Vimoware - a Toolkit for Mobile Web Services and Collaborative Computing [*]

Hong-Linh Truong, Lukasz Juszczyk, Shariq Bashir, Atif Manzoor, Schahram Dustdar

Distributed Systems Group, Vienna University of Technology, Austria
{truong, juszczyk, bashir, manzoor, dustdar}@infosys.tuwien.ac.at

Abstract

Mobile devices are considered to be very useful in ad-hoc and team collaborations, for example in disaster responses, where dedicated infrastructures are not available. Such collaborations normally require flexible and interoperable services while running on mobile devices and being integrated with various other services. Therefore, middleware and toolkits for developing mobile services which can be accessed by using standard interfaces and protocols are in demand. Due to the lack of tools, the support of the development of Web services and collaboration tools on mobile devices is still limited. This paper presents the Vimoware toolkit which allows both developers and users to develop Web services for mobile devices, to conduct ad-hoc team collaborations by executing pre-defined or on-situ flows of tasks, and to test collaboration scenarios.

1 Introduction

Recently, many research efforts have focused on providing programming tools and software engineering methodologies for developing applications for mobile devices (PDA, smartphone, subnotebook and laptop). First, in most cases, existing tools and methodologies aim at supporting the development of mobile applications acting as a client to access data and services from dedicated, high-end systems. This is partially due to constrained resources of mobile devices on which the applications execute, and the usage mode in which mobile devices access services rather than provide them. Second, there are tools for developing applications for mobile devices in ad-hoc networks but the applications tend to be specific and inflexible as they bind to specific protocols and models. It is not easy to reuse and integrate current mobile applications for/into existing, diverse Web services available in today's pervasive environments.

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

Mobile devices have been increasingly used for critical missions, such as in disaster responses [4], which require access to diverse services. In particular, mobile devices are considered to be very useful in ad-hoc team collaborations where dedicated infrastructures are not available. Such collaborations normally require flexible and interoperable applications to access as well as offer services. This raises the question of how to provide pervasive and mobile devices with middleware and applications so that the devices can provide collaboration services accessible through standard interfaces and protocols. The Web services model which has introduced means to foster the interoperability, flexibility, and reusabilty of software can be used to develop mobile Web services for collaborative computing. Until now, there is a lack of generic Web services based toolkits for developing and testing mobile Web services and collaboration tools.

Our research focuses on a Web services based toolkit that allows the developer and the user to develop and test Web services on mobile devices, to conduct ad-hoc team collaborations by executing pre-defined or on-situ flows of tasks, and to test collaboration scenarios. The goal of the toolkit is to provide means for developing and executing customizable and interoperable mobile Web services and applications. Yet the development process should be simple, for example, based on the concept of POJO (Plain Old Java Object). In this paper, we introduce Vimoware which includes tools for developing and deploying Web services on mobile devices and for specifying and executing user-defined flows of tasks. Vimoware allows for setting up different collaboration scenarios described by different models based on various plugins. In this paper, we present various examples to demonstrate the usefulness of Vimoware.

The rest of this paper is organized as follows. Section 2 presents main related work. The Vimoware toolkit is detailed in Section 3. We describe potential applications and examples built atop Vimoware in Section 4. Performance experiments are presented in Section 5. Section 6 summarizes the paper and outlines possible future work on Vimoware.

2 Related Work

The use of mobile devices for collaborative work is widely known and acknowledged. Therefore, many mobile middleware have been developed, such as intermediate platforms [11], coordination middleware [1], and collaborative working environments [6].

Web services have been widely employed in normal environments due to its ability to address the integration and interoperability. However, Web services have not been well exploited in mobile devices due to limited capabilities of mobile devices. Some potential applications and challenges of Web services hosted in mobile devices are discussed in [3]. In the last years mobile devices have become more powerful, and lightweight implementations, such as kSOAP2[1] for Java or the Microsoft .NET Compact Framework made SOAP communication feasible for mobile devices. Silver [7], a lightweight BPEL engine in mobile devices, also provides some facilities for handling SOAP communication. Part of our Vimoware middleware utilizes kSOAP2 and Silver SOAP communication facilities.

Existing work normally exploits mobile devices as a client to access remote Web services, but not as a service provider. Recently, researchers have investigated Web services solutions for sensor networks [2], embedded systems [15], and ambient environments [8], signalling a strong need for having tools to develop Web services on mobile devices. Another trend to support SOA-based on mobile devices is to use the OGSi model which supports the development of components interacting with each other within a single system; OGSi components can also be exposed as Web services. R-OGSi [13] is a middleware built atop OGSi that transparently connects, deploys and executes OGSi services spanning multiple OGSi containers. We follow the Web services programming model which is different from, but complementary to, the OGSi programming model.

In [9], a lightweight hosting environment for Web services on mobile devices is presented. This environment supports broker-based service publishing and service migration. Vimoware provides more features than just hosting Web services, but it has not addressed the service migration. [12, 7] support BPEL and workflows in mobile devices, however, they are not flexible enough for team collaboration in dynamic scenarios. A SOA-based toolkit for collaborative work based on a network of mobile devices is currently missing. The MoCA framework [14] provides custom APIs and basic services for developing collaborative applications. However, server applications are only for a network of static machines, and MoCA is not SOA-based.

Our Vimoware also differs from the above-mentioned works with respect to supporting collaborative teamwork by providing a flexible toolkit for developers to develop Web services, and to define and execute collaborative processes with customized scenarios. Not only does Vimoware support developers to write Web services on mobile devices in simple way but also provides many specific features for the development and deployment of collaborative work.

[1]http://ksoap2.sourceforge.net/

3 The Vimoware Toolkit

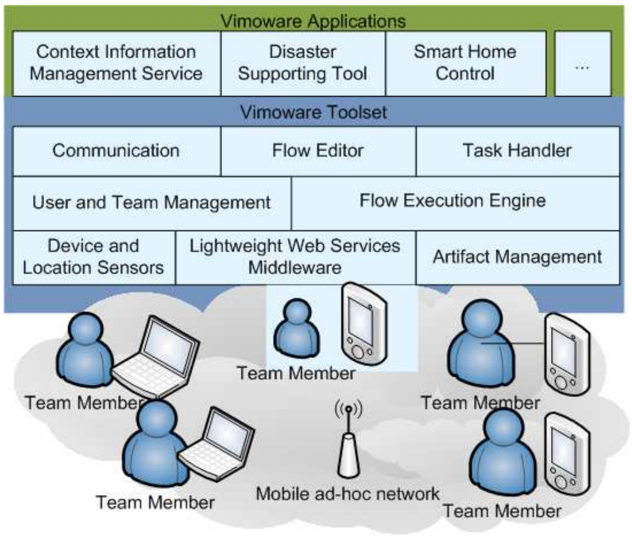


Figure 1. Components of Vimoware

Figure 1 depicts the main components of Vimoware. SOAP-based Web services are hosted and advertised using the *Lightweight Web services Middleware*. The *Artifact Management* is used for sharing documents, figures, multimedia files, etc., through HTTP. *Device and Location Sensors* support the gathering of data about the current status of the device. The *User and Team Management* supports the provision of user and team profiles and the *Flow Execution Engine* facilitates the realization of ad-hoc and pre-defined workflows of tasks. To enable the exchange of messages between end users, the *Communication* is developed to provide instant messaging features. The *Task Handler* is a service that receives tasks sent by the *Flow Execution Engine* and controls the execution of the tasks in a given device. The *Flow Editor* supports the user to design their workflows which are executed by the *Flow Execution Engine*.

3.1 Lightweight Web Services Middleware

In Vimoware, Web services are developed by applying the POJO principle which is widely used due to its simplicity and therefore is preferred by us. The service developer creates a Web service by extending an abstract Java class (see Listing 1). This class requires the specification of a service description, provides basic methods for extracting metadata about the service, and expects the service operations to be implemented as normal Java methods. Based on the description provided by the service developer and on the metadata, we support the generation of WSDL files which can be in turn used by remote clients to generate clients stubs for invoking mobile Web services.

```
public abstract class AWebService {
  public AWebService(String ID, String namespace);
  public final String getServiceID();
  public final String getServiceNamespace();
  public abstract Hashtable getAttributes();
  public void onDeploy();
  public void onUndeploy();
  public abstract Collection getOperationNames();
  public Hashtable getOperationNameMapping();
  public abstract Hashtable getOperationDescription
                              (String operationName);
  public abstract String getConfigurationRessourceName();
  public void setPublished(boolean published);
}
```

Listing 1. The abstract Java class for services

As Vimoware is designed for Web services in ad-hoc network of mobile devices, runtime and reconfigurable service provisioning and service discovery are of paramount importance.

3.1.1 Service Provisioning

With respect to service provisioning, it is important that any middleware for mobile devices is able to support reconfigurable Web services as mobile environments are highly dynamic and different in multitude of capability degrees. Vimoware supports the reconfiguration of communication protocols, multiple types of service invocations, and runtime deployment.

First, Vimoware uses kSOAP2 and selected parts of Sliver [7] to deploy Web services and provide them to remote clients. At the transport level, communication can be established either via HTTP, which is realized by a lightweight version of the Jetty[2] engine, or via direct TCP socket communication. Depending on specific need and performance issue, the TCP or HTTP transport can be selected.

Second, in Vimoware invocations can happen in three manners: (a) one-way message sending, (b) synchronous request-response interactions and (c) real asynchronous request-response. The asynchronous invocation is realized by a callback service which is a mandatory part of each middleware instance. When a client sends an asynchronous request, the middleware automatically appends a message ID to the SOAP header and registers it at the callback service. When the processing of the request finished and the response is ready to be sent back, the middleware checks whether the requesting host is available and the response can be sent back, or whether the requesting host disappeared meanwhile and the response has to be postponed until the requesting host reappears. By decoupling the requesting and responding host this way, it is possible to overcome typical problems related to the dynamics of ad-hoc networks.

Third, services can be deployed into Vimoware at runtime and from remote host. This feature is required in scenarios when local deployment is impossible, for example, in a disaster scenario where volunteers can offer their support by joining teams with their mobile devices. Also, this allows new services to be deployed in mobile/embedded devices without interrupting operations of the devices.

[2]http://jetty.mortbay.org/

3.1.2 Service Advertisement and Discovery

The well-known publish-find-bind triangle of SOA with a dedicated registry is not suitable for ad-hoc networks of mobile devices. In Vimoware we chose to implement a P2P-based subscription/notification approach for service advertisement and a discovery which allows to be notified immediately, reduces network traffic, and is suitable for ad-hoc networks. Currently, discovery works purely decentralized in which all Vimoware instances advertise themselves via UDP multicast. We have developed a customized service discovery protocol optimized for highly dynamic environments.

The customized protocol tackles the changing availability of devices and services, and works as follows[3]. In order to reduce network traffic, advertisements are not sent out for every service but only for each Vimoware middleware instance. These advertisements contain information whether the deployment status of the hosted services has changed since the last update. In this case other Vimoware instances in the environment request an incremental change report in order to update to the new status. If a host disappears without having had the possibility of advertising its unavailability (e.g., on middleware shutdown) the information about its services will automatically expire after a predefined period of time. As a consequence, all participants in the environment are aware of current available services and are being informed as quickly as possible with any status changes. This approach comes at the cost of scalability, and thus, is not suitable for large environments consisting of hundreds of participants. Nevertheless, it is suitable for small or medium scale mobile environments and offers a benefit of real-time notifications. Furthermore, it is possible to establish interoperability with other discovery protocols, such as the Service Location Protocol (SLP), through a bridging mechanism which translates Vimoware-specific advertisements into the SLP format.

[3]The full account of the customized protocol and its evaluation is out of the scope of this paper.

3.2 Location and Device Sensors

Providing context information of devices and location is a need for enhancing collaboration work and decision making and dynamically adapting service behavior according to changes. In our framework, device sensors (CPU, network, memory, and device profile) and location sensor are developed as Web services which describe their provided information as Web service attributes in the middleware. Thus, client application can browse the information and subscribe or query context information using interfaces provided by these sensor services. Client applications can send request specified in XPath/XQuery to sensor services to obtain context information. Client applications can also permanently subscribe for any information. Currently, sensors providing machines information are built atop the Intel Mobile Platform SDK[4]. Implementing the sensors as Web services enhances the interoperability and integration of different context information sources, fostering the development of context-aware mobile Web services.

3.3 User and Team Management

This component manages the profile of the person using a device and a list of teams that the person belongs to. The profile includes basic information, such as name, ID and skills, used to associate human with services deployed in the device. Therefore, other services can search for a specific service provided by a specific individual. In Vimoware, the profile of a human is described by a pre-defined XML representation. The XML-based profile will be provided by the end-user. However, it is not difficult to synchronize this information with existing team and user management services in particular organizations or to utilize information in real PIM (Personal Information Management) implemented based on JSR-75[5] in mobile devices.

Services can publish the profile information through the middleware by expressing the information into service attributes. For example, a quadruple $(name_i, team_j, st_k, URI_l)$ published through the middleware indicates that the user $name_i$ of team $team_j$ provides a service identified by URI_k belongs to the service type st_k. At each device, this component maintains a list of $(name_i, team_i, st_i, URI_i)$. The profile information is used when a service needs to find another service provided by another user. Although the profile information is simple, it is particularly useful when having ad-hoc collaborations, such as volunteering work in a disaster response. In such collaborations, the personal skill is the key information for selecting services and humans for particular tasks. This component is available only in an individual device. Providing access to profile information across a network can be achieved by building overlay services atop Vimoware. One example of such services is the CIMS (Context Information Management Service, in Figure 1) which provides Web service operations for obtaining profile information of teams and users.

3.4 Flow Execution Engine

The *Flow Execution Engine* is used to execute flows of tasks of collaborations. Vimoware's *Flow Execution Engine* is lightweight and built based on a (simplified) backport of the BeanFlow library[6] using POJO concept. The engine can execute flows modeled by an XML schema or defined on demand.

To support lightweight task distribution and control, the *Flow Execution Engine* implements a simple, yet powerful, flow language proposed in [10] by the WORKPAD project[7]. This flow language consists of basic, well-known flow constructs such as *parallel, sequence, loop* and *task*. In this flow language, a *task* is used to describe an atomic activity of a flow. However, the task model in [10] is not flexible enough, as it fixes the way how a task is described and does not include interaction models which are used to model different scenarios. For example, in a disaster response scenario[4], when a team member receives any new task from the team leader, the team member can notify the team leader whether he/she accepts or rejects the task, e.g., in volunteering and flexible responses, or the team member must perform the task - as usually in a professional team. To provide a customizable mechanism for implementing different types of scenarios, we enrich the task model in [10] with metadata to specify (i) whether a task receiver is allowed to reject the task or not, (ii) a task can be assigned to multiple members, and (iii) a task can be executed in synchronous or asynchronous models (see Listing 2). This allows us to configure the flow execution suitable for different scenarios and teamwork.

```
<task name="takePhoto" priority="3" location="A">
 <description>Take photos of building A</description>
 <reqCapability name="camera"></reqCapability>
 <execution mode="synchronous" rejectable="false"/>
</task>
```

Listing 2. Metadata associated with a task

Writing a flow based on the above-mentioned language might be enough for the end user. However, Vimoware also supports the developer to write flows for different purposes. Flows of activities are developed based on the POJO concept by composing activities. We provide a `VActivity`

[4]http://ossmpsdk.intel.com/

[5]http://jcp.org/en/jsr/detail?id=75

[6]The BeanFlow (http://servicemix.apache.org/beanflow.html) is implemented in Java 1.5. We did a backport of the BeanFlow for Java 1.4.

[7]http://www.workpad-project.eu

class which extends `TimeoutActivity` class provided by the BeanFlow. An activity described by `VActivity` will accept a `Task` which captures information related to a task described above. Given a `Task`, `VActivity` automatically assigns the `Task` to a resource or a particular *Task Handler* based on plugins of *task assignment models* or the actor invoking the engine can manually map the `Task` onto a resource. As Vimoware aims at supporting different scenarios where different task assignment models can be applied, we provide a plug-in mechanism to support the developer and the user to write and include various different models for assigning tasks. Listing 3 outlines the generic interface for task mapping plugins supplied by the developer.

```
public class TaskMapping {
    public TaskMapping();
    public Task getTask();
    public void setTask(Task task):
    public String[] findSuitableResource();
    public String assignTask(String resource);
}
```

Listing 3. Interface for mapping task to resource (human or service)

Our objective is to provide customizable mechanism for plugging in different mappings of tasks to resources (person or service), and we test our mechanism with some well-known task assignments, such as FIFO, as a proof-of-concept. Potentially, the developers can write different mappings, such as shown in [5], by utilizing various sources of information given by Vimoware. To write a flow, the developer just uses the POJO concept introduced in the BeanFlow. A flow is constructed like a Java program by using basic constructs, such as `if`, `for` and sequence of statements. Listing 4 shows an example of a parallel construct within a flow.

```
List tActivities = new ArrayList();
Enumeration items = null ;
....
//list of activities in a parallel construct
while (items.hasMoreElements()) {
    final Object task = items.nextElement();
    tActivities.add(new VActivity((Task)task));
}
//start a joint flow
JoinAll flow = new JoinAll(tActivities);
flow.start();
flow.join();
```

Listing 4. Example of a parallel construct

3.5 Task Handler

The *Task Handler* provides a set of operations, as methods of a Web service, for receiving and handling tasks from and interacting with the *Flow Execution Engine*. As mentioned before, Vimoware supports both synchronous and asynchronous task execution models. Figure 2 describes the interaction model between the *Flow Execution Engine* and the *Task Handler*. The engine can send tasks to a *Task Handler*. On receiving a new task, the *Task Handler* passes the task to a corresponding pre-configured *Task Executor*. Our goal is to support different ways to execute a task, such as tasks can be executed automatically by software components to implement different models or manually by human. Therefore, we provide a generic interface for implementing *Task Executors*. The *Task Handler* does not return the result of the task to the engine. Instead, it returns the URI of the result; the URI is obtained from the *Artifact Management* which is a RESTful service providing two main operations for putting and getting the result. Given the URI, the engine or any application can obtain the result. This design is made based on the fact that tasks might return multimedia data as well as it allows us to share artifacts in mobile ad-hoc network, e.g., by integrating the *Artifact Management* with existing P2P file sharing systems.

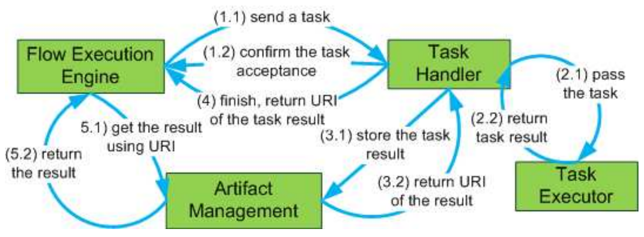

(1.1) send a task
Flow Execution Engine
(1.2) confirm the task acceptance
Task Handler
(2.1) pass the task
(4) finish, return URI of the task result
(2.2) return task result
5.1) get the result using URI
(3.1) store the task result
(5.2) return the result
Artifact Management
(3.2) return URI of the result
Task Executor

Figure 2. Interaction model between the Flow Execution Engine and the Task Handler

4 Vimoware Applications

4.1 Mobile and Collaborative Services

To illustrate how the development of mobile and collaboration Web services is simplified with Vimoware, we present an example of the *Communication* component which is developed based on the *Lightweight Web services Middleware* and utilizes the *User and Team Management* module. It is realized as a Web service that makes use of the subscription facility of the Web services middleware to place messages which are being either sent immediately, if the recipient is available, or are being postponed until he/she reappears again. In the next code snippets we demonstrate how such a behavior can be implemented in a simple way.

Listing 5 shows a simplified communication service which offers a single operation for printing out messages to the console. This service is described by its name and namespace, and publishes additionally information about it's owner and the teams he/she belongs to (see `getAttributes()`). The published operations and their descriptions are retrieved via `getOperationNames()`

and `getOperationDescription()`. These methods are mandatory and provide information which is being advertised by the middleware.

```
public class IMService extends AWebService {
  public IMService() {
    super(IMSERVICENAME, IMSERVICENAMESPACE);
  }
  public Hashtable getAttributes() {
    Hashtable attributes=new Hashtable();
    attributes.put(TEAM, TeamManagement.getTeams());
    attributes.put(MEMBER, TeamManagement().getUser());
    // ... additional attributes
    return attributes;
  }
  public Hashtable getOperationDescription(
                              String operationName) {
    Hashtable description=new Hashtable();
    if ("message".equals(operationName)) {
        // userdefined descriptions
        description.put(TEXT, "...");
        description.put(KEYWORDS, "...");
    }
    // ... describe operations
    return description;
  }
  public Collection getOperationNames() {
    Vector operations=new Vector();
    // published operations
    operations.add("message");
    return operations;
  }
  // Web service method
  public void message(String msg) {
    System.out.println(msg);
  }
}
```

Listing 5. Instant Messaging Service

At the sender side, transferring a message consists of two steps: finding the receiver endpoint and sending the message. By utilizing the middleware facilities, a query used to match services can easily be created for finding service endpoints via their published attributes and descriptions, as shown in Listing 6.

```
public class IMQuery extends ServiceQuery {
  String memberName;
  String teamName;
  public IMQuery(String memberName, String teamName) {
    this.memberName=memberName;
    this.teamName=teamName;
  }
  public boolean matches(ServiceInfo info) {
    try {
      return teamName.equals(info.getAttribute(TEAM)) &&
        memberName.equals(info.getAttribute(MEMBER)) &&
        IMSERVICENAME.equals(info.getId()) &&
        IMSERVICENAMESPACE.equals(info.getNamespace());
    } catch (MissingServiceAttributeException e) {
      return false;
    }
  }
}
```

Listing 6. Instant Messaging Service Query

This query object can be used to find a matching service immediately or to subscribe and wait for a notification when the service appears in the network. In Listing 7 a subscription is done and notifications are caught by the method `notification()`. The advantage of using subscriptions is that if a recipient service is already known to exist the notification will be fired immediately, which means that there is not any particular delay compared to normal invocations. However, if the recipient has become unavailable meanwhile the invocation will be postponed until the recipient reappears. This kind of communication provides a high flexibility and convenience for asynchronous interaction in collaborative work.

```
public class IM implements ServiceSubscriber {
  ServiceDB db=Middleware.getServiceDatabase();
  // ...
  public void sendTo(String member, String team) {
    IMQuery query=new IMQuery(member,team);
    db.subscribe(query, this);
  }
  public void notification(ServiceQuery query,
                           ServiceInfo info) {
    try {
      info.invokeOperation(
            "message", new String[]{"hello"}, null);
      db.unsubscribe(query);
    } catch (Exception e) {
      System.err.println("Error:␣"+e.getMessage());
    }
  }
}
```

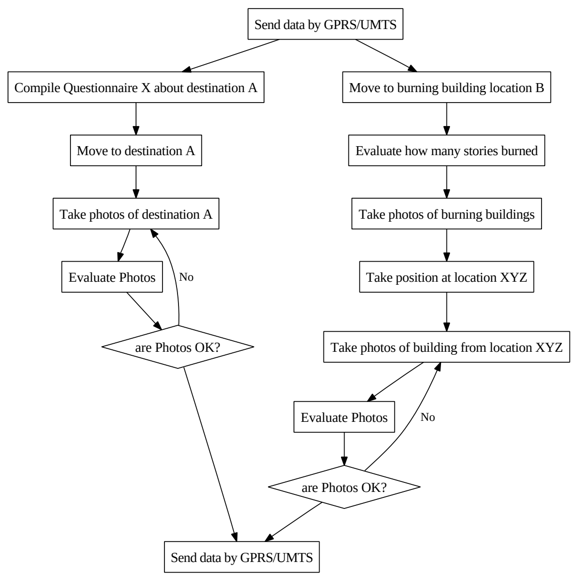
Listing 7. Sending Messages

4.2 Modeling and Executing Collaborative Teamwork

For the ease of understanding how Vimoware supports team members in different collaborative situations, let us look at a simple scenario in a disaster response. The response task flow, designed by the team leader, is shown in Figure 3; this flow was described in detail in [5]. Based on the middleware, our *Flow Execution Engine* automatically detects all *Task Handlers* in a team. Figure 4 shows four different members detected by the engine (see Section 5 for our testbed).

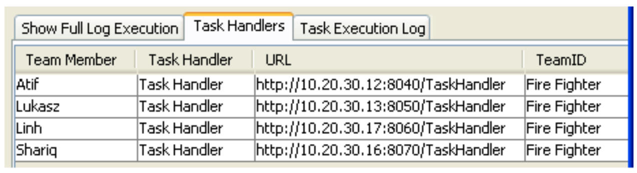
Based on the available information, the engine can execute different tasks by assigning the tasks to corresponding team members; task assignment can be done manually by the team leader or automatically based on a task assignment configuration (see Section 3.4). In this example, the engine just uses a simple FIFO model to assign tasks to the members who have no running tasks. As the above collaborative example is related to disaster response application, a single task configuration file is loaded on all *Task Handlers* with four set of options: `show` (show the description and goals of the task), `accept` (if a member has checked the task), `reject` (if a member has rejected the task), and `complete` (if a member has completed the task).

For example, for the task `Take photos of burning buildings`, the engine found `Lukasz` as the member who can handle this task, and it sent this



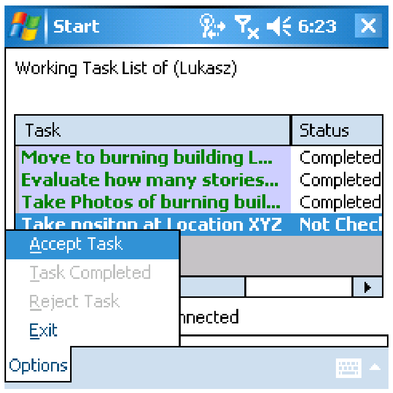
Send data by GPRS/UMTS
Compile Questionnaire X about destination A
Move to burning building location B
Move to destination A
Evaluate how many stories burned
Take photos of destination A
Take photos of burning buildings
Evaluate Photos
No
Take position at location XYZ
are Photos OK?
Take photos of building from location XYZ
Evaluate Photos
No
are Photos OK?
Send data by GPRS/UMTS

Figure 3. An illustrating flow in a disaster response [5]



Team Member	Task Handler	URL	TeamID
Atif	Task Handler	http://10.20.30.12:8040/TaskHandler	Fire Fighter
Lukasz	Task Handler	http://10.20.30.13:8050/TaskHandler	Fire Fighter
Linh	Task Handler	http://10.20.30.17:8060/TaskHandler	Fire Fighter
Shariq	Task Handler	http://10.20.30.16:8070/TaskHandler	Fire Fighter

Figure 4. Detecting existing team members



Working Task List of (Lukasz)

Task	Status
Move to burning building L...	Completed
Evaluate how many stories...	Completed
Take Photos of burning buil...	Completed
Take positon at Location XYZ	Not Checl

nnected

Figure 5. Interface of the Task Handler

task to `Lukasz`'s *Task Handler*. Figure 5 shows the list of tasks that `Lukasz` has received and performed. The team leader was notified when any member accepted or rejected a task and a team member can inform the leader when the member finishes the task and the result of the task can be received via the *Artifact Management*. The engine also logs all interactions for further analysis.

5 Performance Analysis of Vimoware

As Vimoware supports Web services on mobile devices, it is important to examine its performance in current mobile devices to prove its applicability. Our testbed consists of *2 HTC Tytn II* PDAs (Qualcomm MSM7200TM, 400MHz, 128 MB RAM, Windows CE 6.0, 2GB external MicroSD), *3 HP iPAQ 6915* PDAs (Intel PXA 270 416 MHz, 64 MB RAM, Windows CE 5.0, 2GB external MiniSD), a *Dell XPS M1210* (Intel Centrino Duo Core 1.83 GHz, 2GB RAM, Windows XP) notebook, and a *Dell D620* (Intel Core 2 Duo 2GHz, 2 GB RAM, Debian Linux) notebook. For running Java applications on the PDAs we used the IBM J9 implementation of the J2ME CDC 1.1 profile. Our tests were focused on the PDAs and the overhead caused by Vimoware.

First of all, it is important to state that even in the idle state, without Vimoware, the iPAQ uses 10-15% of the CPU and 18 MB of memory. Thus, approximately 30 MB are left for client applications. We noticed that the plain virtual machine of J9 uses 5 MB. The speed of the wireless link in our testbed rarely exceeded 300 KBit/s and the network had a high packet loss rate. These factors characterize the highly unstable ad-hoc network in our experiences.

The system load caused by Vimoware depends heavily on the applications running atop it. This includes, e.g., the number and complexity of deployed services, the number of clients and frequency of invocations, and the complexity of executed workflows. Hence, we examined only the performance of the core components. The middleware, including a running lightweight Jetty HTTP servlet container, consumes less than 2 MB of memory and uses only a marginal amount of the CPU. Using the Dell D620 to perform 1000 sequential invocations of a dummy service on the iPAQ, which delivered simple data (wrapped in a SOAP envelope of 2.5 KB), we obtained the following results:

	sync	async
average data size	2.7 KB	5.3 KB
average response time	74 ms	210 ms
average CPU load	50%	30%
average network traffic	296 Kbit/s	204 Kbit/s

Given that the network bandwidth was approximately 300 KBit/s and no other applications used the network, the bandwidth utilization in our experiments mostly reached the limit. For the synchronous invocation the transferring time of the request and response messages was approx. 95% of the whole response time. For the asynchronous invocation, which causes the double amount of traffic, due to the separately incoming response message, the transferring time used only approx. 70% of the response time. This was caused by the delay in the internal handling of asynchronous invocations in the middleware. Another observation was the CPU load which increased linearly with the network traffic. Because of no calculations performed, the CPU load was obviously caused by the operating system due to the transfer, encryption and decryption of the wireless messages.

Let us consider the impact of packet loss on the advertisements in the tests by examining the time taken to detect new services in the testbed. In Vimoware advertisements are relied on multicast and sent out in flexible intervals which adapt to the deployment times of the services. These intervals range from 1 second, which is the predefined minimum interval, to a configurable maximum interval, which was set to 5 seconds in our test cases. In the ideal scenario, where no packet loss exists at all, new services will be advertised within one second which means that the expectation value for the detection by the other peers is half a second plus the network delay for the exchanged packets. However, in cases of packet loss, the detection of new services for the other peers was delayed to the next advertisement round. In our experiments, which were conducted in an unstable wireless network, we experienced that depending on the loss rate $70 - 90\%$ of the services could be detected after approx. 0.6 seconds, while for the rest it took around 5.7 seconds. Such a delay of the detection is acceptable in real scenarios with such a high rate of packet loss.

6 Conclusion and Future Work

In this paper, we introduced Vimoware for supporting end-users to conduct collaborative work and developers to develop Web services and collaborative tools on mobile devices. Vimoware is Web services-based and consists of components which can be customized and programmed for different tasks. We have presented different applications, with the focus on collaborative work, to demonstrate the usefulness of Vimoware. We are currently integrating Vimoware with advanced task assignment and adaptation like in [5] and data sharing in the framework of the EU WORKPAD project. We are extending Vimoware to cover also RESTful services and to address reliability and checkpointing issues to deal with the service disruption problem.

References

[1] Lime: A middleware for physical and logical mobility. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 524, Washington, DC, USA, 2001. IEEE Computer Society.

[2] E. Avilés-López and J. A. García-Macías. Providing service-oriented abstractions for the wireless sensor grid. In C. Cérin and K.-C. Li, editors, *GPC*, volume 4459 of *Lecture Notes in Computer Science*, pages 710–715. Springer, 2007.

[3] S. Berger, S. McFaddin, C. Narayanaswami, and M. Raghunath. Web services on mobile devices - implementation and experience. *wmcsa*, 0:100, 2003.

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

[5] M. de Leoni, M. Mecella, and G. D. Giacomo. Highly dynamic adaptation in process management systems through execution monitoring. In G. Alonso, P. Dadam, and M. Rosemann, editors, *BPM*, volume 4714 of *Lecture Notes in Computer Science*, pages 182–197. Springer, 2007.

[6] H. D. H. Duong, C. Melchiorre, E. M. Meyer, I. Nieto, M. Arrufat, P. Pelliccione, and F. Tastet-Cherel. Popeye: a simple and reliable collaborative working environment over mobile ad-hoc networks. In *The 3rd International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2007)*, number 1-4244-1317-6. IEEE Computer Society, 2007.

[7] G. Hackmann, M. Haitjema, C. D. Gill, and G.-C. Roman. Sliver: A bpel workflow process execution engine for mobile devices. In A. Dan and W. Lamersdorf, editors, *ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 503–508. Springer, 2006.

[8] V. Issarny, D. Sacchetti, F. Tartanoglu, F. Sailhan, R. Chibout, N. Lévy, and A. Talamona. Developing ambient intelligence systems: A solution based on web services. *Autom. Softw. Eng.*, 12(1):101–137, 2005.

[9] Y.-S. Kim and K.-H. Lee. A light-weight framework for hosting web services on mobile devices. In *ECOWS '07: Proceedings of the Fifth European Conference on Web Services*, pages 255–263, Washington, DC, USA, 2007. IEEE Computer Society.

[10] M. de Leoni, A. Marrella, M. Mecella. D2.1 The WORKPAD Architecture, November 2007. WORKPAD Consortium. http://www.workpad-project.eu/documents/D2.1v1__v1.0.pdf.

[11] S. Oh, G. C. Fox, and S. Ko. Gmsme: An architecture for heterogeneous collaboration with mobile devices. In *The Fifth IEEE International Conference on Mobile and Wireless Communications Networks (MWCN 2003) Singapore in September / October, 2003*, September 2003.

[12] L. Pajunen and S. Chande. Developing workflow engine for mobile devices. In *EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, page 279, Washington, DC, USA, 2007. IEEE Computer Society.

[13] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-osgi: Distributed applications through software modularization. In R. Cerqueira and R. H. Campbell, editors, *Middleware*, volume 4834 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2007.

[14] V. Sacramento, M. Endler, H. Rubinsztejn, L. Lima, K. Goncalves, F. Nascimento, and G. Bueno. Moca: A middleware for developing collaborative applications for mobile users. *Distributed Systems Online, IEEE*, 5(10):2–2, Oct. 2004.

[15] E. Zeeb, A. Bobek, H. Bohn, S. Pruter, A. Pohl, and H. Krumm. WS4D: SOA-Toolkits making embedded systems ready for Web Services. In *Open Source Software and Productlines 2007 (OSSPL07)*, Limerick, Ireland, 2007.