Toward Dynamic Application Protocols in Heterogeneous Distributed Computing Systems

Dennis Patrone
Bina Ramamurthy
Department of Computer Science and Engineering
University at Buffalo
{dpatrone, bina} @ cse.buffalo.edu

Abstract

In order to achieve the full transformational promise of Sea Power 21 and net-centric warfare, a highly efficient and adaptable service oriented architecture (SOA) is required. We propose an XML-based language to enable heterogeneous distributed systems to dynamically modify the application-layer network protocol in an SOA at runtime and without individual service-client agreements. We argue that an SOA that supports dynamic protocols can operate more efficiently and is more agile and spontaneous than an architecture based on standardized messaging such as Web Services. We provide an overview of the XML-based language called the Application Logic Markup Language (ALML) and our innovative approach to an SOA that supports multiple protocols in heterogeneous systems; ALML lays the foundation for a truly dynamic, interoperable, and adaptable SOA.

1. Introduction

From the remote procedure call (RPC) systems of the mid-1970's and 1980's, through the Common Object Request Broker Architecture (CORBA) in the 1990's, to the concept of Web services today (including SOAP-based, XML-RPC-based, and those following REST principles), the main concept underlying most heterogeneous distributed computing systems (DCS) has remained largely unchanged for three decades or more: they achieve inter-process communication via standardized message passing [1]. The benefit is that clients and services can statically link in libraries that produce and consume the standardized messages and are able to communicate with any other process that also produces and consumes the messages. The messages are simply a standardized application-layer protocol for exchanging data and requests.

Any protocol will have biases introduced at design time. Protocol designers must make decisions that will affect attributes such as complexity (in both space and time) and extensibility. Any large-scale, standardized protocol must favor generality over efficiency. The protocol will be designed for the average case and will be unable to take advantage of (or compensate for) application-specific knowledge about specialized processing, component capabilities, expected usage patterns, data priority, etc. This deficiency is magnified in DCS such as sensor networks that operate over unreliable and low bandwidth connections, with limited hardware, and using battery power.

We propose an architecture that will allow DCS to move away from static, standardized messaging and toward dynamic application layer protocols. This flexibility will allow systems to tune the application layer protocols to their specific requirements and constraints.

For the remainder of this paper, we will refer to the application layer protocol simply as the "protocol". We will focus on improving the application layer exchange. We assume lower-level protocols that provide connectivity between two executing processes over a communications link are pre-established.

Section 2 provides the motivation for our approach that is explained in Section 3. The application logic markup language (ALML) that supports the dynamic logic sharing is explained in Section 4. Section 5 discusses the performance improvement possible with our approach. Section 6 compares the related work, followed by future directions in Section 7 and a summary in Section 9.

2. Motivation

Auerbach and Russell [2] divided distributed computing systems into two broad categories: distributed programming languages and interface definition language (IDL)-based programming. In both cases, the system consists of a marshaling subsystem. IDL-based systems translate the user-defined interface into the marshalling subsystem. Distributed programming languages often automate the marshalling within the language or with a language extension. For the purpose of this paper we will not concern ourselves with distributed programming languages. In general, distributed programming languages only interoperate with other systems implemented in the same distributed language; in this paper we are concerned with heterogeneous systems.

While the protocols and implementations may vary across IDL-based systems, high-level views of the architectures look quite similar. In general, an interface is defined for some capability or "service". From that interface server-side and client-side libraries are created. Figure 1 shows the general flow of information for a single request in an IDL-based DCS system. The arrows are numbered in increasing order in which information flows. The lighter gray area on the left represents the client-side library while the darker gray area on the right represents the server-side library. The client-side library contains stubs that implement the service interface by marshalling requests to the server-side and unmarshaling the server-side responses for the calling client code. In this paper, we will refer to the client-side marshalling subsystem as the "*proxy*" since it acts on behalf of the service within the client's address space. The server-side library listens for client requests, unmarshals those requests, invokes code provided by the service implementer to actually handle the requests, and marshals the responses back to the client. These libraries are normally automatically generated by a tool that processes the interface definition.

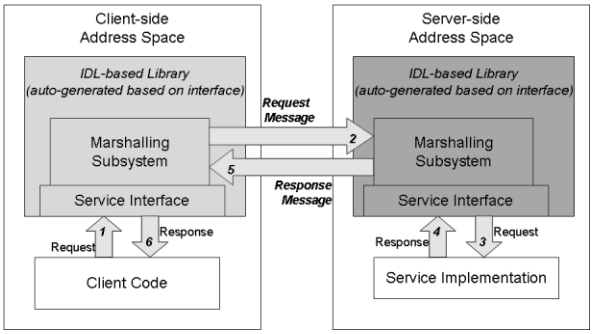

Figure 1: General IDL-based DCS request

Since the messages flowing between the separate address spaces are standardized, heterogeneity is achieved. It does not matter what hardware, operating system, or language exists on either side. As long as both sides adhere to the message specification they will be able to communicate with each other. Each side statically links to the appropriate library to manage the exchange of messages.

This interoperability comes at a price. Since the messages are standardized and based on the interface, changing what flows between the address spaces requires a change in the interface itself. This change affects all previously developed service implementations and client applications. Furthermore the message passing standards must favor generality over other considerations such as efficiency and performance in their definitions.

Conversely, an architecture that enables dynamic protocols will allow services and clients to determine the best protocol at runtime based on the current operating context. The *in situ* values of available attributes can be considered and any controllable aspect can be optimized. These factors may include the participating components' capabilities and limitations, current network topology and load, the overall system's goals and constraints, task priorities, latency thresholds, etc. By tuning protocols to remove inefficiencies in the process, overall system performance can be improved. Also, by allowing updates to protocol after deployment of services and clients, the DCS will be more capable of adapting to unanticipated changes in system goals, new users and uses, and technology advances. For example, Liu and Martonosi [3] demonstrated in simulation how dynamic protocols can improve the routing performance and energy efficiency in sensor networks.

At the same time, we need to maintain the loose coupling of services and clients in traditional service oriented architectures (SOA). Potential protocols must not be required to be negotiated up front nor is it acceptable to require each service and client to implement a bank of protocol handlers for every potential partner with which they would like to communicate.

3. Our Approach

Our approach to allow dynamic protocols while maintaining loose coupling among clients and services is to allow services to inject their own proxy definitions into clients at runtime. These proxy definitions can contain logic to implement a single, service-defined protocol, or negotiate with the service to choose from a

range of protocols as appropriate. The service can also inject updated and completely new proxies into the client over the life of their collaboration, thus allowing the protocol to change even after the initial negotiation.

Client applications are developed against a service interface like traditional IDL-based DCS. However the application is not linked with a static, auto-generated library that contains the proxy definition. The details of how the client will communicate with the server are left empty; a library that is capable of downloading and installing executable code is provided instead. At runtime, when the client application finds a *specific* service implementation that is needed, the client-side library downloads the proxy's details from that service. The service is free to provide its "best" available protocol handler to the client based on the current operating environment.

This is similar to the approach taken in the Jini Network Technology architecture [4]. The main disadvantage of Jini is that is requires Java. The proxy definitions shared from services to clients in Jini is compiled Java Virtual Machine (JVM) bytecode. Our approach differs in that we intend to provide an XML-based language to define the proxy. By capturing the logic of how to communicate with the service in a language and (virtual) platform independent way, we can extend the range of clients we can support to any system that can parse and process XML. The next section will detail our XML-based language.

4. Application Logic Markup Language

As a first step toward enabling dynamic proxies, we are proposing an XML-based language that can be used to describe application logic. We call this language the Application Logic Markup Language (ALML). Services will define their proxies for clients in this language.

ALML is actually an XML Schema Definition (XSD) that defines an object oriented language. That language is heavily influenced by Java's syntax and terminology. We currently have a proof-of-concept implementation of a library that processes ALML in Java. The implementation is capable of importing an ALML-compliant XML specification and executing the logic contained therein within a JVM. The concepts are translatable across other contemporary object oriented languages and we intend to provide mappings for multiple languages in the future.

The ALML language defines standard object oriented constructs and attributes. There are packages which are collections of classes. Classes are collections of member variables (data) and member methods (operations). Methods are collections of statements. Like Java, ALML supports single inheritance and multiple interface implementations. ALML defines specific primitive types and sizes that are mapped to language-specific types. Currently ALML does not support more complex constructs such as anonymous classes, inner-classes, or generics.

Due to space constraints we are unable to present the entire language here. Instead we will provide a few examples to show the "flavor" of the language. We have also left off some details to keep the example sizes small; we do not expect these modifications will impact the utility of the examples.

4.1. Class definition

A class in ALML has the following attributes: a name, a package, a visibility declaration, a flag to determine if the class is final and a flag to determine if the class is transferrable. The name and package are used to uniquely identify the class definition. The visibility determines what other classes have access to this class (any class, only classes within the same package, etc). The final flag is used to determine if the class can be extended by other classes or not. The transferrable flag is used to determine if instantiated objects of this class type can be transferred across the network. There is also a version specification on the "almlDeclaration" element that wraps the class definition that is used to determine when a class definition has been modified.

A class definition consists of a list of all of the interfaces it implements, its parent class (if any), and sets of constructors, member variables, and member methods, and optionally a single destructor and static initializer method. An excerpt of the ALML XSD for class definitions is shown in listing 1.

```
<xs:complexType name="classDef">
  <xs:sequence>
    <xs:element name="uses" type="usesDef"
        minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="extends" type="identifierDef"
        minOccurs="0" maxOccurs="1" />
    <xs:element name="implements" type="identifierDef"
        minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="constructor"
        type="constructorDef" minOccurs="0"
        maxOccurs="unbounded" />
    <xs:element name="destructor" type="destructorDef"
        minOccurs="0" maxOccurs="1" />
    <xs:element name="variable" type="variableDef"
        minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="method" type="methodDef"
        minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="abstractMethod"
        type="abstractMethodDef"
        minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="staticInitializer" type="staticDef"
        minOccurs="0" maxOccurs="1" />
  </xs:sequence>
  <xs:attribute name="name" type="identifierDef"
      use="required" />
  <xs:attribute name="package"
      type="packageIdentifierDef" use="required" />
  <xs:attribute name="visibility" type="visibilityDef"
      use="required" />
  <xs:attribute name="final" type="xs:boolean" />
  <xs:attribute name="transferrable" type="xs:boolean"
      default="false" />
</xs:complexType>
```

Listing 1: ALML XSD Class Definition

A partial definition of the class "ExampleClass" defined in ALML XML is shown in listing 2.

```
<?xml version="1.0"?>
<almlDeclaration
   xmlns="http://www.cse.buffalo.edu/alml"
   … (other schemas)
   version="1.0.0">

 <class name="ExampleClass"
   package="example.pkg"
   visibility="public" final="true">
   <uses name="ServiceInterface"
      package="another.pkg" />
   <implements>ServiceInterface </implements>
   …
```

Listing 2: An example ALML class declaration

4.2. Method definition

A method in ALML has the following attributes: a name, visibility, and flags to determine if the method is final, abstract, or static. The name is used to uniquely identify the method within the class definition. The visibility defines what other methods can invoke this method (any method, only methods within this same class, etc). The final flag is used to determine if the method can be overridden by subclasses or not. The static flag determines if the method is associated with the class definition or individual instances of the class.

A method consists of a signature and a statement block. The signature for a method is the name and visibility, a flag if it is static (associated with the class), a flag if it is final (cannot be overridden), the return type, a list of parameters, a list of possible exceptions thrown. An excerpt of the ALML XSD for method definitions is shown in listing 3.

```
<xs:complexType name="methodDef">
  <xs:sequence>
    <xs:element name="signature"
      type="methodSignatureDef" />
    <xs:element name="block" type="blockDef" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="methodSignatureDef">
  <xs:sequence>
    <xs:element name="type" type="typeDef" />
    <xs:element name="parameter"
      type="parameterDef"
      minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="throws" type="identifierDef"
      minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="name" type="identifierDef"
      use="required" />
    <xs:attribute name="visibility" type="visibilityDef"
      use="required" />
    <xs:attribute name="static" type="xs:boolean" />
    <xs:attribute name="final" type="xs:boolean" />
</xs:complexType>
```

Listing 3: ALML XSD Method Definition

A partial example of a method declaration in ALML XML is provided in listing 4. The example defines a public method called "getValue" that returns a 64-bit floating point number. The method has a single Boolean "flag" parameter and does not throw any declared exception.

```
<method>
  <signature name="getValue" visibility="public">
    <type>
      <primitive>float64</primitive>
    </type>
    <parameter name="flag"> <type>
        <primitive>boolean</primitive>
      </type> </parameter>
  </signature>
  <block>
    <statement>…. </statement>
  </block>
</method>
```

Listing 4: An example ALML Method Definition

4.3. Statement definition

A statement in ALML is a choice between one of the many different statement types that ALML supports. An excerpt of the ALML XSD for statement definitions is shown in listing 5.

```
<xs:complexType name="statementDef">
  <xs:choice>
    <xs:element name="variable" type="variableDef" />
    <xs:element name="expression"
      type="expressionDef" />
    <xs:element name="if" type="ifDef" />
    <xs:element name="for" type="forDef" />
    <xs:element name="while" type="whileDef" />
    <xs:element name="try" type="tryDef" />
    <xs:element name="return" type="expressionDef" />
    <xs:element name="throw" type="expressionDef" />
    <xs:element name="noop" type="empty" />
    <xs:element name="break" type="empty" />
    <xs:element name="assign" type="assignDef" />
  </xs:choice>
</xs:complexType>
```

Listing 5: ALML XSD Statement Definition

Expanding down one level from "statementDef", listing 6 shows the ALML XSD definition for an "if" statement. An "if" statement in ALML is comprised of a test condition and its consequence, an optional list of additional test conditions and their associated consequences (i.e., the "else if" clause), and finally an optional consequence if all other test conditions have failed (i.e., the "else" clause).

```
<xs:complexType name="ifDef">
  <xs:sequence>
    <xs:element name="condition"
      type="expressionDef" />
    <xs:element name="then" type="blockDef" />
    <xs:element name="elseIf" type="elseIfDef"
      minOccurs="0" maxOccurs="unbounded" />
    <xs:element name="else" type="blockDef"
      minOccurs="0" maxOccurs="1" />
  </xs:sequence>
</xs:complexType>
```

Listing 6: ALML XSD "If" Statement Definition

5. Improving Performance through Dynamic Protocols

As demonstrated in [3], this approach has the potential to provide a number of benefits to DCS and sensor-based networks. Protocols can be tuned to minimize utilization of scarce resources at the desecration of each individual service rather than at a global level. For example, a service implemented with a slow, unreliable connection might compress data prior to transmission to minimize the network traffic. Another service with a limited CPU might forego the compression to free the CPU to perform other tasks. A client can connect to either service and will be unaware and uninterested if the proxy code it has downloaded and is executing locally is uncompressing the data stream or not.

Another advantage of systems built on such an architecture involves the reduced service latency for clients. The service could define the client-side proxy to cache the last service-provided value for some time window, e.g., based on the service's update frequency. If the client requests the "current" value within that window, its proxy can immediately return the cached value since there will be no more current information available from the actual service. Not only has network bandwidth and service CPU, memory, and battery been conserved, but the client will presumably receive its response significantly faster than requiring a round-trip to the service only to find out the client already has the most current information.

We hypothesize that a distributed system architecture that supports sharing application logic will lead to systems that are more efficient, scalable, and adaptable than systems based on standardized network protocols. Distributed systems based on standardized network protocols by definition cannot modify network communications. These systems utilize compile-time bound client-side proxies—the code that communicates with the server over the network on the client's behalf. They are bound to the client at compile time since the proxy details are completely defined as part of the architecture's protocol standardization. Distributed systems that are capable of sharing application logic at runtime (mobile code) can exploit situational knowledge by deferring client-side proxy implementation details until runtime and changing them at any time. The details can then be tuned to any specific criteria which is important to the specific system (e.g., minimize network utilization, reduce latency, improve scalability).

5.1 Improving efficiency

Efficiency can be measured in a number of different ways: network throughput, CPU utilization, memory footprint, power consumption, etc. Every application will have its own criteria and thresholds for acceptable performance across scarce resources. Every protocol and usage pattern will affect different performance measurements uniquely. Selecting one generalized, standardized protocol will introduce a design bias and cannot possibly efficiently cover every situation. A system which allows services to provide mobile code to

define smart proxies at runtime enables the system to react and adjust to changing situations. For example, proxies can be configured to batch specific requests instead of sending each request immediately without changing the service interface.

A mobile code-based architecture does not automatically solve the efficiency problem but rather provides a framework in which system designers and software engineers are able to solve distribution issues in a domain-specific and application-aware way.

There is extra cost associated with setting up a connection in a mobile code-based architecture, especially if that code is interpreted. There are additional up-front costs with respect to the network as the proxy code is downloaded. There are additional costs to CPU usage as the proxy code is compiled or interpreted on the first time it is used. There is also additional latency introduced in the initial call as the mobile definition is downloaded and interpreted. These costs, however, should be incurred only once and therefore are independent of *n*. Furthermore, in special circumstances system clients could also be seeded with initial proxies to further reduce initial connection costs when required.

5.2 Improving scalability

Tuning application-level protocols based on system usage can reduce demands placed on specific resources and/or distribute the processing of a distributed system, allowing the same software and hardware to provide greater capacity. Even so, if that capacity is reached, mobile code-based smart proxies can continue to help by providing the ability to redirect entire services, individual methods, and even specific requests across multiple servers helping systems achieve scalability. These redirections can be based on past performance, current situations, or overall system goals—and all managed internally to the smart proxy on the client, but without tight client coupling. In some instances, the smart proxies can contain the entire logic to perform algorithmic-based services, further distributing the execution of a service across client machines, but maintaining the algorithm implementation definition at the service. If the algorithm needs to be updated in the future, the single service implementation is updated and the new algorithmic solution will be automatically pushed to all clients. This maintains the service distribution while helping with system maintenance.

5.3 Improving adaptability

Mobile code-based architecture has the potential to enable smart proxies. These smart proxies, in turn, enable a distributed system to adjust runtime, application-level protocols to changing system demands. Section 5.2 describes how a mobile code-based architecture can also aid in adaptability when a service method or set of methods have algorithmic solutions. The algorithmic solution can be provided in the proxy, removing the need for any client-side communication back to the original service. The service is in effect "teaching" clients how to perform the service for themselves. This is very adaptable as clients can learn new "skills", and services can automatically update clients with improved algorithmic solutions, code that can manage additional requirements or to handle emerging issues by changing the proxy implementation being provided.

6. Related Work

There has been work performed related to enabling dynamic protocols in DCS and sensor networks specifically.

The Web Service Invocation Framework (WSIF) ([5], [6]) defines a framework that abstracts protocol handling for clients. In theory, this architecture could be used to integrate clients with services using dynamic protocols. However, the definition of the protocol handlers is expected to be provided as a Java JAR file and the implementation is a Java Application Programming Interface (API). Therefore the client (at least the part that interacts with the framework) will have to be implemented in Java even though the protocols that the client uses may be language independent.

Mckee [7] and more recently Aberer, Hauswirth, and Salehi [8] provide different views on how XML can be used to define the semantics of individual service calls (i.e., describe *what* a service can do for a client). The purpose is to enable a DCS that can autonomously orchestrate interactions with services and clients in various ways. However the descriptions stop short of containing *how* the client should communicate with the provider to perform the service, thus suffering from the same drawbacks as general IDL-based DCS once the messaging chain is configured.

A Web Service-based approach for sensor networks is presented by Priyantha *et al* [9]. Being an IDL-based DCS, the approach can suffer from the same issues raised in our paper. However they also provide evidence that XML processing is possible in resource-limited devices. Using their XML processing approach, coupled with other techniques such as just-in-time compiling and smart versioning techniques like those in discussed by Liu and Martonosi [3], our

approach could be extendable to resource-limited devices on the network's "edge" where efficiency and adaptability are generally a more critical issue than for machines with greater capability.

Gibbons, *et al* [10] present work in enabling large-scale sensor networks which include the concept of uploading executable to sensors in the form of sensor data filters which they call "senselets". Their architecture is focused on sensor-based data collection and efficient distributed querying. Our goal is to provide a more generalized architecture that can be used for a wide range of distributed tasks.

Possibly the most closely-related work to ours is the Impala system [3]. Impala is a network architecture that allows for software updates including the modules that implement network communications (i.e., the proxy). However, Impala shares compiled logic which implies that there is some knowledge specifically about which clients will need to communicate with which services. This knowledge would be needed so that the service-specific proxies could be compiled to the necessary client architectures. Our approach differs by defining the proxy logic in a platform- and language-independent way so it can but uploaded to any (potentially unanticipated) client.

Waldo [4] presents a compelling case for an architecture where each individual service provider defines the "best" protocol they can and, at runtime, provides any interested client the necessary code to communicate with that protocol. However, the Jini architecture is defined specifically for the Java programming language. While there are ways of integrating this architecture with other languages, it is our goal to provide an architecture that integrates directly with the native language and platform of the client, thus reducing the complexity and overhead involved in communicating within a heterogeneous DCS.

XML-based programming languages have been proposed by others as well (e.g., [11], [12]) but they are targeted toward their own specific purposes and their own runtimes rather than being a general-purpose, portable application logic description language. We are attempting to develop enough capability in ALML to support sophisticated proxy definitions but need to constrain the language to constructs that can be mapped and supported in multiple target programming languages. When the concepts in ALML are more mature, we will revisit existing XML languages and other similar approaches to see what definitions and standards can be leveraged.

As a primarily XML-driven process, we expect to be able to "borrow" a number of concepts and advances from the Web Services realm directly or without much customization. For example, we expect to be able to easily use SSL for secure communications, a binary XML representation (e.g., [13], [14]) to reduce the overhead associated with XML transfer and parsing when necessary, and potentially use the Web Service Description Language (WSDL) [15] directly as our service interface description language.

7. Future Work

This is a work-in-progress. To move ALML out of the "toy system" realm and into a viable architecture, we need to expand the capabilities of the ALML language. We also must support ALML in multiple programming languages; we are targeting the .NET platform next. To realize the full potential and vision of the ALML language we will determine how ALML can be incorporated into an SOA such as one of those referenced in this paper. This work is currently underway in parallel with evolving the language.

Security in this type of architecture will be critical. We expect to be able to adapt a number of security strategies from web service security models and existing mobile code architectures. Secure socket layer communications, digitally signed exchanges, and fine-grained control over access to critical system components and data through "sandboxing" downloaded code are a few of the concepts we will be pursuing.

Communication between address spaces is generally orders of magnitude slower than local access. Clearly adding another layer of abstraction on top of an already inherently slow process will not be without additional cost. However, we feel that the long-term benefits of using this approach—including overall reduction in network, CPU, and battery utilization, and increases in service response times—will far outweigh the initial (and constant) cost of downloading and interpreting the XML-based definition of the proxy once. Furthermore, techniques such as seeding the initial proxy definition on the initial clients prior to deployment, just-in-time compiling of ALML, and intelligent caching and versioning of previously loaded proxy definitions can further reduce the overhead associated with such an architecture. We can also investigate other, more compact and processed representations of application logic such as using abstract syntax trees. To what extent these techniques can reduce the architecture's overhead is another evaluation to be performed.

8. Summary

In this paper, we have argued that a DCS architecture that supports dynamic protocols can operate more efficiently than an architecture based on standardized messaging. We have provided a brief overview of an XML-based language called ALML that we feel will enable a DCS that supports multiple protocols in heterogeneous system. By defining IDL-based proxies in ALML, services can inject the protocol handling details into heterogeneous clients at runtime. This will allow services to take advantage of application-specific knowledge and, based on system goals, constraints, and performance attributes, tune network protocols to achieve greater system performance.

9. References

[1] Andrews, G. "Paradigms for process interaction in distributed programs," *ACM Computing Surveys*, Vol 23, No 1. March 1991.

[2] Auerbach, J., Chu-Carroll M. "The Mockingbird System: A Compiler-based approach to maximally interoperable distributed programming," Research Report RC 20718, IBM T. J. Watson Reasearch Center, February 1997.

[3] Liu, T., Martonosi, M. "Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems," *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2003.

[4] Waldo, J. "The End of Protocols," Available at http://java.sun.com/developer/technicalArticles/jini/protocols.html. Accessed on: November 16, 2008.

[5] Duftler, M., Mukhi, N., Slominski, A., Weerawarana, S. "Web Services Invocation Framework (WSIF)", *OOPSLA Workshop on Object Oriented Web Services*, 2001.

[6] Mukhi, N., Khalaf, R., Fremantle, P. "Multi-protocol Web Services for enterprises and the Grid," In *Proceedings of the EuroWeb 2002 Conference on the Web.* 2002.

[7] Mckee, P., Marshall, I. "Behavioural specification using XML," In *Proceedings of the 7th IEEE workshop on Future Trends of Distributed Computing Systems - FTDCS'99*. IEEE Computer Society Press, pp 53-59.

[8] Aberer, K., Hauswirth, M., Salehi, A. "A Middleware For Fast And Flexible Sensor Network Deployment," *VLDB '06*, September 2006, Seoul, Korea, ACM, 2006, pp 1199-1202.

[9] Priyantha, B., Kansal, A., Goraczko, M., Zhao, F. "Tiny Web Services: Design and Implementation on Interoperable and Evolvable Sensor Networks," *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2008.

[10] Gibbons, P., Karp, B., Ke, Y., Nath, S., Seshan, S. "IrisNet: An Architecture for a Worldwide Sensor Web," *Pervasive Computing*, 2(4):22-33, 2003.

[11] Plusch, M. and Fry, C. *Water: Simplified Web Services and XML Programming*. John Wiley & Sons, Inc. New York, NY, 2003.

[12] Klang, M. "XML and the art of code maintenance". *Extreme Markup Languages. Proceedings of*, 2003.

[13] Sandoz, P., Triglia, A., Pericas-Geertsen, S. "Fast Infoset". Available at: http://java.sun.com/developer/technicalArticles/xml/fastinfoset/. Accessed on: November 16, 2008.

[14] Schneider, J., Kamiya, T. (ed). "Efficient XML Interchange (EXI) Format 1.0." W3C Working Draft 19 September 2008.

[15] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S. "Web Services Description Language (WSDL) 1.1." W3C Note 15 March 2001.