1. INTRODUCTION

Open Distributed Processing (ODP) systems reduce the complexity of designing and implementing applications in distributed computing environments. Applications that run on an ODP system follow standards that allow them to be portable across heterogeneous platforms, and also allow them to interoperate with other distributed applications that follow the same standards. This paper describes our experiences augmenting an ODP system with a toolset that can automatically add high availability to distributed objects. These objects adhere to the Common Object Request Broker Architecture standard (CORBA) [OMG91, Vinoski93]. A highly available object continues to run in the presence of hardware or software faults, as well as planned maintenance activities such as hardware and software upgrades. Applications where high availability is important include financial transaction-processing systems, telecommunications, medical systems, and real-time process control.

The toolset includes a number of software-based techniques for providing high availability with little or no intervention from the programmer. Application developers implement their objects following the CORBA standard and link them with our toolset to provide the required high availability. The implementation uses a locally-developed CORBA library [Diener94] which runs over OSF's Distributed Computing Environment (DCE) [Millikin94, OSF91] on SparcStations running SunOS 4.1 and DEC Alphas running OSF/1.

We chose a CORBA-compliant platform to implement our toolset because we believe that many future distributed applications will adopt the CORBA standard. Thus, our toolset could be ported to other platforms and application domains. We decided also to rely on DCE to provide the networking support. There are several alternatives to this decision, each typically consisting of a CORBA package that implements its own name service, and interacts with the network directly through the socket layer in Unix® or using another RPC system. We decided against using these CORBA packages because their reliance on non-standard naming, and lack of security facilities. DCE does not have these problems and complements the CORBA library that we had with its naming, security, and RPC services. Our choice also offers a potential for interoperability with other "pure" (i.e. non-CORBA) DCE applications and tools.

The implementation of our toolset was able to benefit from many of the facilities that CORBA and DCE provide, such as the name server, the uniform Interface Definition Language, and RPC groups, among many others. These contributed to the simplification of the implementation effort and we were able to verify the benefits that both standards offer for the development of distributed applications. Unfortunately, our laboratory experiences revealed a number of problems with both CORBA and DCE. Though some problems were specific to our platform, others were resulting from the definitions of both standards and could impede other researchers and developers who would be involved in projects using CORBA and/or DCE. We discuss these problems and we suggest solutions to them.

The primary focus of this paper is our experiences using CORBA and DCE to build a high-availability toolset. A detailed description of the implementation of the toolset itself and a performance evaluation can be found elsewhere [Elnozahy95]. Section 2 includes an overview of the high availability toolset to provide the necessary background and context for describing our experience with CORBA and DCE, which is detailed in Section **3**. We present a summary and our conclusions in Section 4.

® Unix is a registered trademark of Novell, Inc.

2. The Design of a High Availability Toolset

2.1. Design Goals and Overview

We have constructed a toolset that provides high availability to distributed applications with little or no additional support by the application programmer. This approach relieves the programmer from the mundane and often error-prone tasks of handling failures and recoveries at the application layer. Also, the approach has the potential of improving the availability of existing applications that were written without consideration for high availability.

The system uses a local implementation of the CORBA standard, called Touring Distributed Objects (TDO) [Diener94]. In this system, application programs consist of distributed server and client objects that communicate by remote method invocation according to the CORBA standard [OMG91]. Server objects follow a multithreaded programming model, where a thread is automatically started to execute the method invoked by a remote client object. It is assumed that the execution of a method invocation is short and the corresponding thread lives only during the course of serving the invoked method. This model is consistent with the familiar remote procedure call paradigm for client/server applications.

TDO objects are written in C++ and use CORBA's Interface Definition Language (IDL) to define the exported methods. Figure 1 illustrates how the components that make up a TDO CORBA/DCE application fit together.

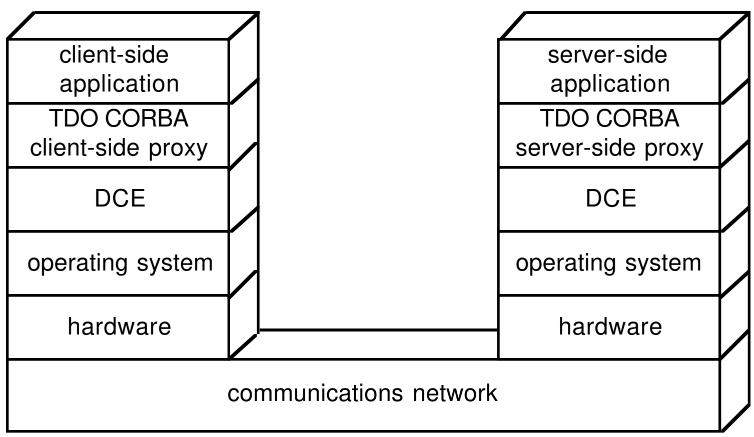

Figure 1 — The Structure of a TDO CORBA/DCE Application

Figures 2a and 2b show the C++ source code for a simple TDO CORBA/DCE application. The TDO runtime system provides support for exporting the server methods, implemented by server implementation (SI) objects, by assigning a unique name in a hierarchical name space for each instance of an exported class. The TDO compiler automatically generates two proxy classes, one for the client side and the other for the server. These classes act as stubs of remote communication for each IDL interface. The server proxy class (SP) handles all the details of translating the remote procedure calls into C++ method invocations, and of exporting the name of the class into the global name space. Client objects that wish to interact with a certain server must be linked with the corresponding client proxy of the server. The client can then access the server through normal invocations of C++ methods. The client proxy locates the required server and handles the details of translating the remote invocations into remote procedure calls.

```
// simple TDO server
main(int argc, char *argv[])
{

  // create a proxy and an implementation
  simple proxy ;
  Simple impl ;
  // plug the implementation into the proxy
  proxy.represent(&impl);
  // "advertise" the proxy in the CDS so that
  // clients can bind to it...
  proxy.export("/.:/tm/simple.server");
  // fork a thread which listens for incoming messages
  CORBA::BOA::impl_is_ready();

  // wait for the thread to terminate (i.e. wait forever...
  // use ctrl C to kill off the server)
  CORBA::BOA::wait() ;
  exit(0) ;
}
```

Figure 2a — C++ Code for a Simple TDO CORBA Server

```
// simple TDO client
main()
{
  // create a proxy
  simple clientProxy;
  // "bind" it to its implementation using the
  // represent member function.
  clientProxy.represent("/.:/tm/simple.server");

  // send it a ping message
  clientProxy.ping() ;
  exit(0) ;
}
```

Figure 2b — C++ Code for a Simple TDO CORBA Client

The system uses DCE to provide the underlying support in the form of remote procedure calls, multithreading, naming, RPC groups, and security. Most notably, the proxy class uses DCE RPC to implement remote method invocation. The proxies also use the CDS name space of DCE to implement the global name space. Additionally, the implementation relies on DCE's multithreading support for implementing lightweight threads. The application, however, does not need to call the DCE layer directly, since TDO abstracts the support provided by DCE. This approach simplifies the task of the programmer.

2.2. Mechanisms for High Availability

The toolset provides a repertoire of mechanisms that can be used to add high availability to objects. These mechanisms differ according to their costs and reliability guarantees, with the cost increasing with the degree of reliability desired. High availability is added by linking the server proxy of a highly-available object with a library that includes the necessary support. Application developers can select the appropriate availability mechanism for their applications by linking with an appropriate library. The system is designed to make this choice as transparent as possible to the application. A detailed description of the availability mechanisms used by the system appears elsewhere [Abbott90, Birman93, Elnozahy93, Gray91, Huang93, Segal93]. We focus here on describing the implementation of one technique, namely the primary/backup scheme and how it made use of CORBA and DCE support. In this technique, several copies of a server object exist on different machines. One copy is designated the primary while the others are backups. The primary is the contact point for the client proxies, which are unaware of the presence of the backups. Each backup has its own private copy of each proxy class that is exported by the server. Only the primary's proxy, however, exports the corresponding class instance into the global name space.

During normal operation, the primary receives invocation requests from client proxies for the methods that the server object exports. The primary executes the appropriate method and sends to the backup objects the corresponding updates to the object's state due to the method invocation. After the backups acknowledge the receipt of these updates, the primary replies to the client object with the result. This sequence of steps is shown in Figure 3.

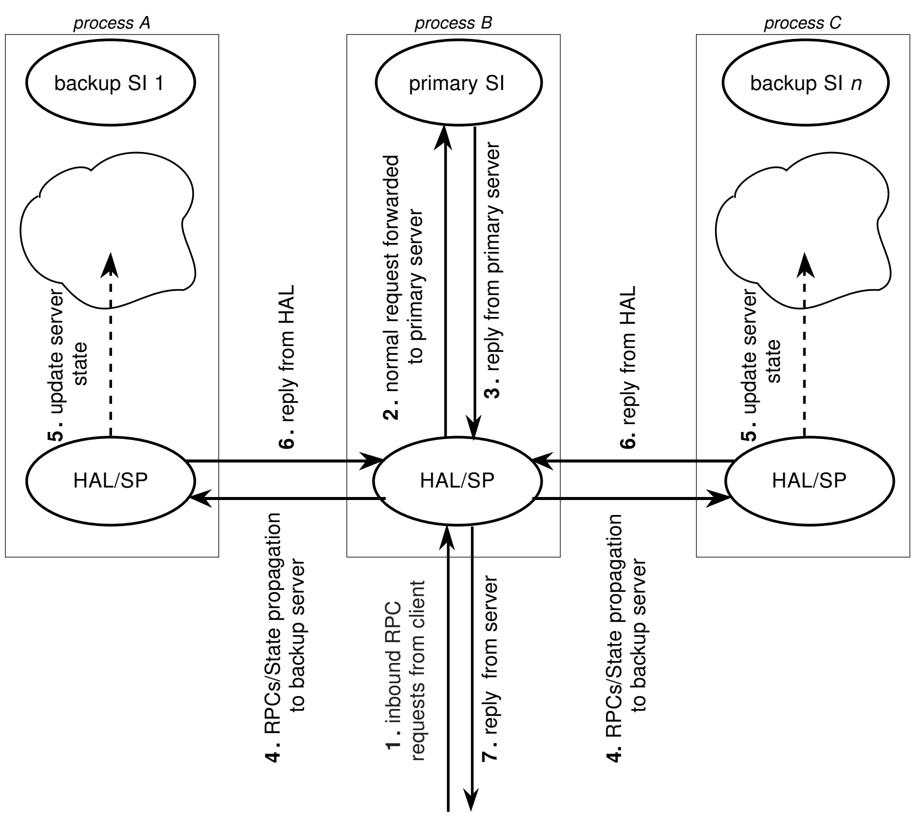

Figure 3 — Communication Between HALs, Primary and Backup Servers

Support for high availability is added through a high-availability layer (HAL) that is linked with the proxy object at the primary and each of the backup (see Figure 4). The HAL intercepts all remote procedure calls intended for the object as well as those originating from the object. The HAL encapsulates all functions related to providing high-availability and is completely transparent to the application.

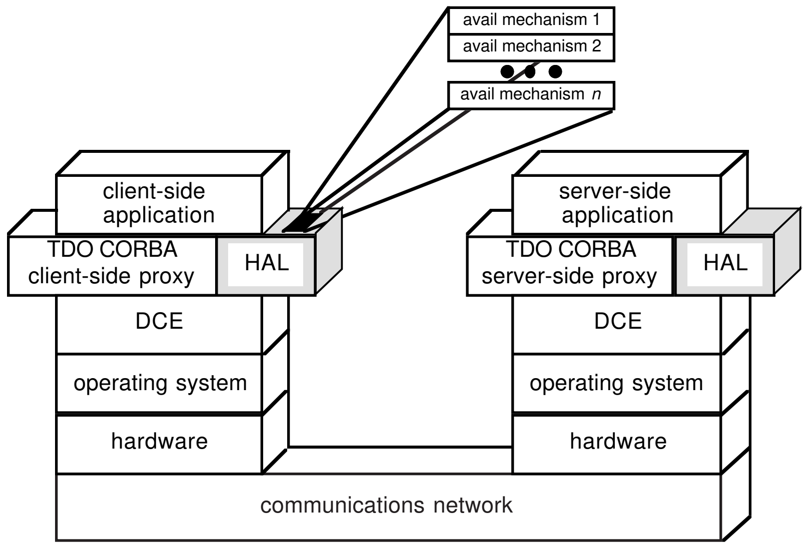

Figure 4 — The Architecture of a Highly Available Application

The SP for the primary server object exports its interface to the global name space, enabling client proxies to bind to the replicated object. The HAL at the primary object also creates an RPC group in a special subtree of the CDS space which contains the names of the primary server and the backups. This naming scheme allows the primary and backup proxies to communicate among themselves at the HAL level to support high availability, but is completely transparent to client objects. Figure 5 shows how a highly-available server, `my_app`, would be registered in the CDS namespace.

The HAL implements several functions related to high availability, such as failure detection, the election of a new primary after a failure, and the communication of state updates. These functions rely on the existence of the RPC group that was created by the primary. The HAL communicates using remote method invocation following the CORBA standard.

3. EVALUATION

In this section we describe our experiences using CORBA and DCE to build our high-availability toolset. Overall, the facilities provided by CORBA and DCE greatly simplified our implementation efforts. The HALs at the primary and backups use CORBA remote method invocation for communication. We found this layering to be very convenient and simpler than the alternative of performing the HAL-level communications directly through DCE. We initially considered using DCE to get better control of the underlying communications protocol and better performance. We eventually decided against it, however, because the low-level computation and communication abstractions offered by DCE would make our implementation efforts more difficult than working at the object-oriented CORBA layer. Thus, our code was

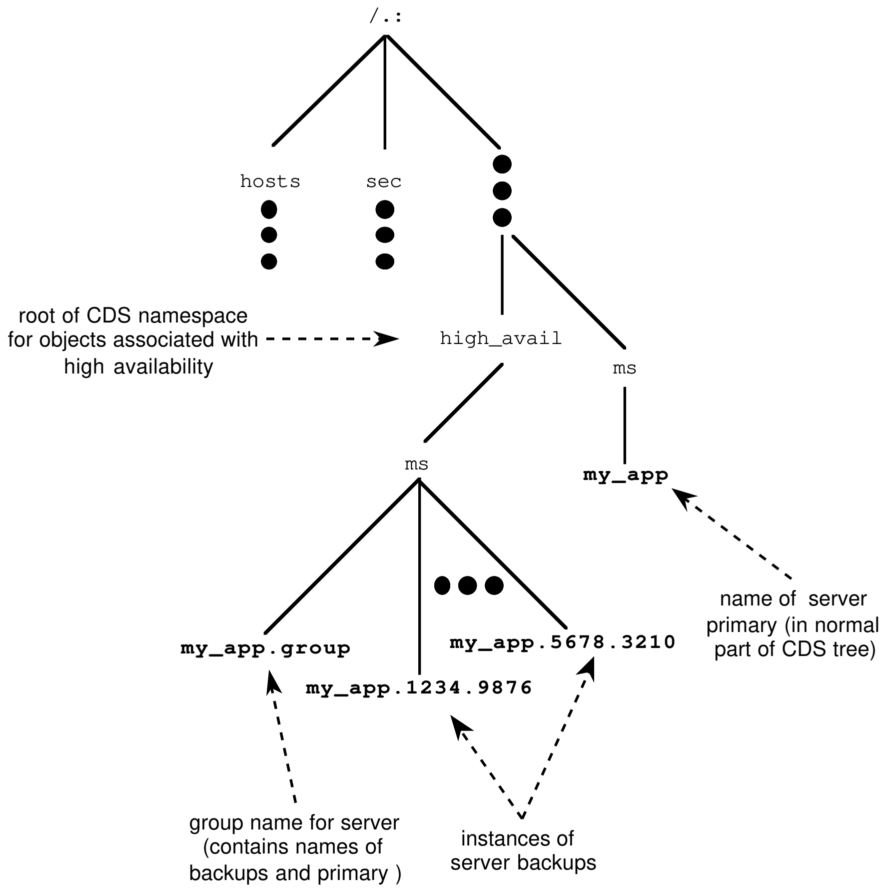

Figure 5 — A Highly-Available Application in the CDS Namespace

operating at a high level which was independent of the hardware or the network protocol. We believe that this choice led to the simplicity of the design and implementation. The naming services made available by DCE, especially RPC group naming, were also very valuable in simplifying the implementation and reducing its complexity. We were also able to use vendor-supplied tools to monitor the CDS name space while our programs were executing, which was extremely valuable for debugging the HAL. The alternative of implementing a name service on top of a lower layer such as Unix sockets or some vendor-supplied name server would perhaps have required more coding and would not have been useful during debugging. DCE's excellent integration of RPC, threads, and exception handling provided a good foundation for building the CORBA layer as well as the HAL.

Besides these positive facts, our experience also outlined several problems with the CORBA and DCE standards. Some of these problems are specific to the nature of providing high availability, while others are more fundamental. We discuss these problems in the remainder of this section. We have been careful to isolate the problems that were merely bugs in our implementation or related to our environment. We do not discuss these problems here.

3.1. CORBA Standards Issues

The TDO CORBA platform we used was implemented locally and provided a subset of the CORBA standard and the likely C++ language mapping as they existed when TDO was written (early 1994). The first problem that we found with the CORBA standard is that the Interface Definition Language (IDL) restricts arrays to be of fixed size. This is fundamentally restrictive for general programming problems and ignores a wealth of experience in distributed computing which shows that communicating variable-sized data is a very common operation (e.g., the implementation of TCP/IP [Postel81]). In our case, this restriction proved costly when the primary object was transmitting state updates to the backups. Since it is impossible to predict at compile time the amount of state changes that would result from a remote method invocation, we had three alternatives. The first is to pick a large array size, which would force the transmission of useless data over the network for small updates, causing a lot of overhead. The second is to pick a small array size and perform several method invocations until the state update transfer is complete. This alternative also causes a lot of overhead in the form of repeated interrupts, and complicates the code because of the need to handle partial updates in the presence of failures. The third alternative was to export several methods for state transfer, each having a different array size and use the method with the most effective size during run time. This alternative reduces the overhead, but it complicates the implementation both at the sender and receiver. None of these alternatives is satisfactory (we currently use the second one). It may be argued that this problem can be addressed by using CORBA's `sequence`, a data structure type that does not have limitations on the size of the data to be transmitted. This data structure, however, is designed to support complex data structures such as trees and lists, and a typical implementation will have to perform marshalling and unmarshalling on transmission and receipt, respectively. Sending a variable-sized array as a `sequence` will thus incur unnecessary performance overhead. We believe that variable-sized arrays are essential and should be added to the standard.

The second problem that we found with the CORBA standard is the lack of specifications concerning threads and exception handling. CORBA-compliant objects are multi-threaded by default, yet the CORBA standard has intentionally omitted any specification related to the interactions between threads, exceptions, and the programming language. These problems are likely to produce CORBA implementations that differ in the way they handle threads and will create problems in porting applications among heterogeneous platforms, which is against the spirit of ODP. Perhaps a better way to solve this problem is to follow the approach used by DCE, which has integrated communication, multi-threading, and exception handling to some degree. We believe similar integration strategies should be added to the CORBA standard.

The third problem that we encountered was relatively minor. CORBA IDL does not have a specification for 64-bit data types. On a machine like the DEC Alpha, CORBA's `long` data type is actually mapped to a 32-bit quantity which is different from the machine's concept of a `long` variable, which is a 64-bit quantity. It is not clear how this problem could be solved, though. Supporting 64-bit types would make interoperability between 64-bit and 32-bit machines difficult. On the other hand, by only supporting 32-bit quantities, application programmers writing code on 64-bit machines should be aware of the difference, which may require additional marshalling and unmarshalling, and exposes the network issues at the application level. DCE, which already provides a 64-bit integer type (`hyper`), faces the same problem. On a 32-bit machine, `hypers` are converted to structures which makes even simple arithmetic operations (e.g., addition) difficult to implement.

3.2. RPC Groups in CDS

Our toolset uses CDS RPC groups to implement a scheme for keeping track of the backups and primary of a replicated object. The CDS RPC group mechanism allows a set of servers registered with CDS to be associated with a single group name. The documentation states that updates to a group are to be propagated consistently to each node that caches a copy of the group's list of names. However, the documentation is vague about how soon updates to a group should be propagated to all cached copies in the network, and what happens if nodes crash while a group update is being propagated. We found this vagueness to be a problem for three reasons.

First, each copy of the replicated object depends on the CDS to know the identities and locations of the other currently running copies of the object. This information is vital, especially when a primary process dies and the replicas have to decide who will become the new primary. The replicas must ensure that only one primary is chosen. As replicated objects die or are newly created, the RPC group in CDS containing information on the replicated copies changes. Since group lookups at each node go to the local CDS cache at that node, replicated objects can have an inconsistent view of the currently active set of objects due to DCE's loose cache coherency semantics.

Second, implementations of DCE adopt lax interpretations of these specifications to improve performance. For example, deleting a name from a group takes from an hour to a day to be propagated to all cached copies in one implementation, while it is propagated every 12 hours in another. Such implementations assume that changes to the name space are not frequent, and therefore they are not suitable for supporting the type of mechanisms that we implemented.

Third, if a node that is running the primary CDS or a cached backup copy of CDS crashes during a group update propagation, some CDS caches could be properly updated, while some may be incorrect. Even worse, we don't know when (or if) the CDS caches will be made consistent. Since the semantics of the group naming during failures is not specified, we cannot rely on this technique as a basis for our toolset in the long run.

We were able to work temporarily around the cache coherence problem by patching the DCE object code to force lookups of group members to go directly to the central CDS database, thereby bypassing the local cache. A permanent solution to all of these problems is to build a separate name server that is optimized to maintain a more consistent view of group membership. This name server could be integrated into CDS via the CDS junction mechanism, which would allow adding high availability to CDS without modifying CDS itself.

3.3. Proxies and Multicast

The primary/backup scheme discussed in Section 2.2 as well as other software-based mechanisms for high availability could take advantage of a multicast capability for propagating updates and synchronizing replicas. We are only interested in a multicast that provides FIFO, reliable message delivery in the presence of communication failures. We don't believe though that a multicast protocol with a sophisticated ordering semantics would be useful for our purposes. Because our TDO CORBA/DCE programs are multi-threaded and therefore may execute in a different order on different machines, the receivers will have to implement a mechanism to ensure that the different interleaving of thread executions would not violate consistency among the replicas. The end-to-end argument in system design then suggests that a sophisticated multicast ordering would not be very useful for our purposes. Currently, the closest thing to a multicast that DCE provides is unreliable broadcast RPC execution semantics to processes on the same local-area network, which is not useful for our purposes because we need reliable delivery to a specific group of processes.

We implemented our multicast primitives by explicitly coding the multicasting in the HAL. While this approach is inefficient, it hides complexity from the application programmer and allows us to experiment with multicasting in our toolset. If future versions of DCE or a CORBA-compliant platform provide multicast in a manner that can take advantage of multicasting network hardware, the efficiency of our toolset will increase dramatically.

3.4. General Observations on DCE

Perhaps the biggest criticism we have for DCE is its enormous complexity. To write even a small DCE program requires initially climbing a steep learning curve. If DCE is going to be used to build large systems, tools must be provided to abstract the enormous amount of details involved in order to make it easier to use. Based on our TDO experiences, we can say that a well-designed layer over DCE can hide many of its complexities and make it more manageable for new application developers.

Another significant problem we encountered with DCE is that its implementations are not robust. While constructing our toolset, the DCE implementations we used crashed frequently. These crashes made building our toolset more difficult, and would also increase the difficulty of building highly-available applications. Because the crashes occurred in different DCE implementations, we attribute these crashes to the complexity of the DCE standard, which makes the development of a robust DCE implementation difficult.

We have also observed that different DCE implementations behave differently. A stated advantage of ODP in general, and DCE in particular, is application portability across heterogeneous computing platforms. Common behavior across platforms is critically important for building a portable high availability toolset. We attribute these problems to the imprecision of the specification of many aspects of DCE, especially when it comes to defining behavior during failures. For example, server-side exceptions are supposed to be caught and possibly propagated to the client. The integration of communication and exception handling is an important feature of DCE, but these capabilities behave differently on different platforms. This is disappointing given the elaborate specification of the standard.

4. Conclusions and Future Work

We have described our experiences in building a toolset to add high availability to distributed objects on a CORBA-compliant platform which, in turn, runs over DCE. The toolset allows application developers to create highly-available applications without having to worry about the underlying technical details of failure handling and recovery. The toolset provides a variety of mechanisms for increasing program availability that can be selectively incorporated into a program, depending on its needs.

We built the toolset over CORBA and DCE because we believe these technologies will be used to create many kinds of distributed applications in the future. Both CORBA and DCE hide many low-level inter-process communication details and provide facilities to make writing distributed programs easier. By using these technologies, we were able to build on their strengths and add capabilities to provide high availability to those applications that require it. Specifically, we found that building the HAL using CORBA was very convenient, and the high-level view of the network as a set of distributed objects has greatly simplified our implementation effort. We also found the naming and RPC group services of DCE to be very valuable.

Although CORBA and DCE offered many benefits to our project, we also discovered a number of problems with both standards that introduced additional complexity in our implementation. The CORBA standard for instance does not allow the transmission of variable-sized arrays over the network, something that we believe very essential for distributed applications. We also struggled because of the lack of specification of the interactions of exception handling and threads, especially during failures. These led to many ambiguities that are likely to be handled differently by different implementations of the standard. The CORBA standard should precisely define the behavior of CORBA-compliant objects to resolve these ambiguities, while maintaining the overall simplicity of the current standard and avoiding the complexity and overloading that characterize the DCE standard.

For DCE, we were disappointed that the specifications were imprecise in many situations, especially concerning the behavior during failures. The specifications were also lacking in defining what happens to CDS during failure. This imprecision led to implementations of DCE that behave differently, which is against the spirit of open distributed processing. We also believe that the lack of multicast support and the complexity of the standard are going to be serious problems that researchers and developers will have to face when building distributed applications. We also observed that even though DCE implementations are sold as finished products, they still have many bugs in them, which we attribute to the complexity of the standard that makes it very difficult to build robust implementations.

In spite of these criticisms, we believe that CORBA and DCE can form a platform that offers a good potential for building many kinds of distributed systems. Over time, we hope the problems we described will be resolved by the suppliers of CORBA and DCE products, as well as by the appropriate standards organizations. As people attempt to use these systems to build tools and applications of significant size, more problems will be discovered. As long as there are ways for people working “in the trenches” to affect the design of CORBA and DCE, these systems will continue to evolve into better tools for building large applications.

5. ACKNOWLEDGMENTS

Glen Diener implemented TDO and provided helpful suggestions on its use. Peter Bates, Gita Gopal, R. C. Sekar, and Marc Shapiro gave us many helpful comments on earlier versions of this work. Finally, the authors would like to thank the anonymous reviewers for their comments, which helped clarify a number of technical details and dramatically improved the readability of the paper.

REFERENCES

[Abbott90] Abbott, R., “Resourceful Systems for Fault Tolerance, Reliability, and Safety”, *ACM Computing Surveys*, 22(1), March, 1990.

[Birman93] Birman, K., “The Process Group Approach to Reliable Distributed Computing”, *Communications of the ACM*, 36(12), December, 1993.

[Diener94] Diener, G., “Touring Distributed Objects”, Internal Bellcore Memorandum, June, 1994.

[Elnozahy93] Elnozahy, E. N., *Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication*, Ph.D. Dissertation, Rice University, October, 1993. Also available as Technical Report 93-212, Department of Computer Science, Rice University.

[Elnozahy95] Elnozahy, E. N., V. Ratan, and M. Segal, "A Toolset for Building Highly-Available Software Systems Using CORBA and DCE", in preparation.

[Gray91] Gray, J., and D. Siewiorek, "High-Availability Computer Systems", *IEEE Computer*, 24(9), September, 1991.

[Huang93] Huang, Y., and K. Chandra, "Software Implemented Fault Tolerance: Technologies and Experiences", 23rd Symposium on Fault Tolerant Computing, Toulouse, France, pp. 2–9, June, 1993.

[Millikin94] Millikin, M., "DCE: Building the Distributed Future", *Byte*, 19(6), pp. 125–134, June, 1994.

[OMG91] —, "The Common Object Request Broker: Architecture and Specification", Object Management Group, TC Document Number 91.12.1, Revision 1.1, December, 1991.

[OSF91] —, "Introduction to OSF DCE", Open Software Foundation, 1991.

[Postel81] Postel, J., "Internet Protocol", Internet Request for Comments RFC 791, September, 1981.

[Schneider90] Schneider, F., "Implementing Fault-Tolerant Services Using the State Machine Approach", *ACM Computing Surveys*, 22(4), December, 1990.

[Segal93] Segal, M., and O. Frieder, "On-the-Fly Program Modification: Systems for Dynamic Updating", IEEE Software, pp. 53–65, March, 1993.

[Vinoski93] Vinoski, S., "Distributed Object Computing with CORBA", *C++ Report*, pp. 34–38, July–August, 1993.