Mobile Component Runtime Environment for Mobile Devices

Minhong Yun, Seokjin Yoon, Sunja Kim

**Electronics and Telecommunications Research Institute*

138 Gajeonro, Yuseong-gu, Daejeon, South Korea

{mhyun, sjyuun, sunjakim}etri.re.kr

***Abstract*— The evolution of software must be concurred with that of hardware to provide users with best services. Despite of currently prominent evolution of hardware, lazy evolution of software prohibits users from utilizing full capabilities of mobile devices. So our institute is developing a mobile component runtime environment which can extend and optimize platform capabilities. It makes it possible that software platform can be extended automatically and reduce software platform upgrade and reorganizing issues. To achieve the goal of the mobile component runtime environment our component model supports two different types of components. One is in-process component, and the other is out-of-process component. Various and beneficial services are expected using the two types of components. The mobile component runtime environment includes component runtime engine and out-of-process component runtime engine to run the two types of components. To enhance the accessibility to our-of-process components, the mobile component runtime environment uses light-weight RPC for inter-process communication mechanism. And it enables Java applications to access C/C++ components.**

***Keywords*— Mobile Platform, Mobile Component**

I. INTRODUCTION

Mobile devices have been evolved in hardware and software sides, and currently with the growth of smarphone users mobile devices face new challenges [1]. Cellular phones which have evolved into smartphone via vanilla phone and feature phone are trying to meet users' needs that users want various applications and services. The needs of users' exceed specialized features such as multimedia playback. These are needs for applications and services that are useful for wide-range area from game to mobile office. Smartphone, one of the best known mobile devices, has progressed a lot in both sides of hardware and software to provide users with various applications and services. Most prominent progress in hardware side is instalment of another application processor to process applications. Before installing extra processor for applications, baseband processor which is made for processing radio frequency and data/voice communication was used for processing all kinds of applications. Current smartphones provide various applications with connecting baseband processor to applications processor. MMU is also biggest progress in hardware side. Using MMU makes phones install general purpose OS such as Windows Mobile and Linux.

Installing application processor on phone causes hardware changes and it also requires alternation of software [1]. The requirement for alternation of software is mainly resulted from in-stalling general purpose OS. Before installing general purpose OS, phones adopt some middleware platforms such as Wireless Internet Platform for Interoperability (WIPI) [2], Binary Environment for Wireless (BREW) [3], Java VM and etc to enable applications to be executed. In contrast to vanilla and feature phones, smartphones that install general purpose OS with MMU are able to run various applications without middleware platforms. Since 2003, our research institute is perpetually researching and developing software platform based on embedded Linux. Now according to the recognition of requirements for new software platform architecture to support hardware progress, our research institute is researching and

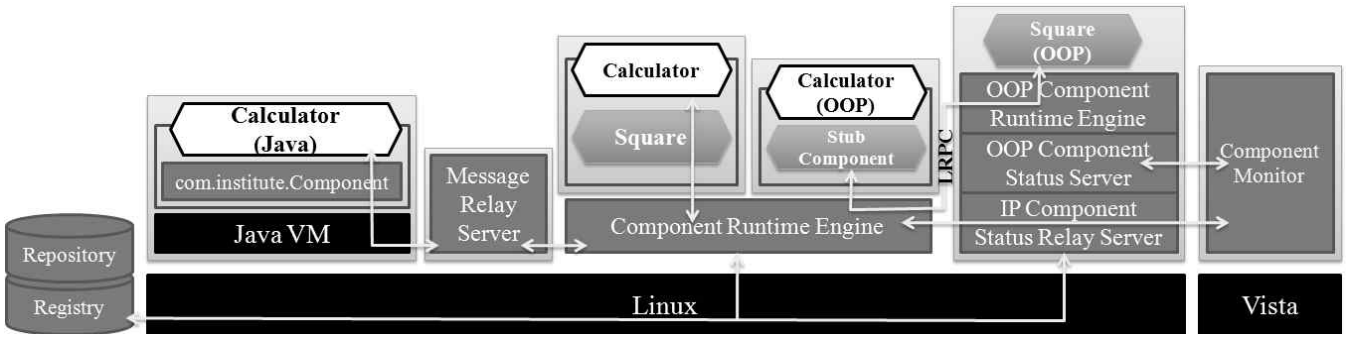

Figure 1. Architecture of Mobile Component Runtime Environment

developing mobile component runtime environment to meet the requirements. The mobile component runtime environment will be introduced and discussed in this paper.

II. MOBILE COMPONENT RUNTIME ENVIRONMENT

The mobile component runtime environment consists of three major parts; component runtime engine, out-of-process component runtime engine, and light-weight RPC.

Component runtime engine has two major roles. The first one is to provide applications or services with transparent accessibility to mobile components. It enables applications and services not to care about mobile component types. The second major role of the component runtime engine is to run and manage in-process components. It loads appropriate components into applications' memory, returns interfaces, releases interfaces, and unloads components from memory.

Out-of-process component runtime engine has two major jobs, too. The first one is to run and manage out-of-process components. Similarly to component runtime engine it loads appropriate out-of-process components, returns interfaces, releases interfaces, and unloads out-of-process components from memory.

The last important element of the mobile component runtime environment is light-weight RPC. Because out-of-process components run in different address space from those of applications or services, there should be a way to access components running in different address space. Light-weight RPC gives a way for applications to access out-of-process components. With the light-weight RPC applications can transparently use capabilities of out-of-process components which are located in different address spaces.

We will discuss the above three parts and other subsystem parts in this section.

	COM	EJB	.Net	CORBA	Mobile Component
In-Process	O	O	O	O	O
Out-of-process	O	O	O	X	O
Remote	O	O	O	O	X
Multi-thread	O	O	O	O	O
Multi-language	O	X	O	O	O

Figure 2. Mobile Component and Related Systems

A. Mobile Component Runtime Environment Overview

The mobile component runtime environment looks similar to Microsoft .Net Framework [9], Microsoft Component Object Model (COM) [8], Object Management Group's Common Object Request Broker Architecture (CORBA) [4][5][6], and Sun Microsystems' Enterprise JavaBeans (EJB) [7]. But the mobile component runtime environment considers mobile environment from design phase. There is also other distinction that the mobile component runtime environment intends platform independence and provides light-weight RPC for inter-process communications.

In Figure 2, the mobile component runtime environment and other related systems are compared. Mobile component runtime environment does not support remote components which are in different devices, but local components in the same device. Supporting remote components is intentionally avoided, because it is expected that accessing remote component in mobile environment is rare according to our experience. Many applications run on mobile phones are made in Java language, so mobile component runtime environment is designed to support Java applications.

As shown in Figure 2, the mobile component runtime environment supports in-process components and out-of-process components and includes component runtime engine, out-of-process (OOP) component runtime engine, light-weight RPC (LRPC), com.institute.Component, message relay server, repository, registry, and component monitor.

We will discuss each part of the mobile component runtime environment in this section.

B. Component Types: In-process and Out-of-process Component

There exist two component types as shown in Figure 2. One is in-process component type and the other is our-of-process component type. "Square" in the figure is in-process type component, and "Square (OOP)" in the figure is out-of-process type component. These are classified by the memory location where they are executed.

In-process type components run at the same address spaces as applications using the components. When different applications use a component, each application loads its own component instance into its own address space. In other words, there will be multiple component instances in memory. Because in-process components are loaded and run in the same address spaces as those of applications, it is relatively easy and quick to communicate between component and application. Even if multiple applications use a component it is not necessary to consider concurrency issues, because each application has its own component instance. However, in-process components can be loaded and executed only when applications are running. And if an application using a component ends, the component ends too. In other words, component's life cycle can't exceed that of application using it.

However, out-of-process type components run at the different address space from those of applications using the components. Because out-of-process components and applications run at different address spaces, to access and use out-of-process components applications need way to transmit and receive data from and to them. In the mobile component runtime environment, light-weight RPC has been adopted for IPC mechanism to address the problem. Out-of-process type components take more execution time owing to the IPC latency. However, out-of-process type components can save memory spaces. When different applications use a component, there will be only one component instance in memory. If an application first uses a component, the component is loaded in the address space of out-of-process component runtime engine. And when another application uses the component, the component in the address space of out-of-process component runtime engine is referenced. Details of the out-of-process component runtime engine will be discussed latter in this section. In other words, different applications share a component. It avoids redundant component interfaces, and saves memory spaces in mobile

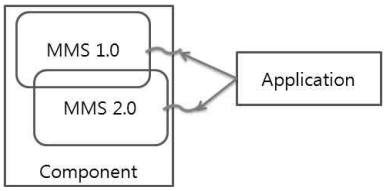

Figure 3. Component with Multiple Interfaces

environment. Because different applications share a component, each out-of-process component must consider about concurrency to avoid data loss or collision. It may increase the complexity of components. However, out-of-process type components have strong point that components' life cycles do not belong to those of applications. An out-of-process type component doesn't need to end even if there is no application using the component. Because out-of-process components can determine their own life cycles, even though all applications end, theses can run on the out-of-process component runtime engine and process jobs. This feature of out-of-process type components enables various services and background processing.

C. Component and Interface

In mobile component runtime environment, component is a binary chunk. And each component can have one or more interfaces. Applications cannot use the components directly. They should use interfaces to access components. For example, a component providing math capabilities can have two different interfaces; "math" interface for normal math capabilities and "mathEx" for complicated and high resolution math capabilities. To utilize the capabilities of mathematics application should use one of the interfaces.

It is also possible that a component can have two different versions of interfaces, and it is a good way to use components and interfaces. In Figure 3, there is a Multimedia Messaging Service (MMS) component with two different interfaces. Interface MMS 1.0 can only handle SMIL 1.0 and interface MMS 2.0 can only handle SMIL 2.0 [10]. The component provides two versions of interfaces for the purpose of backward compatibility. And applications can access the appropriate interfaces to process MMS documents.

Each interface uses UUID for its ID. The UUID consists of 128 bits, and correct way to construct the ID is described in RFC 4122 [11], ITU-T Rec. X.667 [12] and ISO/IEC 11578:1996 [13]. It can be easily generated using many tools. Some web site provides web-based UUID generator. For Linux system, there is a library called libuuid which is easy to use, and Microsoft provides GUID generator [14], too.

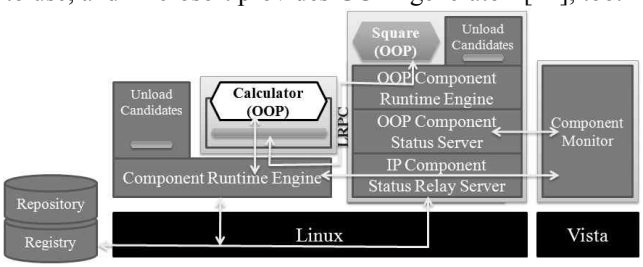

Figure 4. Out-of-process Component Behavior

D. Component Runtime Engine

Component runtime engine provides applications with transparent accessibility to in-process type components and out-of-process type components, and loads in-process components into applications' own address spaces which use the components.

In-process type components and out-of-process type components have different life cycles and behaviours. However, applications can access two different types of components using same way. By providing applications with transparent accessibility, the types of components used by applications can be disregarded.

When an application queries an interface for using an in-process type component, component runtime engine first looks up component registry to find the location where the component locates. Then, the component runtime engine loads the component into the address space of the application. It returns the interface that the applications queried, and increase reference counter of the inter-face. If the application using the component requires one more same interface, the component runtime engine does not load another component into memory, but increases reference counter of the interface and returns the same interface. When application does not want to use the interface any more, it tells the component runtime engine to release the interface. The component runtime engine decreases the reference counter of the interface. And once reference counter reaches zero, the component is moved to unload candidate list. Before unloading the component from memory, the component runtime engine keeps it for a while according to the policy of the component runtime environment. Now the policy is mixed one of LRU and time. If the application queries an interface in a component which is in unload candidate list, the component runtime engine just uses the loaded component.

E. Out-of-process Component Runtime Engine

Out-of-process component runtime engine is responsible for executing and controlling out-of-process components. It loads out-of-process components into the address space of out-of-process component runtime engine, and runs them. There exists only one out-of-process component runtime engine in a mobile device. Therefore, all components are loaded into the address space of the out-of-process component runtime engine.

Figure 4 shows detailed behaviour of the out-of-process component runtime engine and an out-of-process component. In the figure, there is a calculator application that uses Square component which is out-of-process type. A stub component is in the address space of the application. The stub component works only for inter-process communication with light-weight RPC. It does not provide any operation about square. The applications using an out-of-process component can have transparent access to the out-of-process component with the stub component.

When an application queries an interface for using an out-of-process type component, component runtime engine first looks up component registry to find the location where the component exists. After searching the component from the registry, the component runtime engine recognizes that the

requested component is out-of-process type. Then, the component runtime engine loads an appropriate stub component into the address space of the application, and send a message to out-of-process component runtime engine to make the requested component get ready to work. Finally the component runtime engine receives a message notifying that the out-of-process component is ready, and establishes the connection between the stub component and the out-of-process component using light-weight PRC. After all preparation to use out-of-process component is completed, the component runtime engine returns the queried interface of the stub component to the application. Now, the application can access the out-of-process component.

The application doesn't need to care about the inter-process communication mechanism. It doesn't even need to know the type of the component it wants to use. All components have only one type to the application. The application accesses the component capabilities though light-weight RPC. The application just uses the interface of the stub component, and then the stub component process all thing about light-weight RPC.

If an application using a component queries the same interface again, the component runtime engine increase the reference counter of stub interface. The out-of-process component runtime engine doesn't do anything to out-of-process component. The reference counter of the out-of-process component does not change. However, when another application queries the same interface, the component runtime engine loads a new stub component instance into the address space of the application and sends a message to the out-of-process component runtime engine to increase the reference counter of the out-of-process component. In other words, if there are multiple applications which are using a same out-of-process component, there will be multiple instances of the stub component and only one out-of-process component of which reference counter is same as the number of the applications using the out-of-process component. For example, if five applications are using a out-of-process component, there will be five sub component instances in each application's address space and only one out-of-process component will be loaded into the address space of the out-of-process runtime engine. And the reference counter of the out-of-process component will be five.

Because releasing and unloading of stub components in applications' address spaces is same to those of in-process components, please refer the previous section. When a stub component is un-loaded from the unload candidate list, component runtime engine sends out-of-process component runtime engine a message to decrease the reference counter of out-of-process component. If all stub components related to an out-of-process component are un-loaded, the reference counter of the out-of-process component becomes zero and the out-of-process component is moved into unload candidate list of out-of-process component runtime engine. Like component runtime engine, the out-of-process component runtime engine keeps the out-of-process component for a while before unloading from memory. If an application queries an inter-face in a component which is in the unload candidate list, the out-of-process component runtime engine just uses it, not loads a new instance into memory.

In Figure 2 and Figure 4, OOP Component Status Server and IP Component Status Relay Server are shown. The OOP Component Status Server reports status changes of out-of-process components, and the IP Component Status Relay Server relays status changes of in-process components. Because these two are not important parts of mobile component runtime system, detailed descriptions are omitted.

F. Light-weight RPC

Inter-process communication mechanism is necessary for an application to communicate with out-of-process components [15][16]. To achieve the goal which makes applications to communicate with components in standardized method, Microsoft's RPC[17], Sun Microsystems' RPC [18], Java Remote Method Invoca-tion (RMI) [19], and CORBA ORB [6] have been surveyed. In the mobile component runtime environment, light-weight RPC has been made by modifying Sun Microsystems RPC. In Figure 6. Mobile Component and Related Systems, the light-weight RPC and related RPC systems are compared.

	SUN RPC	MS RPC	Java RMI	CORBA	Light-weight RPC
Local RPC	TCP UDP	IPC	TCP UDP	TCP UDP	IPC
Async. RPC	X	O	X	X	O
User Defined Marshaling	X	O	O	X	O
Language	C/C++	C/C++	Java	C/C++ Java	C/C++
Multiple Languages	X	X	X	O	X

Figure 6. Mobile Component and Related Systems

The light-weight RPC has been designed with considering compatibility with Sun RPC. Adding a new API to Sun RPC is avoided, and only a new protocol parameter has been added to support compatibility with Sun RPC. In other words, existing applications using Sun RPC can run without using light-weight RPC. The prototype of clnt_create() which is a part of Sun RPC and initializes Sun RPC system is shown in Figure 8. The last parameter used in this function, nettype, is for determining the way how to communicate. Sun RPC supports only "TCP" and "UDP" for this parameter. "IPC" is added for the last parameter in light-weight RPC. In other words, applications which want

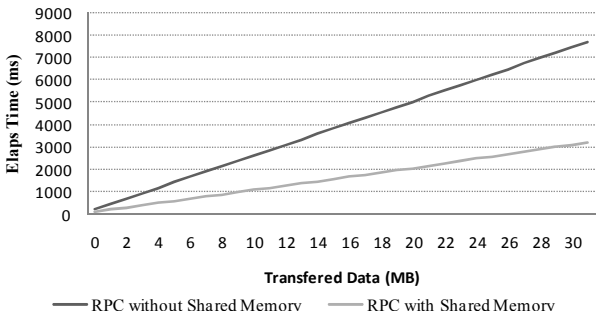

Figure 5. Transfering Data with and without Shared Memory.

to use the light-weight RPC instead of Sun RPC can use the light-weight RPC by passing "IPC" through this parameter.

```
CLIENT* clnt_create ( const char*
hostname, const u_long prognum, const
u_long versnum, const char* nettype);
```

Figure 8. The Prototype of clnt_create()

The light-weight RPC has three major features. The first one is that the light-weight RPC can transfer massive data very fast using shared memory. The light-weight RPC supports both message queue and shared memory. When shared memory is used for transferring data, the light-weight RPC only transfer the information about shared memory, and the massive data are transferred through shared memory. Using shared memory to exchange massive data reduces time required for sending and receiving data. Figure 6 shows the way how the light-weight RPC uses shared memory to transfer massive data. As shown in the figure, information about the shared memory is exchanged through message queue, and massive data are exchanged using shared memory.

The second feature is that it supports asynchronous communications, whereas Sun RPC provides ways for synchronous communications. Through support of the asynchronous communications, out-of-process components have flexibility to provide various services.

The last feature is that the light-weight RPC supports user de-fined marshalling and demarshalling. If there are data which should be marshaled or demarshaled manually, components can register special handlers to process the data. The handlers are invoked before transferring data or after receiving data for components to process the data manually. It gives flexibility to the mobile component runtime environment, and helps to provide more various services.

Figure 6 is a graph to show time needed to transfer data with and without shared memory. The data size is 33 megabytes. It takes 7728 ms to transfer 33MB data without shared memory, and takes 3196 ms to transfer the same data with shared memory. So, it takes more than twice to transfer massive data without shared memory which is the way for Sun RPC to transfer data. Equipment used for the test includes 2.80 GHz Intel Pentium 4 processor and 512MB main memory. The operating system is Fedora Core 9 with Linux kernel 2.6.25.

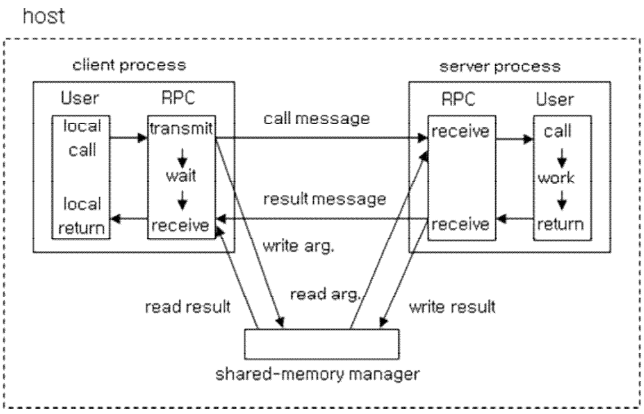

Figure 10. Light-weight RPC Using Shared Memory

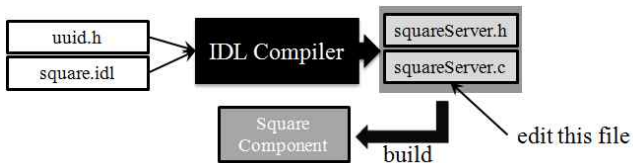

Figure 7. From IDL File to Component Binary

G. File Structure for In-process and Out-of-process Components

Two types of components are made by different sequence. Making an in-process component starts from creating an IDL file. Once an interface is defined in IDL file, the IDL file is compiled into a .h file and a .c file.

Figure 9 shows how an IDL file becomes a component binary with an interface. Suppose that we are making Square component with Square interface in in-process component type. The first two files, uuid.h and square.idl, are made by hand. The uuid.h file contains interface UUID and the square.idl includes definition of Square interface. These two files are compiled into two source codes; squareServer.h and squaerServer.c. The two skeleton codes have the right form to be run on component runtime environment. Editing squareServer.c to add some codes to calculate square is the only one we should do. After inserting codes, Square component with Square interface can be built.

Making an out-of-process component starts from creating a RPC specification file. Once an interface is defined in a RPC specification file, the RPC specification file is converted into some .h files and .c files.

Figure 10 shows how a RPC specification file becomes two component binaries with interfaces. Suppose that we are making Square component with Square interface in out-of-process component type. Like in the case of making an in-process component, the first two files, uuid.h and square.idl, are made by hand. The uuid.h file contains interface UUID, and the square.x file includes RPC specification in RFC 1831 [20] form. These two files are used by the new rpcgen. The new rpcgen is based on Sun rpcgen. In the figure, four files (square_clnt.c, square_svc.c, square.h, and square_xdr.c) in boxes with dotted line are generated by original rpcgen, but three files (squaerClien.c, squareServer.c, and square-Server.h) in boxes with solid line are generated by the new rpcgen. The

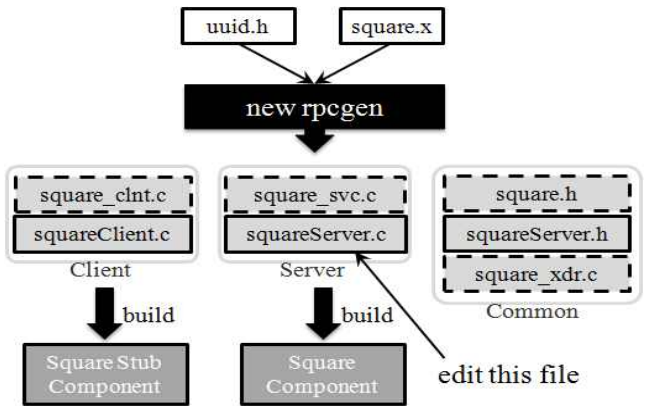

Figure 9. From RPC Specification File to Component Binaries

three files in common section in the figure are used for client and server. The two files in client section in the figure are used to build a stub component which is the same one of that of Figure 2 and Figure 4. The two files in server section in the figure is used to build Square component which is the out-of-process component. Editing the squareServer.c file in the server section to add some code to calculate square is the only one we should do by hand. In summary, the new rpcgen generates source codes for stub component and out-of-process component from RPC specification file and only one file is needed to be edited.

H. Supporting Java Applications

The needs of supporting Java applications in the mobile component runtime environment are from mobile industry. To reuse current Java applications in the mobile component runtime environment, mobile companies request the mobile component runtime environment to support Java applications. A method using message relay server is used instead of using JNI and etc to provide independence from Java VM.

Figure 11 shows details of the message relay server which is the same one of Figure 2. The message relay server and the component runtime engine run on C/C++ area. There exists only one message relay server in a mobile device, which interacts with com.institute.Component class. Java applications use com.institute.Component class to access C/C++ components. If a Java application wants to access C/C++ component, it create com.institute.Component class with interface UUID. The com.institute.Component sends a message to the message relay server to load an appropriate component into memory. After the message relay server loads the component, the com.institute.Component class returns a class instance which is used to access the loaded C/C++ component. From now on, the Java application can access the C/C++ component using com.institute.Component instance. When the Java application wants to use C/C++ component, it passes method name and parameters to the com.institute.Component instance. Then, the com.institute.Component instance sends the message relay server a message to invoke the method.

Because the message relay server doesn't know which method will be called in runtime, there should be a general way to call arbitrary methods with different parameters. In the mobile com-ponent runtime environment, the message relay server manipulates stack and registers to solve the problem. Before calling a method, the message relay server copies all parameters into appropriate registers and stack. Then, call the method with no parameter. Because all parameters have been copied into right locations, the method can use all the parameters well.

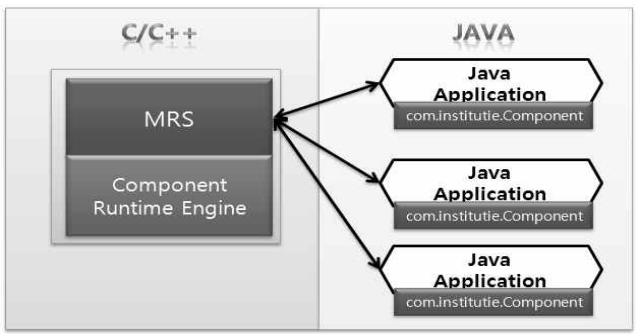

Figure 11. C/C++ Component and Java Applications

III. Conclusion and Future Works

It is pretty sure that the evolution of mobile device hardware enables users to get various services. One of the current trends of smartphones is related to full touch screen with larger LCD. The large LCD makes full-browsing and multimedia playback more effective and easier. However, there is an opinion that it is based on partial evolution of software, not whole evolution of software, and the partial evolution is attributed to the evolution of hardware. So, to utilize the advanced hardware fully it is necessary to develop a software platform to support it.

So our institute is researching and developing a mobile middleware platform based on the mobile component runtime environment, and core implementations including light-weight RPC, component runtime environment are completed. The mobile component runtime environment is designed to support operating system independence and multi-language for the compatibility with Java applications.

The remained jobs include implementing peripheral parts and deploying the mobile component runtime environment onto real mobile devices. The target devices include Nokia N810 and Linux smartphone we have made before.

References

[1] J.E.Yu. *Key Enabler of Smartphone: Software*, SW Insight, April 2009, p5-35
[2] WIPI Official Site, http://www.wipi.or.kr
[3] BREW Official Site, http://brew.qualcomm.com
[4] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++ First edition*, Addison Wesley, 1999
[5] Randdy Otte, Paul Patrick and Mark Roy. *Understanding CORBA,* Prentice Hall, Inc., 1996
[6] Introduction to CORBA, http://java.sun.com/developer/onlineTraining/corba/corba.html
[7] Kung-Kiu and Zheng Wang. *Software component models, IEEE Transactions on software engineering,* Vol 33, No. 10, October 2007
[8] John Cadman. *Waite Group's COM/DCOM Primer Plus*, Sams, Nov. 1998
[9] Microsoft. Microsoft *.NET Framework Reviewers Guide*, Microsoft Corporation, Feb. 2002.
[10] W3C. Synchronized Multimedia, http://www.w3.org/AudioVideo/, Dec 2008
[11] P. Leach, M. Mealling, R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace*, http://www.ietf.org/rfc/rfc4122.txt, July 2005
[12] Object Identifiers (OID) and Registration Authorities Recommendations, http://www.itu.int/ITU-T/studygroups/com17/oid.html, April 2009
[13] Information technology -- Open Systems Interconnection -- Remote Procedure Call (RPC), http://www.iso.org/iso/catalogue_detail.htm?csnumber=2229
[14] Create GUID (guidgen.exe), http://msdn.microsoft.com/en-us/library/ms241442(VS.80).aspx, 2009
[15] John Bloomer. *Power Programming with RPC*, O'Reilly & Associates, Inc., February 1992
[16] W. Richard Stevens. *UNIX network programming: Interprocess Communications, Volume 2*, Second edition, Prentice Hall, Inc., 1999
[17] Remote Procedure Call (Windows), http://msdn.microsoft.com
[18] Remote Procedure Call (RPC), http://docs.sun.com
[19] The Java Tutorials: RMI, http://java.sun.com/docs/books/tutorial/rmi
[20] RPC: Remote Procedure Call Protocol Specification Version 2, http://tools.ietf.org/html/rfc1831