RECOM: A Reflective Architecture of Middleware

Yang Sizhong　　Liu Jinde

(MCI, College of Computer Science and Engineering, UEST of China　Chengdu　6110054)

[Abstract] Current middleware is limited in its flexibility and adaptability in face of environment varying and different user requirements. Applying the reflection technology to the middleware design has become a new research field. First, the concepts of reflection and reflective middleware are introduced. This paper further compares the processing of middleware to the reflection mechanism, and then the reflective view of middleware is yielded. Based on this, the idea of employing binding-reification reflective model in middleware design is proposed and the design principles of a reflective middleware prototype named RECOM are deduced.. Whereafter, this paper details the implementation of RECOM about its reflective structure, and configurable reflective layers. Finally, some related work is discussed, and some concluding remarks and topics for further study are presented.

[key words] reflection　　middleware　　reflective view　binding-reification　　reflective layer

1 Reflection

Abstractly, reflection refers to the capability of a system to reason about and act upon itself. More specifically, a reflective system is one that provides a representation of its own behavior, which is amenable to inspection and adaptation, and is causally connected to the underlying behavior it describes. "Causally-connected" means that changes made to the self-representation are immediately mirrored in the underlying system’s actual state and behavior, and vice-versa. It can therefore be said that a reflective system is one that supports an associated causally connected self-representation (CCSR)[6].

Just as objects in conventional OOP are representations of *real world* entities, they can themselves be represented by other objects, usually referred to as *meta-objects*, whose computation is intended to observe and modify their *referents* (the objects they represent)[4]. Meta-computation is often performed by meta-objects by *trapping* the normal computation of their referents; in other words, an action of the referent is trapped by the meta-object, which performs a meta-computation either substituting or encapsulating the referent’s actions. Of course, meta-objects themselves can be represented, i.e., they may be the referents of meta-meta-objects, and so on. A reflective system is thus structured in multiple levels, constituting a *reflective tower*. The objects in the base level are termed *base-objects* and perform computation on the entities of the application domain. The objects in the other levels (termed *meta-levels*) perform computation on the objects residing in the lower levels.

Each reflective computation can be separated into two logical aspects: *computational flow context switching* and *meta-behavior*. A computation starts with the computational flow in the base level; when the base entity begins an action, such action is trapped by the meta-entity and the computational flow raises at meta-level (*shift-up* action). Then the meta-entity completes its meta-computation, and when it allows the base-entity to perform the action, the computational flow goes back to the base level (*shift-down* action).

In all reflective models, an essential concept is that of *reification*. In order to compute on the lower levels’ computation, each level maintains a set of data structures representing (or, in reflection parlance, a reification of) such computation. In other words, reification means making the hidden aspects explicit, so these aspects can be inspected and adapted by applications. Of course, the aspects of the lower levels’ system that are reified depend on the reflective model (e.g., structure, state and behavior, communication). In [7], according to the relationship between meta-entities and base-entities, a first reflective model classification has been pointed out. Ferber remarks the existence of three major reflective approaches: meta-object model, meta-class model, and meta-communication model. In any case, the data structures comprising a reification are causally connected to the aspect(s) of the system being reified; that is, any change to those aspects reflects in the reification, and vice versa. It is a duty of the reflective framework to preserve the causal connection link between the levels: the designers and programmers of meta-objects are insulated from the details of how causal connection is achieved.

2 Reflective View of Middleware

Reflective Middleware is concerned with applying techniques from the field of reflection in order to achieve flexibility and adaptability in middleware platforms. Geoff Coulson offers a concise definition of reflective middleware: reflective middleware is simply a middleware system that provides inspection and adaptation of its behavior through an appropriate CCSR[13]. The reflection does for middleware what it does for any system: it makes it more adaptable to its environment and better able to cope with change.

In distributed applications based on middleware, in order for two objects in different address spaces to interact, the client first gets a proxy for the server. The so-called proxy is a local object representing the remote object, and has the same interface to the remote object which it represents. When the client invokes a method on the remote object through the proxy, the proxy relays the invocation to the target server using the communication mechanism supplied by the middleware. After getting the results, the proxy then returns them to the original client. The client thus completes the procedure of a remote method invocation. The process of associating the client proxy to the service object for communication and the results of the process are referred to as binding.

After examined closely, the middleware processing is found to be similar to the mechanism of reflective computation described above. These two are analogized as follows: when the base entity begins an action (The client and the server start to interact.), such action is trapped by the meta-entity (The client doesn't invoke the server directly, but passes the message to the middleware instead. In this case, the proxy is in charge of trapping the message.), and the computational flow raises at meta-level. Then the meta-entity completes its meta-computation (Marshal the invocation message, and relay it to the server side through network. Then, on the server side, invoke the target server, marshal the results, and pass the results back to the client side. The client side unmarshals the results.), and when it allows the base-entity to perform the action (The proxy returns the results to the original client), the computational flow goes back to the base level.

Thus it is nature to look on the binding established between the client and the server involved in interaction as a composition of several meta-objects with different functions. That is to say, binding is reified as the self-representation of the middleware system, which may be inspected and adapted by applications in order to meet the different needs of quality of service in different scenarioes. Figure 1 illustrates the *reflective view of middleware*.

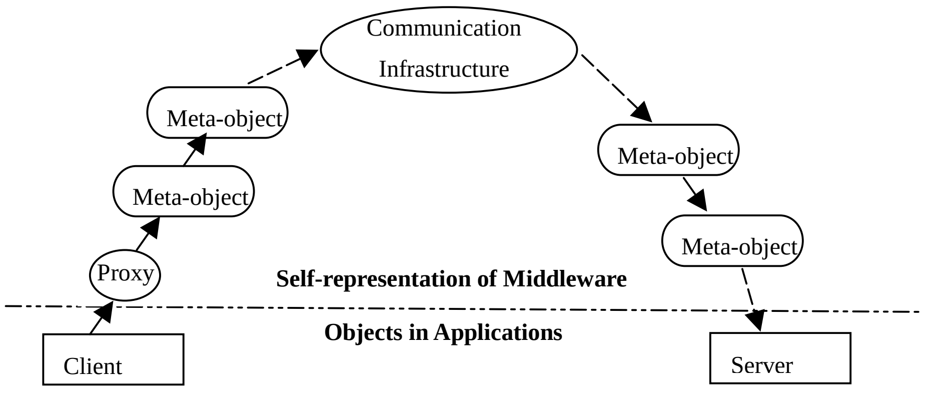

Figure 1 Reflective View of Middleware

This reflective model is distinct from the three models mentioned above. It reifies the binding between distributed objects, and is named *binding-reification reflective model*. The other three reflective models reify the action of a single object, but not the action between objects. Employing the binding-reification reflective model, we have designed a prototype of reflective middleware, RECOM (REflective Configurable Middleware).

3 Design Principles of RECOM

(1) Changes made to the self-representation are immediately mirrored in the underlying system’s actual state and behavior. In this case, the self-representation is the binding between distributed objects. Bindings may be considered to be typeable. The type of a binding involves the invocation semantics (synchronized or asynchronized), choice of protocols, possible quality of service, different resource usage and other binding properties. Thus the application programmer may make choice among bindings of different types, and it is also possible to adapt (or configure) the chosen binding.

(2) Reflective infrastructure should support reflective transparency. On one side, this requires the meta-level (the representation of middleware) to be transparent to the base-entities

(application objects), which is in line with the access transparency in conventional middleware. On the other side, the meta-entities performing meta-computing are demanded to be independent of the specific types of the corresponding base-entities. In other words, the modules comprising the binding are expected to be of general purpose, so they can perform meta-computation on base-entities of different types.

(3) Reification means making the hidden aspects explicit, so these aspects can be inspected and adapted by applications. According to different needs of quality of service, application programmers may change the binding, or probably design their own modules and configures them into the original binding. This also means that the modules comprising the binding should present an unified interface; in addition, there should be a configuration mechanism open to the application programmers, so it is necessary to design an explicit binding protocol.

4 Implementation of RECOM

RECOM is based on Java. Built on the Java platform, we can use the Java Virtual Machine to mask the heterogeneity of hardware and operating systems.

4.1 Reflective Structure

An interface to a remote object is represented in RECOM by a local proxy object. Typically, this is a simple stub object that turns a typed invocation into a generic (but fully typed) form and then passes the request to the top layer of a protocol stack. RECOM stubs are very lightweight, so stub classes are generated on demand within client process, by introspection on the interface definition using Java Core Reflection[15], and link them dynamically, when interface references are resolved.

The layers of a RECOM communication stack can be viewed as a set of meta-objects. Each meta-object in turn performs a meta-computing on the invocation as a data structure before it is ultimately invoked on the destination object, which happens in the ServerCallLayer also using Java Core Reflection. This means that RECOM provides support to represent invocations as generic first-class objects, so it allows meta-objects composing the middleware to examine and modify the parameters and semantics of the invocation.

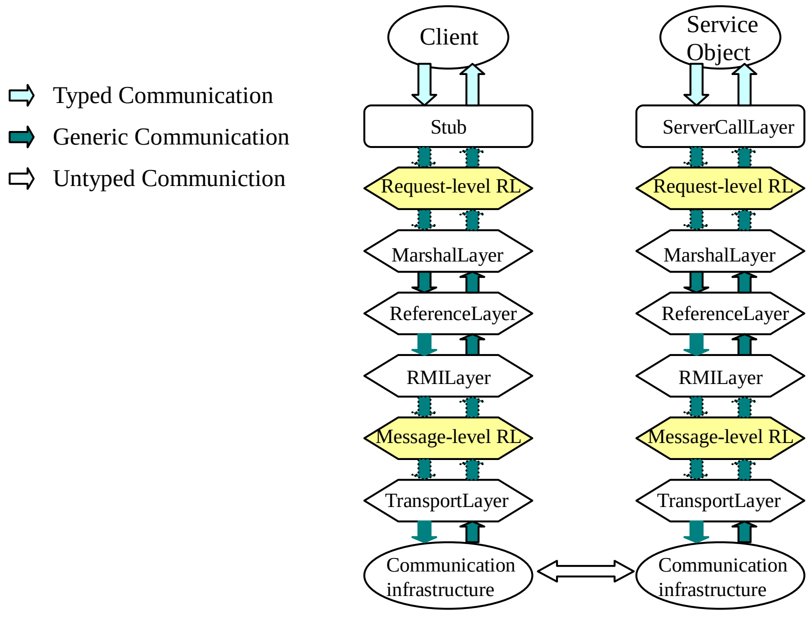

Figure 2 Reflective Layers in the Protocol Stack

Figure 2 illustrates how a communication stack can be assembled as a number of meta-objects that perform meta-computation on an invocation. Meta-objects are fully general Java objects and fully type-safe. A protocol stack is usually materialized in layers, so we also term a meta-object as a *reflective layer*.

At the top of the client side stack, an invocation consists of the reference to the destination interface, the method to be invoked, and the parameters to the method as an array of objects. Interface references are complex objects that can be resolved by a protocol stack to provide a route from the client to the destination interface.

As the call proceeds down the stack, each layer can manipulate the invocation. In figure 2, MarshalLayer is used for marshaling or unmarshaling parameters and return values, ReferenceLayer for differentiating different objects on the same network address, and RMILayer for the implementation of the remote method invocation protocol. By the bottom of the stack, the original reference will have been resolved to an appropriate endpoint or connection identifier, and sufficient information will have been marshaled into a buffer to allow reconstruction of the invocation on the server. On the server side, the reverse process takes place, so that ultimately the destination object, method and parameters are available.

4.2 Configurable Reflective Layer

RECOM supports dynamic configuration of the binding between the specific client and server, such as inserting into the protocol stack some reflective layers of high level features. These reflective layers meet the needs of some non-functional properties required for middleware, and are usually generic, so they can be configured into different protocol suites. They are divided into two classes: one is the request-level reflective layer (request-level RL), and the other is the message-level reflective layer (message-level RL). Request-level reflective layers can be inserted between the stub and the MarshalLayer on the client side, or between the ServerCallLayer and MarshalLayer on the server side. They intercept typed invocation information, and can be used to cache the results on the client side in order to improve performance, or redirect the invocation to an alternative server to gain high availability when the initial server is down. Message-level reflective layers can be inserted between the RMILayer and the TransportLayer on the client side or the server side. What they intercept are marshaled messages in byte arrays, and can be used to encrypt/unencrypt or compress/uncompress messages. While RECOM supplies some optional reflective layers, application programmers can also implement their own reflective layers for special uses according to the open interface of the reflective layer.

In order for application programmers to insert (or delete) reflective layers into (or from) invocation chains through appropriate configuration interface, RECOM provides with an explicit binding protocol. Employing the explicit binding protocol, the client (or the third parties) can obtain not only the reference to the server interface, but also the references to the client-side binding and server-side binding. That is to say, bindings can be operated (or configured) by users, or in reflection parlance, the binding is reified. The binding configuration interface is offered as follows. Two groups of methods are respectively used for configuration of request-level reflective layers and message-level reflective layers.

```
Interface BindMgr
{
   ID    addReqRftLayer(ReqRftLayer rlayer);
   Boolean    removeReqRftLayer(ID id);
   Boolean    removeAllReqRftLayers();

   ID    addMsgRftLayer(MsgRftLayer mlayer);
   Boolean    removeMsgRftLayer(ID id);
   Boolean    removeAllMsgRftLayers();
}
```

It deserves attention that multiple reflective layers can together be configured in the same binding.

5 Related Work and Conclusion

CORBA2.2 specification introduces the interceptor concept[12]. Interceptors allow users to monitor invocation requests and replies, and can be used on both server sides and client sides. But, CORBA interceptors are based on callbacks. Compared with RECOM reflective layers, the main limitation of callbacks is that it is hard (if not impossible) to alter the basic control flow of message processing in the middleware system. On the contrary, after obtaining the information of invocation and doing some processing, RECOM reflective layers will explicitly invoke the next layer in the call chain. So they can catch the exceptions thrown by the middleware, other layers, or server objects, and then deal with these exceptions by themselves, or rethrow them. Since invoking the next layer is explicit, a reflective layer can avoid propagating a request forward simply by not calling its next layer. A common sort of configuration that may use this facility is one that does client-side caching of results.

There is a growing interest in the use of reflection in distributed systems. With respect to middleware, researchers at Illinois have carried out initial experiments in ORB supporting robust and time-critical distribution[9]. The level of configuration is however restricted to marshaling, dispatching, concurrence and other lower level mechanisms. Research at Lancaster University conduct research about an architecture of open distributed multimedia platform[10]. They define four distinct meta-spaces that reify specific aspects of the middleware architecture: encapsulation, composition, environment and resource. In their approach they build an ORB at the base-level, that can configure itself through the deployment of these four meta-spaces. Instead, our work builds middleware at the meta-level, because we treat the binding between the specific client and server as the self-representation of middleware. Every binding monitors and controls the whole process of communication between the specific distributed objects. In other words, this self-representation completely reflects all aspects of the implementation of middleware, so RECOM is more flexible. In the design of the reflective structure of RECOM, we refer to the work of an experimental middleware platform called FlexiNet developed by researchers at APM[8], but FlexiNet only supports static configuration of protocol stacks at compile time. Our work is new regarding to this related work, since we offer an explicit binding protocol used to adapt protocol stacks at runtime.

We are striving for enriching RECOM binding factories, which are used to produce bindings of different types, and reflective layers. It is also desirable to separate the program used for the implementation of reflective layers and the configuration of bindings (which is termed as a meta-program in reflection parlance) from the application program, and associate them when necessary, because meta-programs usually deal with non-functional properties of a system and should be accomplished by the experts of special knowledge. Finally, conditionality among different reflective layers deserves study.

References

1 A. Campbell, G. Coulson, and M. Kounavis. Managing Complexity: Middleware Explained. *IEEE IT Pro*, September/October, 1999

2 M. Roman, F. Kon and R. Cambell. Reflective Middleware: From Your Desk to Your Hand. Technical Report UIUCDCS-R-2000-2195. UIUC Software Research Group, 2000

3 W. Cazzola, S. Chiba, and T. Ledoux. Reflection and Meta-Level Architectures: State of the Art, and Future Trends. In Jacques Malenfant, Sabine Moisan, and Ana Moreira, editors, *ECOOP'2000 Workshop Reader*, Lecture Notes in Computer Science 1964, pages 1-15. Springer-Verlag, 2000.

4 W. Cazzola. Evaluation of Object-Oriented Reflective Model. In Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98). Brussels, Belgium, Jul. 1998.

5 B. C. Smith. Reflection and Semantics in a Procedural Language. Technical Report 272, MIT Laboratory of Computer Science, 1982.

6 P. Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87*), volume 22 of *Sigplan Notice*s, pages 147–156, Orlando, Florida, USA, October 1987. ACM.

7 J. Ferber. Computational Reflection in Class Based Object Oriented Languages. In *Proceedings of 4th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89*), volume 24 of *Sigplan Notice*s, pages 317–326. ACM, October 1989.

8 R. Hayton, ANSA Team. FlexiNet Architecture. Architecture Report, Citrix Systems (Cambridge) Limited, Feb. 1999.

9 F. Kon, R. Campbell, and M. Roman. Design and Implementation of Runtime Reflection in Communication Middleware: the DynamicTAO Case. In *proceeding of ICDCS'99 Workshop on Middleware*, 1999.

10 G. Blair, F. Costa, G. Coulson, etc. The Design of a Resource-Aware Reflective Middleware Architecture. In Pierre Cointe, editor, *Proceedings of the 2nd International Conference on Reflection'9*9, LNCS 1616, pages 115–134, Saint-Malo, France, July 1999.Springer-Verlag.

11 G. Kiczales. Beyond the black box: Open Implementation. IEEE Software, 13(1), 1996.,

12 Object Management Group. CORBA2.3.1/IIOP Specification. OMG, Oct. 1999

13 Geoff Coulson, Refective Middleware, Available at http://computer.org/dsonline/middleware/RM.htm, 2001

14 RM'2000. Workshop on Reflective Middleware of Middleware'2000. Available at http://www.comp.lancs.au.uk/computing/RM2000. April, 2000.

15 Sun Microsystems. Java Core Reflection. Available at http://java.sun.com/products/jdk/1.2/docs/guide/reflection/.