"

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="person" type="personType"/>
<xsd:complexType name="personType">
    <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="birthdate" type="xsd:date"
                                        minOccurs="0"/>
        <xsd:element name="student" type="xsd:boolean"/>
    </xsd:sequence>
</xsd:complexType>
```

Listing 4: "Valid Document for XML Schema Definition in Listing 3"

```
<?xml version="1.0" encoding="UTF-8"?>
<person>
    <name>John Jackson</name>
    <birthdate>1985-05-21</birthdate>
    <student>true<student>
</person>
```

Web services use XML Schema mainly within the Web Service Description Language (WSDL) to define and check exchanged messages and data types.

2.1.2 SOAP

For encoding and exchanging messages and RPC-calls via Web services an XML-based protocol named SOAP [24] is used. In version 1.1 SOAP was an abbreviation for Simple Object Access Protocol, but due to that its evolution had moved away from being simple and towards offering more functionality than just accessing objects, the name was changed to simply *SOAP* in version 1.2.

SOAP mainly defines an optional header and a mandatory message body. The header contains meta information such as digital signatures, routing information, or authorization data while the actual serialized message payload is located within the body or its MIME attachments.

Listing 5 contains a short SOAP example, in which the data from Listing 3 is sent with a simple digital signature in the header.

Listing 5: ”SOAP Message Example”

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
    xmlns:env="http://www.w3.org/2003/05/soap-envelope">
    <env:Header>
        <s:digitalsignature
                        xmlns:s="http://www.url.com/crypto">
            <![CDATA[12AD5...9C]]>
        </s:digitalsignature>
    </env:Header>
    <env:Body>
        <b:person xmlns:b="http://www.url.com/path">
            <b:name>John Jackson</b:name>
            <b:birthdate>1985-05-21</b:birthdate>
            <b:student>true<b:student>
        </b:person>
    </env:Body>
</env:Envelope>
```

For transporting SOAP messages HTTP was established as the most convenient method, because of its ability to work well with firewalls. Alternatively, SOAP can be sent via an unlimited pool of protocols, including SMTP, FTP, proprietary protocols, etc. An advantage of using SMTP is that requests may be processed whenever the service provider is ready to fetch it, which results in an asynchronous message-based communication.

2.1.3 WSDL

For describing the characteristics and provided functionality of Web services, the Web Service Description Language (WSDL) [27] was introduced, which is again an XML-based meta language. WSDL files are usually retrieved from UDDI [26] registries to find out details about operations, data types, transport methods, location, etc. for building invocation stubs which are used to access the Web service.

Listing 6 contains an example WSDL file, automatically generated by Apache Axis [9] for its VersionService with only one operation named "getVersion". If one considers this example, the big amount of XML data to describe even simple Web services becomes apparent, since the following details have to be defined:

- `Lines 4-8`: type declarations for input/output messages, which can be either empty, primitive types, such (e.g. integer, boolean, string) or XML Schema Definitions for complex types.
- `Lines 9-16`: port declaration linking input/output messages to the operation "getVersion".
- `Lines 17-32`: service binding, defining HTTP as the transport method and "getVersion" as the only operation:
  - `Lines 21-30`: operation declaration, with "getVersionRequest" and "getVersionResponse" as input/output types, defining also the encoding styles for the messages.
- `Lines 33-37`: service description, including name, binding and location.

In a nutshell, this example describes a Web service named "VersionService", located at `http://somehost:8080/axis/services/Version` and invoked via HTTP. It provides one operation named “getVersion”, accepts no arguments and returns a string.

Listing 6: ”WSDL Data for Axis VersionService. (Declarations of namespaces and encoding styles were omitted for clarity)”

```xml
<?xml version="1.0" encoding="UTF-8"?>
<w:definitions targetNamespace="..."
        xmlns:w="http://schemas.xmlsoap.org/wsdl/"  ... >
    <w:message name="getVersionResponse">
        <w:part name="getVersionReturn"
                type="soapenc:string"/>
    </w:message>
    <w:message name="getVersionRequest"></w:message>
    <w:portType name="Version">
        <w:operation name="getVersion">
            <w:input message="impl:getVersionRequest"
                     name="getVersionRequest"/>
            <w:output message="impl:getVersionResponse"
                      name="getVersionResponse"/>
        </w:operation>
    </w:portType>
    <w:binding name="VersionSoapBinding"
               type="impl:Version">
        <ws:binding style="rpc" transport=
                "http://schemas.xmlsoap.org/soap/http"/>
        <w:operation name="getVersion">
            <ws:operation soapAction=""/>
            <w:input name="getVersionRequest">
                <ws:body encodingStyle="..."
                         namespace="..." use="encoded"/>
            </w:input>
            <w:output name="getVersionResponse">
                <ws:body encodingStyle="..."
                         namespace="..." use="encoded"/>
            </w:output>
        </w:operation>
    </w:binding>
    <w:service name="VersionService">
        <w:port binding="impl:VersionSoapBinding"
                name="Version">
        <ws:address location=
                "http://somehost:8080/axis/services/Version"/>
    </w:port>
     </w:service>
</w:definitions>
```

2.2 Ad-hoc Networks

In 1997 the IEEE 802.11 Working Group [18] specified the first standard for connecting hosts via wireless links (Wireless Local Area Network, WLAN), which has numerously been modified since then - primarily for increasing the speed. This new network technology was designed to be used either in a managed mode, using access points, or in an ad-hoc mode, where a number of hosts can set up a network structure spontaneously without requiring any preexisting infrastructure.

Even though both modes are highly flexible and open up completely new fields of application, this thesis concentrates on ad-hoc networks and the problems they bear. By virtue of the much more dynamic topology of ad-hoc networks they pose a higher challenge to the flexibility of the replication mechanism. If a solution is able to handle ad-hoc networks, it is also able to handle the difficulties of managed networks, resulting in an almost universally applicable replication mechanism.

The most important feature of ad-hoc networks is the ability of each host to act also as a router to facilitate a coherent network. This is quite a hard job for the routing protocols, since they have to react quickly to changes in the network in order to ensure that every node is reachable from every other connected device. Due to the fact that the nodes have limited knowledge not only about the currently existing but also the future topology, all these protocols rely on broadcasts informing about changes, which have to be taken into consideration in the local routing tables.

The reason for the dynamics of ad-hoc networks is mainly the fact that hosts move around in the network and thus change its structure, or depart beyond the range of the wireless network, which results in a disconnection (see Figure 3). Furthermore the feature of routers connecting hosts to networks may result in a networks splitting to subnetworks, due to a disconnection of a router (see Figure 4). This can be often avoided by using redundant routes.

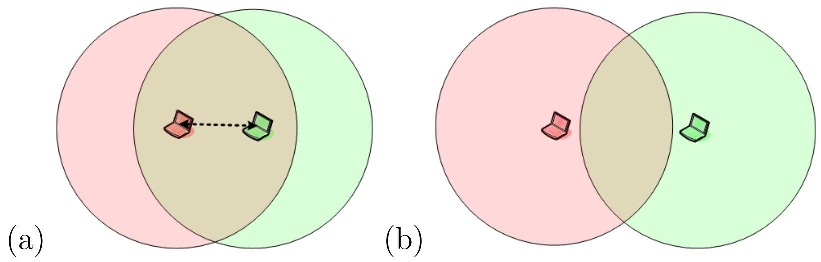

Figure 3: Wireless Connections. (a) Connected hosts within wireless range. (b) Hosts beyond wireless range.

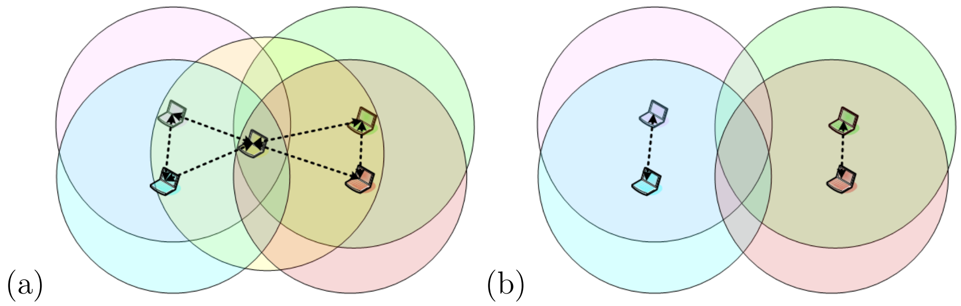

Figure 4: Host Structures in Ad-hoc Networks. (a) Coherent network structure. (b) Split networks due to disappearing of connecting router.

2.3 Web Services in Ad-hoc Networks

As explained in Chapter 2.1, Service Oriented Architectures based on Web services partly rely on static resources, such as UDDI registries or the actual locations of the services, retrieved from WSDL files. However if one looks at the dynamics and unpredictability of ad-hoc networks, explained in Chapters 1.2 and 2.2, one possibly gets a (false) impression namely that is is impossible to combine the two technologies. The main difficulties are on the one hand to discover Web services in the network and on the other hand to ensure their availability.

To meet this challenge was the aim of this thesis and Jaroslaw Lazowski's master thesis entitled "Web Service Discovery in Ad-hoc Netzwerken" [8]. The combination of both systems makes it possible to apply SOAs in a dynamic network environment by:

- providing a discovery technique which registers Web services at distributed UDDI registries, updating them automatically after changes in the network were detected. This is done by J. Lazowski's system
- replicating Web services for high availability is the task of this thesis

J. Lazowski explains in his thesis how discovery and registration of Web services is done. This includes the following steps:

1. Nodes are classified as either common nodes or master hosts. Master hosts have sufficing resources for running a light-weight UDDI registry plus for monitoring other nodes.
2. The network is arranged in star structures, as illustrated in Figure 5, with master hosts running UDDI registries and controlling groups of common nodes. Furthermore all master hosts are aware of each other and this way are able to retrieve a complete view of the network at any time.

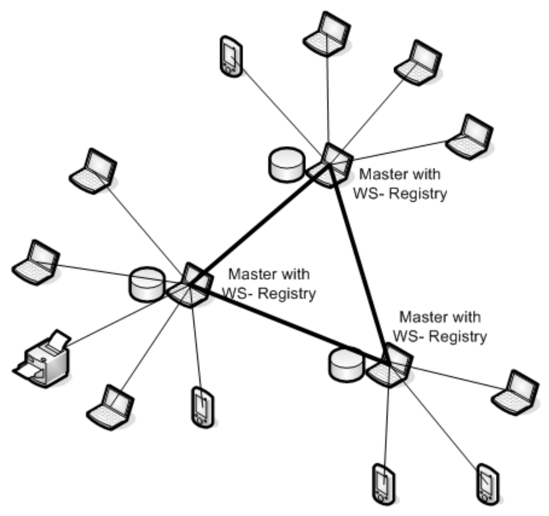

Figure 5: Network Topology with three Star Structures. (Figure modified from [8], ©J. Lazowski, 27 March 2005)

3. Hosts are monitored and their services registered and unregistered if a host connects to a group or disconnects from it. New hosts find their master hosts via JXTA-messages [20], publish their Web services, and confirm their continuous availability by sending periodical heartbeat messages.

4. Deployments and undeployments of Web services are detected via an extension to Apache Axis, which updates the UDDI registries.

As a consequence the network is populated with several UDDI registries at well-known locations, containing up-to-date WSDL data for all discovered services. Without relying on static centralized registries, this solution fulfills the requirement of Service Oriented Architectures to have the possibility of finding Web services easily.

A more detailed description of the discovery system and its cooperation with the replication mechanism can be found in J. Lazowski's thesis [8] as well as in the technical paper [7].

2.4 Related Work

As a matter of course the method of using replication to achieve higher availability and fault tolerance is not a new idea. In fact replication in computer science has been used for a very long time and in various forms, for instance by employing redundant devices in early space flight missions. Today the following areas of application belong to the most important and best known ones:

- *HA-Clusters* achieving high availability due to redundant nodes
- *Mirroring RAIDs* masking hard disk crashes by replicating disks
- *Replicated databases* benefiting performance and fault tolerance by employing distributed copies
- *Replicated file systems*, such as CODA, distributing redundant data on multiple hosts
- *DNS servers* on the Internet, using replication of records to backup servers

Furthermore the scientific problem of high availability, replication, and synchronization is addressed in numerous technical papers, reports, books, etc. Although most of them do not focus on dynamic networks, they often contain a lot of good ideas which can partially be adapted to ad-hoc networks. In the following, three selected papers are presented which has a great impact on the development of the system.

In the paper entitled “Adding High Availability and Autonomic Behavior to Web Services” [2] K. Birman et al claim that Web services miss some components which have become unavoidable within other highly available systems and, at the same time, are expected to be self-configuring and reliable. They propose several extensions to Web services for self-diagnosis and self-repairing as a solution to this problem. They distinguish between monitoring of single components on the one hand and aggregated properties of the system on the other hand. The second method is able to detect failures noticed only by a group of clients. Moreover they introduce event notification for informing other components about own failures, giving them them opportunity to roll over to backup resources. Another important feature is the consistent and reliable group communication, viz talking transparently to a replicated group of components, using virtual synchrony.

Although E. Dekel et al in “Easy: Engineering High Availability QoS in wServices” [5] do not focus on Web services (even if “wServices” might suggest this) they address performance-aware high availability by using replication. Thus their paper contains a lot of useful information for this thesis. “Easy” presents a solution, for making development of distributed applications faster and cheaper by decoupling it from platform and QoS specifics. Furthermore it provides a quite detailed list of service aspects which have to be taken into account while replicating.

R. Friedman describes in “Caching web services in mobile ad-hoc networks: opportunities and challenges” [6] how to place caching proxy-services within an ad-hoc network in an optimal way, taking into consideration its structure, quality of connections, load of hosts, etc. The component-based separation of program-code and the data it operates on is turned to account for partial caching of service modules, which helps to create a flexible cache replacement mechanism. Furthermore Friedman uses experiences described in “Consistency Conditions for a CORBA Caching Service” [4] to allow choosing multiple consistency guarantees for service-states at run-time.

3 Concept of Solution

3.1 Requirements, Restrictions, and Assumptions

For designing the concept of the solution it was necessary to take a close look at the challenges and problems that will be met. Within dynamic networks in particular, replication and synchronization are far more difficult and complicated than they seem at first sight. In the following chapters, some important requirements and restrictions are listed, which had a great impact on the system's design.

3.1.1 Classification of Failures

As the list of possible failures in Chapter 1.2 demonstrates, Web services can become unavailable due to many reasons. Particularly in ad-hoc networks, where the quality of connection heavily depends on the distance to the next wireless link, services can disappear very quickly. However, since it is most often neither possible to find out the causes for unavailability nor reasonable to distinguish them, all hosts and services have to be classified as *available* or *unavailable*, without paying further attention to the details.

3.1.2 Application in Various Network Environments

Great importance was attached to the ability of running the implemented system, not only within pure ad-hoc networks, but also to enable its application within any network where increased availability is required. For instance one would like to deploy constantly available Web services within a virtual network, belonging to a community spread over the whole planet where hosts are known to be shutdown sometimes. The only strict requirement to the environment at the destination is the usage of TCP/IP and the mandatory ability of unhampered communication between all hosts, excluding firewalls blocking connections from a part of the network. Such a behavior would complicate the monitoring of hosts and services for availability, since the results would be only valid for a certain subset of the network. If this restriction is followed, it is also possible to connect different types of networks to a large one, which shares replicated services.

For this reason, instead of using a hard-coded host discovery, the replicator provides a plug-in interface, and accepts lists of addresses of known hosts in the network, which will be considered while monitoring and placing the replicas.

However, since environments can vary from highly dynamic to more static and as well differ in speed and stability of communication, it is not reasonable to perform the same techniques for monitoring and replica placement in each of them. Instead the parameters of monitoring are tunable and a further plug-in interface is opened for applying custom replica placement mechanisms.

3.1.3 Combination of Various Web Service Platforms

Since the specification of Web services prescribes only SOAP-based communication and sets no restrictions to the implementation there exist numerous middleware platforms for deploying Web services. Examples include Apache Axis [9, 10], the out-dated Apache SOAP [11], IBM WebSphere [17] and Colombo [16], Microsoft .NET [23], and many more. Most of these middleware platforms use Java as the programming language, even though C++ or C# provide a better performance. Nevertheless it might be regarded as a useful feature of the replicator that it makes it possible to move Web services from one platform to another, assuming they support the same programming languages. However, this would mean that one has to deal with the following issues, which depend on the platform.

- *Method of deployment:* does the server use deployment descriptors or does it just need an archive file in a repository directory?

- *Structure of deployment descriptor:* this includes not only the XML structure but also some individual commands for the platform for deployment.
- *External libraries:* where shall they be replaced and how are they registered to be used by the service? What about version conflicts?
- *Security policies:* what platforms support them?
- *Platform-specific functionality:* does the destination host provide the same environment?

In short all these issues can be summarized by the questions, how compatible the different platforms are and if moving services is possible at all. For some combinations it is not possible, due to insurmountable differences although for some, such as Apache Axis & Apache SOAP, it is theoretically feasible. However, this would imply that the replicator mechanism is able to translate the deployment descriptor to the destination format. Unfortunately the Web service would be also forced to discard the application of sophisticated platform-specific functionality, as it is the case with the pluggable request handlers in Apache Axis.

The result would be a Web service deployable on multiple platforms. However, it would only use common functionality provided by all of them. Moreover the replicator would not only have to be aware of all details for deployment on each platform but would also have to able to translate the deployment descriptors. If one considers the advantages and disadvantages of this idea, it becomes quite obvious that the disadvantages outbalance the benefits and so it does not pay off.

As a result the implemented solution only supports Web services based on Apache Axis [9, 10], which was chosen because of its open source license, speed, useful functionality, and clearly arranged architecture.

3.1.4 Control of Replica Placement

For applying an effective replication strategy numerous details have to be taken into consideration, including the current number of running replicas, health and performance properties of hosts, preferences and requirements of Web services, the character of the network environment, etc. All these circumstantialities may have an influence on the decision whether to move replicas and if so, to what location. This raises the question if the Web services should have the opportunity to perform these decisions based on their own logic, or if the whole placement mechanism should be controlled by the replicator system, limiting the service's influence. Allowing only the replicator system to manage it, results in a better control of the network, while letting the Web services decide on their own replicas would enhance flexibility but also enable malicious behavior.

To keep the design simple and clearly structured, the Web service's influence on the replication logic was limited to setting only the performance requirements and the minimum and maximum desired number of replicas within a network. All decisions about moving replicas are performed by the replication logic, which is free to follow or to ignore the Web service's preferences.

3.1.5 Transfer of Data Sources

Web services can operate on all possible data sources, including local variables and objects, files, local and remote databases, socket-based connections to remote resources, other Web services, etc. While moving the working directory of a service with all its files does not pose a problem, moving local databases is almost impossible. First of all the destination host would have to run the same database server. It would have to allow the creation of the database with all involved users and access rights and then the database dump would have to be populated in the new environment. This would result in a huge effort for checking all destination hosts for the necessary infrastructure, let alone the unacceptable usage of network bandwidth for transfer and synchronization. Particularly in situations where hosts appear and disappear, transfer of Web services has to be quick and simple.

As a consequence, movable Web services have to be able to do without local persistence such as files or databases and to save data only in local variables which can be transferred and synchronized easily.

3.1.6 Synchronization

Web services can be either stateless or stateful. Examples for stateless services are proxies invoking other services, processors for translating data from one format to another, and some business processes relying on other services persisting the state. However, most of the Web services used today have an internal state, which is desired to be synchronized within all replicas. To ensure this synchrony one can either use the approach of Primary-Copy [3] (also known as Primary-Backup) or Active Replication (also called State Machine):

Primary-Copy: In this approach one of the replicas is selected to be the primary one and all others are declared as backups. All clients must direct their invocations to the primary-copy which is responsible for propagating the changes to the backups. After a failure of the primary-copy one of the backups becomes the new leader.

Active Replication: Another way how to use replicated services is to send invocations to all of them, waiting for all responses and computing the correct response. This way the states of all replicas are modified the same way and thus kept in a consistent state, assuming the developers avoided using functions with a random behavior. However, the invocation of all replicas has the issue of a necessary suppression of multiple nested invocations, as illustrated in Figure 6.

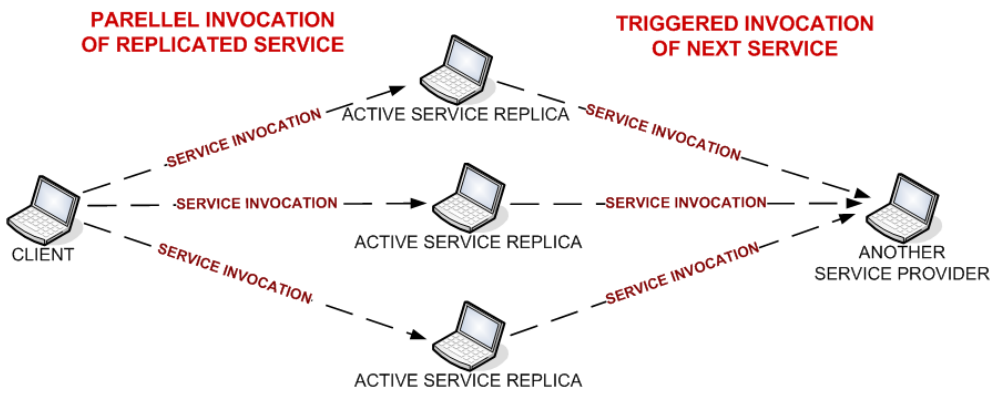

Figure 6: Multiple Nested Invocations with Active Replication. Client invokes all service replicas simultaneously, with each one of them implying another nested invocation of a service. This results in a triple-invocation if no suppression mechanism is applied.

This comparison makes clear that Primary-Copy is the simpler approach, and, therefore, it was chosen for this thesis.

However, the dynamics of ad-hoc networks raise a lot of new problems for synchronization such as splitting and merging networks, as it is illustrated in Figure 4 in Chapter 2.2. This can cause scenarios where replicas of the same service exist in separate networks, leading to multiple divergent service states which cannot be easily combined if the networks merge again. Often this is completely impossible and poses a serious problem for certain Web services which rely on perfect consistency (e.g. ticket distribution). This leads to an unavoidable trade-off in cases where the usage of Web services with strict requirements to consistency is, although possible, simply not reasonable and should be then avoided.

3.1.7 Security

Security is a very important issue for systems where some external code, which is Web services in this case, has to be accepted and executed on the local machine. Since these Web services may contain malicious code, it is absolutely necessary to prevent them from harming the system. Neither Apache Axis nor other platforms provide an adequate solution for this because, usually the administrator of the server is expected to know what he is deploying and tightening the permissions of Web services is hard and often impossible. A reasonable solution to provide a high level of security is to use a public key infrastructure, signing all Web services which were checked and are regarded as secure and to disallow deployment of unsigned services. In the replicator system this is possible by plugging in a new ServiceContentChecker for inspecting the signatures and rejecting unknown services.

The same approach of signing and encrypting data should also be used for securing the communication between the replicator systems. This is especially important since static firewall rules are completely useless in dynamic networks. Therefore it is hard to achieve, to exclude individual hosts from accessing the system. Due to the prototype character of this thesis, these security mechanisms were not implemented.

3.2 Architecture of the Replicator System

In order to increase flexibility and maintainability of the code, the replicator system was designed with high modularity in mind. Figure 7 illustrates the system's architecture, with the individual modules operating on the internal data base of hosts and services:

Internal Data Base: The whole replicator system operates on a data base containing all known hosts and their services (see simplified relation in Figure 8). This data base stores its records in simple but indexed

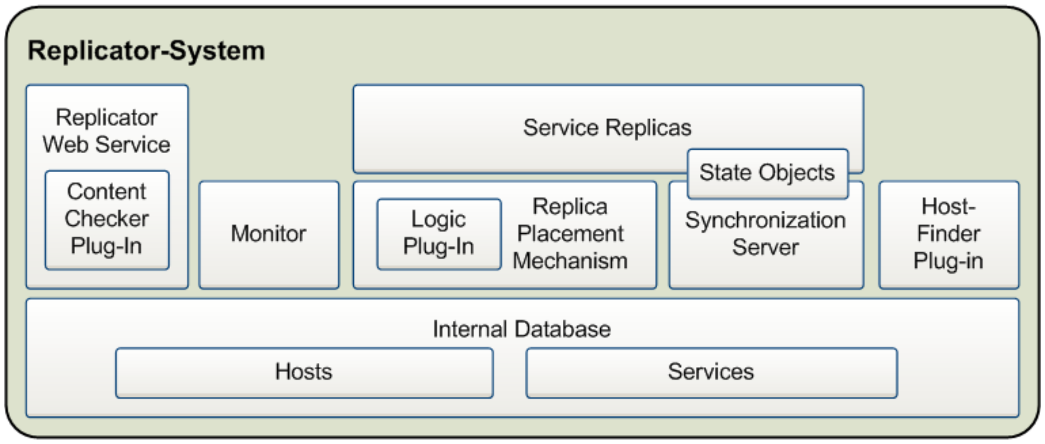

Figure 7: Architecture of the Replicator System

Java-HashMaps. Therefore it is not comparable with sophisticated persistent data bases, such as Oracle, DB2, etc. All other modules of the system perform their manipulations on this data base, which includes retrieval of information, update of host and service properties, deletion of inactive ones, etc.

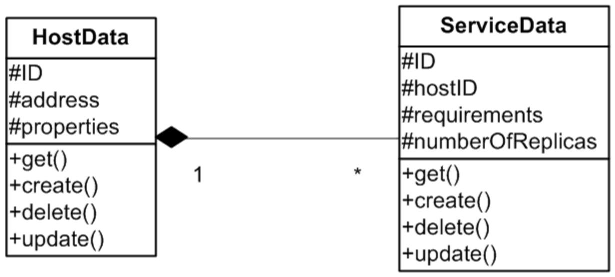

Figure 8: Host-Service Relation in the Data Base

Hostfinder Plug-in: This plug-in interface is used for retrieving lists of known hosts in the network, since hard-coded discovery methods were avoided. New hosts are automatically registered in the data base.

Monitor: To become aware of changes in the availability of hosts and services, it is necessary to monitor them in intervals. As soon as changes have been detected the data base is updated.

Replicator Web service: Sending and receiving of Web service replicas is done via a Web service too. Furthermore it provides information about the local host and its services, which is retrieved periodically by the monitors. This Web service is the passive part of the system, waiting for commands without invoking anything independently.

Replica Placement Mechanism: This module contains the whole replication logic. Viz the decision whether a service has to be sent or removed anywhere in the network are made in this module. It accepts custom replication logics as plug-ins if a different behavior than the default one is desired.

Synchronization Server: Web services declare their state objects at the synchronizer, which is responsible for propagating the changes to all service backups.

Detailed documentation of the functionality can be found in the subsequent chapters. Chapter 4 explains the replication of Web services, which consists of the Replicator Web service, the monitor and the replica placement mechanism. The synchronization of stateful Web services is documented in Chapter 5.

3.3 Simple Replicator Protocol

All communication with the outside world is done via the Replicator Web service or a small server using a simple TCP-based protocol. The advantage of the Web service is the convenience of an easy serialization and transfer of even complex data. Therefore it is used for providing information about the local services, host properties, etc. as well as more sophisticated functionality for cooperation of multiple monitoring hosts. However, the main disadvantage of Web services is their inferior speed. Thus they are useless for communication which has to be fast and with as little overhead as possible. Considering the synchronization of services as example, where data has to be exchanged frequently and quickly, Web services are not an adequate technique.

To fulfill this need for fast communication, a small server is provided, bound to a fixed port and communicating via the light-weight TCP-based Simple Replicator Protocol. This protocol works in a command-response manner (see Listing 7) with the client sending commands with optional arguments and the server responding with a status code, an optional response payload, and a single dot marking the end of the response.

Listing 7: "Command and Response Protocol Sample. (“>” prefixes client-to-server communication. “<” prefixes server-to-client communication.)"

```
> command arg1 arg2
< 100 OK
< response line 1
< response line 2
< more response lines ...
< response line n
< .
```

Since a command may be malformed or cause other errors, various status responses were specified to provide additional information explaining the error, as shown in Listing 8.

Listing 8: "Possible Responses from the Server"

```
100 OK
200 Connection closed
210 Too many connections
300 Client error
```

```
400 Unknown command
400 Illegal arguments
500 Internal server error
```

1xx Success.

2xx Closing the connection. This can be caused by too many simultaneous connections or simply by exiting the session.

3xx Client errors. These errors are thrown if the command would cause inconsistencies, an invalid state, or other errors.

4xx Command errors. Either an unknown command was called or the arguments were invalid (e.g. wrong number, malformed).

5xx Server errors. Indicating a corrupted configuration of the server.

Every module in the replicator system is free to attach additional functionality to the server as a plug-in. For instance, the synchronization server provides commands for setting and retrieving the internal state of a service. Listing 9 shows a sample communication using these commands:

- `Lines 1-5`: list all state objects of service "testService". The response contains the names of the objects, including additional information.
- `Lines 6-9`: retrieve the serialized form of object "o1".
- `Lines 10-13`: retrieve the serialized form of object "o315". This causes an error since "o315" does not exist.
- `Lines 14-16`: close the connection.

Listing 9: ”Sample Communication Reading Service State Object.”

```
> LISTSTATE testService
< 100 OK
< o1 8 3056
< o2 19 18273
< .
> GETITEMS testService o1
< 100 OK
< <43|object1|0|java.lang.String|rO0ABXQAAmEx>
< .
> GETITEMS testService o315
< 300 Client error
< Error (java.lang.IllegalArgumentException): Unknown ID(s).
< .
> QUIT
< 200 Connection closed
< .
```

4 Description of Replication

Performing an effective replication mechanism is a complex task. Since it has to be clearly structured, it is unavoidable to split the whole functionality into more or less independent modules. Figure 9 shows the three modules of the system responsible for replication of Web services, which are explained in-depth in this chapter.

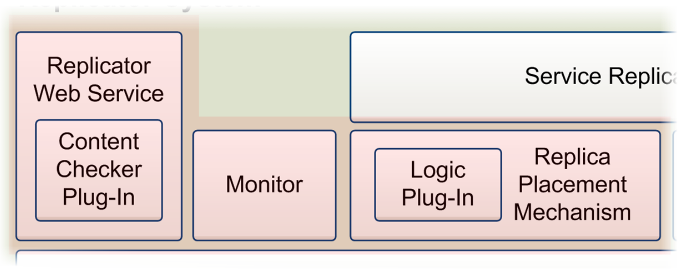

Figure 9: Modules for Replication of Web Services

The system is composed of (a) the Replicator Web service, (b) the monitor, updating the current view of the network periodically, and (c) the the replica placement mechanism, electing leaders for control of replication and finally managing all controlled Web service replicas.

4.1 Replicable Web Services

4.1.1 Web Service Archives

The structure of the Web service's archive file is strictly predefined (see sample in Listing 10) with all important files at fixed locations. The replicator does only accept Jar files containing a directory named “webservice” with the deployment descriptors and an additional file named “info”, specifying the preferences for replication. Furthermore the archive is checked whether it contains the Web service's class file, specified in the deployment descriptor.

After a successful validation the archive file is registered as a Java library at the class loader of Apache Axis.

Listing 10: "Sample Structure of Service Archive"

```
/webservice/info
/webservice/deployment.wsdd
/webservice/undeployment.wsdd
/packagepath/SomeService.class
/...additional files...
```

4.1.2 Hibernation of Web Services

Depending on the size of a Web service's archive its transfer from one host to another can be very costly. Especially in ad-hoc networks where replicas have to be moved quite often this can cause a high usage of network bandwidth and slow down the whole replica placement mechanism. This happens mainly when a Web service requires a fixed number of replicas deployed in the network and when hosts are disconnecting and reconnecting to the network frequently. To keep this number, surplus replicas have to be deleted and if too few exist: new ones have to be deployed, implying the transfer of the whole archive file. To cushion these dynamics, Web services are free to specify their desired number of replicas as a range instead of a fixed value, decreasing the number of transfers this way. A more significant decrease can be achieved by following the approach which is used by operating systems to quickly recover the machine after a shutdown. That is hibernation.

The idea of hibernating Web services which are currently not needed, is to keep them installed on the host but in an undeployed state. When the number of replicas in the network sinks below the minimum value, the replica placement mechanism can check for hibernated services and wake them up, which requires only a simple command to be sent instead of the whole archive. The state diagram in Figure 10 illustrates the possible deployment states of

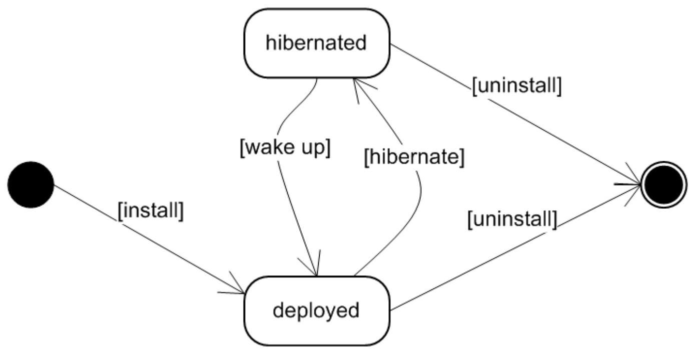

Figure 10: Deployment States of Web Services

a Web service, which can be deployed, hibernated, or not installed at all marked as the starting and ending state.

4.1.3 Deployment Hooks

As Figure 10 demonstrates, the deployment state of a Web service can change infinite times. Often it is useful for Web services to be informed about these state transitions. This is possible by overwriting the hook methods “onInstall()”, “onHibernate()”, “onWakeup()”, and “onUninstall()” of the abstract class “Service” which is the mandatory base class of every replicable Web service (see Listing 11). The replicator does not set any restrictions to the functionality of the hooks and by throwing an Exception the whole state transition is abandoned, and the service returns to the former state. For this reason all hooks are executed before the state transition is put into effect, except for “onInstall()” because of necessary registrations which have to be performed before the hook. But also in this hook an Exception revokes the transition, deleting the whole Web service again.

Listing 11: ”Abstract Java Class “Service””

```
public abstract class Service {

    public Service() {
        super();
    }

    //executed after installation
    public void onInstall() throws Exception {}

    //executed before hibernation
    public void onHibernate() throws Exception {}

    //executed before waking up
    public void onWakeup() throws Exception {}

    //executed before uninstallation
    public void onUninstall() throws Exception {}

    public abstract String getServiceID();
}
```

4.2 Host Properties and Service Requirements

Web services can differ heavily in generating load on the system depending on their implementation and frequency of invocations. To optimize the placement of their replicas they are allowed to specify their requirements, including the desired amount of free memory, CPU power, network bandwidth, and disk space. These data are compared to the system properties of all hosts in the network while choosing a new destination for deploying a new replica.

To calculate a host's system properties is not a difficult task, although not completely possible within pure Java code. Instead, these values can be retrieved for example from Linux/Unix utilities, such as "vmstat", "df", etc. and propagated periodically to the replicator via an extension to the Simple Replicator Protocol. However, it is far more complicated to calculate the actual requirements of a Web service. In fact, it is only possible by using sophisticated Web service monitoring techniques, which would go beyond the scope of this thesis. Instead the Web services declare estimated values which can often deviate from the actual requirements. Nevertheless they are adequate for improving the placement of replicas. Important for distributing replicas in a network is also the estimated battery usage. Hosts with a low battery will most likely disappear in the near future and, therefore, it is wise to move their deployed replicas to other destinations.

The matching of service requirements to host properties is an important feature of the whole system, aiming at balancing the load and preventing bottle necks. Modules such as the monitor or the replica placement mechanism use it for sorting potential destination hosts to optimize the usage of resources in the network.

4.3 Replicator Web Service

The Replicator Web service is a special case of a module. It works completely in passive mode and is invoked mainly by other hosts to manipulate the deployment state of Web service replicas, to retrieve health properties, or to exchange the currently monitored view of a network. Deployed within the Apache Axis SOAP container [9] it operates in the same environment as all replicas. This makes it possible for other hosts to find out about the destination environment for placing new replicas. For example it is necessary to know whether all required code libraries (Jar files) and classes are available or have to be installed first before deploying a replica.

A very important feature of the Replicator Web service is the ability to perform hot deployment of Web service replicas, viz installing them without restarting the containers illustrated in Figure 11. Usually, Web services are deployed by first copying the archive file and all necessary libraries into the designated library directory of one of the containers. Then the containers are restarted in order to reread all libraries. To allow on-the-fly installing and uninstalling of Web services, it is necessary to extend the class loader of Apache Axis by plugging in a new loader which is able to register and unregister libraries at run time. This way hot deployment becomes possible.

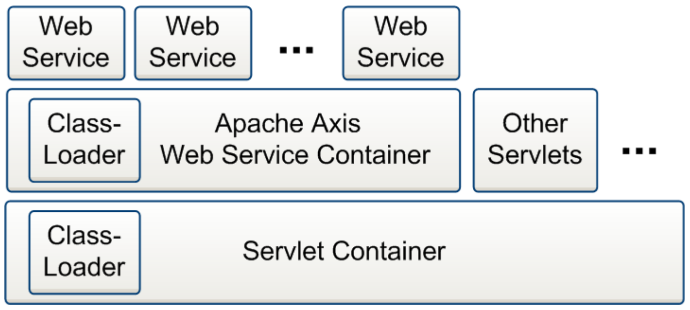

Figure 11: Web Service Container Stack

The Replicator Web service provides the following functionality:

Manipulation of Web service replicas: This primarily includes hot deployment, undeployment, hibernation and waking-up of Web service replicas. Furthermore, it is possible to retrieve lists of classes and libraries available in the Web service environment and to install and register new libraries if this is necessary.

Information about the local host and its Web services: The Replicator Web service returns also serialized instances of the classes "HostData" and "ServiceData" describing the local host and its Web services. These classes contain all necessary information about the state, such as lists of hibernated and deployed Web services, system health properties, service requirements and preferences, etc.

Exchange of network state information: The internal data base (see Figure 8 in Chapter 3.2) contains the current view of the network, which is updated periodically by querying monitors. The Replicator Web service offers to the possibility to retrieve the content of the data base and to update it, which is especially important for monitors dividing the network into groups, observing the own group and propagating changes in the state to all other monitors.

Manipulating the state of replicas and content of the internal data base raises the problem of possible misuse or missing security. This can be solved by using the WS-Security and WS-Trust extensions to sign and encrypt Web service based communication and to build a trustworthy infrastructure.

4.4 Monitoring

Most of the computations, such as leader elections or the entire replica placement mechanism, are based on a global view of the network. And since the nodes cannot be expected to send notifications about their failures, it is necessary to actively monitor the state of the whole network in intervals to have always a quite up-to-date view of it. This task can be performed by every node itself, by checking the availability of the others, causing a high usage of network bandwidth acceptable only for very small networks. However, the bigger the network gets, the more scalability becomes important. To reduce the load caused by monitoring, it is advantageous to elect a few hosts, responsible for checking the availability of nodes and services and to let all other hosts query them to retrieve the view of the network. Especially if the monitors divide the network into groups as shown in Listing 12, and each group is checked by only one monitor, the scalability is improved significantly.

The monitoring module of the replicator system is capable of both being an active monitor and passively retrieving the actualized state of the network from other monitors. Regarding the current global view of the network, each host can determine whether it is expected to perform active or passive

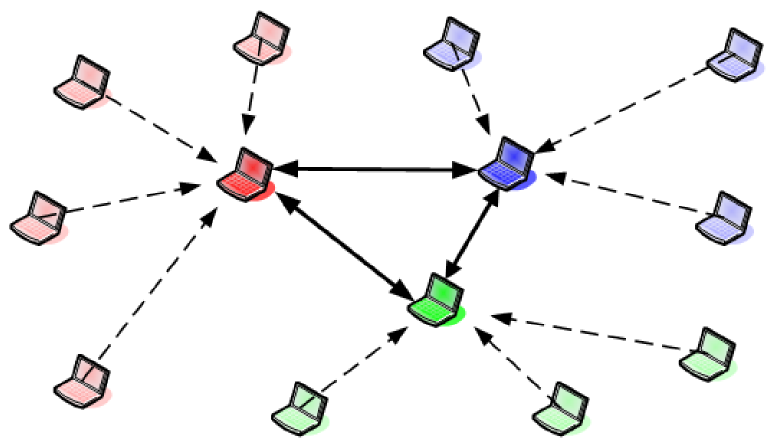

Figure 12: Distributed Monitoring of Network State. Each group is checked by a monitor exchanging the information about its group with other monitors to have a complete view of the network.

monitoring in the current cycle by applying the following algorithm:

Listing 12: "Election of Monitoring Hosts (Pseudocode)"

```
monitoring() {
// bootstrap: list of hosts exists, but no information about
// their states or Web services is available -> retrieve it
    retrieve complete view from random host
// main loop, intervals depend on size of the network
    loop in intervals {
        sort hosts by free bandwidth
// election: number of monitors depends on size of network
        monitors = list of fastest hosts
// is localhost one of the fastest hosts => monitor ?
        if (monitors contain localhost) {
            active_monitoring(monitors)
        } else {
// fetch a monitor, try to reuse it in the next cycle
// retrieve the view of the network from it
            mon = random monitor
            passive_monitoring(mon)
        }
    send event notification to leader elector
    }
}
```

This algorithm uses the most recently retrieved view of the network to find out whether it is expected to monitor a part of the network actively. Hence it accepts inconsistencies which happen due to quick changes in the network and which occurred after the last monitoring cycle. It tries to swing again into a consistent state within the next round. Such a behavior is necessary since the dynamics of mobile ad-hoc networks make it impossible to rely on perfect consistency and, therefore, each host has to perform its computations to the best of his current knowledge of the network.

4.4.1 Active Monitoring

The main functionality of the active monitoring procedure is to find out which hosts have to be monitored, to check them, and then to send their states to the other monitoring hosts, which are in turn responsible for their own groups and propagate their states as well. The invocation of "active_monitoring(monitors)" in Listing 12 passes as argument a list of hosts which are believed to be the current monitors. Of course this list may deviate sometimes from the actual state in the network but, again, this is an unavoidable trade-off in ad-hoc networks. In theory this inconsistency can result in an unbalanced monitoring of the network, with some hosts being monitored more often and others not checked at all since the segmentation of the network into groups is based on this list. However, in practice this scenario is most improbable and disappears completely in less dynamic network environments.

Listing 13: "Active Monitoring (Pseudocode)"

```
// expects the list of all monitors of the network as argument
active_monitoring(monitors) {
    pos = position of localhost within sorted monitors
    num = number of monitors
// current group of hosts to monitor
    mygroup = initialize empty list of hosts
    loop for all hosts in network {
// ID of host decides whether it is member of this group
        if (host.id%num == pos) {
            add host to mygroup
        }
    }
// do the actual monitoring
    loop for all host in mygroup {
        check host
        check services
    }
// exchange data with other monitors
    loop for all hosts in monitors {
// send only changes since last exchange to save bandwidth
        send changed state of mygroup to host
// all monitors must see the same nodes in the network
        send list of new node addresses to host
    }
}
```

The segmentation of the network into groups is done by comparing the ID number of the hosts to the position of the local host within the monitors. This is a critical part of the algorithm since it relies heavily on the accuracy of the passed list of monitors. This accuracy is improved continuously by exchanging monitored states and addresses of all known hosts between the monitors.

4.4.2 Passive Monitoring

If a host was not elected to perform active monitoring, it has to retrieve the global view of the network from a monitor. The functionality of the passive procedure is much more simple than the active monitoring. The algorithm tries to retrieve the view only from one monitor in an incremental manner. This means that the first request returns the whole view and all subsequent ones retrieve only the changes, saving bandwidth this way.

Listing 14: "Passive Monitoring (Pseudocode)"

```
// retrieves the current view of the network from monitor
passive_monitoring(monitor) {
    if (monitor was used in the last loop) {
// save bandwidth
        retrieve changed data
    } else {
// get the whole view of the network
        retrieve all data
    }
// inform monitor of new nodes
    send list of new node addresses to monitor
}
```

4.5 Replica Placement Mechanism

To have the current view of the network and the Replicator Web service running on the hosts, means to have already an infrastructure for managing the placement of Web service replicas. In the replicator system this placement mechanism is split into the modules of leader election and replication logic. Figure 13 shows the sequential alignment of execution of both modules and the monitor. Instead of running as independent threads, these modules wait until the predecessor has finished, look at the new state and decide whether further modifications are necessary. This sequence is more effective since the leader elector waits for the monitor and operates on the updated view of the network. Furthermore the replicator logic needs to know whether some replicas are controlled by the local host.

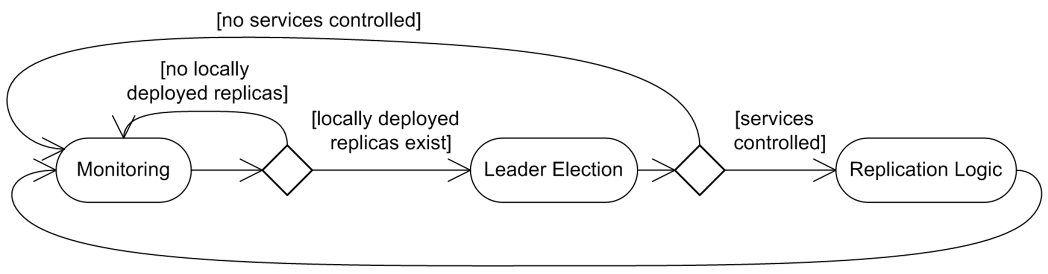

Figure 13: Execution Sequence of Replicator

4.5.1 Leader Election

In order to keep the strategies for placing replicas simple, it is essential to allow only one host to control the movements of a particular service. This leader is unique in the network and every host running this service has to apply the algorithm shown in Listing 15 to find out whether it is expected to control it.

Listing 15: "Election of Replication Leader (Pseudocode)"

```
// run after monitor has finished
loop for each deployed web service {
// list all replicas, including the local one
    replicas = list of its replicas in the network
    if (only one replica exists) {
// only one means only on localhost
        localhost is leader
    } else {
// more than one requires election
        leaders = collect declared leaders of each replica
// the most popular one will be accepted
        sort leaders by popularity
// do more than 2 leaders share the first place ?
        if (more than one most popular leader exists) {
// try to put leaders on monitors to be earlier
// informed about state changes
            sort hosts by bandwidth and monitoring status
            accept the fastest one
        } else {
            accept the most popular leader
        }
    }
}
send event notification to replication logic
```

The effect of this algorithm is that the host on which the service was initially deployed stays the leader as long as it is available. Since all potential leaders execute this algorithm after every monitoring cycle, a new leader can be elected quickly in case a failure or disconnection of the old one has been detected. Another important feature is the merging of multiple running leaders within one calculation cycle. This becomes necessary if an ad-hoc network was split into sub-nets, these independent sub-nets elected their leaders and after a time the networks merged again. In this case the algorithm accepts the most popular leader to minimize the necessary change-overs.

4.5.2 Replication Logic

Having determined which replicas are controlled by the local host, the leader elector notifies the replication logic which is responsible for managing them. This includes checking whether the proper number of replicas is deployed, installing/waking up of replicas in case too few exist, and deleting/hibernating surplus ones. Furthermore, the logic is expected to pay attention to the system properties of hosts and preferences of services while performing modifications. For instance, it makes sense to move replicas away from hosts which only have little time left due to low batteries.

As explained in Chapter 3.1.2, network environments may vary extremely and therefore may have different requirements to the replicator system. Hence it is not reasonable to prescribe defined replication strategies but to permit the application of customized ones as plug-ins. For instance, one would like to adapt the P-Grid [1] peer-to-peer platform with its sophisticated and flexible management system to distribute Web services. Such a plug-in must be specified in the configuration file and it has to extend the abstract Java class “AbstractReplicationLogic” from the following Listing:

Listing 16: ”Java Class “AbstractReplicationLogic””

```java
public abstract class AbstractReplicationLogic {

    private List<Command> commandList;
    private Set<InetSocketAddress> lockedHosts;

    void sendService(String serviceName, HostData host) {
        Command cmd=new SendCommand(serviceName,host);
        // ignore redundant commands and locked hosts
        if (!commandList.contains(cmd) &&
                            !lockedHosts.contains(host)) {
            commandList.add(cmd);
            lockedHosts.add(host);
        }
    }

    void deleteService(String serviceName, HostData host) {
        // similar to sendService() ...
    }

    void hibernateService(String serviceName, HostData host)..

    void wakeupService(String serviceName, HostData host)...

    // observer function, notified by leader elector
    public void update(Observable o, Object arg) {
        if (o instanceof LeaderElector) {
            WorkerThread thread=new WorkerThread(commandList);
            commandList.clear();
            thread.start();
        }
31      }
32
33      private class WorkerThread extends Thread {
34          public void run() {
35              // execute commands sequentially and unlock hosts
36          }
37      }
38
39  // more declarations ...
40  }
```

This abstract class provides the basic functionality to perform convenient replica placement without taking care of details such as SOAP communication or packing Web services into archive files. Unfortunately this module is not safe from inconsistencies in dynamic networks either. Again, splitting and merging networks pose a problem since they can cause a service to be controlled by more than one host simultaneously for a short time. Although the leader elector is able to correct it within the next calculation cycle, a situation may be caused where coexisting leaders want to perform some modifications concurrently. Consequences could be collisions or another inconsistent state. To avoid this, commands are postponed to the next cycle after then being executed immediately. This opens the opportunity to check in the next cycle whether the local host is still the leader and the command can be executed or whether the leader elector has already corrected this inconsistency and the command has to be canceled. Although this makes the whole placement mechanism more lazy, since modifications have to wait until one cycle has been performed, this approach is worth to be used due to the prevented inconsistencies. The abstract class in Listing 16 implements this approach by queuing the commands (lines 7-13) and starting a thread responsible for executing them in the next cycle (lines 27-29). The actual modifications are implemented within the individual “Command” classes.

In case no custom plug-in has been specified a built-in replication strategy called “SimpleReplicationLogic” is used. As Listing 17 shows, its behavior is very simple but adequate for the usual needs.

Listing 17: ”Simple Replication Logic (Pseudocode)”

```
// run after leader elector has finished
loop for each controlled web service {
// which hosts are better suited to this service?
    sort hosts regarding service preferences
// need more running replicas ?
    if (number of replicas too low) {
        if (services are somewhere hibernated) {
            wake up on fastest hosts
        } else {
            send new replicas to fastest hosts
        }
        synchronize new replicas
    }
// too many replicas? -> delete ...
    if (number of replicas way to high) {
        delete surplus replicas on slowest hosts
    }
// ... and hibernate
    if (number of replicas slightly to high) {
        hibernate replicas on slowest hosts
    }
// avoid hosts with only little time left
    if (replicas exists on transient hosts) {
        move services to other/fastest hosts
        synchronize new replicas
    }
}
```

5 Description of Synchronization

If one considers a replicated Web service for distributing tickets as example, where each invocation increments the internal counter and therefore also changes the subsequent responses, the necessity for synchronization of this counter between all replicas becomes obvious. Without synchronization a change-over to a backup service would result in using an old state and thus in distributing tickets which have maybe already been handed out before. To solve this problem in a convenient way for the Web service developers, the replicator system extends all stateful services with functionality to declare their internal state and to distribute changes to all other replicas.

Usually Web services keep their internal state saved in local variables, files, data bases, resources on remote hosts, etc. Whereas synchronizing variables is comparatively simple, synchronizing the changes in files and data bases is much more complex, particularly in cases when a source gets out of sync for some time and needs to be completely resynchronized. However, as explained in Chapter 3.1.5, the replicator system does not support the use and transfer of any data source except for local variables. Hence this restriction affects the storage of the internal state too.

Even though synchronization of replicated Web services is absolutely necessary, it sometimes causes the problem of a high usage of network bandwidth. The actually produced traffic depends on the frequency of invocations, the number of running replicas, and the size of the state. While mobile ad-hoc networks are not suited for synchronization of services with large amounts of data, this problem disappears on faster networks. Therefore the synchronization server does not set any constraints on the exchanged data. However, the Web services should be chosen wisely, depending on the network environment.

5.1 Synchronizable Web Services

The synchronization mechanism is based on Web services declaring their state objects and notifying the synchronization server to distribute the changes to the other replicas. To be able to use this synchronization functionality every service must extend the abstract base class “SynchronizedService” which in turn is an extension to the mandatory base class “Service” (see Listing 11 in Chapter 4.1.3).

Listing 18: ”Abstract Java Class “SynchronizedService””

```java
public abstract class SynchronizedService extends Service {

    // synchronization server
    protected static Synchronizer synchronizer=null;

    public SynchronizedService() {
        super();
    }

    // hook, executed after installation
    public void onInstall() throws Exception {
        super.onInstall();
        registerAtSynchronizer();
    }

    public void registerAtSynchronizer() throws Exception {
        initializeStateObjects();
        synchronizer=Synchronizer.getLocalSynchronizer();
        synchronizer.registerService(this);
    }

    // hook, executed before uninstallation
    public void onUninstall() throws Exception {
        super.onUninstall();
        synchronizer.unregisterService(this);
    }
27
28      // called by service operations after state has changed
29      protected final void synchronizeStateObjects()
30      throws IOException, Exception {
31          synchronizer.synchronize(this);
32      }
33
34      protected abstract void initializeStateObjects();
35
36      public abstract HashMap<String,IStateObject>
37                                          getStateObjects();
38  }
```

Extending this base class has the effect of an enforced registration at the synchronization server to access its functionality:

- `Lines 11-26:` The Web service is automatically registered and unregistered at the local instance of the synchronization server. This is realized via the hook methods "onInstall()" and "onUninstall()" which are triggered during the deployment and undeployment of the service.

- `Lines 29-32:` It is the task of the Web service to notify the synchronization server about a changed internal state, which will be propagated to all other replicas of this service. This is done by simply calling "synchronizeStateObjects()", which is a non-blocking method. This means it performs the propagation in the background, without delaying the service's response.

- `Lines 36-37:` Furthermore the service has to declare a map containing all its state objects in order to give the synchronization server access to its state at any time. This is especially important while propagating a state or receiving updates from other replicas.

In Appendix A a sample implementation of a synchronized Web service can be found.

5.2 State Objects

A state object can contain its information in any possible form including variables of primitive or complex types, data composed of other objects (e.g. lists and maps), etc. The main requirement it has to fulfill is to be serializable and deserializable. Otherwise it cannot be transferred to other hosts. Listing 19 contains the Java interface “IStateObject”. It defines the most important functions, including the serializer/deserializer methods, and a function to check whether the state object has changed and should be synchronized. The manner of accessing and manipulating the data of the state objects is up to the individual implementations, which must extend this interface.

Listing 19: ”Java Interface “IStateObject””

```
public interface IStateObject {

    public String serialize() throws Exception;

    public void assign(String serializedData) throws Exception;

    public boolean hasStateChanged();

    // more method definitions...
}
```

The replicator system provides two default implementations of this interface, called “FieldSetterStateObject” and “MethodSetterStateObject”, which access the actual state data either by manipulating it directly or via getter/setter methods. Both classes accept optional serializer and deserializer methods to transform the objects into strings and vice versa. If a declaration of these methods is omitted the transformation is done by using Base64-encoded Java-ObjectStreams. The following listing contains a short code snippet which serializes a String by using a “FieldSetterStateObject” without a custom serializer:

Listing 20: "Serialization Example"

```
// global declaration of string object
public static String s=new String("abc");

// declaration of state object and serialization
FieldSetterStateObject testobject=
    new FieldSetterStateObject(this.getClass().getField("s"));
System.out.println(testobject.serialize());
```

This code returns the following string as output:

```
<42|s|0|java.lang.String|rO0ABXQAA2FiYw==>
```

The first field specifies the length of the string to make it easier to parse sequences of serialized objects. After that comes the name of the state object, followed by an integer telling how often this object has already been changed. This is particularly important for adjusting the states of different replicas. The last two fields are the Java class name and the actual serialized object, encoded in Base64.

Serialization and deserialization of state objects is usually performed by the synchronization server only. Web services just have to declare a list of state objects, such as "testobject" in the last listing, and to notify the synchronization server about relevant changes which have to be propagated to the other replicas.

5.3 Propagation of Updates

In Chapter 3.1.6 the approaches of Active Replication and Primary-Copy are compared, and it becomes obvious that Primary-Copy is the simpler approach and therefore better suited to be applied in dynamic networks. This technique is based on a primary copy of the replicas which serves all requests and synchronizes all backup copies, as illustrated in Figure 14. An implementation of this technique within the replicator system implies that for all replicas of a particular service, a primary one has to be elected. It accepts all invocations solely and is responsible for propagating changes of the internal state to the backups. Fortunately, this election is already done in the replica placement mechanism while unique leaders for the control of each particular group of replicas are elected.

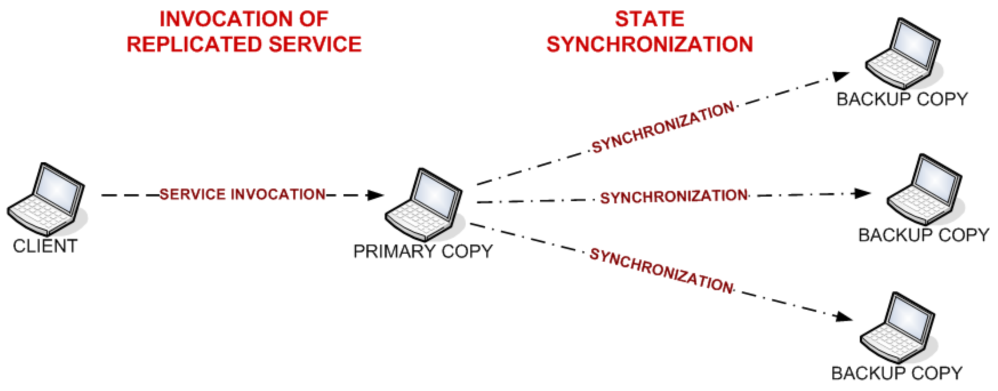

Figure 14: Synchronization of Web Services using Primary-Copy

Once a Web service has finished processing a request which changed its internal state, it is necessary to synchronize these changes with the other replicas, by calling the method “synchronizeStateObjects()” from the base class “SynchronizedService”. This method notifies the synchronization server to contact all replicas, to compare the states, and to check if synchronization is necessary. Obviously this communication is performed very often and should therefore be light-weight and fast, disqualifying SOAP this way and calling for the TCP-based Simple Replicator Protocol introduced in Chapter 3.3. The synchronization server extends the functionality of the protocol by the following commands:

- `LISTSTATE <servicename>` lists all state objects of a service, including the serial stamp number and the hash sum for each object. The serial stamp is incremented each time the state object is changed, thus allowing to compare the activity of the objects to each other. The hash sum is used to find out whether objects are equal or need to be synchronized.

- `GETITEMS <servicename> <itemname ...>` returns a sequence of serialized items, defining the state objects.
- `ASSIGNITEMS <servicename> <serializeditems>` accepts a sequence of serialized items which will be assigned to the corresponding state objects.

Listing 21 contains a sample communication using these commands. First of all a list of state objects of a service named "syncTest" is retrieved. It contains the objects "i" and "s", their serial stamps, and hash sums. After that the state object "s" is retrieved in a serialized form and, finally, a new value is assigned to object "i" incrementing automatically its serial stamp number to 4.

Listing 21: "Sample Communication for State Manipulation"

```
> LISTSTATE syncTest
< 100 OK
< i 2 123
< s 4 96354
< .
> GETITEMS syncTest s
< 100 OK
< <94|s|4|java.lang.String|rO0ABXQAK2FiYy4uLi4uLi4uL...4uLi4=>
< .
> ASSIGNITEMS syncTest <136|i|4|java.lang.Integer|rO0A....AB9>
< 100 OK
< .
```

5.4 Consistency Problems in Ad-hoc Networks

Propagation of a changed internal state is usually done by comparing the remote list of state objects with the local one and refreshing all objects which are out of date. This simple behavior is fine for networks where exactly one replica is always the primary one and therefore always has the most current state. However, the state synchronization task has to deal with the same consistency problems as the replica placement mechanism, since ad-hoc networks are highly dynamic and the primary copy of a group of replicas is elected by the replica placement mechanism. These problems include scenarios where the primary copy disappears and a new one is not elected until the next calculation cycle or situations where more than one primary copy exist concurrently. Although both types of inconsistency are corrected in the next cycle, the question raises how clients have to deal with such situations and how this affects the consistency of states.

Especially the scenario with multiple primary copies poses an almost insolvable problem if perfect consistency is desired. This becomes clear in the following example, where a Web service for distribution of tickets is deployed.

1. The replica placement mechanism spawns new replicas which are perfectly synchronized.
2. A laptop which establishes this network by acting as a single router between two subnetworks is suddenly shut down. As a consequence two autonomous networks exist, which have absolutely no information about each other any more. However, both are running replicas of the ticket service.
3. Both networks act completely individually and manage their replicas, which are synchronized among themselves but not with the ones from the other network. Moreover, clients are changing the states of the services continuously by requesting new tickets.

4. The connecting router appears again and merges the subnetworks.
5. Until the next election cycle the newly merged network contains two primary copies with two states that have to be merged. Furthermore clients which have received the same tickets exist in the same network at the same time.

In this scenario the internal states of the ticket services can be adjusted adequately by accepting the higher ticket counter as dominant and replacing the other. However, clients which received their tickets while the networks were split and hence have other clients with the same tickets in the network pose a serious problem. The tickets they got could have been used for other calculations and, therefore, it is neither reasonable nor possible to revoke them in order to establish consistency again.

This example demonstrates the dilemma of using replicated Web services which require strict consistency within ad-hoc networks. Unfortunately, their application in dynamic networks is not reasonable at all since perfect synchrony of states cannot be guaranteed. The synchronization server rather solves such conflicts by comparing the states, electing the one which was modified more often as dominant, and replacing all others with it. This election is done by summing up all serial stamp numbers of all state objects and declaring the service with the highest sum as the dominant one. The idea of this method is to keep the state which was used to serve most of the requests with a high probability.

5.5 Invocation of Web Services

When a client wants to invoke a replicated Web service, it has to find its primary copy first. Unless it has its own replicator system running, and thus also the internal data base with the updated view of the network, it is necessary to retrieve the location from another source. An ideal source is one of the hosts running a replica of the desired Web service, since it actively elects the primary copy. Hence its information is more up-to-date than the one from any other hosts, which are notified about it via monitoring. However the client can be confronted with situations where either the primary copy is not available anymore or too many exist concurrently because two networks have merged and a common leader has not been elected yet. To disburden the application developer of handling these difficulties, the replicator system provides a simple tool which takes care of finding the primary copy and returns the corresponding WSDL file. Hence, an invocation of a replicated and synchronized Web service is done by performing the following steps, illustrated in the sequence diagram in Figure 15:

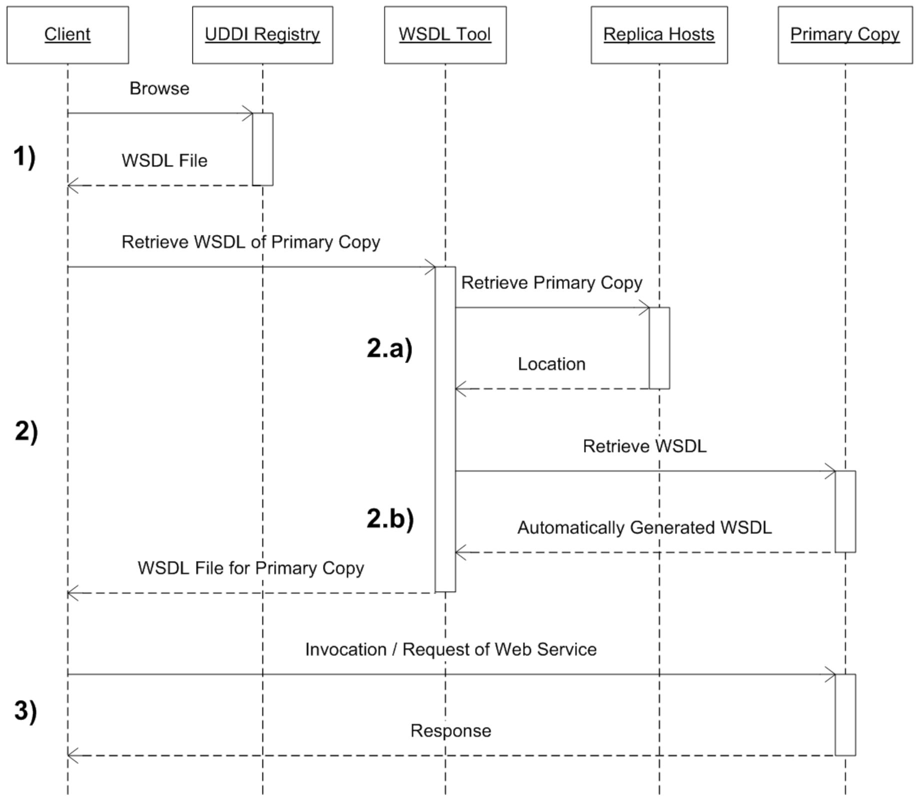

Figure 15: Sequence Diagram for Retrieving WSDL File of Primary Copy

1. The tool accepts either the name of the Web service replica plus the location of a host running it or the service's complete WSDL file retrieved from a UDDI registry [8] as argument. It is not the task of the tool to query the registry. This has to be done by the client.

2. The tool's functionality is to provide a WSDL file pointing to the primary copy. This includes:

   (a) contact the replica host and get the locations of the other replicas. Moreover, it has to collect the location of the primary copy from all replicas, choosing the most popular one to cushion inconsistencies. This communication is done via the Simple Replicator Protocol and is therefore fast and light-weight. Furthermore, the locations of all replicas are cached and updated during each run. This way it is possible to follow the movements of a replicated Web service without querying the UDDI registry continuously.

   (b) contact the primary copy host and retrieve the automatically generated WSDL file from the Apache Axis SOAP Container.

3. The client can pass this WSDL file as an argument to the Apache Web Service Invocation Framework [12] and invoke the proper Web service replica.

6 Evaluation

6.1 Practical Application of the Replicator System

The replicator system needs a predefined structure of its working directories with full read and write access in order to work properly. Listing 22 shows the most important directories. The configuration files are located in “conf”; “lib” is the destination directory for all additional libraries needed by the Web services. They are in turn unpacked to the “repository” directory. All logging output is saved in the “logs” directory; “work” contains the working directory of the Jetty HTTP Servlet Server [19] and the Apache Axis Servlet [9].

Listing 22: ”Directory Structure of Replicator System”

```
./conf/
./lib/
./logs
./repository/
./work/jetty
./work/webapps/axis
```

The path to a directory, which contains all these subdirectories, has to be passed to the replicator system. This is done during the invocation via the mandatory Java system property “wsreplication.workingdirectory”:

```
java -cp $CLASSPATH -jar replicator.jar \
     -Dwsreplication.workingdirectory=/repldir
```

The first task of the bootstrapping module is to read in and process the main configuration file located at “conf/main.conf” (see Listing 23). This file contains the most important options, which can be specified by the user. For instance it is possible to set the maximum number of concurrent connections, to specify the plug-ins for the replication logic and the security checker, to set the port of the HTTP server, etc.

Listing 23: "Main Configuration File of Replicator System"

```
wsreplication.replicationlogic.plugin =
wsreplication.server.maxcon = 256
wsreplication.service.checkerclass =
wsreplication.syncro.sticky = true
jetty.port = 8070
```

Once the configuration has been approved as valid and the replicator system continues with the bootstrapping, it logs the most important configuration and system properties to the screen, as shown in Listing 24.

Listing 24: "Logs of Replicator System During Bootstrap"

```
16:22:35,359: Configuration directory : /repldir/conf
16:22:35,394: Reading from 'main.config'.
16:22:35,394: Repository directory : /repldir/repository
16:22:35,394: Working directory : /repldir/work
16:22:35,395: Library directory : /repldir/lib
16:22:35,395: Server port : 8071
16:22:35,395: Jetty port : 8070
16:22:35,397: Jetty home directory : /repldir/
16:22:35,397: Jetty work directory : /repldir/work/jetty
16:22:35,397: Server connection limit : 256
16:22:35,398: Sticky services : true
16:22:35,398: Service checker class : (none)
...
16:22:37,900: Jetty started.
16:22:37,912: Control Server started.
16:22:37,913: Startup complete.
```

The logging mechanism is based on Log4j [21]. This makes it possible for the user to choose different levels of logging. In the default level (INFO) the logging is limited to messages about leader election, replica placement (see Listing 25), and possible errors. If this level is raised to DEBUG or TRACE, everything is logged with more detail, such as the monitoring process, in order to be able to trace the system's behavior.

Listing 25: "Logs of Leader Election and Successful Sending of a Web Service Replica"

```
16:34:29,672: Controlling service 'ticket'."
...
16:34:33,289: Executing SendCommand(ticket,desthost:8080).
16:34:34,831: Success.
...
16:48:37,291: Service 'ticket' now controlled by somewhere:80.
```

6.2 Case Study

In Chapter 4 the theory of the replication of Web services is explained in-depth. In order to make it besser comprehensible its functionality will be pointed out by discussing a pratical example. In this case study, which is based on a small ad-hoc network consisting of five hosts, the management of replicas is demonstrated. The dynamic topology of the network is simulated by using loopback network interfaces, which can be deactivated at any time. Each one of them has a class-C IP address assigned and runs an instance of the replicator system:

```
Host 0  ->   lo:0  ->   replicator system @ 192.168.2.100:8000
Host 1  ->   lo:1  ->   replicator system @ 192.168.2.101:8010
Host 2  ->   lo:2  ->   replicator system @ 192.168.2.102:8020
Host 3  ->   lo:3  ->   replicator system @ 192.168.2.103:8030
Host 4  ->   lo:4  ->   replicator system @ 192.168.2.104:8040
```

The detailed setup of the network, including the shell script for modifying its topology in a random manner, can be found in Appendix B.

This case study acts on the following assumptions:

- The hosts are sorted by their slightly different performance properties, with Host 0 acting as the fastest node and Host 5 as the slowest one.

- A disconnection of a host is comparable to departing too far away from the range of the wireless network. This implies that even though it has lost its connection to the rest of the network, it is, nevertheless, up and running. Thus it acts as a single node in an individual network.
- Only one Web service is deployed. This service desires to have at least two and at the most three replicas running concurrently.

Table 1 on page 65 illustrates the scenario, showing all hosts and the states of their Web service replicas. In the beginning all five hosts are connected and the network topology starts changing as soon as the Web service has been deployed on the first host.

1. *Initiation:*
   Initially the Web service replica is deployed only on Host 1, implying that this host becomes automatically its leader. However, the minimum number of replicas is set to two, which means that it must additionally be deployed somewhere in the network. Due to its superior performance properties, Host 1 is chosen as the destination.

2. *Host 1 disconnects:*
   As Host 1 is not available anymore, a new replica must be spawned to meet the minimum required number. This time Host 2 poses the fastest destination, and therefore the replica is deployed on it. Meanwhile Host 1 does not see other hosts anymore, and elects itself as the leader of its service.

3. *Host 0 disconnects:*
   With the disconnection of Host 0, the leader of the service becomes unavailable and Host 2 has to take over. Again too few replicas are deployed in the network, which results in sending the Web service to Host 3.

4. *Host 3 disconnects:*
   Host 3 disconnects and the leader sends a new replica to Host 4. This scenario is similar to Part 2.

5. *Host 0 reconnects:*
   Due to the fact that Host 0 regarded itself as the leader after it has lost the connection, two leaders exist as it reenters the network. This poses an inconsistency which has to be corrected. During the election in the next cycle, Host 2 stays the leader because it has already been accepted by the other hosts running the replica. As a consequence three replicas exist in the network.

6. *Host 1 reconnects:*
   Also in this situation two leaders exist concurrently for a short time, and again the majority of hosts elects Host 2 as the leader. Now four replicas exist, which means that one has to be eliminated. Since Host 4 is the slowest of them, the replica on it is hibernated and therefore undeployed.

7. *Host 3 reconnects:*
   This scenario is similar to Part 6, but in this case the replica is hibernated right after it has reconnected to the network, due to its inferior performance.

As a matter of course, this case study does not describe all possible failures, such as the interrupted sending of a Web service replica, hosts disappearing and reappearing frequently, hosts relocating, etc. Moreover, the synchronization of stateful replicas has been omitted for clarity. The purpose of this scenario is to demonstrate intelligibly how the replicator system handles simple dynamics of ad-hoc networks, such as disconnections, reconnections, multiple leaders, etc.

#	Activity	Host 0	Host 1	Host 2	Host 3	Host 4
1	**Host 0 receives the service**					
1.a	*deployment*	D				
1.b	*leader election*	L				
	-> Host 0 sends service to Host 1					
1.c	*deployment*	L	D			
2	**Host 1 disconnects**					
2.a	*disconnection*	L	D			
2.b	*leader election*	L	L			
	-> Host 0 sends service to Host 2					
2.c	*deployment*	L	L	D		
3	**Host 0 disconnects**					
3.a	*disconnection*	L	L	D		
3.b	*leader election*	L	L	L		
	-> Host 2 sends service to Host 3					
3.c	*deployment*	L	L	L	D	
4	**Host 3 disconnects**					
4.a	*disconnection*	L	L	L	D	
4.b	*leader election*	L	L	L	L	
	-> Host 2 sends service to Host 4					
4.c	*deployment*	L	L	L	L	D
5	**Host 0 reconnects**					
5.a	*reconnection*	L	L	L	L	D
5.b	*leader election*	D	L	L	L	D
6	**Host 1 reconnects**					
6.a	*reconnection*	D	L	L	L	D
6.b	*leader election*	D	D	L	L	D
	-> Host 2 hibernates service on Host 4					
6.c	*hibernation*	D	D	L	L	H
7	**Host 3 reconnects**					
7.a	*reconnection*	D	D	L	L	H
7.b	*leader election*	D	D	L	D	H
	-> Host 2 hibernates service on Host 3					
7.c	*hibernation*	D	D	L	H	H

Table 1: Case Study Results. (D = deployed service, L = leading service, H = hibernated service, red marked = disconnected host, yellow marked = currently changing state)

6.3 Further Work

During the coding of the replicator system, the implementation was always regarded more as a prototype and proof of concept, than as a complete software product. This means that it lacks some essential functionality necessary on systems which are used in companies, organizations, etc. This includes, for instance, security mechanisms. In particular it is necessary to sign, and sometimes also to encrypt, the communication between the individual replicator systems, in order to use this software in a real environment without opening huge security holes. Even though it is possible to tighten the security of the replicator system by various plug-ins, they have not been written yet.

Another important challenge for the near future is the tuning of monitoring parameters. Almost all calculations of the replicator system are based on a global view of the network, which is retrieved periodically by the monitors. Therefore the monitoring intervals have a strong impact on the systems flexibility to react on changes. Frequent monitoring allows to inform earlier about changes in the network, but can cause too much traffic. Infrequent monitoring solves the problem of consuming too much bandwidth, but slows down the placement of replicas. Furthermore it is often necessary to adjust the frequency to the size and dynamics of a network. Thus ideal trade-offs for various kinds of networks have to be found. They will be determined in numerous tests, by using sophisticated simulations of networks consisting of more than 200 hosts.

7 Summary

The goal of this thesis was to develop a solution for ensuring high availability of Web services in ad-hoc network environments, which have an unpredictable topology and therefore hamper the provision of dependable Service Oriented Architectures. In order to grant availability, the approach of replication and synchronization was adapted to the dynamics of ad-hoc networks. Although numerous solutions exist for how to replicate resources in static environments, the adaptation to a completely decentralized and unpredictable environment calls for a new and highly flexible solution. As specified in Chapter 1.2, such a solution has to fulfill several mandatory requirements, which were followed during the design process of the replicator system.

- *Avoiding centralization*
  Although the management of a group of Web service replicas is always controlled by a single host, centralization is only temporary and does not pose a single point of failure. In cases where the leader becomes unavailable, a new one is quickly elected.

- *Monitoring changes in availability & reacting to them*
  Distributed monitoring is performed to provide a complete and global view of the network's hosts and services. Inconsistencies, such as invalid numbers of replicas or unavailable leaders, are corrected immediately after they have been detected.

- *Monitoring health of hosts & considering this while replicating*
  The performance properties of hosts are monitored and compared to the preferences of Web services during the placement of new replicas. This way the load in a network is kept balanced.

- *Keeping bandwidth usage low*
  The network traffic is kept low by using the light-weight Simple Replica-

tor Protocol for communication purposes and by performing the monitoring of the network in a distributed and incremental manner.

- *Synchronizing service states quickly*
  The individual Web services are free to notify the server about necessary state synchronizations, which are then performed via the Simple Replicator Protocol.

- *Making invocation of Web services convenient for the clients*
  A WSDL-finder utility disburdens the application developers from handling ad-hoc specific difficulties and looking up the proper replica instances.

7.1 Conclusion

The contribution of this thesis is an enhancement to Service Oriented Architectures to enable their application within typically unreliable networks. The developed replicator system provides the necessary flexibility to handle such environments. Nevertheless, it is not bound to ad-hoc networks exclusively, but can be used in any other environment where high availability of Web services is desired. This makes it a powerful all-round solution for providing dependability to Service Oriented Architectures.

References

[1] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: a self-organizing structured P2P system. *SIGMOD Record*, 32(3):29–33, 2003.

[2] K. P. Birman, R. van Renesse, and W. Vogels. Adding High Availability and Autonomic Behavior to Web Services. In *ICSE*, pages 17–26. IEEE Computer Society, 2004.

[3] N. Budhiraja and K. Marzullo. Highly-Available Services Using the Primary-Backup Approach. In *Workshop on the Management of Replicated Data*, pages 47–50, 1992.

[4] G. Chockler, R. Friedman, and R. Vitenberg. Consistency Conditions for a CORBA Caching Service. In M. Herlihy, editor, *DISC*, volume 1914 of *Lecture Notes in Computer Science*, pages 374–388. Springer, 2000.

[5] E. Dekel, O. Frenkel, G. Goft, and Y. Moatti. Easy: Engineering High Availability QoS in wServices. In *SRDS*, pages 157–166. IEEE Computer Society, 2003.

[6] R. Friedman. Caching web services in mobile ad-hoc networks: opportunities and challenges. In *POMC*, pages 90–96. ACM, 2002.

[7] L. Juszczyk, J. Lazowski, and S. Dustdar. Web Service Discovery, Replication, and Synchronization in Ad-Hoc Networks. In *ARES*, pages 847–854. IEEE Computer Society, 2006.

[8] J. Lazowski. Web Service Discovery in Ad-Hoc Netzwerken. Master's thesis, University of Technology Vienna, May 2006.

[9] Apache Axis SOAP Container. `http://ws.apache.org/axis`.

[10] Apache Axis2 SOAP Container. `http://ws.apache.org/axis2`.

[11] Apache SOAP Implementation. `http://ws.apache.org/soap`.

[12] Apache Web Service Invocation Framework. `http://ws.apache.org/wsif/`.

[13] CORBA 3. `http://www.omg.org/technology/documents/formal/corba_2.htm`.

[14] HyperText Markup Language. `http://www.w3.org/MarkUp/`.

[15] HyperText Transfer Protocol. `http://www.w3.org/Protocols/`.

[16] IBM Colombo Middleware for Web Services. `http://www.research.ibm.com/journal/sj/444/curbera.html`.

[17] IBM WebSphere. `http://www.ibm.com/websphere`.

[18] IEEE 802.11 Working Group. `http://www.ieee802.org/11/`.

[19] Jetty HTTP Servlet Server. `http://jetty.mortbay.org`.

[20] JXTA Peer-to-Peer Protocols. `http://www.jxta.org`.

[21] Log4j. `http://logging.apache.org/log4j/`.

[22] Microsoft Distributed COM. `http://www.microsoft.com/com/`.

[23] Microsoft .NET. `http://www.microsoft.com/net/`.

[24] SOAP. `www.w3.org/TR/soap/`.

[25] Sun Enterprise JavaBeans. `http://java.sun.com/products/ejb/`.

[26] Universal Description, Discovery and Integration. `http://www.uddi.org`.

[27] Web Service Definition Language. `www.w3.org/TR/wsdl`.

[28] Extensible Markup Language. `www.w3.org/XML/`.

[29] XML Schema Definition. `www.w3.org/XML/Schema`.

[30] Wikipedia: Web service. `http://en.wikipedia.org/wiki/Web_service`.

A Sample Web Service Implementation

The following listing contains the source code of a simple but stateful Web service, distributing random integer tickets via its function “newTicket()”. Furthermore, it keeps the date of the last request saved. This date and the ticket counter pose the service’s internal state and are synchronized during every invocation.

Listing 26: ”Sample Implementation of a Synchronized Web Service”

```java
// web service class
public class SampleService extends SynchronizedService {

    // contructor
    public SampleService() {
        super();
    }

    // state variables
    public static Integer counter=0;
    public static Date lastRequest=new Date();

    // state objects
    private static FieldSetterStateObject counterObject=null;
    private static FieldSetterStateObject dateObject=null;

    // service name
    @Override
    public String getServiceID() {
        return "sampleservice";
    }

    // state object initializer
    @Override
    protected void initializeStateObjects() throws Exception {
        if (counterObject==null) {
            counterObject=new FieldSetterStateObject(
28                  SampleService.class.getField("counter"));
29          }
30          if (dateObject==null) {
31              dateObject=new FieldSetterStateObject(
32                  SampleService.class.getField("lastRequest"));
33          }
34      }
35
36      // map of state objects
37      @Override
38      public HashMap<String,IStateObject> getStateObjects() {
39          HashMap<String,IStateObject> result=
40                              new HashMap<String,IStateObject>();
41          result.put(counterObject.getID(),counterObject);
42          result.put(dateObject.getID(),dateObject);
43          return result;
44      }
45
46      // web service function, returns a new integer ticket
47      public int newTicket() {
48          counter=new Random().nextInt();
49          lastRequest=new Date();
50
51          try {
52              // synchronize the state
53              synchronizeStateObjects();
54          } catch (Exception e) {
55              // ignore errors now
56          }
57
58          return counter;
59      }
60  }
```

- `Lines 10-11`: The variables for the internal state are declared.
- `Lines 14-15`: The state objects are declared but not yet initialized.

- `Lines 19-21`: “getServiceID()” returns the string-ID of the service.
- `Lines 25-34`: “initializeStateObjects()” binds the state objects to the corresponding variables.
- `Lines 38-44`: “getStateObjects()” returns a HashMap containing all the service’s state objects.
- `Lines 47-59`: “newTicket()” is the only operation of the service. After the new ticket and the date have been calculated, this method initiates the synchronization, ignoring all possible exceptions.

B Simulation of Transient Host Lifetimes

The simulation of transient lifetimes of hosts in a network is done by setting up multiple loopback network interfaces, which are bound to class-C IP addresses and turned on and off in a random manner. Listing 27 contains the interface declarations from a Debian Linux configuration. The bash script simulating the random behavior can be found in Listing 28.

Listing 27: ”Interface-to-IP Mapping”

```
iface lo:0 inet static
        address 192.168.2.100
        netmask 255.255.255.255

iface lo:1 inet static
        address 192.168.2.101
        netmask 255.255.255.255

iface lo:2 inet static
        address 192.168.2.102
        netmask 255.255.255.255

... more interface declarations ...
```

Listing 28: ”Bash Script for Manipulation of Interfaces”

```
#!/bin/bash

NUMBER_OF_INTERFACES=5

# random intervals, but not longer than 5 seconds
while sleep ‘expr $RANDOM % 5‘
do
    # 0 means down, 1 means up
    UP=‘expr $RANDOM % 2‘

    # number of the interface
    INTERFACE=‘expr $RANDOM % $NUMBER_OF_INTERFACES‘

    # check if the interface is already in this state ...
    # ... or has to be changed
    if [ x‘printenv IF$INTERFACE‘ != x$UP ]
    then
        # print the date
        echo -n "‘date +\"%H:%M:%S:%N \"‘ "

        if [ $UP == 1 ]
        then
            # bring the interface up
            echo "interface if:$INTERFACE is brought up"
            ifup lo:$INTERFACE
        else
            # bring the interface down
            echo "interface if:$INTERFACE is brought down"
            ifdown lo:$INTERFACE
        fi

        # save the current state of the interface
        export IF$INTERFACE=$UP
    fi
done
```