COMERA: COM Extensible Remoting Architecture

Yi-Min Wang
Microsoft Research
Woei-Jyh Lee
New York University

COMERA: COM Extensible Remoting Architecture

Yi-Min Wang
Microsoft Research
Woei-Jyh Lee
New York University

Abstract

In a distributed object system, remoting architecture refers to the infrastructure that allows client programs to invoke methods on remote server objects in a transparent way. In this paper, we study the strength and limitations of current remoting architecture of COM (Component Object Model), and propose a new architecture called COMERA (COM Extensible Remoting Architecture) to enhance the extensibility and flexibility. We describe several application scenarios and implementations to demonstrate the power of such a componentized remoting architecture.

1. Introduction

Distributed object systems such as DCOM [Brown96], CORBA [CORBA95][Vinoski97], and Java RMI [Wollrath95], have become increasingly popular. In essence, they provide the infrastructure for supporting remote object activation and remote method invocation in a client-transparent way. A client program obtains a pointer (or a reference) to a remote object, and invokes methods through that pointer as if the object resides in the client's own address space. The infrastructure takes care of all the low-level issues such as packing the data in a standard format for heterogeneous environments (i.e., marshaling and unmarshaling), maintaining the communication endpoints for message sending and receiving, and dispatching each method invocation to the target object. In this paper, we use the term ***remoting architecture*** [COM95] to refer to the entire infrastructure that connects clients to server objects.

In general, a distributed object system does not have to specify how the remoting architecture should be structured. It can be treated as a black box as far as user applications are concerned. This black-box approach has the advantage of allowing vendors to put in their best performance optimization techniques. A disadvantage is that such architectures are usually not extensible. As a result, when low-level system properties such as load-balancing and fault tolerance are desirable, they need to be either tightly integrated with the infrastructure [Maffeis95] or provided through interception mechanisms outside the infrastructure [Narasimhan97].

In this paper, we propose an extensible remoting architecture and demonstrate that it facilitates the incorporation of low-level system properties into the infrastructure and allows them to be customized in a flexible way. We use COM's remoting architecture [COM95] [Brown96] as a starting point for the following two reasons. First, it has ***built-in extensibility***. By supporting a mechanism called ***custom marshaling***, COM allows a server object to bypass the standard remoting architecture and construct a custom one without requiring source code modifications to the former. Second, it is ***componentized***. COM's remoting architecture not only provides the basis for building distributed component-based applications, but also can be a distributed component-based application by itself. More specifically, the remoting architecture is constructed at run time by instantiating and connecting various dynamic components, and so a custom architecture can reuse some of the binary components from the standard one.

We point out the limitations of current COM remoting architecture, and propose a truly componentized, extensible architecture called COMERA. The approach is to use custom marshaling to implement COMERA, and then use COMERA to implement the low-level system properties. Three application categories are used to demonstrate the flexibility provided by COMERA: ***configurable multi-connection channels*** allow clients to use a single pointer to transparently talk to multiple server objects for performance or fault tolerance; ***transport replacement*** allows applications to run DCOM on any transport by wrapping protocol-specific client and server programs as COM objects and plugging them into COMERA; ***client-transparent object failover and migration*** allows a client to use an existing pointer to reach an object that has moved to another machine.

The paper is organized as follows. Section 2 gives the background for COM, and describes the current COM remoting architecture. Section 3 discusses the limitations of current architecture, presents the new COM-ERA architecture and specifies the interactions among its components. Section 4 describes the three application categories. Section 5 surveys related work, and Section 6 summarizes the paper.

2. Component Object Model

2.1. Overview of COM

In COM, an ***interface*** is a named collection of abstract operations (or methods) that represent one functionality. An ***object class (or class)*** is a named concrete implementation of one or more interfaces. An ***object instance (or object)*** is an instantiation of some object class. An ***object server*** is an executable (EXE) or a dynamic link library (DLL) that is responsible for creating and hosting object instances. A ***client*** is a process that invokes a method of an object. Figure 1 shows a client holding a pointer to one of the interfaces of an object. Each interface of an object represents a different view of that object and is identified by a 128-bit globally unique identifier (GUID) called the ***interface ID (IID)***. The object server contains multiple object instances from different classes, each of which is identified by a GUID called the ***class ID (CLSID)***. COM objects are usually created by class factories, which are themselves COM objects with standard interfaces for creating other COM objects.

COM specifies a binary standard that objects and their clients must follow to ensure dynamic interoperability. Specifically, any COM interface must follow a standard memory layout, which is the same as the C++ virtual function table [Rogerson96][Box98]. This allows COM applications to reuse binary code at run time through the client/server relationship, in contrast with the common notion of source code reuse at compile time. In addition, any COM interface must inherit from the IUnknown interface, which consists of a QueryInterface() call for navigating between interfaces of the same object, and two calls AddRef() and Release() for reference counting.

2.2. Remoting architecture

Figure 2 shows the current COM remoting architecture. The initial mechanism by which the client connects to the server is not shown. It can be an object activation call such as CoCreateInstance(), a binding call through a moniker (specifying a CLSID as well as particular persistent data) [Chappel96], or a lookup from a naming service. When the server object is created and is about to *export* an interface pointer, COM run-time will ask the object if it supports an IMarshal interface. If no such interface is supported, COM starts the following standard marshaling process [Chung97]:

- A ***standard marshaler*** is invoked to *marshal the interface pointer*, i.e., to pack sufficient information in an ***object reference (OBJREF)*** to be shipped to the client so that the client can use the remote pointer in a transparent way.
    - The standard marshaler loads and creates an interface stub according to the requested IID. An ***interface stub*** is itself a COM object that knows how to unmarshal input parameters and marshal output parameters for all method calls of an interface identified by a particular IID.
    - The standard marshaler gives *the pointer to the created interface stub* to a stub manager, and gets back an ***Interface Pointer ID (IPID)***. The ***stub manager*** will be in charge of dispatching each client call to the target interface stub based on the IPID tagged to the call.
    - The standard marshaler packs the IPID, the communication endpoint information (e.g., RPC string binding), and other information into a standard OBJREF and gives it to COM run-time.
- COM run-time ferries all the information obtained from the marshaler to the client side, activates a standard unmarshaler, and hands it the OBJREF.
- The standard unmarshaler creates a standard ***object proxy***, which serves as the proxy for all IUnknown method calls to the remote object. If the requested interface is not IUnknown, the object proxy also loads and creates an appropriate ***interface proxy***, aggregates it [Rogerson96], and exposes the interface of the interface proxy as if it is the object proxy's own interface.
- The object proxy also loads and creates a standard ***RPC channel object***, and uses the information in OBJREF to initialize the channel. The channel object can then use the communication endpoint information to reach the server. It also tags each call with the associated IPID so that it

can be properly dispatched once it reaches the server.
- Finally, COM run-time returns to the client an interface pointer. If it's an IUnknown pointer, it points to the object proxy; otherwise, it points to an interface proxy aggregated into the object proxy.

Once the standard remoting architecture is established, the client can make calls through the obtained pointer. Each call first enters a proxy, gets appropriately marshaled in the Network Data Representation (NDR) format [DCE95], sent by the channel object to the server endpoint, dispatched by the stub manager, gets unmarshaled by an interface stub, and finally delivered to the server object. Since the entire process is a sequence of local and remote function calls, the reply is done by simply returning from those function calls and reversing the marshaling procedure.

An object can declare that it wants to implement custom marshaling by supporting the IMarshal interface. In this case, the standard remoting architecture is not created. Instead, the object specifies (or itself acts as) a ***custom marshaler*** that is responsible for constructing ***custom OBJREFs*** and specifying the CLSID of a ***custom unmarshaler*** to be activated at the client side to receive the OBJREFs. The custom unmarshaler can create (or itself act as) a ***custom proxy*** that does application-specific processing and uses a application-specific communication mechanism to interact with the server.

3. Extensible Remoting Architecture

3.1. Extensibility issues in current COM remoting architecture

Custom marshaling provides the basis for extensibility in COM remoting architecture. Applications can achieve stronger low-level system properties by plugging in their own custom remoting architecture without having to modify the source code of the standard one. However, most such applications do not want to rebuild the entire remoting architecture; instead, they often want to reuse existing architecture as much as possible and replace only those parts that are specific to them. For example, very few applications need to replace the interface proxies and stubs that do marshaling and unmarshaling; some applications need to replace only the client-side architecture, while some need to modify only the server-side architecture. The examples described in the next section illustrate these different requirements.

The above discussion motivates the concept of a *componentized remoting architecture*. In addition to providing the infrastructure for higher-level componentized applications, if the remoting architecture itself is also componentized, then the benefits of software reuse can also be realized at the lower level. Figure 2 shows that current COM remoting architecture is partially componentized: the proxies, channels, stubs, and marshalers are COM components, but the server endpoints and stub managers are not. This limitation makes it hard to replace the transport and to control the IPID assignment and call dispatching. The second limitation is a result of the intimacy between object proxy and channel object. Although they interact through COM interfaces, this intimacy makes it hard to replace one of them without replacing the other. Specifically, the CLSID of the standard RPC channel object is not published, so it is hard for a custom proxy to connect to a standard channel. Also, the object proxy always creates and connects to a standard channel, so it is hard to reuse the object proxy while replacing the channel object.

Figure 2 illustrates the strength and the weakness of current COM remoting architecture in terms of extensibility. Basically, the architecture is extensible only at the upper layer where it interfaces to the client and the server applications. Applying custom marshaling at this layer is usually called ***semi-custom marshaling*** (or handler marshaling) as it essentially builds custom marshaling on top of standard marshaling. An arbitrary number of components connected in an arbitrary way can be inserted between the server object and the interface stubs as part of the *interface pointer marshaling process*. Such extensibility is useful for parameter tracing and logging, input value checking, etc. Similarly, arbitrary components can be inserted between the client and the proxies as part of the *interface pointer unmarshaling process*. This can be an ideal place for data caching logic, for example. In contrast, current COM remoting architecture has limited flexibility for applications that require extensibility at the lower layers. For example, fault tolerance mechanisms often need to get access to call parameters in their marshaled format for efficient logging or replication. Currently, that would require rebuilding the entire remoting architecture.

3.2. The COMERA architecture

We propose a new architecture called ***COMERA (COM Extensible Remoting Architecture)*** to address the above issues. Our approach is to first use custom marshaling to rebuild the standard remoting architecture, and then redesign parts of it to enhance extensibility. Figure 3 shows the overall COMERA architecture. Since the original interface proxies and stubs are packaged as binary COM objects, COMERA can reuse them without requiring a new IDL compiler or any recompilation. COMERA improves upon current COM remoting architecture in the following aspects:

- COMERA stub manager is a COM object and that offers two advantages. First, applications can replace the stub manager with a custom one to control IPID assignment and call dispatching. Second, COMERA marshaler interacts with the stub manager through a specified COM interface, and so replacing stub manager does not require the marshaler to be replaced as well.
- COMERA endpoint is also a COM object. It can be replaced to enable pre-dispatching message processing such as message logging and decryption. Since it provides communication endpoint information through a specified COM interface, a custom endpoint object can work with a COMERA marshaler.
- COMERA extends the IMarshal interface to include one more method call GetChannelClass() that allows a server object to specify the CLSID of a custom channel. When COMERA object proxy receives the OBJREF (standard or custom), it creates a channel object of the specified CLSID and initializes it with the OBJREF through a specified COM interface.
- The CLSID of the COMERA RPC channel object is specified so that any custom proxy can reuse this standard channel.

4. Applications

In this section, we describe three application categories to illustrate the benefits of COMERA's componentized remoting architecture. Configurable multi-connection channels enable dynamic transparent fault tolerance and support the notion of Quality-of-Fault-Tolerance. Transport replacement facilitates the low-level manipulation of marshaled data stream. Finally, the ability to restore server-side communication and dispatching state makes it possible to implement transparent object failover and migration.

4.1. Configurable multi-connection channels

A generic mechanism for transparent fault tolerance is for a client-side infrastructure to connect to multiple equivalent servers. The infrastructure can then mask server failures by retrying another server when one fails, or by sending each request to multiple servers simultaneously. This can be implemented on current remoting architecture using semi-custom marshaling as follows. The server object IMarshal routine packs multiple standard OBJREFs (corresponding to multiple equivalent objects) into one custom OBJREF; a custom proxy extracts and unmarshals each standard OBJREF into a pair of object proxy and standard channel connecting to one of the objects. Clearly, this is not efficient because the object proxy and the aggregated interface proxies are unnecessarily duplicated. Also, the marshaling and unmarshaling routines may be unnecessarily executed multiple times.

Figure 4 shows how COMERA allows the above fault tolerance mechanism to be implemented in a more natural and efficient way. In addition to packing multiple standard OBJREFs into a custom OBJREF, the server object also specifies the CLSID of a configurable multi-connection channel object. Connections to multiple objects are encapsulated inside the custom channel instead of a custom proxy. This allows the same marshaled data stream to be shared to reduce both time and memory overhead. Such architecture can also be used to provide *configurable timeouts*, which is currently not supported by COM.

We have implemented a system based on the architecture shown in Figure 4 to support ***Quality-of-Fault-Tolerance (QoFT)***. A server object dynamically determines the level of fault tolerance that should be provided for each client when the client first connects to the object, based on the client's login account. If it is a base-level client, standard remoting architecture without any fault tolerance is established. Otherwise, the determined level and the OBJREFs of the chosen server objects are transmitted to the client side to initialize a QoFT channel, of which the CLSID is also specified by the server. A level-1 client normally connects to a primary object, but will switch to a backup object when the primary server fails and the call times out. For a level-2 client, every call is sent to multiple objects and the first response is delivered to the client. This approach masks failures as well as improves response time. The system also supports *dynamic code downloading*: the DLL code of the QoFT channel object can be downloaded to the client as part of the custom marshaling stream. A custom unmarshaler is activated to extract the code and register it with the registry so that a QoFT channel object can be instantiated from it. This feature allows a service provider to try out different QoFT plans without requiring clients to install new software.

4.2. Transport replacement

According to the specification [Brown96], DCOM runs on top of RPC. In turn, RPC can run on top of different transports. For example, on Windows NT, Microsoft RPC can be configured to run on TCP, UDP, NetBIOS, and IPX by simply changing a registry setting [Nelson97]. In addition to this flexibility, applications may wish to replace the standard RPC channel altogether for a number of reasons. For example, running DCOM on HTTP may be necessary for passing through certain firewalls. Some organizations may need to run DCOM on proprietary transports in order to interoperate with existing legacy systems. Some applications may require encrypted channels for additional security. Information-dissemination applications may want to replace unicast channels with multicast channels for efficiency. Transport replacement can of course be accomplished on current COM architecture by using custom marshaling, but that would generally require rebuilding the entire remoting architecture.

Figure 5 illustrates how a new channel can be plugged into COMERA without modifying the upper layer of the architecture. The custom endpoint object wraps the transport-specific server code with a COM interface. It hosts the server-side communication endpoint and supplies binding information to the marshaler. The custom channel object wraps the transport-specific client code with two COM interfaces: one for initialization with the binding information and one for the actual communication. Some transports exist in the form of protocol stacks and can be completely wrapped inside the two COM objects. Others may require the channel object to connect to a client-side daemon, which is connected to a server-side daemon that is in turn connected to the endpoint object.

4.3. Object migration

Object migration is a generic mechanism for load balancing and fault tolerance. The goal is to move an object to another machine while still allowing existing clients to connect to it. On the one hand, the remoting architecture facilitates implementing object migration in a transparent way by providing a natural hiding place for the migration logic. On the other hand, the abstraction that it provides to the applications may hide too many low-level details, which makes transparent object migration difficult. Specifically, an object instance is uniquely identified by an RPC string binding (containing an IP address and a port number) and an IPID. Since current COM remoting architecture hides the assignment of port numbers and IPIDs from the applications, it is difficult to migrate an object while maintaining the same port number and IPID so that existing client-side channels can still reach the migrated object.

With current architecture, transparent object migration can be implemented using semi-custom marshaling as follows. A custom proxy containing the migration logic is inserted between the client and the object proxy. When a migration occurs, the custom proxy either gets notified through a special callback interface or detects that when a call times out. It then queries a migration manager process that maintains a mapping between pre-migration OBJREF and post-migration OBJREF for each migrated object. When the custom proxy gets back the new OBJREF, it creates a new pair of object proxy and channel object to connect to the migrated object, and discards the original pair.

The COMERA architecture facilitates transparent object migration in two ways. First, similar to the discussions in Section 4.1, object migration should involve only channel objects but not object proxies. By pushing the migration logic from the proxy level down to the channel level, COMERA allows a custom channel to simply update its RPC binding to connect to the migrated object, without any object activation and deactivation overhead. Second, it is particularly useful for implementing transparent object failover, which is a special case of object migration where the migrated-to machine has the same IP address as the original one. Figure 6 shows how COMERA supports transparent failover without requiring any custom objects or migration logic on the client side. The failover of the IP address can be provided by commercial clustering software [NTMag97]. The failover endpoint object checkpoints and restores the RPC string bindings. The failover stub manager checkpoints and restores IPID assignments. Since the

RPC layer has a built-in reconnection capability when an existing connection is broken, the client will be able to automatically reach the failed-over object (with the same IP, port number and IPID) upon the reconnection.

5. Related Work

CORBA does not specify a standard remoting architecture. As a result, incorporating stronger system properties such as fault tolerance into CORBA-based systems is usually not done by exploiting the extensibility in the remoting architecture. Instead, three other approaches have been taken [Narasimhan197]. *Electra* [Maffeis95] and *Orbix+Isis* [Landis97] build the mechanisms for object replication and consistency management into the ORB itself. The *Eternal* system [Narasimhan97] intercepts IIOP-related system calls through the Unix /proc interface, and maps them to routines supported by a reliable multicast group communication system. In contrast with the above two application-transparent approach, a third approach is to provide fault tolerance through a CORBA-compliant *Object Group Service* [Felber96].

COM currently supports a *channel hook* mechanism to allow piggybacking out-of-band data, which can be considered as a simple form of extensibility. By supporting an IChannelHook interface, a sender can fill in additional data to be transmitted as body extensions [Brown96] in a DCOM message, and a receiver can retrieve each body extension using its unique ID. Iona Orbix allows eight *filters* to be inserted at different places to get access to marshaled or unmarshaled call parameters or return parameters [Iona96]. Similar capabilities can be implemented on COMERA through component insertion or replacement.

The newly announced *COM+* runtime and services [Kirtland97] promise to provide a general extensibility mechanism called *interceptors*. The interceptors are used to interpret special class attributes, to receive events related to object creation/deletion and method invocation, and to automatically enable appropriate services. The *Coign* runtime system [Hunt97] provides similar instrumentation capabilities for *Inter-Component Communication Analysis (ICCA)* that serves as the basis for optimal distribution of component-based applications across a network. Compared to COM+ interceptors and Coign, COMERA does not intercept object creation calls, but the architecture for component insertion and replacement is more flexible.

Legion [Grimshaw96] is a metasystems software project that aims at providing flexible support for wide-area computing. Among its top design objectives is an extensible core consisting of replaceable components for customizing mechanisms and policies. The emphasis on transparent fault tolerance, migration, and replication is similar to COMERA. The main difference is that Legion targets high-performance parallel computing, while COMERA places more emphasis on client-server based systems.

The *Globe* project [Homburg96] proposed an architecture for distributed shared objects. Each local object consists of four subobjects: a control object handling local concurrency control; a semantics object providing the actual semantics of the shared object; a replication object responsible for state consistency management; and a communication object that handles low-level communication. COMERA can be used to support this architecture by implementing the control and the semantics objects in a custom proxy, and the replication and the communication objects in a custom channel.

6. Summary

We have proposed COMERA as an extensible remoting architecture for COM. By componentizing the architecture into COM objects, COMERA makes the low-level distributed objects infrastructure itself as dynamic, flexible, and reusable as the applications that it supports. We used three application categories as examples to demonstrate the advantages of the new architecture. For the multi-connection channels category, we have implemented a Quality-of-Fault-Tolerance subsystem on top of COMERA to support dynamic determination of fault-tolerance levels and dynamic code downloading for custom proxies and channels. For the transport replacement category, we have successfully plugged a commercial reliable multicast protocol implementation into COMERA. A programming wizard can be provided to further simplify the tasks of channel and endpoint object wrapping. For the object migration category, we have implemented a transparent failover subsystem for COM objects on top of IP failover. Future work includes building active replication and distriuted shared objects on COMERA.

Acknowledgement

The authors would like to express thanks to Li Li (Cornell) for his contributions to the initial implementation of COMERA, to Emerald Chung, Chung-Yih Wang, and Yennun Huang from Lucent Technologies for their valuable discussions, and to the software architects and engineers from the DCOM mailing list for their positive feedback on the architecture.

References

[Box98] D. Box, Essential COM, Addison-Wesley, 1998.

[Brown96] N. Brown, C. Kindel, Distributed Component Object Model Protocol -- DCOM/1.0, http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-01.txt.

[Chappell96] D. Chappell, Understanding ActiveX and OLE, Redmond, Washington: Microsoft Press, 1996.

[Chung97] P. E. Chung, Y. Huang, S. Yajnik, D. Liang, J. C. Shih, C. Y. Wang, and Y. M. Wang, "DCOM and CORBA Side by Side, Step by Step, and Layer by Layer," in *C++ Report*, Vol. 10, No. 1, pp. 18-29,40, Jan. 1998.

[COM95] The Component Object Model Specification, http://www.microsoft.com/oledev/olecom/title.htm.

[CORBA95] The Common Object Request Broker: Architecture and Specification, Revision 2.0, July 1995, http://www.omg.org/corba/corbiiop.htm.

[DCE95] DCE 1.1: Remote Procedure Call Specification, The Open Group, http://www.rdg.opengroup.org/public/pubs/catalog/c706.htm.

[Felber96] P. Felber, B. Garbinato, and R. Guerraoui, "Designing a CORBA group communication service," in *Proc. the 15th Symp. on Reliable Distributed Systems*, pp. 150-159, Oct. 1996.

[Grimshaw96] A. Grimshaw and W. A. Wulf, "Legion," in *Proc. the Seventh ACM SIGOPS European Workshop*, Sep. 1996.

[Homburg96] P. Homburg, M. van Steen, and A. S. Tanenbaum, "An architecture for a wide area distributed system," in *Proc. the Seventh ACM SIGOPS European Workshop*, Sep. 1996.

[Hunt97] G. C. Hunt and M. L. Scott, "Coign: Efficient instrumentation for inter-component communication analysis," Tech Report 648, Dept. of Computer Science, University of Rochester, Feb. 1997.

[Iona96] Orbix 2.1 Programming guide, Iona technologies Ltd., http://www.iona.com/.

[Kirtland97] M. Kirtland, "Object-oriented software development made simple with COM+ runtime services," *Microsoft Systems Journal*, Vol. 12, No. 11, pp. 49-59, Nov. 1997.

[Landis97] S. Landis and S. Maffeis, "Building reliable distributed systems with CORBA," *Theory and Practice of Object Systems*, John Wiley & Sons, New York, 1997.

[Maffeis95] S. Maffeis, "Adding group communication and fault-tolerance to CORBA," in *Proc. Usenix Conf. on Object-Oriented Technologies*, June 1995.

[Narasimhan197] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "The interception approach to reliable distributed CORBA objects," in *Proc. the 3rd Conf. on Object-Oriented Technologies and Systems*, June 1997.

[Narasimhan97] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "Exploiting the Internet Inter-ORB Protocol interface to provide CORBA with fault tolerance," in *Proc. the 3rd Conf. on Object-Oriented Technologies and Systems*, June 1997.

[Nelson97] Michael Nelson, "Using Distributed COM with Firewalls", http://www.wam.umd.edu/~mikenel/dcom/dcomfw.htm, 1997.

[Rogerson96] D. Rogerson, Inside COM, Redmond, Washington: Microsoft Press, 1996.

[Vinoski97] S. Vinoski, "CORBA: Integrating diverse applications within distributed heterogeneous environments," in *IEEE Communications*, vol. 14, no. 2, Feb. 1997. http://www.iona.com/hyplan/vinoski/ieee.ps.Z.

[NTMag97] "Clustering Solutions for Windows NT," *Windows NT Magazine*, pp. 54-95, June 1997.

[Wollrath95] A. Wollrath, R. Riggs and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Journal, Computing Systems*, Vol. 9, No. 4, pp.265-289, Fall 1996.

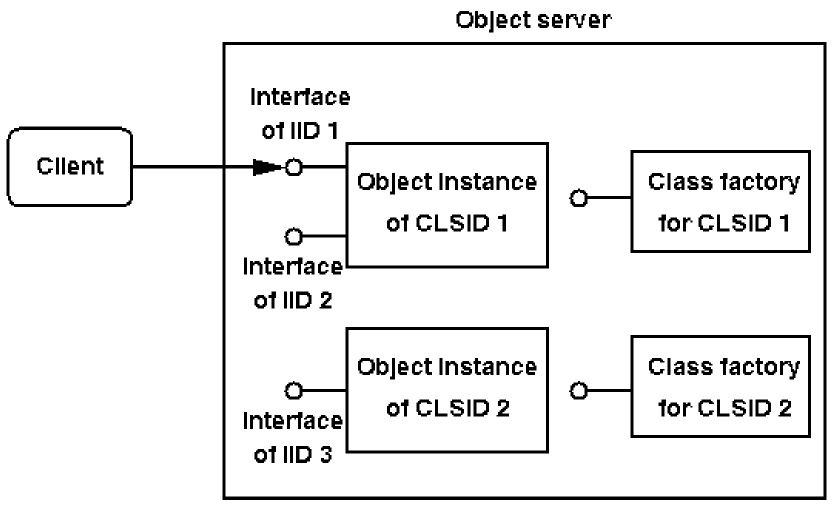
Object server
Interface of IID 1
Client
Object instance of CLSID 1
Class factory for CLSID 1
Interface of IID 2
Object instance of CLSID 2
Class factory for CLSID 2
Interface of IID 3

Figure 1. COM server, classes, objects, interfaces and client.

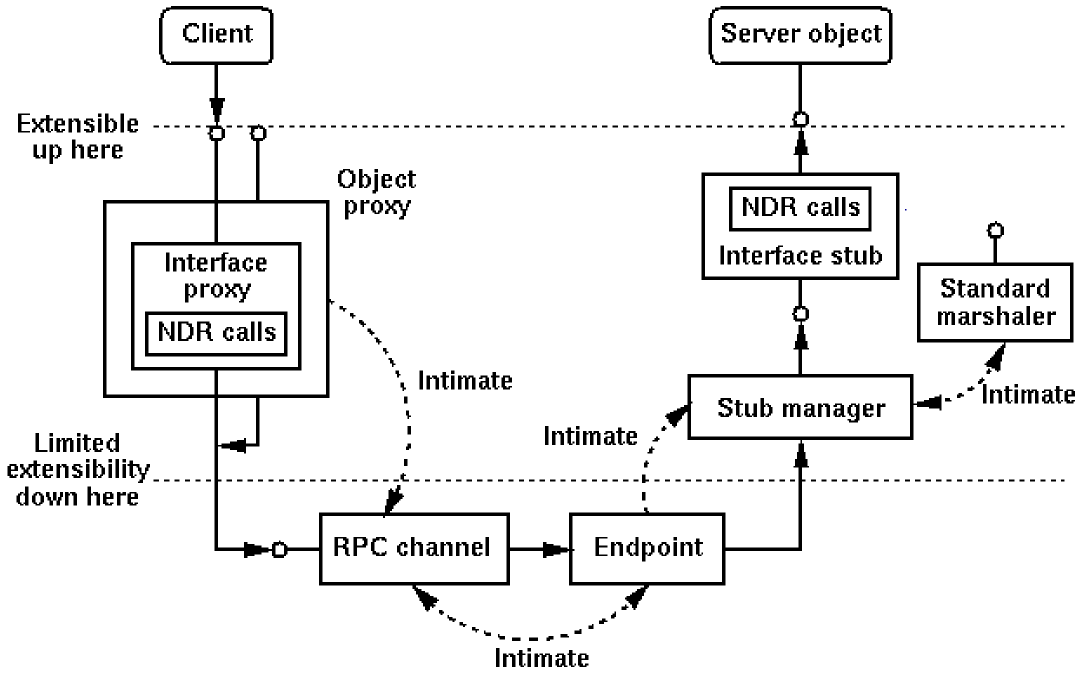

Figure 2. Limitations of current COM remoting architecture.

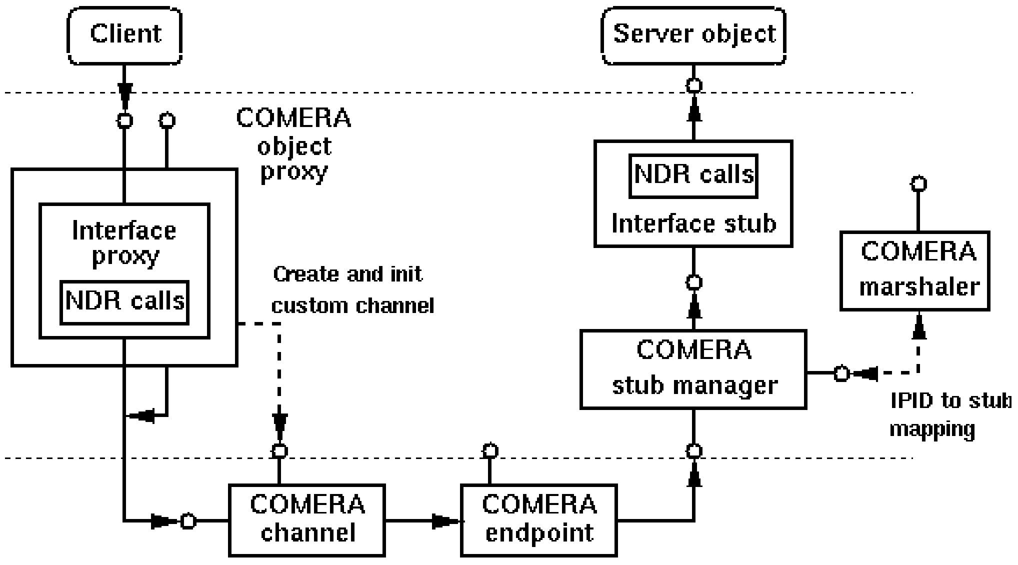

Figure 3. The COMERA architecture.

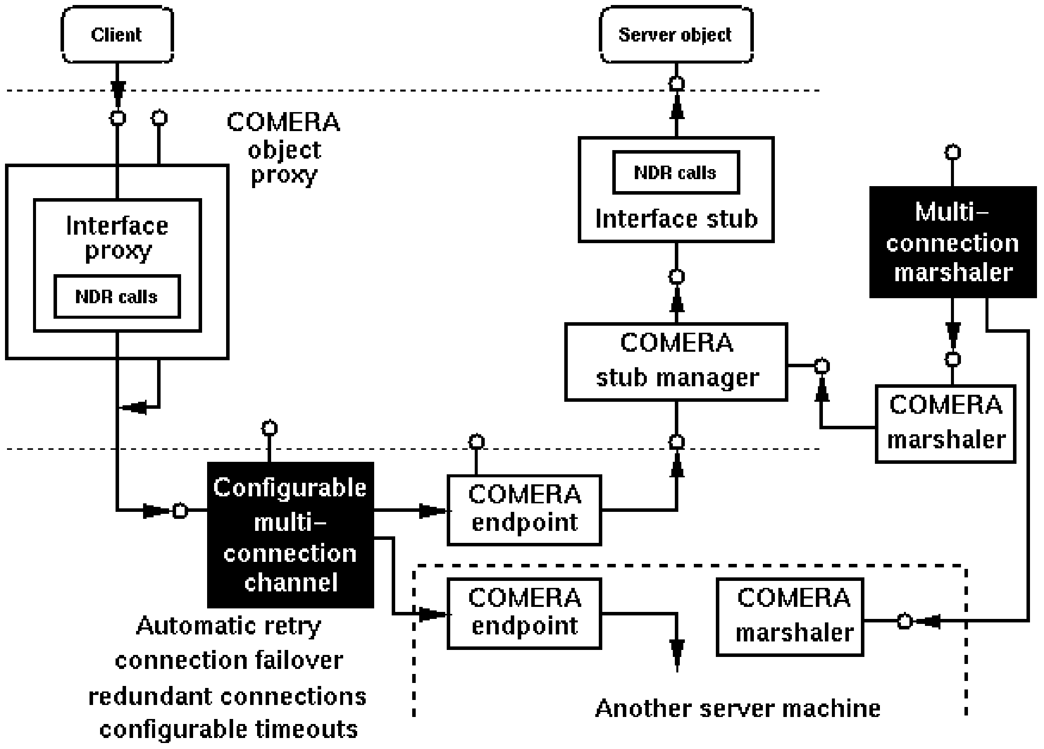

Figure 4. Configurable multi-connection channel on COMERA. (Inserted or replaced components are indicated by the black boxes.)

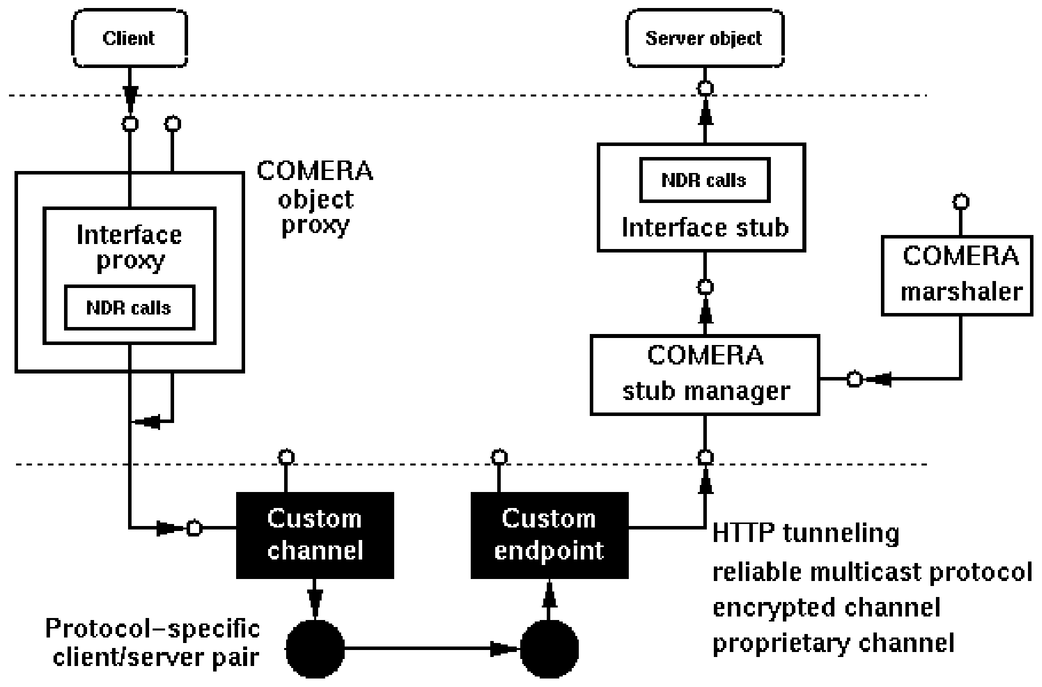

Figure 5. Transport replacement by replacing the channel and the endpoint objects.

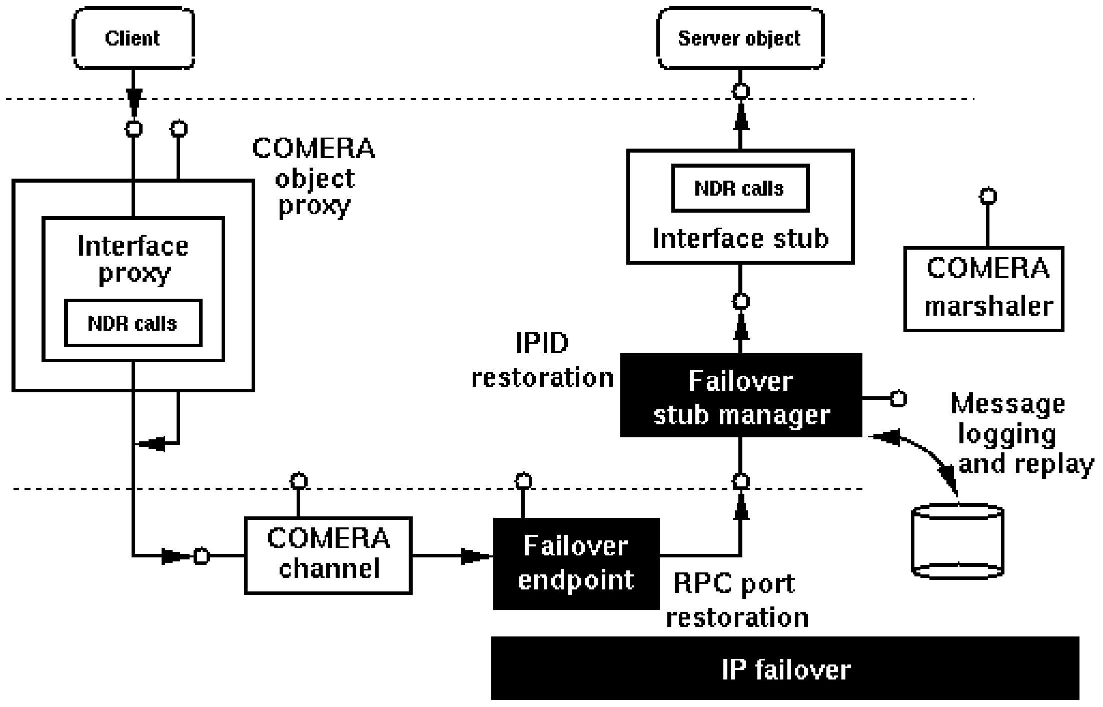

Figure 6. Client-transparent object failover with COMERA.