A proxy-based architecture for dynamic discovery and invocation of Web Services from mobile devices

Hassan Artail, Kassem Fawaz, and Ali Ghandour
American University of Beirut
Department of Electrical and Computer Engineering
P.O.Box: 11-0236, Riad El-Solh Beirut 1107 2020, Lebanon
E-mails: {hartail, kmf04, ajg04}@aub.edu.lb

Abstract

Mobile devices are becoming more pervasive, and it is becoming increasingly necessary to integrate Web services into applications that run on these devices. We introduce a novel approach for dynamically invoking Web service methods from mobile devices with minimal user intervention that only involves entering a search phrase and values for the method parameters. The architecture overcomes technical challenges that involve consuming discovered services dynamically by introducing a man-in-the-middle (MIM) server that provides a Web service whose responsibility is to discover needed services and build the client-side proxies at runtime. The architecture moves to the MIM server energy-consuming tasks that would otherwise run on the mobile device. Such tasks involve communication with servers over the Internet, and XML-parsing of files and on-the-fly compilation of source code. We perform an extensive evaluation of the system performance that includes scalability measurements as it relates to the capacity of the MIM server in handling mobile client requests, and device battery power savings resulting from delegating the service discovery tasks to the MIM server.

Keywords: Web service discovery, dynamic invocation, mobile devices, mobile computing

Digital Object Indentifier 10.1109/TSC.2010.49 1939-1374/10/$26.00 © 2010 IEEE

1. INTRODUCTION

Recent trends in mobile computing and the popularity of mobile devices has motivated interest in accessing Web services from mobile devices in order to extend their functionality and gain access to remote data. However, the particularities and limitations of mobile devices and the mobile environment pose great challenges for consuming Web services [39]. To begin with, when using UDDI registries for service discovery, multiple costly network round trips are needed, and frequent unavailability of the wireless network may cause failures in the service discovery process [39], and will hinder the completion of the user request [34]. Additionally, there are several issues and challenges that emerge from the fact that mobile devices have lower processing power, limited bandwidth, less memory, and finite battery power when compared to desktops [34]. All the above suggest that architectures which target mobile devices should aim to minimize their interactions with the network and reduce their resource-consuming processing whenever possible. Actually, the mentioned issues and challenges led to the architectural configuration proposed in this paper which introduces a server "in the middle" whose role is the provisioning of services in a way that is adapted to the capabilities and the constraints of mobile devices. Most of the workload involved in the dynamic discovery of Web services is passed to the server, thus relieving the mobile device of the time- and processor-consuming tasks, mainly related to parsing of WSDL files [33].

To access a Web service dynamically, a solution is for its WSDL file to be parsed by the mobile device application that will interact with the service directly; or another solution is to build a compiled assembly from the WSDL file at runtime and use it as a proxy by the mobile application to interact with the service. Given the argument we made earlier for a server-based solution to interface mobile devices with Internet-based Web services, it makes little sense to do the parsing of the WSDL files on the device. On the other hand, by building the proxy on the server when needed and sending it to the mobile device, the latter will be able to interface with the Web service as though it were a local process. Another reason, although technical, why the proxy should be generated by the server is that language compilers are not meant to run on mobile devices, and no commercial compilers have been developed to date for such devices. Later in the paper we compare these two approaches from different aspects, and prove the advantageous features of the second approach, which we have adopted. Moreover, since mobile devices depend primarily on their battery power to communicate and run applications, power conservation becomes critical if the device needs to communicate over the Internet and serve user requests that involve discovering services and invoking their methods. For this reason, we conduct experiments which illustrate the battery energy savings that are attributed to the implemented design. In summary, this paper describes an effective architecture which allows mobile users to acquire needed data through dynamic invocation of Web service methods by merely providing search phrases and method argument values into a dynamically generated GUI. Our design is based on the framework defined in [11] for realizing dynamic service invocation, and stresses practical aspects as perceived by the user in terms of interface simplicity and effectiveness in returning desired results.

There are two different techniques for accessing Web services: using SOAP and the REST approach. SOAP defines the XML-based information which can be used for exchanging structured and typed information in a decentralized distributed environment between peers, including Web services and UDDI registries that represent service brokers through which providers advertise their services. The process of Web service invocation starts when the client-side proxy wraps the user's request (method call) into a SOAP message and sends it to the service, which extracts the call from the received message, executes the call to produce the results, wraps the results into a SOAP message, and sends it to the client. Upon receiving the message, the same proxy extracts the results and hands them over to the calling client application. Before an application can begin communicating with a service, it must first discover it and get its specifics (supported methods and invocation details), and then generate its proxy. The proxy allows a client application to make method calls as though it were calling a local function. The proxy is created by first generating a source file from the service's WSDL (Web Service Description Language) file, which describes the Web service, how to access it, the operations it performs, the types of parameters to be passed to each of the supported methods, and the types of returned results. After the source file is generated, it is compiled into a proxy class that is finally registered with the client application.

REST, on the other hand, stands for Representational State Transfer [9] and is an architecture style, according to which a Web service makes available a URI that returns an XML document (representation of the resource) which can include links to other documents. This allows the user to drill down to get more detailed or other related information. It should be noted that the URIs are logical, not physical, and could correspond to conceptual entities (not static documents). To send data to the server, the service also provides a URI that allows the user to create an instance document which conforms to a schema publicized in an XML document, thus allowing him to submit the input document as the payload of an HTTP POST.

The process of accessing a Web service is illustrated in Figure 1 for both approaches, where in the case of REST, the shown "Connector" encapsulates the activities of accessing resources and transferring representations (states of resources) [12].

2. RELATED WORK

Several approaches have been proposed for enabling dynamic invocation of Web service methods from mobile devices. However, these methods either proposed conceptual solutions, or suggested architectures that include components for assisting mobile devices in accessing Web services. Our proposed solution is of the second type, but differs from the proposed schemes in that it is complete, fully dynamic, employs generic technologies, and is not restricted to a particular platform. Indeed, the survey that follows shows how the proposed

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

solutions in the literature either only allow non-dynamic access to deployed Web services from mobile devices, or provide dynamic access to services, but not from mobile devices.

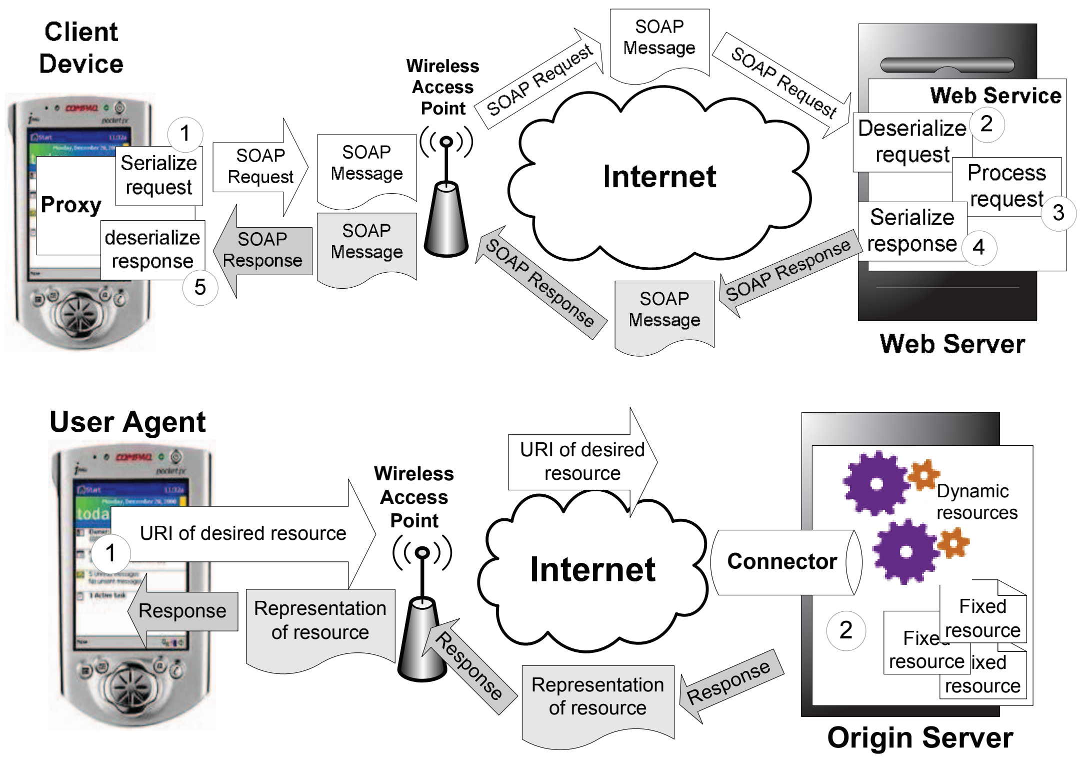


Figure 1. Using SOAP (top) and REST (bottom)

Starting with the first class of solutions mentioned above, an application for Peer-to-Peer (P2P) Web services is suggested in [14] for use in ad-hoc networks. A distinction between two different P2P realizations is made: one in which a stand-alone node acts as a broker and another one where no centralized service-broker is available. In the second scenario, some nodes within the environment must provide the broker service. The paper presents a Java-2 Micro Edition (J2ME) implementation that was used to analyze the memory usage and response time of the SOAP server. In [7], a system was presented to study the resource consumption and performance of providing Web services on mobile phones. The analysis of a prototype which was developed using Personal Java on a Sony-Ericsson P800 phone shows that the implementation is able to handle up to eight concurrent users with reasonable time delays. Finally, two architectures were proposed in [29] for accessing Web services from mobiles. They use the J2ME Web Services API (WSA) and Short Messaging Service (SMS) to target high- and low-end mobile devices, respectively. WSA integrates basic support for Web services invocation and XML parsing into the runtime environment of the mobile device. A network application that resides on a WSA-enabled device must include a stub, which is generated by a tool on a development workstation, and then deployed to the device. The application can then employ the stub and the runtime to parse XML SOAP messages using JAXP, and consume the service using JAX-RPC. To access Web services using SMS, an SMS gateway that translates short messages to HTTP requests was used to connect the SMS Center to an application server, which in turn translates HTTP requests/responses to/from SOAP messages. Every time a Web service needs to be accessed, programming and configuration is required in order to create a stub or to set up communication using SMS.

In another type of approaches, servers in the middle were employed to make accessing Web services faster or to host Web services on mobile devices. In [24] it was argued that the normal Web service architecture which integrates a requestor, a broker, and a provider does not allow for designing a suitable service invocation on mobile devices. A service gateway is suggested to be included between the requestor and the rest of the architecture. It was recognized that such a solution requires additional code on both the mobile phone and the service side. Simulation results showed that a mobile phone accessing a Web service via the service gateway is faster than using the KSOAP protocol. On the other hand, [18] describes a platform for hosting Web services on mobile devices by building on the Jini Surrogate Architecture Specification [31]. In this platform a service that is hosted on a mobile device has a surrogate that runs on a server connected to a fixed network. With this arrangement, clients in the wireless and fixed networks communicate with the mobile service through the surrogate, whereas communication between mobile services and their respective surrogates takes place over HTTP. The paper briefly discusses three application scenarios but it does not present any performance results nor does it mention the limitations of the architecture.

For dynamic invocation of Web service methods, few platform-specific technologies were developed. The *DynWsLib* library [32] is a .NET Framework-based technology that works by generating the client-side proxy class at runtime. However, according to one developers' forum, this library worked consistently with services running on the local server, but not always when trying to reference remote services. Moreover, this library, which is no longer supported, does not provide an effective mechanism to handle specific types during the proxy generation process. The *ProxyFactory* project [8], which builds on DynWsLib, tries to address this issue, and more generally, tries to take advantage of the unification of communications technologies provided by.NET. However, ProxyFactory cannot run on the Compact Framework (lightweight version of .NET that targets "smart" devices) because the class that is needed for serialization is not available in this framework.

Finally, we discuss the work in [30] which proposes a design for automatic generation of multimodal user interface from discovered WSDL files and a mechanism for service invocation. The approach relies heavily on the XForms technology [36] and requires the client browser to understand another markup language. Although the architecture provides a transparent invocation mechanism based on runtime binding of Web services, the provider is required to publish the WSDL files for the available Web services to a local server, which acts as a WSDL files repository. Upon receiving a new WSDL file, the local server generates corresponding multimodal abstract user interface components based on XForms and adds them to a service list. This work, however, does not address the issue of automatic proxy generation, nor does it offer a proof of concept or any experimental results.

3. PROPOSED SYSTEM DESCRIPTION

Our design is based on SOAP, but we describe later how it can work with REST. The architecture incorporates a man-in-the-middle (MIM) server which will be used by mobile devices to discover needed Web services and build their proxies. After getting the proxy, a mobile device can invoke a particular method of the Web service and get the desired results. More specifically, the MIM server offers a Web service which exposes a Web method that the mobile device invokes and passes to it a search string. The MIM server's special service (or simply the MIM server) compares the submitted string to cached short-descriptions of Internet Web services and generates a short list of services that best-match the user's string. Next, the MIM server downloads the WSDL files of the short-listed services, and uses the included descriptions of the supported methods to identify the most appropriate service (i.e., the one whose method's description matched the user's query the most). After this step, the MIM server generates a source code file from the WSDL file of the chosen service and then compiles it using libraries that target the mobile device platform to generate the client-side proxy and ship it to the mobile device. At this point, the call that was originally made by the mobile device application to the MIM server's Web method returns with information about the Internet Web service. This includes: name of service and its chosen method, number and types of input method parameters, and number and types of returned results. With such information, the mobile application generates a dynamic GUI for the user to supply values for the Web method parameters, and then another GUI to display the results. The cached short descriptions mentioned above are downloaded by an independent process on the MIM server which periodically queries UDDI servers. The above operations are summarized in Figure 2.

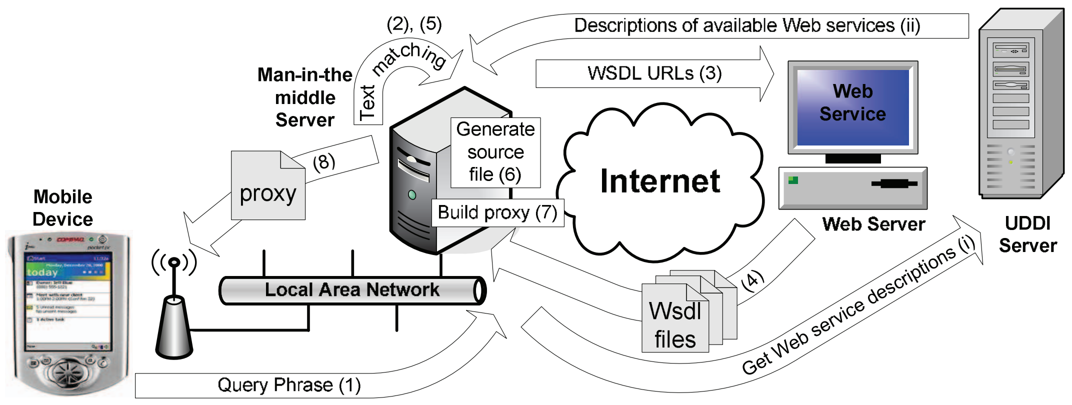


Figure 2. Overview of operations in the proposed system

The general architecture of the proposed system is shown in Figure 3, where it is evident that almost all the client processing has been moved to the MIM server. To start with, the MIM server has a process that wakes up periodically to download a list of Web service descriptions and associated URIs from a designated set of UDDI registries. More notably, the MIM server offers a Web service that interfaces to three main processes which jointly fulfil the user's request. The first of these is the *Text Matching Process* which serves two purposes: first, to generate a short list of candidate Web services based on matching their cached short descriptions with the user's supplied search string, and second, to identify the most appropriate Web service among those short-listed based on matching the methods' description found in their downloaded WSDL files with the user's string.

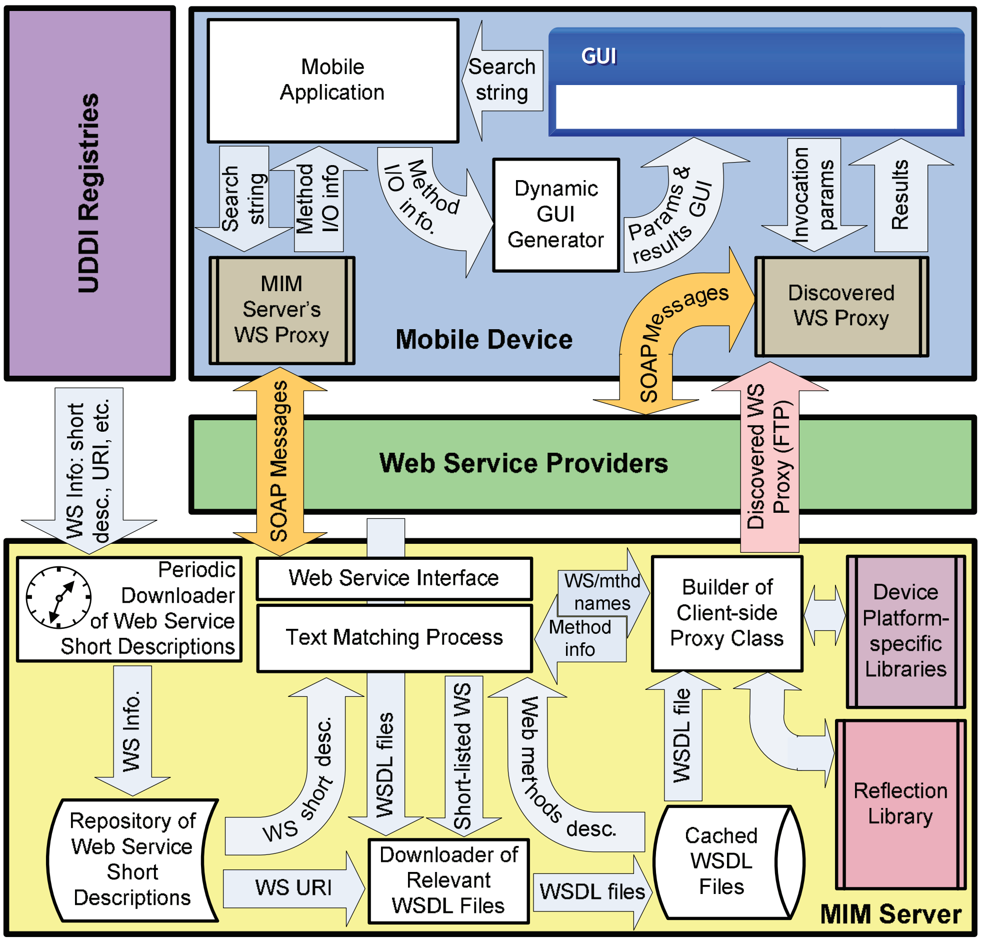


Figure 3. System architecture

The second process is the *WSDL File Downloader* which takes as input the URIs of the shorted-listed services, and downloads the files if they are not in the cache. Finally, the third main process is the *Proxy Builder*, which generates a client-side proxy class and sends it to the mobile device via FTP. This same process identifies the number and types of input and output parameters of the Web method whose name was passed by the Text Matching process, and returns this information to the latter so it can be passed on to the device.

To speed up processing on the server and improve the scalability of the architecture, the described three processes are multi-threaded and communicate through socket interfaces by receiving and sending messages, and in some case sending files using FTP. Moreover, we also improve the average response time of the server by caching the WSDL files while assigning them time-to-live (TTL) values and keeping track of their access rate. This keeps the list of cached WSDL files manageable and gives preference to those that are most used.

On the mobile device, the call to the MIM Server's Web service returns the names, types, and descriptions of the input and output parameters of the Web method to invoke. This is used by the Dynamic GUI Generator to draw the GUI on the device, thus enabling the mobile application to interact *directly* with the discovered Internet Web service by invoking its method with parameter values input by the user and getting from it the corresponding results.

4. ANALYSIS

In this section we analyze the cache hit rate of the system and then consider its scalability, which is about examining the performance of the MIM server under increased request rate.

4.1 Cache Hit Rate

The MIM Server caches WSDL files for its own use to reduce network traffic and speed up the processing of the mobile device's request. The cached WSDL files form a subset of the set of possible WSDL files that may be downloaded. To get an idea of the size relationship, about 570 files could be cached if a 10MB cache size is allocated and an average WSDL size of 18KB is used. Following the lead of other works [3], we use the Zipf distribution [40] to model the access pattern of Web services. With Zipf, the probability of requesting item i (item 1 is the most popular) is $1/\left(i^{\theta}\sum_{k=1}^{N}1/k^{\theta}\right)$, where N is the total number of items and θ is a parameter that controls the non-linearity of the distribution. Eventually, most WSDL files in the cache will be popular. This roughly makes the probability of a cache hit be the part of the area under the distribution curve: $p = \sum_{i=1}^{S_C/S_{WSDL}} 1/\left(i^{\theta}\sum_{k=1}^{N}1/k^{\theta}\right)$, where S_C is the allocated cache size and S_{WSDL} is the WSDL file average size. For illustration, the left part of Figure 4 illustrates the effect of the value of θ on the access distribution, while the right part shows the probability of cache hit for different cache sizes and θ parameter values.

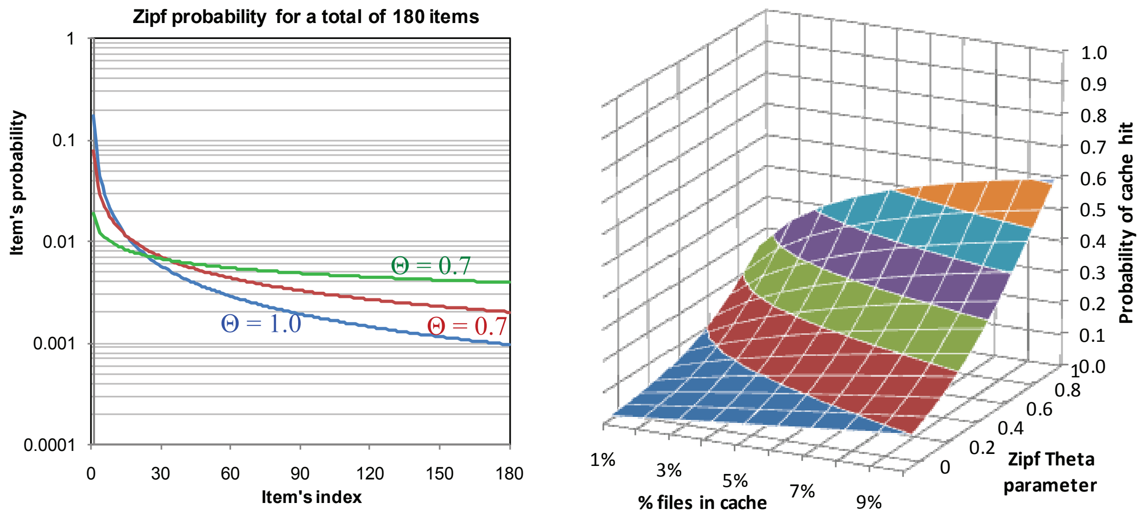


Figure 4. Zipf probability (left) and probability of cache hit (right)

Since the space allocated for caching the WSDL files is finite, a mechanism is needed for cache replacement. For this, we use a version of the least recently used (LRU) policy that works with objects having different sizes [1]. That is, when a file is to be added to a full cache, more than one WSDL files may need to be removed in order to create sufficient space. The size and a time stamp for each cached file are stored in a hash table, and each time there is an access for a WSDL file, a counter is incremented and the time stamp is updated. The dynamic frequency of a given WSDL file is the inverse of the number of accesses since the last access. To create space for an incoming WSDL file with size S_n, we remove from the cache the least number of files whose cumulative size is greater than S_n such that the sum of their dynamic frequencies is the minimum. This removal policy will not change in a major way the access distribution of WSDL files in the cache relative to the files on the Internet because it is the files that fall on the tail end of the distribution curve that will tend to be selected for replacement, and thus, the probability of hit remains generally valid.

4.2 Scalability Analysis

To analyze the MIM server's scalability in terms of the number of users, we abstract the operations of the server main processes and describe quantitatively the interactions between each process and the underlying hardware resources. In our analysis, we define the main three hardware resources that affect the server operation: memory, processor, and network. Storage utilization was ignored as it poses no bottleneck in current server implementations. In line with [5], for a smooth server operation and to insure affordable server response time, 1) memory utilization must be below 85% to avoid page faults and swap operations, 2) processor utilization must stay below 75% to make room for kernel and other third party software to operate with no effect on the overall server operation, and 3) network utilization should be kept under 50% to prevent queuing delays at the network interface.

In our analysis, it is convenient to model the processor and network performances using queuing theory, but first, we need to decide on the appropriate queuing model. Considering processor performance, it is well established that an M/G/1-RR (round robin) queuing model would be suitable [4], [17], [37]. It is designed for round-robin systems (like operating systems) and is generic, as it requires the mean and variance without the full distribution of the service time. This model assumes that requests to the processor follow a Poisson distribution, so that the distribution of the inter-arrival time between requests is exponential with mean λ requests/second, and each request is given a time slice on the processor. Since all requests in our case have the same priority and have low variations in their sizes, the queuing model can be reduced to M/G/1-PS (processor sharing). We assume that at full utilization, the processor can serve μ_p requests per seconds, basically the inverse of a job size, denoted by total processing time. It then follows by queuing theory and little's theorem that the processor utilization is $\rho_P = \lambda/\mu_p$. The memory utilization ρ_M is given by the amount of memory used by the server M_u divided by the total memory M_T: $\rho_M = M_u/M_T$. Finally, the network utilization ρ_N is the number of arriving requests λ over the number that can be handled μ_N: $\rho_N = \lambda/\mu_N$, also by applying little's theorem. The requests to the network card can be modelled by a Poisson random process, where the service time is constant, basically the transmission delay [22], so an M/D/1 queuing model is appropriate to be applied. We now derive the utilizations so we can obtain an expression for the maximum number of simultaneous user requests.

1. The periodic downloader process is a single threaded process that sleeps for T_s and then wakes up to download information from the UDDI registries. It is independent of the number of users and thus represents a fixed cost in terms of server resource consumption.
2. The text matching process is multithreaded, whereby the number of threads is equal to the number of users having pending requests. Each thread performs a first stage text matching, sleeps until WSDL files are ready, and then performs another text matching function. Hence, each connected user maps to a thread that consumes memory to maintain its stack, utilizes the processor to run the two instances of text matching, and incurs additional overhead resulting from thread context switches. For the computations below, we suppose that the first text matching instance takes T_1 seconds to execute, the second takes T_2 seconds, and each thread incurs a context switch of c seconds, while thread creation and destruction overheads can be ignored as they are in the order of microseconds [26]. Considering the processor utilization, an expression for the number of served concurrent users can be found by applying the expression of the average number of requests in the processor, and it is $N_P=\lambda/(\mu_p-\lambda\mu_p)$ [21]. It follows that the total memory used by the process is $H+N_P\times ST$, where ST is the thread stack size and H is the dynamic

memory allocated. The main components of H are the WSDL files that must be loaded into memory for the second text matching operation. Assuming the shortlist of WSDL files contains on average 3 WSDL files, and the average WSDL file size is S, then the memory utilization of this process can be approximated by $3N_P \times S + N_P \times ST$. Concerning the network utilization, the external network interactions of this process are limited to the incoming user requests from the mobile devices and the outgoing response, and so, it will not constitute a bottleneck. Actually, the MIM server interacts with the mobile devices through its Web service, and the SOAP requests/responses are received/sent by its Web service interface (shown in Figure 2). However, in this analysis we assume for simplicity that the text matching process is the one that interacts with the mobile devices.

3. The WSDL file downloader process is also multithreaded, where in this case each thread maps to a WSDL file to download. For each considered file, this process checks for its existence in the cache, and downloads it from the Internet if a miss occurs. This process receives λ requests per second from the text matching process. It does not cause high processing load, but rather a networking load that also affects the memory utilization. The network can serve users at a rate of μ_N users per seconds, where each user corresponds to an average of three WSDL files, each having an average size of S KB. Using a cache hit rate of h, each user request will effectively be downloading $3S(1\text{-}h)$ KB. Now, assuming the network interface to the Internet has a bit rate of B kbps, we conclude that it can serve $\mu_N = B/(24S(1\text{-}h))$ users per second. Using the queuing model M/D/1, the number of concurrent users waiting on the network interface is given by $N_N = \lambda \times (2\mu_N – \lambda\mu_N) / 2\mu_N \times (\mu_N – \lambda\mu_N)$ [10]. Each thread will have a stack allocated, and dynamic memory whose size is equal to that of the file downloaded. Given that the execution code size is negligible, the memory usage of this process can be approximated as $3S \times N_N (1\text{-}h)$.

4. The proxy builder process is multithreaded too. Each thread maps to a user's request, and executes the software that will create the source code and compile it ("wsdl.exe"), and FTPs the proxy class to the client device. A single invocation of "wsdl.exe" will incur a processing load as well as a memory load. The latter is denoted by M_W and includes the linked libraries and the WSDL file to generate the proxy class from. This results in a total memory usage of $N_P \times M_W$. For processing, we use T_W to symbolize the time of one process and c_w to denote the context switch time. Finally, the interaction between this process and the client devices (i.e., FTP'ing the proxy) occurs in the internal network and is not likely to be the bottleneck in the presence of the external network interface.

We now use the above definitions to develop expressions that lead to a measure of the load on the server. To start with, the CPU can serve $\mu_p = 1/(T_1+T_2+T_W+c+c_W)$ users per second. From before, the processor utilization is $\rho_P = \lambda/\mu_p = \lambda(T_1+T_2+T_W+c+c_W)$, which must be less than 0.75, or else, the processor will be the bottleneck and will limit the server's scalability. Next, the total memory usage of the server processes is $M_u = 3N_P \times S + N_P \times ST + 3N_N \times S(1-h) + N_P \times M_W$. Then, the memory utilization of the processes is given by $\rho_M = (3N_P \times S + N_P \times ST + 3N_N \times S(1-h) + N_P \times M_W)/M_T$ and must be below 0.85. Finally, the utilization on the external network interface is given by $\rho_N = 24S \times \lambda(1-h)/B$ and should be below 0.5. The expressions of ρ_P and ρ_N are linear in λ, and their solutions yield $\lambda_P < 0.75/(T_1+T_2+T_W+c+c_W)$ and $\lambda_N < B/(24S(1-h))$, respectively. On the other hand, the expression of ρ_M is cubic in λ, and its solution λ_M, if it exists, is in the form $\lambda_M < \lambda_{M1}$ and $\lambda_{M2} < \lambda_M < \lambda_{M3}$, or $\lambda_{M1} < \lambda_M < \lambda_{M2}$ and $\lambda_M > \lambda_{M3}$, where λ_{M1}, λ_{M2}, and λ_{M3} are the possible solutions of $\rho_M - 0.85 = 0$. We hence conclude by stating the maximum number of *sustained* simultaneous requests per second: $\lambda_{\max} = \min(\lambda_P, \lambda_N, \lambda_M)$. The solutions are shown in the left chart of Figure 5 as plots of the utilizations versus λ (by setting h=0 and using values for the other parameters in accordance with the literature). To explain these results, we consider as an example one request that entails downloading 3

18KB WSDL files. Having an available download rate of 256 kbps, the network interface will be fully utilized for 1.7 seconds. This implies that for $\lambda \geq 1$, the network interface will remain fully utilized and queuing will occur! This however is not unusual in computer networking, where a given network card transmits or receives at full capacity for a period of time, since packet buffering can mitigate the effects of temporary bottlenecks. However, the network will start dropping requests if the high request rate is sustained for prolonged periods. This leads us to investigating the probability of simultaneous requests: with *K* users who could use the MIM server's service to discover Web services, the probability of more than *k* users having requests being processed simultaneously is $1-\sum_{i=0}^{k}\binom{K}{i}p^{i}(1-p)^{K-i}$, where *p* is the fraction of time the user will have a request in process. Considering the above scenario, setting *k* to 1 (i.e., two or more simultaneous requests), varying the average duration *T* during which the user submits one request ($p=u/T$, where *u* is the network utilization time per one request), and considering several values of *K*, we get the results in the right chart of Figure 5. These theoretical results indicate, for example, that if for every 1000 users a MIM server is allocated, and given normal user access rates, the MIM server will be able to serve every request. We confirm this conclusion when we describe the experimental results below.

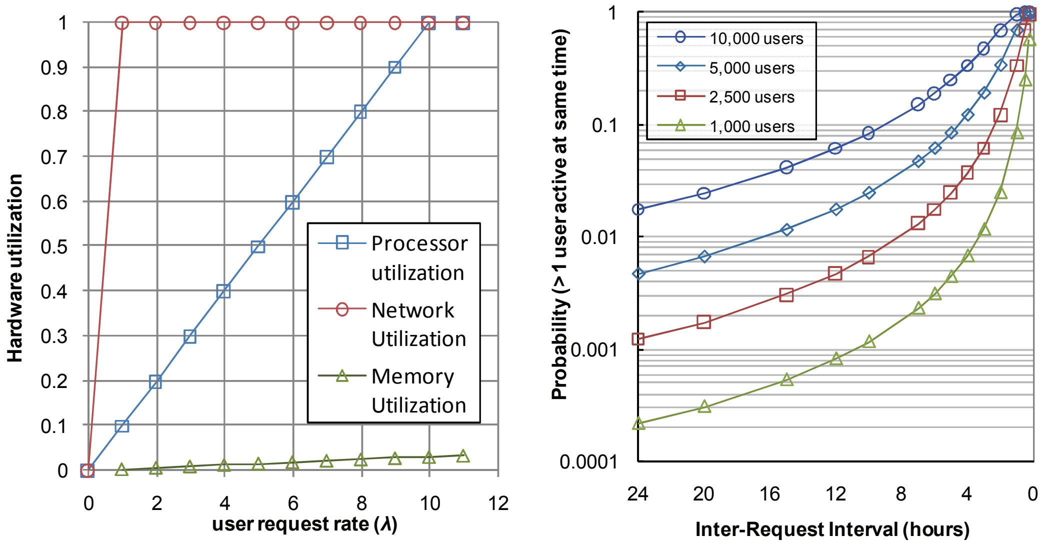


Figure 5. Hardware utilization (left) and probability of simultaneous access (right)

We finish this section with a brief discussion of the server average response time, which depends on the loading of the three hardware resources considered above. This time is mainly the total of the processing time and network processing and delays. The components of the former were derived above, while the network part mostly includes the delay of receiving the WSDL files, queuing delay, and the Internet round trip time (*RTT*). For each request, we assume the MIM server requests all WSDL files immediately one after another, thus resulting in nearly one RTT for all files, since they will be served in parallel by remote servers. The reception delay is 24S/B (assuming on average of 3 WSDL files), as was defined above, while the queuing delay depends on the number of requests in process: with *N* users, it is $\frac{1}{N}\sum_{n=1}^{N-1}(n-1)(24S/B)$ = $(24S/B)(N\text{-}1)/2$. As an example, using 150 ms for the total processing delay, the same values for *S* and *B* as before, 4 seconds for RTT, and having 60 concurrent requests in process, we get an average response time of 0.15+4+1.73+50.98=56.86 seconds.

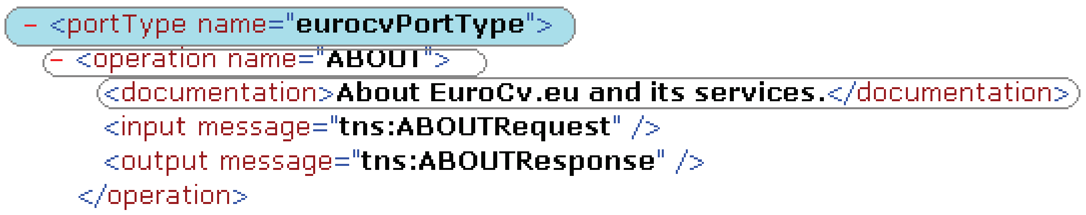
5. IMPLEMENTATION

The Web service client on the mobile device was implemented using the Windows .NET Compact Framework (CF) and the C# programming language. The application was installed on an HP iPAQ hx 2790 Pocket PC running the Windows Mobile operating system and set up to communicate with a wireless access point using Wireless LAN. The MIM server was implemented on a Fujitsu Siemens laptop with a Centrino Processor and 1GB of memory.

5.1 Discovering Web Methods

There are several UDDI registries available on the Internet that offer APIs for locating desired services. For the purpose of evaluation, we restricted ourselves to the private *XMethods* registry, which offers methods that allow for service summary structures for all active services listed at Xmethods. After it starts, a background process on the MIM server calls this method and caches the obtained service descriptions locally in a simple MySQL database table. The mobile application on the Pocket PC binds to a proxy class that interfaces to the MIM server's developed Web service, which in turn offers a method that encapsulates the functions discussed below. It is through this "remote" method that the user submits his search query. At the MIM server, the query is matched against the cached descriptions of the Web services using the *Boyer-Moore* algorithm [2], which takes two strings and searches for the occurrence of one in the other. This algorithm is suitable for this type of applications as it works the fastest when the alphabet is moderately sized and the pattern is relatively long. It scans the characters of the pattern from right to left beginning with the rightmost character. In our implementation, the measure of correspondence between the user-supplied search string (e.g., *P*) and the Web service description string (*T*) was the aggregate length of the found non-repeating matches between *P* and *T*. The output of the matching process is a short list of web services: ones that relate most to the input phrase. If the search returned more than five services, the user was asked to supply a more specific phrase that better describes his need.

The next step is to obtain the WSDL files of the selected services. For each service, if the file is not found in the cache, an HTTP Web request is created from the WSDL URL, and an HTTP Web response is then generated from the request and put into a stream that constructs a string builder that will contain the actual WSDL file. Each of the needed WSDL files is now parsed in order to get the web methods of the web service and their corresponding documentations. First, the whole XML file is searched to find the word "portType", then the word "operation" (tells the name of the web method), and finally the word "documentation", as seen in the example of Figure 6. For each service, the parser returns an array of operations, which is a class consisting of two strings: the Web method's name and its documentation.



```
- <portType name="eurocvPortType">
  - <operation name="ABOUT">
      <documentation>About EuroCv.eu and its services.</documentation>
      <input message="tns:ABOUTRequest" />
      <output message="tns:ABOUTResponse" />
    </operation>
```

Figure 6. Identifying method names and their documentations in WSDL files

The next step in the process searches the documentations in the array of operations for the best match with the search phrase, again using the Boyer-Moore algorithm. The outcome of this search is one Web method that best-matches the input phrase.

5.2 Proxy Class Generation and Compilation

After identifying the Web service, the proxy (dll) is built from its WSDL file. This task involves generating a class (source code), compiling it, and then sending it to the mobile device where it is used for direct method invocation. The .NET/C#-specific implementation is depicted in the class diagram shown in Figure 7, which illustrates the used classes and the associationss among them. All the involved code runs on the MIM server, obviously except the mobile application and the client proxy after being sent to the device.

Basically the URL of the discovered Web service by the MIM Server is used to get the service's details. More specifically, the Server retrieves the service descriptions and schemas and saves them in a list, and then performs a set of steps to generate instances of classes through which the source code for the service's WSDL file is programmatically generated. After generating the C# class, the assembly file (compiled proxy) is created through a set of additional steps that uses compiler parameters and assemblies specific to smart devices.

The *Reflection* API allows a C# program to inspect and manipulate itself. It can be used to effectively find the types in an assembly and dynamically invoke methods in it. In our case *Reflection* was used to get the names and types of the identified Web method's input and output parameters and to store them in an array of structures that is returned to the mobile application on the Pocket PC. With this information, the mobile application generates a dynamic GUI. In our implementation, for every input parameter a text box along with a corresponding label (using the parameter's name) was drawn. Finally, after getting the user's input through the generated GUI, the method is invoked at run time, also using *Reflection*. At last, the returned results are displayed in another dynamic GUI generated based on the output parameters determined earlier. If the invocation returns a set of answers, we display two buttons at the bottom of the form to allow the user to navigate through the results.

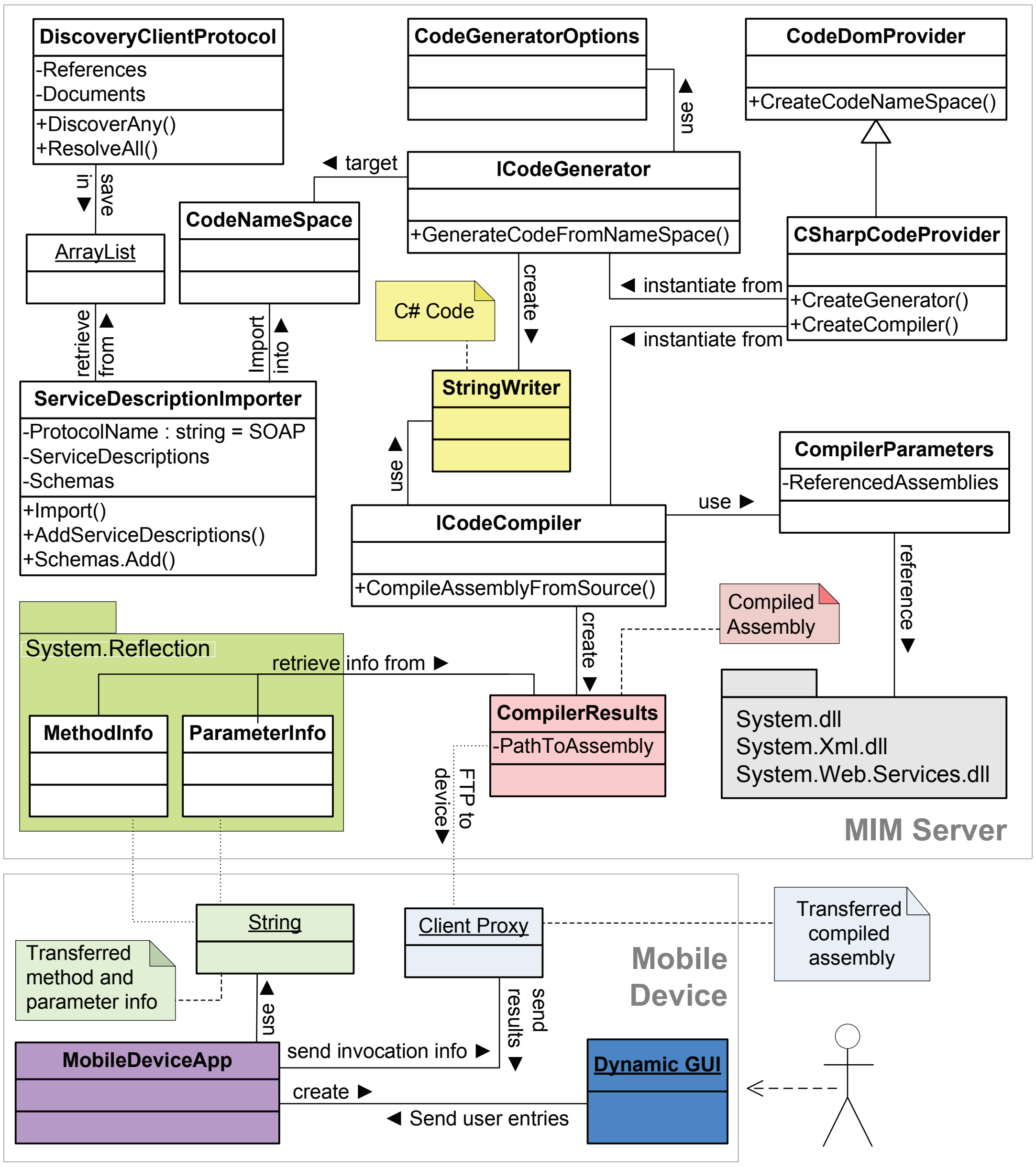
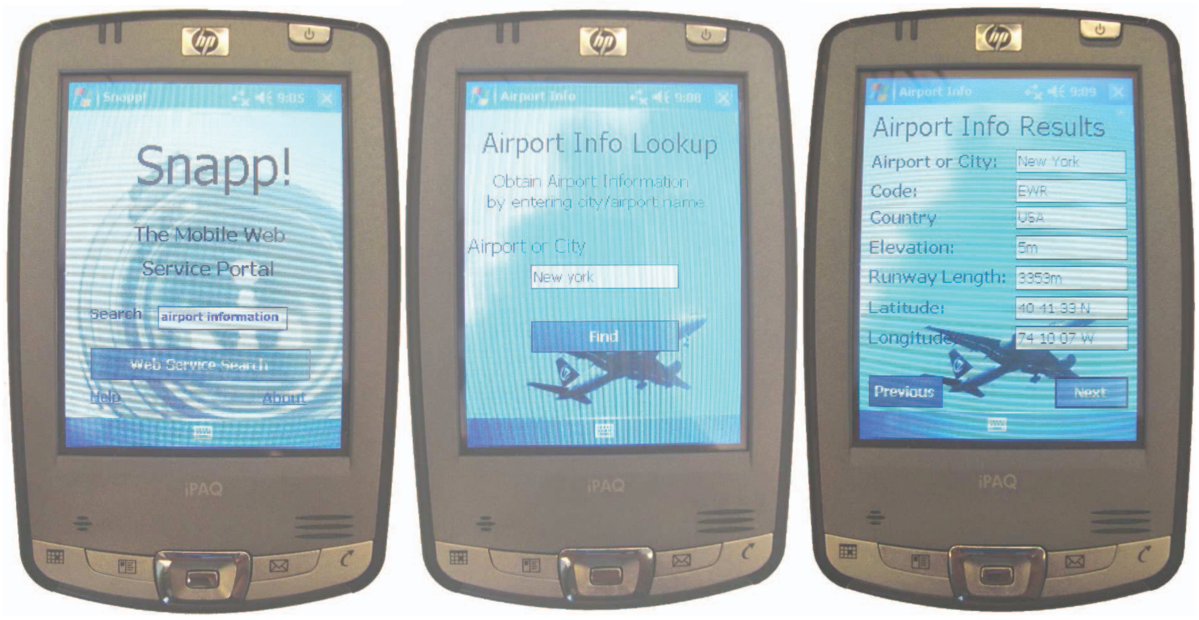


Figure 7. Class diagram that shows the implementation of the application

The developed mobile application was called SNAPP, which stands for Service Navigator at Palm Proximity. Once the user starts SNAPP, the initial form is launched to display a GUI comprising a textbox into which the user can enter the phrase that represents the desired service functionality. Figure 8 shows a case of a user wanting to find an airport in a city. The program determined that the best Web method is "getAirportInformationByCityOrAirportName" found in the published "airport" Web service. The method takes one input parameter: an airport name or a city name. As shown in the middle image, we display the description supplied with the Web method to help the user interpret the purpose and use of the input parameters correctly. Additionally, to enrich the dynamic GUI rendering, a library of over a 100 representative images were downloaded and stored on the MIM server and then linked with themes or topics that could be associated with the requested services. When a proxy is transferred to the mobile device, the corresponding image, if found, is sent along with it.



Search: airport information

Obtain Airport Information by entering city/airport name

Airport or City: New york

Airport or City: New York
Code: EWR
Country: USA
Elevation: 5m
Runway Length: 3353m
Latitude: 40 41 33 N
Longitude: 74 10 07 W

Figure 8. Application screenshots: user phrase (left), input parameter (middle), results (right)

6. EXPERIMENTAL EVALUATION

In this section we focus on the device battery power saving feature of the system and also study its scalability since it is a concern with any server-based architecture. We also present measurement results that indicate the user wait times under varying conditions.

6.1 Battery Energy Savings

To measure the energy consumption of the mobile device, we use a technique suggested in [23] to compute the power drain by measuring the voltage drop across a known resistor on the line between the power supply and the device. Figure 9 illustrates the setup. It consists of an Agilent DSO3062A oscilloscope connected to a PC that runs data acquisition software for capturing instantaneous battery voltages. After connecting the battery to the Pocket PC externally, experiments revealed that two of the seven pins connecting the battery showed non-negligible current, and supply power to the processor, memory, and Bluetooth and WLAN communication interfaces. Two 0.375 ohm resistances were placed in series between the Pocket PC and the battery to measure the voltage drop using the oscilloscope that feeds the data to the desktop, which computes the current by dividing the measured voltage by the resistance, and then the power by multiplying this current by the battery's voltage (3.7 V).

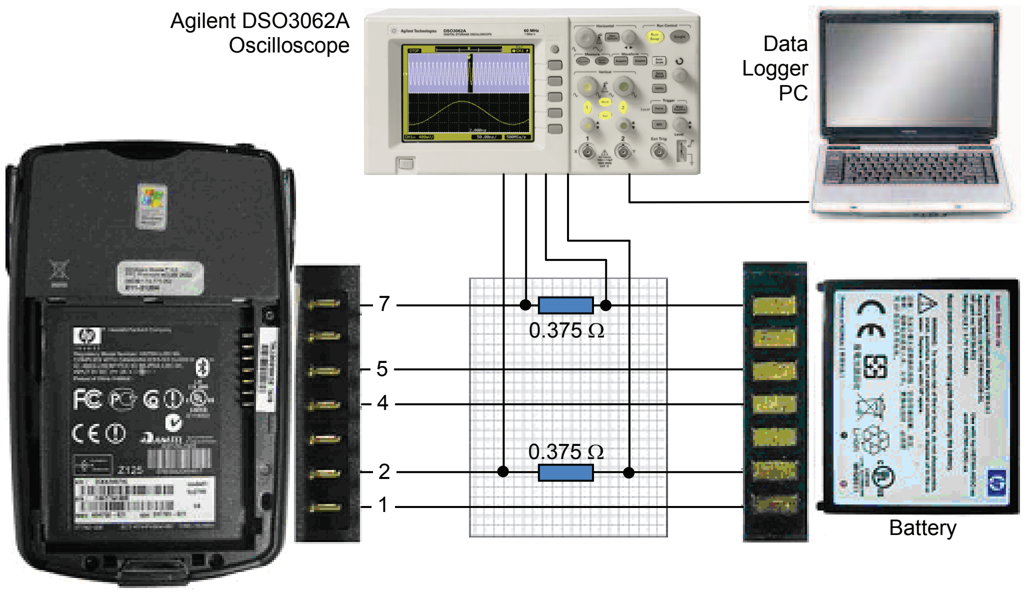


Figure 9. Diagram of the power measurement setup

In the experiments, the idle power, processing power, and WLAN communication interface power were isolated. Additionally, the power consumed by transmission and reception were also differentiated. To obtain the average power value consumed during the execution of a particular process, the instantaneous powers were averaged over its duration, which was determined by modifying the code to record the times when the process starts and when it finishes. Getting the actual energy spent in Joules was simply a matter of multiplying the average power by the process duration. Figure 10 shows a sample of the captured waveforms which yielded the average power values. The left graph shows the idle power which rises for few hundred milliseconds every now and then in a random way.

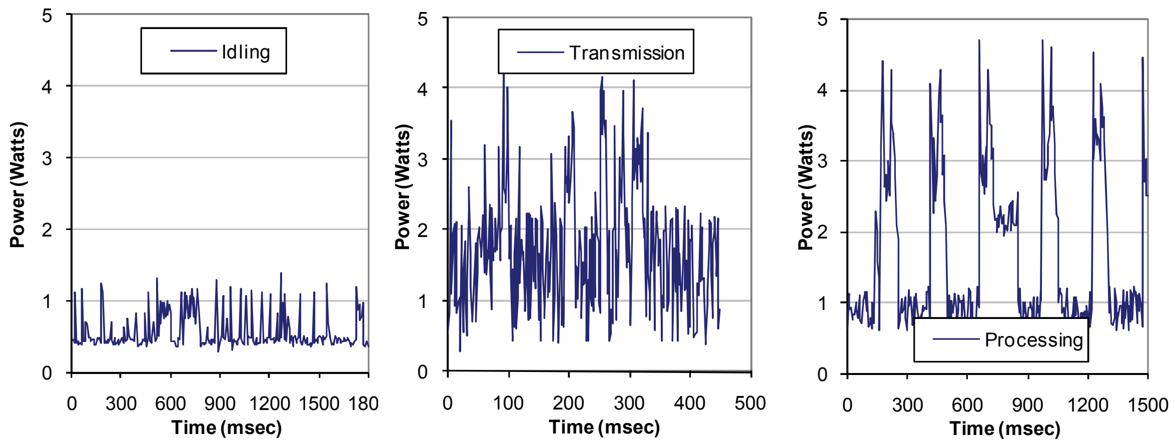


Figure 10. Sample power measurements

The repetitive tasks, which run on the MIM server and were described in Section 3, are: 1) performing first round of string matching, 2) communicating with the Web servers to get the WSDL files, 3) receiving the WSDL files, 4) applying a second round of string matching, and finally 5) generating the source code and compiling the proxy. As we mentioned earlier, our objective is to provide an estimate of the power *savings* that the mobile device achieves by delegating the above tasks to the MIM server. To determine these savings, we measured the consumed battery energy by running on the Pocket PC similar and mock-up tasks. For the first four tasks, reasonably-close functions were developed and deployed, while for the last one, a long loop that involved calling string manipulation and mathematical functions was implemented to simulate the processing required to generate the source file and compiling the proxy. The length of the loop was determined by measuring the time it took the MIM server to generate and compile the source code from an 18 KB WSDL file. A list of 180 service descriptions and associated WSDL files were stored on a server on the LAN to simulate the Web servers. The purpose of this setup was to simplify and speed up the experiments and to localize the delays (used in computing the energy) to the components of the architecture and minimize the variability associated with communicating over the Internet. As part of the setup for the experiments described below, 180 strings (sets of keywords) were developed to respectively match the service and Web method descriptions in the associated WSDL files.

These strings were used to automate the experiments and eliminate the need for user intervention. Moreover, the clock of the used Pocket PC was synchronized with that of the PC that connected to the oscilloscope and computed the power. This was done to accurately associate the measured values on the PC with the processes running on the device.

In the first experiment, the application was made to process all the search strings in a random order. The recorded task durations and mean power were then averaged over all 180 runs. The obtained values are shown in Table 1, where the last row describes the idle power, which is consumed by the PDAs even when no processing seemed to be involved. This power is used to refresh the 32 MB of RAM and the LCD screen, and process some kernel events. We measured the power consumption in idle mode, and found it to be 1694 mW with the Bluetooth and WLAN interfaces on and Power Saving (PS) mode off, and 563 mW with the Bluetooth, WLAN, and PS on. All measurements were taken using the second mode along with the screen backlight off. In this regard, it should be noted that the average power values shown in the table all include this "power floor" value.

Task Description	**Duration (sec)**	**Avg. Power (W)**
Generating short list of service descriptions	0.54	2.08
Transmitting requests for WSDL files	0.45	2.8
Receiving WSDL files	1.36	1.44
Identifying matching Web method	0.68	2.14
Generating client-side proxy	0.85	2.26
Idling between tasks (cumulative)	6.14	0.56

Table 1. Average task durations and measured powers

To shed more light on the device battery power saving feature of the MIM server setup, an additional experiment was conducted to study the request rate's effect on the amount of energy saved since it affects the processing duration. The request rate was varied between one request every 15 seconds and one every 4 minutes. In this regard, it should be noted that the high end of this range is equivalent to 120 users connecting to the MIM server, each submitting a request every half hour, on average. The setup of the experiment is as follows: for each of the 9 used request rates, the mobile client application submitted a request at the end of the corresponding time period by randomly selecting one of 180 search strings. The application was programmed to submit requests over a period of one hour, and as a result, the number of total requests per run ranged between 240 and 15. To account for the effect of caching the WSDL files on the MIM server, these files were not saved on the mobile device, but instead, a simple data structure was developed to store the list of files that were downloaded (after clearing it at the start of each run). A cache hit is then simulated by skipping the download process of the WSDL file if its name is found in the data structure. Regarding the random selection of search strings, two selection patterns were used: uniform and strict Zipf (i.e., for θ values equal to 0 and to 1, respectively). The energy savings are presented in the left graph of Figure 11 which relates greater savings to higher request rates. This can be attributed mostly to the increase in cache hits, especially when the Zipf access pattern is used (i.e., when the requested Web services are not equally popular).

To appreciate the significance of the savings, we derived approximate values for the additional requests (Web method invocations) that a device can make due to the deployment of the proposed architecture. For this, an estimate of the energy consumed by the device was computed while the user is interacting with the application to enter the search string and supply the parameters' values. The average measured power was 1.69 Watts while the average cumulative duration was around 25 seconds. With this data, the average number of requests that could be made by the application on a single battery charge was calculated with and without the developed architecture. This was done by dividing the capacity of the HP iPAQ hx2790 Pocket PC (specified at 1440 mA-hours, which is equal to 19,181 Joules when considering the battery voltage of 3.7 Volts) by the estimated total energy per request. The difference is a rough estimate of the additional number of requests that could be processed by the application on a single battery charge as a result of using our architecture when compared to having the device do all the work. In this analysis, we assumed for convenience the device is not running other applications, which is most likely untrue. Hence, we can think of the obtained values that are also presented in the right graph of Figure 11 as best case scenarios.

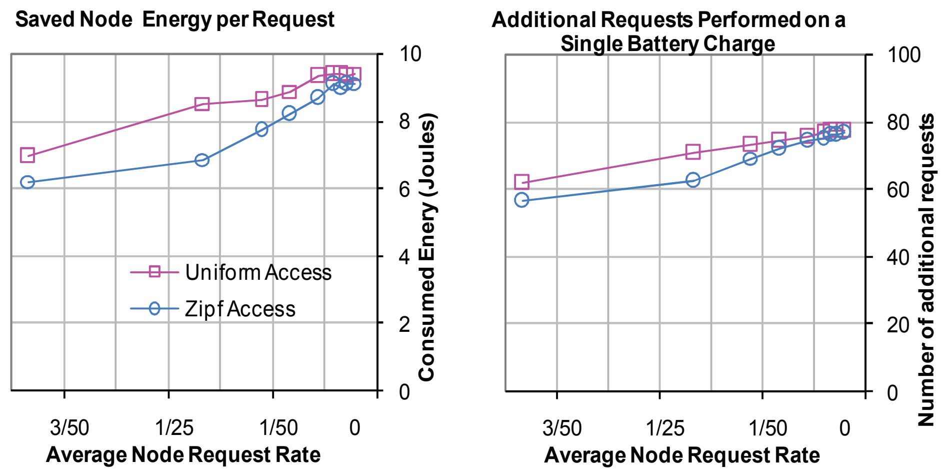


Figure 11. Saved energy (left), and additional requests per charge (right)

At the end of this subsection, we should note that the average request response time follows the same trends in the left graph (since the energy is the average power times the duration), while pointing out that the average duration across all request rate values is 7.3 seconds in the case of uniform access, and 6.9 seconds in the case of strict Zipf access.

6.2 Scalability

The MIM server provides Web method discovery and proxy building services to mobile clients, thereafter allowing them to interact *directly* with the Web servers. Moreover, the MIM server caches the WSDL files of the requested services so they can be served from the cache when requested in the future by other devices. This reduces the load on the server and allows it to serve more users. To evaluate the scalability of the architecture, we conducted an experiment which simulated the clients using threads running on another computer that we refer to as the *client computer*. This computer was an NEC i Select M5410 laptop, with Pentium M processor and 1 GB RAM. Hence, each mobile device client was simulated by a thread on the client computer, which allowed for easily increasing the load on the MIM server by increasing the number of concurrent threads that submit search phrases. For each request it receives from the client computer, the MIM server spawns a thread and runs in it the code that actually downloads the WSDL files and identifies the Web method, generates the proxy and sends it to the client computer by FTP, and finally returns to the client the name of the method and that of the Web service it belongs to.

In all the experiments described below, caching of WSDL files was not simulated, and therefore, the presented results correspond to worst case scenarios. The sizes of the downloaded and processed WSDL files varied between 4KB and 20KB. The search strings used in the experiments were restricted so as to result in downloading WSDL files from a collection of over 200 files which were determined a priori and which included the 180 files mentioned earlier. Within each set, the number of simultaneous requests that were sent from the client computer to the MIM server was increased until the server started dropping requests. Moreover, every single experiment was run twice and the average was taken. The measured values that we report are the total communication and processing times. The former included mainly the total time it took to fetch the WSDL files from actual Web servers on the Internet, as the communication delays with the client computer were negligible.

The results of the experiments are shown in the left two graphs of Figure 12, where the left-most graph shows the total communication delays while the middle one shows the total processing times. As we have explained earlier, the generated requests on the client computer were virtually sent concurrently to the MIM server by the threads. The MIM server, on the other hand, sent the replies (answers of the method invocations) sequentially to the client computer as they were completed, as illustrated in the left graph. The times took increasingly longer mostly because of queuing delays and processing of previous requests. The middle graph presents the total processing times that correspond to the results in the left graph. They initially increase as requests wait for their turns to be processed, but seemingly because of multi-processing by the CPU, the remaining requests get almost equal shares of processing.

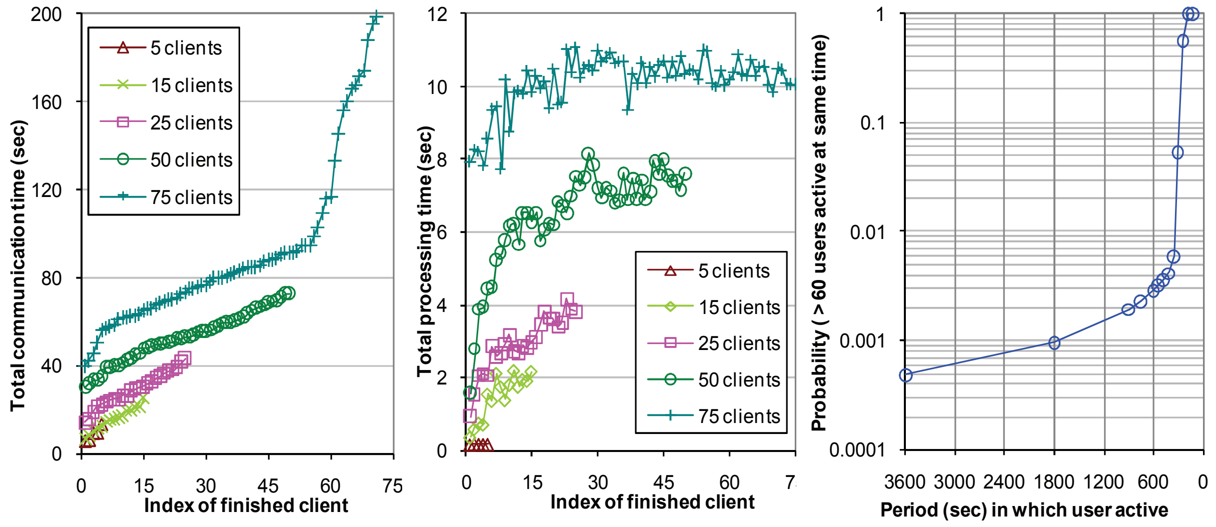


Figure 12. Scalability results: individual communication (left) and processing (right) times

The left graph shows that the MIM server was able to serve up to 75 simultaneous requests, but upon examining the graph closely, we notice that a threshold for an acceptable performance may be derived, and it corresponds to about 60 simultaneous requests. However, by using a similar argument as that made in Section 4.2, the probability of more than 60 concurrent requests arriving at the server will be very low, even when the number of potential users is large. The right graph in Figure 12 illustrates this probability in the case of 2500 total “subscribers” accessing the server once within periods that are less than 1 hour. With practical access rates (no more than once every ½ hour), the probability in this case will be less than 0.1%, implying that the MIM server will be able to offer acceptable performance.

7. DISCUSSION

In this section we provide a qualitative assessment of the characteristics of the proposed system and analyze its suitability to emerging development frameworks and device platforms.

7.1 Effectiveness

We define effectiveness as the ability of the application to give the user the needed service. For a sample of search phrases, the upper part of Table 2 indicates the matched Web method and the service exposing that method. It shows that the requests and responses are highly correlated, which reflects the effectiveness of the search algorithm. The system gives back desired results most of the time, but sometimes the invoked method does not meet the user's requirements. This usually occurs when the user enters a very general search phrase.

Input phrase	*Shakespeare*	*Weather in US*	*Weather*	*Zip code*	*Get zip code*	*Stock prices exchange*	*Sagsagsa*	*Currency exchange*	*Driving directions*	*Phone number verification*	*IP address location*	*Data encryption*	*Remove Profanity*	*Send SMS*
Discovered Web service	Shakespeare	WeatherForecast	"Input phrase is too general"	"Input phrase is too general"	WeatherForecast	Stock Quote	"No service/Web method found"	DOTS Currency Exchange	Mapquest Driving Directions	Phone Number Verification service	DOTSGeoPinPoint	TWS-DE	CDYNE Profanity Filter – FREE	SendSMSWorld
Chosen Web method	GetSpeech	GetWeatherByZipCode			GetWeatherByZipCode	GetQuote		ConvertCurrency	SimpleRoute	PhoneVerify	GetLocationByIP	Encrypt	ProfanityFilter	sendSMS
Task1	0.8	0.5	0.8	1.1	0.6	0.5	0.9	0.6	0.6	0.7	0.6	0.6	0.8	0.7
Task2	3.7	5.3	5.4	5.6	4.9	4.6	6.0	5.2	4.6	6.2	4.9	4.8	5.3	6.0
Task3	5.31	7.60			6.43	8.68		3.91	4.32	3.92	4.04	4.80	5.20	7.80
Legend:	Task1 = Generating short list of service descriptions Task2 = Getting the WSDL files and identifying Web method Task3 = Generation and loading proxy, dynamic GUI, and method invocation													

Table 2. Effectiveness as inferred from the top rows (times are shown for reference)

7.2 Speed

After the server application starts, and then periodically later, it downloads available service summaries from UDDI registries. The average time it took to download and store all summaries is close to 8 seconds, but this delay corresponds to an offline step that does not affect the user wait time. The response time of the user query depends on how busy the MIM server is, as was elaborated in the Scalability section, and ranges between 5 seconds (when the server is processing a single request) and 100 seconds (when the server is processing 75 requests). This time excludes the time taken by the user to type in the values of the input parameters. Since the user "interferes" by supplying values to the input parameters almost half-way into the process, the perceived wait time can be half of what it actually is.

7.3 Security

The MIM server does not cache content (Web service responses, or user supplied data). This eases the user's privacy concerns about data exchanged with the Web server, and about tracking his activities. On the other hand, encryption and server authentication can be used to protect the MIM server records and keep eavesdroppers from learning about the interests of users. Moreover, to protect the proxy files sent from the MIM server to the mobile device, many traditional mechanisms for securing the communication sessions between the wireless devices and the server can be used. Nowadays, practically all Secure Socket Layer (SSL) certificate providers support mobile devices. An example is GeoTrust [15], which offers SSL certificates that enable secure connections to enterprise servers, like our MIM server. There is even support for very-constrained devices to communicate securely with servers. An example is the Kilobyte SSL (KSSL) by Sun Microsystems which is a small-footprint, client-side-only implementation of SSL v3.0 for handheld devices [20]. This technology accounts for weaker CPUs, and for network latency, low bandwidth, and intermittent connectivity.

7.4 Suitability to Recent Platforms

In this section we consider the applicability of our proposed approach to two recent mobile device platforms, namely Google's Android and Apple's iPhone. Android does not have a built-in feature for consuming SOAP-based Web services, but a third-party component, such as *ksoap2*, can be installed on the device to provide this capability. Such a component will enable an Android client device to take part of our proposed architecture and access offered Web services through MIM-server-generated Java proxy classes. In the case of the iPhone, the *Core Services* framework found in the *Cocoa Touch* iPhone SDK can be used to access Web services. More specifically, the `WSMakeStubs` utility can be called on the MIM server to generate proxy classes that can be sent to iPhone clients.

7.5. Adaptability to REST

The major vendors continue to build the core Web services stack around SOAP, but REST [19] is preferred by certain groups. Recently proposed systems for publishing REST Web service descriptions, such as [35], and evolving technologies, like WSDL 2.0 [27], make migration of our design to the REST style quite possible. Specifically, a hosting environment, named SOAlive, was proposed in [35] for describing and deploying REST Web services. We believe that this and similar efforts will soon lead to standardized models for publishing REST Web service descriptions in a manner that is analogous to UDDI registries. Second, the new WSDL 2.0 standard, which was designed with REST Web services in mind, includes the semantics for describing such services. Therefore, to support RESTful services in parallel with RESTless services, which is the likely future scenario, the MIM server can access a system like SOAlive to download REST service descriptions and use them for short-listing services based on the user request, and utilize the WSDL 2.0 file to identify the desired REST service. Also similar to SOAP services, this WSDL 2.0 file can be used by a tool, such as Axis 2 from Apache, to generate a proxy class [27] which can be used by the mobile device in the same manner as that generated from the SOAP service WSDL file. The only difference is that the proxy class generated from the REST WSDL will use the HTTP libraries, while that generated from the SOAP libraries will use the SOAP libraries. This makes the MIM server a common interface for both the SOAP and REST web services paradigms.

Our proposed solution compiles the proxy code generated from the WSDL file at the MIM server and sends it to the mobile device. An alternative solution is to have the device directly interpret the WSDL file, which, after all, is designed to be easy to parse and interpret by a client in order to understand how to call the service. This is certainly a valid observation, but to recognize the advantages of the proposed solution, we provide in the following a detailed comparison between the two models. In the first one, the MIM server offloads most of the processing from the mobile device and saves it processing and communication overhead. In the second model, the mobile device has to contact the UDDI registries itself, fetch the descriptions of available web services, compile a short list, fetch and interpret the WSDL files, and then select the most suitable service and method to use. Table 3 compares the two models, while referring to them as the "client-stub execution" and the "wsdl interpretation" approaches, respectively. As illustrated, the proposed approach offers much more advantages, especially when considering execution overhead and bandwidth efficiency.

Execution overhead	
Client-Stub Execution	Minimal overhead processing is required, once the proxy class is ready, it can be used directly after reflecting its methods.
WSDL Interpretation	The short-listed WSDL files must be parsed and interpreted at run time. Moreover, there is a need to do the two stage comparison to find the appropriate WSDL file to use. However, online service finders might be utilized to find a candidate WSDL file, without any comparison stage on the mobile device. Although online service finders implies two or three stages of HTML file parsing to get the WSDL file URL.
Comparison	The overhead from the client-stub execution approach is limited to the method reflection. However, in the case of WSDL interpretation, the overhead consists of processing a high number of descriptions to get candidate WSDL files, and then parsing them to get the final WSDL file before interpreting its methods. Moreover, if an online service finder is used, the overhead execution involves parsing several HTML files, and three WSDL files. Clearly, using client-stub approach has a less execution overhead when compared to the WSDL interpretation approach in the context of a mobile device.
Software Engineering Practices	
Client-Stub Execution	Using existing libraries that process SOAP messages at the mobile device and using the proxy class to interface to them.
WSDL Interpretation	Either use the interpreted methods to interface to the existing SOAP libraries (generate the proxy on the mobile device, which is infeasible), or develop new SOAP libraries that can be used to process incoming and outgoing SOAP messages
Comparison	It is obviously a better software practice to use existing libraries that are optimized to the device environment and the operating system. In fact, WSDL interpretation methods were used mainly to bridge a gap till proxy class methods were implemented. This was the case in J2ME, for example, when JSR-172 was released in 2004.
Bandwidth efficiency	
Client-Stub Execution	The client's interaction with the network is limited to exchanging SOAP messages and receipt of the proxy class. The MIM server performs all the remaining interactions on behalf of the mobile device.
WSDL Interpretation	The client has to search for matching web service, which involves exchanging data with online web services finders, then interacting with servers to download the WSDL files, before interpreting them. Another alternative is to download all the services' description instead of doing the online search, but for a mobile device communicating over a wireless link, this is prohibitively costly in terms of wireless bandwidth consumption and mobile device memory utilization.
Comparison	The proposed approach involves much less interactions with the external network, knowing that the MIM server caches the downloaded service descriptions. The bottleneck for bandwidth efficiency is the wireless link between the mobile and the MIM server. In the proposed approach, the extra traffic incurred is due to sending the proxy from the MIM to the mobile. However, in the WSDL interpretation case, the mobile has to first search online for the service, get the search results and then download the WSDL files for the top matching services. The mobile will then have to access the service customized pages to get the WSDL links. Assuming 3 matching services, the traffic will amount to $3\times(200+18)\approx 650$KB (experiments conducted on webservices.seekda.com showed that the online search page is around 200 KB, while [25] reports the average WSDL file size to be 18 KB).

	On the other hand, in our approach, the compiled proxy has an average size of 100KB which is much less than the 650 KB needed in the WSDL parsing approach.
Code Complexity	
Client-Stub Execution	Requires developing code that utilizes reflection to call web services methods dynamically.
WSDL Interpretation	Requires developing code to parse and interpret WSDL files (no specific WSDL interpreters for mobile devices are available). Additional code complexity is incurred in the two possible options: 1. If the existing mobile device SOAP libraries are used, then interfacing code (proxy generation!) needs to be developed. 2. 2. Else, new libraries need to be developed, or third party libraries are to be used
Comparison	It is quite obvious that the proposed approach is simpler and requires no third party libraries, just native libraries developed by the vendor or operating system developers.
Client Code Updates	
Client-Stub Execution	No third party libraries are needed, updates occur automatically as part of the operating system updates.
WSDL Interpretation	More code is placed on the mobile client. This code is the developed application along with the utilized third party libraries.
Comparison	The proposed approach does not require additional client updates, as it does not utilize third party libraries, in contrast to the WSDL interpretation approach.
Flexibility	
Client-Stub Execution	We require the development of an application that performs minimal processing on the mobile device, and makes use of the existing functionalities on the mobile device. Moreover, proxy generation is done using the vendor supplied tool (wsdl.exe as an example).
WSDL Interpretation	The client code is dependent on the specific platform, but more functionality needs to be implemented using each platform, as more processing is needed on the mobile device. In addition, third party libraries specific to each platform might be needed.
Comparison	The proposed approach is more flexible as it makes use of existing functionalities. Furthermore, as mentioned above, it requires less client updates as compared to the WSDL interpretation method.

Table 3. Comparison of proxy class generation and direct WSDL file interpretation

8. CONCLUSION AND FUTURE WORK

The presented architecture makes it possible for mobile device users to dynamically invoke web service methods that meet their needs. The implemented solution overcomes technical limitations, and also saves device battery power, thus extending its participation in the wireless network. The scalability study can be used to decide on deployment of MIM servers in the network: given the capacity of the server, the number and distribution of MIM servers can be determined, knowing the cumulative expected request rate from users.

Our design provides Web service discovery services to personal applications running on mobile devices, where individual services can be used to extend the functionality of such applications. However, nothing precludes these applications from accessing composite Web services that perform computationally-intensive tasks, as in bioinformatics [6], data mining [38], and multimedia processing [16]. But, since Web service entities are usually autonomous and heterogeneous, how to connect and coordinate them is a challenging task that is not suited for mobile devices. As a future work, the MIM Server can be programmed with the intelligence to identify a set of services whose collective functionality can serve the user's request. In fact, the MIM Server is well-suited to coordinate the functions of such services and provide an interface to the mobile device which is consistent with the current design.

ACKNOWLEDGEMENT

This research was funded by the Lebanese National Council for Scientific Research (LNCSR). The authors would like to acknowledge the valuable comments and technical opinions of Mr. Habib Haddad and Dr. Imad Jureidini, the founders of Yamli.com.

REFERENCES

[1] C. Aggarwal, J. Wolf, and P. Yu, "Caching on the World Wide Web," *IEEE Transactions on Knowledge and Data Engineering*, v. 11, n. 1, 1999, pp. 94-107.
[2] R. Boyer and J. Moore**,** A fast string searching algorithm, *Communications of the ACM*. v. 20, pp. 762-772, 1977.
[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: evidence and implications," *Proc. IEEE INFOCOM*, 1999, pp. 126-134.
[4] J. Cao, M. Andersson, C. Nyberg and M. Kihl, "Web server performance modeling using an m/g/1/k* ps queue," *10th Int'l Conf.* in *Telecommunications, 2003. ICT 2003*.
[5] Celimaris Vega Citrix Consulting, MetaFrame XP Oracle 11i Application Scalability Analysis, 2002, http://support.citrix.com/article/CTX101887
[6] A. Chakravarti, G. Baumgartner, and M. Lauria, "The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network," *IEEE Trans. Systems, Man, and Cybernetics*, v. 35, n. 3, pp. 373-384, 2005.
[7] M. Chatti, S. Srirama, D. Kensche, and Y. Cao, "Mobile Web Services for Collaborative Learning,"*IEEE Int'l Workshop on Wireless Mobile and Ubiquitous Technology in Education*, Nov 2006, pp.129-133.
[8] CodePlex, ProxyFactory Home Page, available at www.codeplex.com/ProxyFactory
[9] R. Costello, Building Web Services the REST Way, available at http://www.xfront.com/REST-Web-Services.html
[10] G. Dattatreya, *Performance Analysis of Queuing and Computer Networks*, Chapman & Hall/Crc Computer & Information Science Series, 2008.
[11] I. Duda, M. Aleksy, T. Butter, "Architectures for Mobile Device Integration into Service-Oriented Architectures", *Int'l Conf. on Mobile Business* (*ICMB'05*), 2005.
[12] R. Fielding and R. Taylor, "Principled Design of the Modern Web Architecture", *ACM Trans. on Internet Technology*, Vol. 2, No. 2, pp. 115–150, 2002.
[13] J. Flinn and M. Satyanarayanan, "PowerScope: A tool for profiling the energy usage of mobile applications", *2nd IEEE Workshop on Mobile Computer Systems and Applications*, New Orleans, Louisiana, pp. 2, 1999.
[14] G. Gehlen and L. Pham "Mobile Web services for peer-to-peer applications,"*IEEE Conf. on Consumer Communications and Networking*, Jan 2005, pp. 427-433.
[15] GeoTrust Corp, http://www.geotrust.com/enterprise-ssl-certificates/georoot/
[16] X. Gu and K. Nahrstedt, "On Composing Stream Applications in Peer-to-Peer Environments,"

IEEE Trans. Parallel and Distributed Systems, v. 17, n. 8, pp. 824-837, Aug. 2006.

[17] V. Gupta, "Finding the optimal quantum size: Sensitivity analysis of the M/G/1 round-robin queue," *ACM SIGMETRICS Performance Evaluation Review,* v. 36, pp. 104-106, 2008.

[18] A. Halteren, P. Pawar, "Mobile Service Platform: A Middleware for Nomadic Mobile Service Provisioning", *WIMOB 2006*, Montreal, Canada.

[19] R. Heffner, SOAP versus REST: A Comparison, available at www.forrester.com/research/document/excerpt/0,7211,35361,00.html

[20] C. King, "Securing the Wireless Internet Using "Kilobyte" SSL", available at http://www.sun.com/bigadmin/content/developer/howtos/kssl.html

[21] L. Kleinrock, "Time-shared systems: A theoretical treatment," *Journal of the ACM (JACM),* v. 14, pp. 242-261, 1967.

[22] J. Kurose and K. Ross, "Computer Networks and the Internet," in *Computer Networking: A Top-Down Approach*, 4th ed., Pearson Education International, 2008.

[23] R. Lee and R. Nathuji, "Power and performance analysis of PDA architectures", Techical Report, MIT, Dec., 2000, available at http://www.cag.lcs.mit.edu/6.893-f2000/project/lee_final.pdf.

[24] L. Li, M. Li, X. Cui, "The Study on Mobile Phone-Oriented Application Integration Technology of Web Services", Lecture Notes in Computer Science, v. 3032, Springer Berlin Heidelberg, April 2004.

[25] Y. Li, Y. Liu, L. Zhang, G. Li, B. Xie and J. Sun, "An Exploratory Study of Web Services on the Internet," *IEEE International Conference on Web Services (ICWS 2007)*, pp.380-387, 2007.

[26] Y. Ling, T. Mullen, X. Lin, Analysis of optimal thread pool size, *ACM SIGOPS Operating Systems Review*, 2000, v. 34, n. 2, pp. 42-55.

[27] L. Mandel, Describe REST Web services with WSDL 2.0, Technical guide, IBM Co., May, 2008. Available at http://www.ibm.com/developerworks/webservices/library/ws-restwsdl/

[28] Oracle Corporation, Oracle9i Application Server Oracle HTTP Server powered by Apache Performance Guide Release, Part Number A86676-02, 2001 http://download.oracle.com/docs/cd/A95434_01/a86676/sizing.htm#1032856

[29] O. Rendón, F. Pabón, M. Vargas, J. Guaca, Architectures for Web services access from mobile devices, *3rd Latin American Web Congress* (*LA-WEB 2005*), pp. 93 – 97, 2006.

[30] R. Steele, K. Khankan, and T. Dillon, "Mobile web services discovery and invocation through auto-generation of abstract multimodal interface", *Int'l Conf. on Information Technology: Coding and Computing (ITCC 2005)*, Vol. 2, pp. 35-41, 2005.

[31] Sun Microsystems, "JINI Technology Surrogate Architecture Specification", http://surrogate.JINI.org/sa.pdf, Oct. 2003.

[32] C. Weyer, DynWsLib tutorial, available at www.thinktecture.com/resources/software/DynWsLib/default.html

[33] E. Sánchez-Nielsen, S. Martín-Ruiz and J. Rodríguez-Pedrianes, "Mobile and dynamic web services," *Emerging Web Services Technology,* pp. 117-133, 2007.

[34] Q. Sheng, B. Benatallah, Z. Maamar, and A. Ngu, "Configurable Composition and Adaptive Provisioning of Web Services," *IEEE Trans. Services Computing*, v. 2, n. 1, pp. 34-49, 2009.

[35] I. Silva-Lepc, R. Subramanian, I. Rouvcllou, T. Mikalson, J. Diament, A. Iyengar, "SOAlive service catalog: a simplified approach to describing, discovering and composing situational enterprise services," *Int'l Conf. Service Oriented Computing, ICSOC 2008*, pp. 422-37, 2008.

[36] W3C, XForms – the next generation of Web forms, available at http://www.w3.org/markup/forms/, 2007

[37] A. Willig, "A Short Introduction to Queuing Theory," [online document] July 21, 1999, available at http://www.tkn.tu-berlin.de/curricula/ws0304/ue-kn/qt.pdf

[38] R. Wolff and A. Schuster, "Association Rule Mining in Peer-to-Peer Systems," IEEE Trans. Systems, Man, and Cybernetics, v. 34, n. 6, pp. 2426-2438, 2004.

[39] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed, "Deploying and Managing Web Services: Issues, Solutions, and Directions," The VLDB J., v. 17, n. 3, pp. 537-572, 2008.

[40] G. Zipf, *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.

Hassan Artail is a Professor at the American University of Beirut (AUB) where he is doing research in Internet and mobile computing, mobile ad hoc networks, and vehicle ad hoc networks. During the past six years, Dr. Artail has published over 85 papers in top conferences and reputable journals. He obtained his BS and MS degrees in Electrical Engineering with high distinction from the University of Detroit in 1985 and 1986, and a PhD in Electrical and Computer Engineering from Wayne State University in 1999. Before joining AUB in 2001, Dr. Artail was a system development supervisor at the Scientific Labs of DaimlerChrysler, where he worked for 11 years in the field of system development for vehicle testing applications.

Kassem Fawaz received the BE degree with high distinction in Computer and Communications Engineering from AUB in 2009. He is currently a graduate student at the Department of Electrical and Computer Engineering at AUB, where he is doing work in the areas of mobile computing and ad hoc networks. Kassem received the Distinguished Graduate Award upon graduation in 2009, and has to date published eight papers in Web systems and pervasive computing. He is an IEEE member.

Ali Ghandour earned the BE and ME degrees in Computer and Communication Engineering with distinction from AUB in 2008 and 2010, respectively, and he is currently a PhD student in the Department of Electrical and Computer Engineering at AUB. His research interests include web systems, vehicular ad hoc networks, and cognitive networks. He worked with Ericsson as an intern during 2007, and has so far published 5 papers in the areas of cognitive networks and Web systems.