Meso: An Object-Oriented Programming Language for Building Strongly-typed Internet-Based Network Applications*

Stefan Hong
Dept. of Information Management
National Taiwan University
stefanhong@ntu.edu.tw

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

ABSTRACT

Programming languages rely on type systems to safeguard the correctness of a program. In existing technologies, types defined in a program are meaningful only within its runtime, so the definition of a remote resource must be obtained and verified to ensure a correct and type-safe interaction. However, obtaining a consistent meaning of a resource in a distributed environment requires much human effort and coordination, thus making the development process difficult to scale to an Internet-like open environment.

The key lies in the lack of a truly global naming scheme to unambiguously and automatically resolve programmatic types in the Internet. In light of this, we propose Uniform Type Locator (UTL), a worldwide scope type naming scheme. We also design and implement a new strongly typed object-oriented programming language called Meso to natively support UTLs, and a protocol to facilitate program interoperations. We illustrate through examples how Meso can be used to make developing applications in an Internet-like open environment as easy and type-safe as developing them in a single runtime environment.

1. INTRODUCTION

Programming languages for network applications must be able to express interactions such as to operate Service.func() at host foo.com. Typically, an expression involves the name of the service but not its interface definition. To ensure a correct and safe interaction, however, the interface definition of a remote service must be obtained and type-checked prior to the interaction. For services to interact in an Internet-like open environment, a service must be unambiguously interpreted regardless of where its name is referenced. Surprisingly, existing languages and techniques (e.g., CORBA [6], WS-* stack [8], .NET Remoting [5], Java RMI [9], and DRb [7]) provide little help to ensure this without human intervention.

For example, in Java RMI, one must manually obtain a remote interface definition and incorporate it into the client program, or configure the class loader to allow the Java Runtime to dynamically load the interface definition from a designated location. Both processes are tedious, but more importantly, they cannot prevent one from mistakenly taking an interface definition as another. As a result, a client program might be compiled without errors but still not be correctly executed. Moreover, two different remote resources with the same name cannot be used simultaneously in a program, preventing independently developed programs from being seamlessly integrated into a third party program.

The above problems are less of a concern when an application is developed in a closed environment (e.g., within an organization), as one can enforce a naming convention to avoid conflicts and designate a common resource repository for obtaining a resource’s interface definition. They are, however, inevitable in an Internet-like open environment, as the naming conventions cannot be strictly enforced and no authority exists to resolve a conflict.

A similar problem occurs in CORBA, which relies on IDL files to generate client-side programs. WSDL tried to solve the naming conflict problem by using name space URIs, but there is still no way to associate a unique definition to a service name. Everyone could create a new WSDL file that gives an arbitrary definition to the service name. In short, existing technologies for programming languages provide no mechanism to guarantee that a remote resource referenced in a program is unambiguously interpreted in a truly global scope—the Internet.

Our contributions

To solve the above problems, we propose a new type naming scheme called Uniform Type Locator (UTL) to define data type in a truly global scope—the worldwide scope. Syntaxically, a UTL is identical to a URI, but semantically, it is a subset of URI that contains only programmatic data types. A UTL consists of three parts: Type Scheme, Type Authority, and Local Name, as follows:

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

Type authority specifies the location where the type is defined. Local name identifies the type within its authority location. Type scheme identifies the technology related to the type. Technology can be a language (e.g., Java or C#) or a standard (e.g., .NET’s Common Type System or MIME). Associated with a type scheme is a mechanism or proto-
col to interpret the definition of the type. All together, a UTL unambiguously identifies a unique resource type in the Internet. 

Type authority may be absent from a UTL. A UTL with an authority address is a global UTL; otherwise it is a local UTL. For example,

```java
namespace meso:service;
public struct Msg {
    public string user;
    public string body;
}
```

are two local UTls defined respectively in Java and in C#. We have deliberately made the UTls of language specific types local so as to explicitly distinguish them from those that are remotely accessible. This allows local and remote objects to be handled differently, as there are irreconcilable differences between them in, e.g., latency, memory access, security, failure, and concurrency [10]. To facilitate interactions over the network and across different platforms, we designed a new type scheme called imop: to define resources for remote operations, along with a new protocol called Internet MetaObject Protocol (IMOP) for operating them. Every imop: object has a global UTL. Its definition can be obtained from the authority address specified in its UTL via a reflection operation through the IMOP protocol. As such, every remote resource available in the Internet can be unambiguously interpreted in the global scope. 

Finally, we designed and implemented a new strongly and statically typed object-oriented programming language called Meso to natively support UTls. Many languages can be extended to support UTls. We chose Java’s syntax for its popularity. Although the syntaxes are similar, the use of UTls makes Meso very different from Java. Resources identified by UTls may exist across the network and/or in different language runtimes. Incorporating UTls in the type system gives Meso an unprecedented expressing power to operate and compose resources in heterogeneous environments. In particular, the language may act like a mediator to synthesize functionalities of different languages and network protocols into one program. We therefore coined the name “Meso” for the language, meaning “in the middle”.

The roles of UTL, IMOP, and Meso can be compared to those of URL, HTTP, and HTML in the world wide web (WWW). The web links documents by URLs, while Meso serves as the de facto protocol for accessing documents in the web, while IMOP serves as the de facto protocol for programs to interoperate. Finally, HTML is the predominant language for combining various resources in the web into a document, while Meso is the primary language for assembling various programmatic entities into a program. All together, we believe that the paradigm has the potential to allow programmatic resources to be interconnected in a scale comparable to the web. We therefore call this paradigm the Global Program Grid.

This paper is devoted to the introduction of the Meso programming language.

2. AN OVERVIEW OF MESO

We use some examples to illustrate Meso programming. For space consideration, the examples are kept as simple as possible to allow readers to quickly understand the language and our design philosophy. The language tutorial and more complex and practical examples can be obtained from our project website http://gggrid.org. While readers are going through the examples, we strongly encourage them who are familiar with existing technologies such as Java RMs, CORBA, SOAP, and REST [3], to try to write a similar program using the languages provided in these technologies.

One can easily overlook the aforementioned problems without getting his hands dirty by writing some Internet-based network applications.

2.1 A local bulletin system

We begin with a typical object-oriented design in which an abstract interface is provided for one to implement. We use a bulletin system as an example, start it with a pure local program, and then gradually evolve it into a sophisticated network application. Since our focus is on the interactions between objects, non-related object content will be kept as simple as possible for ease of presentation. For example, in the bulletin system, we shall assume that a bulletin board provides only two methods: post for posting a message, and list for listing posted messages.

In the following figures, each box contains a Meso source file that defines a UTL listed right above it. All host names in the examples are fictional. They serve to make the examples easier to grasp.

Fig. 1 shows the abstract interface modeled by two types, meso:service.Msg and meso:service.IBulletin. The namespace keyword, similar to that in C# or the package keyword in Java, is used to declare a scope. The compiler prepends the namespace to a type’s name to construct the type’s full UTL. These two UTls do not contain a host name inside, so the scope of their definitions is limited to the local runtime environment, just like types in a typical language. The meso: portion in their names means that the types can be accessed in the way defined by the Meso language. The meso:service.Msg is a struct type, which is a value type similar to that in C#. Fig. 2 shows an implementation of the interface. Board stores messages in an array, while Client sends it a fixed message. In this program, a new Board instance is created when Client starts from the main() method.

![Figure 1: An abstract bulletin interface.](image)

Fig. 1: An abstract bulletin interface.

We often want a class like Board, especially if it uses a database back-end, to be singleton so that it creates exactly one object. Meso has a special way to express a singleton class. In Fig. 3, Board is declared as object, which makes it a singleton type. A singleton’s UTL, such as meso:service .Board in this example, represents not only the name of the type, but also the reference to its object. The instance is created when it is referenced for the first time. Therefore, Board is used directly as the argument value to create a new Client in Fig. 3.

2.2 A typical network version

So far the entire program works only in a single machine, and is not very different from a Java program. Suppose we instead want to serve the bulletin board publicly so that anyone can write a client to access it over the Internet. This is a typical client/server architecture for a network application.

1Note that we use “Meso” (capitalized ‘M’) to refer to the programming language, and “meso” to refer to its type scheme.
To achieve this, we first redefine `meso:service.Msg` and `meso:service.IBulletin` to make their names and definitions uniquely identified throughout the Internet. Assume that `api.org` will host the definitions. We then replace the `meso:.portion` in the UTLs with `imop:api.org/`, as shown in Fig. 4, and physically store the code at `api.org`.

The `imop:api.org/` portion means that the type definition should be obtained from the authority address `api.org` via the IMOP protocol, so the source code in Fig. 4 are meaningful only when they are placed at `api.org`.

Next, we redefine the singleton object `Board` to make it worldwide accessible. Assume that the object is offered at `news.com`. The new code is shown in Fig. 5, which is almost identical to that in Fig. 3, except that the object’s UTL is changed to `imop:news.com/pub.Board`, and the `IBulletin` interface it implements is now defined at `api.org`.

Note again the code must be executed at `news.com` so that others can reach the `Board` object at the location indicated by its UTL.

The `Client` class, on the other hand, does not have to move to the worldwide scope. Each computer can have its own client implementation. The only modification to the `Client` class in Fig. 5 is that import statements now refer to remote types such as `imop:api.org/service.IBulletin`. When compiling the source code, the Meso compiler will acquire the definitions of all remote types via the IMOP protocol for type checking. During Client’s execution, `b.post(m)` invokes a method on the object stored in `b`, which is the remote object `imop:news.com/pub.Board`. The Meso runtime performs a remote method invocation, again via the IMOP protocol.

The following can be learned from the above illustration. First, developing an application with components interacting over the Internet is as easy as developing an application whose components interact only in a local runtime environment. For example, the main difference between Fig. 5 and Fig. 3 is just the UTLs: the former uses global UTLs, while the latter uses local UTLs. Moreover, when using remote resources, Meso requires no tedious name bindings and service registrations as those required in Java RMI, CORBA or Web Services. Each resource can be unambiguously and consistently interpreted via its UTL by every Meso compiler and Meso runtime regardless of where it is referenced. As such, interactions between components, whether local or remote, can be easily and intuitively expressed in a program. This improves code readability and thus reduces maintenance cost. Even so, because definitions of remote resources can be obtained over the wire from the authority addresses specified in their UTLs, strong type checking can still be offered to safeguard nonlocal interactions.

2.3 Object orientation and polymorphism
Next, we demonstrate how the object-oriented design paradigm [2] helps programming in an Internet-like open environment. In the previous example we have an abstract bulletin interface standardized by an organization api.org. Another company news.com followed the standard to implement a bulletin board for its clients. In the next example, we assume that several companies have followed the standard to implement their own bulletin boards. A site search.com then creates a directory service for a client program to post a message to multiple sites at once. Fig. 6 illustrates a program for the directory service.

Figure 6: A directory service at search.com

The imop:search.com/b.Directory object registers three services upon initialization, and accepts new registration through the register() method. The modified Client in Fig. 6 invokes the list() method on the Directory object to get an array of IBulletin objects. The post() method can be correctly invoked on each object inside as the type saw by the client is consistent with the type used by each IBulletin object and the compiler is able to verify this. Note that objects are always passed by reference—their UTLs that serve both as a singleton type name and as a reference to them. Therefore, the IBulletin[] array returned from Directory is really just a list of UTLs.

An interface defined in the worldwide scope can be extended just like that in a typical object-oriented programming language. Fig. 7 shows a new interface type imop:leet.net/type.IBulletinEx, provided at leet.net, that extends the definition of IBulletin provided at api.org. It overloads the post method to accept multiple messages in bulk. The Group object at social.com, shown in the right of Fig. 7, implements the IBulletinEx interface. Thanks to polymorphism, the Group object can still register itself into Directory and be used by the same Client in Fig. 6, because Group can be operated as an IBulletin object.

2.4 Abstract Representation vs. Implementation

In the previous example we illustrated how object orientation helps programming in the Internet platform. An (object-oriented) interface can be defined and made available in the worldwide scope so that others may implement and extend it. An extended interface in turn can be made available via a global UTL (e.g., Fig. 7). However, once an interface is made concrete, for it to be available in the worldwide scope, it must be of a singleton object type object in the imop: scheme. The type is a special form of the class type that has exactly one object instantiated when it is first referenced in the runtime. It cannot be further instantiated nor extended. When operated over the network, a client sees only the abstract representation of the object, not its concrete implementation. This design choice is made primarily to ensure interoperability across heterogeneous platforms. This is because a class definition may involve platform dependent execution code (e.g., local database connection) which cannot be easily ported to a different platform. Allowing a class to be extended at a different host (and so on) would make the system implementation too complex to be feasible, not to mention the accompanying security issues.

Although a singleton object cannot be instantiated, one can still ask a server program to create a new object through factory methods (where a factory method is just a regular method that returns an object reference [4, 1]). In the next example we demonstrate this approach. Consider a scenario in which book.com wishes to provide one bulletin board per book for posting book reviews, so it would need to create many IBulletin objects for its clients to access. In Fig. 8 we show a possible implementation for the program at book.com. The site offers an imop: object ReviewManager, which uses a local class type meso:tool.Reviews to create instances of IBulletin objects when requested by its clients via the getReviews() method.

When a client invokes getReviews() with a book’s ISBN as its argument, the manager returns an instance of the local class Reviews that handles the book. For the instance to be operated over the network, it must have a global UTL. For this, the Meso runtime at book.com generates a reference, such as imop:book.com/objI23, and passes it to the client.

It is worth noting that a method that is not declared by a type in the worldwide scope cannot be invoked from the
network. For example, among the methods of the Reviews class in Fig. 8, only the post() and list() methods, which are declared by the imop:org/service.IBulletin interface, are accessible from the network. The public method getISBN() and the constructor are not; they are public only in their local runtime environment. This clean separation helps prevent local implementations from being inadvertently exposed to the world.

Figure 8: Using a local class type to instantiate objects for remote use.

Figure 9: Integrating with Java

We further show a client side program in Fig. 9 to illustrate how Meso is able to integrate with different type schemes. Notice the statement import java:java.lang.System in the code. When encountering the statement System.out.println(), the Meso compiler can resolve, by the import statement, that the method belongs to the Java library to do the printing. We can also see how three different schemes are seamlessly integrated into a Meso program under a unified syntax: imop: (e.g., ReviewManager.getReviews()), java: (e.g., System.out.println()), and meso: (e.g., c.start()).

To summarize, the examples in this section demonstrated how Meso embraces UTLs and object orientation to make developing applications in an Internet-like open environment as easy and type-safe as developing them in a single runtime environment. As such, programmers will be able to build complex Internet-based network applications far beyond what is achievable today.

3. DESIGN AND IMPLEMENTATION

In this section we address some design and implementation issues of Meso. First, we discuss the imop: scheme and the associated protocol for interactions between different programs. Then, we discuss Meso’s multi-scheme type system and the benefits. Finally, we briefly discuss the current implementation of Meso, and address some security and performance concerns.

3.1 imop: type scheme and IMOP protocol

As motivated in Section 1, resources available for local operations and resources available for remote operations may have different concerns. Thus, they are explicitly distinguished by their UTLs: with and without authority addresses. This is different from the conventional wisdom that favors location transparency in a distributed system so that a program can still function correctly when the objects it uses are moved to a different place. Location transparency is difficult to achieve in an Internet-scale open environment due to the lack of an authority to unambiguously resolve name-to-address bindings. As a result, either the same program might behave differently at different places, or a tedious registration and binding process is required to avoid potential conflicts. The overwhelming success of URL vs. URN in naming documents in WWW provides a strong support to our argument. Using URN instead of URL would make the hyperlinks in a document less intuitive, and would not yield WWW the same scale as we see today. We believe that this is also the case for programmatic resources.

When local and remote resources are explicitly distinguished, operationally, it might appear that a protocol for passing untyped messages to operate remote resources would suffice. This is the case in the REST style web services [3] and in DRB [7], where a remote resource is identified by a URL and untyped messages are passed over to invoke it via a default protocol. For strong type checking and a more complex program interaction model beyond client-server (e.g., the examples in Fig. 6-8), we need to define data types for interoperations over the network. The imop: type scheme was designed for this purpose.

Data types in the imop: type scheme are somewhat similar to that in a typical object-oriented language. Fig. 10 shows some of them. Primitive types are all predefined with explicit value ranges and precision. Structure types are composite value types that may hold named fields but do not have methods. An array type represents an array of homogeneous data whose type can be of any type in the scheme. All these types are called value types because their data are always passed by value when sent over the network. For structures and arrays, their entire content is serialized and transferred over.

Among the types, three of them (structure, interface, and object) use global UTLs so that they are accessible over the Internet. Structures and interfaces represent abstract data types that have no concrete value or implementation,
while objects represent concrete instances that offer services to clients. **imop:** objects are always passed by reference—their UTLs. Since there is only one instance of an **imop:** object, its UTL can also be viewed as an URL pointing to the instance. That is, the UTL is both a type name and a reference to the instance.

For type checking, the definition of a global UTL is provided by the authority specified within the UTL. The Internet MetaObject Protocol (IMOP) was designed for this purpose. It also supports the Meso runtime to operate on a remote **imop:** object. In short, the IMOP protocol is a remote method invocation protocol that also supports over-the-wire type reflection.

When the authority of a global UTL receives a reflection request, the authority responds with a **type descriptor** expressing the type’s definition. The descriptor format is subject to the handshaking between the requester and the authority. Nonetheless, to maximize interoperability, IMOP defines a default descriptor format based on JSON that all participating peers must support. Other descriptor formats, such as an XML-based format or a compressed binary one, may also be used so long as they can express the same type definition as the JSON-based descriptor does.

As a final note, recall from Section 2 that creating an **imop:** object for remote access is as easy as creating a local **meso:** object (e.g., Fig. 5 vs. Fig. 3). The main difference is simply in the namespace declaration: whether or not to give it a global UTL. Because the Meso language supports different type schemes (see the next section), two **imop:** objects implemented in different type schemes (e.g., **java:** and **meso:**) are still able to interact. That is, the **imop:** type scheme can be viewed as a bridge for resources in different platforms to interoperate. As such, there is no need to use a global UTL for language specific types (e.g., **java:** bar.service instead of **java:** foo.com/bar.service).

### 3.2 Multi-scheme type system

A type scheme represents a context wherein a type name has a meaningful interpretation. The interpretation may lead to information such as the type’s definition, or the mechanism to obtain its definition and verify its integrity. For example, Java’s type scheme gives the type name **int** its definition, and is also able to determine whether a class type **foo.Bar** is a legal Java class. A type scheme is an indispensable portion of a data type contract, and is often integrated into a language’s type system. Most conventional languages have a **uni-scheme** type system, and thus rarely need to specify their type schemes explicitly.

However, because UTL is a global naming scheme for addressing types in different languages and technologies, the support of UTL in a language implies that the language’s type system must cope with different type schemes; that is, a multi-scheme type system. This is the case in Meso, and it enables several interesting features.

First, Meso is a “hyper-language” that allows resources of different technologies to be expressed and integrated under a unified syntax. Fig. 11 illustrates how Meso can be used to define classes in different type schemes. There are three source files, all written in Meso, but respectively defining three different classes: a **meso** class, a **Java** class, and a **.NET** class. Also noteworthy is that each source file bears a different file extension. In general, the file extension must match the type scheme defined by the file. Each source file must be compiled into a specific binary representation to be used in the corresponding runtime environment. The Meso compiler relies on internal **type scheme handlers** to generate appropriate code for each type scheme. Type scheme handlers can be designed as plug-ins, thereby allowing the schemes supported by the compiler to be extended. Whether each source file in Fig. 11 can be compiled or not depends on the availability of a corresponding type scheme handler in a particular Meso compiler configuration.

![Figure 10: UTLs of IMOP data types](image)

![Figure 11: Defining types in different schemes](image)

![Figure 12: Using types from different schemes](image)

---

2. In the implementation, it is actually the daemon of the runtime that is listening and processing IMOP requests (from a default port).

3. Subject to the condition that the syntax in these technologies is compatible with Meso’s Java-like syntax.
namespace imop:api.org/abc;
public class foo4 { ... }

3.3 Marshalling

Network applications often need to deal with multiple type systems. Traditionally, it is a programmer’s responsibility to perform marshalling (that is, converting data of a foreign type system to a format native to the language, or vice versa). Thanks to Meso’s multi-scheme type system, we are able to shift the responsibility from programmers to the Meso compiler. The compiler is able to identify the places where marshalling is needed: places where data is used in a context that requires a different type scheme to interpret the data. Therefore, it can insert marshalling code appropriately or signal a type error if such marshalling is not possible.

The marshalling code inserted by the compiler is provided by the type scheme plug-in. Ideally, a scheme should provide marshalling code for all possible target schemes. However, given the openness of Meso’s multi-scheme type system (that new type schemes can be added), it is impractical to provide all the marshalling code.

Our solution lies in the meso: scheme, a type scheme that all Meso language implementations must support. Basically, each scheme needs only provide marshalling code for converting to and from data types in the meso: scheme if a conversion is possible. Therefore, the meso: scheme can be used as a bridge to convert data types across two different schemes.

A marshalling algorithm involves several subtle issues. For example, mixing cross-scheme marshalling with same-scheme conversions (e.g., from java:int to java:double) can result in multiple conversion paths. Moreover, creating internal proxy objects is sometimes necessary for marshalling reference type objects across schemes. It also needs a reasonable policy to specify when marshalling should be implicitly applied (i.e., coercion), or explicitly requested (i.e., casting). While these details are beyond the scope of the paper, Fig. 13 illustrates how marshalling can be applied on an expression in Fig. 9.

![Figure 13: Marshalling between type schemes](image1)

3.4 Multi-scheme reference types

A value type like java:int or meso:string belongs to exactly one type scheme. In contrast, a reference type may belong to multiple schemes at the same time. For example, the tool.Reviews class in Fig. 8 is declared in the meso: scheme, but since it implements the interface imop:api.org/service.IBulletin, it belongs to the imop: scheme as well. In general, a meso: scheme class can implement interfaces from different schemes, so an object instantiated from the class can be used in each of these schemes. However, when operated in a scheme different from the scheme the implementation belongs to, a moniker is created to represent the object in that scheme. A moniker acts as a proxy of a meso: scheme object to its caller in a different scheme, and reveals the object’s capability only pertaining to the caller’s scheme. Operations performed on the moniker are redirected back to the actual object it represents.

![Figure 14: Using an object in multiple type schemes](image2)

Fig. 14 illustrates this. The Reviews class has four methods (including the constructor) and is compiled into an internal code used by the Meso runtime. Assume that an object (a) of the class is created. Initially, there is only one internal runtime reference pointing to the object. When the object is returned to a client by the getReviews() method in Fig. 8, a moniker (b) is created and is assigned with a new random URL, say imop:book.com/obj123 as its reference in the worldwide scope. When a remote computer requests (1) a definition of the moniker’s URL, the moniker describes itself as an imop: object that implements the imop: IBulletin interface. Subsequently, a request that invokes the list() method (2) on the imop: object—which is actually the moniker—is redirected to invoke the concrete code (3) on the object (a).

From the network, a remote computer cannot know the existence of the getISBN() method behind the moniker, nor can it invoke the method. That is, the imop: object (b) presented by the moniker looks just like what the interface IBulletin has described and nothing else. This provides a layer of protection when an object is exposed to the network. Only the imop: interface it implements can be seen over the network. All internal functions are concealed from the network.

3.5 Implementation

Meso is designed for building Internet-scale network applications. A challenge for ensuring type-safe interactions in such a scale is that the Internet environment is dynamic. Data types defined today may be unilaterally changed tomorrow. Thus, it is preferred to have the Meso compiler perform type checking as late as possible to reflect the latest state of the execution environment.

To do so, our implementation compiles Meso source code on-the-fly upon loading, and performs type checking against latest type definitions reflected from the network. To reduce network traffic, type definitions may be cached locally using a cache-control header similar to that in HTTP. This implementation makes Meso looks like a dynamic language where programs are executed directly from the source code,
even through the code can mostly be statically analyzed like Java. This approach suits rather well for building Internet applications, since the development process is often highly dynamic. Future implementations may provide statically compiled solutions for applications where performance is a main concern.

For language processing, compiling a Meso program that refers to a remote IMOP type may require network reflections, and thus takes longer compile time than conventional languages do. It is possible to cache interface definitions to reduce the number of network reflections. Nonetheless, since conventional languages do not perform network reflections, programmers have to manually download and manage interface definitions (as in CORBA, Java RMI, and Web Services) or study service documentations (as in REST) to make sure they have correctly referenced a remote type. This actually costs much more than the time one spends in compiling a Meso program.

From the runtime execution point of view, the performance depends on the type of operations invoked. Operating a remote object whose type is previously known is not much different from remote method invocations in conventional technologies. The performance depends mostly on the internal implementation of the runtime system or library. Operating a remote object whose type is not previously known at the compile time, on the other hand, requires a runtime type reflection to introspect whether the object supports the required interface. This extra cost gives us a layer of protection to dynamically and correctly operate a remote object encountered at the runtime, which cannot be achieved by other means.

4. CONCLUDING REMARKS

Prototypes of the language compiler and the runtime environment have been implemented and available for download from our project web site http://gpgrid.org. Our current project priority is to use the paradigm to develop more practical applications. Formal specification of the meso programming language will be left as future work.

5. REFERENCES