

The Proteus Multiprotocol Message Library

Kenneth Chiu, Madhusudhan Govindaraju, Dennis Gannon

Computer Science Department
Indiana University
Bloomington, IN 47405-7104
{chiuk,mgovinda,gannon}@cs.indiana.edu

Abstract

Grid systems span manifold organizations and application domains. Because this diverse environment inevitably engenders multiple protocols, interoperability mechanisms are crucial to seamless, pervasive access. This paper presents the design, rationale, and implementation of the Proteus multiprotocol library for integrating multiple message protocols, such as SOAP and JMS, within one system. Proteus decouples application code from protocol code at run-time, allowing clients to incorporate separately developed protocols without recompiling or halting. Through generic serialization, which separates the transfer syntax from the message type, protocols can also be added independently of serialization routines. We also show performance-enhancing mechanisms for Grid services that examine metadata, but pass actual data through opaquely (such as adapters). The interface provided to protocol implementors is general enough to support protocols as disparate as our current implementations: SOAP, JMS, and binary. Proteus is written in C++; a Java port is planned.

Keywords

multiprotocol, Grid, middleware, SOAP, component

1 Introduction

With rapid hardware advances and the Internet boom, the small, isolated distributed systems of the past are transforming into massive, dynamic, Grid[9] systems spanning different projects, organizations, and application domains. No single body can, or should, oversee the research taking us through this transition, and thus each community imbues its efforts with its own requirements, understanding, and heritage. The result is an eclectic, perhaps even chaotic, mix of standards, protocols, and APIs.

This disparity hinders the realization of pervasive, seamless access, but imposing uniformity on this diversity would stifle experimentation and innovation—ultimately leading to inferior results. Thus, we suggest planning for, and even encouraging, heterogeneity by adopting software infrastructures that dynamically integrate multiple protocols within one client¹. Such infrastructure can be utilized by: (1) end-user applications to select the best protocol at run-time, and (2) adapters or other services to bridge incompatible domains.

¹We use the term *client* here to mean any program which sends or receives messages, as opposed to a only programs that send messages.

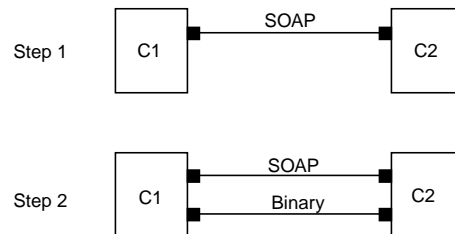


Figure 1. In Step 1, client C1 initiation communication to client C2 with SOAP. After discovering that C2 understands our binary protocol, C1 switches to it in Step 2, but can still fall back to SOAP if necessary.

Services from the latter are examples of Grid *intermediaries*. A Grid intermediary is a service which handles data, but is neither the original producer nor the final consumer of the data. It typically examines the metadata, but not the actual data. Examples of Grid intermediaries include:

- A data cache which tracks data's provenance, but simply forwards the actual data.
- A gateway between two incompatible systems.
- A high-level, message router such as might be used for the Web Services Routing Protocol[14] (WS-Routing).

Proteus is a library for decoupling the client from the messaging protocols. Clients developed using Proteus can communicate using a variety of means, such as SOAP[2], Java Message Service[17] (JMS), or a locally developed, experimental protocol. Other protocols can be added dynamically without recompiling or halting.

As suggested above, Proteus is motivated by two primary use cases. The first follows from our work with Web services in general and the Open Grid Services Architecture[7] (OGSA) in particular. Web services has emerged as a promising model for Grid middleware design, including such projects as the Grid Portal Testbed and several of the Grid monitoring projects being constructed by Global Grid Forum working groups. In this approach, each Grid service is described in the Web Services Description Language[4] (WSDL) and accessed through one of possibly several message protocols listed in its WSDL description. With its emphasis on interoperability, the SOAP protocol is the *de facto lingua franca* for such work, but it can be inefficient for the transmission of the large data sets typical in scientific comput-

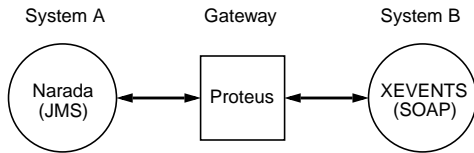


Figure 2. Proteus can also be used to construct a gateway between two otherwise incompatible systems. In this diagram, the gateway uses Proteus to understand both JMS and SOAP. An event sent to a Narada subscriber from XEVENTS first passes through the gateway, and vice versa.

ing. With Proteus, however, scientific applications can use faster protocols when available, yet fall back to SOAP when required (Figure 1). This furthers interoperability by giving developers the freedom to adopt SOAP without fearing efficiency issues.

The second use case is as an efficient platform for Grid intermediaries. For example, Narada[10] events are JMS-based, while our internal research events (XEVENTS) are SOAP-based. Thus a system A built using Narada cannot directly interoperate with a system B built using XEVENTS. One solution would be to add Narada support to system B and vice versa, but because Narada and XEVENTS use different APIs, this would require significant effort. The other choice is to create a gateway between the two systems, which avoids modifying either system (Figure 2). Such a gateway could be created as a one-time, *ad hoc* effort, but will benefit from using a flexible, extensible library such as Proteus.

In Section 2 we detail our requirements for a multiprotocol library. We then describe Proteus in depth in Section 3. Section 4 then presents some performance results to verify that our design allows efficient implementations. In Section 5 we review related multiprotocol efforts. We conclude in Section 6.

2 Requirements

In this section we detail the requirements and goals, in varying degrees of specificity, that we believe a multiprotocol library for Grid systems must satisfy.

Mechanism, not policy. As middleware for experimentation, a multiprotocol library should avoid over-specification. For example, Proteus provides mechanisms for protocol selection (Section 3.6), negotiation (Section 3.6), and message routing (Section 3.1), but refrains from specifying exactly how these mechanisms should be used.

High performance. As middleware for scientific computing, a multiprotocol library must be capable of handling large messages efficiently. Performance limitations of the underlying middleware should not consign experimental systems to failure. In Proteus, we address this primarily through the matter concept (Section 3.2).

Allow flexible invocation modes. By *invocation mode*, we refer to the programming model used for invoking remote operations. Though request/response is predominant, no one invocation mode is universally suitable. Since most modes can be built from a sequence of asynchronous one-way messages, we chose this as the primary mode presented to clients. A lower-level model, such as an unstructured sequence of primitive values (like strings, integers, and floats), does not provide the level of abstraction necessary to naturally map between the different standards, thus impeding interoperability.

Allow interoperability with non-multiprotocol clients. For example, a multiprotocol client should be able to interact with a generic SOAP server. This requirement is crucial to implementing Grid intermediaries, because they will often be used to provide bridges to preexisting, non-multiprotocol servers. Proteus addresses detailed control of protocol behavior through protocol-specific message maps (Section 3.5).

Allow generic intermediaries. Using typical middleware systems, a server can only accept requests for which it has compile-time type information. This complicates implementing intermediaries such as adapters and proxies. We address this in Proteus through the use of matter (Section 3.2), and the separation of a message into parts (Section 3.1).

Allow adding protocols without recompiling. Requiring all components to be recompiled for every significant protocol change is burdensome. Through careful abstraction, C++ static objects, and dynamic loading, Proteus allows a client to make use of new protocols without recompiling or halting (Section 3.3).

3 Proteus

Proteus is best described as a mediator between clients, which send and receive messages, and *providers*, which implement the actual mechanics of message transmission. Typically, each provider implements a different standard, such as the CORBA Internet Inter-ORB Protocol[15] (IIOP) or JMS, but maps the Proteus message model (Section 3.1) to its native model. Proteus abstracts both the protocol and any API associated with the protocol, so works equally well with standards that are protocols, like IIOP, and ones that are technically only APIs, like JMS.

Because Proteus mediates between clients *and* providers, a distinct interface must be supplied to each group. Utilizing the abstractions and encapsulation of the *client interface*, applications can avoid provider dependencies. New providers (and their associated protocols) can thus be added without changing client code. The *provider interface*, on the other hand, allows new providers to be added without changing Proteus code. This also facilitates dynamically loading providers at run-time. A Proteus provider is typically constructed by wrapping existing protocol implementations.

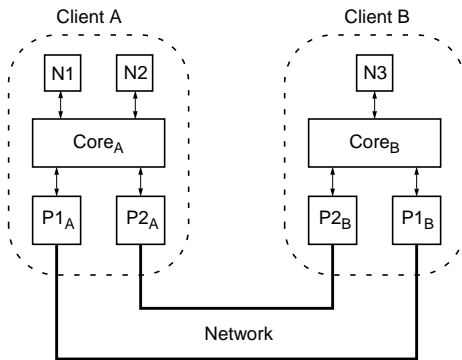


Figure 3. Client A contains two nodes, N1 and N2. Client B contains one node N3. Both clients include providers P1 and P2, and so can use either. Typically clients are processes, as indicated by the dotted containers. The lines between the clients relate provider instantiations that can communicate with each other. The lines may, but do not necessarily, correspond to an actual network connection.

Messages in Proteus are sent between *nodes*, which serve as endpoints and provide a binding context for reception and transmission parameters. Each client can contain many nodes. This is further detailed in Section 3.7.

The Proteus *core* object, instantiated once per process, is responsible for managing the interactions between the clients, nodes, and providers. A diagram of a typical Proteus system is shown in Figure 3.

We now proceed to describe Proteus in greater detail. Note that some Proteus terms have corresponding C++ classes. We use `typewriter` font when referring to the actual class, and normal font when referring to the concept. For clarity and brevity, source samples may omit syntactic details not crucial to understanding, such as variable declarations, etc.

3.1 Message Model

A *message type* is an abstract, hierarchical structure for data. The primary building block is a *vobject*¹ type, which is an ordered set of typed, named fields. Each field can be of primitive type, such as an integer, or another vobject type. Message types are named by strings. Proteus does not attach any semantics to the format of the string, but applications will likely impose some hierarchical namespace syntax, such as that used by XML or C++.

Analogously to how objects are instances of object types (classes), messages are instances of message types². At the provider end, a message is eventually converted to the provider-specific transfer syntax of the implemented protocol.

¹The term vobject derives from value object, and was chosen to avoid confusion with C++ objects or distributed objects (such as CORBA objects). Vobjects are similar to valuetypes in CORBA and serializable objects in Java RMI.

²We will often leave off the word *type* when referring to message types and vobject types. Whether we are then referring to the type or an instance of the type should be clear from context.

3.1.1 Vobject objects

The Proteus model for manipulating the contents of messages is data binding. In this model, the message is mapped to a programming language object, in our case C++, with fields of the message represented by member variables of the object.

Assuming that we have a variable `vobj` of class `MyVobject` defined as:

```
class RecordVobject {
    int i;
};
class MyVobject {
public:
    RecordVobject record;
};
```

One would then access the `i` field of the message with:

```
int i = vobj.record.i;
```

This is convenient, natural, and fast (since access is via an offset, calculated at compile-time, from a base address), but note that every client that wishes to receive a message containing a `MyVobject` must be compiled with the `MyVobject` class definition. This is onerous if the client is an intermediary that has no need to actually access the fields of `MyVobject`. This problem is resolved by the use of *matter* (Section 3.2).

Vobjects are created by *vobject factories*, which given a vobject type will call `new` on the appropriate C++ class. They are registered in a manner similar to providers (Section 3.3).

3.1.2 Message Parts

A Proteus message consists of a set of named parts. Each part contains either a vobject, or *matter* (Section 3.2). The primary purpose of this distinction is to allow a message's information content to be partitioned according to likely roles. Thus, an intermediary that only needs routing information would examine only the routing part. This is examined in more detail in Section 3.2.

3.2 Matter

Deserializing a message into vobjects is not always desirable. First, it requires the corresponding vobject factory. Requiring intermediaries to have a factory for every possible message type is impractical. Second, deserializing content that will never be examined, such as an intermediary processing an irrelevant (to the intermediary) message body, is inefficient. Third, always deserializing into vobjects would preclude intermediaries from handling message parts in unrecognized formats.

Thus, Proteus allows a message part to be deserialized as *matter*. *Matter* is a vobject in raw, protocol-specific form. When deserialized as *matter*, a part's contents are not explicitly parsed. This especially speeds scientific data in SOAP encoding[3] as shown by the test in Figure 8.

Matter must retain enough information so that the end user can deserialize it into a vobject. Thus, we attach a *provider*

descriptor and an *encoding* to all matter. The provider descriptor includes the provider name and other details required to deserialize the matter at its final destination. The encoding, on the other hand, is used by intermediaries to determine how to forward the matter.

For example, an XML-based intermediary must know whether the matter is binary, or text-based. If binary, it must transform it to base64 for transmission. If text-based, it may need to scan for the '<' character. Or, it may already have been converted to XML character data¹ by a previous intermediary, in which case it can be forwarded as is. Such information is specified in the encoding.

We currently define two encodings of matter, XML character data and binary. XML character data is guaranteed to be safe for direct inclusion within an XML element. Binary data is an arbitrary byte stream. The encoding is described as a string, so an actual system built with Proteus is free to impose additional semantics, such as a Uniform Resource Name[13].

Just knowing the encoding is not enough for processing at the final destination however. For example, a number of different protocols, such as Nexus[8] and IIOP, use binary encodings. Thus, we identify this higher-level information with a provider descriptor.

3.2.1 Wormhole Matter

A typical Grid intermediary looks at the message header, modifies it, and then immediately retransmits it along with the unmodified body. This implies that the body part need not be in memory all at the same time, and that the retransmission of this part can be initiated before its end has arrived at the host.

Through the message map, a Proteus client can designate that such a part be deserialized into wormhole matter. Wormhole matter cannot be accessed directly, only retransmitted. Furthermore, it can be retransmitted at most once. A provider can exploit then exploit these qualities by reading it in small chunks and then retransmitting each chunk before reading the next. This can greatly reduce memory usage for large messages, and will enhance overlap of reception with transmission.

3.3 Providers

Providers are code modules (plug-ins)² that implement the actual mechanics of message transmission and reception. Providers are responsible for traversing the message structure and serializing the contents into a provider-specific transfer syntax. Possible providers might be a CORBA Object Request Broker (ORB), a JMS implementation, or a SOAP implementation.

¹XML treats '<' as a special character. Arbitrary text can thus not be embedded in XML until it has been processed into *character data*, which escapes special characters.

²Providers are deployed as shared libraries, which are .SO files on most UNIX systems. Proteus currently runs on Linux, Solaris, and IRIX.

A provider announces its presence by registering itself with the core. This is accomplished by defining a `ProviderRegisterer`:

```
class ThisProvider
    : public proteus::Provider {
    ...
} thisProvider;
// Unnamed namespace
namespace {
    ProviderRegisterer
        thisRegisterer(&thisProvider);
}
```

Since `thisRegisterer` is a static object, its constructor will be invoked automatically when the containing shared library is loaded, which occurs whenever the client invokes the core function `Core::loadModule()`. The `ProviderRegisterer` constructor registers the passed provider with the core.

When a client creates a node, the core normally requests each provider to supply an address that can be used to send a message to this node. Each of these is termed a *uniaddress*, since it is associated with only a single provider. The core composes these uniaddresses into a single Proteus *multiaddress*.

Each uniaddress is a provider-specific polymorphic object, whose contents are determined entirely by the provider. When a selector (Section 3.6) decides upon a specific uniaddress for transmission, it calls `send()` on the uniaddress, which invokes provider-specific code.

For servers that must reside at stable, well-known addresses, a node can also use a given address by specifying it in the constructor as an XML document. For example, this multiaddress contains two uniaddresses:

```
<address>
  <uniaddress provider="XSOAP">
    http://foo.com/service
  </uniaddress>
  <uniaddress provider="JMS">
    <host>a.b.edu</host>
    <queue>fooqueue</queue>
  </uniaddress>
</address>
```

Nothing prevents one provider from supplying two of the uniaddresses within a single multiaddress, which can be used by selectors to implement generalized load balancing over a set of network channels, some of which may use the same protocol but different IP addresses, for example. The result is parallel network I/O, but it is obtained by elegantly combining Proteus constructs instead of explicit programming. We show a simple example of this in Section 4.1.

Proteus does not dictate any discovery or negotiation protocol. One plausible scenario would be to store a multiaddress containing only one uniaddress in a registry. An initial query using that multiaddress can be sent to the remote client, which would then respond with a multiaddress containing all supported providers.

We have currently implemented three providers: SOAP, JMS, and a binary protocol.

3.4 Generic Serialization and Deserialization

Transmitting a vobject first requires conversion from its in-memory representation to a provider-specific format, which is also referred to as a *transfer syntax*. The transfer syntax needs to preserve both the hierarchical structure of nested vobjects and the values of primitive fields. Typically, each serializable object type has a corresponding function that encodes the object into its wire representation.

```
serializeMyObject() {
    SOAP_pack_int(i);
    SOAP_pack_float(f);
}
```

Borrowing from the terminology of aspect-oriented programming, we say that this code has two aspects, the type of the vobject, as embodied in the fields and nesting structure, and the transfer syntax, which governs how the vobject is converted. Thus, the code is both protocol-specific and type-specific.

In uniprotocol systems, this entanglement is generally not an issue. Only one instance of such code is necessary per object. For multiprotocol systems, however, we now require one instance per object per protocol. The result is a multiplicative increase in code, and thus programmer burden. Furthermore, this entanglement greatly impedes deployment and packaging; adding a provider requires modification of each application.

Straightforward application of virtual functions are not enough to solve this problem, but by using the double-dispatch pattern[11], Proteus provides *generic serialization* and deserialization.

We first define an abstract `Serializer` class. Every vobject is then required to implement the `serialize()` member function taking a `Serializer` parameter. The `serialize()` method invokes functions on the polymorphic `Serializer` object, which is implemented by the provider. The `Serializer` object can maintain any required provider state, since its contents are completely determined by the provider. We thus achieve a separation of concerns, and avoid the multiplicative increase in code size of a naive multiprotocol design. This is illustrated in Figure 4 for the case of a SOAP provider.

We now illustrate this with a short example. Suppose we have a message part that contains a `MyVobject` as defined by

```
struct NestedVobject;

struct MyVobject {
    NestedVobject nested;
    int i;
};
```

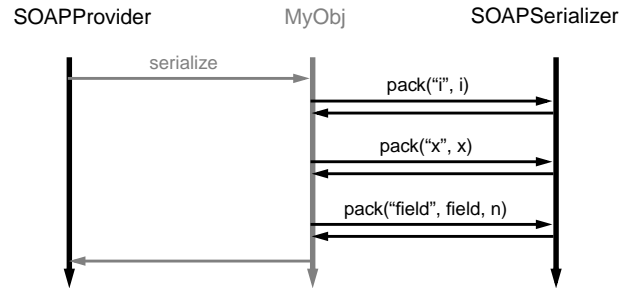


Figure 4. Each vertical line represents a single object. A horizontal line to the right represents a method invocation, and a line to the left represents the method return. The `SOAPProvider` and `SOAPSerializer` are both provider-specific and in the SOAP provider, while the `MyObj` is type-specific.

`MyVobject` contains a nested vobject named `nested` of type `NestedVobject` and an integer field named `i`. The serialization function for such a vobject is given by

```
MyVobject::serialize(const char *name,
                    Serializer *ser)
const {
    ser->beginVobject(name, "MyVobject");
    nested.serialize("nested", ser);
    ser->pack<int>("an_int", 12345);
    ser->endVobject();
}
```

A similar mechanism provides generic deserialization.

3.5 Message Maps

To control particulars of how messages are transmitted and received, Proteus provides *message maps*. With message maps, for example, a client can say that the part named “body” is to be deserialized as matter part, while the part named “header” should be deserialized into a vobject part.

Since a given message may be forwarded to multiple destinations with different mappings for each, maps are bound to nodes rather than messages. Also, a separate map is used for reception and transmission, thus allowing a gateway to receive a part in one form and retransmit in another.

Maps are organized hierarchically, paralleling the message structure. A *part map* controls a single, named part. A set of part maps, along with a message type, constitutes a message map. Finally, a set of message maps compose a *repertoire map*, which, when bound to a node, specify the handling details of all message types recognized by that node.

The Proteus abstractions are not always enough to completely determine how a Proteus message should be represented in the underlying, provider-specific message model. For example, a SOAP server may or may not require a specific `SOAPAction` to be set in the HTTP header. This can be controlled with provider-specific message maps, which have interfaces determined by the provider.

3.6 Selectors

When a message is sent through a node, one of the available providers must be selected. This selection is accomplished with a selector bound to the node at construction. The selector can inspect each of the uniaddresses in turn, deciding which one to use based on collected statistics such as current throughput. A sample selector `send()` function is given below:

```
MySelector::send(const Scheme &scheme,
                const Nodepnt &ret_addr
                const Message &msg) {

    max_rank = -1;
    for (i = 0; i < scheme.size(); i++) {
        rank = rank_map(scheme[i].route());
        if (rank > max_rank) {
            max_rank = rank;
            best = i;
        }
    }

    start_timer();
    scheme[best].send(m, np);
    stop_timer();
    update_rank_map_stats();
}
```

The Scheme object (Section 3.7) can be indexed to access each uniaddress composing the multiaddress. Because a Selector object may be used for many different nodes, it may often encounter two similar uniaddresses with identical transmission characteristics (e.g., they belong to the same subnet), but belonging to different nodes. Such uniaddresses would then return the same Route object from the `Uniaddress::route()` function, thus allowing the selector to combine the resulting data to produce better statistics. The example above uses the `rank_map` function to maintain a priority queue of routes ordered by increasing throughput. The function maps from Route objects to the rank of the route.

3.7 Nodes

The node serves as the binding locus for the several pieces of information required for transmission and reception. All messages are sent and received through nodes. Note that one client may contain many nodes.

Though each node capable of reception has a multiaddress, the multiaddress does not function as identity. Instead, nodes are identified by a node ID, which is a 128-bit universally unique ID, similar to that used in DCE and DCOM. This separation between address and identity facilitates the ability of mobile agents to change their address yet retain their identity.

The fragment below shows how a node with a given address is created. The variable `addr_str` is obtained

through extra-Proteus means, such as a server configuration file, and is in XML.

```
MyMessageHandler handler;
MySelector selector;

Address addr(addr_str);
Nodepoint nodepoint(addr);
RepertoireMap repertoire_map;
Scheme scheme(nodepoint, repertoire_map);

Node node(&selector, scheme, &handler);
```

The address and node ID are bound together by creating a Nodepoint object. The code then creates a repertoire map to specify deserialization details. Here we just use the default, which deserializes all parts into vobjects. The nodepoint and repertoire map are combined into a Scheme object, which is passed to the node constructor, along with the message handler.

The message handler is an object conforming to the Proteus MessageHandler interface, and is called whenever a message is received. Messages can also be received by calling `receive()` on the node.

Messages are also sent through a node:

```
RepertoireMap repertoire_map;
Scheme scheme(dest_str, repertoire_map);

MyVobject vobj;
VobjectPart part("a part", vobj);
Message msg(part);

node.send(scheme, msg);
```

A client will likely use one node to send messages to disparate destinations, each with a distinct characteristics. Thus the scheme is specified individually with each transmission.

4 Performance

The goal of this work is not to develop a high-performance implementation, but rather a design that *facilitates* high-performance, multiprotocol communication. Thus, these tests serve primarily to validate the design, rather than being ends in themselves.

4.1 Adaptive Selector

The first test illustrates that the Proteus selector mechanism is sufficient to allow the implementation of adaptive, parallel network I/O (Figure 5). Messages were sent from node A to node B, which used a multiaddress containing two uniaddresses.

Node A used a simple adaptive algorithm. The frequency with which a given uniaddress is selected was proportional to the square of the throughput, with the additional rule that each uniaddress is guaranteed to be selected at least 1% of the time on average. This last rule allows the algorithm to discover if a

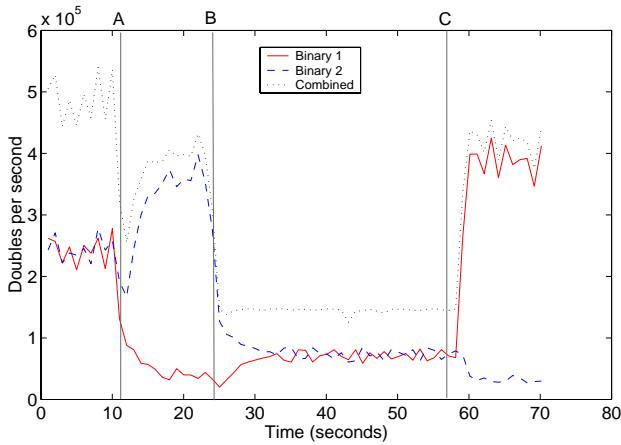


Figure 5. This graph shows the results of a simple adaptive selection algorithm for parallel network I/O. Messages containing an array of 10,000 doubles were sent from a node on a Sun Ultra E450 (4x440 MHz) to a Sun Ultra 80 (2x500 MHz) connected via Gigabit Ethernet. The destination node used a multiaddress containing two binary uniaddresses. Initially, both uniaddresses had similar performance, so were selected about equally. At time A, a transmission delay of 5 milliseconds was simulated in uniaddress 1, so the selector started to predominantly choose uniaddress 2. At time B, a similar delay was introduced into uniaddress 2, so the selector again chooses each uniaddress about equally. At time C, the delay is removed from uniaddress 1, so the selector starts choosing it with greater frequency than uniaddress 2.

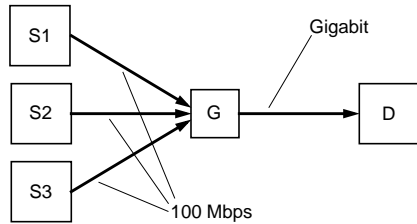


Figure 7. The configuration for the matter test consisted of three source machines S1, S2, and S3 (Sun Blade 100s), a gateway G (Sun Ultra 80, 2x500 MHz), and a destination D (Sun Ultra E450, 4x400 MHz).

previously slow uniaddress has improved. The selector code is given in Figure 6.

4.2 Matter

This test demonstrated that matter and wormhole matter are effective in improving the performance of intermediaries. The test used five machines: S1, S2, S3, G, and D. S1, S2, and S3 sent messages to G, which then routed them to two different nodes on D (Figure 7). Each message had one part containing an array of doubles. Two protocols were tested, SOAP and our binary protocol (Figure 8).

Because the binary protocol processes doubles with little overhead, using matter offers little performance benefit over a vobject. Wormhole matter, however, offers significant enhancement when the message size is large. Matter perfor-

```

AdaptSelector::send(const Scheme &scheme,
                   const Nodepnt &ret_addr,
                   const Message &msg) {

    int u;
    double r = drand48();

    // Choose each uniaddress at least
    // 1% of the time.
    if (r < .01) {
        u = 0;
    } else if (r < .02) {
        u = 1;
    } // Otherwise, choose based on how
    // fast it is.
    } else {
        // Renormalize.
        r = (r - .02)/.98;
        double sqr0 = t_put[0]*t_put[0];
        double sqr1 = t_put[1]*t_put[1];
        double cut = sqr0/(sqr0 + sqr1);
        if (r < cut) {
            u = 0;
        } else {
            u = 1;
        }
    }

    Time start(Time::Now());
    scheme[u].send(msg, ret_addr);
    Time elapsed = Time::Now() - start;

    t_put[u] = .9995*t_put[u]
               + .0005/elapsed.toDouble();
}

```

Figure 6. Code for the adaptive selector. The variable *u* is the index of the uniaddress to be used. *t_put* is a double array used to maintain the current throughput of each uniaddress. A moving average is used to smooth the measured throughput.

mance actually degrades with large message sizes because the data is read in one long pass and then transmitted in another long pass. The latter part of the first pass flushes the beginning of the message from the cache, so it must be reloaded for the second pass. Since the wormhole matter can be read in small chunks, each chunk can be retransmitted while still in the cache. This was verified by examining cache misses with Sun's Performance Analyzer.

SOAP serialization/deserialization overhead is significant, so shows considerable benefit from using matter. (SOAP wormhole matter has not been implemented yet.)

5 Related Work

As mentioned previously a WSDL[4] port can be bound to multiple protocols. WSDL is a description language, however, and provides no facilities for actual client implementations.

Madeleine II[1] provides a low-level mechanism for multiple protocols. Its abstractions are too low to satisfy our requirements, however. For example, an intermediary cannot

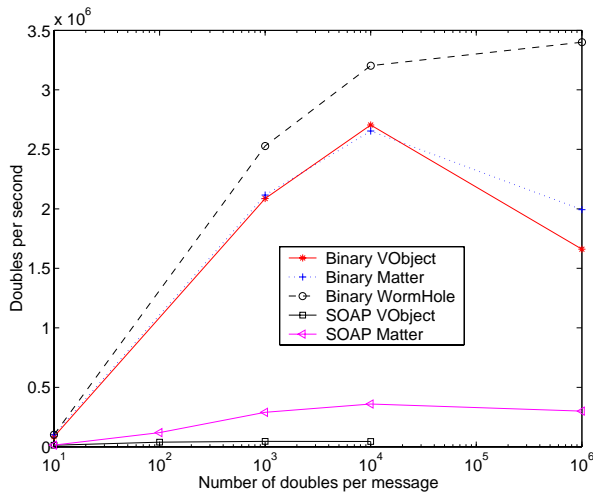


Figure 8. This graph shows the performance benefit of matter and wormhole matter. For CPU-intensive protocols such as SOAP, the performance of an intermediary is significantly enhanced by deserializing a part as matter instead of a vobject. For more efficient binary protocols, however, deserialization to matter offers little benefit over complete deserialization. Such protocols still benefit from the use of wormhole matter.

process messages for which it does not already know the contents, because it must always process each field.

CORBA[15] specifies an interoperable reference (IOR) that can contain multiple protocol profiles. CORBA provides no means to add different protocols to an Object Request Broker, however. Such details are internal to the implementation.

Diwan[5] investigated how global pointers can be used to reference remote objects through multiple protocols, similarly to CORBA IORs. His system did not provide a way to implement generic intermediaries, nor a standard mechanism for adding providers.

Nexus[8] provides protocol modules that can be used to send data over multiple protocols. The multiplexing occurs at the transport level, however, and thus cannot be used to bridge different higher-level protocols.

The PBIO[6] package allows efficient binary communication between disparate architectures. However, it does not provide means to communicate through different high-level protocols such as JMS or IIOP.

Govindaraju et al.[12] proposed an architecture for Java RMI multiprotocol communication.

Microsoft .NET Remoting also has facilities for multiprotocol communication, but, as far as we have been able to determine as of this writing, does not have the ability to read message parts without deserializing them, as can Proteus through matter (Section 3.2).

The networking community has also investigated multiprotocols, but in the context of transport protocols such as ATM or Frame Relay. For example, Multiprotocol Layer Switching[16] is a standard for establishing end-to-end circuits crossing multiple Layer 2 transports.

We note that much of the previous low-level multiprotocol work is entirely complementary to Proteus. In fact, we antici-

pate that providers of high-level protocols such as SOAP will use a low-level multiprotocol library to support multiple transport bindings.

6 Conclusions and Future Work

The Proteus multiprotocol library provides a flexible infrastructure for the research and development of Grid systems. By specifying the message contents with vobjects, clients are shielded from provider specifics. Any required details about how the provider should map vobjects to the transfer syntax can be specified through message maps. Furthermore, message maps can be created from strings, obviating the need to recompile for reconfiguration. We have shown that our abstractions are suitable for a range of providers by implementing a binary provider, a SOAP provider, and a JMS provider.

The actual provider to be used for sending a message is controlled through the selector mechanism. By specifying a general selector interface, clients can experiment with various adaptive protocols (Figure 5).

Performance was an important goal in the development of Proteus. To address this we introduced the concept of matter and wormhole matter. Matter allows Grid intermediaries to process messages without the full onus of deserialization. Wormhole matter allows intermediaries to process messages without consuming memory equal to the size of the message, and allow the front of a message to be retransmitted before the end has arrived. Our test has demonstrated that these concepts can be used to improve performance (Figure 8).

Issues for the future include security and providing more information to the selector. A WSDL layer to facilitate service descriptions is also planned.

7 References

- [1] Olivier Aumage, Luc Bougé, and Raymond Namyst. A Portable and Adaptive Multi-protocol Communication Library for Multithreaded Runtime Systems. In *Parallel and Distributed Processing. Proceedings 4th Workshop on Runtime Systems for Parallel Programming (RTSPP 2000)*, Volume 1800 of Lecture Notes in Computer Science, pages 1136-1143, Cancun, Mexico City, May 2000.
- [2] Don Box, et al. *Simple Object Access Protocol (SOAP) 1.1*. May 2000. <<http://www.w3.org/TR/SOAP/>>.
- [3] Kenneth Chiu, Madhusudhan Govindaraju, Randall Bramley. Investigating the Limits of SOAP Performance for Scientific Computing. In *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing*. July 2002.

- [4] Erik Christensen, et al. *Web Services Description Language (WSDL) 1.1*. March 2001. <<http://www.w3.org/TR/wsdl>>.
- [5] Shridhar Diwan. *Open HPC++: An Open Programming Environment for High-Performance Distributed Applications*. Ph.D. Thesis. Indiana University. 1999.
- [6] Greg Eisenhauer and Lynn K. Daley. Fast Heterogeneous Binary Data Interchange. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW 2000)*, pp 90-101.
- [7] I. Foster, C. Kesselman, J. Nick, S. Tuecke. *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. <<http://www.globus.org/research/papers/ogsa.pdf>>.
- [8] Ian Foster, Jonathan Geisler, Carl Kesselman, and Steve Tuecke. Managing Multiple Communication Methods in High-Performance Networked Computing Systems. *Journal of Parallel and Distributed Computing*, 40:35--48, 1997.
- [9] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., San Francisco, 1999.
- [10] Geoffrey C. Fox and Shrideep Pallickara. The Narada Event Brokering System: Overview and Extensions. *Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*.
- [11] Erich Gamma, et al. *Design Patterns*. Addison-Wesley, 1995.
- [12] Govindaraju, et al. Requirements for and Evaluation of RMI Protocols for Scientific Computing. In *Proceedings of the 2000 Conference on Supercomputing*. November 2000.
- [13] R. Moats. *URN Syntax*, RFC 2141. May 1997. <<http://www.ietf.org/rfc/rfc2141.txt>>
- [14] Henrik Frystyk Nielsen and Satish Thatte. *Web Services Routing Protocol*. <<http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-routing.asp>>
- [15] Object Management Group. *CORBA/IIOP 3.0 Specification*. 1999. <http://www.omg.org/technology/documents/recent/corba_iiop.htm>.
- [16] Eric C. Rosen, A. Viswanathan, and R. Callon, *Multiprotocol Label Switching Architecture*, RFC 3031, January 2001. <<http://www.ietf.org/rfc/rfc3031.txt>>.
- [17] Sun Microsystems, Inc. *Java Message Service Specification*. April, 2002. <<http://java.sun.com/products/jms/docs.html>>.