Supporting Multiple Transport Protocols in a CORBA System

Tatsuo Nakajima
Department of Information and Computer Science
School of Science and Engineering
Waseda University
3-4-1 Ookubo Shinjuku Tokyo 169-8555, JAPAN

Abstract

This paper reports supporting dynamic transport selection in omniORB2, which is a CORBA 2.0 compliant CORBA system that has been developed at AT&T Laboratories, Cambridge. We describe our design and implementation of supporting dynamic transport protocol selection in our system, and some initial experiments with the system. In our approach, IIOP can be selected for ensuring interoperability between applications. Thus, applications adopting our system can communicate with those adopting other CORBA systems, which run on the Internet environment. On the other hand, an application can select more suitable transport protocols for improving its performance or ensuring its real-time constraints if a server also supports the same transport protocol according to the characteristic of the application.

1 Introduction

CORBA[11] has became a popular middleware for building large and complex distributed applications. CORBA enables programmers to build applications without taking into account low level details of underlying communication infrastructures. Therefore, the development speed of distributed applications is dramatically increased, and application programs become more interoperable and portable. CORBA is now adopted in a lot of research prototypes and commercial products of distributed applications such as network management, telecommunication, and on-line transaction processing.

However, most current implementations of CORBA provide only IIOP(Internet Inter-ORB Protocol) that adopts TCP as an underlying transport protocol for ensuring interoperability among different CORBA products. However, since the CORBA specification is independent of underlying communication infrastructures, more appropriate transport protocols can be chosen for respective applications if they are available. If a CORBA system is able to support multiple transport protocols simultaneously, and an application program can select the most appropriate one among them, advanced features of respective transport protocols such as bandwidth reservation can be available from the program directly.

A large number of computers may be connected via low bandwidth networks in future computing environments. Also, various consumer devices such as TV and Hi-Fi audio systems will be connected via high bandwidth networks. In such environments, the most adequate transport protocol might be changed according to available communication infrastructures and the characteristics of applications.

Future computing environments make is possible that a lot of context information can be retrieved from various sensors embedded in our surrounding environments[8]. An application program should be migrated among various types of computers according to its user's location information for achieving better performance[18, 19]. Also, an application might be controlled by any devices near its user. Thus, the application needs to support various types of input/output devices that manage the interaction with the user. For example, a display device near from a user will display a control panel and the panel transfers events from the user to his/her application[25]. Also, in future network environments, a part of an application can be carried with a packet, and a router and a switch will execute it for improving an application's performance[1].

The above visions require ubiquitous middleware like CORBA for building distributed applications that can be executed on various types of platforms. In order to realize the goal, CORBA should support multiple transport protocols simultaneously, and these protocols should be dynamically selected since various types of communication infrastructures require different transport protocols for making the benefits of the infrastructures maximum.

This paper reports the support of multiple transport protocols and the dynamic selection of the protocols implemented in omniORB2, which is a CORBA 2.0 compliant CORBA system that has been developed at AT&T Laboratories, Cambridge. Our system has extended

[1] † The research described in the paper was done while the author was visiting at AT&T Laboratories, Cambridge, and Laboratory for Communication Engineering of University of Cambridge.

the standard interface for controlling transport protocols and provides mechanisms for the dynamic selection of the protocols. The current implementation supports three transport protocols, *IIOP*, *IIOP over SSL*, and *GIOP over ATM*, where *IIOP over SSL* is to transmit GIOP(General Inter-ORB Protocol) messages over SSL, and *GIOP over ATM* is to transmit GIOP messages over a simple reliable transport protocol on native ATM.

2 Why IIOP is not Enough ?

In this section, we consider four situations where IIOP is not suitable for building various types of distributed applications. The first situation occurs when using a connection oriented network protocol below the IP layer. For example, IP over ATM emulates the IP datagram service by creating a new ATM connection automatically when an IP packet is transmitted. The approach makes packet transmission latency unpredictable since the connection setup of ATM is completely hidden from the IP layer. Also, the approach does not allow the traffic management features of ATM to be used by applications. A lot of future network systems such as IEEE 1394's isochronous mode which is suitable for multimedia communication will support bandwidth management for transmitting timing critical data. Therefore, IIOP is not enough to support advanced real-time network systems.

The second situation occurs when using a network protocol that does not ensure the assumption for processing the TCP protocol. For example, *Piconet*[2] is a low power and low bandwidth ad-hoc network, which can be embedded in a lot of small devices in our surrounding world such as PDAs, cellular phones and microwave ovens. In this environment, IIOP may not be adequate for controlling such devices that are connected via Piconet, since TCP is not an appropriate protocol for reducing energy consumption.

The third situation occurs when requiring to support secure transport protocols. Practical business applications need to support security for ensuring to protect secrets of users. However, traditional approaches require to modify existing applications for supporting security. The support of secure transport protocols should take into account the reusability of existing applications, but IIOP provides no mechanism for supporting security.

The last situation occurs when using a network system that does not provide TCP/IP. For example, clustered computers require ultra high performance networks since communication performance is very important for increasing application's performance. In such networks, a network interface device is mapped into each application's address space, and operating systems are bypassed when transmitting and receiving data[16, 21]. In this case, a lightweight transport protocol needs to be implemented at user-level. This means that the networks do not allow the overhead of the TCP protocol. Thus, TCP is also inappropriate as a transport protocol for clustered computers.

3 Design Issues

In this section, we describe three issues for supporting dynamic transport protocol selection. The first issue is how to specify a transport selection policy. The second issue is the granularity of the transport selection policy. The last issue is how an error is handled when specified transport protocol cannot be available.

3.1 Transport Selection Policy

The first issue is how to select a suitable transport protocol among currently available protocols. We discuss two choices for the issue. In the first choice, a system provides several different policies for selecting a transport protocol. Programmers select suitable policies among policies provided by the system for their applications. In the second choice, a system allows programmers to implement appropriate transport selection policies for their applications.

The first choice makes a system more stable, since there is no possibility for adding incorrect policies by a programmer. However, the selection policy will depend on the characteristics of an application and a communication platform. For example, a programmer wants to use a special tool monitoring traffic on networks to select the most suitable protocol, but the tool may be available on only a special platform. The second choice enables programmers to create selection policies for their applications, and the applications can install the policies for respective objects. Our system adopts the second choice for achieving higher flexibility.

3.2 Granularity of Transport Selection Policy

The next issue is the granularity for specifying explicit binding. In this paper, we discuss two choices. The first choice is to specify a QOS parameter for a client. This means that the specified QOS is applied for all objects that are invoked from the client. The choice makes the implementation is very simple and the effect of introducing explicit binding primitives minimum. However, the requirements of an application may be changed for accessing respective objects. The second choice is to allow a programmer to specify a QOS parameter for each object. Thus, our system chooses the second choice, since we believe that flexibility is more important for supporting multiple transport protocols. We also provide the first choice by allows a user to specify an option when a program is started.

3.3 Handling Connection Setup Error

The issue described in the section is an error handling when the currently selected transport protocol cannot be used. The situation occurs when both a client and a server support the protocol, but they cannot communication with each other by using the protocol. We can consider two policies for solving the issue. The first solution is to automatically change the currently selected transport protocol to the protocol that can be actually available now. In the second solution, the run-time throws an exception when the currently selected transport protocol is not available. Both solutions are desirable according to an application's requirements. Our system provides two classes for transport selection policies. The first class implements the first policy, and the second class implements the second policy. When a programmer defines a new class implementing his own transport protocol selection policy, the class should inherit one of the two classes.

4 Dynamic Transport Protocol Selection in omniORB2

In this section, we describe a brief overview of CORBA and omniORB2, and present how our extended interface is used by showing some fragments of programs.

4.1 CORBA and omniORB2

CORBA provides standard programming interface for constructing distributed applications. A client accesses an object in a server through a proxy object in the client's address space. A server has object adaptors that deliver requests from a client to a target object. The current CORBA specification[11] defines a basic architecture, interface definition language(IDL), application programming interface, and protocols for ensuring interoperability among different CORBA products. The specification also contains language mapping for several major programming languages. Thus, applications written by different programming languages can be communicated with each other.

Although CORBA is carefully designed for satisfying various requirements of a wide range of distributed applications, the current specification has some limitations for supporting all types of distributed applications. In fact, there are a lot of ongoing research projects that attempt to remove the limitations of the current CORBA specification. DIMMA[6], ReTINA[5], and GOPI[4] have extended CORBA for supporting to build continuous media applications. The IDLs of these systems are extended to support stream interface for delivering audio and video streams. TAO[23] has been enhanced for supporting distributed real-time computing, which focuses on avoiding priority inversion in the CORBA run-time. Also, TAO extends the CORBA IDL for supporting QOS parameters such as period and worst case execution time, which are required to guarantee all timing constraints of applications. Electra[17] enables programmers to build fault tolerant applications with CORBA. Reflective CORBA[3] provides customizable CORBA run-time, which allows programmers to replace some parts of the run-time according to the characteristics of their applications.

OmniORB2[1] , which provides CORBA 2.0 C++ mapping, especially focuses on achieving high performance, since a lot of distributed applications do not tolerate large overhead caused by most existing CORBA systems. The high performance CORBA makes it possible that a lot of performance critical applications can adopt CORBA. This makes applications more interoperable and reusable.

Currently, omniORB2 has been used in several projects at AT&T Laboratories, Cambridge. For example, it is used as a communication infrastructure for controlling the Virtual Network Computer(VNC)[22] by using the Active Badge System. Also, it is used for the SPIRIT middleware[25] for building fine-grained location-aware applications that use Active Bat[24]. Also, some companies and universities have been adopting omniORB2 for building their research prototypes and products, and some research groups have implemented the COS event service[14] and the COS transaction service[15] on omniORB2.

4.2 Dynamic Transport Selection in Client

In this section, we describe how a customized transport selection policy is created. In the current implementation, the default transport selection policy does not change the current transport protocol. Also, when the current transport protocol is rejected by a server, the GIOP client engine throws an exception to a client program. In this case, it may change the transport protocol by calling the *rebind* method explicitly or terminate itself. If the policy is not suitable for an application, it needs to define its own transport selection policy.

The new policy can be installed by calling the *setTransportSelectionPolicy* method which is defined in the *omniQOS* class, and the policy should be a subclass of the the *SelectionPolicy* class. Our system allows a program to select different transport selection policy for respective object references.

The following is a sample client program that specifies an application specific transport selection policy. The program creates an instance of the *DynamicSelectionPolicy* class, which is described later, and installs the policy for invoking an *Echo* object(Line 8). Before invoking the *echoString* method in the *Echo* object, the

[1] OmniORB2 is publicly available under GNU public licenses. More information is available from http://www.uk.research.att.com/omniORB/.

program calls a policy specific method of the *DynamicSelectionPolicy* class for specifying the preference of the application(Line 10).

```
1  void
2  hello(CORBA::Object_ptr obj, hint_t hint)
3  {
4    DynamicSelectionPolicy policy;
5
6    Echo_var e = Echo::_narrow(obj);
7
8    omniQOS::setTransportSelectionPolicy(e, &policy);
9
10   policy.setTransportHint(hint);
11
12   .....
13   dest = e->echoString(src);
14   .....
15 }
```

The *omniQOS* class which is provided by our system defines several methods for managing transport protocols. When a transport selection policy object is not installed, the current transport protocol can be explicitly changed by invoking the *rebind* method. The method returns a new object reference which a specified transport protocol is bound to. If a program uses a returned object reference for invoking a method in a target object, the specified transport protocol transmits requests and replies to the target object. Also, the *omniQOS* class provides the *openConnection* method which opens a new connection explicitly. In [20], we describe how to use these primitives in details.

4.3 Transport Selection Policy

The programmer needs to define two methods for defining a new transport selection policy when implementing a subclass of the *SelectionPolicy* class. The first method, *isStaticPolicy* determines the behavior when the currently selected transport protocol is not available. When the method returns "true" and the current transport protocol is not available, the run-time throws an exception. On the other hand, if the method returns "false" and the current transport protocol is unavailable, the run-time changes the current transport protocol to IIOP automatically, and call the *changeTransport* method for notifying the change to an application. The second method, *runSelectionPolicy* is called whenever a remote object is invoked. The method checks whether the current transport protocol is suitable or not, and changes the protocol if there is a more appropriate protocol.

A selected transport protocol determined by the *runSelectionPolicy* method may be changed according to QOS parameters which are provided by policy specific methods defined in a transport protocol selection policy object. These methods allow us to specify QOS parameters that enable an application to satisfy the requirements for invoking a method of a remote object.

In the current implementation, we provide three transport selection policy classes. The first class is *StaticSelectionPolicy*, which is a default policy in the current implementation. When the policy is selected, the runtime throws an exception when the current transport protocol is not available. Also, the policy have no policy specific method, and *runSelectionPolicy* does not change the current transport protocol. The second class is *ExplicitSelectionPolicy*. The policy automatically changes the current transport protocol to IIOP when the current transport protocol is not available. The policy provides the *changeTransport* method that allows a programmer to change the current transport protocol explicitly without using the *rebind* method. The last class is *DynamicSelectionPolicy*. When the policy is adopted, the run-time changes the current transport protocol to IIOP when the current transport protocol is not available. The policy provides a method for specifying only a simple QOS parameter, but a more complex policy which allows us to specify a more complete QOS parameter and mechanism can be implemented by inheriting these classes.

The following class definition shows the signature of the *DynamicSelectionPolicy* class. The class allows the run-time to change the current transport protocol when a server rejects the protocol or the protocol is not available at a client since the *isStaticPolicy* method returns "false". Also, the class provides the *setTransportHint* method for notifying an application's preference to the policy object. The method has one argument whose value is either *HighThroughput* or *LowThroughput*. If *HighThroughput* is specified, a system selects a transport protocol that can transmit a large amount of data. On the other hand, if *LowThroughput* is specified, a transport protocol that can transmit a small request with the fastest latency will be selected.

```
1  typedef enum { HighThroughput,
2                       LowThroughput } hint_t;
3
4  class DynamicSelectionPolicy
5                : public SelectionPolicy {
6
7  public:
8    virtual void runSelectionPolicy();
9    virtual CORBA::Boolean isStaticPolicy()
10       {
11            return false;
12       }
13
14   void setTransportHint(hint_t hint);
15 };
```

The following program shows the body of the *DynamicSelectionPolicy* class. The code is simplified for showing only essential points. In the implementation, if *HighThroughput* is selected, the current transport protocol is changed to GIOP over ATM(line 9). Also, *LowThroughput* is chosen, the program checks whether there is a currently opened ATM connection for transmitting a re-

quest(line 17). If it is available, GIOP over ATM is selected(line 19). Otherwise, IIOP is selected(line 21).

```
1  void
2  DynamicSelectionPolicy::runSelectionPolicy()
3  {
4    ConnectionState  *cs;
5
6    switch(transport_hint) {
7    case HighThroughput:
8      if(getConnectionType() != TRANS_ATM) {
9        changeCurrentTransport(TRANS_ATM);
10     }
11
12     break;
13
14   case LowThroughput:
15     cs = getConnectionState(TRANS_ATM);
16
17     if(getConnectionControl()
18                        ->getConnectionStatus()) {
19       changeCurrentTransport(TRANS_ATM);
20     } else {
21       changeCurrentTransport(TRANS_IIOP);
22     }
23
24     deleteConnectionState(cs);
25
26     break;
27   }
28 }
29
30 void
31 DynamicSelectionPolicy::setTransportHint
32                                (hint_t hint)
33 {
34   transport_hint = hint;
35 }
```

4.4 Supporting Multiple Transport Protocol in Server

The following server program supports both *IIOP* and *GIOP over ATM* for receiving invocations to the *Echo* object. In line 4, ORB is initialized, and BOA(Basic Object Adaptor) is initialized in line 6.

The *atmContext* class is initialized in line 9. The class is a subclass of the *TransportContext* class, which is used for initializing the ATM protocol module. After the initialization, the server can accept ATM connections. Since the *atmContext* class is the singleton class, the *getContext* method in the *atmContext* class returns a unique instance of the class. In this example, the server receives both IIOP and GIOP over ATM. In line 13-14, an instance of the *MyConnectionInterceptorFactoryForATM* class is created and installed. The class is a subclass of the *ConnectionInterceptorFactory* class, which contains the *create* method for creating a new connection interceptor object. The method creates an instance of the *MyConnectionInterceptorForATM* class, which defines the default behavior for validating the setup of ATM connections whenever the program accepts a new ATM connection. The *MyConnectionInterceptorForATM* class is a subclass of the *ConnectionInterceptor* class that defines methods invoked before and after opening a transport connection.

In line 16, an instance of the *Echo_i* class is created, then it is installed in the BOA in line 17. In line 19, the server enters into an event loop that waits for the acceptance of the setup requests of both TCP and ATM connections.

```
1  int
2  main(int argc, char **argv)
3  {
4     CORBA::ORB_ptr orb = CORBA::ORB_init(argc,
5                                  argv, "omniORB2");
6     CORBA::BOA_ptr boa = orb->BOA_init(argc,
7                                  argv, "omniORB2_BOA");
8
9     atmContext::initContext(argc, argv);
10    atmContext *atm_context
11             = atmContext::getContext();
12
13    atm_context->setConnectionInterceptorFactory(
14            new MyConnectionInterceptorFactoryForATM());
15
16    Echo_i *myobj = new Echo_i();
17    myobj->_obj_is_ready(boa);
18    .....
19    boa->impl_is_ready();
20    .....
21 }
```

The following program shows the definition of the *afterAccept* method contained in the *MyConnectionInterceptorForATM* class. The method is invoked after a new ATM connection setup request is received in the server program. If the method returns "Reject", a client program will catch an exception.

In line 7, the current bandwidth of the newly accepted connection is retrieved. If the bandwidth is greater than 1000(1Mbps) or equals 0(This means that the client opens a ATM connection by the UBR mode.), the server program notifies to the run-time for reducing the bandwidth to 1Mbps. Otherwise, the ATM connection setup request is accepted and method invocations will be received. The *ATMConnectionState* class provides several methods for controlling ATM networks. In our system, each transport protocol provides a class that contains several methods for controlling the protocol. In [20], we describe how to use the class in details.

```
1  userRequest_t
2  MyConnectionInterceptorForATM
3           ::afterAccept(ConnectionState& cs)
4  {
5     ATMConnectionState& atm_cs
6                   = (ATMConnectionState&)cs;
7     CORBA::ULong bandwidth = atm_cs.getBandwidth();
8
9     if((bandwidth > 1000) || (bandwidth == 0)) {
```

```
10      cout << ``The current bandwidth
11                  is reduced to 1000 kbps'' << endl;
12      atm_cs.setConnectionInfo(ATM_REDUCE_REQUEST,
13                  1000);
14    } else {
15      cout << ``The current bandwidth
16                  can be acceptable'' << endl;
17    }
18
19    return Accept;
20 }
```

5 Supporting Multiple Transport Protocols

In this section, we describe how multiple transport protocols are supported in our system. First, we present how a server exports information for multiple transport protocols. Then, we show how a client program checks the availability of a specified transport protocol between the client program and a server program. Lastly, we describe how a server delivers a notification to a client when the server knows that the client can use a more suitable transport protocol.

5.1 Exporting Transport Information

The current implementation encodes information about *IIOP over SSL* and *GIOP over ATM* in the tagged components of the IIOP profile in an IOR(Interoperable Object Reference). We adopt the *TAG_SSL_SEC_TRANS* tag defined by OMG for *IIOP over SSL*[12]. For *GIOP over ATM*, we define a new tagged component that contains an ATM address, a port number, and the expected bandwidth of a server. The expected bandwidth information is used for negotiating bandwidth between a client and a server. The approach makes the implementation easy, but it requires that IIOP should be always implemented on all platforms. In future computing environments, small computers such as consumer devices and embedded computers may not implement TCP/IP for making the system size small. Thus, the future implementation will allow a server to select IOR encoding strategies of each transport protocol whether it is encoded as a separated profile or in the IIOP profile.

5.2 Transport Selection at Client Side

In our scheme, a client program can know transport protocols supported by a server program. However, the scheme does not ensure that the client program can communicate with the server programs via any protocols encoded in an IOR. The problem can be solved by using a mechanism for knowing the topologies of networks between the client and the server, but such a mechanism cannot be available for all network infrastructures, and the mechanism is significantly difficult to be implemented. Our system assumes that a network topology between a client and a server is unknown, and we adopt a simple policy that a client removes the currently selected transport protocol among available protocols when its connection setup causes an error.

5.3 Transport Selection at Server Side

The server may reject to use the current transport protocol that a client issues a method invocation request. For example, a server must control that a client does not consume too much bandwidth. Also, a server should deliver a notification to a client when the server knows that the client can use a more suitable transport protocol. For example, a secure protocol may not be required when both a client and a server reside in a trusted network domain. Our system uses a *location forward* message for notifying that a more suitable transport protocol can be available to a client. For example, if a server decides that a client should not use GIOP over ATM, the server returns an IOR that does not contain information about ATM. However, the mechanism is very heavy for checking every request. Thus, we decide to execute the procedure only when the *Locate Request* message is received, which is used for checking whether an object exists in a server. Since the request is transmitted only before the first method invocation to the object, the approach does not degrade the performance of an application.

6 Performance Evaluation and Discussion

The section presents the performance evaluation of the current implementation. Also, we describe several experiments with our current prototype.

6.1 Performance Evaluation

The result as shown in Table 1 presents the round-trip latency of a null string echo. The evaluation was measured on two Sun 4 SS4 machines on which Solaris 5.5.1 is running. The two machines are connected via 155 Mbps ATM network. The machines have the Fore SBA-200 SBus ATM card. The implementation of GIOP over ATM uses the XTI interface for the Fore ATM card in order to access ATM directly. To measure the performance of IIOP, Classical IP over ATM is used to provide TCP/IP connectivity. The result shows that the connection setup overhead of GIOP over ATM. Also, the difference between *GIOP over ATM(with transport)* and *GIOP over ATM(without transport)* shows the overhead of our simple ATM transport protocol that provides reliable packet transmission. The result shows that connection setup takes a long time. Thus, an application that requires predictable latency should use the *openConnection* and *closeConnection* method explicitly. The

approach is especially useful to support distributed real-time systems. Traditional CORBA systems that do not provide the explicit binding model and explicit connection management cannot support real-time applications since a programmer cannot control the CORBA run-time for predictable method invocations.

6.2 Discussion

In this section, we describe two experiences with our current prototype. The first experience is about automatic transport selection provided by a transport selection policy. The second experience is about dependencies among protocols when dynamic selection is adopted.

6.2.1 Automatic Transport Selection

Our system provides two ways for supporting explicit binding. The first way is to use the *rebind* method described in [20], and the second way to use policy specific methods provided by transport selection policy objects. The first way specifies the actual transport protocol name as an argument, and the returned object reference is bound to the specified protocol. The reference can be passed as an argument in a method call since the approach does not violate the semantics of CORBA. Also, the object reference is strictly bound to a transport protocol. Thus, the semantics of the program is very clean. On the other hand, the second way is more flexible. A program can choose any abstraction levels for specifying QOS parameters by creating a new transport selection policy class. However, the approach may change the current transport protocol implicitly according to a transport selection policy. Therefore, the method call latency becomes unpredictable due to the implicit changes of transport protocols. Also, respective transport protocols provide different semantics for reliability and error handling. Therefore, a programmer may confuse the changes of the semantics if the changes occur without notifying to an application. Moreover, a program needs to prepare connection interceptor objects for all transport protocols that are used in a transport selection policy object. This means that our system assumes that a programmer needs to know protocols that may be chosen by a transport section policy object.

If a transport selection policy object automatically selects a suitable transport protocol, it is difficult to control the setup and the shutdown of connections by a program in an explicit way since a program does not know which protocol is chosen before invoking a selection policy object. For example, the connection setup latency of ATM networks is very long. Thus, a connection should be opened before invoking a target object if an application has real-time constraints. We may need to introduce a new primitive to open a connection before invoking methods according to a transport selection policy.

6.2.2 Dependencies among Transport Protocols

The original architecture of omniORB2 does not consider to support multiple transport protocols at the same time. The implementation allows us to use a different protocol by creating the instances of the rope and strand object for the protocol at the initialization time. The approach successfully supports multiple transport protocol easily. Also, the approach does not make the size of the run-time big, because only the rope and strand class that are currently used are linked in the run-time. Thus, when supporting multiple transport protocols, a necessary rope and strand object should be linked dynamically at the first use. However, dynamic protocol selection creates dependencies among protocols. The current implementation of the GIOP engine may call the methods of rope and strand objects for respective protocols in order to determine the most suitable protocol. This means that all codes for transport protocols must be linked for dynamic selection. In future, we need to consider how to make the dependencies minimum and the size of the run-time small.

In a client program, different transport protocols need to install respective connection interceptors, or a connection interceptor that can handle all protocols that may be selected by a transport selection policy object is install for respective transport protocols. The former approach needs to define a new connection interceptor whenever a transport selection policy object support a new transport protocol. Also, the *changeTransport* method defined in a connection interceptor needs to install an appropriate connection interceptor object. If a programmer forgets to install the connection interceptor, a connection setup request may not be validated correctly. The later approach needs to modify the connection interceptor for handling a new protocol.

The problem can be solved by abstracting information about each transport protocol, but the approach requires to map abstract information to concrete information for each protocol. The solution is difficult to be implemented, and it is not easy to implement efficiently. The current solution is to install respective connection interceptors for all transport protocols that are chosen by transport selection policy objects. Therefore, the current solutions requires that a programmer needs to know all protocols that may be chosen by transport selection policy objects, and a program should install an appropriate connection interceptor when the currently used transport protocol is changed.

7 Conclusion

In this paper, we described the support of dynamic transport selection in a CORBA system. Our approach provides a transport selection policy object for each proxy object. Our system is very flexible, since a programmer can select an appropriate abstraction level for

	w/o Connection Setup	w Connection Setup
IIOP	2.5 ms	13.27 ms
GIOP over ATM(with transport)	2.2 ms	88.80 ms
GIOP over ATM(without transport)	2.1 ms	88.80 ms

Table 1: Round Trip Time of Null String Echo

respective transport selection policy.

Acknowledgement

I would like to thank Sai-Lai Lo, who is a great omniORB2 developer. He always gives me a lot of suggestions and helps the implementation of our system. Andy Harter, Steve Pope, Pete Steggles, and Paul Webster's suggestions are very useful for the work. I especially would like to thank Sai-Lai Lo and Steve Pope for reading the draft of the paper and giving me a lot of useful comments. Finally, I am grateful to Andy Hopper for giving me a chance to stay at AT&T Laboratories, Cambridge for 10 months.

References

[1] D.S. Alexander, W.A. Arbaugh, M.W. Hicks, P. Kakkar, A.D. Keromytis, J.T. Moore, C.A. Gunter, S.M. Nettles, and J.M. Smith, "The SwitchWare Active Network Architecture", IEEE Network, Vol.12, No.3, 1998.

[2] F. Bennett, D. Clarke, J.B. Evans, A. Hopper, A. Jones and D. Leask, "Piconet - Embedded Mobile Networking", IEEE Personal Communications, Vol. 4, No. 5, 1997,

[3] G. Blair, G. Coulson, P. Robin and M. Papathomas, "An Architecture for Next Generation Middleware", In the Proceedings of Middleware'98, 1998.

[4] G. Coulson and M. Clarke, "A Distributed Object Platform Infrastructure for Multimedia Applications", Computer Communications, Vol.21, No.9, 1998.

[5] F. Dang Tran, V. Perebaskine, J. Stefani, "Binding and Streams: the ReTINA Approach", In Proceedings of the TINA'96 Conference, 1996.

[6] D.I. Donaldson, M.C. Faupel, R.J. Hayton, A.J. Herbert, N.J. Howarth, A. Kramer, I.A. MacMillan, D.J. Otway, S.W. Waterhouse, "DIMMA - A Multi-Media ORB", In Proceeding of the Middleware'98, 1998.

[7] S. Frolond, and J. Koistinen, "Quality of Service Specification in Distributed Object Systems", Distributed Systems Engineering Journal, Vol. 5, No. 4, 1998.

[8] A. Jones, "Sentient Computing", A Seminar at Computer Lab, University of Cambridge, 1998.

[9] J.P. Loyall, D.E. Bakkea, R.E. Schang, J.A. Zinky, D.A. Karn, R.Vanegas, and K.R. Anderson, "QOS Aspect Languages and Their Runtime Integration", In Proceedings of the Fourth Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers, 1998.

[10] S. Lo, S. Pope, "The Implementation of a High Performance ORB over Multiple Network Transports", Middleware'98, 1998.

[11] Common Object Request Broker Architecture and Specification, Revision 2.2. Available electronically via http://www.omg.org/.

[12] Secure Socket Layer/CORBA Security, OMG Document, orbos/97-02-04, 1997.

[13] "omniORB2 Home Page", http://www.uk.research.att.com/omniORB/.

[14] "COS Event Service for omniORB2", http://www.uk.research.att.com/omniORB/contribapp.html.

[15] "OTS/JTS Arjuna Home Page", http://arjuna.ncl.ac.uk/OTSArjuna/.

[16] L. Li, A. Forin, G.Hunt, and Y.-M. Wang, "High-Performance Distributed Objects over a System Area Network", MSR-TR-98-68, Microsoft Research, 1998.

[17] S. Maffeis, "Electra - Making Distributed Programs Object-Oriented", In the Proceedings of USENIX Symposium on Experiences with Distributed and Multiprocessor Systems, IV, 1993.

[18] T. Nakajima, and A. Hokimoto, "Adaptive Continuous Media Applications in Mobile Computing Environments", In Proceedings of the international Conference of Multimedia Computing and Systems, 1997.

[19] T. Nakajima, and H. Aizu, "System Supports for Environment-Aware Migratory Continuous Media Applications", In Proceedings of the Sixth International Conference on Distributed Multimedia Systems, 1999.

[20] T. Nakajima, "Practical Explicit Binding Interface for Supporting Multiple Transport Protocols in a CORBA System", AT&T Laboratories, Cambridge, Technical Report, 1999.

[21] S. Pope, S. J. Hodges, G. E. Mapp, D. E. Roberts, A.Hopper, "Enhancing Distributed Systems with Low-Latency Networking" , In Proceeding of International Conference on Parallel and Distributed Computing and Networks (PDCN98), 1998.

[22] T. Richardson, Q. Stafford-Fraser, K.R. Wood, A. Hopper, "Virtual Network Computing", IEEE Internet Computing, Vol. 2, No. 1, 1998.

[23] D.C. Schmidt, L. Levin and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers", Computer Communications, Vol.21, Apr, 1998.

[24] A. Ward, A. Jones, A. Hopper, "A New Location Technique for the Active Office", IEEE Personal Communications, Vol. 4, No. 5, 1997.

[25] P. Webster, P. Steggles and A. Harter, "The Implementation of a Distributed Framework to support 'Follow Me' Applications" In the Proceeding of PDPTA'98, 1998.