SOCA (2007) 1:19–33
DOI 10.1007/s11761-007-0006-z

ORIGINAL PAPER

Dynamic replication and synchronization of web services for high availability in mobile ad-hoc networks

Schahram Dustdar · Lukasz Juszczyk

Received: 20 October 2006 / Revised: 22 December 2006 / Accepted: 10 January 2007 / Published online: 13 March 2007
© Springer-Verlag London Limited 2007

Abstract Web services are gaining high popularity and importance on mobile devices. Connected to ad-hoc networks, they provide the possibility to establish spontaneously even complex service-based workflows and architectures. However, usually these architectures are only as stable and reliable as the underlying network infrastructure. Since topologies of mobile ad-hoc networks behave unpredictably, dependability within them can be only achieved with a dynamic replication mechanism. In this paper we present a highly flexible solution for replication and synchronization of stateful Web services and discuss the behavior of the implemented prototype in large-scale simulations.

Keywords Web Services · Replication · Synchronization · Fault-Tolerance · Peer-to-Peer · Mobile Ad-Hoc Networks

1 Introduction

Web services, as standardized and extensible software systems for machine-to-machine interaction, open up many new possibilities to perform automated workflows based on loosely coupled services. This flexibility and the high interoperability, due to the use of open standards, had a great impact on the rising popularity of Web services on mobile devices [1–7], especially in collaborative working environments [8–11]. The area of application ranges from services enabling access to personal data such as cryptographic keys, payment methods, identifications, etc. to services providing desired functionality for a whole group of clients, such as registries, data transcoders, proxy services, etc. While the first group of services is often strictly bound to the user's device, services of the second group need to be constantly available. In ad-hoc networks, fulfilling this basic requirement is hampered, since the behavior of each node is unpredictable, which results in a highly dynamic topology. Nodes are able to relocate in the network, can disappear due to shutdowns or unstable connections, just to mention some common characteristics. To apply reliable Service Oriented Architectures (SOAs) in such environments, it is necessary to replicate individual Web services and to ensure their synchrony, including those of stateful Web services. Such a solution has to be able to deal with transient nodes and changing network structures, to place replicas dynamically, and to allow their convenient invocation for the clients.

In [12] we presented a system for dynamic discovery and replication of Web services in ad-hoc networks, for utilizing the *publish–find–bind* paradigm of SOAs even in typically unreliable network environments. The contribution of the current paper is concerned with the evolution of the replication mechanism and the results of a case study, simulating the system's behavior in large-scale ad-hoc environments, in order to find optimal trade-offs for an effective replication.

The structure of this paper is as follows. Section 2 starts with discussing the concept and the architecture of the proposed solution. Section 3 presents the case study and the results of the simulations. Section 4 contains a

S. Dustdar (✉) · L. Juszczyk
Distributed Systems Group, Vita Lab,
Institute of Information Systems,
Vienna University of Technology,
Argentinierstrasse 8/184-1, 1040 Vienna, Austria
e-mail: dustdarl@infosys.tuwien.ac.at

L. Juszczyk
e-mail: ljuszczyk@infosys.tuwien.ac.at

short overview of related work and Sect. 5 finally concludes and contains ideas for future work.

2 Dynamic replication of web services in ad-hoc networks

Replication has been used for a long time to achieve dependability. High-Availability Clusters, Redundant Arrays of Independent Disks (RAIDs), replicated databases, and the root servers of the Domain Name System (DNS) are probably the best known examples for using redundancy in order to ensure fault tolerance. Furthermore, solutions for the replication of Web services have already been developed, e.g., [13,14]. These solutions frequently have the drawbacks that they use replicas at static and predefined locations, are based on centralized request dispatchers or controllers, or use other techniques which are suited solely to managed infrastructure networks, in which they operate.

However, taking the requirements and restrictions of mobile ad-hoc networks into consideration, a completely new approach has to be applied. Ad-hoc networks are established spontaneously via wireless network links and without the need of any preexisting physical infrastructure. All communication is routed via nodes within the wireless range, which are, in turn, responsible for forwarding the packets until they reach the final destination node. Therefore, the availability of individual nodes as well as the coherence of the whole network is vulnerable to the movements of all participants in the network. This poses a challenge to any replication mechanism. Firstly, the replicas have to be placed in a dynamic manner, which avoids predefined and static locations, hence, the replicator is able to react dynamically to changes in the network. Secondly, the detection of changes in the availability of individual nodes and their services is important. Since disconnections might happen due to crashes or because the node moved too far away from the wireless range, one cannot rely on the nodes to report their unavailability via event notifications. Therefore, it is necessary to perform active monitoring in intervals. The third important requirement lies in the peer-to-peer characteristics of mobile ad-hoc networks, which strictly require a completely decentralized solution. Furthermore, mobile devices have the disadvantage of consuming battery power. To reduce its consumption, it is necessary to keep the produced network traffic as low as possible.

These limitations disqualify the currently available solutions for replication and call for a new approach which (a) is flexible enough to handle the highly dynamic network structures, (b) is completely decentralized, since ad-hoc network are not suited to static and centralized resources, (c) monitors the availability of Web services, (d) takes performance properties into consideration while placing replicas, and (e) produces as little network traffic as needed.

In [12] we introduced a system consisting of a combination of dynamic Web service discovery based on distributed UDDI [15] registries, and of a Web service replicator mechanism for ad-hoc networks, which was work in early progress at that time. Since then, the replicator evolved towards being more scalable, flexible, and bandwidth-saving, and, therefore, better suited for mobile ad-hoc networks. The following sections discuss the concept of our approach, with Sect. 3 presenting the results of an intense testing in a simulated ad-hoc network with up to 140 nodes.

2.1 Concept and architecture

Replication of stateful Web services can be achieved in (a) an active manner, also known as state machine [16], (b) in a semi-active one, or (c) in a passive manner, also called primary copy [17]. The state machine is based on the idea of sending invocations to all replicas and waiting for all responses. Combined with methods for suppressing nested invocations [18] and for ensuring that all replicas receive their requests in the same order, this technique guarantees automatically synchrony of states. However, it assumes that all operations produce deterministic output and state transitions, excluding, for instance, functions which use random data. This limitation can be overcome by declaring explicitly all indeterministic functions and by directing their invocations to only one service, which is in turn responsible for forwarding possible changes of the internal state to all other replicas. This approach is called semi-active and is a composition of the ideas of the state machine and the primary copy approach. In primary copy all invocations are sent to only one destination. This primary service updates automatically all backups and stays the master of all replicas until a failure occurs, in which case a new one has to be selected.

Comparing these techniques considering the dynamic destination environment, it becomes obvious that the active and semi-active approaches are unsuitable. By virtue of the possibility of ad-hoc networks to split and merge, this would require a expensive synchronization before the invocations, in order to have all replicas performing their calculations based on the same internal state. Furthermore, it is essential to keep network traffic as low as possible, which disqualifies an approach where SOAP-based invocations are sent to all replicas. Therefore, we chose primary copy for our replicator system,

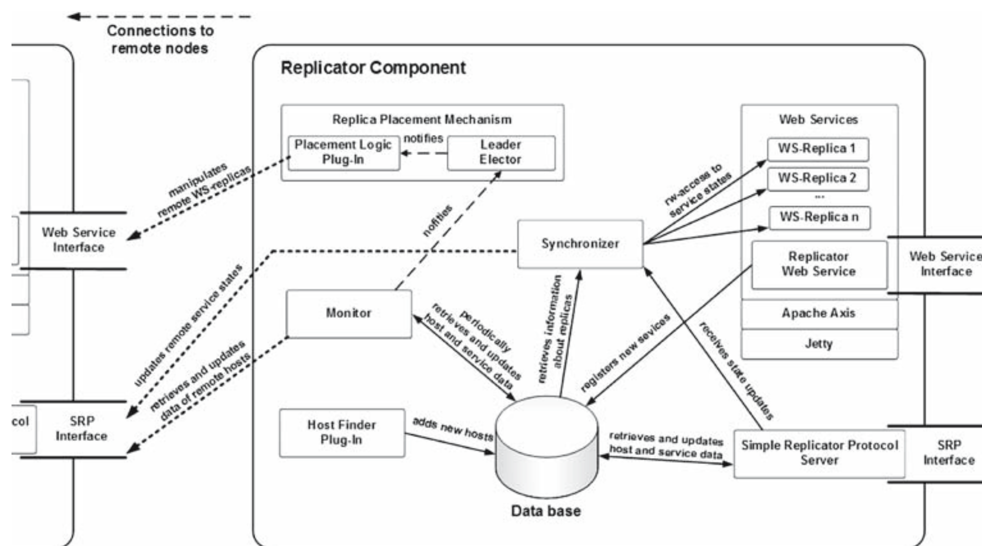

Fig. 1 Modularized architecture of the replicator system

with one service instance handling all requests and synchronizing all backup instances via the *Simple Replicator Protocol* (SRP) which saves bandwidth.

To clearly structure the complex task of replication and to make it maintainable and flexible, the web service replicator was designed as a collection of individual modules (see Fig. 1), cooperating via the lightweight *Internal Database* which stores all relevant data. These modules have a small memory footprint and are part of every instance of the Web service replicator, which means that all nodes are theoretically able to execute all particular tasks, such as monitoring or managing replicas. The result is a true peer-to-peer community of nodes where elections decide about distributing the necessary tasks, preferably on the most powerful nodes, while the remaining ones stay idle.

- Replication is handled by the modules *Replicator Web Service*, *Monitor*, and *Replica Placement Mechanism*.
- Synchronization is done separately within the *Synchronizer* module and extensions to the replicable and stateful Web services.
- All web services, viz. the *Replicator Web Service* and the individual replicas, are deployed at a combination of the Jetty Servlet Container [19] and Apache Axis [20].
- The *Simple Replicator Protocol Server* is handling all communication which is done via the lightweight SRP protocol.

Furthermore, instead of using hardcoded techniques for finding nodes in the network, a plug-in interface is provided for allowing the system to operate within all kinds of TCP-based computer networks, not only mobile ones. For instance, one might want to provide reliable web services in a wide area network (WAN), belonging to some community, which is distributed over the world and where hosts are known to be shut down at times. Each node joining a network must know at least one of the already connected nodes or must wait until it gets discovered itself (active vs. passive connecting). After that, the system puts automatically the rest of the integration to the peer-to-peer network into effect.

2.2 Concept of replication

All calculations of the replication mechanism are based on a global view. This means that nodes are informed about all other nodes in the network which have the replicator system installed, their properties, and their hosted services. Although, compared to gossip-based approaches, this hampers scalability, a global view is unavoidable to keep the replicas of a particular service in a synchronized state.

The simplified replication mechanism works as follows: the *Monitor* checks periodically for changes in the network and sends after each cycle a notification to the other modules about having finished its task. The newly monitored state is then analyzed by the *Replica Placement Mechanism* which evaluates whether the controlled replicas are in an inconsistent state which needs to be corrected. This procedure consists of first electing leaders for the replicas of each web service, and eventually controlling the ones for which the local host was elected as the leader. The following sections

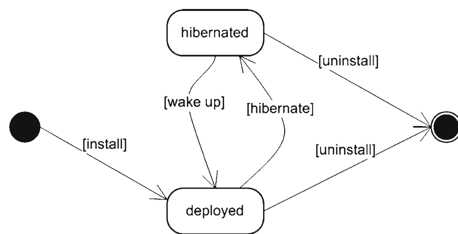

Fig. 2 Deployment states of web service replicas

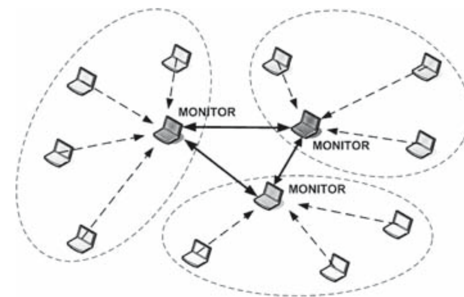

Fig. 3 Distributed monitoring

provide a more in-depth insight into the functionality of the individual modules.

2.2.1 Internal database

The database is kept entirely in memory and acts as a medium between all modules of the system, by storing information about hosts and services, their availability, performance properties and requirements, and information about current leaders and monitors in the network. Every module can retrieve and update the records in order to perform its calculations and to store the results. To keep the database scalable, all records are fully indexed and queries are performed via a set-based language, which is able to process most of the necessary queries in O(1) or O(log n).

2.2.2 Replicator web service

The *Replicator Web Service* is a special case of a module, which works completely in passive mode and is invoked only by remote hosts. Its task is to provide the facility to hot-deploy web services and, furthermore, to manipulate their states remotely. As Fig. 2 illustrates, these states can be either deployed, hibernated, or not installed at all. The idea of hibernating web services which are no longer needed, is to disable them instead of deleting them completely. In case when too few replicas exist and new ones have to be deployed, hibernated ones can be woken up quickly without resending the whole archive. This way the replication mechanism can react faster to changes in the network, reducing network traffic at the same time.

The realization of this module as a web service allows to communicate with a destination which is operating in the same Java environment as the replicas. This is necessary for evaluating whether all necessary resources of a replicable web service (e.g., Jar-files, classes) already exist or need to be installed before the deployment.

2.2.3 Monitor

The *Monitor* is the system's source for getting information about changes in the network and, therefore, has a significant influence on the response time of the whole replication mechanism. All hosts and services have to be checked periodically for evaluating whether their states have changed, whether they are still available, and what properties they have. For better scalability, this procedure is performed in a distributed and incremental manner (see simplified scenario in Fig. 3). By performing the algorithm in Listing 1 the most powerful nodes are elected as monitors, which are responsible for partitioning the nodes by their unique IDs into groups, detecting changes within their groups, and forwarding them to the other monitors. Furthermore, the monitors exchange addresses of known nodes in order to merge possible subnetworks and to guarantee the coherence of the whole environment. This technique is referred to as *active monitoring* (Listing 2). In contrast, all other nodes in the network perform *passive monitoring* (Listing 3), which means that they select one of the active monitors and retrieve periodically the actualized state from it.

The nodes of the network usually notice changed states within one or two monitoring cycles.

1. The monitor of the changed node's group detects it within one cycle and forwards it immediately to the other monitors.
2. All other nodes, which perform passive monitoring, retrieve it from the active monitors during the next cycle.

All communication necessary for monitoring is done via the *Simple Replicator Protocol*, which was designed to keep exchanged messages as short as possible by serializing all data into a compact format, in order to save network bandwidth. Furthermore, network traffic is reduced even more by exchanging data in an incremental manner, which means that only changes which took place after the last request are transferred.

Listing 1 "Election of active monitors (Pseudocode)"

```
monitoring() {
// bootstrap: address(es) of host(s) were retrieved by host-finder,
// but no data about their properties or services were retrieved yet
// -> retrieve complete view from random host
    passive_monitoring(random host)
// main loop
    loop in intervals {
    sort all hosts by performance properties
// 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 use it again in the next cycle
// retrieve the view of the network from it
        if (mon from last loop not available) {
        mon = random monitor
        }
        passive_monitoring(mon)
        }
    send event notification to leader elector
    }
}
```

Listing 2 "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 which must be checked
    mygroup = all hosts where (host.id%num == pos)
// do the actual monitoring
    start concurrent threads for all hosts in mygroup {
    check host and properties
    check services and requirements
    }
    wait until threads finish
// exchange data with other monitors
    start concurrent threads 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
    }
}
```

2.2.4 Replica placement mechanism

After the changed state of the network was monitored, the *Replica Placement Mechanism* comes into play. Its task is mainly to follow the declared requirements of the Web services (e.g., min/max number of replicas, system performance) in order to have them placed at the best suited locations, balancing the load on the nodes this way. Moreover, it is responsible for moving replicas away from nodes which are constantly under heavy load or have only little time left to live, due to low batteries. This task is split into the *Leader Elector* module and the actual placement logic.

The *Leader Elector* has to check whether the local host is expected to manage the replication of a particular Web service (Listing 4). This decision is done solely by querying the database for the properties of the other replicas in the network, analyzing their preferences and accepting the resulting leader, which can, of course, be the local host. The advantage of this method is that all

Listing 3 "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 incremental data
    } else {
// get the whole view of the network
    retrieve all data
    }
// inform monitor of newly connected nodes
    send list of new node addresses to monitor
}
```

calculations are based on the information in the local database, without the need of contacting other nodes, which would slow down the overall process.

A feature of this algorithm is the ability to correct quickly the inconsistency of multiple concurrent leaders of the same group of replicas. Multiple leaders can occur in situations where a group of replicas was split due to movements of routers in the ad-hoc network, the newly independent groups elected their leaders, and were later merged again.

After the election, the placement logic is notified to manage the replicas controlled by the local host. We do allow the application of custom logics as plug-ins, which must use the provided API for transferring services between nodes. The API postpones all received commands to after the next monitoring cycle, in order to check then whether the local host is still controlling this replica. If the control was lost in the meanwhile, e.g., because another node won the election, the command will be discarded, otherwise, it will be executed. This way possible collisions between multiple leaders can be solved in most cases.

By default, or if an invalid plug-in is used, a predefined logic is applied (Listing 5). Its main task is to control whether the desired number of replicas is deployed and to correct this in case of an inconsistency. Furthermore, it checks whether some of the replicas have to be moved to better suited locations by comparing the performance properties of hosts and the requirements of services. The case study in Sect. 3 was performed with this default replication logic.

2.3 Concept of synchronization

Since most of the Web services, which are deployed on mobile devices, are stateful, it is important to ensure synchrony between all replicas. Unfortunately, this is not always possible. Ad-hoc networks can split into multiple subnetworks and merge again after a time. In these subnetworks the internal states of the replicas can get totally out-of-sync with the ones in the other nets, so that resynchronization after a merging becomes impossible. Imagine a Web service with a similar functionality to DHCP, where clients request unique addresses from a limited pool, e.g., for identification within some virtual network structure. If replicas of this service get split, each group only knows the addresses it has offered to the clients before, but it does not have any information about the other groups anymore. This can result in addresses being assigned to more than one client, which poses a conflict in case the groups merge again. This limitation for stateful and replicated services in ad-hoc networks is a fact which has to be accepted. Therefore, Web services which need perfect consistency of states are not applicable within them. However, for the rest of the services, we provide a facility to synchronize their states quickly.

To make use of this functionality, each stateful Web service (see sample service in Listing 6) must extend a class, which provides the necessary functionality to perform synchronization but also registers the service automatically at the *Synchronizer* module. This registration also implies that the Web service grants full read-write access to its so called *State Objects*, which encapsulate all data (e.g., variables, objects) relevant for the internal state. Now the service is able to command the module to synchronize its state with the other replicas. Usually this is done after an invocation which changed the state. As a consequence the *Synchronizer* checks which *State Objects* have changed, contacts the remote Synchronizers running on the replicas, serializes the changes, and updates the remote states. To keep this communication fast and lightweight, it is realized again via the *Simple Replicator Protocol*. Listing 7 contains a short sample of a SRP communication.

- Lines 1–5: A list of state objects of service "syncTest" is retrieved. It contains two objects, named "price" and "book_title", including their serial stamps which

Listing 4 "Election of leader for replication (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 = declared leaders of all replicas
// the most popular one will be accepted
    sort leaders by popularity/frequency
// 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
```

are incremented after each synchronization, and the hash sums for comparing equality.

- Lines 6–9: Object "book_title" is retrieved in a serialized form. The first field holds the length of the serialized string. The rest of the item consists of the object's name, its serial stamp, the Java class name and the Base64-encoded value of the variable.
- Lines 10–12: A new value is assigned to object "price".

Possible state collisions, which might happen after two desynchronized groups of replicas merge again, can be resolved by either fusing the states or by simply declaring one state as dominant and withdrawing the others. Although the first variant seems to be preferable it is only possible for a subset of stateful Web services, e.g., for registries which can be rejoined. For these services we provide a possibility to plug-in a module which is able to merge multiple states into a single consistent one. However, for most Web services merging states is either not possible or at least too expensive. In such situations the *Synchronizer* automatically determines the dominant state, which is the one that was accessed most often, and replaces all conflicting ones. Although this method has the drawback of revoking a state which was used for servicing past requests, it is in many cases unavoidable to reinstall consistency between all replicas.

As mentioned in Sect. 2.1, the concept of our solution is based on the primary copy approach. This means that of all replicas, one is selected as the master and all invocations are directed to it. This selection is already done in the *Leader Elector* module, which elects a leader node for controlling all replicas of a particular service.

2.4 Concept of invocation

To have the replicator system running on the nodes of a mobile ad-hoc network raises the question how this all affects the clients which want to invoke a certain Web service. How does a client find the proper primary copy? What happens when the primary copy fails and a new one is not elected until this failure is detected? How shall a situation where, due to the merging of two subnetworks, two primary copies exist, be handled? In order to disburden the client developers from solving all these difficulties we provide a simple Java tool named *WSDL-finder*. The task of this utility is to find the proper Web service instance in the network, to track the movements of the replicas, and finally to return a WSDL file pointing to the primary copy. The sequence diagram in Fig. 4 illustrates these steps.

1. At first, an instance of the desired Web service must be found and passed to the *WSDL-finder*. Discovery of Web services in dynamic and transient networks can be achieved with methods as presented in [12,21]. However, this is not a task of the tool and must be done by the client.

Listing 5 "*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, e.g., due to low batteries
    if (replicas exists on transient hosts) {
    move services to other/fastest hosts
    synchronize new replicas
    }
}
```

2. Then, the tool retrieves the correct location of the primary copy instance and returns a WSDL file pointing to it. This consists of:
   (a) contacting the host of the discovered service and retrieving the locations of all replicas. Moreover, the *WSDL-finder* requests the location of the primary copy from each replica and accepts the most popular one, in order to correct temporary inconsistencies. 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 registries continuously.
   (b) contacting the primary copy and retrieving the automatically generated WSDL file from the Apache Axis SOAP Container [20].
3. The client can now pass this WSDL file as an argument to the Apache Web Service Invocation Framework [22] and invoke the proper Web service replica.

The *WSDL-finder* must be used before every invocation of a replicated Web service in order to be aware of a changing location of the primary copy. Since most of the communication is done via the fast and light-weight *Simple Replicator Protocol*, the additional traffic and delay is kept very low.

Temporary inconsistencies, such as an unavailable primary copy or multiple concurrent ones, are handled automatically. Since the election of a new primary copy after a failure of the old one is usually only a matter of a few seconds (depending on the monitoring intervals), the utility simply waits and polls one of the cached replicas periodically to retrieve the new location. In contrast to this, multiple concurrent replicas, which may occur after a merging of subnetworks, do not pose a problem for invocation. In fact, it is just a continuation of the scenario with split networks, where separated replicas of the same service are invoked. Possible state conflicts are resolved by the synchronization mechanism later anyway, when the leaders of the separated replicas have to be merged.

3 Case study

The replicator system is based on the idea of having nodes, which know about the other nodes in the network and their services, are able to determine whether they are expected to perform some tasks (e.g., monitoring, controlling particular replicas), and also know which other nodes are currently performing which tasks. In short, it is based on a global view. This knowledge about the state of the distributed replicator system is retrieved periodically from the monitoring nodes. These, however,

Listing 6 "*Sample Web service with a synchronized String object*"

```
public class SampleService extends SynchronizedService {

    // synchronized state object
    private static FieldSetterStateObject synchronizedString;
    // variable holding the actual state value
    private static String stringObject=``ello world'';

    public SampleService() throws Exception {
    // SynchronizedService() registers at synchronizer module
    super();
    }

    // called during registration at synchronizer module
    @Override
    protected void initializeStateObjects() throws Exception {
    // create the state object pointing to the string
        synchronizedString=new FieldSetterStateObject(
                SampleService.class.getField(``stringObject''));
    // grant read/write access to the synchronizer
        registerStateObject(synchronizedString);
    }

    // Web service operation
    public String getString() {
        return stringObject;
    }

    // Web service operation
    public void setString(String str) {
        stringObject=str;
        try {
        // propagate updated state to all replicas
        synchronizeStateObjects();
        } catch (Exception e) {
            // handle or ignore failed synchronization
        }
    }

    // ...
}
```

Listing 7 "*Simple Replicator Protocol* - Sample commands for manipulating state objects"

```
1  > LISTSTATE syncTest
2  < 100 OK
3  < price 2 123
4  < book_title 4 96354
5  < .
6  > GETITEMS syncTest book_title
7  < 100 OK
8  < <51|book_title|4|java.lang.String|V2ViU2VydmljZXM=>
9  < .
10 > ASSIGNITEMS syncTest <35|price|4|java.lang.Integer|OTk=>
11 < 100 OK
12 < .
```

are in turn elected by using the last monitored state. Moreover, changed states (e.g., failures of nodes, new available nodes, changed properties) may imply further changes, such as relocations or elections of new monitors and controllers, which have to be propagated again. Therefore the replicator system works in a recursive manner and was designed to correct possible inconsistencies by swinging into a consistent state again, usually not later than after two or three monitoring cycles.

The purpose of the case study was to analyze this behavior, to evaluate how the system behaves in networks consisting of up to 140 nodes, and how fast it reacts

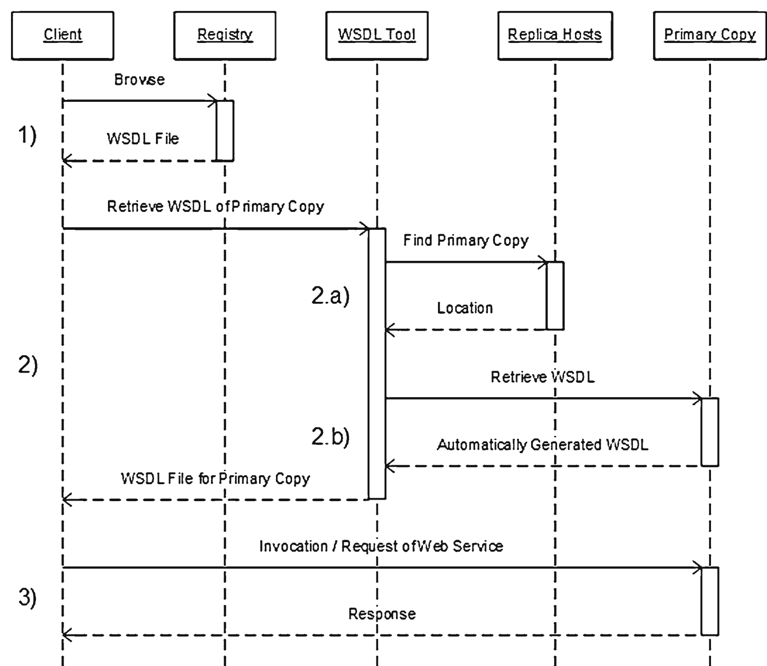

Fig. 4 Invocation of replicated Web services by using the WSDL-finder

to changes in the network. Furthermore, the simulations were used to determine proper configuration values, such as monitoring intervals, which have an immense impact on the systems performance and response time.

During this case study we concentrated only on the replication mechanism and did not include any simulations for testing the performance of service state synchronization. The reason for this is that the synchronization is completely controlled by the individual Web services, which are free to decide when and how their states have to be synchronized, and are expected to do it wisely regarding the capabilities of the environment they are operating in. Especially the size of their *State Objects*, the number of deployed replicas, and the frequency of invocation have a significant influence on the load of the network. However, these values are neither part of the configuration of the replicator system nor should be restricted by it.

3.1 Simulation of transient networks

The case study was performed on two blade servers which each have four Intel Xeon 3.2 GHz CPUs, 2 GB of RAM, and Linux as the operating system. The Web service replicator uses only a marginal amount of CPU power, however, each instance comes with an own Jetty and Apache Axis server and has to run in a separate Java Virtual Machine (JVM). As a result, the memory usage of each replicator instance is approximately 27–29 MB (including 11 MB of shared memory). This consumption can be reduced to a few MB by using a JVM for PDAs, which has a much smaller memory footprint, and by replacing Apache Axis with a more light-weight Web service container or SOAP API, such as kSOAP [23].

For the evaluation we started 140 instances of the replicator system in a simulated mobile ad-hoc network with transient node availability, limited the bandwidth of the network links (using Traffic Control [24]) to WLAN-typical 11 MBit, and extended the replicators with a possibility to disable them, in order to simulate failures. The actual simulation was controlled by a utility which disabled single nodes in a random but balanced manner, and enabled them again after a defined period of time. The balancing was applied in order to keep statistical variations of the results low in spite of the randomized testing, by paying attention to adjusting the downtimes of the nodes. Furthermore, randomized performance properties were assigned to all nodes during the bootstrapping process.

Due to an extended and detailed logging, we were able to trace the behavior and bandwidth usage of the system by analyzing the log files.

3.2 Evaluation

Two of the main requirements to the replicator system are: (a) to react as quickly as possible to changes in the network and (b) to consume as little network traffic as possible. However, faster response times can only be achieved as a consequence of a more frequent moni-

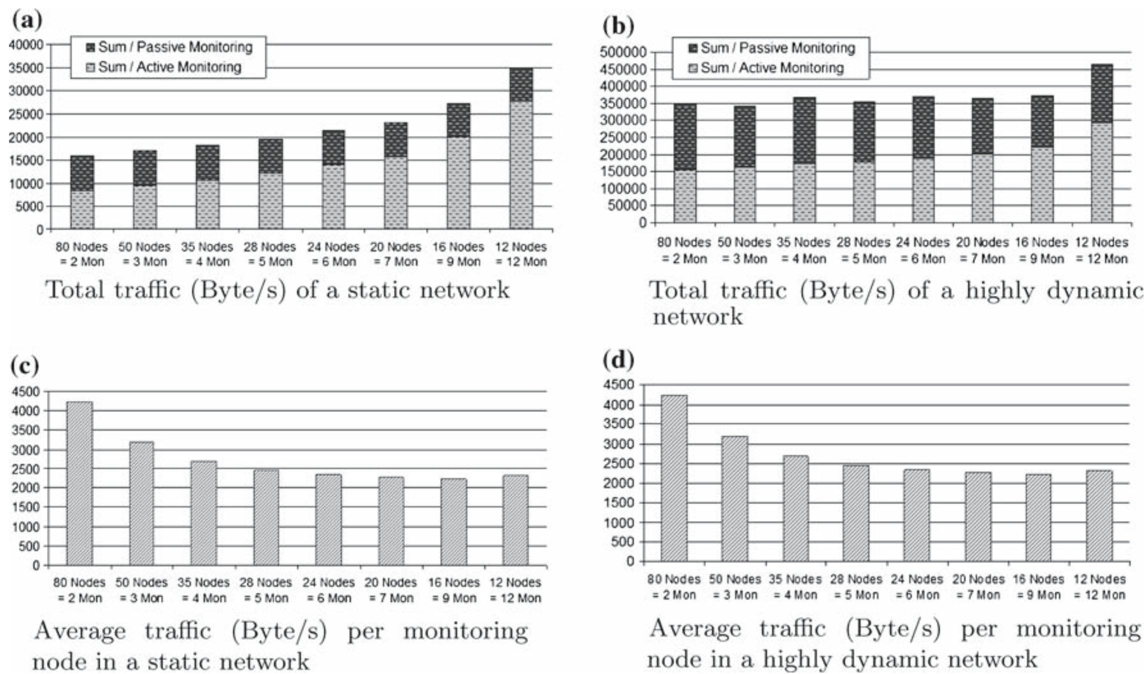

Fig. 5 Traffic of networks with various sizes of monitor groups

toring, which, in turn, produces more traffic. Therefore, it was necessary to find the best trade-offs, in order to satisfy the need for a quick reaction of the replica placement mechanism without consuming too much bandwidth. These trade-offs were determined by collecting and analyzing the following statistics:

- Network traffic of passive/active monitoring in static/dynamic networks with various numbers of monitors
- Network traffic of passive/active monitoring in static/dynamic networks of various sizes
- Network traffic, depending on the monitoring interval
- Amount of time necessary to detect changes in the network, depending on the monitoring interval
- Amount of time necessary to elected a new leader after a failure, depending on the monitoring interval
- Amount of time necessary to deploy a new replica after a failure, depending on the monitoring interval

Since mobile ad-hoc networks vary in size, bandwidth, dynamic behavior, and the performance of the nodes, it is impossible to find an optimal general configuration for all environments. Furthermore, the number of deployed Web services, the size of their replicas, and preferences for replication have also a strong influence on the performance. However, it makes sense to analyze the system's general behavior in a case study and to find trade-offs which will perform well in most environments. This way, we determined a default configuration for the replicator system.

3.2.1 Network traffic and scalability

For finding a configuration which keeps network traffic low, it was necessary to take a close look at the system's behavior in static and dynamic environments, in networks of different sizes, and, first of all, to analyze the produced traffic depending on the size of the monitoring groups.

As explained in Sect. 2.2.3, the monitoring works in a distributed manner, which partitions the network into groups, each checked by a single active monitor. These monitors exchange the state information about their groups among themselves. Although a higher number of monitors reduces automatically the load, caused by checking the group, on each one of them, it also increases the exchanged amount of data:

$$amount_exchanged_data = \frac{amount_all_data * (num_monitors - 1)}{num_monitors}$$

The purpose of the first simulations (see Fig. 5) was to evaluate a proper number of monitors, viz. a proper size of their groups, in order to find a balance between the individual load on each monitor and the total traffic of the network. For this reason, we tested the replicator in a static environment, where all nodes were already

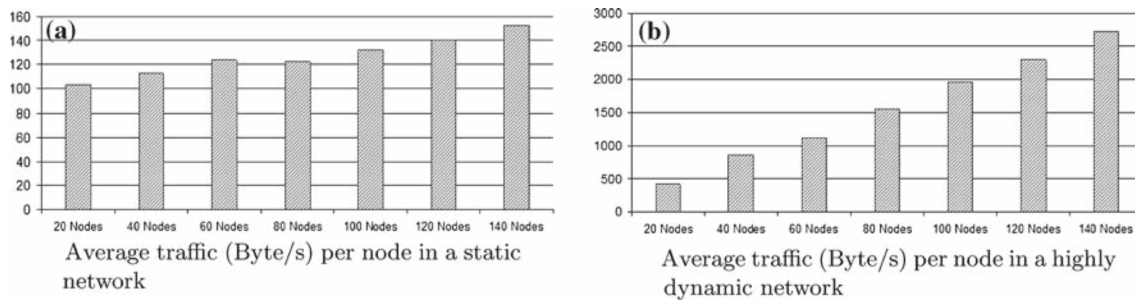

Fig. 6 Traffic of nodes in networks of various sizes

in a consistent state, and in a highly dynamic network, where every 5 s 10% of the nodes were deactivated and reactivated again after 15 s. The size of the network was 140 nodes, the monitoring interval was set to 2.5 s, and the simulations were performed for group sizes of 12, 16, 20, 24, 28, 35, 50 and 80 nodes per monitor.

As Figs. 5(a) and (b) demonstrates, the total traffic of the network grew with the number of monitors. Especially in the static environment, where incremental monitoring reduced the traffic significantly, the communication overhead of too many monitors became obvious. On the other hand, if too few monitors were elected, their groups were too large, which resulted in a high individual load on each of them (see Figs. 5(c) and (d)). Scenarios like this call for a compromise. Following the average traffic of the monitors, one can observe that their load was not decreased relevantly any more if the groups became smaller than 24 nodes. Moreover, the total traffic in the network was growing significantly then. Therefore we regarded a group size of 24 nodes per monitor as a reasonable compromise for the systems default configuration and used it during the rest of the simulations.

As the replicator system is based on a global view, its scalability is obviously limited, since a changed state has to be propagated to all nodes. However, comparing this disadvantage with the benefits a global view is providing for synchronization, and taking also into consideration that mobile ad-hoc networks do usually consist (at most) of a few hundred nodes, the facts militate in favor of a global view approach. Particularly, because of the limited scalability, it was necessary to find out how the system performs in networks of different sizes, regarding its produced traffic.

Our test environments consisted of 20, 40, 60, 80, 100, 120, and 140 nodes, the monitoring interval was again 2.5 s, and as a result of the first set of simulations we used a monitoring group size of 24 nodes. Again we examined the traffic in static networks and in dynamic ones, where every 5 s for 10% of the network's nodes a disconnection was simulated.

Figure 6(a) presents the average traffic per node in the static networks. Although everything was in a consistent state and only small amounts of data had to be propagated due to incremental monitoring, the traffic was slightly increased with the growing size of the network. This mainly took place due to the fact that a larger number of nodes automatically implied a larger number of monitors, which were exchanging data. In contrast, the average traffic per node in the dynamic networks (see Fig. 6(b)) was growing proportionally, due to the changes which had to be propagated to all nodes. As a consequence of these testings, it is safe to say that in both environments, the static and the dynamic one, the per-node traffic was growing linearly with the size of the network, which of course resulted in a quadratically growing total traffic. However, in general, the gradient of the curves is mainly impacted by the level of dynamics in the network. The less changes occur, the more gently inclined the curves are. This leads to the conclusion that the global view approach is, in spite of its scalability limitations, capable of being applied in mobile ad-hoc networks which are on the one hand unpredictable, however, on the other hand, are usually much less dynamic than the test environments of our case study.

3.2.2 *Response time*

The last simulations of the case study served the purpose of determining a good trade-off between the need of a low network traffic and the need of a quick response time. Obviously, this poses a conflict, since both requirements are contradictory to each other. For this reason we tested the replicator system in identical environments, but with different monitoring intervals (2.0–6.0 s) in order to compare the curves of average response times and traffic per node.

The response time was almost solely impacted by the monitoring interval (see Fig. 7(a)) and, therefore, was

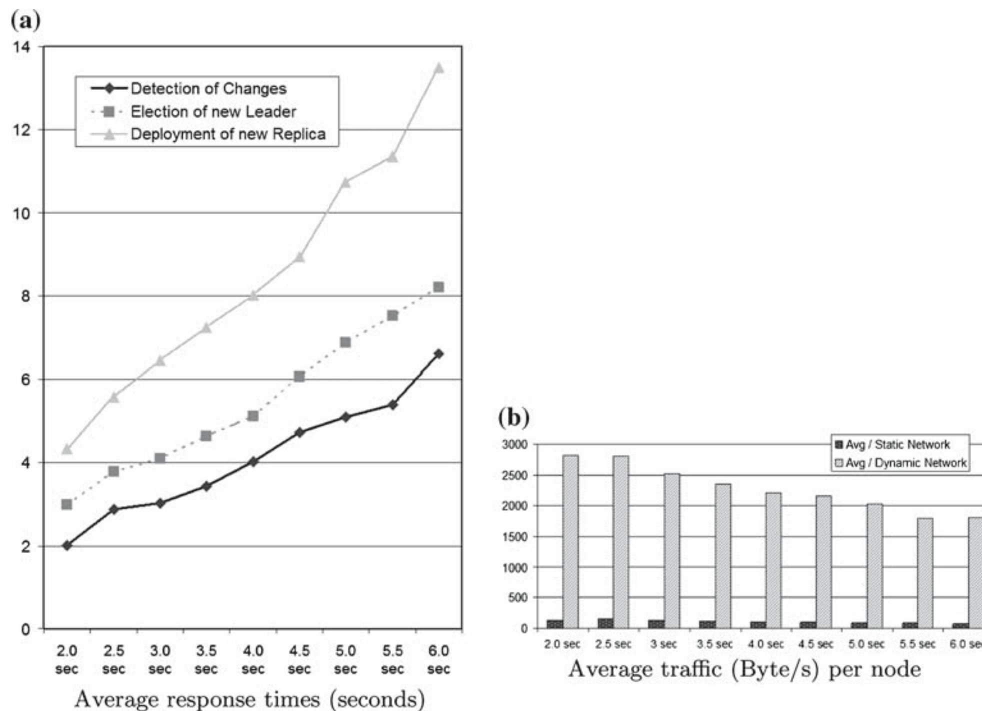

Fig. 7 Traffic and response times depending on the monitoring interval

growing proportionally with it. As anticipated, the time until a node detected a change in the network (e.g., failure, new leader) averaged the monitoring interval, since it usually takes two cycles (as explained in Sect. 2.2.3) and each cycle has an expectation of half the interval.

The election of a new leader took slightly longer, since leaders are preferably placed on the monitoring nodes by the election algorithm. This has the advantage that they are informed faster about changes in the network (one cycle instead of two), and thus are also able to react faster. However, if a node which is a monitor and a leader at the same time fails, all nodes which were using it for retrieving the state (passive monitoring) have to select a new monitor and to retrieve the state afresh, which causes a delay.

The last part of the simulation dealt with the necessary time to deploy a new replica after an old one failed. Since the performance of this task usually also depends on the size of the Web service archive, we kept it small (3 kB) in order to determine the actual response time of the system and to avoid delays caused by transferring large archives. Figure 7(a) demonstrates that on average this task took twice the time as for detecting a simple change, viz. one interval to detect the change and one to react. Although the decision how to react is done immediately, this delay is happening since the command is postponed to the next cycle, as explained in Sect. 2.2.4.

In a nutshell, all these simulations made clear that the response time grows proportionally with the monitoring interval. In contrast, the produced network traffic is not increased proportionally with a smaller interval, as the results in Fig. 7(b) show. This convenient behavior is the consequence of the incremental monitoring, where changes have to be propagated only once and subsequent requests receive an empty response. Hence, although choosing a smaller monitoring interval, in order to be able to react faster to changes in the network, does not affect the produced traffic significantly, it does affect the CPU load. Especially for the monitoring nodes, which have to service requests of groups of approximately 24 nodes and which may have only limited resources (e.g., older PDAs) and other software running concurrently, this might be a criterion for avoiding very short intervals. However, today it is highly probable that a mobile ad-hoc network consists of at least a few nodes which are more powerful (e.g., notebooks, newer PDAs), and which will be then preferably elected as monitors. Therefore, we do not see the necessity of using longer intervals, sacrificing quick response times this way, and regard an interval of 2.0 s as favorable.

3.2.3 Results

The result of this case study is mainly that we gained insight in to the system's behavior in various simulated ad-hoc networks of different parameters. Moreover, we determined a default configuration which represents a good trade-off for most of the mobile ad-hoc networks. These values can be tweaked if the system is going to be applied in untypical environments or if a more optimal configuration, suited perfectly to the destination environment, has to be used.

4 Related work

Birman et al. [25] present several useful extensions to Web services for self-diagnosis and self-repairing, which, however, are not suited to ad-hoc networks. They distinguish between monitoring of single components, on the one hand, and aggregated properties of the system, on the other. 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 missing availability, giving them them opportunity to roll over to backup resources.

Dekel et al. [26] present with "Easy" a system which addresses performance-aware high availability by using replication. Although this solution deals neither with Web services nor with replication in dynamic networks, it provides a quite detailed list of service aspects which have to be taken into account while replicating.

With "WS-Replication" [13] Salas et al. propose an infrastructure for WAN replication of Web services. It uses group communication based on SOAP-multicast and provides a transparent replication and fail-over. Although this infrastructure also is able to deploy Web services on remote sites, the concept is not suited for mobile ad-hoc networks because of its bandwidth-consuming SOAP-multicast and the lack of a dynamic replica placement.

Fault tolerant SOAP (FT-SOAP) [27], developed by Liang et al., provides a primary copy-based replication of Web services, which uses an extension to WSDL, pointing to a group of replicas. Since this group is static, this approach is unusable for our problem.

Ye and Shen [14] introduce a middleware that supports reliable Web services built on active replication. Their system is based on proxies which multicast requests to the replicas and return the results to the clients. Furthermore it contains a suppression of duplicate messages. Again this system is not applicable in dynamic networks, due to static resources.

Jeckle's and Zengler's Active UDDI [28] is an extension to the UDDI's invocation API in order to enable fault-tolerant and dynamic service invocation. It is able to detect changes in availability of services and replaces unavailable services with alternative ones from the registry, which provide the same functionality. However, this approach neither replicates services actively nor does it ensure synchrony of stateful ones.

Friedman [29] developed a concept for partial caching of Web services in ad-hoc networks. His solution places proxy services in an optimal manner, which takes the structure of the network as well as qualities of connections into consideration. However, proxy services still rely on the initial instance and therefore do not provide fault tolerance.

5 Conclusions

In this paper, we presented a solution for dynamic replication and synchronization of stateful Web services in mobile ad-hoc networks. Following the requirements, which ad-hoc networks pose to a replication approach, we have developed a system which is highly flexible in order to handle all the difficulties caused by unpredictable topologies. Our approach is completely decentralized, places replicas in a dynamic manner by following the requirements of the services, produces a low amount of network traffic, and, furthermore, makes the invocation of replicated Web services convenient for the clients. This makes it a generally applicable solution which can be used in all kinds of networks which require fault-tolerant Web services.

Furthermore, we analyzed selected aspects of the system and tested them in a case study, simulating large-scale ad-hoc networks. This way we were able to evaluate how the system performs in dynamic environments. These insights allowed to determine important configuration parameters.

Hence, the contribution of our paper is a solution that combines already existing and newly developed ideas for replication to a system which makes it possible to apply reliable Service Oriented Architectures even in typically unreliable network environments.

5.1 Future work

The main drawback of the currently existing replicator system is that it places the replicas solely by taking performance criteria into consideration and ignores the structure of the network. This makes the worst case possible where all replicas are located next to each other and a disconnection of a single router makes all of them

unavailable. This is a problem which can be solved if methods for prediction of ad-hoc network partitioning [30,31] are applied and the replicas are placed wisely, for instance, by following the ideas in [32–34]. Once these strategies are developed and implemented, they can be attached to the replicator system as a plug-in to the placement mechanism.

References

1. Berger S, McFaddin S, Narayanaswami C, Raghunath MT (2003) Web services on mobile devices—implementation and experience. WMCSA, IEEE Computer Society, Washington, pp 100–109
2. Gehlen G, Pham L (2005) Mobile web services for peer-to-peer applications. CCNC, IEEE Computer Society, Washington, pp 427–433
3. Lee W, Lee K, Lee S (2006) Intermediary based architecture for mobile web services. In: ICACT, IEEE Computer Society, Washington, pp 1973–1978
4. Schall D, Aiello M, Dustdar S (2006) Web services on embedded devices. Int J Web Inf Systems 2(1):1–6
5. Steele R (2003) A web services-based system for ad-hoc mobile application integration. ITCC, IEEE Computer Society, Washington, pp 248–252
6. Jørstad I, Dustdar S, Thanh DV (2005) Service-oriented architectures and mobile services. In: Castro J, Teniente E (eds) CAiSE workshops (2), FEUP Edições, Porto, pp 617–631
7. Dorn C, Dustdar S (2006) Achieving web service continuity in ubiquitous mobile networks the srr-ws framework. UMICS
8. Jørstad I, Dustdar S, Thanh DV (2005) A service oriented architecture framework for collaborative services. WETICE, IEEE Computer Society, Washington, pp 121–125
9. Schreiner W, Dustdar S (2005) Collaborative web service technologies. CCE
10. Dustdar S, Fenkam P (2004) Formally designing web services for mobile team collaboration. EUROMICRO, IEEE Computer Society, Washington, pp 469–476
11. Dustdar S, Gall H, Schmidt R (2004) Web services for groupware in distributed and mobile collaboration. PDP, IEEE Computer, Society Washington, pp 241
12. Juszczyk L, Lazowski J, Dustdar S (2006) Web service discovery, replication, and synchronization in ad-hoc networks. ARES, IEEE Computer Society, Washington, pp 847–854
13. Salas J, Perez-Sorrosal F, Patino-Martinez M, Jimenez-Peris R (2006) Ws-replication: a framework for highly available web services. WWW, ACM
14. Ye X, Shen Y (2005) A middleware for replicated web services. ICWS, IEEE Computer Society, Washington, pp 631–638
15. OASIS (2001) Universal description, discovery and integration. http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm
16. Felber P, Schiper A (2001) Optimistic active replication. ICDCS, pp 333–341
17. Budhiraja N, Marzullo K (1992) Highly-available services using the primary-backup approach. Workshop on the Management of Replicated Data, pp 47–50
18. Fang CL, Liang DR, Chen C, Lin P (2004) A redundant nested invocation suppression mechanism for active replication fault-tolerant web service. EEE, IEEE Computer Society, Washington, pp 9–16
19. MortBay (2006) Jetty, Java HTTP server and servlet container. http://jetty.mortbay.org
20. Apache (2000) Axis SOAP implementation. http://ws.apache.org/axis/
21. Dustdar S, Treiber M (2006) Integration of transient web services into a virtual peer to peer web service registry. Distributed Parallel Databases 20:91–115
22. Apache (2003) Web service invocation framework. http://ws.apache.org/wsif/
23. Enhydra (2003) kSOAP. http://ksoap.objectweb.org/
24. Hubert B (2004) Linux advanced routing & traffic control. http://www.lartc.org/
25. Birman KP, van Renesse R, Vogels W (2004) Adding high availability and autonomic behavior to web services. ICSE, IEEE Computer Society, Washington, pp 17–26
26. Dekel E, Frenkel O, Goft G, Moatti Y (2003) Easy: engineering high availability qos in wservices. SRDS, IEEE Computer Society, Washington, pp 157–166
27. Liang D, Fang CL, Chen C, Lin F (2003) Fault tolerant web service. APSEC, IEEE Computer Society, Washington, pp 310
28. Jeckle M, Zengler B (2002) Active uddi—an extension to uddi for dynamic and fault-tolerant service invocation. In: Chaudhri AB, Jeckle M, Rahm E, Unland R (eds) Web, web-Services, and database systems. Volume 2593 of Lecture Notes in Computer Science., Springer, Heidelberg, pp 91–99
29. Friedman R (2002) Caching web services in mobile ad-hoc networks: opportunities and challenges. POMC, ACM, pp 90–96
30. Derhab A, Badache N, Bouabdallah A (2005) A partition prediction algorithm for service replication in mobile ad hoc networks. WONS, IEEE Computer Society, Washington, pp 236–245
31. Milic B, Milanovic N, Malek M (2005) Prediction of partitioning in location-aware mobile ad hoc networks. HICSS, IEEE Computer Society, Washington
32. Hara T (2001) Effective replica allocation in ad hoc networks for improving data accessibility. INFOCOM, pp 1568–1576
33. Hara T (2005) Data replication issues in mobile ad hoc networks. DEXA Workshops, IEEE Computer Society, Washington, pp 753–757
34. Ishihara S, Tamori M, Mizuno T, Watanabe T (2004) Replication of data associated with locations in ad hoc networks. Mobile data management, IEEE Computer Society, Washington, pp 172