Analytical Study of Object Components for Distributed and Ubiquitous Computing Environment

USHA BATRA DEEPAK DAHIYA and SACHIN BHARDWAJ
Department of CSE&IT,
Institute of Technology and Management,
Gurgaon 122017, India
batrausha@rediffmail.com, deepak.d@acm.org, sachin.dsu@gmail.com

Abstract: - The Distributed object computing is a paradigm that allows objects to be distributed across a heterogeneous network, and allows each of the components to interoperate as a unified whole. A new generation of distributed applications, such as telemedicine and e-commerce applications, are being deployed in heterogeneous and ubiquitous computing environments. The objective of this paper is to explore an applicability of a component based services in ubiquitous computational environment. While the fundamental structure of various distributed object components is similar, there are differences that can profoundly impact an application developer or the administrator of a distributed simulation exercise and to implement in Ubiquitous Computing Environment.

Key-Words: - Ubiquitous Computing, one.world, COM, DCOM, RMI, CORBA, SOAP

1 INTRODUCTION

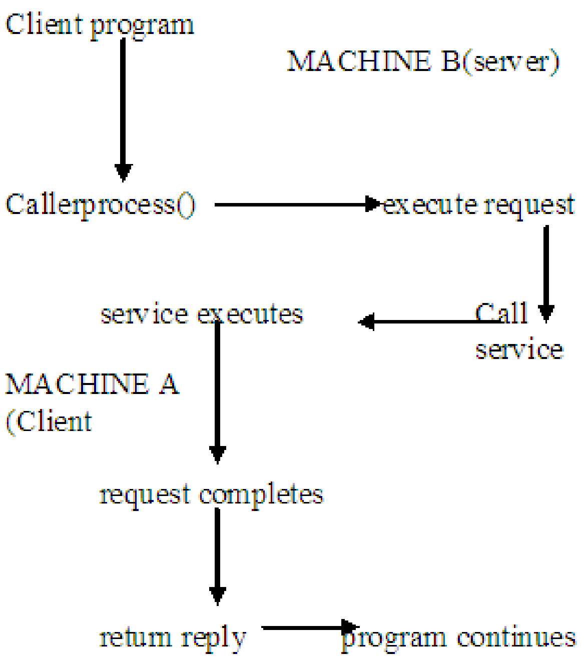
The overall technical goal of distributed object computing is to advance distributed information technologies so that they may be more efficient and flexible, yet less complex. The benefits of distributed objects are indeed solutions to the problems with existing monolithic client / server paradigms. The most advantageous aspect of this component oriented services framework is to facilitate developing various service applications for ubiquitous computation as the occasion demands. That is, service applications are modeling, simulation, monitoring, web services and others.
Web Services [1, 2, and 3] present another alternative distributed computing infrastructure that is being promoted to the use of distributed object components such as RMI or CORBA and SOAP.

Web Service implementations support different client-side application programmer interfaces; client code may work by constructing “call” objects that are dispatched to the server , or may use a higher level interface that hides the communication level entirely through the use of client-side stub objects with an operational interface that mimics that of servers. The client-stub approach results in code that is very much similar to Java-RMI, CORBA or SOAP clients. Each of these three architectures for distributed computing has a fundamental world view that affects the structure of its architecture. Software engineers developing ubiquitous computing [4] applications have a number of choices of platforms on which to build. This paper compares and contrasts the programming models and capabilities offered by these platforms relative to one.world, our chosen platform.

2 RPC

RPC is a powerful technique that provides a remote procedure infrastructure for building distributed, client-server based applications. We have two processes in a RPC call: Client Process and Server Process. The caller process (i.e. Client) sends a message to the Server process and waits (blocks) for a reply message. The call message contains the procedure's parameters (among other things), and the reply message contains the procedure's results (among other things). When the reply message returns, the caller extracts the results of the procedure and resumes execution.The two processes may be on the same system, or they may be on different systems with a network connecting them. RPC [5, 6, and 7] is based on enhancing the notion of conventional or local procedure calling, so that the called procedure (server) need not exist in the same address space as the calling procedure (client). By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC

isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.
But why would someone want to execute a procedure on another person's machine?



The following scenario is realistic. A caller in India wants to insert a tuple into an PostgreSQL database in Germany and has no local installation of PostgreSQL. A client could call a remote procedure to do the job. Provided the remote machine has a valid installation of PostgreSQL, an insertion could be accomplished. A Server RPC could be set up in Germany to accept an SQL statement from the client, execute it, and then return the result.
RPC's essential concept is hiding all the network code in the stub procedures. The goal of the RPC is to make the writing of distributed applications easier. RPC transparency allows such; however, these transparencies raise other problems.

RPC Transparency Issues:

Parameter passing - Passing parameters across process/host boundaries is surprisingly tricky. Parameters that are passed by value are fairly simple to handle: The client stub copies the value from the client and packages into a network message._ Parameters passed by reference are much harder. e.g., in C when the address of a variable is passed. e.g., passing arrays or more generally, handling pointer-based data structures e.g., pointers, lists, trees, stacks, graphs, etc. Typical solutions include:{ Have the RPC protocol only allow the client to pass arguments by value. However, this reduces transparency even further. RPC facilities typically provide an interface definition language" to handle this. e.g., CORBA or DCE IDL

Binding – Binding is basically locating the remote host, or finding the correct server process. Binding is the process of mapping a request for a service onto a physical server somewhere in the network. Typically, the client contacts an appropriate name server or \location broker" that informs it which remote server contains the service

Exception Handling - Client or server crash detection. With a local procedure call there are a limited number of things that can go wrong, both with the call/return sequence and with the operations e.g., invalid memory reference, divide by zero, etc.

Call Semantics - When a local procedure is called, there is never any question as to how many times the procedure executed. With a remote procedure, however, if you do not get a response after a certain interval, clients may not know how many times the remote procedure was executed. Call Semantics tell us how many times has the procedure actually executed? At least once and at most once semantics are available. When an RPC can be executed any number of times, with no harm done, it is said to be idempotent. i.e., there are no harmful side effects. Some examples of idempotent RPCs are: Returning time of day, calculating square root. Reading the 512 bytes of a disk file, returning the current balance of a bank account etc. Some non-idempotent RPCs include: A procedure to append 512 bytes to the end of a file, a procedure to subtract an amount from a bank account.

Data Representation - Different machine architectures implement data in different formats.

Performance - Usually the performance loss in RPCs VS regular procedure calls is a factor of ten due to

1. Protocol processing

2. Context switching

3. Data copying

4. Network latency

5. Congestion

Security - Remember, RPCs are usually executed across networks and the data is subject to public scrutiny.

RPC implementations are nominally incompatible with other RPC implementations, although some are compatible. Using a single implementation of a RPC in a system will most likely result in a dependence on the RPC vendor for maintenance support and future enhancements. This could have a highly negative impact on a system's flexibility, maintainability, portability, and interoperability.

Because there is no single standard for implementing an RPC, different features may be offered by individual RPC implementations. Features that may affect the design and cost of a RPC-based application include the following:

- support of synchronous and/or asynchronous processing
- support of different networking protocols
- support for different file systems
- whether the RPC mechanism can be obtained individually, or only bundled with a server operating system

Because of the complexity of the synchronous mechanism of RPC and the proprietary and unique nature of RPC implementations, training is essential even for the experienced programmer.

Other middleware technologies that allow the distribution of processing across multiple processors and platforms are

- Object Request Brokers (ORB)
- Distributed Computing Environment (DCE)
- COM/DCOM (see Component Object Model (COM), DCOM, and Related Capabilities)
- Transaction Processing Monitor Technology
- Three Tier Software Architectures

3 COMPONENT BASED DEVELOPMENT

When Microsoft designed COM[8,9] (Component Object Model) , which serves as a basis for DCOM (Distributed Component Object Model) and COM+, the main standard in the world of object languages was C++. COM was designed as a runtime specification. COM provides the component technology for Microsoft Windows Distributed interNet Applications (Windows DNA) architecture, which enables developers to integrate Web-based and client-server applications in a single, unified architecture. Using COM, developers can create distributed components that are written in any language and that can interact over any network. The Distributed Component Object Model (DCOM) is a protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner. Based on C++, COM is a complete specification at binary level and nothing prevents other languages from producing or calling COM objects.

Developing COM components is fairly complicated. The web services require a protocol which supports characteristics that DCOM does not have, thus DCOM huge failure to be used on web. COM does not offer a suitable modern flexible infrastructure that supports business object system for web services type applications that require rich run time with a real garbage collector allowing object/ relational mapping.

The Common Object Request Broker Architecture (CORBA) is a non-commercial venture, created at the beginning of the 90s by the Object Management Group (OMG), an imposing consortium of over 800 members. It is the oldest and perhaps the most mature architecture of its time. CORBA is an extremely large and complex collection of specifications and protocols, and in a brief paper such as this, we can only touch on its most salient features.

4 JAVA IDL AND CORBA

The Common Object Request Broker Architecture (CORBA) is an emerging open distributed object computing infrastructure being standardized by the Object Management Group. CORBA automates many common network programming tasks such as object registration, location, and activation; request demultiplexing; framing and error-handling; parameter marshalling and demarshalling; and operation dispatching. Sun refers to CORBA as

'Java IDL (Interface Reference Language)', a language to describe the class interface.

The steps to implements CORBA object:

- Write the interface using IDL to specify that how the object works.
- Using the IDL complier for the target language, generate the needed stub
- Implement the server object using the language of your choice.
- Registering the server object using CORBA naming service similar to rmiregistry.
- Write a client program that locate and invokes the server object.
- Start the naming service and the server program on the server and start the client program.

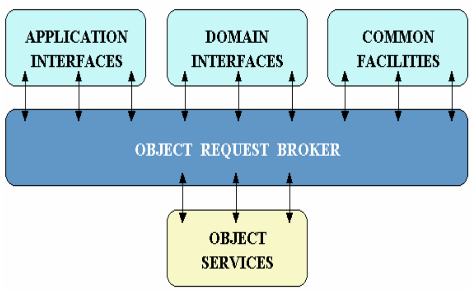


Figure 1: Multiple Interfaces of CORBA

4.1 Interface Definition Language

One of the essential features of CORBA[9,10] is that it sits at a high level of abstraction; for example, the various frameworks and services are specified not using a particular programming language, but via an IDL (Interface Definition Language). An IDL has four basic elements: modules, interfaces, operations and attributes. A module is a namespace that contains one or more interfaces. An interface is a collection of attributes and operations that correspond to an object. CORBA 2.0 specifies that an object can have only one interface. However, for CORBA 3.0 there are proposals to support multiple interfaces as (shown in fig. 1), Attributes correspond to instance variables and are used to represent data. An operation corresponds to a method. A method is identified by a name, signature, and return type.

IDL to Java compiler translates the IDL definitions for java interfaces(see figure 2). The rules for translation are called the java programming language binding. This language binding defines some standardization and CORBA vendors are required to use same mapping for IDL constructs to a particular programming language.

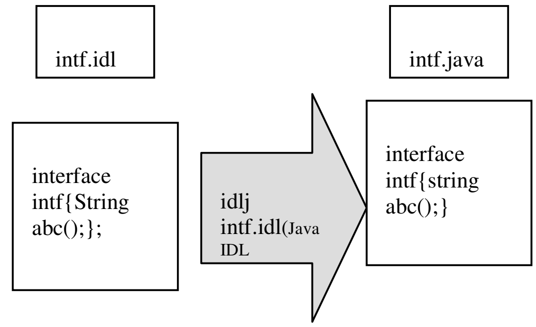


Fig 2: Translation of IDL to Java Compiler

IDL compiler also generates a number of other files and stub classes such as Interface definition, the interface that contains actual operations, CORBA specific interfaces, holder class for out parameter, stub class for communicating with ORB

Now at the server side, we can code the program in C++ or in java that does the following:

- Start the ORB
- Create an object and register it with ORB so that a client can use it
- Use the name server to bind the object to a name
- Wait for invocation from a client

At the client side, the program does the following:

- Start the ORB
- Locate the naming service by retrieving an initial reference to 'NameService' and narrowing it to a NamingContext reference.
- Locate the object to call it's method using resolve method of NamingContext.
- Cast the returned object to the correct type and invoke your methods with the help of narrow() method.

5 REMOTE METHOD INVOCATION

Beginning with the Java Development Kit 1.1 (JDK1.1), the Java programming language has included Remote Method Invocation[11,15,18] as part of the standard Java libraries(Javasoft, 1997). RMI allows client Java objects to invoke methods in server Java objects, no matter if they reside in the same JVM or even in the same host. RMI can be thought of as the object-oriented type of Remote

Procedure Call (RPC). It is quite natural way of implementing distribution when all participants are written in Java, because no extra burden of additional communication protocols or handling of underlying connection mechanisms (like sockets) is needed in the distributed objects themselves but they can invoke the remote objects almost like local ones.
Remote method invocation allows applications to call object methods located remotely, sharing resources and processing load across systems. Unlike other systems for remote execution which require that only simple data types or defined structures be passed to and from methods, RMI allows any Java object type to be used - even if the client or server has never encountered it before. RMI allows both client and server to dynamically load new object types as required.
There are three processes that participate in supporting remote method invocation.
The Client is the process that is invoking a method on a remote object. The Server is the process that owns the remote object. The remote object is an ordinary object in the address space of the server process. The Object Registry is a name server that relates objects with names. Objects are registered with the Object Registry. Once an object has been registered, one can use the Object Registry to obtain access to a remote object using the name of the object. There are two kinds of classes that can be used in Java RMI. A Remote class is one whose instances can be used remotely. An object of such a class can be referenced in two different ways: Within the address space where the object was constructed, the object is an ordinary object which can be used like any other object. Within other address spaces, the object can be referenced using an object handle. While there are limitations on how one can use an object handle compared to an object, for the most part one can use object handles in the same way as an ordinary object. For simplicity, an instance of a Remote class will be called a remote object. A Serializable class is one whose instances can be copied from one address space to another.

5.1 RMI 3-Tire Layered Architecture

To the programmer, the client appears to talk directly to the server. In reality, the client program talks only to a stub object that stands in for the real object on the remote system (as shown in figure 3).

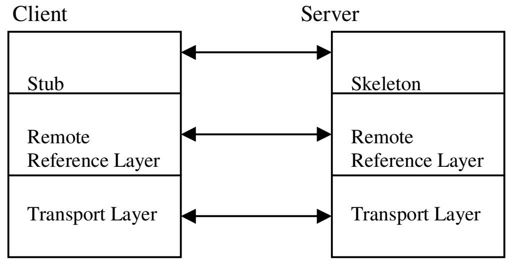


Figure 3: RMI 3-tire layered architecture

The stub passes that conversation along to the remote reference layer, which talks to the transport layer. The transport layer on the client passes the data across the Internet to the transport layer on the server. The server's transport layer then communicates with the server's remote reference layer, which talks to a piece of server software called the skeleton. The skeleton communicates with the server itself. (Servers written in Java 1.2 and later can omit the skeleton layer.) In the other direction (server-to-client), the flow is simply reversed. Logically, data flows horizontally (client-to-server and back), but the actual flow of data is vertical. The goal of RMI is to allow your program to pass arguments to and return values from methods without worrying about how those arguments and return values will move across the network. In reality, the client is only invoking local methods in a stub. The stub is a local object that implements the remote interfaces of the remote object; this means that the stub has methods matching the signatures of all the methods the remote object exports. In effect, the client thinks it is calling a method in the remote object, but it is really calling an equivalent method in the stub. Stubs are used in the client's virtual machine in place of the real objects and methods that live on the server; you may find it helpful to think of the stub as the remote object's surrogate on the client. When the client invokes a method, the stub passes the invocation to the remote reference layer.
The remote reference layer carries out a specific remote reference protocol, which is independent of the specific client stubs and server skeletons. The remote reference layer is responsible for understanding what a particular remote reference means. Sometimes the remote reference may refer to multiple virtual machines on multiple hosts. In other situations, the reference may refer to a single virtual machine on the local host or a virtual machine on a remote host. In essence, the remote reference layer translates the local reference to the stub into a remote reference to the object on the server,

whatever the syntax or semantics of the remote reference may be. Then it passes the invocation to the transport layer.
The transport layer sends the invocation across the Internet. On the server side, the transport layer listens for incoming connections. Upon receiving an invocation, the transport layer forwards it to the remote reference layer on the server. The remote reference layer converts the remote references sent by the client into references for the local virtual machine. Then it passes the request to the skeleton. The skeleton reads the arguments and passes the data to the server program, which makes the actual method call. If the method call returns a value, that value is sent down through the skeleton, remote reference, and transport layers on the server side, across the Internet and then up through the transport, remote reference, and stub layers on the client side. In Java 1.2 and later, the skeleton layer is omitted and the server talks directly to the remote reference layer. Otherwise, the protocol is the same.
Since RMI is Java based, it may only interact with non-Java applications via. Java Native Interface (JNI).

6 SIMPLE OBJECT ACCESS PROTOCOL

Simple Object Access Protocol also known as SOAP is a communication protocol. Communicating with SOAP can be viewed either as XML based remote procedure calls, or as a way of submitting XML documents to remote endpoints. These two different perspectives represent RPC-centric and message-centric. In Java RPC-centric model has become the primary model for SOAP APIs. The Java APIs representing the two different perspectives are JAXM [12] (Java API for XML messaging) and JAX-RPC [13] (Java API for XML based RPC) SOAP [10, 14, 15, 16, 17] is an XML protocol for invoking remote methods. Like IDL in CORBA, here WSDL (Web Service Description Language) provide the interface for the web service. The WSDL descriptor describes the service in a language-independent manner. The description can be easily converted into java for example: a description regarding data types in WSDL (xml schema) will be converted into java classes. JWSDP (Java Web Services Developer Pack) is freely available which contain the tools to convert the descriptor files into java class files. SOAP also works as CORBA or RMI i.e. the client program calls a local method on the proxy object and the proxy connects to the server to invoke the method.
Of course this is all very well, but SOAP is still just an RPC, calling low-level functions and leaving pretty much everything to developers. In order to make it easier to use, there is a format for describing services that can be invoked by SOAP-WSDL.
WSDL can be seen as a complement to SOAP, as it facilitates interoperability between web services. Like IDL (Interface Definition Language), which acts as a service describer with CORBA, WSDL (Web Service Development Language) is an XML syntax to describe web services. The specifications for WSDL come from a joint initiative by Microsoft, IBM and Ariba.

7 COMPARISONS

There are a number of disparities between the various approaches to distributed objects and components.

7.1 RMI vs. CORBA vs. SOAP

This section will compare the relative features of RMI and CORBA and SOAP that might effect on selecting a specific distribution mechanism.
CORBA and RMI take similar approaches to enabling distributed computing and have roughly analogous mechanisms for object interface language, object manager(for security) and naming service. Both CORBA and RMI use specific communication protocol for network transmission. RMI uses TCP/IP, the most common internet protocol, whereas CORBA uses Internet Inter-Orb Protocol (IIOP) that builds on TCP/IP.
RMI being a Java based technology is essentially not cross language at all. Interoperability to non-Java programs must be done through JNI and has no common interface, as with CORBA. However, Java's inherent cross-platform capabilities substantially increase the number of platforms on which distributed applications may run.
An advantage of RMI over CORBA involves security. The RMISecurityManager ensures that no holistic code can have access to local resources. These are classes in JDK that implement encryption and digital signatures.

7.1.1 RMI
Advantages

- Portable across many platforms
- Can introduce new code to foreign JVMs

- Java developers may already have experience with RMI (available since JDK1.02)
- Existing systems may already use RMI - the cost and time to convert to a new technology may be prohibitive

Disadvantages

- Tied only to platforms with Java support
- Security threats with remote code execution, and limitations on functionality enforced by security restrictions
- Learning curve for developers that have no RMI experience is comparable with CORBA
- Can only operate with Java systems - no support for legacy systems written in C++, Ada, Fortran, Cobol, and others (including future languages).

7.1.2 CORBA
Advantages

- Language neutral services can be executed on many different platforms, with an interface definition language (IDL) mapping.
- With IDL, the interface is clearly separated from implementation, and developers can create different implementations based on the same interface.
- CORBA supports primitive data types, and a wide range of data structures, as parameters
- CORBA is ideally suited to use with legacy systems
- CORBA is an easy way to link objects and systems together
- CORBA systems may offer greater performance

Disadvantages

- Implementing or using services require an IDL mapping to your required language - writing one for a language that isn't supported would take a large amount of work.
- IDL to language mapping tools create code stubs based on the interface - some tools may not integrate new changes with existing code.
- CORBA does not support the transfer of objects, or code. The future is uncertain - if CORBA fails to achieve sufficient adoption by industry, then CORBA implementations become the legacy systems.
- Some training is still required, and CORBA specifications are still in a state of flux.
- Not all classes of applications need real-time performance, and speed may be traded off against ease of use for pure Java systems.

7.1.3 SOAP
Advantages

- SOAP may have stronger support from the open source community (but CORBA is quite strong here too), and stronger support for recent programming languages that people will be using more now and in the near future.
- SOAP's greatest advantage and disadvantage at the same time, is its lack of standard above it and below it. SOAP is as simple as the implementer of the middleware wants it to be. SOAP may be nasty and complex but flexible when using tool A, but very easy (though inflexible) when using tool B.
- The SOAP community has the possibility to happily construct some SOAP XML and send it out on the net, then listening for an answer. Using simple (and free!) script tools this can be easily done, so the barrier to use SOAP appears to be very low.

Disadvantages

- Elaborate coding required at a low level of abstraction
- Lack of interoperability, i.e. many competing SOAP implementations
- Saturation of firewall port 80 (HTTP)
- Lack of performance due to the requirement to parse and transport XML

Lack of compile time type checking; harder debugging

8 OBJECT COMPONENTS FOR BUILDING UBIQUITOUS COMPUTING ENVIRONMENT

Some of the most common types of software platforms for developing distributed applications are remote procedure call (RPC) systems. RPC systems have been around for decades and have proven to be flexible and long-lived. Sun/RPC is an example of a classic RPC system; more recent examples include XML/RCP and SOAP. Distributed object systems such as CORBA, Modula-3's network objects and Java RMI are the object-oriented counterparts of RPC systems. The fundamental benefit of these systems is that they allow developers to easily build simple distributed systems using familiar programming models, namely, procedure calls and object/method invocation. Ironically, while this transparency provides for a familiar programming model, it is what makes standard RPC systems a poor choice for building ubiquitous applications. Whenever the thread of control is executing remotely, the execution context of the caller is at the mercy of network connections that may be unreliable. Because the programming model intentionally hides the difference between local and remote execution, it is difficult to reason about the conditions under which a given component can safely execute. Furthermore, the inherently synchronous nature of RPC interactions limits the responsiveness of applications, as they need to wait for remote services to complete processing each procedure call. As a result, it is difficult to build robust and responsive ubiquitous applications using RPC systems.

The problems with RPC are actually just symptoms of a more general issue: ubiquitous computing environments are dynamic. Thus, we have to find a way to write software components that respond appropriately to change. One.world addresses this problem in a general way: it insulates software components from changes that can be handled by the system, and it notifies them of changes that should be addressed at the application level. This, in turn, gives the application the opportunity to treat the user in a similar way. The application can insulate users from changes when possible, and interact with them when input is needed.

one.world [19] is a Java-based run-time system that executes on a standard JVM. Components are objects that are prohibited from accessing system resources directly. In the one.world programming model, RMI is prohibited, as are application-level ownership and control of threads. Direct access to local resources such as the file system and the native operating system is discouraged but can be enabled by the developer. By restricting software components to consist of nothing but a collection of asynchronous event handlers, all interactions can be mediated by the one.world system. In this way, the system has the opportunity to handle changes or failures, and it can notify the component when necessary. Some examples of system notification events are activation and deactivation of components, event delivery failure, and relocation of a component from one node to another. By exposing change to the application, one.world allows the developer to determine how best to respond in terms of benefits to the user. One.world provides alternatives to mitigate what is prohibited or restricted: a location independent tuple-store instead of file system access, event queues and timer events instead of threads, remote event passing instead of RMI, and a variety of other system-like utilities. By adhering to these alternatives, one.world guarantees that a component can execute on any node given enough physical resources. Because all interactions between components are through asynchronous events, one.world provides a rich event delivery infrastructure including early- and late-binding discovery, multicast, and lookups on the events themselves.

In addition, one.world allows components to be dynamically organized into hierarchical execution environments. Each environment contains a tuple-store for persistent data, and can contain running components and subordinate environments. The tuple-store can be used by components for data storage and for check pointing execution state. Environments, not components, are the unit of migration in one.world, thus ensuring the availability of all state that is essential for continued execution after migration.

9 CONCLUSIONS

Now the question is, “what’s the bottom line?” Which technology is the best to use, DCOM, Java RMI, or CORBA [7, 8]? The answer to this question really depends on the needs and existing infrastructure. In fact, the listed technologies are not the only options. If the system will run completely on Windows machines, then DCOM might be a good solution. If the system will run completely on Java machines, then Java RMI is an option.

However, any heterogeneous system will probably work better with CORBA. While DCOM has good roots in COM, the network integration is still immature. In addition Windows NT servers with DCOM enabled are vulnerable to denial of service attacks because of a bug. Thus, any DCOM infrastructure must stay within a firewall. Assuming that in the future this bug does not exist, then DCOM as an infrastructure is a lot more feasible since it contains packet level encryption. Development tools for DCOM are the best out of all these technologies, and probably will be for a long time. This is because of Microsoft's commitment to DCOM as well as its huge market share of compilers and desktop operating systems. However, DCOM support outside of Windows should be taken with a grain of salt, and tools on those platforms will definitely not be mature.

In the long run, the hype about Web Services will diminish (as all hypes do), and CORBA and Web Services will both have their place. Just as CORBA and COM, and for that matter RMI, coexist. When necessary, bridges will be used to tie everything together. It looks like we're going to use SOAP to talk to the outside world, and we'll be using CORBA in the inside. In the short term, it will take a number of years for SOAP to mature, and the missing pieces to be defined.

10 FUTURE WORK

There are several issues for future work. The layered architecture of middleware platforms (such as CORBA, SOAP, and RMI) is a mixed blessing. On the one hand, layers provide services such as demarshaling, session management, request dispatching, quality-of-service (QoS) etc. In a typical middleware platform, every request passes through each layer, whether or not the services provided by that layer are needed for that specific request. This rigid layer processing can lower overall system throughput, and reduce availability and/or increase vulnerability to denial-of-service attacks. As a future research work in this area we can further make improvements in throughput by implementing bypass in the middleware layers using Aspect Oriented approach for building robust and efficient Ubiquitous Computing Environment.

The Database Object Components one.world based study for building ubiquitous computing environment is a formal user study. But most importantly, we are confident that we can continue to maintain and extend these distributed objects components in the future while we learn more about how one.world's guarantees can provide unique capabilities.

References:

[1] Woon-Yong Kim, Seok-Gyu Park ,A Policy Management System for Dynamic Web Services Connection , WSEAS TRANSACTIONS on INFORMATION SCIENCE & APPLICATIONS, pg no. 1528 Issue 8, Volume 3, August 2006, ISSN 1709-0832 http://www.wseas.org

[2] The Rise of Web Services: Completing the Picture, Johann Dumser & Jean-Christophe Cimetiere, TrendMarkers, Vol. 3 Number 1, TechMetrix Research. http://www.techmetrix.com/trendmarkers/tmk0101/tmk0101-4.php3.

[3] Web Services: SOAP, UDDI, and Semantic Web, Justin R. Erenkrantz, Institute for software research, ISR Technical Report # UCI-ISR-04-3 www.isr.uci.edu/tech-reports.html

[4] Paivi Kallio , Challenges and Requirements of Ubiquitous Computing, WSEAS TRANSACTIONS on INFORMATION SCIENCE & APPLICATIONS, pg no. 234 Issue 1, Volume 1, July 2004, ISSN 1790-0832 http://www.wseas.org

[5] "Distributed Network Systems: From Concepts to Implementations" by Weijia Jia, Wanlei Zhou.

[6] "Power Programming with RPC" by John Bloomer

[7] Srinivasan, R., Remote Procedure Call Protocol Specification Version 2, Internet Engineering Task Force, RFC No. 1831, Augm 1995, http://www.ietf.org/rfc/rfc1831.txt?number=1831

[8] Dionisis X. Adamopoulos, Engineering Open Interoperable Quality Driven Web Services , WSEAS TRANSACTIONS on INFORMATION SCIENCE & APPLICATIONS, pg no. 1528 Issue 8, Volume 2, August 2005, ISSN 1790-0832 http://www.wseas.org

[9] CORBA: Common Object Request Broker Architecture, http://www.omg.org, 2008

[10] Elfwing, R., Paulsson, U., and Lundberg, L, Performance of SOAP in Web Service Environment Compared to CORBA, In Proceedings of the Ninth Asia-Pacific Software Engineering Conference, IEEE, 2002.

[11] Java: Remote Method Invocation; http://java.sun.com/j2se/1.4.2/doc/guide/rmi/index.html, 2008
[12] N. Kassem, A. Vijendran, and Rajiv.Mordani, “Java API for XML Messaging (JAXM),” Sun Microsystems, available online http://java.sun.com/xml/downloads/jaxm.html, 2008
[13] R. Chinnici, “Java API for XML-Based RPC (JAXRPC),”Java Community Process, Tech.
[14] Davis, D., and Parashar, M., Latency Performance of SOAP Implementations, In Proceedings of 2nd IEEE/ACM. International Symposium on Cluster Computing and the Grid, IEEE, 2002.
[15] Gudgin, M., Hadley, M., et al. Simple Object Access Protocol (SOAP) 1.2: Adjuncts. http://www.w3.org/TR/soap12-part2/>, W3C, HTML, 2003.
[16] Gudgin, M., Hadley, M., et al. Simple Object Access Protocol (SOAP) 1.2: Messaging Framework. <http://www.w3.org/TR/soap12-part1/>, W3C, HTML, 2003.
[17] Jay Ongg “An Architectural Comparison of Distributed Object Technologies” Available online http://web.mit.edu/profit/PDFS/OnggJ.pdf, 2008
[18] Cay S. Horstmann, Gary Cornell,Core Java 2, Volume II-Advanced Features, Seventh Edition, Pearson Education, 2007.
[19] Robert Grimm, "One.world: Experiences with a Pervasive Computing Architecture," IEEE Pervasive Computing, vol. 3, no. 3, pp. 22-30, Jul-Sept, 2004