Web Service Discovery, Replication, and Synchronization in Ad-Hoc Networks

Lukasz Juszczyk, Jaroslaw Lazowski, Schahram Dustdar
{ljuszczyk, jlazowski, dustdar}@infosys.tuwien.ac.at

Vienna University of Technology
Distributed Systems Group
Vienna Internet Technologies Advanced Research Lab
Argentinierstraße 8, A-1040 Wien

Abstract

Mobile ad-hoc networks with their arbitrary topologies are a difficult domain for providing highly available Web services. Since hosts can move unpredictably, finding services and featuring their constant and reliable functionality poses a challenge. In this paper we present a flexible system which is not only bound to ad-hoc networks, but can be used in any other environment. Our solution offers a discovery technique which keeps UDDI information in distributed registries up-to-date and includes a replication and synchronization mechanism which provides backup services for a highly increased service dependability.

Keywords: Mobile Ad-Hoc Networks, Peer-to-Peer, Web Services, Discovery, Registry, Replication, Synchronization

1. Introduction

The main advantage of mobile ad-hoc networks is the self-configuring and self-maintaining topology. But unpredictable movements of hosts make it difficult to provide a reliable *Service Oriented Architecture* (SOA) where services are required to be easily found and constantly available. If a host disappears or changes its position, information about the service location in WSDL data [10] becomes invalid, clients are not notified about these changes, request fulfillment is not possible any more, and the whole workflow is brought to a halt. In such a situation we are confronted with the problem of applying an architecture which was not designed with dynamic networks in mind to a highly dynamic environment. The presented solution focuses on this by introducing a *Web Service Discovery and Registry* system for having up-to-date UDDI information [9] and a *Web Service Replication and Synchronization* mechanism which strongly improves reliability of services within typically unreliable network environments. Both parts were designed with simplicity and flexibility in mind and can be used separately. Nevertheless, this paper describes the combined solution which offers a light-weight infrastructure based on Apache Axis [2] and jUDDI [6], open to be applied in any kind of network.

Usually Web service registries have a well-known and static location and are managed by a handful of administrators. This method is not feasible within ad-hoc networks. Our *Web Service Discovery and Registry* system sorts out this problem by combining different benefits from existing solutions [8, 5, 19, 11] and providing an automatic discovery functionality which keeps track of alternating network structures. Descriptions of Web services are published in distributed UDDI registries and updated automatically once a service ceases to be available or changes its location.

This is combined with the approach of replicating services, to ensure the whole workflow in the network is not interrupted by failures or simple relocations of hosts. Although many papers propose static replication of services to gain a higher availability, there is again a lack of answers to this problem within networks with transient topologies. Our *Web Service Replication and Synchronization* system is self-configuring, uses flexible algorithms which adapt their behavior to the dynamic characteristics of the network, and is able to perform the replication regarding performance properties of hosts and requirements of services. Furthermore, we are providing an interface which enables to apply replication strategies suited for the individual requirements.

Web services are getting more and more popular since they provide the possibility of a standardized, platform independent, and easily extensible communication. We are currently implementing a prototype of the described system and detailed measurements will determine the area of applicability. These may be for instance the needs of groups of engineers at conferences, action forces in emergency situations, ad-hoc BPEL-processes [4], etc.

0-7695-2567-9/06 $20.00 © 2006 **IEEE**

IEEE COMPUTER SOCIETY

This paper is organized as follows. Section 2 starts with describing possible faults in ad-hoc networks. Section 3 gives an overview of the service discovery and registry system while Section 4 explains the replication and synchronization methods and Section 5 the practical application of our solution. Sections 6 and 7 finally present our review of related work and conclusions.

2 Classification of Faults

While designing a fault-tolerant infrastructure for Web services, it is important to identify possible kinds of faults first and to specify how they are classified and handled. Especially in ad-hoc networks a host can become unavailable due to many reasons, such as crashes or long delays caused by communication media:

- *Unavailable hosts:*
  - Crash
  - Shutdown
  - Unstable network communication
  - Moving out of wireless network range
- *Unavailable Web services:*
  - Crash or shutdown of Web service container
  - Crash or undeployment of Web services
  - Firewalls blocking
- *Delays:*
  - High load on host or its Web services
  - High load on network link
  - Unstable network communication
- *Changing network topology:*
  - Relocating hosts and routers
  - Splitting ad-hoc networks caused by router movements

It is not reasonable to distinguish between faults caused by hardware-failures, software obstacles such as firewalls, or greater delays due to unstable network links, etc. Furthermore it is often impossible to find out the reasons for unavailability. Therefore checking and monitoring is performed by simple ping-like requests sent to services, awaiting a response within a defined timeout interval. This way slow hosts are sorted out automatically as bottlenecks, even if they are available.

Alternations in the topology, such as relocations or changing addresses, can be often handled by assigning *Universally Unique Identifiers* to hosts and this way tracking their movements.

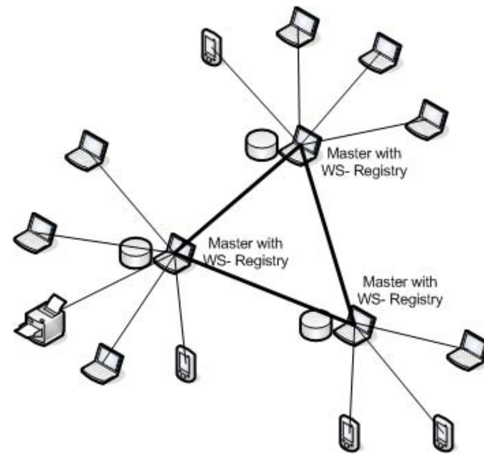

Figure 1. Network Topology with three Star-structures

3 Web Service Discovery and Registry

A main requirement of SOAs is the ability to find Web services by querying UDDI registries. Usually the location of both the registries and the services is static and changes only rarely. Following the dynamic characteristics of ad-hoc networks it is necessary to use a completely different approach, by avoiding static centralization as far as this is possible. Our solution fulfills this by partitioning the network into autonomous groups and running distributed registries able to react on failures and disconnections by restoring the lost service-descriptions quickly.

3.1 Structuring the Network

Since ad-hoc networks can grow rapidly and are most often established using slow wireless communication, it is necessary to keep network traffic as low as possible by avoiding broadcast communication in a large scale. Considering this restriction we are partitioning the network into several independent groups of hosts which are organized in star-topologies, as shown in Figure 1. Each group actively monitors its members and consists of one controlling *master-host* and an undefined number of *common nodes*:

- *Master-hosts:*

  Master-hosts have sufficing resources (mainly CPU and memory) for running a light-weight jUDDI registry plus the necessary discovery service. Each group of hosts is controlled by a master, which keeps track of other masters and acts as an access-point to retrieve information about existing hosts and their groups. Figure 2 illustrates the component-architecture, showing the modules of the network management coexist-

IEEE COMPUTER SOCIETY

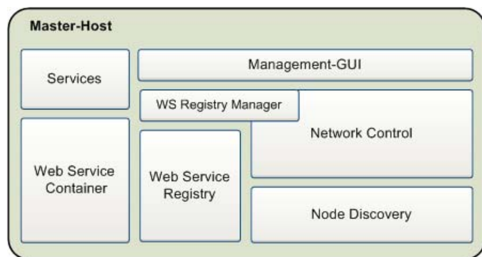

Figure 2. Component Architecture of a Master-host

ing with a Web service container. Apart from the management task, master-hosts are usual nodes which can invoke and provide own Web services.

- *Common network nodes:*

  PDAs, peripheral devices such as printers or scanners, etc. are also allowed to join the network and provide services. But due to lacking performance or possibilities of running the management-code, they are not qualified to become a master.

For building groups, common nodes joining the network use JXTA [7] to discover all master-hosts from which one has to be chosen as the leader. The procedure of deciding whether a host is going to become a master or should join another one can be done either manually by offering this functionality to the user in a GUI, or automatically by performing segmentation-algorithms, which have the handicap of posing an NP-complete problem. Such algorithms can use context-specific information for dividing the network into groups of hosts sharing defined properties or groups representing the physical structure of the network. After joining a group, JXTA is deactivated on the nodes and periodical heartbeat-messages are sent to the master-host, confirming the current availability. These heartbeats allow to detect changes in the topology of the group, such as moving or disconnecting common nodes and master-hosts. After detecting a disconnection of its master, every node of the group has to redo the discovery-process to find a new one.

3.2 Service Discovery and Registry

The idea of UDDI Web service registries is to host information about miscellaneous services at a well-known location and to provide a convenient possibility of querying it. Even in ad-hoc networks, this implies a certain centralization, since it is not reasonable to expect every node to run its own registry. We can accept centralization only if we ensure that registries are notified about the availability of services and rebuild themselves quickly after failures or disconnections. Furthermore it is necessary to place these registries at locations which are easy to find for all clients. As a matter of course master-hosts pose a perfect destination. Their locations are known by all nodes and the monitoring-functionality makes is possible to keep UDDI-information up-to-date. By using non-replicated distributed registries the costs of recovering data of disconnected ones are kept low, speeding up the process of repopulation. But unfortunately, running multiple registries also implies an increased probability of failure of one of them, compared to a single registry. This can be controlled by choosing partial-replication [16] of UDDI-data to avoid situations where records without backups are lost, preventing clients from finding services until the recovery process is completed.

To populate the registries, all nodes are required to publish lists of their services including meta-data which are used for assembling the service-descriptions. This is done immediately after a node is successfully connected to its master-host, as described in Section 3.1. Nodes which lost connection to their group are detected by missing heartbeat-messages and all descriptions of their services are deleted from the registry. Moreover services being deployed or undeployed on running hosts are detected by our extension of Apache Axis and their registries are notified. This way the UDDI-data can be kept in a consistent state all the time.

An aspect of distributed registries is the required multicast of a query. This can be optimized if the structuring of the network was done by using context-specific segmentation-algorithms. In such a case it is possible to select only groups which might contain the desired web-service and to direct the queries there, ignoring the rest. A good example might be a network with business-processes provided by different companies, where each group represents a single company. Searching for specific services could be done by querying only registries of the desired company.

4 Web Service Replication and Synchronization

Our goal was to create a mechanism for replication and consistency of replica-states, flexible enough to deal with the problems in ad-hoc networks, and nevertheless to be usable within any other network. As mentioned in the Introduction part, the discovery-system can be used separately from the replicator. Therefore, the replicator is not bound to specific network-environments or protocols, and just requires a periodically actualized list of host-addresses, while all specific checking and monitoring is done automatically. For this reason we cannot use hard-coded methods for finding new hosts, but instead we are opening a plug-in interface, used for notifying about the states of existing and

IEEE COMPUTER SOCIETY

new hosts in the network. The only connection between the replicator and the discovery system described in Section 3 is realized as such a plug-in.

The whole replication system can be described as a co-operation of these three separate parts (see Figure 3): *Replicator Web Service*, *Replica Placement Mechanism*, and *Service State Synchronization*.

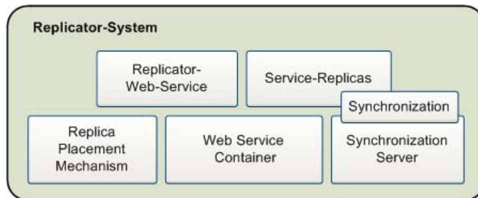

Figure 3. Replication and Synchronization Components

4.1 Replicator Web Service

For moving Web services from one host to another, a hot deployment of services is mandatory. This means, services should be able to be installed and uninstalled at running hosts without the necessity of restarting the service container or any other part of the system, which would interrupt the workflow. We decided to implement the necessary functionality for this also as a Web service. The main reason for that is the convenient possibility of talking to a destination service which is operating in the same run-time environment as the replicas. This allows to perform easy checks for necessary classes, libraries, system attributes, and other properties relevant for replication.

We have chosen Apache Axis as the container for the replicator service and all deployed replicas, because of its open-source license, speed, flexibility, and useful functionality such as request-handler chains. Hence, this constrains all services to be Axis-compatible by providing deployment descriptors in WSDD-format [1] plus an additional configuration-file with preferences for replication. The replicator service provides the following functionality:

- *Send and receive service archives:*

  Replication can be done in a push or pull manner. All necessary files for deploying (e.g., property-files, deployment descriptors, etc.) and for working (e.g., certificates, configurations, images, etc.) have to be contained in this Jar-archive.

- *Undeploy, redeploy, and delete services:*

  If a service is not needed at the moment but is required to be able to be reactivated later, it is possible to undeploy it and redeploy it at a later date. This can speed up replication within networks with a fast alternating structure since it is possible to hibernate services which can be woken up very quickly.

- *Check and register libraries:*

  If services require external code in Jar-archives, it is necessary to send and register them at the destination host's class-loader. Libraries are identified by checksums to avoid deploying identical copies with different namings more than once.

- *Return host and service properties:*

  These functions are called by the monitors to collect properties of hosts and services in the network. The most important properties are hardware parameters, timestamps of deployments, links to hosts controlling service replication, etc.

The replicator service works completely in passive mode, viz it only responds to commands of clients without invoking anything independently. All active monitoring and replication logic is part of the *Replica Placement Mechanism*.

4.2 Replica Placement Mechanism

The main functionality of this mechanism is monitoring of hosts and services, leader election, and the whole replication logic. While designing the algorithms, a top priority was given to a behavior, adapting quickly to changes in the network. This is achieved by avoiding voting-mechanisms to find leaders, but rather using solutions which are able to decide autonomously by comparing properties of the local host to the monitored data.

4.2.1 Monitoring

Autonomous calculations, such as leader-elections, are based on a global view of the network, which has to be periodically monitored. Every single host has the capability of monitoring, but for better scalability and load-balancing we elect a number of `max(2,numberOfHosts/50)` hosts with the least used network bandwidth as our monitors and partition the network into groups, each one observed by a single monitor. Information about changes in state and availability of hosts and services is exchanged between the monitors, so each of of them is able to offer a complete view of the network. Election and monitoring is done as described in Listing 1.

IEEE COMPUTER SOCIETY

Listing 1. "Monitoring"

```
// adapt intervals to network size
loop in intervals {
  monitors = list fastest hosts
  // is localhost a monitor?
  if (localhost within monitors) {
    // adjust host lists with monitors
    get new host addresses from monitors
    // which hosts shall be checked?
    // group them by addresses
    calculate list of hosts to check
    // the actual monitoring
    check hosts and their services
    // get information about other hosts
    exchange data with other monitors
  } else {
    // use another monitor
    fetch random monitor and retrieve data
  }
}
```

Random selection of a monitor-host provides a simple but adequate way of load balancing. Nevertheless the performance may sink, due to other applications or fluctuations in the quality of the connection. In this case the health-properties of the monitor-host will drop, which leads to moving this task to another destination. To relevant information about hosts and services is retrieved from the *Replicator Web Service* which is running on every host accepting service replicas. For evaluating the *health* of the hosts the following properties are used: free CPU cycles, amount of free memory, amount of free disk-space in the working directory, free network bandwidth, and the estimated time left to live which can be calculated using battery statistics.

Unfortunately, in ad-hoc networks one cannot rely on hosts sending events notifying about their disappearing. This is why all the monitoring has to be active and in variable intervals, depending on the size of the network. This raises the problem that the monitored state can become outdated a short time after being retrieved and the algorithms have to perform their calculations using old data. Therefore, this may cause short inconsistencies, such as to many or to few monitoring hosts or replication leaders. Since it is the duty of our algorithms to follow this fact, they are designed to correct it after the next cycle of the monitor. Hence, we have a flexible system that accepts short inconsistent conditions, which are rectified as soon as possible.

4.2.2 Replication Leader Election

For keeping replication strategies 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 in Listing 2 to find out who is the leader.

Listing 2. "Leader election algorithm"

```
after every monitor-cycle {
  if (service not deployed on other hosts) {
    declare localhost as leader
  } else {
    // find all leaders in the net
    // sort by popularity
    leaders = list leaders of service
    // if more than one leader exists
    // grep most popular one
    if (at least 2 leaders equally popular){
      // may be localhost too
      declare fastest one as new leader
    } else {
      // may be localhost too
      accept most popular leader
    }
  }
}
```

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. After a disconnection the fastest host will be elected, although performance is not very important for this task. Another feature is the merging of different leaders within one calculation-cycle. This becomes necessary if an ad-hoc network was split into many sub-nets, these sub-nets elected their leaders and after a time they merged again.

4.2.3 Replication Logic

Since requirements of applications and network environments can vary extremely, we avoid prescribing defined replication strategies, but allow applying custom solutions suited to the individual needs. For giving this freedom, a plug-in interface is provided, which is supported by a collection of classes implementing functions for convenient moving and synchronizing of services in the network. For example one could adapt the *P-Grid* [13] peer-to-peer platform with its sophisticated and flexible management system to distribute Web services. In case no custom plug-in was specified, a built-in replication method, called *SimpleReplicationLogic*, is used. As Listing 3 shows, its behavior is very simple but nevertheless adequate for the usual needs.

Listing 3. "Simple Replication Logic"

```
after every leader-elector-cycle {
  sort hosts regarding their properties
  if (number of replicas too low) {
     // need more running replicas
     if (service is somewhere hibernated) {
        wake up
     } else {
        send new replicas to fastest hosts
     }
     synchronize new replicas
```


IEEE COMPUTER SOCIETY

```
    }
    if (number of replicas way to high) {
        delete surplus replicas on slowest hosts
        // leave some services hibernated
    }
    if (number of replicas slightly to high) {
        hibernate replicas on slowest hosts
        // can be maybe used later
    }
    if (replicas exists on transient hosts) {
        // hosts have only little time left
        move services to other/fastest hosts
        synchronize new replicas
    }
}
```

All services have to declare the minimum and maximum allowed number of replicas within a single monitored network. We do not want to allow them more control of the replication process, since we believe this should be handled by the replication logic.

Remembering the problem with short inconsistencies after merging several sub-networks, as described in Section 4.2.1, it becomes obvious that for a short time more than one leader can exist, even if this is fixed by the election-algorithm in the next cycle. Nevertheless, this could cause a situation where the leaders want to perform some modifications concurrently, causing collisions or another inconsistent state. So it is necessary to postpone these modifications to the next round to ensure that only one leader will be left and all other hosts are forced to abandon their modifications.

4.3 Service State Synchronization

Although replication provides the advantage of enhancing the availability of Web services within any network strongly, it can consume a lot of bandwidth since all replicas have to be synchronized. This can be avoided if the number of replicas is not chosen too high or only services are replicated which do not need a lot of data to be exchanged to keep a consistent state. While mobile ad-hoc networks are not suited for synchronization of services with large amounts of data, this problem disappears on faster networks. Our system does not set any limits but instead we let it up to the users to choose their replicated services wisely, depending on the network environment.

For a clearly arranged synchronization process, a simplified *primary-copy* approach [15] was chosen. This means, all requests from clients are sent to the primary host, which is in our case the replication leader for this service. This host has to notify all replicas of changes in the state of the service. Another alternative to *primary-copy* is the *active replication* approach, where requests are sent to all replicas and all individual responses are received by the caller. This method requires client-multicasts for all invocations and thus would slow down service-calls and make re-synchronization of split networks more complicated.

In our implementation, changes in the state of a service are tracked by declaring all relevant variables in a container-class. After every invocation of the service the container checks for alternations in the state and notifies all known backups about them. Since SOAP-calls would slow down the whole synchronization process, all communication is done via a TCP-based protocol to a small server which is running on a fixed port and serving only synchronization requests. For developers it is recommended to write and register so-called *synchronization-helper-classes*, which can be compared to serializers and deserializers in SOAP implementations. These are used for calculating state changes of an object and to serialize these changes. After being received on the destination host the changes are applied to the state object, using the same classes. This method helps to reduce network traffic, because only necessary changes are being sent instead of the whole objects. Nevertheless this is not mandatory, since otherwise the state is serialized by using a *Java-ObjectStream*.

Unfortunately, it is not always possible to ensure perfect state-consistency in ad-hoc networks which can split and merge again. This problem is explained using the following example, where a ticket-distribution Web service is replicated:

1. The replication mechanism spawns new replicas, which are synchronized.
2. A laptop, establishing this network by acting as a router between two sub-nets, is suddenly shut down. Now two autonomous nets exists, having no information about each others any more, but both containing replicas of this service.
3. Both nets act completely individually, managing their replicas which are synchronized among themselves but not with the ones from the other net. Clients are changing the states of the services continuously by requesting new tickets.
4. The connecting router is up and running again, merging the sub-nets.
5. The newly merged network now contains clients, which received the same ticket. In this case synchronization of service states is impossible without declaring one of them as invalid.

Situations like this one are handled, by regarding the service which was invoked most often as all-dominant and replacing all other states. As a result of this, a part of the clients has formerly received responses based on a service-state which was later nullified.

IEEE COMPUTER SOCIETY

This example demonstrates the issue, that perfect synchronization of states is impossible in highly dynamic networks. Again replicated services have to be chosen respectively to the network environment. While it is not always feasible to ensure ideal synchronization consistency in mobile ad-hoc networks, this issue disappears in less dynamic networks, where parts of the network are not likely to disconnect.

5 Practical Application

In the previous chapters we have described an infrastructure, which helps to build SOAs within dynamic and unreliable networks. But this raises the question, how client-software can stand to benefit from all these features. Of course the UDDI registries can be browsed the usual way, but what about the invocation of replicated services? Every function-call should be directed to the primary-copy and never to the backups. But where can a client find the location of the primary-copy?

For this reason we have written a simple utility which queries the registries for locations of services and returns the corresponding WSDL-file, pointing to the correct location. To achieve this, the following steps are necessary:

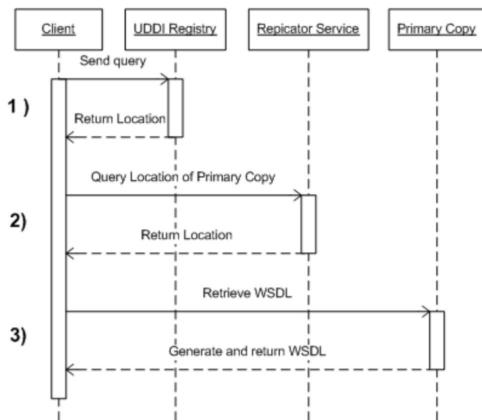

Figure 4. Automatic Retrieval of WSDL-data

1. Find a host which is running the service, even if this is a replica. This can be done easily by querying the registries.
2. Ask the replicator-service running there for the location of the service's primary-copy.
3. Request the Apache Axis container to return an automatically generated WSDL-file for this service.

The returned WSDL-file could then be used as input for the *Apache Web Service Invocation Framework* [3], which enables calling Web services in a convenient way, without having to take care manually of all necessary API-calls. Changes in the availability of a service are noticed and handled by querying the monitoring-hosts and requesting a new WSDL-file from the new primary-copy.

6 Related Work

6.1 Host- & Service-Discovery

WS-Discovery [11] is a new discovery technique, especially for peripheral devices, which will probably replace *Universal Plug and Play* in the near future. Queries for services are sent to all nodes using a multicast discovery protocol, while optional discovery proxies can be used to scale to a large number of endpoints. Nevertheless multicast queries consume a lot of network bandwidth.

DEAPspace [19] provides an infrastructure for a decentralized discovery and description of services on pervasive devices. It uses proactive single-hop broadcasts, and therefore is only usable in short-range networks. Furthermore it does not focus on Web services and registering them.

The *Service Location Protocol* [8] provides a scalable framework for discovery and selection of network services. SLP uses unicast- and multicast-communication, and thus floods the network with search-messages, which increases network usage and thus power consumption.

The *DIANE*-project [5] deals with semi-semantic overlays, structuring the network into logical groups, called *DLanes*. Each service's description is propagated through them, notifying hosts about all services on the current lane. Holding up the structures in the network causes a high overhead, resulting in a reduced flexibility in fast changing network topologies.

6.2 Replication & Synchronization

WS_Reliability and *WS_ReliableMessaging* [12] enable to send a request through intermediate hosts to a Web service while it is unavailable, guaranteeing that it will be processed At-Least-Once, At-Most-Once or Exactly-Once. But if the target service disappeared forever or a quick response is desired, it is inevitable to use replication.

A lot of interesting ideas about replication and synchronization can be found in *Easy* [17]. Although that solution is neither focused on Web services nor suited for highly dynamic network environments, it provides useful guidelines for replicating system states.

Friedman presents in [18] ideas about caching parts of Web service data for reducing load and traffic. It places the caches regarding the structure of the ad-hoc network but all of them are still dependent on the primary service, which is still the single point of failure.

IEEE COMPUTER SOCIETY

The solution in [14] provides extensions to the Web service architecture, featuring detection and reporting of failures, reliable messaging and multicasts for replicated servers. Unfortunately some important parts of the system, such as event notification, monitoring, and distributed control are not usable within ad-hoc networks.

7 Conclusion

Our contributions of this paper include (a) making UDDI registries usable within ad-hoc networks by segmentation, (b) replication of services for increased availability, and (c) synchronization of stateful services. The prototype of our solution suggests an enhancement to *Service Oriented Architectures*, which are usually applied in managed networks. Enabling SOAs within dynamic ad-hoc networks opens a lot of new possibilities, by offering a standardized way of communication in spontaneously established networks, especially interesting for cooperating groups of engineers.

References

[1] Apache Axis Deployment Descriptor Reference. http://www.osmoticweb.com/axis-wsdd/.

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

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

[4] Business Process Execution Language for Web Services. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.

[5] DIANE. http://hnsp.inf-bb.uni-jena.de/DIANE/.

[6] Java Implementation of UDDI. http://ws.apache.org/juddi/.

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

[8] Service Location Protocol. http://www.openslp.org/doc/rfc/rfc2608.txt.

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

[10] Web Services Description Language. http://www.w3.org/TR/wsdl.

[11] Web Services Dynamic Discovery. http://msdn.microsoft.com/ws/2004/10/ws-discovery/.

[12] WS_Reliability & WS_ReliableMessaging. http://developers.sun.com/sw/platform/technologies/ws-reliability.v1.0.%pdf.

[13] K. Aberer, P. Cudr-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: A Self-organizing Structured P2P System. *SIGMOD*, 32(2):29–32, Sept. 2003.

[14] K. P. Birman, R. van Renesse, and W. Vogels. Adding High Availability and Autonomic Behavior to Web Services. In *26th International Conference on Software Engineering*, pages 17–26, 2004.

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

[16] I.-R. Chen and F. B. Bastani. Reliability of Fully and Partially Replicated Systems. *IEEE Transactions on Reliability*, 41(2):175–182, June 1992.

[17] E. Dekel, O. Frenkel, G. Goft, and Y. Moatti. Easy: Engineering High Availability QoS in wServices. In *22nd International Symposium on Reliable Distributed Systems*, pages 157–166, 2003.

[18] R. Friedman. Caching Web Services in Mobile Ad-Hoc Networks: Opportunities and Challenges. In *Second ACM International Workshop on Principles of Mobile Computing*, pages 90–96, 2002.

[19] R. Hermann, D. Husemann, M. Moser, M. Nidd, C. Rohner, and A. Schade. DEAPspace - transient ad-hoc networking of pervasive devices. In *First Annual Workshop on Mobile and Ad Hoc Networking and Computing*, pages 133–1134, 2000.

0-7695-2567-9/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY