Enabling Rapid Feature Deployment on Embedded Platforms with JeCOM Bridge

Jun Li, Keith Moore

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304, USA
{junli, kem}@hpl.hp.com

Abstract. A new class of embedded devices is emerging that has a mixture of traditional firmware (written in C/C++) with an embedded virtual machine (e.g., Java). For these devices, the main part of the application is usually written in C/C++ for efficiency and extensible features can be added on the virtual machine (even after product shipment). These late bound features need access to the C/C++ code and may in fact replace or extend functionality that was originally deployed in ROM. This paper describes the JeCOM bridge that dramatically simplifies development and deployment of such add-on features for the embedded devices and allows the features to be added without requiring the firmware to be reburned or reflashed. After being dynamically loaded onto the device's Java virtual machine, the JeCOM bridge facilitates transparent bi-directional communication between the Java application and the underlying firmware. Our bridging approach focuses on embedded applications development and deployment, and makes several significant advances over traditional Java Native Interface or other fixed stub/skeleton COM/CORBA/RMI approaches. In particular, we address object discovery, object lifecycle management, and memory management for parameter passing. While the paper focuses on the specific elements and experiences with an HP proprietary infrastructure, the techniques developed are applicable to a wide range of mixed language and mixed distributed object-based systems.

1 Introduction

A new class of smart devices is emerging that combines the efficiency of ROM-based C/C++ code with the flexibility of an embedded Java virtual machine. The firmware in these devices is usually burned into ROM, and therefore difficult to update or modify after product shipment. However, after product shipment, there is often a need to introduce a new feature not foreseen in the development phase. The new feature may be a customization of an existing feature (such as a custom logo on the front-panel of the device), or may be a new capability that extends the features of the device (such as adding secure authentication to the login procedure at the device, in order to access the user's email account).

The Java virtual machine creates a safe sandbox in which these augmented features are added; however, extending, invoking and replacing ROM features requires bi-directional communication between Java and the underlying firmware.

In this paper, we present a middleware bridging framework called JeCOM that enables bidirectional invocation and feature replacement between Java and embedded firmware. In our specific environment, the C/C++ firmware is written to an object model known as eCOM (based on the Microsoft COM model) and thus our solution also addresses Java/COM interoperability in the embedded environment.

The contribution in this paper is the creation of a Java/COM bridge that is implemented as a new protocol in a Java-based Object Request Broker. This new protocol provides significant ease-of-use advantages over Java Native Interface, performance comparable to distributed object systems, while requiring no change to existing firmware implementation and infrastructure.

The problem of bridging Java and C/C++ is not new. However, bridging these languages in the embedded domain and addressing object discovery, lifecycle management, and memory management within the ORB protocol for bidirectional communication is new. Please see the related work in Section 6 for alternatives to our approach.

The rest of this paper is structured as follows. In Section 2, we provide the overview of the JeCOM bridge that enables Java and firmware bi-directional interaction. Section 3 presents an example to illustrate how to use the bridge to develop and deploy device features. Section 4 details the core techniques of the bridge that meets the challenges imposed by the fixed eCOM and firmware. Section 5 reports current implementation status and performance results of this embedded bridge. Section 6 identifies the related work and finally in Section 7 we conclude this paper.

2 Bi-Directional Bridging Overview

There are three popular approaches for bridging Java code to C++ code:

- Java Native Interface (JNI)
- Modified Java Virtual Machine (VisualJ++/COM)
- Java Object Request Broker (JORB)

In JNI [11], a Java interface can be implemented in C++ by using a tool to construct an additional Dynamic Link Library (DLL). This library must be deployed with the C++ code and available to the Java virtual machine for the Java class loader to properly execute. Although this is efficient, JNI introduces a versioning problem for extensible Java programs namely that they need to ensure that the embedded firmware supports the DLL containing the desired native interface. In our domain, this guarantee cannot be given for the interface may exist in one version of firmware, but not in another.

In VisualJ++/COM [18], Microsoft addressed this limitation in JNI, by extending the Java virtual machine to check for the existence of the desired class (implemented internally using `CoGetClassObject`) and to verify that the desired interface is still supported by the C++ target (using `QueryInterface`). The most significant disad-

vantage of this approach is that the virtual machine has to be knowledgeable about COM and the layout of a C++ vtable.

In JORB [15, 19], a standard wire protocol is used to bridge Java to C++ code (e.g., IIOP). This works well when the C++ code is written to the CORBA object model. When the C++ code is not in the CORBA object model, it must be wrapped with a CORBA view object. Thus Java calls the view object using IIOP and then the view object (typically written in C++) calls the intended C++ target. OrbixCOMet by IONA [7] is an example of such a system. A problem for such systems is the installation of the view object. Essentially, a view object must be installed as a DLL for each object or interface that is (or may be) exposed in the COM C++ objects. For the embedded domain, this introduces considerable memory overhead for potential access points.

In JeCOM, we address this overhead question by dynamically creating the view object. This avoids needing an explicit C++ view for each COM target and substantially reduces the memory footprint for the released firmware. A single DLL is installed when the C++ firmware is deployed. This DLL is independent of any specific firmware interface or object, but can be used by Java to communicate with any exposed COM interface. The technology is bi-directional (meaning Java can call COM and COM can call Java), thread-safe, re-entrant, and is independent of the underlying virtual machine. Java code can be used to replace, modify, or extend existing firmware components written in C++. In the rest of the paper, we refer to the path from Java to COM as the *forward bridge* and the path from COM to Java as the *callback bridge*.

2.1 Technology Choices

The architecture described in this paper is applicable to any Java ORB implementation and we believe maps to the general Microsoft COM infrastructure [12]. However, the technology used is a particular Java ORB and a proprietary embedded implementation of COM (known as eCOM). The eCOM infrastructure was developed years ago with low overhead for communication and latency. The specific Java ORB supports multiple simultaneous protocols (e.g., IIOP, SOAP, DCE-CIOP) and thus the bridge is implemented at the protocol level as an additional Environment Specific Inter-ORB protocol (ESIOP) [15]. The architecture is shown in Figure 1. The specific ORB used is a Java implementation of ORBlite [13]. HP Chai is used as the embedded JVM, which is called ChaiVM in this paper. The JORB and Java application code can be dynamically loaded into the ChaiVM. The Java client can remotely invoke firmware components via the bridge (from Point 1 to Point 2), and the firmware client can remotely invoke Java components via the bridge as well (from Point 3 to Point 4).

Also, in JORB, with the IIOP (or SOAP) communication channel, the Java components can delegate the requests originated from the firmware client to other components external to the device (from Point 1 to Point 5).

By taking the Java/eCOM bridging approach, we can keep the existing firmware code base untouched, as porting the entire code base from one object model (COM) to

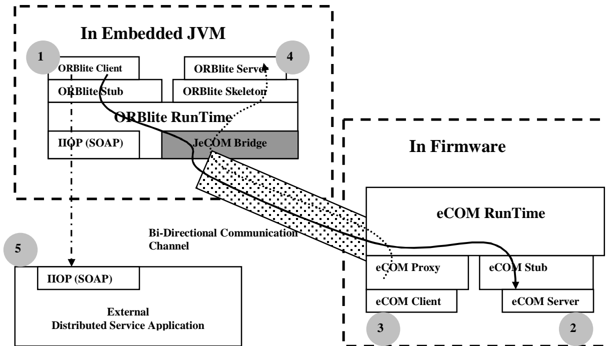

Fig. 1. The JeCOM bridge built upon ORBlite (Java ORB) to establish bi-directional communication between the embedded Java virtual machine and firmware built upon eCOM

the other (CORBA) would require tremendous effort. Furthermore, our approach works with the devices that have already been deployed.

2.2 Flow of Control in Bi-Directional Bridging

Through the forward bridge, a Java client can remotely access firmware components. After the client discovers the firmware component via the bridge, it performs a remote request to this component. The remote request flows through the ORB core runtime and reaches the specific protocol, i.e., the bridge. The bridge then establishes the native eCOM communication channel to the server based on the eCOM target object's object reference, and constructs a request message native to eCOM. After the request message reaches the remote component, and the component finishes processing the request, the bridge translates back the response message into return parameters and delivers them back to the Java client.

Conversely, the callback bridge allows an eCOM client to perform the remote request to the Java component object located in ORB, after the component implemented in Java is discovered in eCOM. The callback bridge is much more complex. During the ORB's initialization, the callback bridge is initialized after ORB creates a server thread dedicated to this protocol (bridge). Shown in Figure 2, this server thread uses JNI to migrate itself into the eCOM domain, where the thread further transforms itself into an eCOM apartment through eCOM initialization routines. This server thread is therefore both an ORB protocol server and an eCOM apartment at the same time, and is called the *Bridge Apartment Thread*. The thread of control then returns to Java.

Later, at ORB's run phase, the bridge apartment thread migrates from Java to eCOM again, and spins in a message loop. Whenever a request comes in, the message queue access returns, along with the retrieved message. The bridge decodes the message, identifies the target Java ORB object, and dispatches the call to the intended Java object. After the object finishes execution, the control falls back to the bridge apartment thread along with the returned results. The bridge translates the results into

the response message and deposits the message back to the eCOM communication channel. The eCOM runtime then delivers the response to the eCOM client. The control is returned back to the message loop, waiting for further eCOM requests.

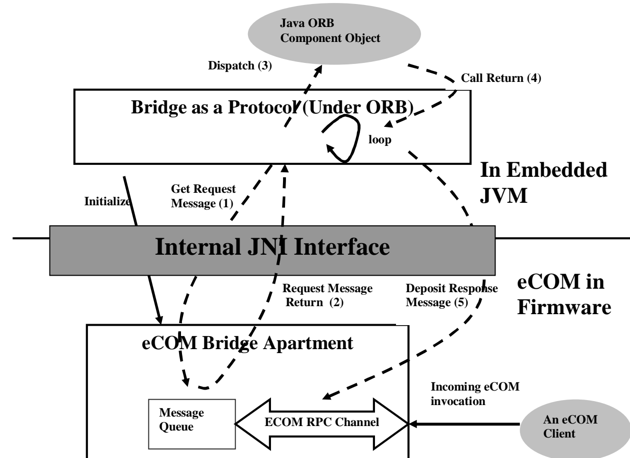

Fig. 2. The callback bridging in the JeCOM bridge

As illustrated in Figure 2 for the callback bridge, the JNI interfaces are well encapsulated in the ORB protocol. Actually, internal to the bridge, we further provide the abstraction of `open/close`, `select`, `read/write` over the RPC communication channel and channel endpoints, such that we can significantly reuse the code implementation from other protocols like IIOP and SOAP.

3 An Example of Using the JeCOM Bridge

This section provides an example to illustrate how to develop device features through the JeCOM bridge. ORBlite and eCOM are involved in our implementation as the two object systems that are bridged. This bi-directional bridge and its related compiler support for code auto-generation, allow us to develop device features in pure Java, such that no tedious and error-prone manual JNI coding is required to enable Java/C++ cross-language invocation.

We use the Component Definition Language (CDL) [4] to illustrate components and their specifications. Although the CDL is specific to our domain, it should be familiar to those knowledgeable of CORBA IDL or COM MIDL.

The example component is a network time related component called `CNT`. Its specification is shown below.

```
namespace NT {
  interface Admin {
      void GetStatus(out NetworkTimeStatus stat,
```

```
                 out NetworkTimeTimeVal time,
                 out NeworktimeConfig config);
           …
    };
};

[singleton, reentrant]
coclass CNT {
        supports NT::Admin;
        …
};
```

3.1 Feature Development

To achieve access from a Java client to the firmware, after compilation of the above CDL specifications to obtain necessary helper classes, only the simple Java code shown below is required to invoke *GetStatus*. The steps are commented inline.

```
//other imported package…
import idlGlobal.ComponentPackage.*;
import NT.*;

public class TestECOMInvocation {
 public static void main (String[] args)
   try {
     //step 1: prepare an eCOM object instance
     CNT admin=new CNTHelper();
     //step 2: prepare all necessary data structures
     NetworktimeStatusHolder status=
            new NetworktimeStatusHolder();
     NetworkTimeTimeValHolder timeval=
            new NetworkTimeTimeValHolder();
     NetworkTimeConfigHolder confval=
            new NetworkTimeConfigHolder();
     //step 3: safe cast to a specific interface
     AdminRef objRef=(AdminRef)admin;
     //step 4: invoke the method
     objRef.GetStatus(status, timeval, confval);
   }
   catch (org.omg.corba.CORBAException ex) {
    ….
   }
….
}
```

Such a simple client implementation is dramatically different from the manual JNI code already in place. The existing implementation starts with a user-defined class `NetworkTimeNativeImpl`, which contains the native method `GetStatus`. Yet another class `NetworkTimeApp` is defined to package the returned structures for `GetStatus`. No one-to-one mapping happens between the `CNT` component interface and the user-defined Java classes `NetworkTimeNativeImpl` and `Network-`

TimeApp. The JNI implementation for the method GetStatus has 142 source lines. With the JeCOM bridge, such JNI coding is completely unnecessary.

Suppose the CNT component is instead implemented in Java. We follow the required component interface to implement the component in Java. Also, on the Java side, we need to perform CNT's cross-bridge object registration, as shown below, in order to make it callable from eCOM. Its registration is hidden in the constructor of CNTFactory, a class auto-generated by the compiler.

```
public class AppServiceImpl implements IAppService,
            IChaiServer {
      public void init(….) {
        org.omg.corba.ORB.init();
        //component registration encapsulated
        CNTFactory f=new CNTFactory();
        com.hp.embedded.ORBlite.SOA.run();
        …
      }
}
```

Object registration takes place in the initialization method init associated with a Chai service, which is called AppService herein. Once this Chai service is started, and the ORBlite component is registered, the eCOM client can then remotely invoke this component, as if it is native in eCOM. Therefore, the eCOM client is completely unaware of the target component's implementation nature.

3.2 Feature Deployment

The entire JeCOM bridge is packaged as a jar file to contain Java classes and native code libraries, which is called the *bridge-related* package. Both user-defined Java implementation (including the Chai service) and auto-generated Java code are packaged in a second jar file, which is called the *feature-related* package. These two packages are loaded to the ChaiVM. When the ChaiVM starts to execute the Chai service's initialization (shown above), the service initialization further invokes Java ORB's initialization method (which includes loading the native libraries of the bridge into the ChaiVM runtime), registers the Java ORB components onto the ORB platform, and then issues run to launch the JeCOM bridge (a protocol).

Once the JeCOM bridge is running, the device can then start the bi-directional Java/eCOM remote invocation.

4 Detailed Design

In this section, we present the details on how the bidirectional communication is achieved in JeCOM. No eCOM runtime and firmware components need to be modified. The changes to the ORB necessary to introduce bi-directional bridging between Java and COM are encapsulated at the protocol level. In our Java ORB implementa-

tion called ORBlite, the bridge is implemented as yet another transport protocol, in parallel to IIOP and SOAP. We focus on how the bridge protocol addresses the following issues that we believe are essential to cross-object-system bridging:

- Discovery: how does an object in Java locate an object in C/C++ (and vice versa)?
- Translation: how are invocation parameters converted between the two runtimes?
- Lifecycle: how long do objects live?
- Concurrency: how to bridge thread concurrency models between the two runtimes?
- Memory management: how is memory management addressed for passed parameters?
- Transparency: how natural does an object in one runtime appear to the other runtime?

As background information, the ORBlite protocol abstraction layer consists of the following major interfaces:

- **`TransportInputStream`** defines marshalers for primitive types like `short` and `float`, and `TxType` (transmittable type) for composite types (structures, sequences, object types, etc.) by following the Visitor pattern [5].
- **`TransportOutputStream`** is the counterpart of `TransportInputStream` for demarshaling.
- **`TransportClient`** allows the application client to perform remote invocation to the server object, with a particular protocol derived from the corresponding IOR profile [15] of the object reference to the server object, and subsequently receives the call response.
- **`TransportServer`** is responsible for remote object registration, and remote object invocation processing by upcalling ORB component objects. The `TransportServer` can be viewed as a service in the `Service Configurator` pattern [8], whose initialization, start, and shutdown is controlled by the Object Adaptor.

4.1 Object Model

The bridge's object model aims to provide the linking between ORBlite and eCOM, the two distinct object models, and to present a uniform view of objects to both ORBlite and eCOM clients with the perception that the remote object (callee) is always native to the caller.

Object Reference Encoding

ORBlite and eCOM have completely different representations for object references. The ORBlite object reference is capable of holding profiles for multiple protocols (including the eCOM ESIOP). Therefore, a *Bridge Profile* encodes object references to the eCOM objects discovered by ORBlite. An eCOM object reference contains the information about the object's apartment thread location (identifiers for process and apartment), and the object's unique identification local to the apartment thread. The

bridge profile, also referred to as the `JNIECOM profile`, is defined with internal data structures in Java to fully store these two information pieces.

On the other hand, eCOM is not capable of holding CORBA object references, because the eCOM object reference is highly optimized (a 128 bit value). The OMG COM/CORBA specification details how a CORBA object reference can be encapsulated in a non-CORBA object reference by creating a surrogate object. We use this approach by holding the actual CORBA IOR inside a temporary eCOM object at the bridge.

The bridge creates an object instance from a component called `Universal Bridge Component` (UBC), whose component specification is the following:

```
namespace Bridge {

  interface Indexer{
        //empty, no methods declared.

  };
};


[reentrant]

coclass CUbc{
    supports Bridge::Indexer;
};
```

These UBC objects are private to the bridge. For each unique ORBlite object reference passed into the bridge, instead of encoding the information of the full ORBlite object reference, the bridge creates a UBC object instance, which is returned to the eCOM client. The bridge further transforms this UBC object reference into a JNIECOM profile, following the reference encoding mechanism described above. The bridge holds a private hash table called *Bridge Object Table* (BOT), to provide mapping between this JNIECOM profile (and therefore the native eCOM object reference)

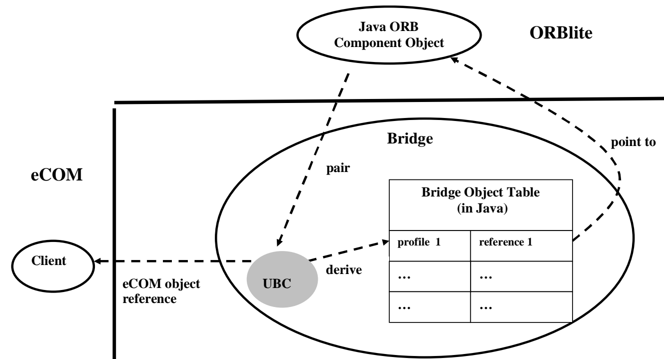

Fig. 3. Universal Bridge Component and its use in object reference encoding

and the true Java ORBlite object reference. As a result, when a remote invocation request comes from an eCOM client, the bridge is able to determine which ORBlite object is the true target object. Figure 3 shows how to use the UBC objects to fully represent ORBlite objects.

Cross-Bridge Object Discovery

Object discovery is separately resolved for the ORBlite client to discover the eCOM component, and vice versa.

An interface method called `initial_reference` is introduced in `TransportServer`, which accepts input parameters that include the CLSID of the target component. The method returns the object reference of the factory object $\mathbf{OBJ}_{\mathbf{Fac}}$ associated with the eCOM component, following the cross-bridge object reference encoding described above.

Note that in eCOM, components and their interfaces are uniquely identified by CLSIDs and IIDs respectively. Such unique identifiers are digitally signed with the keys derived from their corresponding definitions. The signing is automatically performed by the CDL compiler and therefore guarantees unique versioning of eCOM components and their interfaces.

The `initial_reference` method is invoked by the ORBlite client. The thread of control sinks down to the bridge transport, where the thread transforms itself into an eCOM thread (via `CoInitialize`). It then uses the eCOM-supported API (`CoGetClassObject`) to obtain $\mathbf{OBJ}_{\mathbf{Fac}}$ registered under the specified CLSID. The bridge then encodes $\mathbf{OBJ}_{\mathbf{Fac}}$ in a JNIECOM profile contained by an ORBlite IOR-based reference, which then returns to the ORBlite client. The resulting ORBlite object reference is called *eCOM-Encoded Object Reference*.

A helper method is actually generated by the compiler for each component to streamline `initial_reference`, and subsequent creation of the object instance $\mathbf{OBJ}_{\mathbf{ecom}}$ from the discovered $\mathbf{OBJ}_{\mathbf{Fac}}$ via the method `CreateInstance (defined in IClassFactory)`. The ORBlite user-level method invocation can then be performed over this new $\mathbf{OBJ}_{\mathbf{ecom}}$. To simplify programming, this helper method is further encapsulated in the constructor method, which belongs to a Java class auto-generated from the compiler (e.g., the class `CNTHelper` shown in Section 3.1).

To the eCOM firmware, the eCOM client still follows the native eCOM object discovery mechanism to locate a component object, even when this object is actually implemented in ORBlite. Therefore, our bridge transport is responsible for registering the ORBlite component into eCOM to provide such an illusion, as described next.

Cross-Bridge Object Registration

Object registration facilitates ORBlite components to be registered into the eCOM runtime, such that the eCOM clients can discover them. Because no firmware is modified, eCOM components do not perform this cross-bridge registration. The work has to be done in ORBlite unilaterally.

We introduce an interface method called `register_classobject` under `TransportServer`. It accepts an ORBlite singleton class factory instance (the factory-related code is auto-generated from the compiler), along with the component's CLSID. The actual registration is delegated down to the bridge. The bridge provides a private hash table called *Class Factory Table (*CFT*)* to store all these class factory instances, with their CLSIDs being the keys.

Later, before the bridge apartment thread enters its message loop to serve incoming invocation requests (in Figure 2), for each registered ORBlite component factory object $\mathbf{FCTY_{java}}$, the bridge creates a factory object instance $\mathbf{FCTY_{ecom}}$ whose component type is UBC's factory. $\mathbf{FCTY_{ecom}}$ is then registered into the eCOM's global object registry, with $\mathbf{FCTY_{java}}$'s CLSID being the registry key. Each $\mathbf{FCTY_{java}}$ is further stored into a bridge-private hash table called *Bridge Factory Object Table* (BFOT), with the corresponding JNIECOM profile derived from $\mathbf{FCTY_{ecom}}$ being the key.

Figure 4 illustrates the above steps. The details about cross-bridge object registration are encapsulated in a helper method for each component, which can be further encapsulated in the constructor belonging to a Java class automatically generated (e.g., the class `CNTFactory` shown in Section 3.1). This constructor is invoked at the application initialization to register ORBlite components to eCOM.

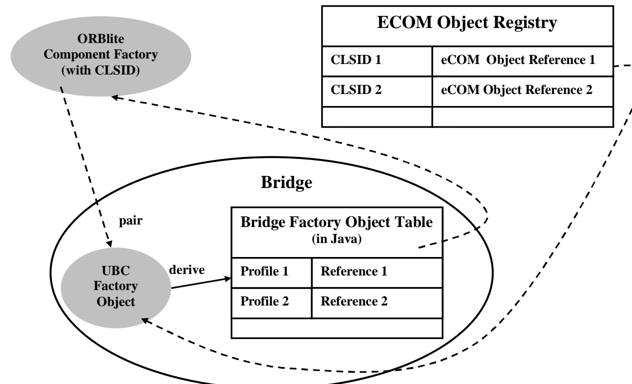

Fig. 4. ORBlite component registration into eCOM

Cross-Bridge Object Lifecycle Management

In eCOM, lifecycle management relies on object reference counting through `AddRef` and `Release` in the `IUnknown` interface. An object is destroyed whenever its reference count decreases to 0. By contrast, life cycle management service is optional in OMG CORBA [15], and currently is not supported in ORBlite.

To ensure firmware runtime correctness, eCOM object reference counting is enforced when eCOM object references cross the bridge to ORBlite. The enforcement is

performed by the bridge, completely transparent to user-level programming in ORBlite. Eventually, when the Java application finishes using the eCOM remote object, the corresponding eCOM-encoded object reference (introduced in Section 4.1) is out of scope and subsequently destroyed by the Garbage Collector. The remote object's reference count is correspondingly decreased. If no other eCOM clients hold additional references to the server object, the server object is then destroyed.

In a pure eCOM environment, the user-level application is responsible for issuing `AddRef` and `Release`, following the COM programming paradigm [1]. The eCOM runtime provides supports to ensure that reference counting is carried out on remote objects correctly. In fact, a user-level's invocation is always only an in-process invocation to the proxy. The impact of local reference counting to remote objects only happens at construction and destruction of proxy objects, from which the associated RPC channels are established or torn down, which then further triggers reference count manipulation on the corresponding eCOM stub objects. Only the reference counts of stub objects have direct impact on the lifetime of server objects. Under other circumstances, local reference counting acts independently.

The bridge performs implicit reference counting, as if the bridge is yet another eCOM apartment. Therefore, the reference counting supports provided by a native eCOM apartment and their RPC channels are mimicked. In particular, reference counting supports happen at:

- **Object Discovery and Registration.** For object registration, when the bridge creates the UBC factory objects, it increases their reference counts by 1 deliberately. Therefore these UBC factory objects are never destroyed and remain valid in the BFOT table. Object discovery deals with eCOM's class factory objects (stored in eCOM's global object registry) and therefore no specific actions are required.
- **Remote Method Invocation.** When the Java client makes a remote invocation, in `TransportClient`, right after the communication channel to eCOM is established, the reference count to this channel (and therefore to the stub of target object) is increased. Correspondingly, right after this remote invocation finishes and before the control exits `TransportClient`, the reference count to the channel (and therefore to the stub) is decreased. In `TransportServer`, the bridge processes remote `AddRef` and `Release` actions requested by other native eCOM clients. When the UBC objects' reference counts are decreased to 0, they are purged from the BOT table.
- **Object Reference Marshaler and Demarshaler.** For an eCOM object reference to be sent across the bridge and converted to an eCOM-encoded object reference in ORBlite, the corresponding `TransportOutputStream` mimics the actions related to the native eCOM object reference's demarshaling. Reference counting in eCOM's object demarshaling is then carried out. For an ORBlite object reference to be marshaled to the eCOM side, we extract the associated JNIECOM profile, convert the profile fields into the respective fields in the eCOM object reference, and then perform marshaling of the eCOM object reference. Reference counting in eCOM's object marshaling is then carried out.
- **JNIECOM Profile Finalizer.** The profile's finalizer ensures that the corresponding eCOM remote object's reference count will be decreased by 1, when the eCOM-encoded object reference in ORBlite is garbage-collected.

4.2 Memory Management for Parameter Passing

eCOM uses a shared-memory based protocol. Unlike IIOP whose marshaling is independent of native memory layout, eCOM heavily relies on shared memory to perform shallow marshaling, in order to reduce marshaling overhead. The shallowness can be very beneficial for bulk data transfer. Overall, the shallowness mandates that for a composite data type parameter, only the first-level data structures are value-copied into the communication channel's message buffer [3]. If further sub-layer data structures exist, they have to be allocated in shared memory regions, and do not directly participate in marshaling and demarshaling, except for the pointers of the top-most shared memory regions. In the extreme case, if the entire composite data structure representing the parameter is populated in shared memory, then only the pointer to the top-most structure is marshaled.

The implementation of `TransportInputStream` and `Transportoutput-Stream` collectively enforce the shallow-marshaling rules. The parameter-passing rules in [3] also demand shared-memory management in parameter passing. Fundamentally, these rules require that the client is responsible for memory allocation at the beginning of the call, and memory release when the call returns. The callee (or server object) is eligible to allocate (for `out` parameters) or modify (for `inout` parameters) shared-memory regions, on behalf of the caller.

Such rules have to be enforced in cross-bridge remote invocations, even though either the ORBlite client (to invoke eCOM object) or the ORBlite object (to serve the eCOM client invocation) is under Java, which does not have knowledge about native memory. The bridge has the implementation of `TransportInputStream` and `TransportOutputStream` to bear this responsibility.

One difficulty happens when the invocation from an ORBlite client to the eCOM object carries `inout` parameters. Suppose the bridge allocates a shared-memory region on behalf of the client, and this region is released by the eCOM object subsequently. When the call response returns, if the bridge tries to release the already-released region, memory-related failure can likely occur if the memory manager has already reclaimed this memory region. In order to have such unpredictable server-side memory modification become predictable, shared-memory reference counting is adopted (the related APIs is supported by the eCOM runtime facility). When the bridge first allocates memory regions for parameter marshaling, it stores these regions' pointers into a private table. Furthermore, the bridge deliberately increases the reference counts of these regions by 1, such that at the eCOM object implementation, the user-level's `CoTaskMemFree` (a COM API) calls cannot decrease the counts to 0. When the response comes back, the bridge retrieves the shared-memory pointers from the table, decreases their counts until the counts all reach 0, and therefore allows these regions to be reclaimed by the memory manager.

4.3 Cross-Programming-Language Server-Side Dispatching

Server-side dispatching in the callback bridge is much more difficult, compared to the client-side invocation in the forward bridge. First, the callback bridge needs to mimic

server-side dispatching of an eCOM server thread, but the target object invocation actually occurs in Java. Secondly, server-side dispatching can potentially involve more than one thread, depending on the adopted dispatching policy [6]. Thirdly, to make the implementation clean, we decide not to use the JNI's Invocation API [11]. To each interaction between Java and C/C++, the control always starts from Java, migrates into C/C++, and returns to Java.

Different types of remote invocation are involved in eCOM, including channel connection/disconnection, user-defined method invocation, remote version of AddRef/Release, invocation acknowledgement, etc. Our bridge, viewed as an eCOM server thread by the eCOM client, handles all these invocation types. Herein we only focus on method invocation. Other invocation types' handling is detailed in [10].

The handling of a method called `CreateInstance` (defined in COM's `IClassFactory` interface) is detailed next, as it requires the most sophisticated switching between Java and C/C++. The complexity mostly is because actual ORBlite object creation has to be done in Java, whereas this object (in Java) cannot be returned to the eCOM client directly. The UBC object is involved, following the similar treatment for cross-bridge object registration described in Section 4.1.

`CreateInstance` is invoked by the eCOM client after it obtains the class factory object $\mathbf{FCTY_{ecom}}$, the result of the cross-bridge ORB component's registration to eCOM. When the thread of control is in Java, the bridge apartment determines the identity of `CreateInstance` by making a call to the eCOM environment to query the interface and method identifier, based on the incoming request message. The native eCOM environment then extracts the class factory object reference $\mathbf{FCTY_{ecom}}$ from the incoming request message header, and encapsulates into a JNIECOM profile $\mathbf{P_1}$ as the return. Then in Java, the bridge searches the BFOT to identify the actual

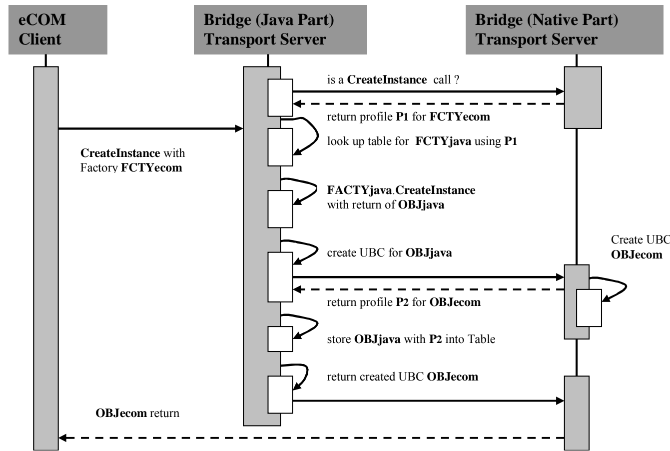

Fig. 5. Migration of thread of control between Java and native eCOM for *CreateInstance*

Java class factory object $\mathbf{FCTY_{java}}$, and subsequently invokes `CreateInstance` associated with $\mathbf{FCTY_{java}}$. The return is an ORBlite component object $\mathbf{OBJ_{java}}$.

Subsequently, the thread of control migrates back to eCOM to create a UBC called $\mathbf{OBJ_{ecom}}$ to represent $\mathbf{OBJ_{java}}$ and return a JNIECOM profile $\mathbf{P_2}$ encapsulating $\mathbf{OBJ_{ecom}}$. $\mathbf{P_2}$ then is returned back to Java. In Java, the bridge registers the ORBlite compoent object $\mathbf{OBJ_{java}}$ into the BOT table, with $\mathbf{P_2}$ as the key. To conclude this method invocation, the thread of control migrates back to eCOM, where the just created $\mathbf{OBJ_{ecom}}$ is returned to the eCOM client. The full sequence of control is shown in Figure 5.

For a user-defined method invocation from eCOM to ORBlite, the bridge handles it similarly. The `CreateInstance` method described above is substituted by a user-defined interface method. The only difference is that upcalling from the bridge to the ORBlite objects, depending on the chosen threading policy, might have object method execution happened in a different Java thread. Currently, to avoid deadlocking under recursive cross-bridge invocation, the thread-per-request policy [6], which leads to a simple implementation, is adopted.

4.4 Compiler Support

Interface specification in ORBlite is based on the OMG IDL. The eCOM is based on CDL [4] for component and interface specifications, and is augmented from the OMG IDL. We have unified component and interface specifications for both object systems under the CDL. Furthermore, both ORBlite and eCOM's compilers share the same front end, such that their code generators are just two different back-ends of the compiler.

A component therefore can be specified in CDL (one example is shown in Section 3.1), independent of the object system from which this component is to be developed and deployed. Such development and deployment choices are explicitly stated in the component's private specification that regards the component's implementation. The existing CDL supports "implemented in C++" or "implemented in C" to specify which programming language (either C or C++) is used for a component's implementation. The CDL can be extended to support "implemented in Java", to specify that this component is to be an ORBlite component implemented in Java (In our current compiler implementation, this private implementation information is instead provided to the compiler via a backend compilation flag).

The unified CDL compiler auto-generates not only traditional stubs and skeletons, but also the helper class and factory class that are necessary to incorporate cross-bridge object discovery and registration. The detailed code generation can be found in [10].

5 Results and Performance

The bridge currently is packaged into the Java and native two parts. The native part includes a shared library loadable to the ChaiVM (our embedded JVM) and two DLLs

designated for UBC (the difference between shared libraries and DLLs is symbol table stripping in DLLs). The shared library is loaded by the ChaiVM, along with other Java classes, when the related Chai service (refer to Section 3) is started (e.g., at device power up). During the bridge apartment thread's initialization, the two DLLs are further loaded via code execution in the shared library. For the MIPS processor based target platform, the entire bridge package (a jar file) is about 360KB, with the native part using the g++ compiler (with O2 turned on).

The bridge is currently running on both the HPUX development platform and the MIPS processor based target platform. We have conducted its performance measurement in the HPUX development platform (a HP 9000/78 workstation with PA 8500 550MHZ CPU, 1536MB RAM, and HPUX11i OS), which is detailed next.

In the execution environment, the ChaiVM hosting the Java application is in one process, launched by the eCOM runtime. Either the eCOM client or the eCOM server is configured in another process. Therefore, performance testing of Java→eCOM and eCOM→Java happens between processes. The results are shown in Columns 2, 3 in Table 1. Column 4 shows performance results for eCOM→eCOM invocation, with the client and the server hosted in two different processes. Column 5 aims to compare our systematic bridge with the manually written JNI bridge that is specific to this performance test suite.

Table 1. Performance measurement of JeCOM bridge in HPUX development environment (all measurement results are in millisecond. **c** stands for client, and **s** stands for server)

Method	Java(**c**) eCOM(**s**)	eCOM(**c**) Java(**s**)	eCOM(**c**) eCOM(**s**) (over native eCOM runtime)	Java (**c**) eCOM(**s**) (over manual JNI bridge)	Java (**c**) Java (**s**) (over IIOP)	
					Inside Firmware	Stand-alone
ping (0 B)	1.04	1.97	0.053	0.056	2.67	1.94
sendString (512 B)	1.63	2.17	0.070	0.179	70.1	5.97
sendStrings (8192 B)	1.78	2.73	0.080	0.342	104.7	66.3
sendInfo (90 B)	1.97	2.59	0.074	0.278	6.34	4.00
sendObjRef	2.53	2.63	0.769	0.772	4.99	3.93
getObjRef	1.50	2.85	0.673	0.712	5.12	3.61

Furthermore, to compare our bridge (an ORB protocol instance) with the IIOP (another ORB protocol instance), we also measured Java→Java remote invocation over IIOP, using the same ORB infrastructure. Since it is difficult to configure the execution environment to launch two ChaiVM processes simultaneously in our HPUX development environment, the IIOP testing is under two different configurations. The first configuration is to have the client and the server hosted in the single ChaiVM

launched by the eCOM runtime. The second configuration is to have the ChaiVM launched from a Unix shell as a standalone Unix process. Also in the second configuration, the client and the server are hosted in two different ChaiVMs at the same machine. In the first configuration, the ChaiVM has its resources (e.g., memory) constrained by the firmware launcher, in order to simulate the real embedded execution environment. Such resource constraints are not enforced in the second configuration. The measurement results of two configurations are shown in the two subcolumns in Column 6, marked as "inside firmware" and "standalone" respectively.

Following [14], we chose 6 testing methods, as shown in Table 1's Column 1, along with the related payloads (in bytes). The data in `sendInfo` is a sequence of elements each of which is a structure that further contains a sequence. In each method test case, the method was invoked for 100 times (expect for eCOM→eCOM in Column 4, and Java→eCOM over manual JNI in Column 5, 1000 invocations were chosen in these two configurations due to much smaller latencies). We then repeated the method test case 3 times, and chose the median value as the method's testing result.

The results in Table 1 show that our bridge always experiences low-millisecond-ranged latencies, for the invocations in both Java→eCOM and eCOM→Java. In particular, the latencies are quite resilient to string sizes, because marshaling in eCOM communication channels is essentially the marshaling of the pointer to the shared-memory region that stores the string content. We also observed some latency difference between Java→eCOM and eCOM→Java. The most significant factor may be due to ORBlite and eCOM's threading models. In eCOM, server threads are with the reentrant single threaded apartment model [1], and therefore no dynamic thread spawning is involved. However, in Java, the thread-per-request model is currently adopted. With thread pooling [17], the eCOM→Java invocation latencies are expected to decrease.

Notice that by taking advantage of the extremely fast eCOM runtime infrastructure (shown in Column 4), our bridge provides much faster response, compared to the IIOP protocol under the same ORB infrastructure. Such dramatic latency difference becomes much clearer when marshaling large strings. This is because our bridge relies on eCOM's shared-memory-based marshaling scheme, whereas the IIOP protocol uses CDR-based parameter serialization scheme. The difference between the two configurations in Column 6 can be attributed to constrained runtime resources in the "inside firmware" configuration, which becomes much clearer for the invocations involved with large strings.

From Column 5, we can also find that the manual JNI bridge is very efficient in terms of latencies. However, each testing method has a designated JNI interface with manual and tedious implementation, similar to the experience described in Section 3.1. We implemented in such a way that the Java client only crosses JNI to the native eCOM domain once, in order to make a native eCOM invocation.

Overall, the performance results in Table 1 indicate that our bridge, as a transport under the Java ORB, outperforms the IIOP protocol under the same ORB. Even though the JNI manual bridge for Java/eCOM invocations, or the eCOM runtime for eCOM/eCOM invocations, performs much better than the JeCOM bridge, the JeCOM bridge brings us the flexibility to develop and deploy Java-based device feature exten-

sion using CORBA, with no firmware modification required. Furthermore, since the extended device features are typically on the execution paths that are not time-critical, the low-millisecond-ranged latencies provided by the JeCOM bridge mostly can satisfy our device application needs.

Currently the JeCOM bridge allows bi-directional method invocation with primitive types, composite data types (struct, sequence, and their nested composition), and object types. It also supports code generation for singleton and factory components. The bridge implementation still needs further improvement for more features, such as marshaling of null objects, multiple-interface supports, and custom marshaling. There is only one fundamental limitation, which is about passing parameters with the `Native` type. In eCOM firmware, a `Native` parameter is actually a pointer without explicit type definition. Consequently, In Java (the other side of the bridge), without explicit type definitions, we cannot populate the actual pointed data structure via auto-generated code.

6 Related Work

In this section, we focus on the bridge frameworks that are between CORBA and COM/.NET. Overall, these bridging frameworks have been developed for desktop or enterprise applications, and therefore they are not focused on runtime execution constraints like code size and memory resource that are important to embedded devices.

IONA's OrbixCOMet [7] supports bi-directional invocation between COM and CORBA applications, conforming to the OMG CORBA/COM Interworking specification [15]. The COM applications are developed with the Microsoft IDL (MIDL) specification. Separately, CORBA applications are developed with the OMG IDL specification. A standalone IDL language conversion tool between these two IDL variants is used for cross-bridge remote invocation. The OrbixCOMet is independent of user-defined interfaces. For the invocation from COM to CORBA, such interface independency is achieved via the DII at CORBA. Standalone GUI-based configuration tools are also available to simplify cross-bridge object discovery and registration.

There exist other bridging between ActiveX and the platforms that include CORBA and Java. They do not always conform to the CORBA/COM Interworking specification. For example, Bridge2Java [2] in WebSphere facilitates Java to COM unidirectional invocation via JNI. The Java/COM bi-directional invocation is supported by J-Integra [9]. These frameworks share the common objective of reusing ActiveX components and exposing them to either Java or CORBA. As a result, automating object discovery and registration, in order to make the client application unaware of the target server's object model and implementation nature, is not supported in their bridging framework.

[14] presents a transparent integration between CORBA and .NET framework, by exploring the extensible .NET remoting infrastructure. The difference to our approach is that we rely on the ORB to establish bi-directional communication channels, and make no assumption about the other middleware runtime's extensibleness. Secondly, component and interface specification languages for the two dissimilar object systems

are unified in our environment, and we explicitly address cross-bridge object discovery and registration with both runtime and compiler support.

Microsoft Visual J++ facilitates bidirectional invocation between Java and COM [18]. The Java virtual machine requires special implementation to know COM, and expose Java objects to the COM. Thus, the bridging actually happens in the virtual machine. The associated tool produces java code that contains specific @com directives, which are compiled into byte code attributes and interpreted at runtime by the JVM. Therefore, such bridging approach requires special Microsoft programming and runtime environment.

Finally, the multi-protocol approach has been addressed to evolve communication protocols in ORBlite [13], to satisfy QoS requirements in TAO [16], and to replace RPC transports in COM [20]. In particular, half-transport [13] has been demonstrated in ORBlite to allow ORBlite to communicate with legacy applications. However, this half-transport does not consider the issues like cross-bridge object discovery and registration, and object lifecycle management.

7 Conclusion

We have designed and implemented a Java/COM bridge called JeCOM that allows bi-directional communication between Java and device firmware components built upon eCOM runtime. This bridge is built as a protocol under a Java ORB running atop the embedded Java virtual machine. This bridge allows add-on device features to be developed as Java applications and rapidly deployed onto the embedded JVM, without device firmware modification.

The key contributions of the JeCOM bridge are:

- It facilitates bi-directional communication between the two object models (i.e., CORBA and COM) that has the efficiency of JNI with the flexibility and easy-of-use of CORBA;
- It addresses object discovery, object lifecycle, and memory management requirements that are critical in the embedded domain. The solution is fully encapsulated in the ORB protocol interfaces;
- It enables rapid development and deployment of after market extensions to the base smart device by leveraging the Java virtual machine while preserving the performance of the underlying firmware.

Some lessons are learned from building the JeCOM bridge. They are applicable to a wide range of mixed language and mixed distributed object-based systems. First, by addressing cross-programming-language interoperability at the middleware level, possibly involved with middleware interoperability, we can take advantage of the infrastructures and techniques of distributed object systems to achieve a simple and flexible environment to develop and deploy applications. Secondly, a protocol abstraction, if designed correctly, can be used to enable interoperability between two different middleware runtimes, by reconciling not only messaging format difference, but also object discovery, life cycle, and memory management, etc. Finally, the bridging-related code auto-generation should be designed to encapsulate the underlying

complex method invocations in some simple methods that are exposed to user-level applications, such that end-user application development is simple, and natural to the already chosen object programming paradigm.

We have demonstrated the bridging technique in HP's embedded devices and will exploit this feature deployment platform for system testing, to ensure that embedded device reliability can be preserved, even in highly unpredictable service integration environments.

References

1. D. Box, *Essential COM*, Additioni-Wesley Pub. Co., 1998.
2. Bridge2Java, https://secure.alphaworks.ibm.com/tech/bridge2java.
3. P. Fulghum, "Parameter Passing in ECOM/CDL," HP internal design document, Mar. 2000.
4. P. Fulghum and K. Moore, "CDL," HP Internal Design Document, Dec. 1999.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Resuable Object-Oriented Software, Reading*, MA: Addison-Wesley, 1995.
6. M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*, Addison-Wesley, 1999.
7. IONA, "OrbixCOMet Desktop Programmer's Guide and Reference," http://www.iona.com/
8. P. Jain and D. C. Schmidt, "Service Configurator: a pattern for dynamic configuration of services," Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems, pp. 209-19, 1997.
9. J-Integra, http://www.intrinsyc.com/
10. J. Li and K. Moore, "Enabling Rapid Feature Deployment on Embedded Platforms with JeCOM Bridge," HP Labs Technical Report HPL-2003-256, Dec. 2003.
11. S. Liang, *The Java Native Interface Programmer's Guide and Specification*, Addison-Wesley, 1999.
12. Microsoft, *The Component Object Model Specification*, Version 0.9, 1995.
13. K. Moore and E. Kirshenbaum, "Building Evolvable Systems: the ORBlite Project," Hewlett-Packard Journal, pp. 62-72, Vol. 48, No.1, Feb. 1997.
14. J. Oberleitner and T. Gschwind, "Transparent Integration of CORBA and the .NET Framework," Proceedings of the International Symposium on Distributed Objects and Applications, Nov. 2003.
15. Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 2.4,* Oct. 2000. The CORBA/COM Interworking architecture and language mapping are addressed in Chapter 17, 18, 19.
16. C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Real-Time Distributed Object Computing Middleware," Proceedings of Middleware 2000, LNCS 1795, pp. 372-95, 2000.
17. D. C. Schmidt, "Evaluating Architectures for Multithreaded Object Request Brokers," *Commun. ACM*, Vol. 41, pp. 54-60, Oct. 1998.
18. K. Siyan, *Inside Visual J++*, New Riders Publishing, 1996.
19. S. Vinoski, "CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments," IEEE Communications Magazine, vol.35, no.2, pp. 46-55, Feb. 1997.
20. Y. M. Wang and W. J. Lee, "COMERA: COM Extensible Remoting Architecture," Proceedings of the Fourth USENIX Conference on Object-Oriented Technologies and Systems. (COOTS), pp. 79-88, April 1998.