Reliability and Availability Issues In Distributed Component Object Model (DCOM)

Yi-Min Wang
AT&T Labs-Research
ymwang@research.att.com

Om P. Damani
Univ. of Texas at Austin
damani@cs.utexas.edu

Woei-Jyh Lee
New York University
wjlee@research.att.com

Abstract

Distributed Component Object Model (DCOM) is one of the emerging standards for distributed objects. Before DCOM can be used to build mission-critical applications, the reliability and availability issues must be addressed. In this position paper, we outline the current research directions of the InterCOM project, which exploits the dynamic behavior, the extensible architecture, and the component software model of DCOM to provide fault-tolerance capabilities to distributed applications.

1. Introduction[1]

In the component software architecture, applications are built from packaged binary components with well-defined interfaces [1]. It allows flexible update of existing applications, provides a higher-degree of application customization, encourages large-scale software reuse, and provides a natural migration path to distributed applications. The *Component Object Model (COM)* [2] is an approach to achieving component software architecture. COM specifies a way for creating components and for building applications from components. Specifically, it provides a binary standard that components and their clients must follow to ensure dynamic interoperability. This enables on-line software update and cross-language software reuse.

Distributed Component Object Model (DCOM) [3] is the distributed extension of COM. It is an application-level protocol for object-oriented remote procedure call (ORPC). The DCOM protocol is layered on top of the OSF DCE RPC specification [4], with a few extensions. For example, it specifies how pointers to remote objects are represented and how they can be resolved to find the actual objects. Effectively, DCOM transparently extends the capabilities and benefits of COM to a networked environment.

DCOM is now part of the Windows NT 4.0 operating system. Due to the increasing popularity of Windows NT, several companies are porting or plan to port DCOM to mainframes (such as IBM MVS) and various versions of Unix (such as Solaris, Digital Unix, and HP-UX) to ensure interoperability in a heterogeneous environment [5]. DCOM is also part of the ActiveX Core Technologies [6] that are being standardized by a software consortium called the Active Group. It can also be expected that DCOM will be increasingly used in the research community as a distributed object platform for systems research.

However, DCOM environment itself has several problems that need to be solved before it can be accepted by the research community as a viable research platform. Also, as demonstrated throughout the paper, the dynamic behavior of DCOM introduces new challenges as well as encourages novel solutions for building reliable distributed applications. These aspects motivated our *InterCOM* project. Due to its desktop document processing origin, many DCOM application programming interfaces (APIs) are not structured and presented in an intuitive way for building distributed client/server applications. As a result, although the design of the DCOM architecture is quite extensible (that is, reasonable defaults are provided for common cases, but can be overwritten through flexible programming hooks), it often involves non-intuitive programming hacks to provide the functionality required in a client/server environment. Some of these are reflected in the discussions in Section 3. The InterCOM project aims at extracting the mechanisms behind the APIs, and providing a toolkit to restructure them in a way that facilitates building higher-level object services [26] and implementing reliable, highly-available, and fault-tolerant distributed applications.

2. Overview of DCOM

In COM, an executable (EXE) or a dynamic link library (DLL) can serve as an *object server*. A server can implement *class factories* for multiple classes, each of which is identified by a 128-bit *globally unique identifier (GUID)*, called the *class identifier (CLSID)*. Each class factory can create object instances of a particular CLSID. An object can support *multiple interfaces*, each

[1] This work was performed during the summer internship of the second and the third authors in AT&T Labs-Research.

representing a different view or behavior of the object. Each interface is identified by a GUID, called the *interface identifier (IID)*. An interface usually consists of a set of functionally related methods. A COM client interacts with a COM object by acquiring a pointer to one of the object's interfaces, and invoking methods through that pointer.

The overall architecture of DCOM can be divided into three layers [7]: basic programming layer, remoting layer, and the wire protocol layer. At the *basic programming layer*, the client is provided with the illusion that it is always invoking methods on objects running in the same address space. The *remoting layer* [2] consists of the COM infrastructure that provides that illusion. The *wire protocol* [3] describes what are actually transmitted across the network when objects do not reside on the client machine.

At the basic programming layer, the client specifies a CLSID and an IID to obtain an interface pointer. The server, upon being activated by the COM infrastructure, creates and registers all supported class factories. A method on the class factory of the requested CLSID is invoked to create an object instance and return a pointer to the interface of the requested IID. The client can then invoke methods of that interface, or navigate to other interfaces of the same object instance.

Upon receiving an activation request from the client, the *Service Control Manager (SCM)* at the remoting layer checks to see if the client can attach to a running class factory. If not, it locates the server implementation through the *registry*, and activates the server. In the process of returning an interface pointer to the client, a server-side *stub* and a client-side *proxy* are created, connected by an RPC channel [2]. When the client invokes a method, the proxy marshals the parameters, and sends the request to the stub. The stub unmarshals the parameters, invokes the actual method on the object, marshals the return values, and replies back to the proxy which unmarshals the values and hands them back to the client.

At the wire protocol layer, the client-side SCM performs remote activation by invoking an RPC interface method on the server-side SCM [23]. The wire-level representation of a returned interface pointer includes a server identifier, the address of the resolver that can translate the identifier to the actual server endpoint, and an interface pointer identifier that uniquely identifies the interface instance within that server. Upon unmarshaling the interface pointer, the client-side RPC subsystem requests the resolution of the server identifier, and caches the returned endpoint information for future method invocations. The parameters and return values of method invocations are marshaled in *the Network Data Representation (NDR)* format [4].

3. The InterCOM Project

Our long-term goal is to provide a *fault-tolerance-programming wizard* that is able to guide programmers to choose the right fault-tolerance techniques and generate much of the boilerplate code. The wizard would cover at least the three main approaches to fault tolerance: *checkpointing and message logging* [10,11], *virtual synchrony* [15], and *transaction* [27]. In this section, we outline the current research directions of the InterCOM project, which exploits interception-based mechanisms to provide reliability and high availability to DCOM applications. It addresses the issue of checkpointing and call logging in a primary-backup, failover configuration. Extensions to the active-active configuration with replicated processes based on group communications will be part of a joint project with the Ensemble group at Cornell [8]. We plan to use Microsoft Transaction Server [23] for the transaction part.

We consider primary-backup systems which may or may not have an underlying clustering software (such as Wolfpack [9]) for providing the failover of system resources such as IP addresses, shared disk, etc. Complete application failover in such systems generally involves the primary server *checkpointing* its critical data for use by the backup; the client *locating* the backup server upon a failure, and issuing a *reconnection*; the server executing a *different piece of code* if software bugs seem to be the cause of the failure. We next describe the challenges and opportunities that DCOM presents in these aspects.

3.1. Data Checkpointing

A stateful server application usually needs to checkpoint its critical data during normal execution so that, upon a failure, the backup server can recover as much pre-failure state as possible from the checkpoint. In a cold backup scheme, the checkpoint is periodically saved on a shared disk, and the backup is activated to reload that checkpoint only upon a failover. In a warm backup scheme, the checkpoint is transferred to the memory of an initialized backup. In both cases, message logging can be employed as a form of incremental checkpointing to improve performance.

COM-based server applications introduce several interesting new twists to the problem of checkpointing. In a traditional monolithic application, the main program is totally in charge of declaring and checkpointing all critical data [10]. This may not be an easy task when unknown critical data inside imported libraries are present. This difficulty becomes the normal case in component-based applications in which reused components may constitute a

large percentage of the total amount of code. Therefore, COM defines a set of standard interfaces (the `IPersist*` family) that components can expose to provide data persistence. This allows the knowledge about the critical data of each component to be encapsulated inside the component itself. The application simply queries each constituent component for one of the standard interfaces, and asks the component to checkpoint itself. To allow critical data from all constituent components to be saved in the same file, COM supports the concept of "a file system within a file", called *structured storage* [12]. It also supports a transacted mode that can be used to ensure either the checkpoint operations of all components succeed, or none of them takes effect. Another challenge in checkpointing DCOM applications is that COM objects come and go due to the inherently dynamic nature of the model. As a result, the issue of tracking and identifying those objects that are still active also needs to be addressed.

3.2. Object Locator

Object locator is used by COM clients to find the desired primary server objects, as well as to locate the backup server objects upon a failure. COM provides several locator-related services. One of the goals of InterCOM is to integrate these services into a single naming service for locating COM objects, and possibly integrate it with other general-purpose, standard naming services.

The most well-known object locator in COM is the registry. It maps a CLSID (or a readable name called ProgID) to the path name of the server executable that supports the CLSID. However, the registry is consulted only after the SCM has failed to locate any running object instance. There are at least two ways for a client to bind to a running instance. First, if a running class factory for the requested CLSID has been registered in the *class object table*, SCM retrieves the class factory pointer directly from the table without activating a new server instance. Alternatively, a specific object instance can be named by using a moniker. A *moniker* [12,24] is itself a COM object supporting the `IMoniker` interface. Each moniker specifies a CLSID and identifies the persistent data for an instance of that CLSID. By registering a moniker with the *Running Object Table (ROT)* [2], an object instance allows clients requesting that moniker to attach to it.

A higher-level object locator service can be implemented using the referral (or broker) components [13]. A referral component manages a pool of interface pointers to object instances possibly running on different machines. It can support a naming scheme as well as perform failure detection and load balancing. In such architecture, a client always contacts the referral component to get access to an initial interface pointer to a server object. The underlying distributed object support of DCOM allows the client to talk to the object directly in subsequent invocations, without passing through the referral component. (This architecture bears some similarities to the single-IP-image approach to supporting Web server clusters [14].)

3.3. Client Reconnection

There are two approaches to enabling automatic client reconnections upon a server failure: toolkits and wrappers [15]. In the first approach, client programs link with a toolkit and invoke special APIs to make connection [10]. In the second approach, source code-transparent wrappers intercept normal connection requests sent by the clients, and issue reconnections when a failure occurs. As explained next, the dynamic behavior and extensible architecture of DCOM facilitate the implementation of dynamic wrappers that allow the server objects to decide when to apply which wrappers based on run-time information.

As described previously, when a client requests an interface pointer, which server object it will eventually bind to depends on several table-lookup operations. It is therefore possible to manipulate the mapping information in those tables at run-time to dynamically change the application behavior. For example, one can provide a wrapper component by using COM's containment or aggregation technology [1]. The former allows the wrapper to receive a client request, perform pre-processing, invoke the actual server component, perform post-processing and then hand the results back to the client. The latter allows changing application behavior by adding additional interfaces that the client or the COM infrastructure may query. Wrappers can be injected into the system by either modifying the registry settings, or placing mappings in the class object table or ROT to bypass registry lookups.

Server objects themselves can also decide which wrapper to inject by using a technique called *custom marshaling* [16,17]. The marshaling architecture described in the previous section is called the *standard marshaling*: data passing between the server and the client are marshaled into a standard format in a standard way. Standard marshaling is actually a special case of the more general custom marshaling. By supporting the `IMarshal` interface, a server object indicates that it wants to establish proprietary communication with the client-side proxy, and so the COM infrastructure should not create the standard proxy/stub pair. That interface allows an object to specify the CLSID of the custom proxy that should run on the client side and can interpret the custom marshaling packet. Custom marshaling is commonly used for caching immutable objects on the client side to efficiently support read operations locally. It can also be useful for injecting dynamic client-side fault-tolerance agents for issuing

reconnections. Custom marshaling is one of many examples that demonstrate the extensibility of the DCOM architecture.

When a wrapper issues a reconnection, it needs a locator for finding the correct backup server object. If the original binding to the primary server object was based on a moniker, a similar binding call can be made to the backup machine for reconnection. If the original connection was made to a fresh instance of a particular CLSID, then a separate mechanism must be provided to allow the server object and the client to agree on a name upon object creation.

3.4. Software-fault Tolerance

In general, there are three approaches to tolerating software bugs. The simplest one is the *environment diversity* approach [18,19], which reexecutes the same program with the same set of input but in a different environment. Usually, this can be achieved by following the same failover procedure for recovering from hardware failures. Another one is the *data diversity* approach [20], which executes the same program on a transformed but consistent set of data. This can be implemented by the containment wrappers described previously, where pre-processing and post-processing consist of application-specific data transformations.

The third approach is the *design diversity* approach [21,22], which executes a different program implementing the same functionality. Design diversity has not been widely used possibly because building multiple versions can be costly and they may still share similar kinds of bugs. The design of COM provides several arguments that design diversity may eventually be practically useful in the component software architecture. First, interface specifications are what COM is all about. By strictly separating interfaces from implementations, COM encourages different implementations of the same interfaces. In other words, the existence of multiple versions supporting the same functionality should actually be the normal case in COM. Moreover, by asking each component to register which *component categories* (i.e., which sets of interfaces) it supports, it is possible to standardize the procedure of invoking an alternate upon a failure, and provide a toolkit to hide all the registry query and update activities. The rules that COM clients must specify a globally unique identifier when requesting an interface, and that an interface is immutable once it is assigned an identifier, ensure that an alternate must unambiguously support the interfaces that a client wants. Finally, COM's language neutrality allows the same interfaces to be implemented in different programming languages. This has the potential of greatly improving the effectiveness of design diversity. For example, a memory corruption error in a component implemented in a language that supports pointers would not appear in another component written in a language with no pointers.

4. Summary

We have described the challenges and opportunities that DCOM presents in terms of building reliable and highly available distributed applications. The current research directions of the InterCOM project were briefly described in the context of primary-backup, failover recovery. The techniques can also serve as the basis for extending InterCOM to the active-active, replicated processes setting as supported in Ensemble. Such extensions may include integrating the notion of object replication into the object naming mechanism; providing a custom IDL compiler for generating client proxies that access object groups in an optimal fashion; and providing object replication management that supports dynamic instantiation of objects based on a QoS description [25].

Acknowledgments

The authors would like to express their sincere thanks to Ken Birman and Werner Vogels for their valuable discussions and suggestions.

References

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

[2] Microsoft Corporation and Digital Equipment Corp., "The Component Object Model Specification," http://www.microsoft.com/oledev/olecom/title.htm, Oct. 1995.

[3] N. Brown and C. Kindel, "Distributed Component Object Model Protocol -- DCOM/1.0", Internet Draft, http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-01.txt, Nov. 1996.

[4] OSF DCE RPC Specification, http://www.osf.org/mall/dce/free_dce.htm, 1994.

[5] COM/DCOM Resources, http://www.research.att.com/~ymwang/resources/resources.htm.

[6] "ActiveX Core Technologies Description", http://www.activex.org/announce/ActiveX_Core_Technologies.htm.

[7] Y. M. Wang, "Introduction to COM/DCOM", tutorial slides, http://www.research.att.com/~ymwang/slides/DCOMHTML/ppframe.htm, 1997.

[8] "The Ensemble Distributed Communication System", http://simon.cs.cornell.edu/Info/Projects/Ensemble/index.html.

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

[10] Y. Huang and C. Kintala, "Software implemented fault tolerance: Technologies and experience," in Proc. IEEE Fault-Tolerant Computing Symp., pp. 2-9, June 1993.

[11] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang, "A survey of rollback-recovery protocols in message-passing systems." Tech. Rep. No. CMU-CS-96-181, Dept. of Computer Science, Carnegie Mellon University, 1996 (also available at http://www.research.att.com/~ymwang/papers/ surveyCR.htm).

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

[13] "DCOM Technical Overview", http://www.microsoft.com/windows/common/pdcwp.htm.

[14] O. P. Damani, P. E. Chung, Y. Huang, C. Kintala, and Y. M. Wang, "ONE-IP: Techniques for Hosting a Service on a Cluster of Machines", Proc. 6th WWW Conference, pp. 735-743, April 1997. (Also available at http://www6.nttlabs.com/HyperNews/get/PAPER196.html.)

[15] K. P. Birman, "Building Secure and Reliable Network Applications", Greenwich, CT: Manning Publications Co., 1996.

[16] K. Brockschmidt, "Inside OLE", Redmond, Washington: Microsoft Press, 1993.

[17] D. Box, "Q&A ActiveX/COM", Microsoft Systems Journal, pp. 93-105, March 1997.

[18] Y. M. Wang, Y. Huang, K. P. Vo, P. Y. Chung, and C. Kintala, "Checkpointing and its applications," in Proc. IEEE Fault-Tolerant Computing Symp., pp. 22--31, June 1995.

[19] Y. M. Wang, Y. Huang, and W. K. Fuchs, "Progressive retry for software error recovery in distributed systems," in Proc. IEEE Fault-Tolerant Computing Symp., pp. 138-144, June 1993.

[20] P. E. Ammann and J. C. Knight, "Data diversity: An approach to software fault-tolerance," IEEE Trans. Computers, Vol. 37, No. 4, pp. 418-425, Apr. 1988.

[21] A. Avizienis, "The N-version approach to fault-tolerant software," IEEE Trans. Software Eng., Vol. SE-11, No. 12, pp. 1491-1501, Dec. 1985.

[22] B. Randell, "System structure for software fault tolerance," IEEE Trans. Software Eng., Vol. SE-1, No. 2, pp. 220-232, June 1975.

[23] R. Grimes, "Professional DCOM Programming", Olton, Birmingham, Canada: Wrox Press, 1997.

[24] D. Box, "Q&A ActiveX/COM", Microsoft Systems Journal, pp. 93-108, July 1997.

[25] W. Vogels, "A programming environment for building cluster-aware DCOM applications",private communication, June 1997.

[26] J. Siegel, "CORBA Fundamentals and Programming", John Wiley & Sons, 1996.

[27] J. Gray and A. Reuter, "Transaction Processing: Concepts and Techniques," San Mateo, CA: Morgan Kaufmann, 1993.

Biography

Yi-Min Wang received his BS degree in electrical engineering from National Taiwan University in 1986, and the MS and PhD degrees in electrical and computer engineering from the University of Illinois at Urbana-Champaign in 1990 and 1993, respectively. He joined AT&T Bell Laboratories in 1993, and is currently with AT&T Labs-Research. His research interests include distributed systems, fault tolerance, and networking.

Om P. Damani obtained his B. Tech. Degree in Computer Science and Engineering from Indian Institute of Technology, Kanpur, India in 1994. He is currently a PhD student in the Department of Computer Science, University of Texas at Austin. His research interests are fault tolerance, distributed objects, and distributed simulation.

Woei-Jyh Lee received his BS degree from Department of Computer Science and Information Engineering, National Taiwan University in 1993. He is working towards the MS degree in Department of Computer Science, New York University. His research interests include distributed objects, open systems, and Internet protocols.