A Middleware for Service-oriented Communication in Mobile Disaster Response Environments [*]

Lukasz Juszczyk
VitaLab, Distributed Systems Group
Institute of Information Systems
Vienna University of Technology, Austria
juszczyk@infosys.tuwien.ac.at

Schahram Dustdar
VitaLab, Distributed Systems Group
Institute of Information Systems
Vienna University of Technology, Austria
dustdar@infosys.tuwien.ac.at

ABSTRACT

Today, the work of disaster response teams is being increasingly supported and coordinated by using portable computing devices. Connected to mobile ad-hoc networks, these devices establish a communication infrastructure immune to damages caused by natural disasters. However, ad-hoc networks are dynamic and volatile environments, which hampers hosting of critical applications relying on fast responsiveness. These difficulties can be mitigated to some extent at the middleware level. In this paper we present RESCUE, an open-source middleware for service-oriented communication in mobile disaster response environments. RESCUE has been designed to address challenges of dynamic ad-hoc networks for service discovery and invocation and provides an infrastructure for flexible mobile systems based on loosely coupled services.

Categories and Subject Descriptors

C.2.4 [**Computer-communication Networks**]: Distributed Systems—*Distributed applications*

Keywords

Service-oriented computing, mobile networks, peer-to-peer, middleware, service discovery, disaster response

1. INTRODUCTION

The portability and steadily increasing performance of mobile computing devices has opened various new areas of application, apart from the traditional usage as mobile calendars and messaging clients. One such area, which has especially gained attention, is the support of emergency teams in the management of natural disasters [14, 19]. During disasters, earthbound infrastructure is, typically, being damaged and, therefore, becomes unavailable. As this problem also affects transmission facilities, such as masts, effective emergency management requires communication technologies which are less vulnerable for they do not rely on preexisting and static infrastructures. Logically, mobile devices supporting ad-hoc communication based on Wi-Fi or mobile WiMAX are predestined for these purposes. Moreover, today's mobile devices have reached a system performance which makes it possible to host more sophisticated software, compared to the restricted usage of the early days. Though, mobile ad-hoc networks are challenging environments in terms of stable communication and reliability in general. The severity of this becomes especially noticeable if these networks must host systems in which availability and responsiveness of participants is crucial, as it is the case for disaster response systems. Although the problems inherent in ad-hoc networks cannot be fully overcome it is yet possible to mitigate their negative impacts by using an optimized communication middleware. In this paper we present RESCUE[1], a light-weight middleware for decoupled service-oriented communication in mobile networks.

RESCUE has been developed inside the WORKPAD project [11, 14] which deals with building an adaptive software infrastructure for supporting collaborative work of human operators in emergency/disaster scenarios. WORKPAD combines a back-end peer-to-peer (P2P) community of inter-organizational hosts, providing advanced services for data and knowledge integration, with a mobile front-end P2P community of human workers operating in the fields of disasters. The front-end community establishes an ad-hoc network in which the response process is executed as an adaptive workflow and where workers provide services which are being dynamically detected and incorporated. RESCUE provides the communication infrastructure for this as it handles deployment and advertisement of local services and discovery and invocation of remote ones. It applies a novel algorithm for active service advertisement, supports real-time notifications about the availability of services, and uses asynchronous messages for decoupled service invocation. RESCUE's functionality has been focused on systems operating in dynamic environments, which, nevertheless, require the highest possible degree of dependability.

The structure of this paper is as follows. In Section 2 we discuss the use of Web services on mobile devices in general and point out our contribution. In Section 3 we review related work. In Section 4 we provide an insight into the concepts and techniques of RESCUE. Section 5 contains

[*]This work is partially supported by the European Union through the FP6-2005-IST-5-034749 project WORKPAD.

[1]Responsive Service-oriented Communication in Unreliable Environments

implementation details and a performance evaluation. Finally, in Section 6 we discuss possible future extensions of our middleware and conclude this paper.

2. WEB SERVICES ON MOBILE DEVICES

Various operating systems have been developed for mobile devices, e.g., Symbian OS, Windows Mobile, iPhone OS, Palm OS, and Linux (e.g., in Android and OpenZaurus). There exist also multiple platforms for the development of software for these operating systems, such as Microsoft .NET Compact Framework, Java Micro Edition, and the Symbian SDK. In addition, each of these listed platforms is based on a different programming language. If the vision of seamless interaction amongst users carrying mobile devices has to become reality, it is necessary to create systems on interoperable and open standards in order to handle the (steadily growing) heterogeneity. This general need for interoperable communications can be split into three main issues: (a) how resources can be found in the environment, (b) how resources can being described, and (c) how they can be used/invoked.

As for the invocation of resources, it is evident that SOAP-based Web services are becoming the preferred transport standard [13, 15, 20]. Web services allow to establish flexible systems consisting of loosely coupled services which can be bound and invoked dynamically. In mobile networks this decoupling is of high importance as service providers may appear and disappear at any time and therefore maximum flexibility, in terms of minimum dependence, is necessary. However, because SOAP is based on XML, which requires costly de-/serialization of data, there were doubts whether it was reasonable to use SOAP on mobile devices at all, but, as shown in [20], light-weight SOAP implementations have solved this performance problem.

For discovery of service instances and for description of service functionality, however, no commonly accepted standards have been established yet. UDDI registries are too complex, have never gained any noteworthy importance, and are already regarded as obsolete. Various discovery protocols, such as the Service Location Protocol [16] and Web Service Dynamic Discovery [12], do exist and do work well, but they are not compatible. Yet, a more severe problem is the lack of an accepted standard for describing the functionality of Web services. Even the Web Service Description Language (WSDL) does only specify the syntax of a service. This raises the question how services can be discovered if there is no de-facto standard, and therefore, how interoperable ad-hoc communication can be realized at all. A frequently applied method to describe services is to use simple keywords, such as {*service.type=printer, printer.type=laser, printer.color=true*} for a printing service, but this still requires some shared knowledge of the service provider and the service clients in order to interpret the semantics of the keywords. Sophisticated semantics based on ontologies, as proposed in [18], are supposed to solve this issue - at least to some extent - but they are still too heavy-weight for mobile computing. As a result, most service registries and discovery protocols follow a pragmatic approach and do not restrict the format of descriptions, leaving the choice, and therefore the problem, up to the users/developers.

2.1 The Contribution of RESCUE

Regarding the challenges and open issues of Web services on mobile devices, it is safe to say that some of them are relevant for the development of mobile disaster response systems, while some are not. For instance, issues such as incompatible representations of service descriptions can be regarded as out of scope of our work as we can safely assume that a rescue team is able to agree on a description format and on a shared keyword terminology. Furthermore, scalability to thousands of nodes, as it is possible in JXTA [3], is not important too, because rescue teams do usually not reach such dimensions. Instead, problems related to the lack of dependability in mobile networks and its effects on the execution of the rescue processes are of high relevance for mobile disaster response systems. This is crucial as human workers do enact critical tasks and the overall performance of rescue processes directly affects the number of saved lifes. Hence, the development of RESCUE has been focused on techniques for tackling dependability problems and for mitigation of their negative effects. And here lies its main contribution which consists of the following features:

- A novel *protocol for service advertisement and discovery*. The protocol works in an incremental manner, reducing network traffic significantly, and propagates changes quickly.
- A *subscription mechanism* for real-time notifications about the availability of services in the environment.
- A continuously updated local database containing a *global view* of all available services and their metadata.
- Pure *peer-to-peer* communication through *asynchronous service invocations*.
- The *open-source license* of the Java prototype [7].

In the design phase of RESCUE, we analyzed the requirements of disaster response systems and applied them to the difficult environment of mobile networks. The result has been that we assessed high priority to the listed features, making a trade-off this way at the cost of other features which were not realizable. Section 4 provides a detailed insight into the concepts and elaborates more on the prioritization inside our middleware.

3. RELATED WORK

In the domain of mobile computing, various concepts and implementations have been developed. These include P2P communication frameworks, discovery protocols, middleware for service provision, etc. However, as requirements of mobile systems differ significantly, there is also no all-round solution which solves all relevant problems adequately but each contribution has its individual strengths and weaknesses.

Middleware for service-oriented communication/collaboration has been developed in projects, such as POPEYE [6] or SPAWN [9]. Similar to RESCUE, these projects are focused on mobile networks and provide a wide spectrum of features, including a flexible handling of underlying networks or a storage of service advertisements and discovery requests in distributed tuple spaces. Yet, they are heavy-weight or do not deal with dependability issues. RESCUE, however, aims at reliable communication with maximum responsiveness.

Sliver [17] is a light-weight BPEL workflow engine for mobile devices and supports the provision of kSOAP2-based [4] Web services. In fact, RESCUE is partially based on Sliver.

We removed the BPEL engine code and extended the SOAP handling for our purposes. However, Sliver does only provide means for service provision and had to be extended with discovery and dynamic binding functionality.

JXTA [3] provides a scalable P2P infrastructure for provision, discovery, description, and invocation of services. The area of application of JXTA ranges from small devices to high performance servers and from static networks to mobile and dynamic ones. Yet, its scalability comes at the cost of a less dynamic discovery. This makes it not applicable for systems which must react to a changing situation quickly.

For service discovery in small-scale networks the following are the most popular protocols which are, though, not suited to our needs: WS-Discovery [12] uses sized SOAP messages and relies on clients polling for services; the Service Location Protocol (SLP) [16] is also based on client-side polling plus it has a too primitive matching mechanism which makes it hard to discover services by their descriptions; DNS Service Discovery (DNS-SD) [1] publishes service details via domain name system records which need to be queried by the clients as well; and the Simple Service Discovery Protocol (SSDP) [8] of Universal Plug and Play (UPnP) [10] advertises services via multicasted HTTP messages. Except for SSDP, these protocols expect clients to poll periodically for services, which hampers a quick notification about changes in the network. Furthermore, by using sized message formats, they cause significant network traffic which limits scalability. In contrast, services in RESCUE are actively advertised by their hosting middleware. This allows to detect new services much faster. Furthermore, our discovery protocol works in an incremental manner which reduces traffic.

This was just a short list of the most relevant works on the field of mobile service-oriented computing. To our best knowledge there is no (published) solution which aims at responsiveness and reliability, and, therefore, none can be used to establish the communication infrastructure of the WORKPAD front-end.

4. CONCEPT AND REALIZATION

Although RESCUE is open to be used in various environments, the designated area of application are mobile networks of rescue teams operating in the field of disasters. As explained previously, the connectivity inside mobile networks is usually unstable. The effects of this range from minor problems (e.g., packet loss) to severe problems, such as lost connections between hosts due to moving away too far from the wireless range, which can even result in a fragmentation of the whole network. This is a fact about wireless networks which must be accepted, though, can be mitigated to some extent. For instance in WORKPAD, the problem of moving away too far from the wireless range is handled by monitoring the movements of the operators and rearranging their positions, if necessary [14]. However, this is being done at a higher level and is out of the scope of our middleware. Instead, the priorities of RESCUE have been concentrated on providing service-oriented communication which is as stable and responsive as the network underneath allows.

The concept of RESCUE is based on every node having a global view on all services in the network. Although this approach is not suited to large-scale environments due to poor scalability, it is well applicable for small- to medium-scale disaster response systems. In RESCUE, nodes advertise their deployed services and listen to the advertisements of others. Information about discovered services, including local and remote ones, is kept in a local database which contains published metadata (descriptions) about all services. The metadata consists of mandatory records which are necessary to identify and invoke the services, but consists also of user-defined properties which describe the functionality. Client applications can search for services by either querying the database or by placing subscriptions to be notified when matching services become available or unavailable. Due to the active advertisement protocol, it is possible for clients to be notified almost immediately after the appearing of a service in the network. Eventually, communication with remote services can be done either by one-way messages or by synchronous as well as asynchronous invocations. RESCUE has 3 mandatory services: (a) a deployment service which is only available locally, (b) an information service delivering metadata about deployed services, and (c) a callback service receiving responses to asynchronous invocations.

The combination of these features creates a flexible infrastructure for service-oriented communication, able to react quickly to changes in the environment. This is mainly realized by combining an optimized discovery protocol with an asynchronous service invocation mechanism.

4.1 Service Advertisement & Discovery

The service discovery protocol of RESCUE has been subject to several trade-offs. On the one hand, wireless communications must be as light-weight as possible because a higher amount of traffic increases the probability that some data packets get lost and the communication is delayed because of timeouts and resending of packets. Of course, a reduced bandwidth consumption does also have positive effects on the CPU load and the battery consumption. On the other hand, in order to make fast responsiveness possible, it is necessary to exchange service advertisements frequently to detect new services and also disappeared ones based on missing refresh advertisements from them. As none of the existing protocols was able to handle these contradicting requirements adequately, we have developed a new advertisement and discovery protocol for RESCUE.

4.1.1 *Techniques for Service Advertisement*

In RESCUE, we use active advertisements sent out by service providers instead of a client-side discovery through continuous polling. This is the most effective way to keep (potential) clients up-to-date about the availability of services, as only providers know best when to propagate update notifications. Furthermore, we regard it as inefficient to send out individual advertisements for every single service, especially when they also contain all metadata, as it is the case for instance in SLP. In contrast, we rely on incremental updates where advertisements are sent out per middleware instance, not per service instance. Figure 1 lists simplified versions of the algorithms for advertisement (done by provider) and discovery (done by clients on receipt of advertisement message) of services. An important feature of RESCUE is its flexible adjusting of advertisement intervals. Advertisements are usually sent out in predefined intervals, however, can be sent with a higher frequency if (a) a new service is being un-/deployed at the local middleware instance or (b) if the last received advertisement of a remote node was too long ago. Obviously, the sense of (a) is to propagate changes of the local state quickly, while the pur-

```
Un-/DeployService($service$)
  $registerAtMiddlewareModules(service)$
  $stateCounterID \leftarrow stateCounterID + 1$
  $sendAdv \leftarrow true$

Advertisement()
  while $Middleware.isRunning()$
  do $waitForNextInterval()$
      if $curTimestamp() - lastAdvRcvd > threshold$
        then // last received adv is too long ago
              // reduce threshold to accelerate interval
              $threshold \leftarrow reduce(threshold)$
              $sendAdv \leftarrow true$
      if $curTimestamp() - lastAdvSent > threshold$
        then // last sent adv is too long ago
              $sendAdv \leftarrow true$
      if $sendAdv = true$
        then $status \leftarrow buildAdvMessage(stateCounterID)$
              $sendViaMulticast(status)$
              $threshold \leftarrow defaultThreshold$
              $sendAdv \leftarrow false$ // mark as sent
              $lastAdvSent \leftarrow curTimestamp()$

Discovery($sender, status$)
  $lastAdvRcvd \leftarrow curTimestamp()$
  // full update necessary?
  if $status.hasNewRuntimeID(sender.lastStatus)$
    then $retrieveAllServices(sender)$
  // or just incremental?
  if $status.hasNewSerialCounterID(sender.lastStatus)$
    then $retrieveIncrementally(sender, status, lastStatus)$
  $sender.lastStatus \leftarrow status$
  // update db and notify clients about matching services
  $updateLocalDB()$
  $checkSubscriptions()$
```

Figure 1: Advertisment & Discovery Algorithms

pose of (b) is to detect situations where the mobile device is disconnected from the rest of the network and to increase the frequency in order to advertise its reappearance faster once reconnected.

At the transport level, we use multicast over OLSR [5] for sending out the advertisements as UDP packets. Clients which receive them initiate a unicast communication with the provider via UDP or TCP, depending on the message sizes. Listing 1 shows a sample conversation. Line 1 contains the advertisement message propagated via multicast. The UUID is used as a static identifier of middleware instances, because IP addresses of nodes can change and, therefore, are not sufficient for precise identification. The URI is followed by a runtime ID which is generated anew at each start of the middleware and is used to tell whether the instance has been restarted since the last advertisement and, therefore, also might host different services now. The last token of the message is the serial state ID, which is incremented every time when a service is deployed or undeployed at the middleware. As a result every client which receives such an advertisement knows whether it should (a) retrieve the whole service state of the middleware instance, if it has never received an advertisement from it or if the runtime ID has changed, (b) whether it should retrieve an incremental update on the state, if only the serial state ID has changed, or (c) whether it should regard the state as unchanged, if the same advertisement has been received before. In case of an unchanged state, the references to services hosted by the sending middleware instance are refreshed. In case of a

```
1  > advertise 275170e0-3ebe-11dd-baa0-0015c5568d5b
              http://10.20.30.40:8080/services 82325 38

2  < ifUUID 275170e0-3ebe-...0015c5568d5b getStatus 35 38

3  > OK + http://www.vitalab.tuwien.ac.at/testSer/ ..

4  < ifUUID 275170e0-3ebe-11dd-baa0-0015c5568d5b
       getAttrib http://www.vitalab.tuwien.ac.at/testSer

5  > OK id=testSer namespace=http://www... abc#YSBi ...
```

Listing 1: Sample Advertisement Communication

changed state ID, the client can retrieve the update in an incremental way by specifying the last seen state ID and the current one from the advertisement. As shown in Line 2, the client also prepends a `ifUUID` command which is necessary for recognizing and discarding requests to wrong nodes. This can occur if a node appears and gets the IP of a previously disconnected node assigned, which was the designated destination of the request. On receipt of a `getStatus` request, the providing middleware sends back a response (Line 3) containing the identifiers of services which have been deployed meanwhile (prepended by a plus) and also of all undeployed ones (prepended by a minus).

This is the minimum amount of interaction done by the middleware in order to keep track of the changes in the network and to update the local database continuously. However, by just performing these steps the database will contain only basic information, such as namespace and URI of the service and the UUID of its middleware instance. Although this is sufficient to locate and identify services in the network, it does not contain any real description and therefore makes service discovery difficult. Nevertheless, RESCUE does not retrieve all metadata from all services automatically. Instead, in order to save bandwidth, we implemented a more flexible approach where service metadata consists of 3 levels of information and is being retrieved on demand:

1. The 1st level contains information which is being retrieved automatically, such as the location and the namespace of the service and the UUID of its node.

2. The 2nd level contains attributes which describe the service in general. As explained in Section 2.1, we do not enforce any description format but expect the attributes to be specified as name-value tuples. Lines 4 and 5 in Listing 1 depict the exchanged messages for retrieval of 2nd level attributes.

3. On the 3rd level, the individual operations are being described by name-value attributes and their signatures (message types). This metadata is being retrieved from the mandatory information service of the remote middleware instance.

The point of splitting the metadata into 3 levels is that a client may identify a desired service solely based on some known identifiers (1st level) or by general descriptions (2nd level, e.g., as in the printer example in Section 2). This mainly happens in closed environments, such as the rescue teams of WORKPAD, where participants share a keyword terminology for descriptions and do not need to analyze all metadata to know how to invoke services and, therefore, the middleware does not need to transfer all information. Yet, this is not the case for ad-hoc collaborations where independent participants provide services of which the details are not known in before. In these, it is necessary to retrieve and analyze all metadata in order to invoke the services correctly.

4.1.2 *Query and Subscription*

In RESCUE, service discovery can be done in 2 ways. Either by querying, which delivers immediately all currently available services matching the query predicates, or by placing subscriptions and waiting for notifications if matching services appear or disappear. Although, querying seems to be the preferred method in many protocols, such as SLP and WS-Discovery, this is probably just due to its simplicity. We believe that this approach has too many disadvantages. For instance, it is not possible to detect a changing availability of a service quickly after its happening, but only at the moment of the query itself. Surely, this can be reduced by applying a higher frequency to the queries, but this makes network traffic grow significantly. As a solution to this problem, RESCUE provides support for subscription and notification. For this, clients define the predicates for matching the services in a specific Java object. This object is passed to the middleware which checks all currently available services immediately and, of course, also checks all services which will be detected in the future whether they match the predicates. In case of a positive match, the middleware notifies the client about the service and its availability. This approach has 2 main advantages. First of all, by combining the subscription mechanism with active advertisements, which are sent out immediately after something changes, it is possible to notify clients quickly. The second advantage lies in the efficient handling of metadata retrieval. For predicates, which are well designed and which reference metadata from more than one level, the middleware checks the predicates of the lower levels first and skips the rest of the procedure in case of a negative match. The effect of this is that services can be ”disqualified” early, without the necessity of retrieving and checking all levels of their metadata.

4.2 Service Invocation

In service-oriented systems, there are two communication paradigms for service invocation. There is the pure message-centric model, in which components communicate by passing asynchronous messages through a middleware, and there is the paradigm of Remote Procedure Calls (RPC), which makes services act as distributed objects. As both models have their advantages, and furthermore the choice of which is the proper one depends on the area of application, RESCUE supports both of them, yet, with a special focus on handling of unstable connectivity in mobile networks. Although an unstable connectivity affects the whole spectrum of communication inside the network, it is especially harmful for RPC communication, which is usually performed in a synchronous manner. In this, the client opens a connection to the remote service endpoint, sends the request and keeps the connection alive until it receives the response. This is acceptable for static environments which are (usually) stable, but definitely not for mobile ones, in particular if the data packets are forwarded between multiple hops. In case of an interrupted connection, the whole request has to be sent and processed anew. In RESCUE, we mitigate this problem - as a 100% solution is not possible - by supporting purely asynchronous service invocations keeping the duration of the connections as short as possible. Clients, which initiate an asynchronous invocation, are registered at the local callback service of the middleware and the request is annotated with additional SOAP headers and sent as a one way message, closing the connection afterwards. At the service side, the request is being processed and once the response message is ready to be sent back, the middleware queries the local database to check whether the client is still available and then tries to send back the response. However, if the client is not available, a subscription is placed which matches the callback service of the client and in case of a positive notification, which means that the callback service reappeared again, the response is transferred and the callback service passes the response to the client. This technique benefits the reliability of service-oriented communication by (a) keeping the duration of connections, which are vulnerable to the dynamics of mobile networks, short and (b) by using the subscription mechanism to transfer the responses immediately when the client’s callback service becomes available again. The effect is a higher probability (a) that the invocation gets processed without being interrupted and (b) that the response will reach the client quickly.

5. PROTOTYPE IMPLEMENTATION

The prototype of RESCUE has been implemented in Java2 ME (CDC 1.1 profile). We have reused (and partially extended) several external libraries, such as the Jetty HTTP server [2] and Sliver [17] / kSOAP2 [4] for handling SOAP communication. As Sliver supports HTTP as well as a more light-weight socket-based TCP protocol for message transport, we kept kept both in RESCUE. However, even though the socket-based protocol consumes less bandwidth and allows a 1.5 times higher throughput of messages, we recommend HTTP due to its wide acceptance as a de-facto standard for SOAP transport. For interoperability we have also developed a tool which acts as a bridge to other environments, by monitoring the availability of services and generating automatically WSDL definitions.

We believe that RESCUE can be useful not only for disaster responses, but also for other domains which deal with mobile service-oriented communication. As a consequence, we decided to make it open-source under the LGPL license and to provide the software at our prototype Web site [7].

5.1 System Performance

For evaluating the performance of RESCUE, we deployed a small testbed consisting of a Linux-based laptop with a Sun Java 1.4 virtual machine (VM) and 3 HP iPAQ PDAs with Windows CE 5 and IBM J9 Java ME (CDC 1.1 profile) VM. The PDAs were equipped with an Intel PXA 270 416MHz CPU and 64 MB RAM, and were used for measuring the resource consumption of RESCUE, while the laptop was mainly used for putting load on them, as a kind of a stress test. Our evaluation aimed at determining the load caused only by the middleware - without any applications running on top of the middleware and causing additional load - and included measurements of resource consumption and the time needed to discover new services.

As usual in Java, the implementation of the VM has a strong effect on the memory consumption. With IBM J9 we noticed that the plain VM consumes 5 MB, while RESCUE (including the Jetty HTTP server) causes approximately 2 MB more of memory usage. For the CPU load, we saw that Windows CE itself already consumes 15% of the cycles, even if no user applications are running. RESCUE causes a short significant load on the CPU during the boot process, yet, consumes only a marginal load (<3%) at runtime, even during high network activity with frequently incoming advertisements. This is a consequence of the light-weight design of the core components. As for the network traffic, we determined how much bandwidth is being consumed for keeping the service database up-to-date and omitted client subscriptions which reference 2^{nd} or 3^{rd} level metadata, as the traffic caused by them fully depends on the size of the metadata and the predicates of the subscriptions. As shown in Listing 1, advertisement messages have an average size of 90-100 byte and are sent out in adaptive intervals (by default between 1 and 5 seconds) via multicast too all other nodes. In case of a changed status, these nodes request via unicast an incremental update, which consumes approximately 120-200 byte. In general, the caused traffic by each node can be calculated with this simplified formula:

$$traffic = \frac{size\_of\_adv}{adv\_interval} + \frac{size\_of\_update}{update\_interval} * num\_of\_nodes$$

However, another factor influencing the amount of traffic, which has not been included in this formula, is the topology of the network, thus, whether peers communicate directly or via routing nodes which forward traffic. In a test, with 3 directly communicating nodes which changed their status every 15 seconds (which is quite frequent) we determined an average traffic of 58 byte per second per node. Then, we investigated on the time necessary for detecting new or disappeared services in the network. As the interval adapts to the deployment activity of the middleware and the minimal interval is 1 second, the expectation value for detecting a change is 0.5 second plus the delay for transferring and processing the messages. In our tests the average delay was less than 0.1 second. However, if packet loss comes into play, the advertisement gets lost and the change is detected in the next round, unless this advertisement does not get lost again. This is quite problematic and causes the average detection time to grow depending on the loss rate according to the following formula:

$$time = \sum_{n=0}^{\infty} (\frac{min\_interval}{2} + delay) * (1 - lossrate) * lossrate^n$$

In our experiments in an unstable network with 20% of loss rate, we experienced that the average time was 1.8 seconds.

6. CONCLUSION AND FUTURE WORK

In this paper we have presented, RESCUE, our ongoing work on a middleware which aims at providing an peer-to-peer infrastructure for responsive service-oriented communication in mobile networks. As the detection of changes in the network is a key feature for responsiveness, we have developed a novel and light-weight discovery protocol which notifies peers quickly about the availability of services. By placing subscriptions, clients can tap the full potential of real-time notifications in order to be up-to-date about the availability of services in the network. Furthermore, we have optimized the invocation of services with regard to handling the stability issues of mobile networks.

Yet, there is still place for improvements, which will be subject to future research. This includes new functionality, such as replication of asynchronous invocation messages to handle dependability problems, as well as optimizations of the currently available features. For example, we identified possible improvements for the discovery protocol which would reduce the necessity of unicast communication by propagating certain updates with the advertisement messages. At last, and this is the most challenging problem, there is more research necessary to develop techniques to handle unnoticed packet losses which hamper the propagation of advertisements.

7. REFERENCES

[1] DNS Service Discovery. http://www.dns-sd.org.

[2] Jetty Server. http://www.mortbay.org.

[3] JXTA. http://jxta.dev.java.net.

[4] kSOAP2. http://ksoap2.sourceforge.net.

[5] OLSR Multicast Forwarding Plugin. http://sourceforge.net/projects/olsr-bmf.

[6] POPEYE Project. http://www.ist-popeye.eu.

[7] RESCUE Prototype. http://vitalab.tuwien.ac.at/prototypes/rescue.

[8] Simple Service Discovery Protocol - IETF draft revision 3. http://quimby.gnus.org/internet-drafts/draft-cai-ssdp-v1-03.txt.

[9] SPAWN Project. http://www.cs.wustl.edu/mobilab/Projects/SPAWN.

[10] Universal Plug and Play. http://www.upnp.org.

[11] WORKPAD Project. http://www.workpad-project.eu.

[12] WS-Discovery Specification. http://specs.xmlsoap.org/ws/2005/04/discovery/ws-discovery.pdf.

[13] S. Berger, S. McFaddin, C. Narayanaswami, and M. T. Raghunath. Web services on mobile devices - implementation and experience. In *IEEE WMCSA*, pages 100–109, 2003.

[14] T. Catarci, M. de Leoni, A. Marrella, M. Mecella, G. Vetere, B. Salvatore, Dustdar, L. Juszczyk, A. Manzoor, and H.-L. Truong. Pervasive software environments for supporting disaster responses. *IEEE Internet Computing*, 12(1):26–37, 2008.

[15] G. Gehlen and L. Pham. Mobile web services for peer-to-peer applications. *IEEE CCNC*, pages 427–433, Jan. 2005.

[16] E. Guttmann, C. Perkins, J. Veizades, and M. Day. Service location protocol, 1999. Version 2. IETF Internet Draft, RFC 2608.

[17] G. Hackmann, M. Haitjema, C. D. Gill, and G.-C. Roman. Sliver: A bpel workflow process execution engine for mobile devices. In *ICSOC*, volume 4294 of *LNCS*, pages 503–508. Springer, 2006.

[18] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[19] M. Portmann and A. A. Pirzada. Wireless mesh networks for public safety and crisis management applications. *IEEE Internet Computing*, 12(1):18–25, 2008.

[20] D. Schall, M. Aiello, and S. Dustdar. Web services on embedded devices. *IJWIS*, 2(1):45–50, 2006.