Design and Implementation of A Listener Module for Handheld Mobile Devices[1]

Sushil K. Prasad[1] Erdogan Dogdu[1] Rajshekhar Sunderraman[1] Bing Liu[1] Vijay Madisetti[2]

[1] Georgia State University
Department of Computer Science
Atlanta, GA 30303
{sprasad,edogdu,raj}@cs.gsu.edu, bliu1@student.gsu.edu

[2]Georgia Institute of Technology
School of Electrical and Computer Engineering
Atlanta, GA 30332
vkm@ece.gatech.edu

ABSTRACT

We have developed a generic "Listener" module in Java called SyDListener, or "System on Mobile Devices (SyD) Listener Module". It is designed and implemented as part of a software package we developed that is called SyDKernel. SyDKernel is a middleware package that can be used in developing distributed collaborative software applications in a rapid and modular way. Its executable code has a small size to fit in small devices like PDAs. Application developers use SyDKernel to easily implement distributed elements of an application such as two-way communication, resource sharing, collaborative services, directory listing, search and discovery services, and remote method invocations. SyDListener is a central module in SyDKernel, it provides a set of interfaces and classes that allows distributed SyD-based application components to communicate via remote method invocations seamlessly. SyDListener is implemented using TCP sockets for remote method invocatins and Java/RMI for locating methods. This paper describes the functionality, design rationale, architecture, and implementation of SyDListener.

Keywords

Listener module, System on Mobile Devices (SyD), mobile applications, PDA, middleware, XML, Java/RMI, dynamic remote method invocation.

1. INTRODUCTION

There is an emerging need for a comprehensive middleware technology to enable development and deployment of collaborative distributed applications over a collection of mobile wireless and wired devices. This has been identified as one of the key research challenges in Mobicom 2002 [1][5]. Our work is an ongoing effort to address this challenge, and this paper reports our first prototype design and its implementation of the core component of our middleware – the listener module. We seek to enable group applications over a collection of heterogeneous, autonomous, and mobile data stores, interconnected through wired or wireless networks of various characteristics, and running on devices of varying capabilities (pagers, cell phones, PDAs, PCs, etc.). The key requirements for such a middleware platform, and therefore, goals for our platform, are that it should provide a uniform connected view of device, high level development and deployment environment and platform for peer-to-peer or distributed server applications.

The current technology for the development of such collaborative applications over a set of wired or wireless devices and networks has several limitations. Developing an application requires explicit and tedious programming on each kind of device, both for data access and for data communication. The application code is specific to the type of device, data format, and the network. The data-stores are typically a centralized logical entity providing only a fixed set of services, with little flexibility for user-defined ad hoc services or the ability of user applications to dynamically configure a collection of independent data stores. Applications running across mobile devices are complex because of the lack of persistence of their data due to their weak connectivity. There are only a few existing middlewares which address the stated requirements. Even these are either not completely functional at this time, or enable only client-side programming on mobile devices, or are geared to a limited domain of applications, or are limited in group or transaction functionalities or mobility support.

System on Mobile Devices (SyD) is a new technology that addresses the key problems of heterogeneity of device, data format and network, and that of mobility. SyD combines ease of application development, mobility of code, application, data and users, independence from network and geographical location, and the scalability required of large enterprise applications concurrently with the small footprint required by handheld devices. SyD uses the simple yet powerful idea of separating device management from management of groups of users and/or data stores. Each device is managed by a SyD deviceware that encapsulates it to present a uniform and persistent object view of the device data and methods. Groups of SyD devices are managed by the SyD groupware that brokers all inter-device activities, and presents a uniform world-view to the SyD application to be developed and executed on. All objects hosted

ACM Southeast Conference'2003, March, 2003, Savannah, GA.

[1] This research is supported by State of Georgia's Yamacraw Research Contract #BLA42, #CLH49 and #DLN01.

by each device are published with the SyD groupware directory service that enables SyD applications to dynamically form groups of objects hosted by devices, and operate on them in a manner independent of devices, data, and underlying networks. The SyD groupware hosts the application and other middleware objects, and provides a powerful set of services for directory and group management, and for performing group communication and other functionalities across multiple devices.

SyD encapsulates different devices into a persistent object framework thus addressing device and data heterogeneity problem. SyD seamlessly and dynamically integrates a collection of disparate devices using a standard SyD middleware that also enables development as well as execution of SyD applications completely independent of device, data, and network. SyD middleware can function with or without a backbone network infrastructure on weakly connected networks as well as on ad-hoc networks providing varying levels of QoS guarantees [10].

2. OVERVIEW OF SyD ARCHITECTURE

The SyD architecture has three layers, as detailed in [4].

1) At the lowest layer, individual data stores are encapsulated by device objects. These device objects export the data that the device devices hold along with methods/operations that allow access, as well as manipulation of this data in a controlled manner. This is enabled by SyD Deviceware consisting of a listener module to register objects and to execute local methods in response to remote invocations, and an engine module to invoke methods on remote objects. SyD device objects also have inherent capabilities link with each other in an interdependent fashion to enable object composition and atomic transactions over multiple objects, provided by a linking module.
2) At the middle layer, there is SyD groupware, a logically coherent collection of services, APIs, and objects that facilitates the execution of application programs. Specifically, SyD groupware consists of directory services module, portion of engine module for group service invocation and result aggregation, and an event handler module for global events.
3) At the highest level are the applications themselves. They rely only on these groupware services, and are independent of device, database and network. These applications include instantiations of SyDApp Objects that are aggregations of the device objects, and SyD middleware objects. The three-tier architecture of SyD enables applications to be developed in a flexible manner without knowledge of device, database and network details.

We have developed a prototype implementation of SyD middleware and several SyD-based applications, such as mobile fleet described in [6]. Figure 1 depicts the layered architecture of SyD runtime environment in the current implementation. SyD in this environment is a middleware providing distribution transparency and management to SyD-based application development. It is located between applications and the communication services provided by primitive distribution middleware (Sockets, RMI, JXTA, CORBA, etc.). Each layer depends on the services provided by a lower layer. Therefore, each layer hides complexities of the tasks provided in that layer from upper layers. In this framework, applications are developed rapidly using SyD Kernel modules without any knowledge about lower layer services (primitive distribution middleware, OS/environment).

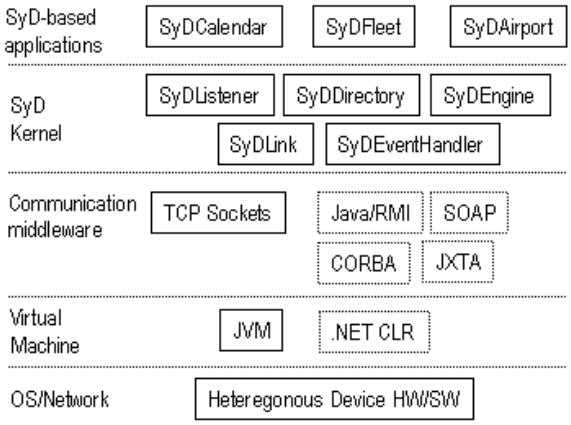

Figure 1. SyD Runtime Environment

SyD Kernel includes the following five modules:

a. SyDDirectory: Provides user/group/service publishing, management, and lookup services to SyD users and device objects. Also supports intelligent proxy maintenance for users/devices.
b. SyDListener: Enables SyD device objects to publish their services (server functionalities) as "listeners" locally on the device and globally via the directory services. It then allows users on SyD network to invoke single or group services via remote invocations seamlessly (location transparency) [8-9].
c. SyDEngine: Allows users to execute single or group services remotely and aggregate results.
d. SyDEventHandler: This module handles local and global event registration, monitoring, and triggering.
e. SyDLinks: Enables an application to create and enforce interdependencies, constraints and automatic updates among groups of SyD entities.

The SyD Listener module defines a set of interfaces and classes that allows distributed SyD-based application components to communicate via remote method invocations seamlessly. SyDListener enables SyD device objects to publish their services (server functionalities) as "listeners" locally on the device and globally via the directory services. It then allows users on SyD network to invoke single or group services via remote invocations seamlessly (location transparency). Figure 2 shows the interaction between SyD Listener module and other SyD modules.

SyD is the first comprehensive working prototype of its kind, with a small footprint of 112 KB with 76 KB being device-resident, and has a tremendous potential for incorporating many ideas in terms of performance extensions and scalability.

Three Utility Patents and many patent disclosures for SyD technology have been filed.

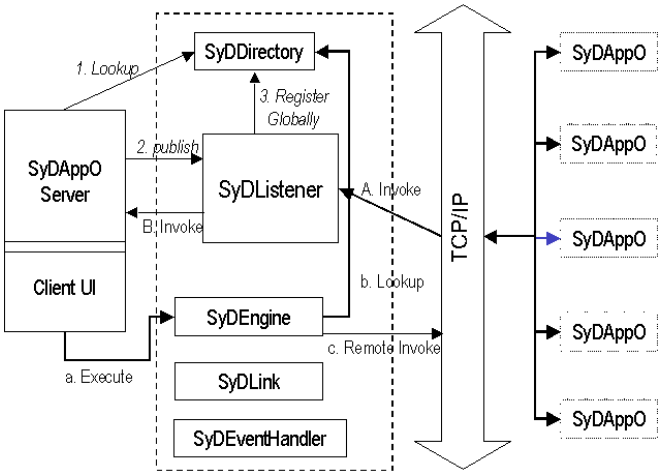

Figure 2. SyD Kernel architecture and interactions between modules and application objects.

3. DESIGN OF LISTENER MODULE

3.1 Architecture

SyD middleware is used for building distributed collaborative applications on heterogeneous devices, data stores, and networks. Today's connected devices range from regular PCs to laptops and to small handheld computers such as PDAs, and even cell phones with software application capabilities (Java-enabled phones). The next-generation computing devices will be "mobile" and "connected" with increasing computing powers. Future software application will therefore empower users with increasing capabilities in functions on their connected devices. Small device will not be just an accessory to read news headlines or check stock prices but they will provide interactive applications that will utilize the connectivity of such devices. This results in a number of requirements for SyDListener considering today's connected devices:

- SyDListener should be deployed over small devices such as PDAs to take advantage of "connectivity" in those devices.
- SyDListener should be lightweight so that it should require only minimal hardware and software to be deployed on small connected devices, such as PDAs and cell phones.
- The architecture of SyDListener should be flexible so that heterogeneity in system, software, hardware, and environment can be encapsulated.

Since TCP/IP is a widely used transport protocol and most handheld devices support it. Therefore, TCP/IP is used as the communication protocol between SyD server and client devices.

Each device in SyD can act as the container for either server application or client application, both of which are SyDApp objects. So the term "server device" only has relative meaning, as well as all the terms of "server" and "client" mentioned in this paper.

Figure 3 shows the first level of abstraction for the architecture of SyDListener. In order to invoke a remote service, a SyDApp object on the client device will open a TCP connection to the SyDListener running on the server device and send an invocation message. Invocation messages in SyD are in XML format. We proposed a specification to represent object information inside an XML message. The creation and parsing of XML messages are handled by another SyD module that is called SyDDoc. SyDListener invokes the actual SyDApp that provides the service requested by passing the method parameters and receiving generated results. Since XML document is passed simply as characters, it is independent from the programming language and operating system as long as all the XML packagers and parsers follow the same SyD specification.

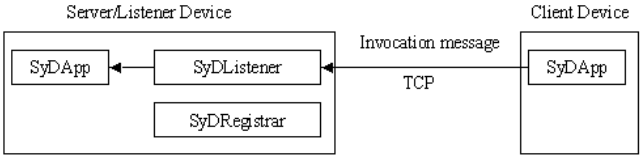

Figure 3. Invocation of SyDApp Service through SyDListener

Here we do not specify which mechanism to use for the listener to invoke the service because this invocation mechanism should be decided by the way the service is provided. The service provider should design and implement the SyDApp in the most efficient and powerful way that is suitable for the specific device on which the SyDApp will run. According to the multiple ways that SyDApps can be implemented, Listener should have multiple implementations and be capable of invoking services using multiple mechanisms. As long as SyDApp is registered and the SyDListener is informed about how to call it, the invocation process will be handled by SyDListener seamlessly and present a simple and uniform interface to the service requester. But the client still has to know how to connect to the SyDListener.

In order to encapsulate the communication details between client and SyDListener for the SyD application developer, a delegate for listener should be provided on client devices. SyD listener is designed with a distributed architecture using adaptor pattern. It consists of a registration and a listening part located at a device that can provide services for the SyD network, and a delegate part at the client device that will request services from the server device. The delegate acts as a local proxy for the listener object on the server device. This architecture has the advantage of hiding invocation details of the service from the service requester as depicted in figure 4.

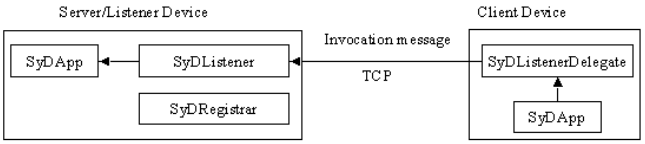

Figure 4. Invocation of Service through SyDListenerDelegate.

Our SyDListener architecture has an extensible design that allows us to use any communication method or middleware solutions between SyDListener and SyDApp service, such as Java/RMI, CORBA, SOAP, DCOM, or raw TCP. In this architecture SyDListener acts as a proxy for the clients. Therefore, it is even possible to deploy SyDListener and SyDApp on separate devices; client does not have to be aware of this, and it is only concerned with its communication to SyDListener and its interface (Figure 5). Similarly, client side SyDApp object talks to a device resident SyDListenerDelegate object to communicate with the remote SyDApp object. SyDListenerDelegate in this framework is a stub

on the client side, providing a local interface for the remote services (Figure 4 and 5).

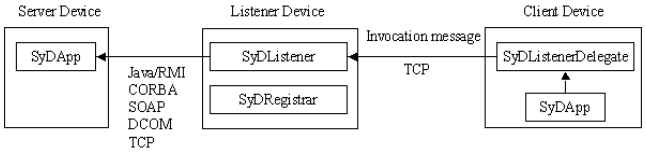

Figure 5. Listener and Server on Different Device

3.2 Object Lifecycle

Up to this point, the SyDListener only behaves passively, which means it does not manage the lifecycle of the SyDApp server object. The SyDListener assumes that the server object was created before registration and will never die in the whole process of service. Whenever a request comes, the SyDListener just forwards the request to the existing server object. If the server object dies for some reason, the SyDListener does not know this and will get no response from the server object, and then finally times out.

A full-fledged SyDListener should have the ability to manage the lifecycle of SyDApp server objects, including creation/activation, deletion/deactivation, and persistence. The following sections discuss the object creation and persistence in SyDListener.

3.2.1 Service Object Creation

When a request for a SyDApp service arrives at SyDListener and there is no existing SyDApp service object, SyDListener should create a SyDApp service object to process the service request. There are two possible ways that SyDListener creates a service object, directly or indirectly. If a factory object for the service is provided and installed, the SyDListener can use the factory object to create the service object indirectly. Otherwise, the SyDListener can only create the service object directly by itself. In the second case, SyDListener must be able to access the code for the service class and the SyDListener object will serve as a container for the service object.

3.2.2 Object Persistence

If SyDListener takes up the responsibility of object creation, it must take care of object persistence for the stateful objects, if there is any.

Through careful design of service class, the service design can make the service class store all information into persistent storage and manage the retrieval and update by itself. The service class designed this way will be stateless and free the SyDListener from the burden of persistence management because each instance of the service class with no internal state will behave in the same manner. But this results in performance degradation. Every operation will involve access of the persistence storage, which is relatively slow.

If we choose to use stateful objects, the SyDListener must manage a persistent data store independently from the SyDApp service. We plan to develop an XML database to store the service object state in flat file when a service object is deactivated and retrieve the state to initialize a new service object when a service object is created. The same XML processing functionalities with the remote service invocation is needed here. And, a persistence specification like PSSDL in CORBA is necessary to handle the mapping between object and storage [2].

3.3 Proxy

When a handheld device that provides a service is down, the proxy of the service will take up the responsibility and continue to serve the coming service requests. SyDListener can help to implement this mechanism and hide the existence of proxy from the clients of the service. A detection mechanism is needed to determine if an out-of-order condition for the service happens due to time-out or some other methods. Assuming the existence of this detection mechanism, there are two possible ways that SyDListener can be designed and the following section will discuss those.

The concept of proxy needs to be clarified to facilitate further discussion. Here the proxy is actually a SyD service that runs on a different device as the original SyD service and implements the same interface with the original SyD service. But they are not identical in that the communication method and invocation method may be different.

3.3.1 SyDListener As Proxy Manager

In the first design, SyDListener acts as the manager for the proxy. When the service is registered to SyDListener, a proxy for the service has already been created on another device and the proxy information is also registered into directory service through SyDRegistrar. So, when the original service is not available, SyDListener knows where to find the proxy and how to communicate with it and invoke methods on it (Figure 6).

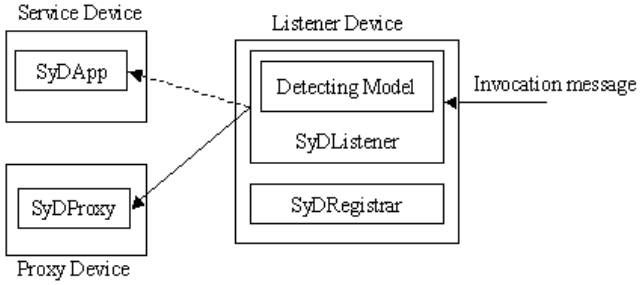

Figure 6. SyDListener As Proxy Manager

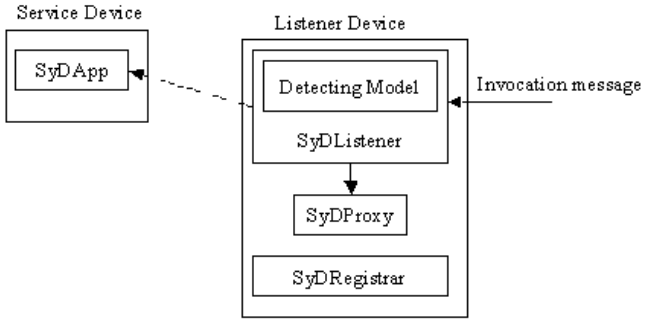

Figure 7. SyDListener As Proxy Container

3.3.2 SyDListener As Proxy Container

As we can see in Figure 7, this model is a distributed model and can operate without the SyD directory service. Or, in other words,

the SyDListener itself acts as a self-managed part of the SyD directory service. This second model requires more processing power on the listener device because it needs to instantiate the whole proxy object instead of only the stub.

4. IMPLEMENTATION OF SyDListener USING Java/RMI AND TCP SOCKETS

4.1.1 Basic Architecture

The first implementation of SyDListener prototype is built upon Java RMI and TCP socket programming [3] on Personal Java platform. SyDListenerDelegate and SyDListener communicate through TCP sockets while SyDListener accesses active service objects from an RMI registry.

SyDListener object will continue listening to a specified port and receiving invocation messages in XML format from client. Then it uses parsing functionalities of SyDDoc, which is a utility class in SyD Kernel for XML processing, to extract object identifier, method name, and parameter values. After parsing, SyDListener makes actual invocation on the service object's method and gets the result. Finally it packages the result in XML format and returns it through the same port. To initiate and terminate the communication process of invocation, SyDListener and SyDListenerDelegate need to perform a handshake protocol to synchronize the behavior of server and client. The architecture of the implementation is shown in figure 8.

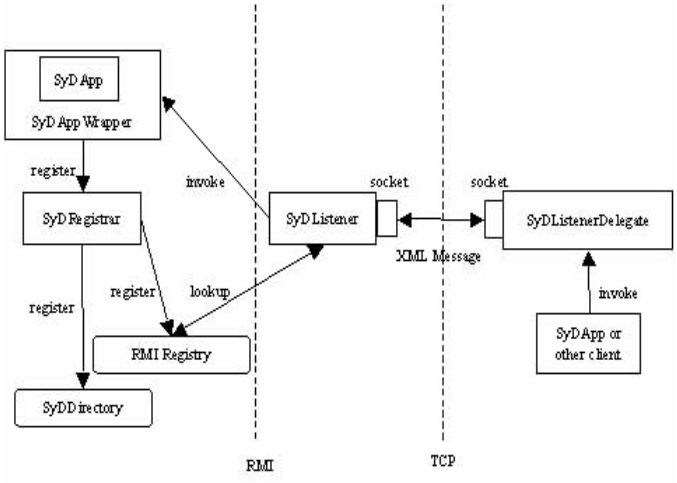

Figure 8. Implementation Using Java/RMI and TCP Sockets

4.1.2 Dynamic Invocation

Java RMI only supports static remote method invocations, which means that SyDApp clients must obtain the stub for the SyDApp services either at compilation time or at run time in order to use the methods defined in the services [7].

The current implementation of SyDListener integrates Java reflection mechanism with RMI and enhances RMI with dynamic method invocation capability. The SyDListener will perform reflection analysis on the remote reference obtained from RMI Registry service and constructs dynamic invocation using the structure information from this analysis and parameter value information from XML invocation message. So, the client of the service has no need to hold a stub of the remote service, either at compile time or at run time.

4.1.3 RMI Wrapper

In order to use RMI mechanism, SyDApp developer must write SyDApp service explicitly as RMI service. To be an RMI service, a Java class must implement an interface, which extends java.rmi Remote interface, and each public method inside the interface must throw java.rmi.RemoteException. This brings extra burden to service development.

An RMI wrapper is a RMI service itself and is able to perform reflection analysis and to activate methods of the objects it has access to. Using a wrapper, the SyDApp service could be written as a normal Java class without consideration of RMI. The wrapper holding a reference to an instance of the SyDApp service will be registered and act as a front end to the service.

4.1.4 Interfaces

There are three classes in listener package, which are SyDRegistrar, SyDListener, and SyDListenerDelegate. There are also two helper classes: SyDAppWrapper is the wrapper class for SyDApp objects and XMLDoc is actually Java String in XML format with extra parsing functions.

- Interface of SyDRegistrar

void register(SyDAppWrapper wrapper, XMLDoc publishDoc)

where wrapper is the instance of the service providing application that need to be registered. Service registration is performed by calling register() method on SyDRegistrar.

- Interface of SyDListener

SyDListener is located at server side and performs listening, parsing and invoking for remote invocation. It has no public method because it is hidden by the SyDListenerDelegate.

- Interface SyDListenerDelegate

XMLDoc invoke(XMLDoc input)

where input is the invocation message. The service requester will make service request by calling invoke() method on SyDListenerDelegate.

4.1.5 Usage Scenario

The sequence of steps that take place at server side and client side are described as following.

At server device:

step 1. Create service object.

step 2. Pass the reference to the service object to a SyDAppWrapper object.

step 3. Register the SyDAppWrapper object to rmiregistry and SyD directory service through SyDRegistrar.

step 4. Create listener for the service.

At client device:

step 1. Query directory service for available service.

step 2. Create an invocation message and invoke the service through SyDListenerDelegate.

step 3. Wait until result is available.

4.1.6 Hardware/Software Platform

Various technologies employed to prototype SyD kernel are as follows:

- HP's iPAQ models 3600 and 3700 with 32 and 64 MB storage running Windows CE/Pocket PC OS interconnected through IEEE 802.11 adaptors cards and a 11 MB/s Wireless LAN.
- Jeode EVM personal java 1.2 compatible, implementing Java Virtual Machine.
- TCP Sockets for remote method invocation and JAVA RMI for local method execution through reflection.
- XML standard for all inter-device communication.

4.1.7 Footprint

The foot-print of the whole SyD kernel including SyDListener is 112 KB, out of which only 76 KB is currently device-resident; rest are for currently maintained on a central location outside for directory and global event handling. For even smaller devices, this can be further reduced to 42 KB. For a fully p2p version, all of 112 KB will be on the device. The SyDListener module itself is only 7.9 KB.

5. CONCLUSION

SyDListener module provides a highly flexible and easy to use mechanism for service registration and service invocation in distributed application development. It provides a unique remote dynamic invocation mechanism and hides all complex details of underlying communication middleware from both service providers and service consumers. Its flexible architecture allows smooth evolution in functionalities and accommodation of heterogeneities seamlessly. It acts as an active application server with small footprint on handheld devices.

Current prototype implementation of SyDListener uses Java/RMI and TCP sockets for method invocations. Small footprint of 7.9 KB of the whole SyDListener module implementation enables easy deployment on small devices. The architecture of SyDListener is extendable and allows easy integration with other middleware or persistent object mechanisms besides Java/RMI.

We are right now enhancing SyDListener module to provide object persistence for session and state information management on device. We are also investigating options on how to port the Listener to smaller devices such as Java-enabled cell phones. Performance issues regarding SyDListener are also being addressed by a concurrent work on proxy management in SyD for tolerating disconnections and/or enabling resource constrained devices to act as servers. To make SyD an open system we are researching on the inclusion of Web Services-based interfaces (SOAP/WSDL) for SyD communication.

6. ACKNOWLEDGMENTS

This research is supported by the State of Georgia's Yamacraw Embedded Software project.

7. REFERENCES

[1] W. Keith Edwards, Mark W. Newman, Jana Sedivy, Trevor Smith, Shahram Izadi, "Recombinant computing and Speakeasy Approach", in Proceedings of MobiCom September 2002

[2] Wolfgang Emmerich, Engineering Distributed Objects, John Wiley & Sons, 2000.

[3] Elliotte Rusty Harold, Java Network Programming, O'Reilly & Associates, 2000.

[4] Vijay Madisetti, Sushil K Prasad, et al., System on Devices (SyD): An Enabling Technology for Programming Applications on Multiple Mobile Data-stores, Utility Patent filed, 2002.

[5] Thomas Phan, Lloyd Huang, Chirs Dulan , "Integrating Mobile Wireless Devices Into the Computational Grid", in Proc. of MobiCom, September 2002.

[6] S. K. Prasad, M. Weeks, Y. Zhang, A. Zelikovskiy, S. Belkasim, R. Sunderraman, and V. Madisetti. 2002. "Mobile Fleet Application Using SOAP and System on Devices (SyD) Middleware Technologies". *Proc. of Communications, Internet and Information Technology (CIIT 2002) Conference*, St. Thomas, Virgin Islands, USA, November 18-20, 2002, pages 426-431.

[7] Sun Microsystems Inc., Java Remote Method Invocation Specification.

[8] Sushil K. Prasad, Anu G. Bourgeois, Erdogan Dogdu, Raj Sunderraman, Yi Pan, Sham Navathe and Vijay Madisetti. 2003. Enforcing Interdependencies and Executing Transactions Atomically Over Autonomous Mobile Data Stores Using SyDLink Technology, *Mobile Wireless Network Workshop held in conjunction with The 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, May 19-22, Providence, Rhode Island, USA.

[9] Sushil K. Prasad, Anu G. Bourgeois, Erdogan Dogdu, Raj Sunderraman, Yi Pan, and Sham Navathe. 2003. Implementation of a Calendar Application Based on SyD Coordination Links, To appear in proceedings of *The Third International Workshop on Internet Computing and E-Commerce (ICEC'03) in conjunction with the 17th Annual International Parallel & Distributed Processing Symposium (IPDPS 2003)*, 22-26 April, Nice, France.

[10] Wanxia Xie, Shamkant B. Navathe, Sushil K. Prasad. 2003. "Supporting QoS-Aware Transaction in the Middleware for a System of Mobile Devices (SyD)". To appear in proceedings of *1st International Workshop on Mobile Distributed Computing* (*MDC'03) held in conjunction with The 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, May 19-22, Providence, Rhode Island.