Babel Remote Method Invocation

Gary Kumfert[1], James Leek[1], and Thomas Epperly[1]

[1]Lawrence Livermore National Laboratory
P.O. Box 808, Livermore, California 94551 USA
{kumfert, leek2, tepperly}@llnl.gov

Abstract

Babel is a high-performance, n*-way language interoperability tool for the HPC community that now includes support for distributed computing via Remote Method Invocation (RMI). We describe the design and implementation of Babel RMI, including its specification in our Scientific Interface Definition Language (SIDL), modifications to Babel's code generators, and support for third-party wire protocols. Babel RMI's programming model consistency, functional capabilities, and runtime performance are compared in context with COM, CORBA, Grid/Web Services, and Java RMI. Babel RMI's current features and performance uniquely recommend it for "short-haul" distributed computing within a machine room or single cluster. We describe the experience of some early adopters who use Babel RMI to couple and coordinate multiple MPI jobs on a single cluster to perform multiscale material science calculations.*

1. Introduction

This paper introduces a major new feature for Babel—support for distributed computing via Remote Method Invocation (RMI). Previously, Babel was strictly a language interoperability tool, allowing software written in various languages (currently Fortran 77, Fortran 90/95, C, C++, Python, and Java) to be simultaneously and arbitrarily mixed in a single address space for maximal performance [6, 14, 21]. Babel has been successfully used by applications [15, 20, 22, 27], community standards groups [28, 34], and component frameworks [1, 2]. The Common Component Architecture's impact on computational science [23] is also relevant as it is built on Babel technology.

There is a great body of existing work on distributed computing. Babel RMI is a newcomer and a niche player in this field. Hence, we present our design and implementation of Babel RMI in constant comparison with established distributed systems, such as DCOM, CORBA, Web Services, and the Grid. Though Babel's technical lineage is arguably closest to CORBA, our HPC customer-driven design frequently deviates from convention.

Babel's support for true object-oriented semantics and dynamic types is more akin to Java RMI [37] than either DCOM or CORBA, the latter two being more accurately characterized as more procedural than object-oriented [35]. Babel RMI does not prescribe a specific protocol between caller and callee. Instead, we define a language neutral messaging API using our own Scientific Interface Definition Language (SIDL). This lets users implement the wire protocol of their choice in the language of their choice to support the Babel RMI semantic. A simple TCP/IP-based protocol is distributed as a reference implementation.

We start laying the foundation for our technical discussion in Section 2, which contains a brief survey of comparable technologies. Section 3 starts introducing Babel-specific details to keep the paper self-contained. Presentation of the new RMI capability is provided in Section 4. The results in Section 5 include a simple comparison of latency and throughput for Babel RMI and comparable technologies. This section also reports some initial experiences of our internal customer, the Petascale Simulation Initiative (PSI) [31], which uses Babel RMI to communicate and manage multiple MPI jobs on a single cluster. We conclude in Section 6 with a brief summary and statement of future research and development.

2. Related Work

Remote Procedure Call (RPC) was first proposed by Birrel and Nelson in 1984 [8]. In the early 1990s, the Object Management Group (OMG) specified a standard for remote objects, the Object Management Architecture (OMA) [33],

1-4244-0910-1/07/$20.00

of which CORBA is a fundamental part. CORBA's communication layer is known as the Internet Inter-Orb Protocol (IIOP). In 1996, Microsoft released DCOM, a distributed version of their Component Object Model (COM) that employs the Open Group's Distributed Computing Environment (DCE) RPC architecture [32]. Both COM and CORBA have Interface Definition Languages (IDLs) to generically specify the component interfaces. Like Babel, both employ source-code generators to create the client- and server-side support in a variety of programming languages.

Sun introduced Java Remote Method Invocation (Java RMI) in version 1.1 of their Java Developer's Kit (1997) [35]. Instead of IDLs and code generation, Java RMI relies on Java's rich introspection and virtual machine capabilities to marshal bytecodes across a network and support remote execution. Two most common protocols for Java RMI are the proprietary Java Remote Method Protocol (JRMP) and the CORBA-compatible RMI-IIOP. In response, work was done on CORBA to add a pass-by-value capability [29, §10.7.2].

Web Services (WS) is a technology stack of standards and protocols to support machine-to-machine interaction over a network. All data is exchanged in text using XML. The messages typically conform to the SOAP standard [19], though older XML-RPC [36] is not precluded. Transmission can occur via any number of Internet protocols including FTP or SMTP, but HTTP is by far the norm. The interfaces are typically encoded in Web Services Description Language (WSDL) [11], which serves a similar function to IDLs. Service providers publish their capabilities to brokers according to the Universal Description, Discovery, and Integration (UDDI) [12] specification. There are a wide variety of third-party tools that will consume WSDL and generate the glue code (in the programming language of choice) to produce and respond to the SOAP messages that are exchanged between services at runtime.

WS is no more powerful than other distributed object systems, such as CORBA [18], but is often favored for its transparency and reliance on abundant web technologies. WS's strength is in asynchronous, latency-tolerant applications such as job scheduling across a wide area network. It emphasizes document exchange and a stateless service-oriented architecture. The major downside of WS is poor performance relative to other distributed systems. The trade off is not unexpected since XML is known to be verbose and inefficient for encoding/decoding scientific data [9].

The Global Grid Forum (GGF) is the community of researchers and vendors that develop and promote grid computing. Their Open Grid Services Architecture (OGSA) specification [17] defines standards for security, execution management, data management, monitoring, discovery and many other details in support of a coordinated computational resource that broadly delivers nontrivial computational services and transcends centralized control.

3. Babel Background

Babel is predicated on two assessments of High-Performance Computing software technology (HPC): (1) that component technology promises to solve critical software infrastructure problems, but (2) commercial offerings don't span the breadth of languages, platforms, simplicity, and performance needed by the broad HPC market [5].

Babel development started internally in 1999 [13] and has contributed to the Common Component Architecture (CCA) Forum since its inception. We made 30 formal releases of Babel in the runup to version 1.0 in 2006. The same year, Babel won an R&D 100 award for its superior in-process performance. We created our own "Scientific" IDL (SIDL) with intrinsic support for complex numbers and multidimensional arrays. Formally, SIDL is the language and Babel is a tool implemented to support the language. Since we maintain, amend, and extend the two in lockstep, it is not uncommon to see the two used interchangeably in literature or simply combined as "SIDL/Babel."

There are six guiding design principles that have shaped Babel and contributed to its quality and growing use.

1. **High Performance**
2. **Consistent** — A feature gets into SIDL if and only if it can be supported in every language.
3. **Portable** — Babel must work with the customer's set of compilers, linkers, debuggers, etc. It can require special flags (or demand certain flags be avoided).
4. **Reliable** — Babel needs to be as reliable as the compilers it is built upon. Coupled with Principle 3, we are confronted with an exceptional configuration and build challenge [24].
5. **Simple** — Babel must be clear and simple to non-computer scientists.
6. **Idiomatic** — Subject to Principles 1 & 2, each language binding should appear as natural as possible to seasoned programmers in that language.

For complete and definitive technical information about SIDL or Babel, see the Babel Users' Guide [14]. The rest of this section provides only the necessary details to frame the discussion of Babel RMI in Section 4.

3.1. Essential SIDL

SIDL is used to specify *user-defined types* and the interactions between them in a language- and platform-neutral manner. The entire SIDL type system is divided between *fundamental types*, such as integers, floats, strings, complex numbers, etc., user defined types, and multi-dimensional arrays of any of the above. There are no arrays of arrays, also

known as "ragged arrays." User-defined types can be simple *enumerations* or arbitrarily complex *objects*— a generic term for classes and interfaces.

Objects have methods (a.k.a. member functions) associated with them. For the purposes of the discussion that follows, there are three important details about SIDL methods that need to be explained.

1. Method names (and all SIDL identifiers) must start with an alphabetic character, not a number or underscore (_). Babel will generate "built-in" methods with a leading underscore since these are guaranteed to not collide with user-defined code.
2. SIDL has an unusual feature (for IDLs) to support method overloading. Method overloading is supported natively in C++, Java, and Fortran90 where different methods with the same name are disambiguated by their distinct argument list. To support C, Python, and Fortran 77, SIDL requires the user to provide a disambiguating suffix to the end of the overloaded method name. It appears in SIDL in square brackets ([]) before the argument list.
3. Every argument in a method has at least three parts, a *mode*, a type, and a name. The type and name are familiar to most programming languages, but the mode is peculiar to IDLs. The mode indicates which way data is flowing through the method invocation and must be one of three values: `in`, `out`, or `inout`.

When the reader sees a method described as:

```
_create[Remote](in string url);
```

they can infer that (1) it is a built-in method (by virtue of leading underscore), (2) it appears as "_create" in languages that support overloaded methods and "_createRemote" in languages that do not, and (3) it takes an input argument of type `string` and name `url`. The return type is not specified in the example above but would appear before the method name. We will revisit this particular call later.

3.2. Object Model

The SIDL language supports the Java/Objective C model of *classes* and *interfaces*, where a class has implementations associated with at least some of its member functions and an interface has no implementation. C++ has an equivalent construct called "pure abstract base class.". In this model, interfaces can *extend* multiple interfaces, classes can *implement* multiple interfaces, but a class can extend at most one other class. This model is often summarized as a "single inheritance of implementation, multiple inheritance of interfaces."

Though Babel supports the same object model as Java, there are some critical differences in the implicit base classes and interfaces, besides naming. Like Java, a SIDL *package* is simply a way of organizing and grouping types to reduce the possibility of name collision. All interfaces will implicitly inherit `sidl.BaseInterface`, and all classes will implicitly inherit `sidl.BaseClass`, even if they are not listed as ancestors in the SIDL file.

Unlike Java, Babel's base exception type is an interface instead of a class. This change was driven by SciDAC customers that (quite correctly) wished to develop a specification purely in SIDL interfaces. If we followed Java and made the base exception type a class, then the standards group would be put in the uncomfortable position of providing implementations to the exception classes in their interface specification. Interfaces cannot inherit from classes, only classes can do this.

Clearly, Java RMI supports an object model based on the Java language. It is surprising, however, to see how other IDL-based systems are not very object oriented.

DCOM's IDL has classes and interfaces, but inheritance is limited. An interface can extend at most one other interface, but there is no class inheritance, and classes opaquely contain a list of interfaces but do not inherit them. With DCOM's system, if a class contains three interfaces and a user wishes to access the functionality of one of them, the user must first call `QueryInterface()` to extract the proper interface handle from its parent. The COM specification assumes implementation details about the underlying C++ compiler. Only a subset of expressible types (called *automation types*) in COM IDL are actually guaranteed to work outside of C++ [32, pg. 101].

CORBA's IDL is far less vendor specific but forgoes classes altogether specifying only interfaces and interface inheritance in the IDL. The rationale is that implementation inheritance is an implementation detail and does not belong in the IDL. A kind of private inheritance could be employed for the C++ or Java bindings, but this is a function of the underlying implementation language and not part of the standard. Babel allows a class implemented in one language to have a derived class implemented in another language, overriding some methods and inheriting the implementation of others. This is not possible with CORBA.

3.3. Layers in Babel Callstack

Babel achieves its *n*-way interoperability by having a true hub-and-spoke design, shown in Figure 1. The hub that all invocations are routed through is Babel's *Intermediate Object Representation (IOR)*[1], which serves two main functions. First, the IOR supports a universal object model by implementing tables of virtual function pointers, casting,

[1]Caution: CORBA uses the acronym IOR to refer to an Interoperable Object Reference, essentially a character string encoding information necessary to connect to a service.

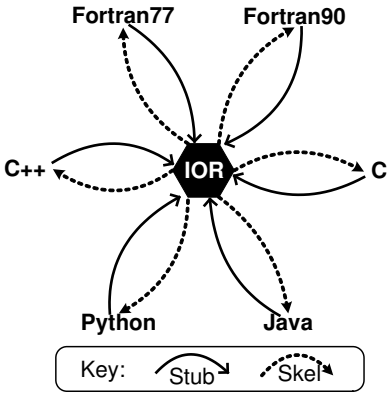

Figure 1. Babel's architecture and separation of concerns for its in-process language interoperability.

polymorphism, etc. Second, the IOR serves as a common target in ANSI C for all language bindings.

Stubs are the layer the callee uses and map from the callee's language to C in the IOR. *Skels* are the layer to which the IOR dispatches, and it maps from C to the language of the implementation[2]. Babel will also generate an empty implementation layer called *Impls*, which is not explicitly labeled in Fig. 1, but is represented by the languages that the Skels invoke. As implied by this figure, Impls in one language can turn around and call stubs of dependent types.

4. Approach

A few words about our technical goals and design constraints are in order before diving into the details. First and foremost, we did not want to compromise the hard-fought performance we achieved in the in-process case. Second, we wanted the support for RMI to be transparent to the user. This means that the same stubs used for language interoperability would also be used as client proxies for RMI. Schematically, the new layers of the Babel RMI are presented in Figure 2. Third, we wanted the messaging layer of Babel RMI to be open to third-party implementations. This last constraint requires special attention because some protocols allow for streaming of data, and others (notably SOAP) are free to reorder how arguments appear in the transmission.

Adding RMI to Babel was a multi-year effort. Constant communication with our customers and collaborators was very important to our process. In addition to presenting work in progress and submitting to *ad hoc* design reviews at

[2]Babel's naming of Stubs and Skels is consistent with CORBA convention, but both are in conflict with COM nomenclature.

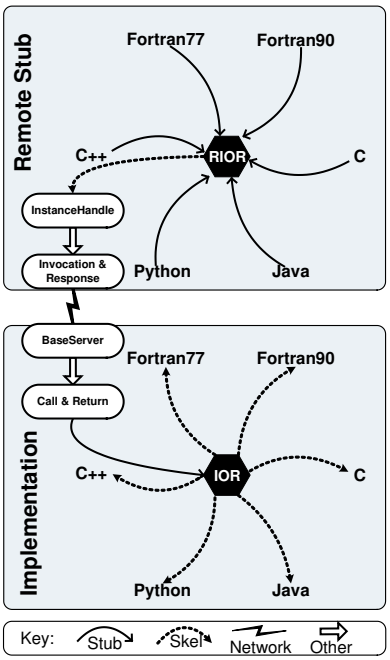

Figure 2. Babel's modified architecture and separation of concerns to support RMI.

quarterly CCA meetings, we published our design in a technical report [25] and updated it periodically on our website to serve as our technical road map. Separately, we have also published a technical report on how to implement a new protocol to our Babel RMI interface [26], but those details are beyond the scope of this paper. Our focus here is on *use* of the Babel RMI feature.

4.1. User Visible Changes

For normal function calls, arguments are passed by initializing registers and pushing things onto the system stack. In RMI, function arguments and return values are passed by a network protocol. From the caller's point of view, the only difference between local and remote invocations is that remote ones have more failure modes. The most visible change to users in adding RMI to Babel was the introduction of an implicit exception that every SIDL method could throw. We added the `sidl.RuntimeException` interface to our standard library to serve as the base object for all such possible exceptions.

A few built-in methods were added to all Babel Stubs in support of RMI.

```
static <T> _connect(in string url);
string _getURL();
bool _isRemote();
bool _isLocal();
```

Here, <T> is pseudo-code to represent that the return type depends on the type of the object from which the static method was invoked. To connect to an instance of pkg_Cls in C, one would call the function pkg_Cls__connect(...) (note the double underscore). In C++, the type would be pkg::Cls and the call would look like pkg::Cls::_connect(...). Whatever the programming language, _connect() is used to create a Stub and Remote IOR (RIOR) connected to a type elsewhere on a server. The argument's name (url) is a bit over specific. The simple binary TCP/IP protocol that comes with Babel does use a URL, but the encoding of the string to refer to a remote Babel object depends on the protocol implementation used at runtime. The method to request the underlying URL may attempt to construct one under certain conditions, but we will avoid the details for the time being. The last two methods simply test if the stub is hooked into a local or remote object. This can be done without actual network traffic by simply testing if a IOR or an RIOR is connected to the Stub. An RIOR is special initialization of an IOR; its internal tables of function pointers are initialized for remote communication, and instead of pointing to an opaque handle of in-process implementation data, it points to the sidl.rmi.InstanceHandle interface.

Concrete classes will also have the built-in method

```
static <T>
_create[Remote]( in string url );
```

that will attempt to connect to a server and request the creation of a new instance of the type. As with _connect, the <T> is pseudo-SIDL indicating that the return type depends on the class to which the static method is bound, and the contents of the url are specific to the protocol being employed.

4.2. Changes to Babel Runtime Library

To add RMI to Babel, we made minimal changes to the sidl package and introduced the sidl.io and sidl.rmi packages. Babel's runtime library includes all of these packages, arrays of basic types, and various odds-and-ends to support a consistent SIDL object model when particular languages need extra help (e.g. casting operators for Fortran, a few wrapper classes for Python, etc.).

The hierarchy of the new sidl.rmi package is illustrated in Fig. 3. All objects defined in this package fall into three main categories: interfaces for protocol implementors to inherit, singleton classes to centralize discovery of the runtime environment, and a litany of exceptions corresponding to the many ways that a remote method invocation could fail. NetworkException and its many subclasses are straightforward with suitably descriptive names, so they require no further exposition in this paper.

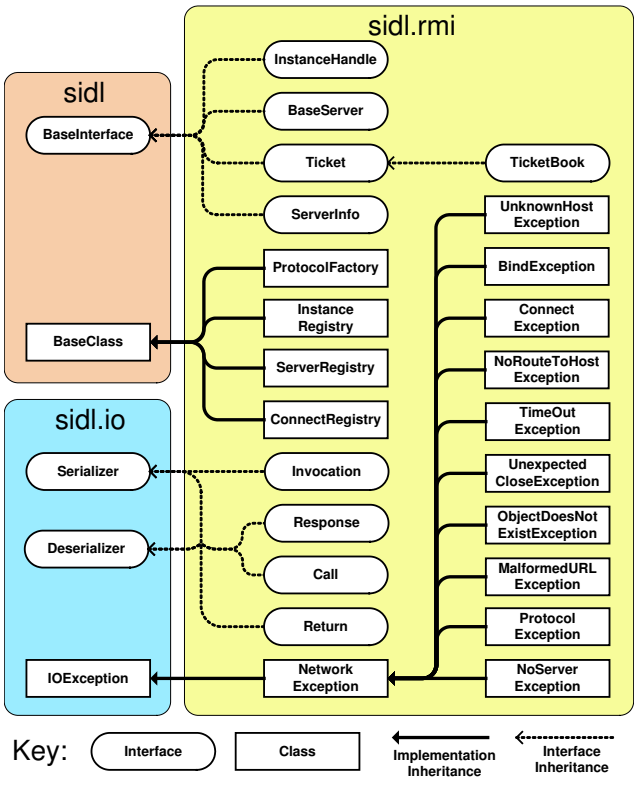

Figure 3. Objects in new sidl.rmi package.

While the interfaces in sidl.rmi are designed for implementors of Babel RMI network protocol "plug-ins," a functional description is warranted in preparation for upcoming topics of object serialization and life cycle. The InstanceHandle persists with the lifetime of the remote stub and is responsible for connecting to its associated implementation of BaseServer. Our documentation will generically use the term *Babel Object Server* (BOS) for any implementation of BaseServer. The Ticket and TicketBook interfaces are for an experimental feature that won't be discussed in this paper, but are included in the figure for completeness. As one might surmise by comparing Figs. 2 & 3, Invocation and Response are responsible for serializing/deserializing arguments on the client side of a remote invocation, Call and Return mainly deserialize the in parameters and serialize the out parameters, respectively.

As we review the singleton classes in the sidl.rmi package, we start describing how the pieces of the Babel RMI system work together. The ProtocolFactory associates the prefix of a URL to an implementation of the InstanceHandle interface. For example, the simple protocol that we distribute with Babel is implemented in a class called sidlx.SimHandle, and we've taken the convention of associating it with the prefix "simhandle." This associa-

tion is made at runtime on both the client and server side as follows:

```
using namespace sidl::rmi;
...
ProtocolFactory::registerProtocol(
      "simhandle", "sidlx.SimHandle");
```

Now to create a remote instance of any type, we simply use `_create[Remote]()` with a string argument that starts with a matching prefix. Creating a remote instance of `sidl.BaseClass` in C++ could look like this.

```
sidl::BaseClass bc =
    sidl::BaseClass::_create(
        "simhandle://server.com:8080");
```

The `_create[Remote]()` method will strip off the prefix (everything up to the first non-alphanumeric character), use the `ProtocolFactory` to instantiate the appropriate implementation, and pass it the remainder of the URL to interpret as it sees fit. Babel has long supported dynamic loading of Babel types by string name through our `sidl.Loader` and `sidl.DLL` classes, so loading an implementation of the `InstanceHandle` interface by string name and passing it to the RIOR is not difficult.

The `InstanceRegistry` is a server-side singleton responsible for generating unique (to the process) identifiers and maintaining a table to retrieve instances by this identifier. When a `BaseServer` receives an RMI from the network, the identifier is included in the message. The `BaseServer` uses this identifier to apply the RMI to the correct instance.

4.3. Distributed Reference Counting

All Babel objects and arrays are reference counted. Reference counting is the only reliable way to deal with resource reclamation in a multilingual environment. In languages such as C++, Python, or Java, the programmer need not worry about reference count at all; it is all tied into the operator overloading, reference counting, or garbage collection of the host language. C and Fortran programmers, however, must explicitly add or delete references if they intend for an object to persist beyond the scope of the single subroutine. This requires some discipline for C and Fortran programmers, but at least the correctness of each file can be determined by visual inspection.

Babel does not reveal the underlying reference count to users under any circumstances. Although this has a side benefit of preventing users from circumventing good programming practice, the truth is that Babel doesn't store reference count in a central location. Instead, there are multiple partial counts. Only when a partial count decrements from one to zero is it necessary to take action. Even in the single process case, partial reference counts may be held in Java stubs, Python stubs, and the `sidl.BaseClass`'s Impl—which every class inherits. In the RMI case, there's also a partial reference count in the `InstanceHandle`. Only when the dependent stubs decrement from one to zero does the last stub notify the `InstanceHandle`, and only when the `InstanceHandle`'s count goes from one to zero does communication actually occur across the wire to decrement the count on the instance itself. When the instance's reference count goes to zero, it removes itself from the `InstanceRegistry` and goes through its destruction sequence.

COM and CORBA have two variations on this theme of managing references in a distributed setting. Remote handles to COM objects are expected to ping their servers every two minutes. After 10 minutes without even a ping, the server is free to declare the handle stale and disconnect it. If there are no outstanding handles, the instance is reclaimed. CORBA takes a more service-oriented approach where reference counting on the client side has no effect on the server. The server is considered to be "always on" and it is up to the user to invent infrastructure for time out or deletion of services.

Babel's approach has the shortcoming that if a client is disconnected, the server may never receive the one-to-zero transition from the `InstanceHandle` to properly decrement its own reference count, and ends up leaking resources. We anticipate that as more robust communication protocols are implemented in Babel RMI, more fault-tolerant schemes can be incorporated.

4.4. Casting

Casting up and down the inheritance hierarchy is a fundamental capability of any object-oriented type system. When performing a successful cast in the in-process case, the new reference points to the same IOR instance as the old reference. All Babel objects must have a concrete class to be `_create()`'ed, so casting is simply a matter of checking if the underlying type inherits the target type or not.

Implementation details to support casting in a distributed context are more complex since the underlying concrete type is not resident in the memory of the client. If the initial remote reference is a `_connect()` to an interface or parent class, a legitimate downcast may demand creation of an entirely new stub. When doing a lot of casting and reference count manipulations (in languages that require manual reference count tracking), care must be taken for each distinct remote handle to delete its reference once. The operation is intentionally *delete reference*, not decrement reference.

4.5. Passing by Reference and by Value

Default semantics for passing arguments locally are that basic types and enums are "pass by value," whereas arrays and objects are "pass by reference." When passing arguments remotely, basic types, enums, *and arrays* are copied across, whereas remote references are passed in lieu of objects. In Babel's type system, arrays are lower-level entities than objects, and it is hard to envision when a user really would want to access a remote array elementwise. If a user really intends to transmit a portion of an array across the wire, we recommend creating the slice locally and then passing the slice through the RMI.

There are two ways one can pass objects in Babel RMI—by reference and by copy. The default method is pass by reference. Any instance passed this way is automatically registered to the BOS to recieve callbacks. Pass-by-copy, or object serialization, is also possible when two conditions are met. First, only objects that implement the `sidl.io.Serializable` interface are eligible to be passed by copy. Second, to trigger the copy, the `copy` keyword must appear in the SIDL file as a modifier to the argument (or return value).

Appropriate error messages are generated if a `copy` keyword appears on a non-serializable type. The entire interface is simply

```
// in package sidl.io ...
interface Serializable {
  void packObj(in Serializer ser);
  void unpackObj(in Deserializer des);
}
```

If an implementor of any Babel class wishes their type to be copyable, they need to inherit this interface and implement these methods. A lot of work has gone into distributed systems providing automatic serialization. Babel takes the other extreme. Its code generators have no idea what implementation data is contained in the classes, so it leaves it up to the implementor of the class to decide what data is packed and unpacked. How that data is encoded on the wire is not the discretion of the implementor of the type but by the implementor of the `Serializer` and `Deserializer` interfaces, which for the purposes of RMI is the implementor of the network protocol. When implementing a serializable class, judicious use of calling the parent class's implementation (using `super()`) is imperative.

An interesting case arises when copying types that are more specialized than the argument list would suggest. A method's argument may indicate a pass-by-value argument that is a SIDL interface. Clearly, the entire class would need to be serialized, but the recipient could be compiled without knowledge of the derived type. Java RMI supports this case trivially because it can simply send the bytecodes of the implementation in the invocation. Babel does the next best thing, sending the name of the concrete type in the hope that the server could get its `sidl.Loader` to create an empty instance. If this succeeds, then the server need only hand the stream to the new instance for it to unpack itself. If the `Loader` cannot find an instance of the type locally, then an exception is thrown back to the caller.

Babel exceptions always obey `copy out` semantics. The recipient of an exception can always rely on the caught instance being local. To support this, we made one major tweak to the `sidl` package by changing the `sidl.BaseException` interface to extend the new `sidl.io.Serializable` interface. Recall that all SIDL methods can potentially throw the `sidl.RuntimeException` interface. The implementations of `sidl.SIDLException` and `sidl.rmi.NetworkException` are good working examples of how users can serialize their own objects.

Table 1. Average round trip latency of 10,000 remote no-ops on 3.06 GHz Intel Pentium Xeons with Elan3 switch using Babel 1.0.1 and Intel 9.1 compilers.

Supporting Middleware	time (μsec)
Babel: in process (C)	0.030
MPI: ping-pong on elan3 (C)	9.43
CORBA RPC: OmniOrb (C++)	251
Babel RMI: Simple TCP/IP (C++)	609
Globus 4.0 Core: no security (Java)	28,000

5. Results

5.1. Middleware Performance Comparison

We conducted two simple experiments to compare the latency and throughput of Babel RMI with other middleware technologies. The experiments were conducted on two nodes of a 3.06 GHz Intel Pentium Xeon Linux cluster with an Elan 3 switch using Babel 1.0.1 and Intel 9.1 compilers. The CORBA implementation we chose is an established open-source implementation called omniOrb [30] version 4.0.7. The Grid implemenation is the Java WS Core distribution of the Globus Toolkit [16] version 4.0.1. Globus is an open-source implementation of much of the OGSA and is maintained by the Globus Alliance.

Latency data presented in Table 1 was generated by exercising a subroutine that has no arguments, no return value, and does nothing. The round trip latency, in this case, is the lower bound for the overhead of the various technologies. Any more useful information exchange between caller to callee and back would only increase the latencies. To con-

struct a reasonably comparable workload for MPI, we chose two point-to-point communications in a ping-pong arrangement. For reference, we also timed Babel on a single node when the invocation remains within a single process.

The in-process Babel call is getting into the tens of clock cycles for our platform, which approaches the cost of a C++ virtual function call. The low overhead for MPI is expected since it bypasses the normal TCP/IP stack and dives directly into libraries optimized for the switch. The 20,000-fold slowdown for Babel RMI over in-process Babel may seem like a concern, but taken in comparison to omniORB's performance, the performance is reasonable. Globus' placement in this study should not be a surprise considering how its underlying communication is XML-based. The Java/WS-based communication suffers 50–100 times more overhead than Babel RMI or omniORB for a no-op.

Throughput data was generated by passing an array of doubles back and forth. The log-log plot in Fig. 4, compares the average elapsed time to the size of the array for various technologies. Since Web Services is based on a text-based protocol, the values in the array matter. There is a surprising performance drop in converting more than 17 digits because IEEE standards specify rounding modes that require extended precision calculations [10]. In all experiments, we initialized the array with a simple sine curve over 2π radians.

For the in-process case, Babel simply passes an array descriptor, so the elapsed time is independent of array size. All other technologies show the expected linear scaling once the array is sufficiently large. To our surprize, omniORB started throwing system errors after the array crossed a certain threshold. We speculate that there is some system setting that preempts messages over a megabyte long but have not investigated it further. At small message sizes, there are dips in MPI and CORBA implementations that we presume are the result of careful optimizations for special cases. The reference implementation that we distribute with Babel has no such performance tuning and shows a simple performance curve. Though our reference implementation starts off more expensive than CORBA, the curves cross at 128 doubles and Babel RMI asymptotes out at roughly 7 times faster than CORBA and 4 times slower than MPI.

Globus scales quadratically in Fig. 4. We could not generate data beyond a 64Ki-long array because the underlying Java sockets were timing out. We investigated this further with appropriate technical leads from the Global Alliance, but they did not contest our finding.

Naturally, there are questions about how this initial data plays out as increasingly complex arguments are passed back and forth between caller and callee. Such a detailed study is beyond the scope of this paper. The protocol that we are currently shipping and reporting data from is in-

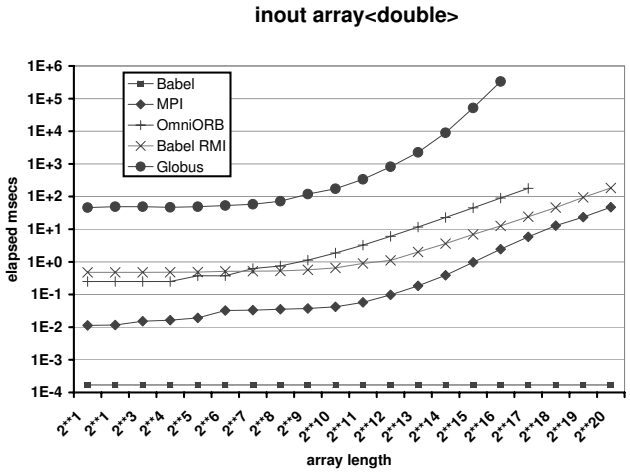

Figure 4. Average elapsed time for 10,000 remote method invocations sending and receiving an array of doubles on 3.06 GHz Intel Pentium Xeons with Elan3 switch using Babel 1.0.1 and Intel 9.1 compilers.

tended as a reference implementation and not production code. More sophisticated implementations based in RMIX, SOAP, IIOP, and RMA are in progress from CCA collaborators. How well Babel RMI could perform using the same switch-specific libraries as MPI—and if we could use those libraries at all without interfering with MPI—are both open questions.

5.2. Multiscale Material Science

The Petascale Simulation Initiative (PSI) [31] is an internally funded exploratory R&D project involving multiple directorates at Lawrence Livermore National Laboratory (LLNL). At the computer science level, the goal is to develop and demonstrate a programming model where disparate parallel codes are composed and managed in an MPMD style. Though originally conceived as a hedge against possible SPMD scaling issues, when we get to the hundreds of thousands of processors expected in petascale systems, the emerging MPMD system is garnering more immediate appeal as a means to achieve better utility of existing platforms for highly dynamic multiscale calculations.

PSI members from LLNL's Engineering directorate are employing this technology to couple their primary engineering code at macro scale with various subscale material models [3, 4]. The need for subscale calculations is highly dynamic, problem specific, and cannot be predicted *a priori*. Under the right conditions, advancing a single finite element one timestep could require launching a slip-surface or multigrain crystal-plasticity calculation across tens to hundreds of processors. The macroscale engineering code is

written primarily in C, the subscale models tend to be in Fortran 90, and the PSI runtime system is mostly C++. In addition, it has been scripted with Python for parametric studies.

The PSI runtime system that coordinates these multiple MPI jobs is a collection of Babel-wrapped, MPI-enabled objects that bootstrap themselves as a parallel daemon process—one instance per node. The rank-zero process removes itself from the node pool and assumes its role as process scheduler and node manager, handing out sub allocations of its current nodes for subscale models. Using SLURM-specific hooks [38], as well as POSIX `fork/join`, the runtime system can launch new MPI jobs on request, set up Babel remote stubs to establish parent-child relationships between launcher and launchee, and deal with child termination, either normal or abnormal. Preliminary timings on the same system as Table 1 show setup and teardown of these spawned MPI subtasks at one half and one fifth of a second, respectively. The PSI has exercised Babel RMI on science runs requiring billions of RMI calls for 340-node runs lasting up to 8.5 hours.

6. Conclusions and Future Work

We introduced Babel's new Remote Method Invocation (RMI) capabilities. The distinguishing characteristics of this work are that it provides a simpler and more consistent object-oriented programming model than CORBA or COM, as well as a simple API for third-party plug-ins to customize the communication layer underneath. Even with the simple TCP/IP protocol that is shipped as a reference implementation, Babel RMI is much faster than Web Services and even outperforms CORBA.

Given these characteristics, Babel RMI currently fills a niche in "short-haul" distributed computing—within a machine room or even in a single machine with concurrent MPI runs. Babel also provides a useful tool for our CCA collaborators to build and extend their CCA component frameworks to new levels of capability. Distributed computing across wide-area networks is beyond the scope of our work and already well served by Grid/Web Services technology.

Future development for Babel is motivated by ongoing collaborations in the CCA Forum and with our in-house material scientists. We are particularly interested in developing support for Parallel RMI [7]. Nonblocking RMI is also of interest, both for cases where machines may not support threaded implementations (e.g. "micro kernel"-based supercomputers) and for cases where co-processors (e.g. GPUs or FPGAs) naturally interact in an asynchronous manner.

Acknowledgments

This work was funded by the U.S. Department of Energy Office of Science SciDAC program as part of the Center for Component Technology for Terascale Simulation Software (CCTTSS). Additional support for RMI customizations and ongoing nonblocking development is provided by the LLNL LDRD program as part of the PSI.

We wish to thank David Bernholdt and Mahdu Govindaraju for their thoughtful feedback on early drafts of this paper, Steve Parker for the impromptu design reviews at CCA meetings, Wei Lu for his help in generating the Globus data in Fig. 4, and our collaborators in PSI and CCA who provided feedback on our evolving design documents. We also acknowledge our 2004 intern from UC Davis, Nija Shi, who implemented an early BOS prototype in pure Python.

References

[1] Y. Alexeev and et. al. Component-based software for high-performance scientific computing. In A. M. et. al., editor, *Proceedings of Scientific Discovery through Advanced Computing (SciDAC 2005)*, volume 16 of *J. of Phys. Conf. Ser.*, pages 536–540, 2005.

[2] R. Armstrong, G. Kumfert, L. C. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, and T. Dahlgren. The CCA component model for high-performance computing. *Intl. J. of Concurrency and Comput.: Practice and Experience*, 18(2), 2006.

[3] A. Arsenlis, N. R. Barton, R. Becker, and R. E. Rudd. Generalized *in situ* adaptive tabulation for constitutive model evaluation in plasticity. *Computer Methods in Applied Mechanics and Engineering*, 196:1–13, 2006.

[4] N. R. Barton, J. Knap, A. Arsenlis, R. Becker, R. D. Hornung, and D. R. Jefferson. Embedded polycrystal plasticity and *in situ* adaptive tabulation. Submitted to Int. J. Plasticity, 2006.

[5] D. E. Bernholdt, B. A. Allan, R. C. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. K. Krishnan, G. K. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou. A component architecture for high-performance scientific computing. *Intl. J. High-Perf. Computing Appl.*, 20:163–202, May 2005.

[6] D. E. Bernholdt, W. R. Elwasif, J. A. Kohl, and T. G. W. Epperly. A component architecture for high-performance computing. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages (POHLL-02)*, New York, NY, June 2002.

[7] F. Bertrand, R. Bramley, K. B. Damevski, J. A. Kohl, D. E. Bernholdt, J. W. Larson, and A. Sussman. Data redistribution and remote method invocation in parallel component architectures. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium: IPDPS 2005*, 2005.

[8] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Computer Systems*, 2(1):39–59, Jan 1984.

[9] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, pages 246–254, Piscataway, NJ, USA, 2002. IEEE Comput. Soc.

[10] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC'02)*, July 2002.

[11] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. Technical report, World Wide Web Consortium, March 2001. `http://www.w3.org/TR/wsdl`.

[12] L. Clement, A. Hately, C. von Riegen, and T. Rogers. UDDI version 3. Technical report, Oasis, October 2004.

[13] Component technologies project website. `http://www.llnl.gov/CASC/components`.

[14] T. Dahlgren, T. Epperly, G. Kumfert, and J. Leek. *Babel User's Guide*. Lawrence Livermore National Laboratory, July 2006. version 1.0.0.

[15] M. Diaz, D. Garrido, S. Romero, Bartolomeé, E. Soler, and J. M. Troya. A CCA-compliant nuclear power plant simulator kernel. In G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. Szyperski, and K. Wallnau, editors, *8th International Symposium on Component Based Software Engineering (CBSE 2005)*, volume 3489 of *LNCS*. Springer, 2005.

[16] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Converence on Network and Parallel Computing*, volume 3779 of *LNCS*, pages 2–13. Springer-Verlag, 2005.

[17] I. Foster, H. Kishimoto, A. Savaa, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Tredwell, and J. V. Reich. The open grid services architecture: Version 1.0. Technical Report GFD-I.030, Global Grid Forum, January 2005.

[18] D. Gannon and et al. Programming the grid: Distributed software components, P2P and grid web services for scientific applications. *J. of Cluster Comput.*, 5(2):325–336, 2002. Special Issue on Grid Computing.

[19] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. Simple object access protocol (SOAP) 1.2. Technical report, World Wide Web Consortium, June 2003.

[20] J. P. Kenny, S. J. Benson, Y. Alexeev, J. Sarich, C. L. Janssen, L. C. Mcinnes, M. Krishnan, J. Nieplocha, E. Jurrus, C. Fahlstrom, and T. L. Windus. Component-based integration of chemistry and optimization software. *J. of Computational Chemistry*, 25(14):1717–1725, 2004.

[21] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. Divorcing language dependencies from a scientific software library. In *Proc. 10th SIAM Conf. Parallel Process.*, Portsmouth, VA, March 2001.

[22] M. Krishnan, Y. Alexeev, T. L. Windus, and J. Nieplocha. Multilevel parallelism in computational chemistry using common component architecture and global arrays. In *ACM/IEEE Supercomputing 2005 Conference (SC'05)*, November 2005.

[23] G. Kumfert, D. E. Bernholdt, T. G. W. Epperly, J. A. Kohl, L. C. McInnes, S. Parker, and J. Ray. How the common component architecture advances computational science. In W. M. T. et. al., editor, *SciDAC 2006, Scientific Discovery through Advanced Computing*, volume 46 of *J. of Phys.: Conf. Ser.*, pages 479–493, Denver, Colorado, USA, June 2006.

[24] G. Kumfert and T. Epperly. Software in the DOE: The hidden overhead of "The Build". Technical Report UCRL-ID-147343, Lawrence Livermore National Laboratory, March 2002.

[25] G. Kumfert and J. Leek. Proposed Babel/SIDL changes to support RMI. Technical Report UCRL-TR-213497, Lawrence Livermore National Laboratory, October 2004. Revised 12/04 and 5/05.

[26] G. Kumfert and J. Leek. Writing a protocol to support RMI. Technical Report UCRL-TR-220292, Lawrence Livermore National Laboratory, February 2006.

[27] J. W. Larson and et al. Components, the common component architecture, and the climate/weather/ocean community. In *84th American Meteorological Society Annual Meeting*, Seattle, Washington, 11–15 January 2004. American Meteorological Society.

[28] L. C. McInnes and et al. Parallel PDE-based simulations using the common component architecture. In A. M. Bruaset, P. Bjørstad, and A. Tveito, editors, *Numerical Solution of PDEs on Parallel Computers*. Springer-Verlag, 2005. invited chapter, acceptedand also available as Argonne National Laboratory technical report ANL/MCS-P1179-0704.

[29] Object Management Group. *The Common Object Request Broker Architecture (CORBA), v3.0*, July 2002. Adopted Specification.

[30] omniORB website. `http://omniorb.sourceforge.net`.

[31] Petascale simulation initiative website. `http://www.llnl.gov/CASC/psi`.

[32] J. Pritchard. *COM and CORBA side by side: Architectures, Strategies, and Implementations*. Addison Wesley, 1999.

[33] R. M. Soley and C. M. Stone. *Object Management Architecture Guide*. J. Wiley, 1995.

[34] T. J. Tautges, P. Knupp, J. A. Kraftcheck, and H. J. Kim. Interoperable geometry and mesh components for scidac applications. In A. M. et. al., editor, *Proceedings of Scientific Discovery through Advanced Computing (SciDAC 2005)*, volume 16 of *J. of Phys. Conf. Ser.*, pages 486–490, 2005.

[35] J. Waldo. Remote procedure calls and java remote method invocation. *IEEE Concurrency*, 6(3):5–7, July–September 1998.

[36] D. Winer. XML-RPC specification. Technical report, XML-RPC.com, June 1999.

[37] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *Second USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 219–232, Toronto, Ontario, Canada, June 1996.

[38] A. B. Yoo, M. A. Jette, and M. Grondona. *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *LNCS*, chapter SLURM: Simple Linux Utility for Resource Management, pages 44–60. Springer-Verlag, 2003.