Internet Metaobject Protocol (IMOP): Weaving the Global Program Grid

Stefan Hong
Dept. of Information Management
National Taiwan University
Taipei, Taiwan
Email: stefanhong@ntu.edu.tw

Yuh-Jzer Joung
Dept. of Information Management
National Taiwan University
Taipei, Taiwan
Email: joung@ntu.edu.tw

Abstract—Software applications are increasingly relying on networks to function, but making programs to interact over the network is still tedious and error-prone. Conventional technologies such as CORBA and the WS-* stack are complicated to use, whereas RESTful style operations rely on costly ad-hoc developments on a per-service basis. We believe the problem lies in the lack of a network protocol that can solely and sufficiently address interoperability needs.

In light of this, we developed Internet Metaobject Protocol (IMOP), a remote method invocation protocol for object-based resource representations. IMOP thoroughly defines operations required to facilitate interactions, from reflecting a resource's definition to invoking its methods. It also rigorously defines the types of data passed between systems, including primitive types, composite value types, and reference types. All of these are programming language neutral.

Index Terms—Internet Metaobject Protocol (IMOP); Meso; Global Program Grid; Uniform Type Locator (UTL); Internet Protocol; Remote Procedure Call (RPC)

I. INTRODUCTION

Finding an easy way to make programs interoperable over networks has been a long quest for many decades, but the rich body of existing work still left a lot to be desired. Technologies such as Common Object Request Broker Architecture (CORBA) [1] and the conventional Web Services (WS-*) stack [2] strive to rigorously define the interaction between programs over networks, while also providing a great flexibility in operational models and networking transport supports [2]. Such ambitious goal inevitably complicates the overall design, and leaves rooms for software vendors to inadvertently or even intentionally implement the specification in a slightly incompatible way [3]. As a result, these heavy-weight solutions often hinder, ironically, the interoperability that they set out to achieve.

Apparently, the most successful network-based system technology that scales both up to the size of the Internet and out to a wide variety of applications is the World Wide Web. However, the web was designed to be a distributed hypermedia system that aimed to facilitate the delivery of a resource from one location to another when it is requested, made mostly by a human upon following a link. It was not intended to facilitate generic machine-to-machine operations.

Nevertheless, more and more people leverage the web, and use the Representational State Transfer (REST) architecture style [4] to build modern distributed applications. The RESTful style aims to model the data transferred between networked systems through a well-known interface, rather than the imperative interfaces between systems with a well-known data format [4], [5], [6]. Many people argue that the dissonance between the data-centric model and the predominate imperative programming style makes REST principles difficult to apply consistently throughout the system [7], [8], [5], [9]. In addition, the interoperability is still hard to achieve without a commonly agreed data representation [9].

In short, the RESTful style defines too little to standardize the interaction between programs over the network; some ad hoc patches are often necessary to make them interoperable. Conversely, technologies such as CORBA and the WS-* stack define too much for the sake of interoperability; the additional goals such as code portability and transport flexibility bloat their overall designs and obscure the interoperability itself.

We believe that building network-based applications can be greatly simplified by having a network protocol whose design focuses on—and solely on—machine-verifiable interoperability. We therefore designed a protocol named Internet Metaobject Protocol (IMOP) to fill this role. IMOP is a remote method invocation protocol for object-based resource representations. It is text-based and works in the application layer. The novelty of IMOP lies in the explicit definition of both protocol-level methods and instance-level methods supported by a resource in the network. The former provides a general framework to establish a conversation with a resource, whereas the later operates the resource itself. Both levels of methods have machine-processable definitions. In particular, protocol-level methods are standardized by the protocol, while instance-level methods are defined in a descriptor that may be obtained through an over-the-wire reflection mechanism.

IMOP is one of the three elements that constitute the Global Program Grid (or simply the grid)—an architecture we proposed to facilitate an Internet-scale network-based object system in which programs are interoperable through machine-verifiable interfaces. The other two elements are: (1) a resource type naming scheme called Uniform Type Locator (UTL); and (2) a strongly and statically typed programming language named Meso [10]. The roles of these three elements can be analogized to the roles of URL, HTTP and HTML in the World Wide Web (see the table below). The web links documents by URLs, while the grid associates programs by UTLs. HTTP
serves as the de facto protocol for accessing documents in the web, while IMOP is the main protocol for programs to communicate in the grid. Finally, HTML is the predominant language for combining various resources in the web into one document, while Meso is the primary language for assembling heterogeneous entities in the grid into one program.

<table>
<thead>
<tr>
<th>Resource</th>
<th>World Wide Web</th>
<th>Global Program Grid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>URL</td>
<td>UTL</td>
</tr>
<tr>
<td>Protocol</td>
<td>HTTP</td>
<td>IMOP</td>
</tr>
<tr>
<td>Language</td>
<td>HTML</td>
<td>Meso</td>
</tr>
</tbody>
</table>

This paper is devoted to the introduction of IMOP. Section II presents the IMOP protocol and some design details. Section III uses examples to demonstrate how IMOP messages enable programs to interact in the grid. Section IV discusses related work and concludes.

II. THE PROTOCOL

A. Overview

IMOP is essentially a text-based remote method invocation protocol that also supports over-the-wire reflection. It works in the application layer. We have chosen TCP to be its default transport protocol so as to realize the full-duplex feature that allows both ends of an IMOP connection to send request messages to the other. However, to speed up the adoption phase of our technology, we have also implemented a version of IMOP over HTTP to ease the development of web-based applications in the grid. Since HTTP is half-duplex, only one end of an IMOP/HTTP connection can actively send request messages to the other.

Resources operated by IMOP are object-based representations. They are object-based in that different resources may implement a common interface type, allowing a client to use them polymorphically through that interface. In addition, these resources are just abstract representations because they do not directly express the implementation behind. An object operable through IMOP might be implemented by one program in a system, or served by hundreds of servers with load-balancing and fault-tolerant mechanisms. Such implementation differences are encapsulated behind the abstract representation which IMOP operates on.

Before interacting with a remote resource, a client may first perform a reflection on the resource to obtain a descriptor of it. The descriptor, whose format is also defined by IMOP, provides detailed information about the resource, such as its type or the interfaces it implements. The client can subsequently operate the resource according to the descriptor.

IMOP is a strongly-typed protocol in the sense that conversations made by the protocol are governed by a type system. Data types are essential for both protocol-level and instance-level methods.

B. The Design Philosophy

To capture the design philosophy of IMOP, we first illustrate the design of HTTP in the top half of Fig. 1. The Web is designed for the sharing of hypermedia resources, so the primary concern of HTTP is the movement of data—mostly from a publisher to a consumer, and sometimes in an opposite direction as well. HTTP fulfills this need by providing standard methods such as GET and POST. Fielding [4] later interpreted this architecture as transferring resource representations through a common interface.

Alternatively, this design may be viewed from how resources are operated, as depicted in the bottom half of Fig. 1. One may envisage HTTP as an RPC protocol designed to invoke a predetermined set of methods, namely GET, POST, PUT, ... etc., on an implicit metaobject addressed by the URL. Through these invocations, the base-object (i.e., the actual resource) associated with the metaobject can be obtained or updated. The available operations applicable to the base-object are implied by the resource’s MIME (Multipurpose Internet Mail Extensions) content type. For example, the text/html type implies some parsing and displaying operations for processing an HTML document instance.

While this design works very well for hypermedia systems, it is less so for program-to-program interactions. The difference lies in the complexity of exchanged data. Hypermedia systems can function properly by supporting only a small number of data formats such as HTML or JPEG, but interactions between programs usually require much finer-grained definitions. Simply understanding the XML format, for example, is not enough to achieve interoperability [11]. In other words, base-level operations have to be more specific than those implied by a MIME content type. HTTP does not further address such fine-grained definitions. After all, it is not designed to facilitate program interactions.

IMOP tackles this problem by elaborating on the concept of metaobject, as illustrated in Fig. 2. IMOP defines meta-methods in the protocol to operate the metaobject addressed by a URL. Through the use of meta-methods, methods applicable to the corresponding base-level object can be reflected or invoked. Meta-methods are just like GET and POST in HTTP, but IMOP defines its own set of methods, among which, DESC and CALL are the most frequently used ones. DESC instructs a metaobject to describe its associated instance by returning a machine-processable descriptor of the object—such operation is known as reflection or introspection [12]. CALL commands the metaobject to invoke a specified method on the associated instance with arguments attached in the message. The fact that HTTP and IMOP have different meta-methods distinguishes
their philosophical differences: HTTP moves resources, but IMOP does not—it operates resources remotely.

Metaobjects effectively separate two layers of interoperability concerns: the meta-level interoperability that is essential to establish base-level communications, and the base-level interoperability that is necessary to facilitate program interactions. Both are clearly addressed by IMOP. The former is achieved by the meta-methods predefined in the protocol, while the latter is governed by interface descriptors obtainable through per-object reflections.

The use of metaobject might seems indirect or even inefficient, but in fact it is only a notion used to clarify the concept behind the protocol. The syntax of an IMOP message is not much different from that used in HTTP, and both can be handled by software at a similar degree of complexity.

C. The rationale of creating a new protocol

Because of the syntactical similarity with HTTP, one might think that HTTP can be easily extended with DESC and CALL methods, thus rendering IMOP unnecessary. However, there are three problems with this approach.

The first one concerns the fundamental philosophical difference between DESC/CALL and existing HTTP methods. Mixing them in one protocol may cause confusion. For example, it is unclear, at least from the URL alone, that whether http://foo.com/bar represents a resource that can be obtained over the network, or must be operated remotely. A different protocol name stated in the URL scheme can clearly distinguish the difference. The second problem concerns the orthogonality between methods. Many existing HTTP methods are able to carry arbitrary query parameters in the request URI or in the message body. They provide a somewhat similar but inconsistent functionality with CALL. Such inconsistency is not preferred in a protocol’s design. The third problem concerns the half-duplex nature of HTTP. Programs often interact in a full-duplex manner that one will actively send requests to the other. If they connect through a firewall or a NAT where one end does not have a public IP address, it may be crucial to perform full-duplex messaging at the application layer. This, however, is not supported by HTTP.

These problems are hard to overcome without breaking the compatibility with existing HTTP. As a result, we believe that creating a new protocol would be more appropriate. It also allows us to improve some aspects of HTTP that are otherwise not possible for the sake of compatibility, such as using less verbose headers.

D. Uniform Type Locator (UTL)

Before discussing the type system used in IMOP, one needs to understand the notation of Uniform Type Locator (UTL). Syntactically, a UTL is just a URL, but we use the term UTL to explicitly refer to the informal subset of URI that identifies a data type using three components: A Type Scheme, an optional Type Authority, and a Local Name, as follows:

\[ \text{imop: \text{api.org} / \text{service.IBulletin}} \]

The type scheme specifies the mechanism required to access the type. The type authority, if it exists, specifies the location where the type is defined. A UTL that contains an authority component is called a global UTL, otherwise it is a local UTL. Finally, the local name component identifies the type within its authority location or the type scheme. The syntax of the components and the delimiter used between them are type scheme dependent, so long as the whole UTL complies with the URI’s specification.

All global UTLs should be reflective so that a descriptor of the type is obtainable from its authority using the mechanism implied by its type scheme. The format of the descriptor may be negotiable between a client and the authority. Being reflective means that each global UTL bears a consistent meaning throughout the Internet regardless of the context. In contrast, the definition of a local UTL is either predefined by the scheme, or is resolvable in a way known to the scheme that may lead to context dependent definitions.

E. Instance-level and Protocol-level types

Fig. 3. Examples of instance-level types

Methods in IMOP can be classified into two different levels: protocol-level and instance-level. Each level of methods have their own data types. Fig. 3 shows some types used by instance-level methods, which include value types and reference types. Data of value types are passed by value over the connection when used as a method argument. This is true even for structure and array values. The entire content of these composite values are serialized and transferred over. Conversely, reference type resources are never moved nor
copied. Instead, the reference to the resource—a URL—is transferred over the network.

Protocol-level types are the format used to express instance-level information in an IMOP message. These types are often known as content types. They are processed by metaobjects, and can be changed without affecting the instance-level methods. For example, a type descriptor may be transferred in a JSON or XML format, but either format provides the same information about the definition of an instance. Similarly, different content types may be chosen to encode method arguments and return values for an IMOP method invocation.

The default content type used by IMOP is `mime:application/imop.json`, which represents a JSON-based format that can express type descriptors, as well as carry data for method invocations. Below we briefly illustrate this format.

Fig. 4 shows a type descriptor returned from a reflection request. The UTL of this descriptor is implied by the request message and is not stated in the descriptor. This descriptor shows that the instance is an object interface, which implements two interfaces. It also provides two additional methods, `func1` and `func2`, that are not defined in the interface types it implements. A client that processes this descriptor might want to further reflect the global UTLs used in the descriptor to obtain their definitions. These definitions may refer to more global UTLs. For example, an interface type may be derived from other interface types. The client therefore might perform reflection operations recursively until all directly and indirectly referred global UTLs are resolved.

Fig. 5 shows an example of a method invocation content, which invokes the `func1` listed in Fig. 4. Notice that value of the first argument is an object instance `imop:xyz.com/obj`, but its type field uses an interface type `imop:bar.com/if3` that the object implements, rather than using the object’s UTL itself. This explicit distinction is needed because IMOP supports method overloading such that multiple methods may have a same name but only differ in their accepted argument types. The selection of a target method may become ambiguous when an object implements multiple interface types that may be polymorphically accepted by multiple methods. Explicitly stating the intended argument type can avoid such ambiguity. In programming language’s term, the object is `cast` to the interface type. The second argument in Fig. 5 is an array, whose value is encoded using JSON’s array syntax. If the value belongs to a structure type, it will be encoded in JSON’s name-value pairs.

III. THE GLOBAL PROGRAM GRID

In this section, we use examples to demonstrate how programs in the grid interact through the IMOP protocol.

A. The Meso language

It is hard to describe the web without using HTML. Similarly, the whole picture of the grid can be more clearly presented with the Meso language. Although programs do not have to be written in Meso to be part of the grid, Meso is a strong testimony of how intuitive building a network-based application can be. Meso is a strongly and statically typed object-oriented programming language. Its syntax is very similar to Java, and it is designed to incorporate UTL and IMOP natively in the source code. A complete description of the language is beyond the paper’s scope, so we offer only the information that is needed to understand the examples given in the section. For tutorial and programming examples of Meso, please refer to http://gpgrid.org/.

A Meso source code always begins with a `namespace` statement, which states the UTL prefix of the type defined in it. For example, the Meso source code at `bb.com` in Fig. 6 states `imop:bb.com/pb` as the namespace, which gives the object `B` a global UTL `imop:bb.com/pb.B` that represents an IMOP object. Per the definition of global UTL, this resource must be accessed from the authority `bb.com` through IMOP. As a result, the source code is only meaningful when executed at the host `bb.com`. In contrast, the source code for the class `Ap1` in Fig. 6 is defined under the namespace `meso:foo`, which gives the type a local UTL `meso:foo.Ap1`. The `meso` type scheme is private to the Meso language, where types defined in this scheme can only be used by the language locally; they bear no identities in the network. A type like `meso:foo.Ap1` (or `java:foo.Bar` in the Java scheme) may have different definitions on different hosts.

B. A simple interaction

We begin with the demonstration of the grid with a simple interaction between two programs shown in Fig. 6. In the figure, Meso source codes are enclosed in rectangle boxes attached to computer icons, while dialog balloons represent
conversation messages in IMOP. The program at the host \texttt{aa} in the left, which does not have a public IP address, invokes the method \texttt{funcB} on the object \texttt{B} at the host \texttt{bb.com} in the right. The \texttt{import} statement in the source code of \texttt{Ap1} allows the identifier \texttt{B} used in the source code to be resolved as the UTL \texttt{imop:bb.com/pb.B}.

Meso is a strongly typed programming language such that no operation is allowed to perform on an entity that does not support such operation. Therefore, when compiling the source code of \texttt{Ap1}, the definition of \texttt{imop:bb.com/pb.B} must be obtained to verify whether it can be invoked with the method \texttt{funcB(123)}. The compiler does so by asking from the compile-time handler of the \texttt{imop:} scheme, which in turns sends out an IMOP DESC request (1) over a TCP connection to the host \texttt{bb.com} on port 90\(^1\) to introspect the type \texttt{imop:bb.com/pb.B}. The dialog balloon in the figure shows the content of the message.

The request arrives at \texttt{bb.com} and is received by a runtime handler of the \texttt{imop:} type scheme, which is configured to listen to the port 90 and handle requests for the authority \texttt{bb.com:90}. The handler accepts the request and asks the Meso runtime to load the type \texttt{imop:bb.com/pb.B} from a local file under a pre-configured directory similar to the document root of a web server. The \texttt{imop} handler then generates a JSON descriptor from the loaded type and sends it back in the response message (2). The JSON descriptor shows that the instance at the UTL is an object, and lists the methods that the object supports. This information allows the compiler at the host \texttt{aa} to verify the usage of the object before generating code to access it. The sequence number “1” used in both request and response envelopes is a unique message ID set up by the requester per connection so that response messages can be paired with request messages when they are delivered out of order.

During the execution of \texttt{Ap1}, the Meso runtime at \texttt{aa} performs the invocation of the method \texttt{funcB} by handing it over to its runtime handler of the \texttt{imop:} type scheme. The handler creates an IMOP CALL request (3) to the object \texttt{imop:bb.com/pb.B}, and serializes arguments sent by the runtime into a JSON-based format stored in the content. The request is processed by the handler at \texttt{bb.com} by deserializing arguments and then invoking the actual method on the object. A response message (4) is returned to indicate a successful invocation.

C. Using interface types

In the previous example, the remote resource used by \texttt{Ap1} is predetermined at the development time since the UTL \texttt{imop:bb.com/pb.B} is hard-coded in the source code. In this section, we demonstrate how IMOP enables programs to interact with an open set of resources that support the same interface type. The example is illustrated in Fig. 7. In this example, a client at \texttt{Ap2} invokes a service \texttt{C} at \texttt{cc.com} to operate an object \texttt{E} at \texttt{ee.com}. Both \texttt{C} and \texttt{E} reference an interface \texttt{D} at \texttt{dd.com}.

1) Conversations for type verification: Fig. 7 depicts the IMOP conversations initiated by the Meso compiler at \texttt{aa} when compiling the source code of \texttt{Ap2} in the left. First, to verify whether \texttt{funcC} can be applied on the identifier \texttt{C}, the compiler makes a DESC request (1) to reflect the definition of \texttt{imop:cc.com/pc.C}. The response message (2) shows that the resource is an object instance, which supports a method \texttt{funcC} that takes one argument of type \texttt{imop:dd.com/pd.D}. This argument type has yet been seen by the compiler at \texttt{aa}, so the compiler makes another DESC request (3) to the host \texttt{dd.com} to reflect its definition. The “kind” field in the descriptor sent by the response message (4) is “i”, meaning that the resource is an interface instance. It is an abstract resource that does not

---

\(^1\)Port 90 is the default port number of IMOP, although its use has not yet been officially registered with the Internet Assigned Numbers Authority (IANA).
provide the methods listed in the descriptor. Sending an IMOP CALL request to it will result in a “405 Protocol-method not allowed” response.

Once the compiler at aa understands the definition of \texttt{imop:dd.com/pd.D}, it continues to process the expression \texttt{funcC(E)} in Ap2. Now the compiler needs to verify whether the argument \texttt{E} supports the interface type \texttt{D} required by \texttt{funcC}. It makes yet another DESC request (5) to the host \texttt{ee.com} to introspect the type \texttt{imop:ee.com/pe.E}. The descriptor in the response message (6) shows that the resource is an object instance, and the “impls” field indicates that the object indeed implements the interface type \texttt{imop:dd.com/pd.D}. The argument \texttt{E} therefore passes the compiler’s type checking.

2) Conversations for execution: Fig. 8 depicts the IMOP conversations when executing \texttt{Ap2}. To invoke the method \texttt{funcC}, the Meso runtime at aa sends a CALL request (7) to \texttt{cc.com}, and puts the object reference \texttt{imop:ee.com/pe.E} as the value of the argument. In addition, the reference is explicitly cast to the type \texttt{imop:dd.com/pd.D} to help \texttt{cc.com} select the correct method should there be multiple \texttt{funcC} methods overloaded with different argument types.

When the Meso runtime at \texttt{cc.com} accepts the request and deserializes the argument, it encounters the reference \texttt{imop:ee.com/pe.E} that it has never seen before. To ensure that the reference can be used by the method, the runtime sends out the DESC request (8) to reflect its definition. The response (9) confirms that the reference does support type \texttt{D}, and so the runtime accepts it and stores the reference into the variable \texttt{obj}, and then starts executing the method \texttt{funcC}. Even though Meso always introspects an unknown UTL before using it, programs written in different languages may have a different policy. They may choose to trust the request message (7) which states that the reference \texttt{E} supports type \texttt{D}, and use the reference without further verification.

To execute the expression in \texttt{funcC}, the runtime at \texttt{cc.com} sends a CALL request (10) to invoke the method \texttt{funcD} on the reference \texttt{imop:ee.com/pe.E} stored in the variable \texttt{obj}. After the runtime receives the response (11) from \texttt{ee.com}, the execution of \texttt{funcC} is completed, and a response message (12) is sent back to the original requester \texttt{aa}.

D. Serving from behind a firewall

In today’s Internet environment, many hosts are located behind a firewall or a network address translator (NAT), and are unable to be actively connected from the Internet. In this section, we demonstrate how the full-duplex feature of IMOP enables these hosts to serve resources to the Internet. The example is shown in Fig. 9. In this example, the host \texttt{aa} represents a host that does not have a public IP address. Its application \texttt{Ap3} wishes to invoke \texttt{funcC} of service \texttt{C} at \texttt{cc.com} on the private object \texttt{meso:foo.F} located at \texttt{aa}. Since the object \texttt{F} also implements the interface type \texttt{imop:dd.com/pd.D}, it can pass the compiler’s type checking.

Executing the invocation is more intriguing. The Meso runtime at \texttt{aa} cannot simply send the reference name \texttt{meso:foo.F} to \texttt{cc.com} since it is a local UTL, which will be interpreted by \texttt{cc.com} as a resource located locally at \texttt{cc.com}, rather than the one at \texttt{aa}. The object \texttt{F} needs a global UTL that can identify its location in the Internet.

IMOP makes this possible by providing an indirect authority address. If the Meso runtime does not have a public authority address when constructing a global UTL for a local resource, it will send an IMOP BIND request (1) to the host where the resource is intended to be used, which is \texttt{cc.com} in this example. The remote host replies with a private name \texttt{h001} and an opaque handle (2), both of which may be generated...
randomly. With that private name, the host aa becomes h001@cc.com—an indirect authority address that identifies the private host h001 reachable from cc.com. This address faithfully reflects the location of aa so long as its connection to cc.com is maintained. If the connection is broken, the host aa may try to re-establish the same identity by sending another BIND request via a new TCP connection, along with the host name h001 and the original opaque handle set in the request envelope. The opaque handle is simple—and albeit insecure—way to authenticate itself. The host cc.com may accept that requested name, or assign a new one in the response message.

Once the indirect address is obtained, the Meso runtime at aa can assign the local object meso:foo.F a new global UTL by combining the authority address h001@cc.com with a randomly generated local name F1 (3). The new UTL imop:h001@cc.com/F1 is then used as the argument to the CALL request (4) to invoke the method funcC at cc.com.

When the request arrives at cc.com, the Meso runtime there, again, needs to introspect this UTL before accepting it as the argument. The runtime knows that one of its existing connections can reach the authority h001@cc.com, and sends the DESC request (5) through that connection. The response message (6) shows that the UTL is an object instance that implements the interface type D, and thus cc.com accepts the reference imop:h001@cc.com/F1 and stores it in the variable obj. Notice that the descriptor in the response message does not reveal the UTL meso:foo.F. The UTL is completely private to aa.

After accepting the argument, the execution of funcC at cc.com begins, wherein the method funcD invokes on the reference stored in obj. The CALL request (7) is sent through the same connection as the DESC request, and is ultimately processed by the object F at aa. When the execution of funcD is completed, a response message (8) is returned. Finally, another response message (9) is sent from cc.com to indicate that the original CALL request (4) is completed.

The UTL imop:h001@cc.com/F1 not only works between aa and cc.com, but can also be passed around the Internet without affecting its identity. To make conversation with the type, a program at a different host will connect to the host cc.com and send requests with the address h001@cc.com/F1 in the envelope, effectively using cc.com as a proxy. The host cc.com may decide whether to fulfill such requests.

E. Summary

As we have illustrated above, with interface types and the over-the-wire type reflection supported by IMOP, not only a program can interact with an open set of remote resources that it has not “seen” before, but such interaction can also be rigorously and automatically verified by a runtime system. In addition, the object-oriented nature of the language and the protocol allows one to extend an existing interface and build objects based upon the new interface. Existing objects that understand only the old interface can still safely interact with these new objects via polymorphism, and vice versa. For example, in the Ap2 example in Fig. 7, suppose site ee.com has a new object E’ that implements an interface D’ extended from D. The site cc.com can still use existing service C to operate E’, even though it does not know the new interface D’.

Although the figures in this section might seem complicated, the actual IMOP messages exchanged between hosts are quite simple. Most operations can be achieved by DESC and CALL messages, and their JSON-based content is very lightweight. These conversations do not need a heavy-weight middleware to process. They can be made by simple libraries written in many programming languages.

Moreover, if we ignore the dialog balloons of IMOP, the program codes in the examples are remarkably simple, even though they are meant to deal with object interactions over the Internet. For example, the program code in the Ap2 example references and operates remote objects simply by their UTLs. These expressions are very intuitive and easy to understand. The global uniqueness feature of UTLs and the type reflection mechanism of IMOP relieve programmer’s burden to manually relate a reference name to the actual target object. As such, developing applications in an Internet-like open environment is as easy and type-safe as developing them in a single runtime environment where one needs not worry about name resolution between different runtime environments.

IV. RELATED WORK AND CONCLUSIONS

Existing technologies for program interactions are often based on remote procedure calls (RPCs), e.g., CORBA, the WS-* stack [13], .NET Remoting [14], Java RMI [15], and DRB [16]. They are designed to make remote objects look like local ones so that detailed procedures in handling a remote method invocation can be hidden from a programmer. Still, cumbersome steps such as object registration and binding are often necessary to relate remote objects in a local code. Fig. 10 gives an example in Java RMI, where a client wishes to use an object created by a server. Before the use, the server must register the object to an RMI registry at foo.com with the network name rmi://foo.com/B, while the client must resolve that name from the same registry and binds the result to a dynamic proxy.

Also note that the object is of class IBimpl, which implements an interface foo.IB. The interface file must be obtained by the server side and by the client side respectively for their code to be compiled. It is possible that the server and the client obtain different interface files of foo.IB, so that each of them can be correctly compiled but execution of them will result in a runtime error due to an inconsistent interpretation of the
The need of human effort to ensure that a remote service expressed in code is indeed the intended one also haunts the development processes using technologies like CORBA [3], WS-* stack [13], [17], [18], and REST-style web services. In CORBA, an IDL file is used to generate the client side and the server side programs. If they are developed independently by different programmers, the programmers must make sure they use the same IDL file. Similarly, WS-* stack relies on WSDL files to describe services so that clients can correctly bridge to servers. Multiple copies of service descriptors under the same name may be circulated, and it is the programmer’s responsibility to pick up the right one. REST-based web services require even more human effort to ensure a correct interfacing with a remote service [4], [7], [5], [8], [9], as there is no machine processable description to reveal what a service is. As a result, a compiler is not able to interpret the actual semantics expressed in a statement like

```
http.get(http://bb.com/fb/B)  // (e.g., calling fb(B) or B(fb), or anything else), thus unable to provide protection against remote interactions. Recent proposals such as hRESTs [19] and WASL [20] attempt to give RESTful services a machine processable descriptor, but the descriptor cannot be uniquely located by knowing a service name alone, thus unable to give services an unambiguously and consistent interpretation throughout the Internet.
```

In short, existing technologies for programming languages provide no mechanism to guarantee that a remote resource expressed in a program can be unambiguously interpreted in a truly global scope, or lack strong type checking thereof. In contrast, through the use of UTLs and IMOP’s reflection mechanism, our approach allows objects and data types to be consistently and automatically interpreted regardless of the context. For example, Fig. 11 illustrates programs that implement the same functionality in Fig. 10. The UTL

```
namespace imop:foo.com/;
import imop:foo.com/IB;
public interface IB { ... } namespace meso:abc;
import imop:foo.com/IB;
import ... object imop:foo.com/B
supports imop:foo.com/IB type
```

```
IMOP object imop:foo.com/B
supports imop:foo.com/IB type
```

with a globally unique resource type naming scheme UTL and an object-oriented and strongly-typed programming language Meso, to achieve this goal. IMOP not only enables heterogeneous resources to interact over the network, but the interactions are also performed in a rigorously defined manner to help tool automation and reduce development cost. In addition, IMOP’s reflection mechanism along with UTL’s global type naming scheme allows every resource in the Internet to bear a consistent definition. Therefore, independently developed systems may interoperate coherently and form a loosely-coupled software mesh over the Internet in an unprecedented scale—the Global Program Grid.

Finally, prototypes of the Meso language compiler and the grid runtime environment have been implemented and available for download from http://gpgrid.org/.

REFERENCES