Towards an Agile Computing Approach to Dynamic and Adaptive Service-Oriented Architectures

Niranjan Suri[1,2], Matteo Rebeschini[1], Marco Arguedas[1], Marco Carvalho[1],
Stefano Stabellini[1,3], and Maggie Breedy[1]
[1]Florida Institute for Human & Machine Cognition
[2]Lancaster University
[3]University of Ferrara
{nsuri, mrebeschini, marguedas, mcarvalho, sstabellini, mbreedy}@ihmc.us

Abstract-**Agile computing is an innovative metaphor for distributed computing systems and prescribes a new approach to their design and implementation. Agile computing may be defined as opportunistically discovering, manipulating, and exploiting available computing and communication resources in order to improve capability, performance, efficiency, fault-tolerance, and survivability.**

This paper describes the realization of the middleware and the AgServe library that supports dynamic and adaptive service-oriented architectures. The middleware supports service definition, instantiation, invocation, relocation, and termination to be performed dynamically at runtime. A coordination mechanism continuously monitors service resource utilization, invocation patterns, and network and node resource availability to determine optimal locations for services to be instantiated and invoked. The continuously adaptive nature of agile computing makes it well suited to providing a foundation for autonomic computing.

Index Terms-**Agile Computing, Service-oriented Architectures, Dynamic and Adaptive Middleware**

I. Introduction

Service-oriented architectures are a popular approach to constructing distributed systems. They provide the basis for grid computing and distributed processing. Services are a convenient abstraction for packaging computing capabilities and providing them to remote nodes over a network connection. Normal service-oriented architectures tend to be somewhat static in terms of the location of services (although their availability may change over time). Agile computing is an approach to building distributed systems that are opportunistic in discovering and using new computational resources. The term agile is used to highlight the desire to both quickly react to changes in the environment as well as to take advantage of transient resources only available for short periods of time. Agile computing thrives in the presence of highly dynamic environments and resources, where nodes are constantly being added, removed, and moved, resulting in intermittent availability of resources and changes in network reachability, bandwidth, and latency. Using an agile computing approach to build service-oriented architectures makes them more dynamic and adaptive. The continuously adaptive nature of agile computing provides a foundation for self-healing systems and autonomic computing.

The AgServe library provides the capabilities for supporting service-oriented architectures on top of the core agile computing middleware. One of the goals for AgServe is to make it easy to port existing services to run in the new architecture and thereby transparently making them more dynamic and adaptive. The AgServe library supports service definition, instantiation, invocation, relocation, and termination of services. The underlying middleware monitors service resource utilization, invocation patterns, and network and node resource availability and uses that information to determine optimal location for services to be instantiated and invoked. A heuristic coordination algorithm performs the service allocation and reallocation on a continuous basis.

This paper is organized as follows. Section two presents the overall architecture for the middleware. Section three describes the AgServe library and the realization of the service-oriented architecture on top of the middleware. Section four describes the implementation details of service deployment, activation, and invocation. Section five presents some initial experimental results. Finally, section six concludes with a summary of the paper and a discussion of future work.

II. Overall Architecture

Figure 1 shows the overall architecture for the middleware. The five major components of the middleware are the Agile Computing Kernel, the Coordinator, AgServe, FlexFeed, and Mockets. In addition, the Visualizer and the KAoS policy and domain services framework [1][2] allow the user to observe and constrain the behavior of the system respectively.

The user-contributed components are represented in the figure as either clients or services. Services may either be

started up via external means and then register with the middleware, or they may be defined and started up by the middleware on demand. The former case is appropriate for situations where services are bound to particular nodes and represent the access mechanism to the node (for example, a service that wraps a database at a host). The latter case is appropriate for situations where the number of instances of a service can vary over time based on load, resource availability, and network configuration.

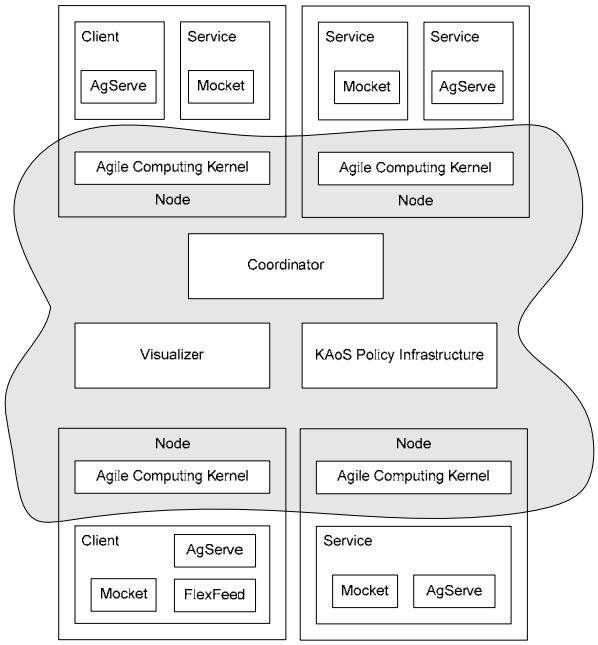

Figure 1: Overall Architecture of the Agile Computing Middleware

The following subsections describe the kernel, coordinator, AgServe, and the other components of the middleware.

A. Agile Computing Kernel (ACK)

The Agile Computing Kernel contains execution environments, a protocol redirector and protocol handlers, a group manager, a local coordinator, and other components. Figure 2 shows the main components of the ACK. These components provide the set of capabilities that the running clients and services rely upon to take advantage of the agile computing framework. Clients and services interact with the kernel through the AgServe library. The following subsections provide a brief explanation of each of the components.

VM Containers

The VM Containers provide a common environment for code execution that hides underlying architectural differences such as CPU type and operating system. The execution environment supports the dynamic deployment and activation of services, dynamic migration of services between kernels, secure execution of incoming services, resource redirection, resource accounting, and policy enforcement.

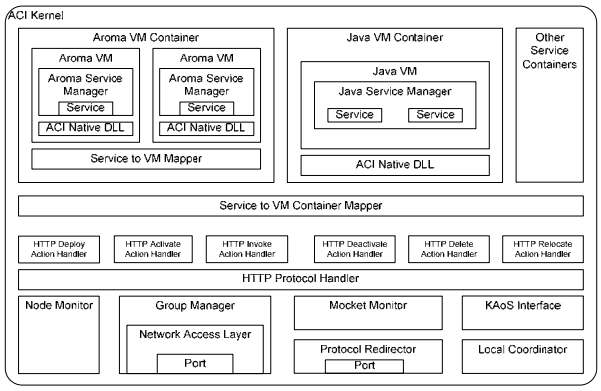

Figure 2: The Agile Computing Kernel

The VM Containers are currently implemented using either the Aroma VM [3] or the standard Java VM. Aroma is a clean-room implementation of a Java-compatible VM designed to provide architecture independence and support for agile computing requirements. Aroma was designed from the ground up to support capture of execution state of Java threads, provide accounting services for resource usage, and control resource consumption by Java threads. Moreover, the captured execution state is platform independent, which allows migration of computations between Aroma VMs that are running on different hardware platforms. The capabilities of the Aroma VM are important to ensure secure execution of mobile code.

A standard Java VM is an alternative to the Aroma VM and provides higher performance. Services deployed in the Java VM Container do not benefit from state capture and migration. A subset of the resource management capabilities of the Aroma VM are available in the Java VM with the addition of the JRaf2 framework [4]. A deployment of the middleware can contain any combination of kernels with Aroma and Java VMs, thereby providing additional flexibility between performance and capability. We are also working on other VM containers including one based on the IBM Jikes Research VM [5].

There are no implicit or explicit requirements to use Java as the language for the implementation of the agile computing infrastructure. However, Java provides many desirable features for a mobile-code based framework. Moreover, the virtual machine architecture of Java provides platform independence.

Protocol Redirector and Handlers

The kernel includes a number of components that handle communication between the kernel and other nodes. The Protocol Redirector is the primary connection point to the kernel. Any external node that wishes to connect to the kernel does so through the protocol redirector. After connecting, the remote node may invoke one of a number of protocol handlers to deploy, activate, invoke, deactivate, delete, or relocate services.

Group Manager
The Group Manager component supports resource and service discovery. Discovery is a challenging problem in dynamic networks that are ad-hoc and peer to peer. This component of the middleware supports bandwidth-efficient discovery of neighboring nodes, their resources (including CPU, memory, storage, and connectivity), and their services. Nodes may join one or more groups that allow resources and services to be categorized into related sets.

The group manager supports discovery of neighbors and the resources and services available at neighbors. In addition, a search mechanism supports discovery of resources and services at non-neighboring nodes. The design principles of agile computing imply that finding resources and services that are nearby is better than finding those that are farther away (in terms of network hops). Therefore, one optimization criterion for finding nodes is network proximity. A second optimization criterion is finding nodes that are resource rich or have excess capacity. Both of these criteria are incorporated into the approach realized by the group manager.

Node Monitor
The node monitor component monitors resource availability at the local node. Resources currently monitored include CPU, memory, disk storage, and network. The local resource information is propagated to other nodes and to the coordinator through the Group Manager component.

Other Components
The KAoS interface is responsible for communicating with the KAoS policy and domain services framework [1][2]. KAoS manages the specification, conflict resolution, and distribution of policies to the enforcement components (in this case, the agile computing infrastructure). The interface component provides a facility for other components in the kernel to query and determine policies and restrictions that apply to local and remote services and nodes.

The Mocket Monitor is used to gather statistics about Mocket connections on the local node. The mockets communications library is discussed in more detail in section 2.3 below.

The Local Coordinator works in conjunction with other local coordinators as well as centralized or zone-based coordinators to perform resource allocation, service deployment, service invocation, and service migration decisions. The coordination mechanism is discussed in more detail in the next section.

B. Coordinator

The Coordinator is an abstract entity that is responsible for allocating and monitoring overall resource utilization in order to ensure that service allocation is in compliance with current optimization criteria and policy constraints. The Coordinator can be implemented following a centralized approach (where there is only one instance for the whole system), a fully distributed approach (where every node behaves in a peer-to-peer fashion), or a zone-based approach. The zone-based approach divides the system into zones within each of which is a master coordinator. The master coordinators manage nodes in their zone much like in the centralized approach, but coordinate in a peer-to-peer fashion with other masters.

In the first case, the Local Coordinator component at each kernel acts essentially as an interface to the centralized Coordinator that maintains the global state of the network. In the second case, the Local Coordinator at each node is directly responsible for the allocation of its local resources at any given time. This is achieved through direct negotiation with other Local Coordinators. The third approach, which is likely to be the most scalable, is zone-based and hierarchical. The network is self-organized into small zones whose combined resources are regulated by a locally elected Coordinator. Zone-level Coordinators are regulated by higher level Coordinators that handle only aggregate information from each zone, allowing for a scalable and efficient model for resource coordination.

The approach used for the purpose of this paper is a centralized approach. The algorithm uses a simple heuristic of monitoring the CPU load on the available nodes. When a request to activate a service is received, the service is activated on a node that has sufficient CPU resources as well as sufficient bandwidth between the client node and the node to which the service is deployed. Subsequent invocations of the service continue on that node until either additional clients request services or the resource availability changes.

C. Mockets and FlexFeed

Mockets (for "mobile sockets") is a comprehensive communications library for applications. The design and implementation of Mockets was motivated by the needs of wireless and ad-hoc networks with low bandwidth, intermittent connectivity, and variable latency. Mockets addresses specific challenges including the need to operate on a mobile ad-hoc network (where TCP does not perform optimally), provides a mechanism to detect connection loss, allows applications to monitor network performance, provides flexible buffering, and supports policy-based control over application bandwidth utilization.

FlexFeed [6] is a publish-subscribe library that supports hierarchical data distribution, policy enforcement, and in-stream data processing. FlexFeed generates data flow graphs as a result of subscribers requesting data feeds from publishers and then maps the data flow graph to the underlying environment, taking into account the resource availability and network connectivity. FlexFeed in turn uses services that are deployed in the agile computing kernels to handle the data dissemination.

III. AGSERVE

The AgServe library supports service oriented architectures on top of the agile computing middleware. AgServe handles service definition, registration, deployment, activation, lookup, invocation, deactivation, deletion, and migration (although migration is yet to be implemented). The following subsections describe each of these operations.

A. Service Definition

Service definition with AgServe is different (but complements) service definition in a service-oriented architecture. For example, Web Services use the Web Service Definition Language (WSDL) [7] to define a service in terms of the set of operations and the input and output messages. This type of service definition supports discovery of services and operations by clients and validation of inputs and outputs.

Service definition with AgServe specifies other parameters including the code for the service (or the location where the code may be dynamically obtained), the resource utilization profile, and the communications profile. This meta information allows the coordination component to identify nodes with adequate resources to host services and to then dynamically deploy services onto those nodes.

B. Service Registration

Service registration is the mechanism through which a service that has been activated registers its instance with the middleware. Service registration occurs in two circumstances. A service that was previously defined may have been activated by the middleware (see section 3.4), leading to a new instance being created that subsequently registers with the middleware. Alternatively, specialized nodes that have predefined services (e.g., a database) register when the node is activated. These services provide the entry points for clients to access the sensors.
Service registration usually involves identifying the location at which the service may be reached. This is the equivalent of a port in WSDL that results from associating a binding with a network address. Once a service has been registered, the service may be reached over a network connection.

C. Service Deployment

One of the goals of agile computing is to be opportunistic and take advantage of new resources that become available. If the resource is a new node, it probably will not have the services that clients are trying to invoke. Therefore, AgServe supports dynamic deployment of services via mobile code. Services are packaged into archive files (described in section 4) and are pushed to the kernel on the new node. The kernel unpacks the archive and installs the service onto the node. Policies may be used to regulate the source of the services as required for security purposes.

D. Service Activation

Service activation is used by the coordination component to instantiate a new service on a node. This action is normally performed in response to a request from an application to invoke a service. In some cases, the middleware may use an existing instance of the service to satisfy the request while in others, the middleware may use this mechanism to create a new instance. The kernel on the target node handles the request and instantiates the specified service. Once activated, the service is identified by a UUID that is returned by the kernel to the Coordinator.

E. Service Lookup

Service lookup is performed by a client in order to find a service to invoke. The result of a service lookup is a set of network endpoints to allow the service to be invoked. Web Services use Universal Description, Discovery, and Integration (UDDI) [8][9] in order to publish and lookup services. Service lookup in AgServe differs from the UDDI approach in two important aspects.

With agile computing, service instances may be activated on demand on nodes that are opportunistically discovered. Therefore, a service instance may not be registered until a client invokes the service. This requires a separation between registering a service type versus a service instance. When a client requests the invocation of service, the invocation may only specify the type of the service. The middleware identifies the best instance (or creates a new instance) of the service and forwards the request.

Also, UDDI was designed in the context of corporate networking environments and the Internet. The network characteristics of these environments differ significantly from the wireless ad-hoc networks in dynamic environments. Some initial work has been done in adapting UDDI-style service lookup to ad-hoc networks [10]. In [11], researchers have shown the benefits of integrating service lookup protocols into reactive routing protocols. The requirements for agile computing are different due to two factors. The middleware needs to discover resource availability at nodes, which changes more rapidly than service availability. Moreover, given the opportunistic nature of the middleware, the resource discovery needs to be proactive as opposed to reactive. Currently, the Group Manager component of the Agile Computing Kernel handles node discovery as well as resource and service discovery.

F. Service Invocation

Service invocation begins by a client making a request for a service. Unlike a Web Services environment, the service invocation is a request to AgServe, not to the service. This allows the middleware to decide the best current instance of the service to handle the request or to create a new instance on a resource rich node.

The standard Web Services approach only supports a request-reply mode of operation. In this case, the service

invocation API supports two additional modes of operation: asynchronous invoke, and subscription oriented (which, in turn, may be ongoing until cancelled or for a specified duration). Request-reply is the simplest mode – the client invokes an operation and blocks until the operation has been completed and the results are available. Asynchronous invocation allows the client to invoke a service asynchronously and interact with the service later, to query the status or push new data to the service. Note that this mode can be realized with Web Services provided some state information is maintained across invocations.

The subscription oriented mode allows the client to invoke a service which then asynchronously pushes data back to the client. This mode is not directly supported by Web Services, but is an essential mode of operation to support publish-subscribe architectures. Without this capability, the client would have to periodically poll the service to check for new data, which is inefficient. Having this capability allows the service to asynchronously push data to the client when appropriate. The subscription-oriented mode is currently being implemented with AgServe.

IV. Implementation Details

A. Service Archives

Services in AgServe are packaged as ACR (Agile Computing aRchive) files. A basic ACR file contains the compiled Java classes that will be used to run the service, as well as a metafile that describes the service and some other options used during activation and invocation times. Additionally, the ACR file might also contain the source code of the service and resources needed by the service, such as additional libraries (in the form of JAR files), and configuration files.

B. Communication Protocol

AgServe uses HTTP as the transport protocol. The design choice of using HTTP arose from the need for maintaining compatibility with Web Services and from the fact that HTTP is a stateless protocol. A stateless protocol is better suited for ad-hoc wireless environments with intermittent connectivity.

SOAP Messages are embedded in the body of HTTP requests/replies. Service arguments can be either serialized into a XML representation or into a binary format.

C. XML and Java Serialization Support

AgServe supports two different representations for invocation requests (and replies): an XML-based representation and a binary representation based on the Java serialization format.

The XML-based representation allows for client applications written in other languages (besides Java) to invoke services with AgServe. However, this approach has a downside: if the service needs complex data types (complex data structures), both clients and services need to agree on a representation for the complex data type and would need to implement the encoding and decoding routines to convert the data types to and from XML.

If the client applications are also using Java and programming language independence is not an issue, they can use the pure Java binary representation. In this approach, all the parameters of the invocation request are serialized into a Java byte array. This byte sequence is transmitted as part of the HTML request to the kernel. Once the kernel hosting the service receives the array, the kernel deserializes the objects from the array and invokes the appropriate method on the service. No extra encoding or decoding steps are necessary.

D. DIME Encoding

DIME (Direct Internet Message Encapsulation) is a proposed internet standard for the transfer of binary and other encapsulated data over SOAP or HTTP. A DIME message is composed of one or more DIME records. Each DIME record has a small header with meta-information that describes the record.

Like MIME, DIME allows for multi-part messages. This feature comes in handy at the time of encoding an XML request. In this case, the first record of the DIME object will contain the XML representation of the request. Nevertheless, in case in the case of binary objects (such as a byte array or an image) being used as parameters, the parameter is replaced by a mark that indicates that the parameter can be found in another record within the same DIME message.

In the case of a serialized (binary) invocation, all the parameters of the request are serialized together into a large byte array, and then that array is placed in a single DIME record.

E. Invocation Process

An application can initiate an invocation request by making use of the service stub. The service stub relies on the AgServe library. The service arguments are marshaled into a DIME message, which constitutes the body of the invocation request. Figure 3 shows the process of service invocation.

When the Protocol Redirector receives a new HTTP connection, it hands the connection to the HTTP Protocol Handler, which analyzes the HTTP header and makes use of the proper HTTP Action Handler to manage the request (HTTP Invoke Action Handler in this case).

Based on the service instance id (contained in the HTTP header), the VM Container holding the desired instance of the service is chosen and the invocation request is passed to the Service Manager. The Service Manager decodes the request by parsing the DIME message and performs the actual invocation.

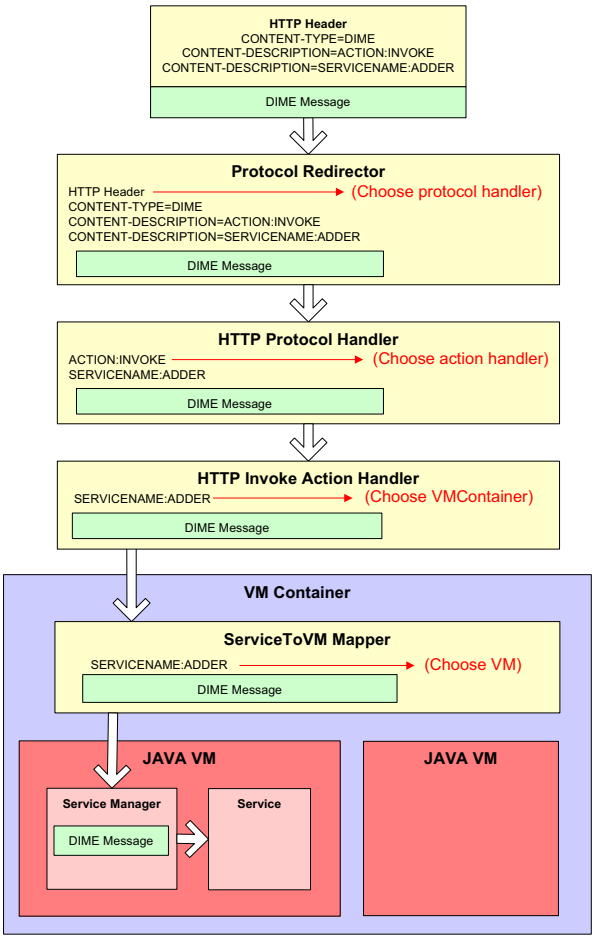

Figure 3: Service Invocation With AgServe

Upon completion of the invocation request, the result is encoded and handed back to the Service Manager, who pre-appends the proper HTTP headers and sends the response back to the client. Once the response is received on the client side, it is decoded and passed back to the application.

V. Experimental Results

Two experiments were conducted to measure the performance of service-oriented architectures with AgServe. Java Remote Method Invocation (RMI) is used as the baseline. While RMI does not provide the capabilities such as language independence, WSDL, etc., it serves as a useful measure of an efficient implementation of remote invocation.

The first experiment measured time to complete a remote service invocation with a service that, given a square matrix, returns the inverse of the matrix. Square matrices of sizes 1, 10, 100 and 200 were used to perform the test. The second experiment measures network bandwidth utilization.

Table 1 shows the performance comparison between AgServe (shows as AgS in the table) and RMI. For this test, three different scenarios were used to measure the performance. First, both client and service were executed in the same host. Next, two tests were performed between two different hosts, first with a 100 Mbps Ethernet connection, and second with an 802.11b wireless ad-hoc connection. The computers used for this experiment were two IBM ThinkPad T42 series laptop computers running Windows XP SP2.

N	Localhost		802.3 100M		802.11b Ad Hoc	
	AgS	RMI	AgS	RMI	AgS	RMI
1	1.91	0.9	2.3	1.21	17.93	2.6
10	2.11	1.4	2.9	1.7	20.03	6.31
100	31.88	24.97	48.97	30.94	388.16	394.97
200	189.62	142.8	247.86	168.94	1584.08	1619.33

Table 1: Invocation Times (in milliseconds) . Comparison between AgServe and RMI, matrix of size N

The results show that RMI is indeed faster than AgServe in many circumstances. For example, with a matrix of size 10 on an 802.11 Ad-Hoc network, RMI is 3.17 times faster. However, with a matrix of size 100 or 200, the performance is approximately the same.

The second experiment measures the network utilization of AgServe in contrast to RMI. Again, matrices of sizes 1, 10, 100 and 200 were used for the test. The test was executed a single time. In order to measure the network utilization, a custom SocketFactory was used that measured the number of bytes sent and received by the application.

Table 2 shows the results of the experiment. The network utilization of AgServe is slightly lower than that of RMI.

N	Object Size	Write		Read	
		AgS	RMI	AgS	RMI
1	128	385	642	239	453
10	1010	1266	1525	1124	1336
100	8110	81367	81625	81225	81436
200	322110	322368	322625	322226	322436

Table 2: Network utilization (in bytes) Comparison between AgServe and RMI for service invocation of a matrix with size N

VI. Conclusions and Future Work

This paper has described the motivation, design, and implementation of AgServe – an infrastructure that supports service oriented architectures on top of the agile computing middleware. The middleware is opportunistic in discovering and using computation, communication, and storage resources dynamically. The continuously optimizing nature of agile computing provides a good foundation for building self-healing and autonomic systems.

Future work includes further optimization of the AgServe infrastructure, implementation of service migration, and further experimental analysis.

ACKNOWLEDGEMENT

This work is supported in part by the U.S. Army Research Laboratory under Cooperative Agreement W911NF-04-2-0013, by the U.S. Army Research Laboratory under the Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-2-0009, and by the Air Force Research Laboratory under Cooperative Agreement FA8750-06-2-0064.

REFERENCES

[1] Bradshaw, J. M., A. Uszok, et al. (2003). Representation and reasoning for DAML-based policy and domain services in KAoS and Nomads. Proceedings of the Autonomous Agents and Multi-Agent Systems Conference (AAMAS 2003), Melbourne, Australia, New York, NY: ACM Press.

[2] Uszok, A., J. M. Bradshaw, et al. (2004). "KAoS policy management for semantic web services." IEEE Intelligent Systems **19**(4): 32-41.

[3] Suri, N. Bradshaw, J.M., Breedy, M.R., Ford, K.M., Groth, T., Hill, G.A., and Saavedra, R.: "State Capture and Resource Control for Java: The Design and Implementation of the Aroma Virtual Machine." USENIX Java Virtual Machine (JVM'01) Conference.

[4] Binder, W. and Hulaas, J. A Portable CPU-Management Framework for Java. IEEE Internet Computing. Vol. 8, No. 5, pp. 74-83.

[5] IBM. Jikes Research Virtual Machine. Available online at: http://jikesrvm.sourceforge.net/.

[6] Carvalho, M., Suri, N., Arguedas, M. (2005) *Mobile Agent-based Communications Middleware for Data Streaming in the Battlefield*. In Proceedings of the 2005 IEEE Military Communications Conference (MILCOM 2005), October 2005, Atlantic City, New Jersey.

[7] World Wide Web Consortium. Web Services Description Language (WSDL) 1.1. Available online at: http://www.w3.org/TR/wsdl.

[8] OASIS-Open Organization. Introduction to UDDI: Important Features and Functional Concepts. Available online at: http://uddi.org/pubs/uddi-tech-wp.pdf.

[9] OASIS-Open Organization. UDDI Version 3.0.2. Available online at: http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm.

[10] Cremonese, Piergiorgio and Veronica Vanni. UDDI4m: UDDI in Mobile Ad Hoc Network. In Proceedings of the Second Annual Conference on Wireless On-demand Network Systems and Services (WONS'05).

[11] Engelstad, P.E., and Y. Zheng. Evaluation of Service Discovery Architectures for Mobile Ad Hoc Networks. In Proceedings of the Second Annual Conference on Wireless On-demand Network Systems and Services (WONS'05).