Dynamic service-based integration of mobile clusters in grids.

Stavros Isaiadis[1]
Vladimir Getov[1]
Ian Kelley[2]
Ian Taylor [2]

[1] Harrow School of Computer Science, University of Westminster
[2] School of Computer Science, Cardiff University

This is a reproduction of CoreGRID Technical Report Number TR-0094, 25 February 2008 and is reprinted here with permission.

The report is available on the CoreGRID website, at:

http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0094.pdf

Dynamic Service-Based Integration of Mobile Clusters in Grids

Stavros Isaiadis and Vladimir Getov
{S.Isaiadis, V.S.Getov}@wmin.ac.uk

Harrow School of Computer Science
University of Westminster, London, UK

Ian Kelley and Ian Taylor
{I.R.Kelley, Ian.J.Taylor}@cs.cardiff.ac.uk

School of Computer Science
Cardiff University, Cardiff, UK

CoreGRID Technical Report
Number TR-0094
25 February 2008

Institute on Grid Systems, Tools and Environments

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

Dynamic Service-Based Integration of Mobile Clusters in Grids

Stavros Isaiadis and Vladimir Getov
{S.Isaiadis, V.S.Getov}@wmin.ac.uk

Harrow School of Computer Science
University of Westminster, London, UK

Ian Kelley and Ian Taylor
{I.R.Kelley, Ian.J.Taylor}@cs.cardiff.ac.uk

School of Computer Science
Cardiff University, Cardiff, UK

CoreGRID TR-0094
25 February 2008

Abstract

The emergence of pervasive and mobile computing has drawn research attention to integrated mobile Grid systems. These new hybrid Grids consist of a typical SOA backbone extended to include mobile and small scale devices such as personal digital assistants, smart-phones, multimedia devices, and intelligent sensors. In a fully integrated model, mobile devices are able to act both as consumers and providers to open up a completely new range of very interesting possibilities in exploiting their mobile nature, unique functionality, and context awareness. However, in resource-limited environments, traditional SOA frameworks cannot easily be deployed since they assume a plethora of available device resources, and have a number of complex dependencies, thus rendering them unsuitable for resource-constrained devices. Therefore, a smaller and simpler server-side container with reduced requirements and dependencies is needed. The contribution of this paper is two-fold: first, we have designed a J2ME-compliant socket-based server-side container, and second, we have demonstrated how an aggregator framework enables such mobile services to be accessed using standard-based Web services in a high-level manner.

1 Introduction

The emergence of pervasive and mobile computing has drawn research attention to integrated mobile Grid systems. Such systems consist of a typical SOA (Service-Oriented Architecture) backbone extended to include mobile and small scale devices (Personal Digital Assistants, smart-phones, sensors and more). There are three hybrid system classes: first, mobile interfaces to Grid resources, where mobile devices are merely interfaces to functionality available in the Grid system, and do not contribute any services. Second, exploitation of raw resources (CPU, memory, storage) in small-scale devices, where focus is on distributing applications for parallel execution usually requiring the partitioning of the application into small independent tasks. However, we believe that the core competence of such devices is their flexibility, pervasiveness, and location awareness and not their (limited) raw resources. Third, exploitation of services in mobile and small-scale devices, where the focus is in supporting mobile services in a SOA system, while also enabling small-scale devices to contribute services.

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

The last model is the one that provides the most complete integration where mobile and micro-devices can be both consumers and providers of services. Such integration could open up a completely new range of very interesting possibilities in exploiting the mobile nature of these devices. SOA systems could extend to reach geographical areas where before it was not possible allowing, for example, an organization to better control its field operations personnel. The functional benefits are also significant as mobile devices increasingly offer unique functionality not found in traditional Grid nodes, such as location aware equipment, multimedia cameras, intelligent wireless sensors, Global Positioning Systems and more.

However, a hybrid Grid system presents a number of interesting challenges. In a limited environment, pure SOA frameworks cannot easily be deployed. For example, typical Web Service (WS) and Grid Service (GS) containers (such as Axis/Tomcat or the Globus Toolkit) assume a plethora of available device resources, and have a number of complex dependencies. These requirements render them unsuitable for resource-constrained devices, where a smaller and simpler container with reduced requirements and dependencies is needed. But even with the development of such container, it is unreasonable to assume that all limited devices will adopt the same approach. In dedicated and relatively static environments certain guidelines can be enforced and adopted so that all members comply with the same policies. But in a dynamic environment that aims at agile and opportunistic computing, flexibility and mobility, imposing such restrictions is undesired.

The contribution of this paper is two-fold: first, we present the design of a fully J2ME compliant (Java 2 Micro Edition) socket-based server-side container that enables the user to export Java classes available in the device (Section 3). Second, we demonstrate how an aggregator framework enables such mobile services to be accessed using standard-based Web services in a high-level manner (Section 4). An aggregator service acts as a proxy to a group of heterogeneous and mobile underlying server-side components (supporting RMI, WS, GS, plain sockets and more). Further, aggregators provide an abstraction layer that hides the dynamicity and heterogeneity of the grouped services, in order to ease programming and provide a simpler view to high-level clients. The paper is completed with an evaluation of the LSC framework that verifies its superiority in terms of performance and resource utilization (Section 5).

2 Related Work

Mobile OGSI.NET [3] is an implementation of an OGSI (Open Grid Services Infrastructure [4]) based Grid services container for mobile devices based on Microsoft's PocketPC. Mobile OGSI.NET, despite being one of the earliest efforts, has been abandoned due to its restrictive implied programming model and nature, and has become obsolete due to the emergence of new standards in the form of the Web Services Resource Framework (WSRF) [5].

A generic component-based platform has been presented in [6, 7] and has been part of research work conducted under CoreGRID [8]. Its design goals are the generality, reconfigurability, dynamic adaptation to system conditions and expandability of the platform. While this approach does not focus on small-scale devices, it offers a generic approach and an attractive model that could be adapted to suit the atypical environment of mobile hybrid Grid systems. At the time of writing, the platform remains at a design level.

WSPeer is a generic SOA-oriented toolkit based on the core Web services technology of SOAP that enables applications to perform the core SOA operations in an environment agnostic manner [9]. It has three bindings to different underlying middleware systems: the conventional HTTP and UDDI binding to the most-frequently used SOA stack; bindings to Peer-to-Peer middleware, such as JXTA and P2PS [10]; and a binding for constrained devices called WSKPeer. It also supports many WS-* specifications, including WS-Security, WS-Addressing, WS-Transfer and WS-RF. WSKPeer [11] (K for kilobytes) exposes a WSPeer restraint-device binding but focuses on the message abstraction, reducing all activity to asynchronous XML message exchanges. WSPeer is implemented using kXML and J2ME and can support service description parsing, including complex types, and sophisticated message exchanges that support notions of state and events, in particular WS-RF, despite the limited availability of processing power. WSKPeer provides an interesting approach to service orientation but it is at an early alpha stage of development and therefore premature for integration in the work presented here.

KSOAP [13] and kXML [12] have been successfully used in experimental settings to support mobile servers, SOAP engines, and XML parsing, installed on limited Java-based devices. One such implementation is the MicroServices framework [1] developed at the Monash University in Australia. This framework comprises of a stripped down mobile HTTP server, supported by a number of components to handle either HTTP or SOAP requests, thus allowing the deployment of Web services in resource-limited devices. The system allows only restricted Web services in terms of functionality, method complexity, and supported data types, while scalability is also reduced, with the number of simultaneous connected clients kept to low numbers to preserve resources.

Raccoon [2] is an effort to port Apache's popular httpd daemon into the Symbian-powered S60 Nokia phones. Raccoon encompasses a connectivity solution able to provide a mobile phone with a unique global URL accessible through HTTP from anywhere once online, thus enabling the concept of mobile Web sites (or ”mobsites”). A significant benefit from having a mobile Web server is the enabling foundation for developing and deploying Web services on the smart phone. This would greatly widen the applicability, and allow for more familiar, service-oriented communication with the mobile devices.

3 Lightweight Server-side Container (LSC)

The inherent limitations of mobile devices enforce certain restrictions regarding the nature of the hosted server-side components. First, while typically a service can handle complex operations, the nature and role of mobile devices in an agile, pervasive, and context-aware environment suggest otherwise. Such services should perform very well defined, simple, and short operations, optimized for resource-constrained devices. Further, the nature of the services must not be transient (like in the WS paradigm) as there is limited scalability and even a small number of client requests can quickly deplete available resources. Instead, permanent and stateful services must be employed, first to simplify service lifecycle management and reduce the cost of service initialization, second, to support state and context aware operations.

Several technical requirements were identified for the LSC design. First, the CPU, memory, and storage footprint of the LSC must be as low as possible. Further, in an effort to preserve battery energy for end-user operations, when the level of battery is low the LSC is automatically shut down and does not accept any new requests. Second, the prototype must be fully J2ME compliant to ensure a high degree of interoperability and compatibility with a wide range of mobile devices. Third, it must be autonomous with no software or technological dependencies to enable easy deployment and wide adoption. Finally, it should offer a user-friendly GUI to enable the easy exportation of server-side components.

The LSC is part of the Mobile Platform Agent (MPA), a small software component that accompanies the Virtual Clusters (VC) platform (more details in the next section), and is responsible for several non-functional properties such as monitoring, discovery, activation and more. The MPA encapsulates the LSC to provide the necessary GUI and high-level function management.

The LSC consists of three distinct components: the Server Socket Listener (SSL), the Server Request Handler (SRH), and the Service Registry (SR) (Fig. 1). The SSL is responsible for initializing, maintaining, and terminating the server socket in the mobile device, listening and receiving client requests on that socket, and forwarding them to the SRH. The SRH validates the TCP requests (against the required protocol format) and interacts with the SR in order to activate the relevant Java service and invoke the requested method. The LSC message protocol has the following format:

```
LSC_INV|<service URL>|<method name>|<serialized arguments>
```

for instance:

```
LSC_INV|lsc://192.168.1.3:5678/vclustercdc.ImageService|
processImage|1984592123<arg byte array>
```

All the service method arguments, as well as the return value if any, are serialized into plain byte arrays before TCP transmission. The SRH is then responsible for the de-serialization of the arguments before supplying them to the relevant Java service. Finally, the SR is a permanent storage facility that stores information relating to the local shared Java services, namely the Implementation and the Interface classes. This information is supplied by the service provider through the GUI, a screenshot of which is shown in Fig. 4.

At the other end (the VC proxy), a number of components perform the opposite operations than the LSC. The Service Request Translator (SRT) is responsible for transforming high-level method invocations into low-level LSC protocol requests, ready to be transmitted through the TCP socket. The SRT also serializes all method arguments into plain byte arrays, before forwarding them to the Socket Communicator (SC), which is responsible for the actual transmission. It has to be noted that developers would never directly make use of the LSC protocol but instead will only access LSC devices through the VC Proxy which knows how to ”talk” in LSC (see Section 4.1 for more details).

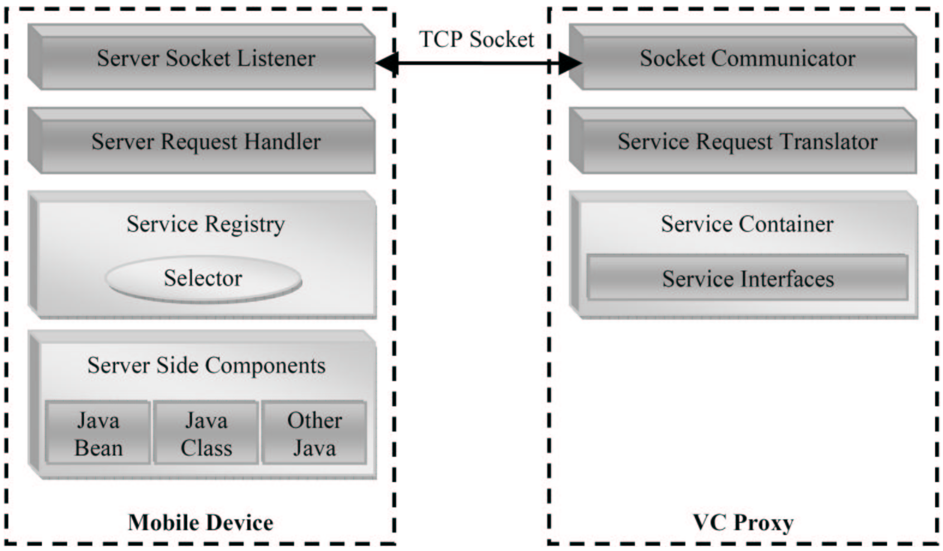


Figure 1: Architectural Overview of MPA/LSC and VC Proxy

4 Aggregator Services and LSC Framework

Aggregator services are at the core of the VC platform. The VC architecture focuses on providing the means to overcome the resource limitations, dynamicity, and mobility that humble the mobile domain. The main conceptual idea is to hide the big cardinality of the mobile domain by exposing all mobile devices as a single virtual entity, thus enabling a smooth integration and reduce the administrative and performance overhead. In order to present a group of devices as a single entity, a representative proxy node is required. Hence, the VC approach is a proxy-based, partially centralized design, where the Grid system is merely extended to include a single extra node: the proxy, which acts as the gateway for the underlying group of mobile devices [14, 15].

The main entity in the realization of a VC is the aggregator service (or simply ’Aggregator’). The Aggregator resides at the proxy node, and is responsible for a group of mobile devices that provide a similar resource or service (where similarity is determined by the implemented interfaces). Aggregators expose single, consistent, and permanent interface points to functionality available in the fabric service layer. Each Aggregator is generated and deployed on the fly, the first time that a particular type of service is made available for sharing in a VC -there is one Aggregator for each different type of service or resource. Fig. 2 depicts the VC architecture, which focuses primarily on providing the functionality of the proxy layer, which consists of the Aggregators, the core platform services that support various non-functional aspects, and a collective layer that enables coordinated and concurrent access to fabric services. Aside of these components, an API is also provided, to enable the development of sophisticated VC-aware hybrid Grid applications. Each main service group is associated with, and is responsible for, one or more of the main challenge areas: monitoring for controlling dynamicity, discovery for managing mobility, collective layer for efficient management of large numbers of dynamic nodes, and there is also a number of supporting service groups, including indexing, invocation and more.

4.1 Overview of Communication Models

The LSC on its own presents a number of interesting challenges. Despite its efficiency (see Section 5) and functionality, it is not an interoperable approach as it is based on a customized communication protocol format over TCP sockets. The low-level socket based communication model employed must be encapsulated to provide a more abstract development environment. The VC platform is used in this case, to decouple the high-level clients from the socket-dependant LSC services on mobile devices and enable high-level interactions. Communication at a conceptual level still flows between the client and the LSC-exposed services, but underneath, the VC proxy is the intermediary that translates the network-agnostic client requests into low-level LSC/TCP socket streams. Communication between the client and the VC proxy (in essence the relevant Aggregator that represents the specific type of service) is unmodified WS based HTTP/SOAP.

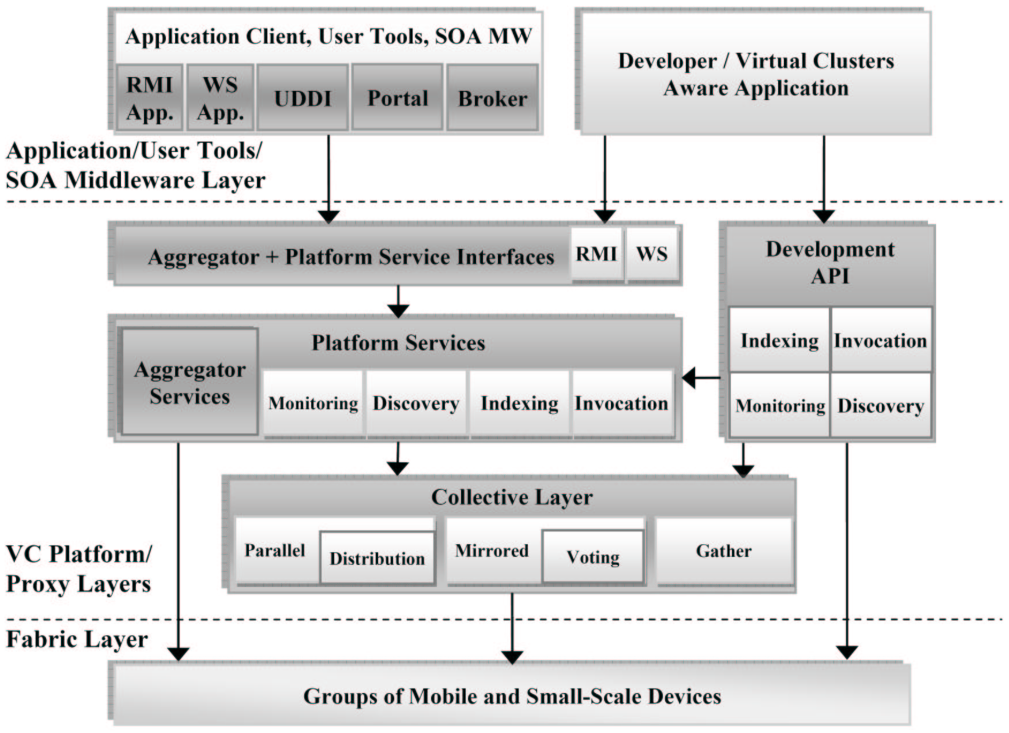

Application Client, User Tools, SOA MW
RMI App.
WS App.
UDDI
Portal
Broker
Developer / Virtual Clusters Aware Application
Application/User Tools/ SOA Middleware Layer
Aggregator + Platform Service Interfaces
RMI
WS
Development API
Indexing
Invocation
Monitoring
Discovery
Platform Services
Aggregator Services
Monitoring
Discovery
Indexing
Invocation
Collective Layer
Parallel
Distribution
Mirrored
Voting
Gather
VC Platform/ Proxy Layers
Fabric Layer
Groups of Mobile and Small-Scale Devices

Figure 2: High Level Overview Depicting the Most Important Components

The Aggregator then translates the request, and contacts the LSC through a TCP socket using the LSC communication protocol. The same socket is used to return the results once the execution on the mobile device finishes.

4.2 Interaction Patterns

Assuming that the mobile device is within the range of a VC, a complete exemplary scenario is presented here, from the local registration of the shared LSC services in the MPA, to the handling of the client request and the return of the results (Fig. 3). The example functionality includes a Java class that returns a Java image stored locally in the device, after scaling it to the specified size. A further assumption to simplify this scenario is that this specific type of Image service has already been aggregated. The example environment consists of a VC proxy (in our example 79.69.101.2) where an Image Service Aggregator has already been deployed in the Axis/Tomcat framework, under the relative URL: /axis/services/aggregators/ImageAggregator, to make up the full Aggregator endpoint address of:

```
http://79.69.101.2:8080/axis/services/aggregators/ImageAggregator.
```

A mobile device such as Sony Ericsson P990i (79.69.99.35), implementing the Image Service, registers and contributes to the VC.

1. **Registration of Local Services**. The service provider specifies which Java components to share. For each Java component, the server-side framework (WS, RMI, or LSC), the implementation and interface classes, and an optional human readable description must be specified (Fig. 4). Obviously, a relevant container must be already running, for instance, the MicroServices framework or Raccoon for WS, an RMI registry for RMI, or the LSC/TCP for plain Java-based services. All service information is first validated and then stored locally either as an XML file in the local file system, or using the Persistence J2ME API for very limited devices. In this example, the Image Gather Java application is registered as an LSC server-side component.During the local registration process, the current possibly dynamic IP address of the device is not important (the device can even be offline), while the LSC port defaults at 12345. The endpoint invocation address of the Image service thus becomes:

```
lsc://79.69.99.35:12345/vclustercdc.ImageApp.ImageServer
```

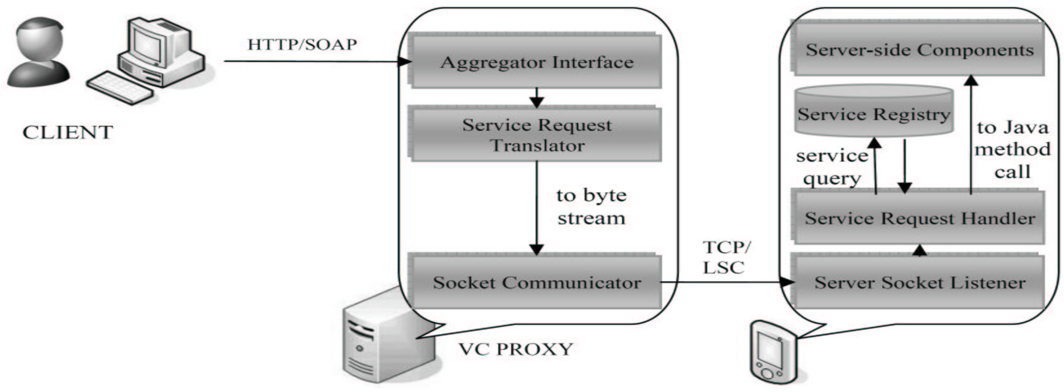


Figure 3: Invocation Interactions

2. **Initialization of the MPA**. When one or more services have been registered locally, the mobile user can initialize the MPA. MPA initialization involves discovering and binding to a nearby VC proxy, and automatic activation of the locally registered Java services. If this is the first time this MPA registers with a VC proxy, a new globally unique ID is generated and assigned to it for future activations, in the form of: 9f941ea6-4579-30b2-96c4-f3a6812e60ce. Since the Image service is hosted on the LSC, the latter is also initialized along with the MPA. Initialization of the LSC is a two-step process: first, the Service Registry is loaded from the permanent storage, and second, the Server Socket Listener creates a local server socket on the designated port (12345), and starts listening for service requests.

3. **Automatic Activation of Local Services**. Immediately after binding to a VC proxy, the MPA activates and thus contributes the Image service (and any other registered services). As already assumed, an Aggregator for the Image service is already deployed in the VC proxy; otherwise, a new Aggregator is generated to represent the Image interface. The Image service is bound to the relevant Aggregator and is now ready to use.

4. **Aggregator Method Invocation**. A potential client connects to the representative Aggregator and invokes the getImage method having no knowledge of the Virtual Cluster and the underlying aggregated services. The communication is facilitated with SOAP over HTTP in typical WS fashion. Programmatically an Aggregator method invocation facilitating the Axis Dynamic Service Invocation API could look like:

```
...
Call call = (new Service()).createCall();
call.setEndpointAddress(
"http://79.69.101.2:80/axis/services/aggregators/ImageAggregator");
Image img = (Image)call.invoke("getImage", new Object[] {640, 480});
...
```

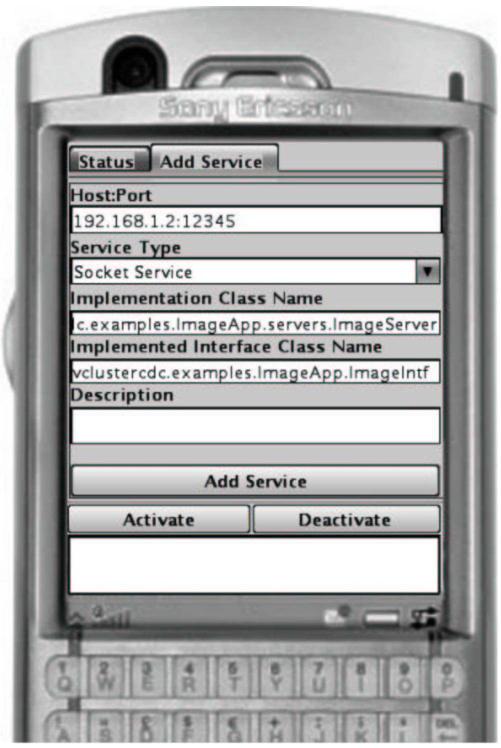
Alternatively, the client could use a portal framework, a web interface, or some other Grid submission software.

5. **Aggregator Processing**. The Aggregator acquires the mobile service's address and binding details from the VC platform component, and forwards the method invocation to the Socket Request Translator (SRL). The SRL serializes the method arguments into a plain byte array, and translates the request into the LSC protocol format, before forwarding it to the Socket Communicator. The method invocation translated in the LSC protocol format would look like:

```
LSC_INV|lsc://79.69.99.35:12345/vclustercdc.ImageApp.ImageServer
|getImage|3982746...<byte array of the parameters>
```

The Socket Communicator opens a client socket connection (in a random available port, for instance 7008) to the LSC server socket (which listens on 79.69.99.35:12345) on the mobile device, and streams the translated request to it.

6. **LSC processing**. The Server Socket Listener receives the TCP socket stream in LSC format, and forwards it to the Service Request Handler (SRH). The SRH validates the stream against the LSC protocol format, and, if valid, extracts the service name from the supplied URL, in this case: vclustercdc.ImageApp.ImageServer, and the method name. The SRH then consults the Service Registry to acquire the registered information for the specified Java service, in order to dynamically recreate the method arguments from



Host:Port
192.168.1.2:12345
Service Type
Socket Service
Implementation Class Name
c.examples.ImageApp.servers.ImageServer
Implemented Interface Class Name
vclustercdc.examples.ImageApp.ImageIntf
Description

Figure 4: The MPA GUI

the byte array. When the arguments have been recreated, a new instance of the service class is created, and the requested method is invoked. The result of the method invocation is then serialized back to a byte array, and forwarded to the client end of the socket, in this case 79.69.101.2:7008.

7. **Aggregator Returns**. When the TCP stream with the result reaches the Aggregator, the Socket Communicator closes the client socket connection (port 7008), and forwards the result to the SRT. The latter de-serializes the byte array into the respective Java type, and returns. The client acquires the result from the Aggregator in typical WS fashion.

8. **MPA and LSC can be shut down if further sharing is not desired**. The mobile end user can easily shut down the MPA if no further contribution to the VC is required. During shut down, the MPA de-activates the Image service by informing the Active Index component of the VC proxy. Finally, the LSC shuts down the server socket and frees up any allocated resources.

5 Evaluation

The evaluation of the LSC consists of two parts: performance or stress testing, and resource requirements testing. For the performance testing, three different devices where used as described in Table 1.

Table 1: Experimental Environment Nodes Specification

Type	Base Model	Memory	OS	Server-side framework
Laptop A	2.2GHz Intel-based	1GB	Win XP	Tomcat/Axis, RMI, LSC
Laptop B	1.6GHz AMD-based	512MB	Win XP	Tomcat/Axis, RMI, LSC
PDA	Sony P990i	128MB	Win Mobile	Microservices, RMI-OP, LSC

For each device, we tested the overall performance for invoking a simple image transformation service method (image size 2048KB, scale factor 3:1) facilitating all three available technologies – WS, RMI, and LSC/TCP. All requests originated from a desktop client where the VC Proxy was installed. Fig. 5 demonstrates the well-documented

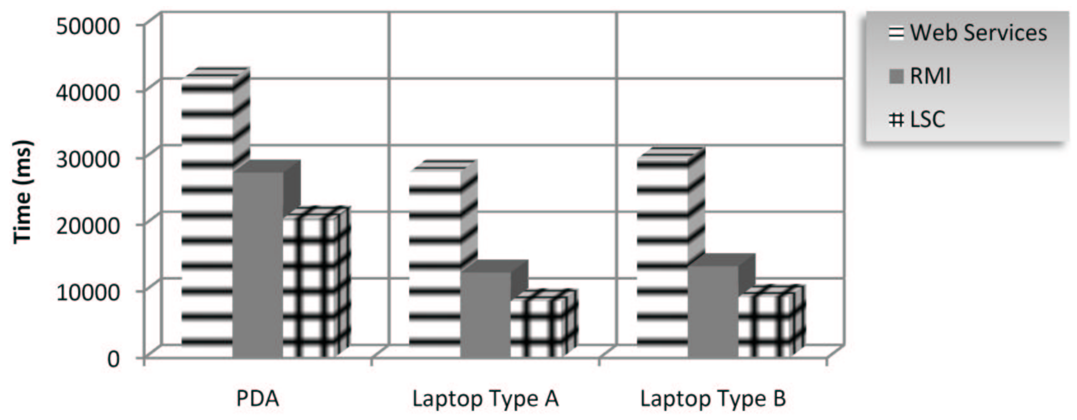


Figure 5: Comparison of WS, RMI, and LSC Single Invocation Performance

difference between WS and RMI performance, but most importantly, the significantly better performance of the LSC. This does not come as a surprise as generally sockets represent the fastest possible communication method among the three, while the interoperability and low-level programming restrictions are diminished with the use of the VC Aggregator scheme. Clients or developers will never have to make direct use of the LSC protocol. Instead it is meant to be accessed through the standards-based Aggregators in the VC Proxy.

Regarding the storage requirements, a total disk space of approximately 76 KB is required, including 42 KB for the vcluster-mobile.jar component, 31 KB for the LSC.jar component, 2 KB for the service-registry.xml component, and 1 KB for the startup scripts. This, however, does not include the RMI-Optional Package, which is required for some non-functional aspects of the VC platform, thus an additional 144KB is required to install the full package. The current version of the VC platform implementation is publicly available.

Evaluating the memory requirements entails testing the MPA under two different states: running/passive and running/active. In the first mode, the MPA and LSC have been bound to a VC proxy, but there are no pending requests. In the active state, the LSC has received an invocation request and is currently executing a Java service. As can be seen in Fig. 6, when the MPA is in passive mode it occupies a mere 2.8 MB of memory. On the other hand, the memory overhead associated with translating a byte stream into a Java method invocation request, and invoking the relevant Java service is 5.7 MB, and the total memory usage in this case climbs up to 8.5 MB, a significant increase, albeit temporary and well within the capabilities of resource-constrained devices. The P990i we used in these experiments never became unresponsive, and coped very well with all invocation requests.

6 Conclusions and Future Directions

The LSC is a lightweight performance-based container that is very easily able to run on any number of pervasive and resource limited mobile devices due to its small memory and storage requirements. To achieve this widespread functionality, certain value-added features that are present in the core MPA have been purposely left out of the LSC, such as the monitoring, dynamic service discover, and activation features. Future research in this project is moving towards the direction of enabling lightweight Peer-to-Peer mechanisms that would facilitate the dynamic loading of extra modules into the LSC depending on the capabilities of a given device. In this scenario, each client in the VC is

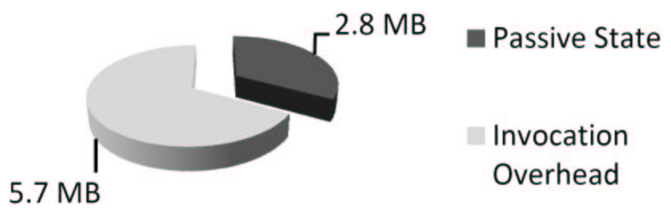


Figure 6: MPA Memory Usage

guaranteed to have a base functionality of those provided by any LSC, however, if they are capable of performing more resource intensive operations, such as sending and receiving notifications of neighborhood monitoring information, or acting as a VC proxy, they would then be promoted on a need-based basis to act as higher-level network participants.

The MPA complements the VC platform and consists of two distinct parts: the core MPA that supports fundamental non-functional properties of the VC platform, such as monitoring, dynamic service discovery and activation, and the LSC that provides a lightweight and performance-oriented alternative to standard containers (such as WS or RMI) suitable for resource constrained devices. The properties, interface, usability, and evaluation of the LSC were presented in detail in this paper.

The LSC framework manages to marry the high performance of low-level TCP sockets with the interoperability of the standards-based Web Services paradigm through the high-level interfaces of Aggregators. Furthermore, usability is ensured with a fully J2ME-compliant graphical interface to assist the mobile service provider in selecting and exporting his services.

The resource requirements evaluation verified the suitability of the LSC framework for even very limited devices, while the performance evaluation puts it ahead of typical service hosting environments at approximately half the cost of RMI interactions. In the context of the VC platform it presents a performance-oriented container, albeit still interoperable through the Aggregator framework.

References

[1] Schall, D., Aiello, M. and Dustdar, S., *Web Services on Embedded Devices,* International Journal of Web Information Systems (IJWIS), 2007.

[2] J. Wikman, F. Dosa, *Providing HTTP Access to Web Servers Running on Mobile Phones,* Nokia Technical Report NRC-TR-2006-005, May 24, 2006

[3] Chu, D. and Humphrey, M., *Mobile OGSI: Grid Computing on Mobile Devices,* 5th International Workshop on Grid Computing (GRID), 2004.

[4] Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C., Maguire, T., Sandholm, T., Snelling, D., Vanderbilt, P., *Open Grid Services Infrastructure (OGSI) Specification,* [Online] http://www.ggf.org/documents/GFD.15.pdf, 27 Jun, 2003.

[5] Czajkowski, K., Ferguson, D., Foster, I., Frey, J., Graham, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W., *The Web Services Resource Framework,* [Online] http://www.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf, 3 May, 2004.

[6] Thiyagalingam, J., Isaiadis, S. and Getov, V., *Towards Building a Generic Grid Services Platform: A Component-Oriented Approach,* in V. Getov and T. Kielmann [ed.], *Component Models and Systems for Grid Applications,* Springer, 2004.

[7] Thiyagalingam, J., Parlavantzas, N., Isaiadis, S., Henrio, L., Caromel, D., Getov, V., *Proposal for a Lightweight, Generic Grid Platform Architecture,* IEEE HPC-GECO/Compframe Workshop, HPDC-15, 2006.

[8] CoreGRID. [Online] http://www.coregrid.net.

[9] A. Harrison and I. Taylor, The Web Services Resource Framework In A Peer-To-Peer Context, Journal of Grid Computing, vol. 4(4): 425445, December 2006.

[10] I. Wang, P2PS (Peer-to-Peer Simplified), in Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies. Louisiana State University, pp. 5459, February 2005.

[11] A. Harrison and I. Taylor, Service-oriented Middleware for Hybrid Environments, in Advanced Data Processing in Ubiquitous Computing (ADPUC 2006), 2006.

[12] kXML Project. [Online] http://www.kxml.org.

[13] kSOAP Project. [Online] http://www.ksoap.org.

[14] Isaiadis, S. and Getov, V., *Integrating Mobile Devices into the Grid: Design Considerations and Evaluation,* Proc. of the 11th Int. Euro-Par Conference on Parallel Processing. pp. 1080-1088, 2005.

[15] Isaiadis, S. and Getov, V., *Dependability in Hybrid Grid Systems: a Virtual Clusters Approach,* IEEE JVA International Symposium on Modern Computing, 2006.