Decentralising Distributed Systems Administration

Christine Hogan – Synopsys, Inc.
Aoife Cox – Lockheed-Martin, Inc.
Tim Hunter – Synopsys, Inc.

ABSTRACT

Nowadays, system administration most often involves maintaining a collection of distributed, interoperating machines. The manner in which this task is carried out, however, is usually more reminiscent of a centralised computing model, with a small number of machines playing host to all of the critical system services, and constituting common failure points for the entire distributed system.

In this paper, we argue that the adoption of a distributed approach to administration of these systems is not only more natural, but can also be shown to have many practical benefits for the system administrator. In particular, we show, by example, how distributed object technology, as reflected in the CORBA (Common Object Request Broker Architecture) standard, can be used to construct a distributed administration framework, tying together services and servers on many different nodes, bringing some of the advantages of distributed systems to the systems administrator.

Introduction

As we have progressed from the age of centralised computing to that of distributed computing, so has the task of the systems administrator had to evolve from maintaining a system with perhaps one or two large servers to maintaining a large network of smaller workstations, and a number of CPU and disk servers. However the mode of operation used by system administrators themselves still tends to be that of the centralised model.

In many large sites there are one or two machines on which the rest rely absolutely for a number of services. For example, at Synopsys, a single machine is the NIS master, the mail server, an ntp time server, a boot server, the SecurID server, the console server, DNS master, NAC administrative host, and runs pcnfsd. If something goes wrong with this server it results in many and varying failures all over the network. We believe that the primary reason for the continued use of this centralised model of administration is the lack of adequate software support for decentralisation of the system administration task.

In this paper we examine the philosophies of distributed computing [1] and, in particular, distributed object technology [2] and see to what extent they could be usefully applied in the development of a systems administration toolkit. In particular, we show how the infrastructure provided by a distributed object technology, such as CORBA [3], could be used to create a framework that ties together the pieces of the existing system administrator's toolset to form a single decentralised distributed toolkit. In addition, we relate our proposal to the ongoing standards efforts, specifically the Object Management Group's proposals for the System Management CORBAfacilities [4] in the set of Core Facilities for CORBA-compliant platforms.

Background

In this section we introduce some terminology that will be used in the paper and provide some background on the research that is being performed in the field of distributed object technology. Initially we discuss each of the components, object technology and distributed systems, separately, and then examine the area that combines the two.

Object Technology

In object technology an *object* is frequently defined as a representation of a real-world entity, which has state, behaviour and identity. For example, an object could be used to represent a user. The structure of an object is generally specified using classes, where a *class* can be viewed as a template for a set of objects having a number of characteristics in common. A class will define the data that determines the state held by an object of that class, together with the operations that can be performed on that data.

There are many advantages in, and motivations for, using object technology as a systems modeling paradigm [5]. It offers a more natural way to model real-world problems through describing the behaviour of each entity in the system and the interactions between those entities.

Object-oriented design paradigms encourage modularity of code. Encapsulation of internal data

structures into objects provides the separation of interface and function from implementation, making code more maintainable and easier to enhance. Modularity and encapsulation also yield flexibility and extensibility due to the design and development of independent modules that are combined together to solve a problem. Object-oriented design allows for incremental growth of a system without major re-design, through the modularity and composibility of the components, and the ability to extend the components and the services provided by a class through the use of inheritance and polymorphism. System administration tools can also benefit from these features of object-orientation.

Distributed Systems

In this paper, when we refer to a distributed system, we mean a collection of loosely coupled processors, such as a network of workstations [1]. Some of the key advantages of distributed systems are that they provide the potential for incremental growth, load balancing, fault tolerance, high availability and reliability through replication of services. Other advantages of distributed systems include resource sharing, and new possibilities in the area of Computer Supported Co-operative Work (CSCW).

Incremental growth means that as new technology and machines become available, they can be added to the network, and obsolete machines can be removed, without any great difficulty.

With a distributed system it is also possible to distribute the load of service providing among a number of machines, spreading the load between them, and not relying on a single overloaded server. Load-balancing in this way also facilitates the replacement of servers with newer, faster machines, since only one or two services need to be rolled over to the new machine, rather than five or more. The use of a distributed system makes the maintenance of a set of distributed servers and the roll-over process easier.

Some system services, such NIS [6] and DNS [7] already implement replication for the reasons that were mentioned above. However, the replication is built into each of them separately. They neither provide nor use an infrastructure which a system administrator can conveniently utilise in order to implement replication of other services.

Distributed systems also introduce problems of their own, however, such as latency, security and the traditional lack of software. Latency is inherent in the nature of a loosely-coupled distributed system. No networking hardware today is as fast as a bus connecting two processors. Security issues include authenticating access from remote machines [8] and other people on the network eavesdropping on potentially sensitive data [9]. There is a lot of research being performed in the area of security, and some distributed systems, such as OSF's Distributed Computing Environment (DCE) [10], have very strong network security [11]. Similarly, there has been considerable research in the area of distributed operating systems over the past years, and the products that have traditionally been available do not particularly simplify the task of distributed programming. The lack of software to simplify the task of distributed programming is a well recognised problem [12]. One of the most essential services that a distributed system can supply is a location service [13]. A location service can be used by server programs to advertise themselves, and by client programs to locate the services that they require.

Recent advances in application-level software have mainly been in the area of distributed object technology, which is reaching maturity and acceptance with the release of the CORBA 2 standard and the development of the common facilities (CORBAfacilities) architecture [4]. Implementations of the CORBA standard are among the first distributed programming solutions that provide a programming interface at a high enough level to be useful.

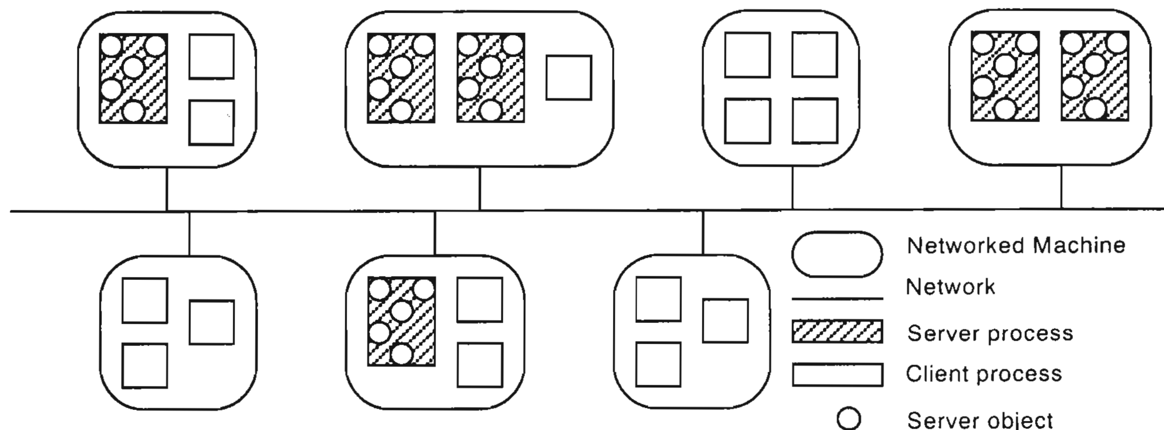

Figure 1: A distributed object support system

Distributed Object Technology

In this section we introduce the concepts behind and motivation for distributed object technology. We also mention the ways in which distributed systems and object technology have been combined in CORBA [3] and the Object-Oriented Distributed Computing Environment (OODCE) [14]. Figure 1 illustrates a distributed object-support system. Communication occurs between a client and zero or more servers, by way of object invocations. The servers with which a client communicates can reside on the same machine as the client, or a remote machine. With standard object-oriented languages, all of the objects must reside in the same address space to be able to access each other. A distributed object support system, such as those defined by the CORBA standard, provides either an interface definition language or language extensions that support the notion of distributed objects. Object can be anything from fine-grained language-level objects, to processes, to physical components of the system, such as printers. Objects should be uniformly accessible by any component of the distributed system in a transparent, location-independent manner.

In the section on distributed systems, we described how services could be distributed throughout the network using standard distributed systems technologies like DNS [7] and NIS [6]. We noted that a location service could provide a more flexible environment. Standard distributed systems architectures also lack the advantages inherent in the use of object technology, as outlined in the section "Object Technology". A distributed object support system, such as a CORBA-compliant system, combines the advantages of object technology with a distributed environment. It also provides the flexibility of a location service.

CORBA

In this section we provide a brief introduction to CORBA. Initially we describe the evolution of CORBA, and then provide an architectural overview. Finally, we briefly answer the questions of language support and availability.

Evolution

A number of different companies produced software to aid in application interworking, such as Sun's Tooltalk, Microsoft's OLE and Hewlett-Packard's SoftBench. In an effort to create a standard in high level application interworking, including across multiple platforms and network architectures, the Object Management Group (OMG) was formed, and produced the Common Object Request Broker Architecture (CORBA) specifications [3]. The CORBA specifications describe a messaging facility for a distributed object environment: a mechanism whereby each object in the environment has a standard way of invoking services of other objects in that environment.

Architectural Overview

The CORBA architecture, as specified by the OMG, is comprised of three main elements. These elements are an Object Request Broker (ORB), an Interface Definition Language (IDL), and a Dynamic Invocation Interface (DII).

The Object Request Broker (ORB) is a fundamental service that enables messaging between objects in centralised or distributed systems. The ORB handles the details of all communications between clients and servers, irrespective of the language in which they are written, or the platform on which they reside. Conceptually, the ORB handles the passing of requests from a client to a server and passing the results back.[1] The Interface Definition Language (IDL) is used to specify the interface to a given object. If a client has a handle to an object and knows the IDL interface that the object supports, the client can invoke a method on that object through the ORB. This form of invocation uses the object's static invocation interface.

The Dynamic Invocation Interface (DII) is used where the client does not know the interface to a server object in advance. The DII can be used to formulate requests at runtime. A server object will not be aware that an incoming request was performed using the dynamic interface, rather than the static one.

Given this specification a number of implementations of the CORBA standard, and in particular the ORB, are possible. For example, Orbix, from Iona Technologies, implements the ORB through three components. These components are a client library, a server library and an Orbix daemon (`orbixd`). This implementation is discussed in more detail in the section "Using CORBA". An alternative implementation might not have a daemon process, but rather implement everything in the client and server libraries.

Languages

The IDL interface definitions are compiled into a high-level language like C or C++. Other language options are available, but their availability depends on the CORBA implementation that you are using. There is not, to the authors' knowledge, an IDL to Perl compiler available, nor any compiler that will translate IDL into a high-level scripting language, such as those that are commonly used by system administrators.

However, scripts that implement the task that the system administrator wishes to incorporate into a distributed architecture, based on CORBA, can be called from the C++, or equivalent, wrappers.

[1]The handling of message passing may be implemented in separate client and server libraries, but conceptually this functionality is in the ORB.

Invoking the scripts in this manner is the way in which we envisage the CORBA architecture being utilised in the development of system administration tools.

Availability

There are a number of different implementations of the CORBA specifications commercially available. These packages include Orbix from Iona Technologies, ObjectBroker from Digital Equipment Corporation, and DOE from SunSoft. All examples and implementation-dependent details in this paper are based on Orbix.

To the authors' knowledge, there are currently no free implementations of CORBA on Unix.

Application to System Administration

In this section we address the practicalities of how we can utilise the infrastructure provided by CORBA to do the hard work of distribution. We describe how you can link existing programs or scripts into this infrastructure, with the gains being simplicity, ease of operation, flexibility, and the potential for higher reliability and availability.

Simplicity and ease of operation are due to being able to run the front-end client code on any machine, irrespective of the machine, or machines, on which the server code is run. Thus all of the tasks that are linked into the infrastructure provided by CORBA can be accomplished without having to log in to the servers themselves. This architecture also provides the flexibility to move services without having to alter the client code at all. The service is located by the CORBA location service, transparently, at run-time. Higher reliability and availability can be achieved through distribution of services over a greater number of machines, and through the potential for duplication of services.

Using CORBA

To develop a CORBA-based application, using, for example, the Orbix software described previously, requires development of client and server programs. Machines that are going to host your Orbix servers must be running an instance of the Orbix daemon. Servers are registered with the daemon. This registration communicates the presence of the server, and the command line parameters needed to launch it, to the daemon. At this point client applications can access the server by getting a handle to it through the location service.

Server Implementation

A server interface is written in IDL. Some sample IDL code is provided in Appendix 1. Skeleton C++ classes corresponding to the IDL interface(s) are then generated using the IDL compiler that is supplied with the product. The developer then supplies the method bodies for each method in the interface. This process is depicted in Figure 2.

The top-level routine of the server (i.e., `main()`) creates objects as required, and notifies the Orbix daemon that it is ready to receive incoming requests. The daemon listens for incoming requests for any of its servers and launches the appropriate server. The server executes the

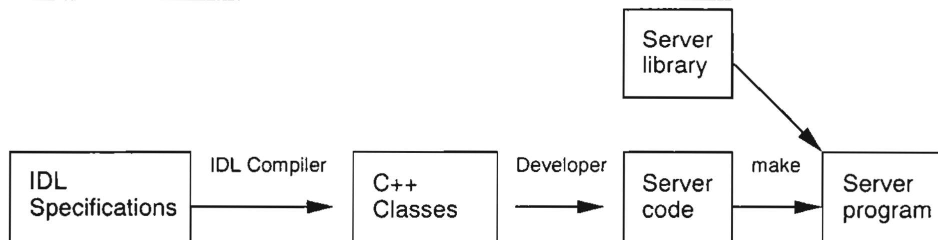

Figure 2: The development process from IDL to server program

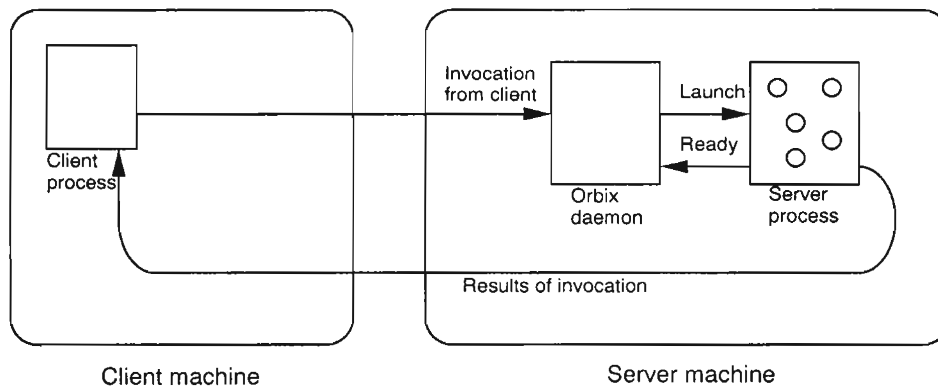

Figure 3: Launching a server

invocation request and passes the results back to the client. This process is depicted in Figure 3. The Orbix daemon acts in a similar manner to the portmapper here. Note that when a server registers itself, it registers a name for itself that the client can use when it is trying to locate an object provided by that server, and that a single server can register more than one object. Servers can also choose one of two different invocation methods. One of these invocation methods is to have a single instance of the server handle all incoming invocation requests. The other form of invocation of the server involves an instance of the server being created for each incoming method invocation, so that they are handled in parallel in different processes. This decision is made by the server when it registers itself the Orbix daemon.

The server is linked with a server library that implements the communication between the server and the daemon, including this notification of readiness. The server library also implements the communications between the server and its clients. This communication is over TCP/IP with XDR encoding.

Client Implementation

A client is linked with the client library, which handles all communication between the client and server, and between the client and the daemons. The initial communication performed by a client involves locating the server objects with which it wants to interact. Server location is performed through the Orbix `bind` operation. Some sample client code is provided in Appendix 1.

When a client calls `bind` it supplies a number of arguments to tell the ORB what kind of server object it is looking for. For example, the client can specify the name of the server process, the server object name, the interface that the object provides and the machine on which the server should reside. In other words, the client can choose to bind to a specific object associated with a named server on a particular host. If the client either doesn't know, or doesn't care about the host on which a server resides, it can specify the server name and the object interface. Equally, the client can omit the server name. If the host name is left unspecified, Orbix consults a number of its configuration files. These configuration files specify what hosts have daemons running, and the order in which to check them to find an object supporting the specified interface.

Once a client has successfully bound to a server object, it can perform method invocations on that object as if the object was in the client's own local address space. The client actually has a *proxy* [15] object in its address space, that represents the remote object, and provides the same interface as the remote object. Invocations on this object are transparently passed to the remote object in the remote server process. The library software takes care of the details of passing the invocation parameters to the server and returning the results back to the client.

Simple Operations

If the program, or script, that you want to link in to CORBA is a simple program that is supplied a series of arguments, and runs to completion, with perhaps some output along the way, then linking it into a CORBA system is simple.

The implementation involves writing a server that calls the script with the arguments that are supplied to the method call. The client program would pass the arguments that are supplied on the command line as arguments to the method call on the remote server object. In this simple case, there is little gained by using CORBA rather than `rsh`, except the ability to move the server transparently to the application, and the ability to provide backup servers that will transparently get called in the absence of the primary server.

Complex Operations

A complex operation is one that involves several servers on a number of different machines. An example of a complex operation is the creation of a new account.

In Synopsys, Human Resources generate an incoming form for each new hire that tells each of the departments the information they need to prepare for the new hire. In particular, we in Network and Computing Services (NCS), get the information necessary to create the account and to order and install equipment. The account creation is automated through a Perl script that retrieves and parses the incoming form.

At the moment, this script must be run on the overloaded server mentioned in the Introduction, for a variety of reasons, including our trust model. The script goes through a series of steps. The incoming notification, with all the details, has to be retrieved from one server. An entry for the new user needs to be created in the NIS maps on the NIS master. A home directory must be created for the user on whatever server is appropriate for their group. They will need to be added to a variety of email lists, which may involve a series of files on different machines. Also, there may be some special requirements, such as a system administrator being added into the call tracking system, or access to a database, which requires an account on another machine.

To implement a complex operation, such as the one described above, under CORBA, there is a server process on each of the machines that can be involved in the operation. Thus there would be a server process on each of the home directory servers, on the NAC administrative host, on the NIS master, on the database machine, on the machine that controls the call tracking system, on the machines that

house email lists that may need changing, and on the machine that supplies the incoming forms.

The servers should each represent one logical service. If, for example, NIS and the call tracking system reside on the same machine, they should be implemented as two separate server objects on the same machine, to facilitate moving one of the services to another machine. There may be more than one server on a single machine for a given complex operation. The client will then bind to each server object that it needs, and can invoke the operations in the appropriate order, independent of where the servers reside.

Implementing an operation like the one described above under CORBA would be advantageous in our environment, because it would obviate the need to run a large, complex script on an overloaded server. Each component of the script would be run on the relevant machine, and the script would not have to be changed if any of the services are migrated around the system. Nor would service migration necessitate a change in our trust model.

Interactive Scripts

Interactive scripts present a greater challenge, because a series of interactions between the client and the server side need to happen, with information being passed in both directions. In CORBA, communication between the client and server takes place when the client invokes a method on a server object, or the server returns the results of a method invocation to the client.

Thus to implement an interactive script within CORBA, the client needs to be more complex than a simple call, or a series of simple calls to remote services. Separate operations that return results must be identified, and implemented as remote method calls. The client contains the logic and the interaction. Thus interactive scripts require a greater redesign than non-interactive ones to be incorporated into a distributed architecture.

Caveats

In this section we discuss two caveats in the use of an infrastructure of this kind to provide system administration facilities. The first of these caveats relates to the use of persistent object support when implementing a system administration task. The other caveat relates to security issues involved with providing servers that perform system administration tasks in response to requests from the network.

Persistence

Many distributed object support platforms provide persistent object support [16] [17]. A persistent object is one that has a lifetime beyond that of the programs that access it. In these systems, application-level objects can be stored in persistent store, such as on a disk, when not in use and thus maintain their state between invocations of an application. They also survive system restarts. Persistence is not a fundamental component of distributed object support, nor of CORBA, but it can be a useful feature for many applications. It may also superficially appear to be useful for system administration applications.

For example, the system could keep a persistent database of what account(s), if any, each person had on each machine in the distributed system, along with all other electronic resources that the individual used, including mailing lists, call tracking systems and database access. This information would be stored in the persistent object associated with that individual. When the "remove user" method was invoked on that persistent object, all the information would be immediately available, which would make it simple to write the script to perform the deletion.

However, we believe that it would be a mistake to use persistence to store system state. Consider what happens when the state of the system is modified either manually, or by a program or script that does not update the persistent state of the objects that represent the altered system state. Worse still is a scenario in which the persistent state of an object gets corrupted, but is still used to determine the behaviour of the machine in some way.

We came to the conclusion that it must be possible to fix the state of the persistent store so that it reflects the state of the standard system files. It should also be possible to do this without rebooting the machine, since the introduction of a new technology should not detrimentally affect the availability of the system as a whole. Thus it must be possible to re-initialise the persistent state of the distributed object system at any time from the system files. Given the need for that feature, and the possibility for conflicting updates due to the state of the files being changed without a corresponding change in the persistent state, we came to the conclusion that persistence was not useful for this application. We also felt that it introduced extra points of failure into the system, and thus was not only not useful, it would be a mistake to employ persistence.

Security

The standard mode of operation of Orbix, and, we believe, the other implementations of CORBA, is not especially secure. If the Orbix daemon is not run as root, all the servers are clearly launched with the same user ID as the Orbix daemon. In this case, the system administration utilities cannot operate. If the Orbix daemon is running as root, the daemon tries to launch a server with the user ID of the remote user, if that user exists on the system on which the server is to be run. The user ID of the remote user is passed with the invocation request by the client library. The security implications of the

server naively believing information that comes in, unauthenticated, off the network are obvious.

There are hooks for applying filters, including an authentication filter, to servers on a case by case basis. This authentication filter could require some form of strong authentication before allowing a server to be invoked. However, the overhead of implementing such authentication may be sufficiently large that it outweighs any advantages of implementing the service over CORBA. Further work is being performed in the area of security, including the provision of standard security services within the framework of the CORBAfacilities, but it remains to be seen what these will provide.

Related Work

The Object Request Broker, as defined by the OMG, forms a part of an overall Object Management Architecture (OMA), which specifies a model for constructing distributed object applications. The ORB is the key communications element of the architecture. The architecture, in addition, defined CORBAservices (formerly known as Common Object Services) and CORBAfacilities [4] (formerly called Common Facilities).

CORBAservices specify standard interfaces to basic functions commonly required in building (distributed) applications. These basic services include object naming, event notification and transactions. CORBAfacilities specify standard interfaces to functions that are required for building applications both in specific domains and across domains. CORBAfacilities include a proposal for a system management facility. The CORBAfacilities for system management will comprise a set of IDL interfaces, and hence a standardised collection of servers providing system management functionality. Guidelines for possible facilities have been outlined by the OMG. However, the actual interfaces for most facilities, including those proposed for system management have not yet been specified.

Our work does not advocate any standard set of facilities or interfaces. In this paper we merely outlined how we believe the infrastructure provided by CORBA can be utilised as framework for providing distribution for existing administrative scripts and tools.

Conclusions

The infrastructure provided by a CORBA-compliant systems is potentially useful for building a decentralised system administration toolset. However, in the absence of an IDL to Perl[2] compiler, it is unlikely to become a tool that is regularly employed by system administrators. Equally, we would expect the system administration community to have reservations about using it, unless the security issues are resolved, or the administrators individually take the view that it's not that much worse than running NIS.

However, we do believe that the architecture has potential, and that it may be something that is worth watching for in the future. In particular the proposed CORBA security services offer interesting potential for deploying a standard security system across all pertinent applications in an environment. Implementations of these security services, when available, will offer convenient access to the building blocks of security for applications developers and system administrators alike. We believe that the availability of these tools will help promote electronic security.

Acknowledgments

Paul E. provided encouragement, advice and practical help from day one, right through to the end, and for that, and for his tolerance we thank him. Paul A. also deserves a special mention for advice, and a push in the right direction at a crucial time, along with much needed encouragement. Our many proof-readers, and Jeff in particular, were of enormous help in straightening the paper out - our thanks to Beth, Arnold, Ted, Dave and Laura for their help.

Author Information

Christine Hogan is the security officer at Synopsys, Inc., in Mountain View, California. She holds a B.A. in Mathematics and an M.Sc. in Computer Science, in the area of Distributed Systems, from Trinity College Dublin, Ireland. She has worked as a system administrator for six years, primarily in Ireland and Italy. She can be reached via electronic mail as chogan@maths.tcd.ie .

Aoife Cox is a research scientist at the Lockheed Martin Artificial Intelligence Center in Palo Alto, California, where she leads the object management infrastructure team on Simulation Based Design (SBD) - a major ARPA project aimed at supporting distributed concurrent engineering. She holds B.A. and M.Sc. degrees in Computer Science from Trinity College Dublin, Ireland, where she spent a number of years working with the Distributed Systems Group in the Computer Science Department. Her research interests include distributed computing, software reusability and concurrent engineering. She can be reached via electronic mail as acox@maths.tcd.ie .

Tim Hunter is a systems administrator at Synopsys, Inc. His current focus is on remote systems administration. He previously worked as a sysadmin for, and received his degree in Electrical and Computer Engineering from, the University of Colorado at Boulder. He can be reached via electronic mail at tim@synopsys.com .

[2]Or other scripting language.

[1] G.F. Coulouris and J. Dollimore. *Distributed Systems Concepts and Design*. Addison-Wesley, 1988.

[2] Rodger Lea and James Weightman. Supporting Object-Oriented Languages in a Distributed Environment: The COOL Approach. In *Proceedings of the Technology of Object Oriented Languages and Systems Conference*, July 1991.

[3] OMG. Common Object Request Broker Architecture. Technical Report OMG Document 93.12.43, rev 1.2, Object Management Group, Inc., December 1993.

[4] OMG. Common Facilities Architecture. Technical Report OMG Document 95.1.2, rev 4.0, Object Management Group, Inc., January 1995.

[5] Bertrand Meyer. *Object Oriented Software Construction*. Prentice Hall, 1991.

[6] Hal Stern. *Managing NFS and NIS*. Nutshell. O'Reilly and Associates, Inc., 1991.

[7] Paul Albitz and Cricket Liu. *DNS and BIND*. Nutshell. O'Reilly and Associates, Inc., 1992.

[8] Aviel D. Rubin. Independent One-Time Passwords. In *Proceedings of the 5th UNIX Security Symposium*. USENIX, June 1995.

[9] Matt Blaze and Steven M. Bellovin. Session-Layer Encryption. In *Proceedings of the 5th UNIX Security Symposium*. USENIX, June 1995.

[10] Open Software Foundation. *Introduction to DCE*, 1991. Part of licensed DCE documentation.

[11] Rich Salz. Dce. Bay LISA, April 1995. This is the 2nd edition of his LISA VIII talk.

[12] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[13] Aoife Cox. An Exploration of the Application of Software Reuse Techniques to the Location of Services in a Distributed Computing Environment. Master's thesis, Distributed Systems Group, Dept. of Computer Science, University of Dublin, Trinity College, September 1994.

[14] John Dilley. OODCE: A C++ Framework for the OSF Distributed Computing Environment. Technical report, Hewlett-Packard Laboratories, 1994.

[15] Marc Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986.

[16] Vinny Cahill, Seán Baker, Gradimir Starovic, and Chris Horn. Generic Runtime Support for Distributed Persistent Programming. In *OOPSLA (Object-Oriented Programming Systems, Languages and Applications) '93 Conference Proceedings*, 1993.

[17] Roy H. Campbell and Peter W. Madany. Considerations of Persistence and Security in Choices, an Object-Oriented Operating System. In *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information*, May 1990.

Appendix: IDL Definitions and Client Code

```
---- user_accounts.idl ----

//
// This file describes the classes
// associated with user accounts
//
module User_Accounts {
    //
    // If an exception is raised we use
    // this class to return the reason
    //
    exception reject {
        // The exception data
        String reason;
    };
    //
    // This is the interface to the home
    // directory class, with its visible
    // attributes
    //
    interface Home {
        attribute String path;
        attribute long uid;
        attribute long gid;
        // The method - create home dir
        void initialise() raises(reject);
    };
    //
    // Described the interface to the
    // User class - operations and
    // visible attributes
    //
    interface User {
        attribute long uid;
        attribute long gid;
        attribute String username;
        attribute String shell;
        attribute String gecos;
        attribute Home home;

        void initialise() raises(reject);
        void remove() raises(reject);
        void disable() raises(reject);
        void reactivate() raises(reject);
    };
    //
    // Interface to the SystemUser class.
    // It's a specialised case of the User
    // class.  Only the SystemUser object
    // itself can modify the real_user
```

```
    // attribute - clients can't
    //
    interface SystemUser : User {
        // Inherits the interface from
        // User and adds one read-only
        // attribute to that interface.
        readonly attribute User
                         real_user;
    };
    //
    // Interface to the UserDatabase class
    // An instance represents the passwd
    // file.  Operations are called on
    // an instance of this class by User
    // objects to get the passwd file
    // modified.  Some User objects may
    // call it twice - e.g. SystemUser
    // objects, to disable privileged
    // and unprivileged accounts, for
    // example.
    //
    interface UserDbase {
        readonly attribute String
          passwd_file;
        attribute String
          default_encrypted_passwd;
        // Re-read the passwd file
        void reinitialise()
          raises(reject);
        // Disable an account
        void disable(in String username)
          raises(reject);
        // Reactivate disabled account
        void reactivate(in String
          username) raises(reject);
        // Delete a user entirely
        void remove(in String username)
          raises(reject);
        // Add a new user
        void add(in User user)
          raises(reject);
    };
};
---- client.cc ----

// This program runs on the client and
// requests the server to perform
// operations on a UserDatabase object.

#include <user_accounts.hh>

main()
{
    User_Accounts::UserDbase *user_db;
//
// This is where the bind magic happens
// Bind to a server object that has the
// UserDbase class interface from the
// User_Accounts module, that lives in a
// server called ''user_database_server''
// on any machine.
//
    user_db =
        User_Accounts::UserDbase::_bind(
        '':user_database_server'');
//
// Standard C++ exception-handling
// syntax.  TRY something, and CATCH the
// exceptions.
//
    TRY {
        ....
    }
    NONE {
        ....
    }
    CATCH(UserAccounts::reject,
      rej_except) {
        cout << ''reason: ''
             << rej_except->reason
             << ''==>[ignored: n]<=='';
    }
    CATCHANY {
        ....
    }
    ENDTRY
    ....
}
```