Late Demarshalling: A Technique for Efficient Multi-language Middleware for Embedded Systems*

Gunar Schirner, Trevor Harmon, and Raymond Klefstad

University of California, Irvine, CA 92697, USA
{hschirne, tharmon, klefstad}@uci.edu
http://doc.ece.uci.edu

Abstract. A major goal of middleware is to allow seamless software integration across programming languages. CORBA, for example, supports multiple languages by specifying communication standards and language-specific bindings. Although this approach works well for desktop systems, it is problematic for embedded systems, where strict memory limits discourage multiple middleware implementations. A common memory-efficient alternative allows sharing of middleware by exposing functionality through language-specific wrappers; for instance, middleware may be implemented in C++ but exposed to Java through the Java Native Interface (JNI). Like most language wrappers, however, JNI degrades performance, especially with aggregate data types.
We present "late demarshalling": a fast, memory-efficient technique for multi-language middleware. By passing arguments at the middleware message level as a packed stream and unpacking them after crossing the language boundary, we obtain both high performance and reduced memory footprint. We provide a proof-of-concept implementation for Java and C++ with measurements showing improved performance and footprint.

1 Introduction

Middleware for embedded systems is becoming increasingly widespread and important. It plays an active role in telecommunication networks (e.g., wireless phone service), manufacturing automation (e.g., vehicle assembly lines), military defense (e.g., avionics mission control), and similar domains. A variety of middleware implementations exist, but they all share a common goal: to provide seamless integration of software across heterogeneous platforms.

On the desktop platform, a number of middleware implementations have reached for this goal. Frameworks such as CORBA [1], COM+ [2], Java RMI [3], and .NET [4] are examples of successful middleware that reduce the complexity of building distributed systems. By managing the interaction of diverse applications, usually without regard to network and platform differences, they offload

* This work was supported by Boeing DARPA contract Z20402 and AFOSR grant F49620-00-1-0330.

R. Meersman, Z. Tari (Eds.): CoopIS/DOA/ODBASE 2004, LNCS 3291, pp. 1155–1172, 2004.

many tedious and error-prone tasks from application developers and move them onto the shoulders of middleware developers.

Middleware is more than just a way of off-loading tedious chores. It can also provide enhanced features, including object location transparency, distributed event management, and language independence. In particular, support for more than one language at the middleware level allows developers to build systems by mixing and matching objects from a variety of sources, focusing on what the objects do rather than how they do it. For instance, performance-sensitive or hardware-dependent tasks could be written in a low-level language such as C, while code not bound to the CPU could be written in a simpler, more developer-friendly language such as Java. This freedom to choose the right language for the right job is one of the key advantages of middleware for distributed systems.

High-level languages like Java are certainly not a requirement for building distributed systems, but they have a distinct advantage with regard to portability and ease of maintenance. As time-to-market pressures constantly push for shorter development cycles, these productivity advantages are making high-level languages increasingly popular. Perhaps the strongest push toward high-level languages comes from an unlikely source: The new generation of university graduates, often well-versed in Java but with little training in low-level languages such as C, are driving Java toward adoption in large-scale distributed system projects.

1.1 The Challenge of Multi-language Embedded Systems

Despite the growing popularity and perceived benefits of high-level languages, bringing them to the world of distributed embedded systems is still a challenging task. The greatest challenge is the severe shortage of memory in an embedded device. This lack of resources has hindered the adoption of high-level languages in embedded systems, as well as in the middleware required to support them, simply because of their greater memory requirements. As a result, developers lose many of the benefits that come from high-level languages, including easier maintenance and shorter production cycles.

This situation must change if embedded systems developers want to reach the same level of productivity as their desktop counterparts. The challenge for researchers and the industry is to bring to embedded devices flexible and powerful middleware that can support multiple languages while maintaining the small footprint that these devices require. Unfortunately, the resource limitations are not the only problem:

- While the codebase of a desktop application usually lasts only a few years, the lifetime of an embedded system can easily reach ten or more. Because of this longevity, many existing embedded systems continue to be maintained in their original language. Thus, middleware for embedded systems will need to support old languages as well as the new.
- Even when an embedded project starts from scratch, there is a resistance to adopting new languages. Java, for instance, does not provide flexibility and

simplicity without cost: Its slower speed and increased memory requirements make it unsuitable for some embedded systems projects.
- Mixing multiple languages could greatly increase the resource requirements of the design, perhaps forcing the purchase of more expensive hardware. In the embedded market, where enough devices are produced that the cost of each individual unit becomes a crucial factor, the consequence of even a slight increase in hardware cost could be painful.

These are significant obstacles to overcome, but the potential benefits of multi-language systems are too important to ignore. A primary goal of our research is to combine all of these benefits from the desktop world—smooth migration between languages, support for high-level languages, and flexible, standard middleware—and bring them to embedded systems developers. Ideally, we want to give these developers the same freedoms that desktop developers have enjoyed for years. Accomplishing this goal is not simply a matter of optimizing memory usage of a middleware implementation or allowing Java and C to call each other's methods. As we discuss in Sect. 2, neither approach goes far enough in overcoming memory limitations while maintaining good performance. Instead, novel designs are required.

In this paper, we present one such design, which we call *late demarshalling*, to enable multi-language[1] middleware that is both time- and space-efficient. It specifically targets the performance bottleneck that exists in the boundary between two languages. By optimizing this boundary, it allows one middleware implementation to support more than one language while maintaining high speed and small size. Thus, it is a significant step forward in achieving our goal of supporting multiple languages in standard middleware for embedded systems.

2 Conventional Solutions for Multi-language Systems

Before discussing our technique, it is helpful to understand current methods for enabling multiple languages in a system and why those methods are often unsuitable for memory-constrained embedded systems. In this section, as well as the remainder of the paper, we focus on methods that depend on CORBA middleware, although the issues are relevant to virtually any type of middleware that supports multiple languages. Likewise, we present explanations and example code under the assumption that Java and C++ are the two languages we are mixing, but the problems we address can be extended to any two languages that provide a direct interface to each other.

2.1 Dual ORBs

The most common of these methods is to supply multiple implementations of middleware, one for each language in the system. The idea behind this straightforward approach is to harness the power of CORBA to do what it was designed

[1] We use the term *multi-language middleware* to mean any middleware implementation that can activate and manage objects not written in its native language.

for: bringing objects together regardless of programming language. The key benefit of this solution is that a minimum of code has to be rewritten. For example, existing Java code can be wrapped with a Java skeleton for CORBA (generated at compile time by the ORB without any effort by the programmer), while C++ code can be wrapped with a C++ skeleton, also generated automatically. At run time, the two ORBs load their respective objects and begin communicating through traditional CORBA mechanisms, as shown in Fig. 1.

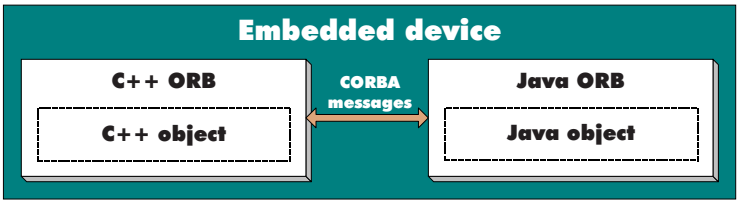

Fig. 1. Dual ORBs. Popular in desktop systems, the two-ORB approach requires two complete and independent implementations of middleware, one for each language in the system. In CORBA, for example, a C++ object and a Java object can communicate if each are registered in a C++-based ORB and Java-based ORB respectively, as shown in this illustration. The appeal of this solution is its simplicity: The ORBs can be obtained from commercial off-the-shelf (COTS) sources, and system integration requires little development effort because the objects need not be rewritten, only tied together. The problem with this approach is the large amount of memory required to support two ORBs, making it unacceptable for embedded systems.

Although this approach works well for desktop systems, which usually hold enough memory to run multiple ORBs at the same time, even one additional ORB can be too many for a resource-constrained embedded system. Although much of our current research [5] has focused on reducing the memory footprint of ORBs, it is unlikely that in the near future we will be able to place two ORB implementations on a typical embedded device. Although simple footprint reduction techniques[2] may mitigate the problem, the two-ORB solution is simply not possible when limited memory prevents it. Nevertheless, it offers good performance and a clean, low-maintenance design, so it may be a reasonable approach in embedded systems with greater memory capacity.

2.2 Simple Wrappers

To avoid the large footprint of the dual-ORB solution, a common alternative is to settle on one language for the middleware, usually C or C++, and provide

[2] The Minimum CORBA [6] specification eliminates the dynamic aspects of CORBA in order to reduce its resource requirements. Because it is static, however, it cannot adapt to the needs of the developer, possibly over-estimating the features required. Thus, even when both ORB implementations conform to Minimum CORBA, the total footprint may still be too large for an embedded system.

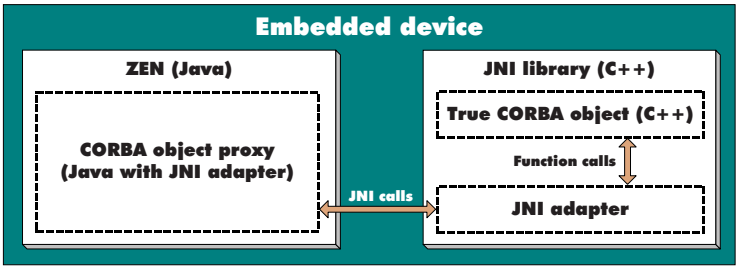

Fig. 2. Single ORB, simple wrapper. The dual-ORB solution is one way to handle both Java and C++ objects on one device, although the memory overhead of running two ORBs is excessive. As an alternative, the C++ ORB can be thrown away, and in its place, a simple Java proxy can be wrapped around the C++ object, making it appear to the ORB as a Java object. As shown in the diagram, the Java Native Interface makes this possible. The Java proxy intercepts calls sent from the ORB and forwards them, via JNI, to the true C++ object.

a simple interface to "foreign" languages. This solution has the advantage of reduced memory because only a single ORB instance is installed and running in the system. Moreover, the interface is entirely transparent to the host ORB and to the CORBA object; it is simply an inter-language adapter that forwards requests from a native-language proxy to the true object implementation in the foreign language.

Because of this clean transparency, code changes are minimal, and thus the development effort is small compared to the cost of reimplementation. More importantly, the work is most often borne by a handful of middleware developers, freeing application developers for more important work. Examples of this type of shared work can be found in omniORB [7], a C++ ORB that can register Python objects through its omniORBpy wrapper, and ORBit [8], a C ORB that can register C++, Python, and Perl objects through its orbitcpp, PyORBit, and CORBA::ORBit wrappers, respectively.

Figure 2 shows a specific example of how we have applied the same technique to ZEN, our Java-based ORB [9]. In this case, we run a C++ object in the same address space as the Java ORB, saving a significant amount[3] of memory that would otherwise be used for implementing a C++ ORB. As shown in the figure, we rely on the Java Native Interface, or JNI, as a means of crossing the language boundary between Java and C++. Although other interfaces are available for calling native functions from Java[4], JNI is the most portable and the most popular, and we focus on it exclusively for this paper.

[3] In Sect. 4, we provide detailed measurements of code size reduction.

[4] The Cygnus Native Interface, popularized by the GNU Compiler for Java, and the K Native Interface for Sun's K Virtual Machine are two alternatives to JNI.

2.3 The Problem with Simple Wrappers

Intuitively, this solution seems ideal. It avoids the memory penalty of duplicate ORBs, and it is relatively straightforward to implement. With the help of a compiler, the work of implementing the language wrapper could even be eliminated, making the technique even more attractive. For example, an Interface Definition Language (IDL) compiler that normally generates Java code could be modified to generate a JNI wrapper for C++ instead.

However, the simple JNI wrapper technique suffers from a serious performance penalty. The works of [10] and [11] have shown through quantitative analysis that JNI with native code can be slower than interpreted Java code. They emphasize avoiding JNI calls as much as possible.

With small numbers of primitive data types, such as integers or strings, minimal JNI calls are required. However, with complex aggregate data types, such as structures composed of structures, performance degrades significantly. The degradation occurs when a large aggregate data type passes through the language boundary and requires many calls to JNI.

Large aggregate data types may seem rare in everyday applications. Typical function definitions consist of just a few parameters of primitive types. Nevertheless, aggregate data types are often seen in complex programs to make them more manageable. Applying such data types is analogous to applying object-oriented design: Developers take advantage of reappearing definitions and logically group their data. This contributes to more readable code and reduces the chance of errors. As a real-world example, consider the data type definition shown in Fig. 3.

Note that the `ADSL_Line` structure (lines 14–17) is the third level of a hierarchy; it contains two structures which in turn contain two other structures. The `ADSL_Line_Card` structure (lines 23–26) makes the data type even more complex by composing a sequence of structures of structures of structures. This complexity, especially when coupled with a large sequence size, makes the data a performance bottleneck when it is passed through a language boundary.

To see why this bottleneck occurs only across a language boundary such as JNI, and not with Java objects alone, we have to examine JNI more closely. Like other Java-to-native interfaces, JNI suffers from the fact that there is no direct mapping of Java to C++. Because of this incongruity, JNI must add overhead such as:

- Locking Java arrays so that they are contiguous and immovable, allowing pointer arithmetic on them in C++
- Adding restrictions, known as "write barriers," on how and when fields in a data structure can be modified so that C++ does not interfere with Java's garbage collector
- Converting Java's big-endian data types to little-endian for C++ (if necessary)

The overhead becomes obvious in the code listing of Fig. 4, which shows an example of how C++ would access the data type of Fig. 3 via JNI. Note that access to a class field requires three function calls: one to get a handle to the

```
1  class ADSL_Line_End_RAW {
2    // Low-level line information (signal loss, signal-to-noise ratio)
3  }
4
5  class ADSL_Line_End_ATM {
6    // ATM related line information (link loss, cell count)
7  }
8
9  class ADSL_Line_End {
10   ADSL_Line_End_RAW raw;
11   ADSL_Line_End_ATM atm;
12 }
13
14 class ADSL_Line {
15   ADSL_Line_End near;
16   ADSL_Line_End far;
17 }
18
19 class Line_Card {
20    // Card-specific data (power consumption, uptime)
21 }
22
23 class ADSL_Line_Card {
24   Line_Card card;
25   ADSL_Line[] line;
26 }
```

Fig. 3. An aggregate data type. Large aggregate data types are not uncommon, especially in object-oriented or otherwise hierarchical organized programs. These definitions exemplify the performance monitoring data types of an Asymmetric Digital Subscriber Line (ADSL) line unit (e.g. following the G.992.1 standard, published by the International Telecommunication Union). Embedded software that implements the performance monitoring improves its readability and maintainability by composing several structures into a single aggregate structure: `ADSL_Line_Card` in this example.

class (lines 9 and 18), one to get a handle to the JNI-specific type identifier (lines 10–11 and 19–20), and another to obtain the field's value (lines 14–15 and 23–24). As our measurements in Sect. 4 reveal, the loss in performance due to this overhead is significant and is directly proportional to the number of JNI calls.

3 The Late Demarshalling Solution

To alleviate the performance limitations of this simple wrapper approach, the number of function calls crossing the language boundary must be minimized. We have developed a technique, which we call *late demarshalling*, that reduces the number of JNI calls to just two, regardless of data type complexity. We have seen performance increases of up to three times for complex data types such as the one shown in Fig. 3. Although the technique does not improve performance for simple data types passed to and from objects, it does not decrease performance in this case.[5]

[5] For local objects, however, where a C++ and a Java object reside in the same address space, the simple wrapper approach holds an advantage when simple data types are

```
1  jfiedlID nearID, farID, rawID, atmID;
2  jclass lineCls, endCls, rawCls, atmCls;
3  jsize length = env->GetArrayLength(lineArray);
4
5  for (i = 0; i < length; i++) {
6   jobject obj = env->GetObjectArrayElement(lineArray, i);
7
8   if ( 0 == i ) {
9     lineCls = env->GetObjectClass(obj);
10    nearID = env->GetFieldID(lineCls,"near","LADSL_Line_End;");
11    farID  = env->GetFieldID(lineCls,"far","LADSL_Line_End;");
12  }
13
14  jobject near = env->GetObjectField(obj, nearID);
15  jobject far  = env->GetObjectField(obj, farID);
16
17  if ( 0 == i ) {
18    endCls = env->GetObjectClass(near);
19    rawID = env->GetFieldID(endCls,"raw","LADSL_Line_End_RAW;");
20    atmID = env->GetFieldID(endCls,"atm","LADSL_Line_End_ATM;");
21  }
22
23  jobject raw = env->GetObjectField(obj, rawID);
24  jobject atm = env->GetObjectField(obj, nearID);
25  ...
```

Fig. 4. The cross-language bottleneck in detail. This code snippet provides some insight into why cross-language interfaces such as JNI perform poorly. The code shows how middleware applying the simple wrapper technique would extract data from an array of `ADSL_Line` objects as in line 25 of Fig. 3. Note that each field access requires two function calls: one to retrieve a handle to the field's type (lines 10–11 and 19–20) and another to retrieve the field's data (lines 14–15 and 23–24). Although the initial function call is necessary only in the first iteration of the loop, the total overhead of these multiple function calls adds up to a considerable performance penalty.

Our technique minimizes cross-language calls by taking advantage of a standard middleware practice known as *marshalling*. When middleware prepares to invoke a remote method, it packs (or marshals) the parameter data into a message stream called, in CORBA parlance, the Common Data Representation (CDR). The stream is then routed through the network to the remote object. On the server side, the CDR stream is unpacked, and the original parameters are recreated.

With the simple wrapper approach, this demarshalling occurs as soon as the middleware receives the data, as shown in Fig. 5. This results in two layers of overhead: one to convert the CDR stream into the middleware's native-language data types, and another to convert these native-language data types into the data types of the foreign language. For instance, with Java middleware and C++ objects, a stream is translated into a Java class, and then the Java class must be translated again into a C++ class. These redundant translations contribute

passed between them. Our late demarshalling technique does not improve performance in this case until data types become significantly complex, as described in Sect. 4.

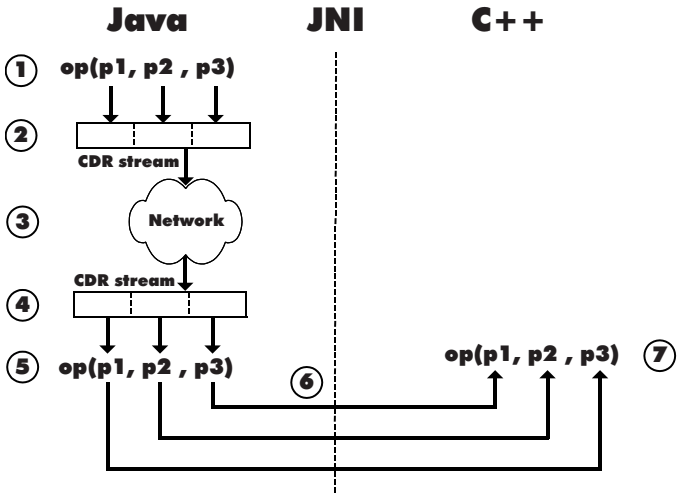

Fig. 5. Simple wrapper in detail. In the simple wrapper approach to multi-language middleware, the parameters of a function call flow from a Java object to a C++ object in the following sequence: 1) The Java object calls a stub acting as the target object; 2) the stub marshals the parameters into a CDR stream; 3) the middleware sends the stream to the target object; 4) the target object's skeleton demarshals the parameters from the CDR stream; 5) the skeleton calls the object proxy; 6) the proxy calls the C++ object implementation via JNI; 7) the C++ implementation uses JNI to retrieve each parameter one at a time.

to substantial overhead, especially for the final translation, which may require a large number of JNI calls for complex data types.

In contrast, our technique postpones demarshalling until the moment it is needed. We alter the proxy servant shown in Fig. 2 to send the entire CDR stream, unmodified, through the language boundary. As illustrated in Fig. 6, we then demarshal the parameters once the stream has arrived on the other side of the boundary. No additional calls to the cross-language interface are necessary, and the object can access the parameters directly and naturally in its native language.

Compared to the traditional approach using a simple wrapper, our late demarshalling technique provides three key advantages:

- Eliminates the step of converting the CDR stream into data types of the middleware's native language
- Reduces cross-language function calls to two: one to lock the CDR stream in memory and another to release it
- Performs demarshalling in the target object's native language, and in the case of Java middleware with C++ objects, gains an increase in speed

Together, these advantages add up to significant gains in performance, verified by the measurements shown in Sect 4. Like the simple wrapper solution, we

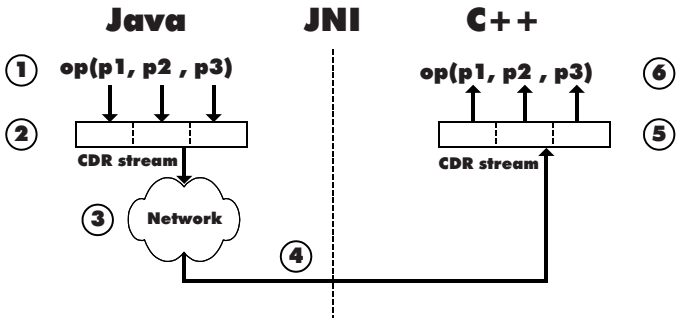

Fig. 6. Late demarshalling in detail. In our late demarshalling technique, we minimize the overhead of the language barrier by postponing demarshalling until after the CDR stream has crossed the language barrier. The parameters of a late-demarshalled function flow from a Java object to a C++ object in this sequence: 1) The Java object calls a stub acting as the target object; 2) the stub marshals the parameters into a CDR stream; 3) the middleware sends the stream to the target object; 4) the target object forwards the CDR stream directly to the C++ object's skeleton via JNI; 5) the C++ object's skeleton demarshals the parameters from the CDR stream; 6) the skeleton calls the C++ object's implementation.

require only one middleware implementation, and thus we maintain the advantage of low memory footprint.

4 Empirical Results

In order to verify that our late demarshalling technique is a significant advance for multi-language middleware, we performed a series of benchmarks and code size measurements.[6] The data show that late demarshalling combines the best of both worlds: the small footprint of the simple wrapper approach plus the high performance of dual middleware implementations. In this section, we present and analyze these results.

4.1 Test Setup

Because multi-language middleware is vulnerable to the complexity of data crossing the language boundary, we focused on the performance of the three techniques as a function of data type complexity. We quantify the "complexity" of a data type according to the number of cross-language calls that are required for the simple wrapper solution. In the case of Java and C++, the number of JNI calls represents complexity very closely; we observed in multiple experiments that the performance of each solution follows a trend according to the number of JNI calls.

[6] The source code to the full benchmark suite can be found in the `perf.JNI` package of the ZEN distribution, available at `http://www.zen.uci.edu`.

To obtain sufficient data points, we selected a wide range of types for our tests, from simple primitives to complex aggregates. Because the size of the data influences performance as well, we also selected various sizes of each data type by defining a sequence of the type and varying its length. We ensured that the benchmarks for different complexities are comparable by scaling the sequence length so that the total size of the aggregate remains constant for a given test.

We then constructed the aggregate data types bottom-up. That is, more complex data types were composed of one or more simpler data types. Because of this inheritance, we present here only the most complex type, `SequenceF`. Figure 7 shows a graphical depiction of `SequenceF`.

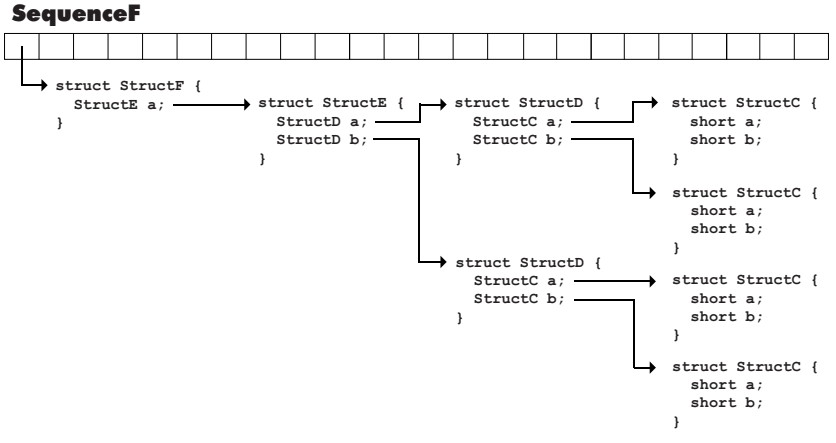

Fig. 7. A complex data type for benchmarking. `SequenceF`, the most complex data type used in our tests, consists of a sequence of structures. Each structure contains a structure of two structures of two structures of two 16-bit integers.

The simpler data types are smaller versions of the structure hierarchy depicted in the figure. For example, `SequenceD` contains elements of type `StructD`, which contains a sequence of `StructC`. This pattern of decreasing complexity continues until `SequenceA`, the simplest data type consisting only of 16-bit integers, or shorts.

We tested our proof-of-concept implementation on ZEN, our Java-based ORB, extended to handle C++ servers. In each test, we transferred a sequence of each data type from a client to a server with the configurations described in Fig. 8. The testbed for all configurations was a dual-processor 1.7 GHz Intel Xeon system with 1 GB of RAM running a Debian distribution of Linux with kernel version 2.4.18. All C and C++ code was compiled with GCC 3.3.3. All Java code was compiled with AspectJ 1.1 and executed under the Sun Java Virtual Machine 1.4.2. The CORBA stubs and skeletons for C++ were generated

with omniORB 4.0.4; for Java, they were generated with ZEN's default compiler, OpenORB 1.3.1.

	Dual ORBs	**Simple Wrapper**	**Late Demarshalling**
Client ORB	ZEN 1.0.1	ZEN 1.0.1	ZEN 1.0.1
Client Language	Java	Java	Java
Server ORB	omniORB 4.0.4	ZEN 1.0.1	ZEN 1.0.1
Server Object Language	C++	C++	C++
Server Object Proxy	None	Simple wrapper with JNI	Late demarshalling JNI wrapper with CDR stream library from omniORB

Fig. 8. Test configurations. This table shows the basic software setup that we used for testing the three basic approaches toward multi-language middleware.

The client, based on ZEN, was identical for all tests, but the server changed according to the method. For example, the dual-ORB approach uses a C++-based ORB to load the C++ server object and therefore requires no proxy. The simple wrapper and late demarshalling approaches use a Java-based ORB and thus require a proxy to cross the language barrier into C++.

To implement these proxies, we modified the skeleton code that had been generated from the IDL we had written for our data types. For the simple wrapper implementation, we added the minimum number of JNI calls necessary to extract the data from the Java-based middleware and transfer it to our C++ server.[7] For the late demarshalling implementation, we added two JNI calls—one to lock the CDR stream in memory and another to release it—and simply forwarded the stream to omniORB's CDR stream demarshalling library. After omniORB unpacked the data from the stream, we forwarded it to the appropriate method of the C++ server.

4.2 Round-Trip Time Measurements

For the performance tests, a common client implemented in Java using the ZEN ORB invoked a method on the server of each configuration 1000 times for each data type. This set of benchmarks was performed under two types of environments:

Remote: Client and server in separate ORB instances. This represents the typical middleware setup in which two objects in different hosts communicate over a network. For our tests, we prevented interference due to network traffic by executing both ORB instances on the same host and directing messages through TCP via the local loopback interface.

[7] Although we refer to "client" and "server" when describing our test setup, the difference between client and server objects is only contextual in this case. Although we tested server implementations exclusively, all of the methods for multi-language middleware are symmetric and apply to foreign-language clients as well as servers.

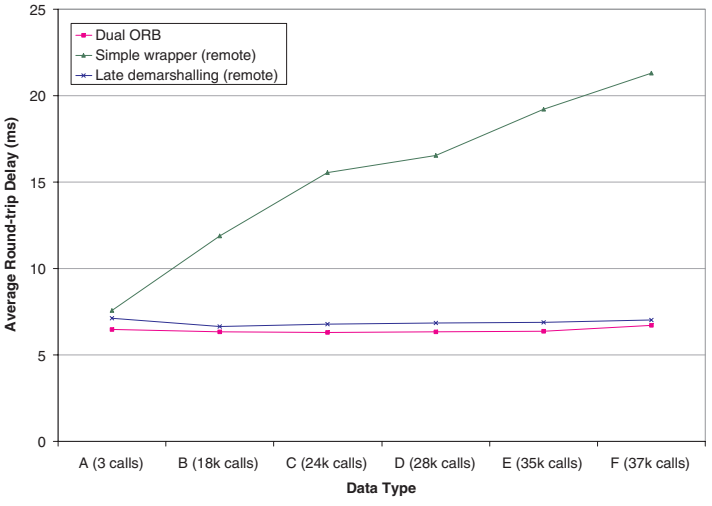

Fig. 9. Performance versus complexity for remote calls. This graph shows the average round-trip delay of a single remote method call as the complexity of the method parameter changes. The horizontal axis represents individual data types sorted by complexity; values in parentheses indicate the number of cross-language calls required to retrieve the data in the simple wrapper approach. The total amount of data for each aggregate data type remains constant at 32 KB to allow comparisons between complexities.

Local: Client and server in the same ORB instance. Although not as common as the remote case, objects may sometimes reside in the same host and in the same ORB instance. Many implementations of middleware perform co-location optimizations in this case.

Figure 9 shows the average round-trip time for the remote case. Of the two memory-efficient single-ORB solutions (simple wrapper and late demarshalling), late demarshalling outperforms the simple approach in all complexities. For simple data types, such as type A, the performance advantage is small; passing a few simple types through JNI does not add significant overhead. However, as complexity increases, the advantage of late demarshalling grows linearly. While the simple wrapper approach must increase its calls to JNI, the number of JNI calls in the late demarshalling approach remains relatively constant.

The performance advantage of late demarshalling can also be attributed to the speedup of C++ versus Java: In this particular setup, CDR stream demarshalling occurs in Java for the simple wrapper approach and in C++ for late demarshalling. We have seen from additional measurements that Java contributes to approximately one-third of the performance penalty for simple wrapping in this case. Note that the baseline measurement, the dual-ORB approach, gives

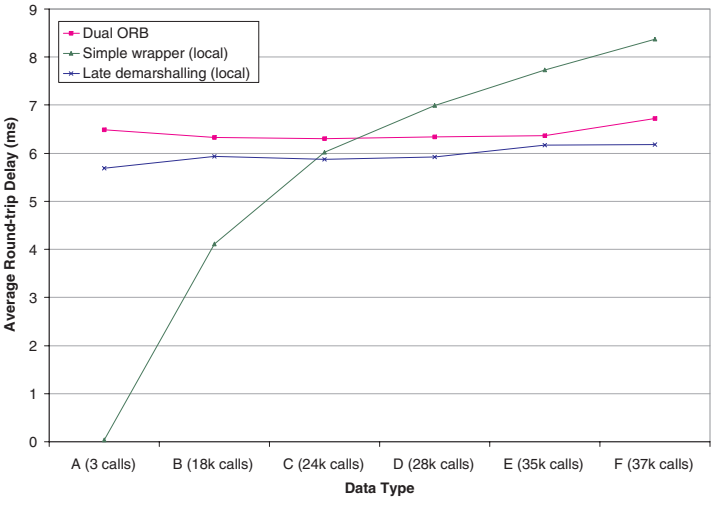

Fig. 10. Performance versus complexity for local calls. This graph is a complement to Fig. 9. The tests in this figure are identical except that they were performed locally with the client and server in the same address space.

slightly better performance in all cases. It contains only C++ code on the server side and therefore does not suffer from any cross-language overhead.

The second set of tests, shown in Fig. 10, represents the local case in which client and server reside in the same address space. The graph reveals the effect of co-location optimizations where the client and server can communicate directly. As a result, the ORB creates no CDR stream, and the method call requires only a small amount of overhead. The simple wrapper approach profits from this optimization, making it faster than late demarshalling for simple data types (such as the sequence of integers in type A and the sequence of simple structures in type B).

As the data grows more complex, however, the overhead of CDR stream demarshalling remains almost constant. This helps late demarshalling outperform the simple wrapper approach starting with moderately complex data structures (in particular, the two-level structure of type C). Note that the dual-ORB approach also benefits from the near-constant overhead of CDR, although it still performs worse than late demarshalling because it cannot take advantage of co-location optimizations.

The results presented thus far are based on constant data size and variable complexity. In contrast, the graph of Fig. 11 shows how the three methods perform with variable data size and constant complexity. With the selected complexity, `SequenceE`, data sizes below 512 bytes can cross the language boundary faster with the simple wrapper approach. Late demarshalling and the dual-ORB approach are limited by the overhead of unpacking the CDR stream.

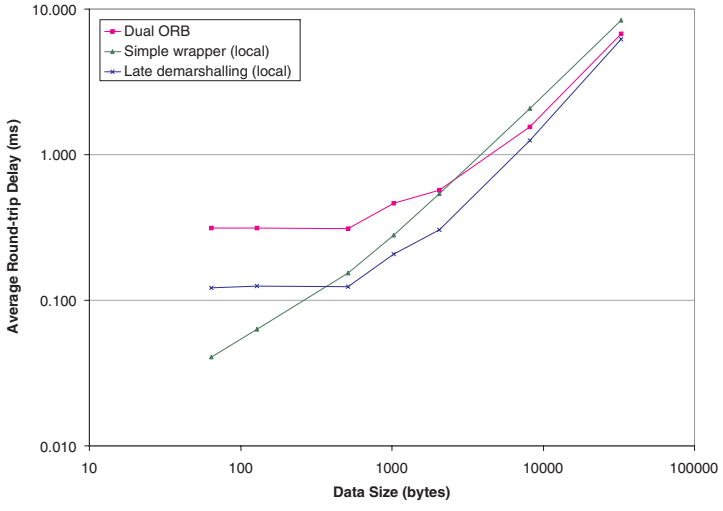

Fig. 11. Performance versus data size. This graph shows the average round-trip delay of a single remote method call for the `SequenceE` data type as the size of the data changes. A logarithmic scale is used for both axes in order to make the differences visible over a large range of data sizes (64 bytes to 32 KB).

With data sizes greater than 512 bytes, the situation reverses. Both late demarshalling and the dual-ORB approach outperform the simple wrapper due to the linearly increasing effort of passing parameters through the language boundary and the near-constant effort of demarshalling the CDR stream. Note that the dual-ORB approach cannot match the performance of late demarshalling because it must send its CDR stream between processes via the TCP local loopback interface.

We have measured the performance of other data types for varying sizes, but we do not present them here. They differ only in the cross-over point at which simple wrapping and late demarshalling have equal performance. This point is reached earlier for more complex data types.

4.3 Footprint Measurements

A second aspect of our measurements is footprint, an important concern for memory-constrained embedded systems. Because all solutions require some form of Java, we do not measure the footprint of Java support code (such as a virtual machine or API library) because it is a constant factor. Instead, we focus on the increase in footprint due to multi-language support. We divide our footprint measurements into three categories: built-in ORB code, marshalling, and dynamic memory.

Built-in ORB code. An ORB contains internal code and built-in functions that provide basic CORBA features independent of user-defined code. These functions manipulate object references, query services, and perform other necessary housekeeping tasks for CORBA objects.

For the dual-ORB solution, these built-in functions are the main reason for the hight footprint requirements; they are duplicated in both ORBs. Even when both ORBs conform to the Minimum CORBA [6] standard, such duplication can increase the code size to a point where this solution is simply not practical for embedded systems. TAO [12], for example, a popular C++ ORB, has a 1.9 MB code footprint [13] in its Minimum CORBA configuration.

The single-ORB solutions (simple wrapper and late demarshalling) do not suffer from this code duplication. However, they require cross-language wrappers that expose the native ORB's built-in functions to foreign languages. Measuring the true cost of this overhead is a massive undertaking because wrappers for each of the more than 200 built-in ORB functions (as specified by the CORBA standard) must be implemented and measured. We have targeted this task for our future work.

To gain preliminary results, we have estimated the size of the wrapper footprint without a full implementation of all function wrappers. We began by separating the functions into categories based on their unique parameter signatures. We then implemented a single generic wrapper for each of the 45 categories and measured its code size. Finally, we multiplied the size by the number of functions in its category and then took the sum. Assuming that wrappers of identical signatures have similar code size, we concluded that exposing the built-in ORB functions of a Java ORB using JNI would require approximately 150 KB.

Marshalling. In addition to built-in ORB functions and user-defined code, marshalling (and demarshalling) algorithms are also contributors to code size. For the dual-ORB and simple wrapper solutions, these algorithms are built-in to the ORB and are accounted for in the code size measurements of the ORB itself. The late demarshalling solution, however, requires multiple implementations of marshalling, one for each supported language. Our late demarshalling prototype, for example, relies on the built-in routines of ZEN for Java marshalling and supports C and C++ by grafting omniORB's marshalling routines onto ZEN. Our measurements show that these extra marshalling routines add about 218 KB to the code size. This relatively large size is due to the thorough implementation of the routines, including full read/write support for Unicode and international character code sets.

Dynamic memory. Dynamic (run-time) memory footprint is a concern, but we did not measure dynamic memory requirements in this first phase of our research. However, dynamic memory footprint can be inferred without measurement because it follows the same pattern as code size. In the dual-ORB solution, for example, the second ORB instance greatly increases the amount of dynamic memory required for runtime data structures (e.g. message buffers), just as it

increases the code size. Likewise, the simple wrapper and late demarshalling solutions, which need only one ORB, require much less dynamic memory overhead. They leverage the resources of the original ORB and therefore have a significant advantage in dynamic memory footprint as well as static code size.

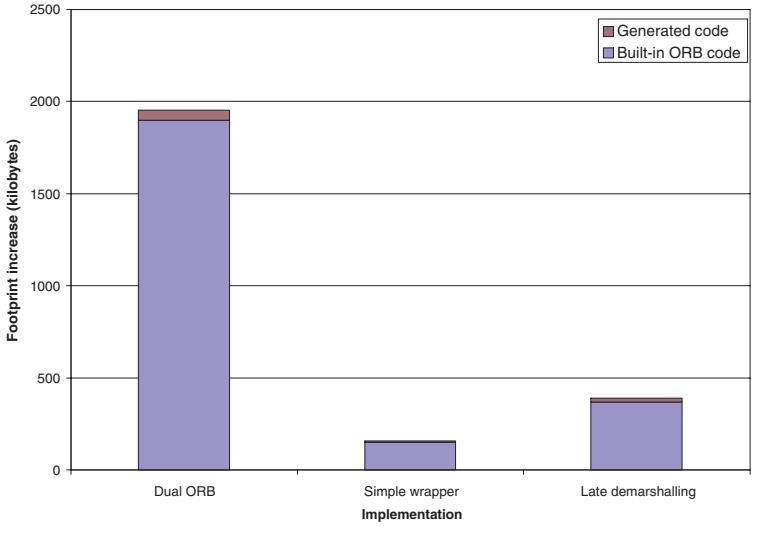

Fig. 12. Code size versus implementation type. This diagram shows the increase in code size required to support a second language (C++) on top of the native language (Java) in the three approaches to multi-language middleware. Generated code refers to code generated by an IDL compiler to support user-defined application interfaces, while built-in ORB code includes the CORBA API and marshalling/demarshalling support.

As shown in Fig. 12, the simple wrapper offers the lowest footprint because it adds only enough wrappers to expose built-in ORB functions to C++ servants. The late demarshalling solution requires a larger footprint to support marshalling in C++. Both approaches, however, easily outperform the dual-ORB solution, which requires more than four times the space.

The generated code yields similar results. At 54 KB, the stub/skeleton code generated by TAO is the largest because it must provide a complete implementation for client and servant. The simple wrapper, a minimal implementation in C, has the smallest footprint at 8.6 KB. The late demarshalling solution is slightly larger at 20 KB because its C++ implementation uses the complete skeleton framework from omniORB.

5 Conclusion

Late demarshalling is an efficient method for multi-language middleware. Our measurements have shown that it performs better than conventional methods of multi-language middleware in the case of remote objects. Even for local objects that employ co-location optimizations, late demarshalling is a performance improvement for complex aggregate data types. Furthermore, late demarshalling provides a reduction in footprint comparable to the simple wrapper approach.

In our future work, we intend to expand the proof-of-concept implementation into an optional feature of our ZEN middleware, including IDL compiler support for the late demarshalling technique. In a second step, we will extend the IDL compiler to detect when a data type is too simple to benefit from the late demarshalling, and in such a case, the compiler will generate simple wrapper code for local invocations, taking advantage of both approaches.

In other avenues of research, we will measure pre-compiled Java solutions and investigate how other Java-to-native interfaces compare to JNI. We will also add footprint measurements for specific embedded Java solutions such as Sun's K Virtual Machine. We believe that research in these directions will show that late demarshalling is valuable outside of the middleware domain.

References

1. Group, O.M.: CORBA: Core specification (2004)
2. Kirtland, M.: Object-oriented software development made simple with COM+ runtime services. Microsoft Systems Journal (1997)
3. Grosso, W.: Java RMI. First edn. O'Reilly & Associates (2001)
4. Lowy, J.: Programming .NET Components. O'Reilly & Associates (2003)
5. Klefstad, R., Krishna, A.S., Schmidt, D.C.: Design and performance of a modular portable object adapter for distributed, real-time, embedded CORBA applications. In: Proceedings of the Distributed Objects and Applications conference. (2002)
6. Group, O.M.: Minimum CORBA specification (2002)
7. Lo, S.L., Pope, S.: The implementation of a high performance ORB over multiple network transports. In: Middleware '98. (1998)
8. Lee, E., Dick Porter, e.a.: ORBit2 (2004)
9. Klefstad, R., Schmidt, D.C., O'Ryan, C.: Towards highly congurable real-time object request brokers. In: Proceedings of IEEE International Symposium on Object-Oriented Real-time Distributed Computing. (2002)
10. Kurzyniec, D., Sunderam, V.: Efficient cooperation between Java and native codes – JNI performance benchmark. In: Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001). (2001)
11. Welsh, M., Culler, D.: Jaguar: Enabling efficient communication and I/O in Java. Concurrency: Practice and Experience **12** (2000) 519–538
12. Schmidt, D.C., et al.: TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. IEEE Distributed Systems Online **3** (2002)
13. Schmidt, D.C.: Minimum TAO (http://www.cs.wustl.edu/~schmidt/ace_wrappers/docs/minimumtao.html) (2004)