1989

Distributed C++ : Design and implementation

Pradeep P. Mansey

Rochester Institute of Technology
School of Computer Science and Technology

Distributed C++ : Design and Implementation

by

Pradeep P. Mansey

A thesis, submitted to
the Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by:

Dr. James Heliotis (Chairman)

Dr. Andrew Kitchen

Dr. Peter Anderson

October 9, 1989

Distributed C++ : Design and Implementation

I Pradeep P. Mansey hereby **grant permission** to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: October 19, 1989

Acknowledgements

This thesis is dedicated to my late father.

I wish to thank my advisor, Dr. James Heliotis for his advice during the various stages of this thesis. I also thank Dr. Andrew Kitchen, member of my advisory committee for his guidance. Dr. Peter Anderson, I wish to thank for his advice and support throughout my graduate study. Thanks to Daryl Johnson, Charles Fung and Paul Allen, working with them, I learned all I know about the Unix operating system.

Special thanks are due to my friend Nagesh Pabbisetty for his helpful suggestions.

Finally, I wish to thank my family and friends for their support and encouragement during difficult times through the course of my graduate study.

Abstract

Distributed C++ is a learning tool developed to investigate distributed programming using the object paradigm. An extension is designed for C++, to enable the use of C++ in programming distributed applications. A user transparent interface is designed and implemented to create and manipulate remote objects on a network of workstations running the Unix operating system. The concept of remote classes is introduced and remote object invocation is implemented over a remote procedure call mechanism.

Computing Review Subject Codes

Programming Languages/Language Constructs (CR D.3.2)

Computer Communication Networks/Distributed Systems (CR C.2.4)

CONTENTS

LIST OF FIGURES

1. Introduction

Object-oriented programming has received considerable attention in recent years as a powerful means of abstracting data and control in the development of software. The object model has a natural flavor to express concepts in software design and efforts have been made to extend the object model for programming in distributed computer environments.

1.1 An Overview of Terminology

The object paradigm can be extended to distributed systems considering the nodes of the distributed systems to be objects, each exporting its functions and communicating with other objects over the network. The terminology associated with these systems is introduced briefly in the following sections.

1.1.1 Object-Oriented Programming

Object-oriented programming is primarily a data abstraction technique, programs being composed of *objects*. A *class* describes the structure and behavior of a set of objects. An object is an instance of its class and is defined by its associated data and the methods of its class. Methods of a class are the operations that can be performed on the data associated with each object of the class and form the interface exported by the class. This isolates the programmer from the implementation details of the object and the programmer need only know what can be done to the object, not how it is being done. Besides information encapsulation,

a key feature of object-oriented programming is inheritance. It allows the programmer to define new objects in terms of existing objects by providing subclassing. An object can thus inherit state structures and behavior from other objects. Inheritance thus, provides a mechanism for refinement and reusability of software.

1.1.2 Distributed Systems

Advances in computer hardware technology have led to the emergence of distributed processing systems. These encompass distributed data, functions, and applications. A distributed system may be a single memory computer system with multiple processors processing concurrently, a set of computer systems managed by a single networked operating system, or a set of autonomous computer systems which are geographically dispersed. The latter is realized by interconnecting multiple computer systems (nodes) using networks. This has resulted in decentralization of tasks traditionally performed by a single computer system, while placing additional burden on the application programmer to write correct and reliable programs executing in heterogeneous environments. High reliability can be achieved through the use of network operating systems managing the resources in the distributed environments. This restricts the desired independent and decentralized nature of the distributed environment. More loosely coupled communication is desired in these decentralized systems. Programming distributed applications in such heterogeneous environments has resulted in the emergence of language level primitives as vital tools in the development and use of distributed

computer systems. The most popular paradigm for programming distributed applications is the *client-server* model. A server is a node which exports a service to other nodes of the distributed system, for example, a file server in a network based file system. The nodes using this service then become the clients in this context. A node of a distributed system may act as a server and/or client, depending upon the context of the application.

1.2 The Thesis

This thesis investigates object based programming in a distributed environment using existing languages that provide facilities for object-oriented programming. Smalltalk, Objective-C, Eiffel, and C++ are some of the object-oriented programming languages, that have gained wide acceptance in academics and industry. C++ [35] was selected as the base language to implement remote object invocation due to its object-oriented features and compatibility with C. An interface is designed and implemented to extend C++ for use in a distributed system.

1.2.1 Scope and Goals

For the purpose of this thesis, a distributed system is defined as a network of autonomous computers (workstations running the Unix operating system) that use a narrow channel (Ethernet) as the primary medium of communication. Distributed applications are viewed, at a higher level of abstraction, as a set of distributed objects communicating with each other.

The concept of a *remote class* forms the basis of this thesis. A mechanism is designed and implemented to define remote classes and create remote objects using C++ and manipulate these objects using the semantics defined for objects local to the system. This provides the application programmer with a simple tool to program distributed applications without worrying about communication and other details. The current implementation is based on the *"stubs principle"*. The compiler implemented compiles class definitions for remote objects into client and server stubs, which together with the user programs form the distributed application. It is operational on a set of AT&T 3B1 and 3B2 computers running UNIX System V on a TCP/IP/Ethernet network. Communication with remote objects is based on a remote procedure call mechanism, implemented using the Unix System V Transport Layer Interface (TLI) library. Key issues addressed in the design of this interface and their implementation are discussed in the following chapters.

2. Review of Existing Systems

Research in the design of distributed systems dates back to the early 1970's. Since then many distributed systems have been designed and implemented. This chapter describes the features of some distributed programming systems which have been developed and are based on the object model.

2.1 Background

Notable among the existing distributed systems designed and implemented are the Xerox Distributed File System [24] and Grapevine [6] at Xerox, the Locus system [30] developed at UCLA, the Apollo Domain [18], the V-system [8] at Stanford, Amoeba [26] at Vrije Univ., Amsterdam, Unix BSD4.2 with Sun NFS [36], Mach [1] at CMU, and Chorus [31] at Chorus Systemes, Paris. Most of these systems have implemented distributed computing at the operating system kernel level.

While a great deal of development has taken place in distributed computer systems, research in language support for these systems has only gained momentum more recently. The two approaches to designing an object-based language for distributed programming are, (1) to design a new language with built in features facilitating distributed programming, or (2) to add features for distributed programming to an existing object-oriented language. The first approach would result in a major project, involving design of a new operating system as well as a language to provide a uniform object based distributed environment. This is a time consuming effort

and beyond the scope of this thesis. Thus, the second approach was adopted to develop a learning tool for distributed programming using the object model. C++ was chosen as the base language due its object oriented features, its availability, and its compatibility with C (and therefore with Unix).

2.2 Object Based Distributed Programming

Several efforts have been made to extend existing languages such as Ada[25], Modula-2 [23], Pascal [34], Smalltalk [4], and C [34] for use in distributed computing. Argus [20], Emerald [8], Avalon/C++ [40], and HERAKLIT [15] are object based systems developed to support distributed programming. Many of the ideas used in this thesis are derived from the implementation philosophies of these languages. The following section briefly describes the features of some of these languages that are based on the object model.

2.2.1 Distributed Smalltalk

Bennett [4] describes the design of a Smalltalk implementation that allows objects on different machines to send and respond to messages. Among the various features of this implementation are user-transparent remote invocation, automatic creation of return references for remote message arguments, incremental distributed garbage collection and support for remote debugging, remote access control, a host independent naming mechanism, object mobility, reactive remote objects, and support for message interaction among heterogeneous hosts. It is based on a message passing scheme, consistent with that of Smalltalk. This makes

communication with remote objects transparent to the user. Distributed Smalltalk provides limited support for object sharing among users. Object names are not system-wide unique and classes and instances must be coresident, thus limiting mobility of objects. The design of a distributed object manager, replacing the object managers on the current Smalltalk systems, is presented by Decouchant [10]. This allows several Smalltalk systems to share objects over a local-area network.

2.2.2 Argus

Developed at MIT by Liskov and other members of the Argus research group [20], Argus is an integrated programming language and system for implementing distributed programs. The language is based on CLU, an object oriented language developed earlier at MIT. Argus provides for long-lived immutable objects with atomic properties based on stable storage. An application program consists of a number of actions that make use of modules called guardians. An action is an atomic activity and is equivalent to a transaction. Guardians are network wide and consist of data objects and processes to manipulate them. Objects in guardians are made crash resilient and communicate through remote procedure calls to handlers, data being passed by value only.

2.2.3 Emerald

Black et al [7] describe Emerald, an object-based language and a distributed run-time system for Emerald that facilitates the construction of distributed programs for a local area network. This language supports abstract data types, a single object model is provided for defining both small (integers, etc.) and large objects

(directories, etc.), and an explicit notion of object location is provided for. Each Emerald object has four components, a unique identity, a representation consisting of the data stored in the object, a set of operations defining the functions that the object can execute, and an optional process operating concurrently with invocations of the object's operations. These objects also have several attributes. A location specifies the node on which the object is residing. The objects are mobile but can be defined to be immutable. Emerald supports concurrency both between objects and within an object. All entities in Emerald are objects and a single semantic model suffices to define them.

2.2.4 HERAKLIT

Designed for efficient development and maintainence of software for distributed systems, HERAKLIT [15] is a general purpose object oriented language with a hierarchic type concept. A special feature of this language is the ability to exchange modules (objects) at runtime without stopping the entire program. Objects have private and public data entries and algorithms as their components. An object can be activated by a synchronous object call (remote procedure call) or an asynchronous object call which leads to independent activities of the caller and the called object. Each object belongs to a particular type, defined by its type interface and type body. The type interface describes the public components. The type body contains the private components, the description of the object activity, and the execution conditions. The syntax of HERAKLIT is similar to MODULA-2.

2.2.5 Avalon/C++

Avalon/C++ [40] is a proper superset of C++, developed using the Camelot distributed transaction processing system at Carnegie Mellon University for implementing reliable distributed programs. Avalon/C++ programs consist of a set of servers, each of which encapsulates a set of objects and exports a set of constructors. Servers communicate by calling one another's operations which are implemented as remote procedure calls, with call by value transmission of arguments and results. Objects may be stable in which case they survive crashes or may be volatile and are lost in the event of a crash. A transaction is the execution of a sequence of operations and is identified with each process. Transaction semantics are provided via atomic objects. A library of built-in atomic types is provided and user defined atomic types can be implemented by inheriting from the predefined types. Recoverable objects that satisfy certain weak properties in the presence of crashes can also be implemented using non-atomic components.

3. Design of Distributed C++

One of the major considerations in the design of Distributed C++ was to implement remote object invocation without modification to the syntax of the language. A stub generator approach was chosen as it did not alter the syntax of object manipulation. The stub generator compiler implemented, translates classes defined for remote objects into client and server stubs through which the client application program communicates with remote objects on the server. This chapter discusses some of the design decisions made for this implementation.

3.1 Objects in C++

The key concept in C++ [35] is a *class*. Classes provide an excellent means of data abstraction. The user can define data and operations to be performed on the data using classes. These are collectively called as members of the class. In C++, objects are instances of classes, each instance defining a unique object. Member functions (methods) of the classes provide an external interface to the objects to manipulate their private data, and the implementation details of these operations need not be visible outside the class definition. By declaring data and methods of a class to be *private* (the default), these cannot be accessed directly by the objects defined by the class. Objects of a class can directly access only members that have been declared as *public*. A third type of member called *protected* member is also available. Declaring a member as protected makes it visible only to members of that class, and those derived from it. Inheritance is provided by using subclasses.

Derived classes describe how the objects of these classes differ from those of their superclasses. No support is provided for object sharing and communication between objects on different machines, or for cooperation between processes on several machines.

3.2 Objects in Distributed C++

Distributed C++ extends C++ to include the concept of a remote class *rclass*, which defines a class on a server machine and exports its definition to all clients on the distributed system. This provides for communication and interaction among geographically remote C++ objects, direct access to remote objects and the ability to construct distributed applications using C++. Figure 1 shows objects of a client application program on machine A referencing objects on remote machine B.

The concept of a remote class is introduced so that the distributed C++ compiler is able to differentiate between local and remote classes in a class definition file. This restricts the transparency of remote classes, but avoids the problem of generating server and client stubs for classes that are not remote to the client. Also, a remote class is defined as a subset of a C++ class with some additional constraints. Thus by identifying a remote class by the keyword *rclass*, no restrictions are imposed on the classes local to a client or server and error checking for remote classes defined can be done at compilation time.

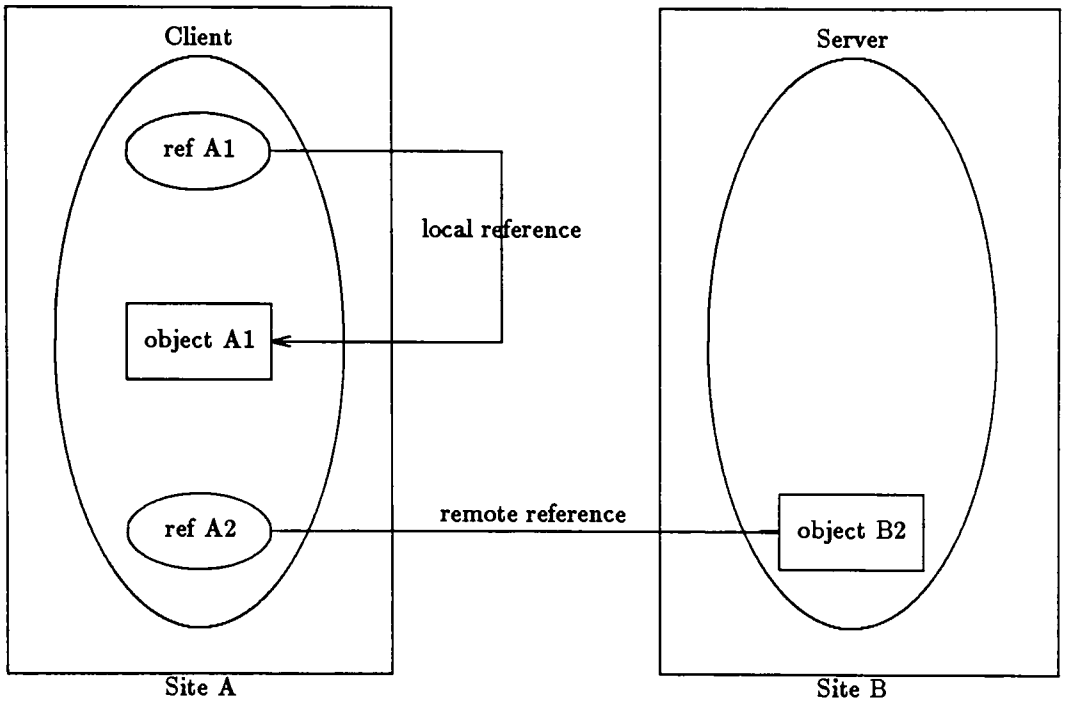

Figure 1. Distributed C++ model

3.3 Design Issues

Several issues have been considered in the design of remote classes. These and their realization in this implementation are discussed in the following sections.

3.3.1 Representation mechanism for remote classes in application programs

Remote classes are defined using the keyword *rclass* in the application programs, using a syntax similar to that of C++ classes. These represent the state structure and behavior of remote objects. This keyword is added to the language to differentiate between local and remote objects. Remote classes can contain private, protected, and public members. Public members make up the interface exported by the remote class and are limited to functions (methods) only. Thus, data of a remote object is accessible only through one of its public methods. This constraint was imposed in order to avoid modifications to the syntax used for accessing

members of objects. In C++ these are accessed using the syntax for accessing members of a structure. In case of remote objects, the objects are not stored on the client machine and data of these objects would have to be accessed across the network, which would have resulted in the change of syntax defined for accessing object members. Public data members could have been allowed, without changing the syntax by which they are accessed, by implementing object migration. In this case if a remote object has public data members it can be transferred to the client machine when it is constructed. Object migration is out of the scope of this thesis, and thus this approach was not considered. Another approach for permitting public data members is to keep a copy of the object on both the client and server machines. This would require an update of both the copies each time the state of an object is modified. This approach consumes data resources on both the machines and it is inefficient to modify both object copies for every operation. An advantage of disallowing public data members is that it enforces the information hiding feature of object oriented languages by requiring that the state of the object be modified only through the operations exported by its class.

Arguments allowed in remote class methods are limited to specific data types permitted by the remote procedure call mechanism implemented. This is discussed in detail in chapter 4. Arguments containing addresses are not allowed in remote class methods and all arguments are passed by value only. This limits the capabilities of remote objects, but these features can be added in future implementations.

3.3.2 Location of remote class definition and instances

Permitting class definitions and instances on different machines makes object migration or duplication of code mandatory. A class definition defines the methods (functions) exported by the class. If an object is instantiated on one machine and its class definition resides on another machine, the object will have to be moved between the two machines to manipulate its state. Another way to permit class definitions and object instances to reside independently is to create copies of the object on both the machines. In this case the object state on both the machines have to be updated each time it is modified. To avoid duplication of code and object migration, class definitions and object instances must be coresident.

3.3.3 Instances of Remote Classes

A predefined class called *Server* is provided for use by the client application programs to create remote objects. Application programs instantiate a Server object which establishes a communication channel between the client application and the server on which the remote objects are to be instantiated. Remote objects are instantiated using the constructors defined for their remote classes. The constructors must be supplied information about the desired location of the remote objects. This is done by passing the Server object representing the desired server machine as the first argument to the constructors for remote classes. When the constructor of a remote class is invoked, an instance of that class is created on the server machine and a proxy object representing this remote object is created on the client machine. The client application references remote objects through the proxy

objects and this is transparent to the user. A client may communicate with several servers at any given time. Figure 2 shows a client application instantiating objects on two remote servers A and B.

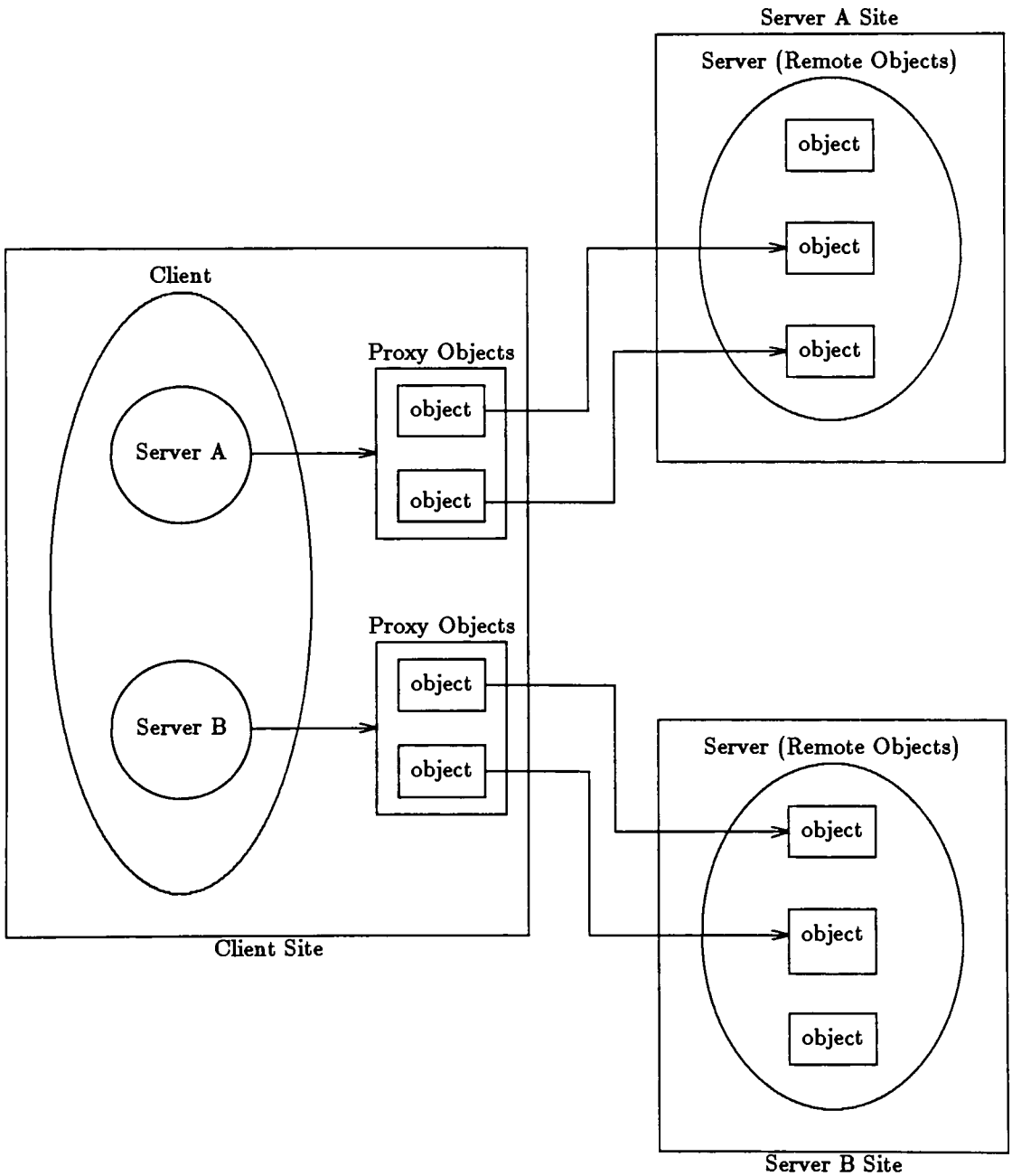

Figure 2. Proxy object organization

3.3.4 Nature of Remote Objects

Remote objects created in distributed C++ are volatile. No mechanism is provided for saving the state of existing objects in the event of crashes and recovering these states when the machine recovers. In the event of a crash of the client or server, the communication channel is destroyed and all the objects are destroyed too. Stable objects could be implemented using atomic operations. In stable objects atomic operations are grouped together in transactions that are guaranteed to have the atomic property. If a transaction commits, then all the operations happen, otherwise if it aborts, none of them happen. After a crash, objects are recovered by continuing the system from the last checkpoint. Implementation of atomic transactions is beyond the scope of this thesis.

3.3.5 Consistency and Access of remote class methods

Remote class method execution is consistent with that of local objects. Invocation of operations on remote objects is through the stub operations generated by the compiler and is transparent to the client application program. When the user program invokes an operation on a remote object, control is transferred to the stub method. The stub method invokes the remote operation through an underlying remote procedure call mechanism. This is discussed in detail in chapter 4. This method eliminates the need for a change in the syntax or semantics of accessing remote class methods. Figure 3 shows the relationship between the user programs and the stubs of the distributed application, when a remote object method is invoked.

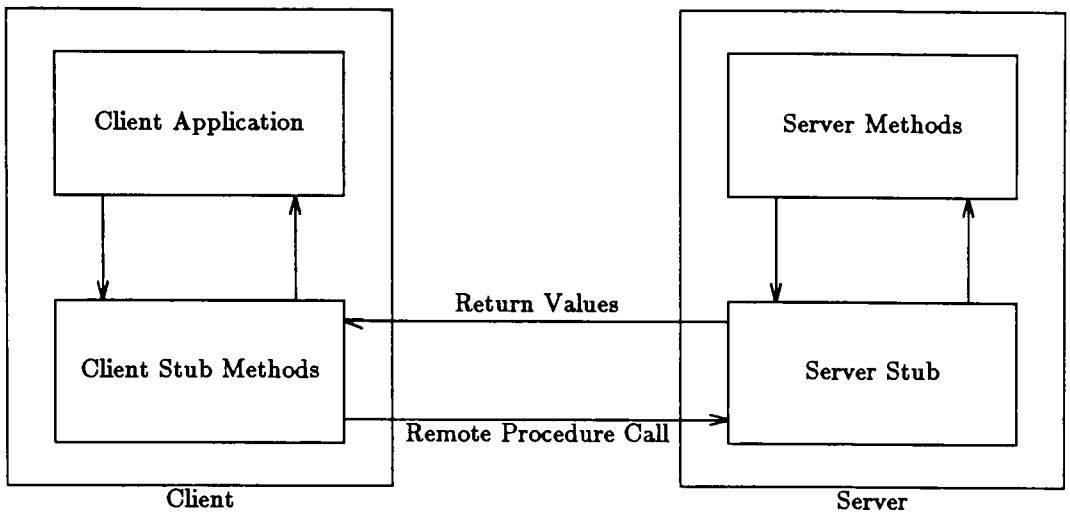

Figure 3. Execution of remote methods

3.3.6 Remote Class naming mechanism

The ideal naming mechanism would be location independent while providing the user with facilities to control the location of remote objects. In this implementation of distributed C++ a remote class is defined by its class identifier (integer) and the address of its server (integer). These are defined in the remote class definition file. This is done to simplify the current implementation. To make the naming mechanism transparent a replicated database can be designed to store system-wide unique class names and locations, making the location of the objects transparent to the user. This will eliminate the need for specifying Server objects in the constructors for remote objects. Also, the user can be given control of the location by allowing the user to create a Server object for a particular machine. Checking can be done at runtime for existence of remote class definitions on that server without trying to establish a communication channel.

3.3.7 Inheritance in remote objects

Remote classes can be derived from other remote classes only. This restriction is imposed as otherwise a mechanism to permit rclass definition and instances on different machines would have to be provided. The base rclass of a derived rclass may be public or private. In case it is public, instances of the derived rclass have access to the interface exported by the base rclass, otherwise, only methods of the derived rclass have access to it. Also, base rclasses and derived rclasses must be present on the same machine.

3.3.8 Local object management

To maintain information about and access to remote objects created on the server machine, the notion of ***proxy objects*** is introduced. For each remote object created, a proxy remote object is created on the local machine. These contain a reference to the actual remote object, and information about its location. A class *RemoteObj* is designed to maintain a uniform representation of all remote objects on the client machine. This is accomplished by automatically compiling all rclasses to have this RemoteObj class as their base class. Thus, a proxy object is an instance of class RemoteObj. Access to remote objects is handled via these proxy objects.

3.3.9 Remote object management

Remote objects are created dynamically on the server machine when their constructors are invoked in the user program, and have the life of their corresponding proxy objects on the client machine. These objects have to be identified and retrieved at runtime when objects on the client invoke their class

methods. A table of remote objects created by the user program is maintained on the server for each importing client. This table stores information to identify a remote object, and a reference to the object, using which it is retrieved when referenced by the user.

3.3.10 Protection

No elaborate mechanisms for protection are designed. The current design permits any user to use a remote class, provided they have information on its location and communication address. The design can be extended to provide a mechanism for authentication of clients instantiating Server objects. In the current design each user is given a logically distinct process and address space on the server. This eliminates the possibility of object sharing among users, but does secure a user's object from being accessed by other users.

4. Implementation of Distributed C++

This chapter describes how the design decisions discussed in chapter 3 are implemented. The implementation of distributed C++ consists of three major components

1. Remote procedure call mechanism
2. Runtime library interface
3. Front end compiler

The following sections discuss each in detail.

4.1 The Remote Procedure Call (RPC) Mechanism

The remote procedure call is modeled on the local procedure call, a well understood mechanism for transfer of control and data within a program running on a single computer. The difference lies in that a RPC is executed in a different process, usually on a different computer, to transfer control and data across the communication network. When a remote procedure is invoked, the calling program's execution is suspended, the parameters are passed across the network to the environment where the procedure is to execute, and the desired procedure is executed there. The results are passed back to the calling program when the procedure finishes execution in the remote environment, and the calling program resumes execution as if returning from a simple single machine call.

Clean and simple semantics, efficiency, and generality are some of the features that have made the idea of RPC more attractive as compared to message passing systems to construct distributed applications. Several RPC systems have been designed and implemented over the past few years. A few of these are listed in references [2][5][12][13][19][21][22][28][29][32-42]. The RPC mechanism implemented for Distributed C++ derives from the Xerox Courier RPC protocol [41] and the Sun RPC protocol [36] specifications. In their classic paper Birell and Nelson [5] discuss several key issues in the design of reliable remote procedure call mechanisms. These issues and their realization in this implementation are discussed in the following sections.

4.1.1 Structure of the Remote Procedure Call

The remote procedure call mechanism implemented is based on the concept of *stubs*. In this approach RPC is implemented using language level stubs generated by a stub compiler. Each remote call involves five pieces of code : the client application program, the client-stub, the run time communication environment, the server-stub, and the server application program. The relationship between these is as shown in figure 4.

When the client (etc.) program executes a remote procedure, it actually invokes a local procedure in the client stub. The client stub creates a data packet into which it packs the arguments of the remote procedure and information to identify the target procedure on the server machine. The runtime communication environment

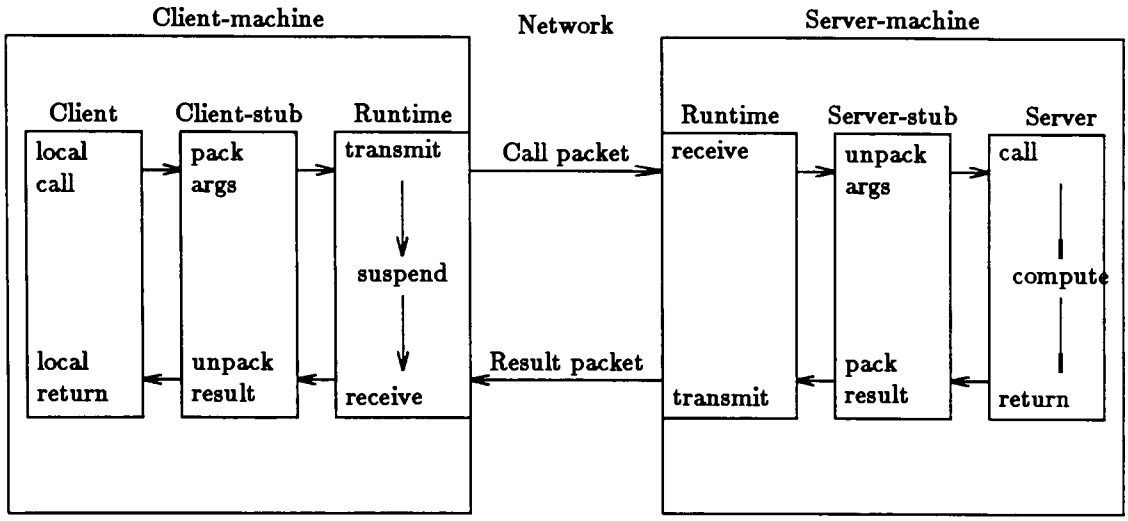

Figure 4. Component interaction for a RPC execution

then transmits this packet over the network to the server machine. The server stub receives this data packet from the runtime environment on the server, and unpacks the arguments and executes the target procedure in the server application program (again a local call). Upon completion of the call in the server, the server stub packs the results returned, and the runtime environment transmits the result packet to the client machine. The result packet is retrieved by the client stub and the results are unpacked. The client stub returns the results to the user program and the call returns in the user program. The remote call in the user program is suspended for the duration of this entire process.

The runtime environment uses TCP (Transmission Control Protocol) for reliable transmission of data over the network. It consists of a library of objects used by the stubs. The user and server programs are part of the distributed application, and are written by the user. The stubs are automatically generated by the front end compiler. The remote procedure call is the underlying mechanism by which methods exported by a class of remote objects are executed.

4.1.2 Remote Procedure Call Semantics

Call semantics describe the behavior of remote procedure calls. The two cases to be considered are, RPC in the presence of crashes, and RPC in the absence of crashes. Crashes may be due to local or remote process failure, or due to communication failure. It is desired that local and remote calls have the same semantics in order to maintain local and remote transparency.

Local procedure calls use exactly-once sematics, i.e. the caller transfers parameters and control to the callee, blocks until the procedure finishes executing exactly once and resumes when the callee returns results. In the case of absence of crashes, RPC achieves these exactly-once semantics. This is achieved by reliable transmission of control passing (call and return) messages, which ensures that these messages are exchanged exactly once as desired. Assuming error free communication and no crashing of host computers, local and remote call transparency is achieved.

In the presence of machine crashes local calls have last-one semantics, i.e. if the machine crashes during the execution of a procedure call and recovers, and its crashed programs are restarted either from checkpoints or from scratch, the procedure call is repeated. This repetition continues until the call finally completes and the program uses the results of the very last call that executes. In the case of a RPC, it can fail due to the local machine crashing, the remote machine crashing, or the failure of the transport mechanism. In the event of any of these crashes, if the processes are restarted and the RPC resumes, the last one is executed and the results returned to the caller. Thus last-one semantics are used.

In this implementation TCP/IP is used for network communication. TCP guarantees reliable, error free communication and exactly once semantics are used. Even though connection overhead is high, TCP was chosen as it performs comparably with UDP once a connection is established. It provides for a straightforward RPC implementation and a reliable communication channel. A connection between the client and server host is established only once and it is destroyed upon the termination of the program or expilicitly by the application program. In the case of crashes of either of the hosts or the network the local and remote processes terminate and have to be restarted from scratch.

4.1.3 Semantics of Arguments of Procedures

The ideal RPC mechanism would permit a full range of types as arguments and result parameters, as in the case of local calls. This is easy for languages with simple scalar and static array types. Problems occur when parameters have implicit or explicit pointers, as in the case of C++. Typically arguments are passed to a local call by reference or by value.

Passing arguments by reference is semantically equivalent to passing a pointer to the argument. Call by reference can also be implemented by using call by value-result, in which case a copy of the argument is made upon procedure entry and the final result is copied back into the argument when the procedure returns. In case of remote procedures since the caller and callee have distinct address spaces, addresses valid in one machine are not meaningful in the other. Thus, using call by reference could be catastrophic. Call by value-result parameters can easily be supported by

RPC, by transmitting back into the caller's reference arguments the final values of the corresponding formal arguments from the site of the remote invocation. This copy-in copy-out mechanism provides uniform local and remote semantics for call by reference value-result parameters.

Pointers to arguments are passed to procedures primarily for efficiency, for passing structures and other bulky values, and for list structures. If pointers are used for efficiency reasons only, call by value-result semantics can be used to implement pointer arguments in remote procedures. In the case of list structures another problem arises in encoding the structure to send it over the network, where it is decoded, reconstructed, and passed to the remote procedure. This requires extensive pointer chasing and storage allocation. Also, the application programmer would be responsible for programming the encoding and decoding of each list structure used.

In C++ arguments can be passed by value, by reference, and by the explicit use of pointers to arguments. Arguments passed by reference are implemented as contextually deferenced pointers to the arguments, as C allows arguments to be passed by value only. In C addresses of arguments are passed by using pointers to the arguments. So in effect C++ allows arguments to be passed by value, and pointers to them. In order to keep the implementation simple and free the programmer from writing the encoding and decoding procedures for address containing arguments, parameters are passed by value only. The semantics of call by value for remote calls are the same as for local calls. This includes array and

character string arguments. Procedure arguments are not allowed. Data types allowed as arguments are discussed later. Values are returned to the client using normal function return semantics of local calls.

4.1.4 Binding Mechanism

For remote procedure calls there are two aspects to binding. The first is *naming* which addresses the problem of how the client specifies what he wants to be bound to, and the second is the problem of *locating* the appropriate server, i.e. how a caller determines the machine address of the callee and how does the caller specify to the callee which procedure is to be executed.

The ideal binding mechanism would provide a location transparent naming mechanism, and also provide facilities for the user to control the location of the remote procedures. This would involve the implementation of a distributed database, possibly replicated to improve reliability, which would store pertinent information about names and locations of RPC interfaces exported on the distributed system. Upon invocation of a remote procedure by the client application program, the client stub would then bind to a server using this database at execution time. Also, server programs that export their interface would do so either via a shell interface or when executed by updating the database if necessary.

The RPC mechanism implemented is the underlying mechanism through which remote objects invoke their class methods. Thus, in this case binding is done at the object level and not for specific procedures. The binding mechanism described in chapter 3 is not transparent and the user explicitly specifies the location of the

remote object instances.

4.1.5 Protocols for Transfer of Data and Control between Caller and Callee

In a heterogeneous distributed environment internal representation of data types may be different on different machines, for example, an integer on one machine may be 32 bits and on another may be 16 bits. In case of RPC the arguments have to be transferred from the caller's environment on one machine to the callee on a remote machine. Thus, arguments have to be transferred in a machine independent form to avoid incompatabilities in representation of data types on the two machines.

The process of packaging and unpackaging parameters into a call or a return message is called marshalling. Remote procedure calls when executed in the application program, transfer control to the client stub procedures. Arguments are marshalled by the the stub procedures and the communication environment transmits the call and return packets between the client and server machines. The order and manner in which parameters are flattened and marshalled define a data protocol for the transmission of data. This implementation defines a data protocol based on the External Data Representation (XDR) protocol [36] specifications, henceforth called the Distributed C++ Language (DCL).

DCL uses a 32 bit representation to transfer data. Each basic data type is converted to a multiple of 32 bits, for example, a character has a size of 8 bits and is stored as one 32 bit word and transmitted, while a double precision number is 64 bits long and is stored as two 32 bit words in the data packet. Marshalling of

pointers requires chasing of pointers, as for a tree structure, each node in the tree has to be traversed and marshalled for transferring the call packet and similarly the tree has to be reconstructed in the callee's environment. It is not possible to automatically generate functions to marshall such pointer data types and the user would have to supply these functions for each pointer type defined. To avoid these problems pointer types are not permitted in DCL. To ensure security of data transmitted, it may be encrypted before transmitting it over the network. Since security is not considered as a major issue, this implementation does not encrypt transmitted data. Data types permitted are described in the following sections.

4.1.5.1 Data Types

Basic data types allowed are the C++ data types.

- [unsigned] int
- [unsigned] long
- [unsigned] short
- [unsigned] char
- float
- double

Derived data types allowed are

- structures
- enumerated types

- fixed size arrays
- discriminated unions
- classes
- remote classes

The runtime library desribed later defines a Packet object into which call parameters are marshalled. Operations for marshalling basic data types are predefined for the object. Code is generated by the compiler to marshall user defined types. A subclass of class Packet is created to marshall all user defined types and a method for each type is defined. Derived data type marshalling is described in the following sections. The grammar for DCL is described in Appendix A.

Type definitions

Type definitions are declared using the keyword *typedef*, as in C++. For example,

```
typedef string char[100];
```

declares a new type string, which is an array of 100 characters.

Constant declarations

Integer constants can be defined using the *const* keyword.

```
const SIZE = 100;
```

defines SIZE to be a constant of value 100. This declaration gets compiled into

```
const int SIZE = 100;
```

The compiler replaces every instance of a declared constant by its value.

Structures

DCL structures are similar to C structures with the restriction that structures cannot be nested. For example, a definition of the form

```
struct foo {
        data_type a;
        struct bar {
                data_type b;
                data_type c;
        } d;
};
```

is not allowed.

This limitation is only syntactical as the same can be written as

```
struct bar {
        data_type b;
        data_type c;
};

struct foo {
        data_type a;
        bar d;
};
```

Another limitation is that structures cannot contain function members, which is permitted in C++. For marshalling structure types a method is defined which invokes methods to marshall each member of the structure. If the structure contains any pointer members, it cannot be used as an argument type for a remote class method, and the compiler does not generate code to perform marshalling of that structure. A structure declaration automatically becomes a type definition when declared.

Enumeration Type

Enumeration types are defined as in C++, but only named enumeration types can be declared. Integer values may be assigned to enumeration types. If there is no explicit assignment, the C++ compiler assigns values to these types. For example,

```
enum BOOLEAN { TRUE = 1, FALSE = 0 } ;
```

declares an enumeration type BOOLEAN which can have two values TRUE and FALSE. Enumeration types also become type definitions upon declaration. These are marshalled using a predefined method for enumeration types assuming that enumeration types have the same representation as integers or short integers.

Fixed Size Arrays

Array declarations must specify the array size. For example,

```
int x[];
```

is not allowed by the compiler. Elements of arrays may be basic data types or user defined types. These declarations are limited to one dimension only, but multi-dimensional arrays can be declared using type definitions.

```
typedef int x50[50];
x50 foo[100];
```

Here foo is defined to be a two-dimensional array (100 x 50). Array declarations are marshalled using a predefined method *pkt_vektor* which marshalls each element of the array using the method defined for the element type. Character arrays are a special case and are marshalled using the method *pkt_bytes* defined for the class Packet.

Discriminated unions

This data type is adapted from the XDR [36] protocol. These unions mandate the use of a tag to specify the union member type being accessed at any instant. For example,

```
union foo switch (tag_type tag) {
        case A : int a;
        case B : char b[100];
        default : unsigned c;
};
```

defines a union with a tag which must be an enumeration type. The default case is optional. In case it is present and the tag is not any of the declared cases the default type is used as the union member being accessed. This union declaration is compiled into a structure declaration of the form

```
struct foo {
        tag_type tag;
        union {
                int a;
                char b[100];
                unsigned c;
        };
};
```

The tag member is checked to determine the union member to be marshalled at run time. This change in syntax was necessitated to determine the union member being accessed for marshalling that type.

Classes

Class declarations using the keyword *class* are similar to C++ classes with some restrictions. Data members of classes cannot contain pointers, as these have to be marshalled for arguments of the class type. Also, class methods cannot contain arguments passed by reference (explicit pointers are allowed). The class type

generated for marshalling user-defined data types is automatically compiled as a friend class of each class declared. This is done so that private data of objects can be accessed for marshalling. In case an object of a class type is passed as an argument to the method of a remote object, all its data members are marshalled and transmitted across the network in the call packet and the object reconstructed on the server. Instances of class types are local to the machine where they are declared. Member functions of classes are defined by the user and must reside on both the client and the server machine.

Remote Classes

Remote classes are declared using the keyword *rclass*. They define objects which are to be instantiated on a remote site. The methods of these classes cannot contain any pointer or reference parameters. Remote objects can be passed as arguments to methods of remote classes. In this case the object identifier of the remote object is transmitted to the remote server. The server stub retrieves the referenced object and passes it as the argument to the invoked method.

4.2 The Runtime Library

The runtime library provides an interface for communication between the client and server programs on the network. It consists of a set of classes which define objects used by the client and server programs to communicate.

4.2.1 Proxy Objects

Remote objects are created by invoking the constructors defined by their remote

classes. An instance of the remote object is created on the server machine and a proxy object representing that object is created on the client machine. Client application programs refer to these proxy objects which store information about the location and identity of the actual remote objects.

Proxy objects are implemented as instances of the class *RemoteObj* defined in the runtime library. All remote classes are automatically compiled to have this class as their base class. This provides a uniform representation of remote objects in the client programs. The RemoteObj class is defined as :

```
class RemoteObj {
    private :
      int ObjId;
      int ServerId;
    protected :
      RemoteObj();
      RemoteObj(int oid, int sid);
      ~RemoteObj();
      int myId();
      int myServerId();
      int setMyId(int oid);
      int setMyServerId(int sid);
};
```

- ObjId : is a unique identifier for that object in the distributed application. It identifies the object on the server and is returned by the server when the object is created.
- ServerId : is the index into a table of servers maintained by the client stub. The table of servers contains pointers to Server objects described later.
- myId : returns the object's identifier (ObjId)

- myServerId : returns the object's server identifier (ServerId)
- setMyId : sets the object identifier value
- setMyServerId : sets the object's server identifier

Instances of class RemoteObj cannot be created or accessed by user programs. These are created and manipulated by the client and server stubs. The server stub maintains a list of RemoteObj instances which represent remote objects created by the user programs on that server. User programs invoke stub methods on the proxy objects which are then translated into invocation of the target methods on the actual remote objects residing on the server.

4.2.2 The Server Object

The Server object provides a mechanism by which objects can be created on a specific server machine. It presents a notion of the server machine to the user programs. User programs must instantiate a Server before creating objects on that server. A communication channel between the user program and the server program residing on the specified server machine is established when a Server is instantiated. Any number of Server objects may be created on a client, upto a maximum number equal to the number of connections permitted by the transport interface (TLI and TCP in this case). Remote objects created on a server have the life of the Server object representing the machine. The Server exports an interface to execute a remote class method via a remote procedure call on a server and to establish a communication channel between the client and server machines. It is

defined as :

```
class Server {
    private :
     char* Name;
     int ConnId;
     int ServerId;
     server_state ConnState;
    protected :
     Server(char* name, int portno);
     ~Server();
     int execRemProc(int oid,int pid,int cid,Packet* args,Packet* ret);
     int svrId();
     int disconnect();
};
```

- Name : name of the server machine (user supplied).
- ConnId : identifier of the TCP connection between the client and server machines.
- ServerId : index into the server table maintained by the client stub. It is set when a Server is instantiated.
- ConnState : specifies the state of the connection between the client and server (connected/disconnected).
- Server (constructor) : A Server is instantiated with its name and a TCP port number on which the server stub program accepts connections for that application. A connection between the client and server machines is established when an instance of a Server is created. In case of a fatal error in establishing a connection the application program aborts.
- ~Server (destructor) : destroys the connection between client and server. All

objects created on a server are destroyed when the destructor executes.

- execRemProc : executes a remote class method. It requires information about the remote object identifier, method to be invoked, the object's class, a packet containing arguments for the method, and a packet to return the results returned by the method. This method is invoked by the client stub and should not be invoked by the user programs.
- disconnect : destroys the connection between the client and server machine.

4.2.3 The Client Object

Client objects represent the clients using the services provided by a server. These are instantiated by the server stubs and are not used by the user programs. The server stub instantiates a client object, which opens a communication port on the server for that application. Methods exported by the Client class permit the server stub to listen for and accept connections from client programs. When a connection request is accepted from a client a new process is created to communicate with and provide service to that client. This ensures security from users trying to access other user's objects. The client object is defined by the class :

```
class Client {
    private :
      int ListenId;
      int ConnId;
      int ConnState;
    public :
      Client(int portno);
      ~Client();
      int listen();
      int accept();
```

```
        int serve();
        int getConnId();
};
```

- ListenId : is the connection identifier specifying the TCP port on which the server is listening for connection requests from clients.
- ConnId : is the connection identifier specifying the TCP connection between the client and server programs.
- ConnState : specifies the state of the connection between the server and client (connected/disconnected).
- Client (constructor) : A Client is instantiated with a TCP port number on which connections requests from clients are accepted by that server application. The server application is bound to that port of the transport interface.
- ~Client (destructor) : destroys the communication channel between the client and the server and all the remote objects created in that server application.
- listen : this method when invoked suspends until a connection request is received from a client. Upon receipt of a connection request it returns success.
- accept : this method acknowledges a connection request from a client and establishes a new communication channel between the client and the server.
- serve : after a communication channel has been established between the client and the server, this method is invoked to create a new process which provides services exported by the server to the connected client. The current process returns and listens for connections from other clients.

- getConnId : returns the identifier identifying a particular connection with a client (ConnId).

4.2.4 The Packet Object

An object interface is provided for marshalling data types for class method parameters. The Packet object stores data packets for the call and return messages of a remote procedure call. Methods are defined to marshall primitive and user-defined data types permitted in DCL. Also, methods are provided to manipulate the data buffer of the object. These methods are used for encoding as well as decoding data types. This object is defined as :

```
class Packet {
    private :
      char*    buffer;
      int      currPos;
      int      size;
      int      packetType;
      pkt_op   mode;
    public :
      Packet();
      Packet(int s);
      Packet(int s, char* buff);
      ~Packet();

      char*  getBuffer();
      int    getSize();
      void   setPos(int p);
      void   setBuffer(char* buff);
      void   setMode(pkt_op op);

      bool_t pkt_void();
      bool_t pkt_int(int* i);
      bool_t pkt_unint(uint* ui);
      bool_t pkt_long(long* l);
      bool_t pkt_unlong(ulong* ul);
      bool_t pkt_short(short* s);
      bool_t pkt_unshort(ushort* us);
```

```
        bool_t pkt_char(char* c);
        bool_t pkt_unchar(uchar* uc);
        bool_t pkt_float(float* f);
        bool_t pkt_double(double* d);
        bool_t pkt_enum(enum_t* e);
        bool_t pkt_vektor(char* v, uint s, uint es, BF func);
        bool_t pkt_bytes(char* b, uint s);
};
```

A subclass of class Packet is generated by the compiler to marshall user defined types for specific distributed applications.

4.3 The DCL Compiler

The distributed C++ compiler was implemented in a Unix System V environment using Unix language development tools Lex and Yacc, and the C programming language. It is a single pass compiler wherein, the lexical analyzer generated by Lex recognizes tokens in the input file, and passes them to the bottom up parser produced by Yacc. Symbols are entered into the appropriate symbol tables during the syntax analyzing phase.

The compiler compiles a file containing a set of data type and remote class definitions, and generates client and server stub files for the distributed application. As a convention the compiler called *dcc* expects the input file to have the extension *dc*, and complains otherwise. Each input file is compiled to generate six files :

1. Client header file
2. Server header file
3. Packet header file

4. Client source file

5. Server source file

6. Packet source file

These files are then compiled using the standard C++ compiler (version 1.2.1 or later), and linked with the user written client and server application programs and the runtime library for *dcc*, to create the client and server executable modules. The following sections describe the output files generated by the compiler.

4.3.1 Client Header File

The client header file contains data type and remote class definitions, and is included in the programs forming the client module of the distributed application. Its name is formed by appending *_clnt.h* to the name of the input file. Thus, for an input file *test.dc*, this file would be named *test_clnt.h*.

Data type definitions other than union definitions are echoed unchanged to this file. Unions are translated into structure definitions as described earlier. Remote class (rclass) definitions are compiled into C++ class definitions with a few changes. All rclass definitions are compiled into class definitions with *RemoteObj* as their base class. The compiled definitions contain only the interface exported by the rclasses, i.e. public member functions. To each constructor of a rclass is added an argument of *Server* type defined in the runtime library interface. For example, a remote class definition

```
rclass foo {
    int x;
```

```
        int y;
      public :
        foo(int, int);
        ~foo();
        int xval();
        int yval();
        void setxy(int, int);
  } = 99;
```

gets compiled into

```
  class foo : public RemoteObj {
      public :
        foo(Server, int, int);
        ~foo();
        int xval();
        int yval();
        void setxy(int, int);
  };
```

The rclass identifier is compiled into a *cpp* macro definition, for the above case the macro definition

```
#define foo_Id   99
```

is generated. Also, macro definitions defining integer identifiers for member functions of the class are generated.

Class definitions in the input file are compiled to have the class defined for marshalling user defined types as a *friend* class. This is done to permit access to private data members of the objects of these classes, to enable marshalling of these objects.

4.3.2 Server Header File

The server header file contains data type and remote class definitions, and is included in the programs forming the server module of the distributed application.

Its name is formed by appending *_svr.h* to the name of the input file. Thus, for an input file *test.dc*, this file would be named *test_svr.h*.

This file is similar to the client header file, the only difference being the translation of rclasses. Remote classes are translated to C++ classes and have *RemoteObj* as their base class, as in the case of the client, but these classes contain all the members defined for the rclasses. Constructors are not changed and private, protected, and public members are echoed unchanged to this file. Thus, the above rclass gets compiled into

```
class foo : public RemoteObj {
      int x;
      int y;
    public :
      foo(int, int);
      ~foo();
      int xval();
      int yval();
      void setxy(int, int);
};
```

The rclass and method identifiers are compiled into macro definitions as in the case of the client header file.

4.3.3 Packet Header File

The packet header file contains the class definition to marshall user defined data types and is included in the client and server stubs. Its name is formed by appending *_pkt.h* to the name of the input file. Thus, for an input file *test.dc*, this file would be named *test_pkt.h*.

A subclass of class *Packet* defined in the runtime library is generated to marshall

user defined types. It is named by concatenating the input file name with *_Packet.* Thus, for the example input file this class would be named *test_Packet.* A member function is defined for marshalling each data type defined in the input file. This function is named by concatenating *pkt_* and the data type name. For example, if the input file *test.dc* contains a structure definition

```
struct foo {
    int x;
    int y;
};
```

the class generated is

```
class test_Packet : public Packet {

    public :
        .
        .
        .
      bool_t pkt_foo( foo* );
};
```

Also, a member function is generated for each method of the rclasses defined. These functions marshall the arguments of the rclass methods.

4.3.4 Client Source File

This file is the client stub C++ file. It contains the stub methods generated by the compiler for the rclasses defined. Its name is formed by appending *_clnt.C* to the name of the input file. Thus, for an input file *test.dc*, this file would be named *test_clnt.C.*

A stub member function is generated for each public member function and constructor defined for a rclass. User programs create remote objects using these

stub constructors which in turn invoke the remote constructors, and create proxy objects on the client machine for each remote object created. The remote objects in the user program invoke the stub methods to manipulate their state. The stub methods marshall the arguments for the invoked method and invoke the method *execRemProc* on the object's Server, to execute the remote method on the server machine. The results returned by the remote method are then returned to the user program, if no errors occur during the execution of the RPC.

A table of Server objects is maintained in the client stub file. This table contains pointers to Servers instantiated by the user program and is updated each time a Server is instantiated or destroyed by the user.

4.3.5 Server Source File

This file contains the source for the server stub and main driver modules. Its name is formed by appending *_svr.C* to the name of the input file. Thus, for an input file *test.dc*, this file would be named *test_svr.C*.

The main function drives the stub and server application programs and executes as a daemon process. It sets up transport connections between the server program and client programs requesting services from the server program. A separate process and connection is created for each user, and exists until the user program destroys the connection, or terminates. The driver calls a service routine to service client requests once a connection has been established between the client and the server.

Remote methods are invoked by the client through this service routine. It receives a call packet from the client and transforms it into the method to be executed. The

results of the method executed are then transmitted to the client in a return packet. A linked list of objects created by the client on that server is maintained in this module. This list consists of pointers to the objects which are created dynamically using the appropriate constructors, when a constructor for a remote object is invoked in the user program on the client machine. For each method invoked by the client, the object referenced is retrieved from the linked list, and the method is invoked on that object.

4.3.6 Packet Source File

This file contains the source for the methods defined to marshall user defined data types. Its name is formed by appending *_pkt.C* to the name of the input file. Thus, for an input file *test.dc*, this file would be named *test_pkt.C*.

A method is defined for each data type defined by the user. These methods invoke methods to marshall each component of the data type. For example, to marshall the structure

```
struct foo {
    int x;
    int y;
};
```

the method *pkt_int* is invoked twice to marshall the x and y components. These methods are used by both, the client and the server programs. Encoding and decoding of data types is performed by the same method.

5. Conclusions and Recommendations

Although limited by its range of applications, a learning tool been developed using C++ as the base language to implement object based distributed programming. Problems associated with implementing reliable distributed programs were addressed, and an operational system has been implemented to program object oriented distributed applications using C++ in a Unix environment.

5.1 Features of Distributed C++

Among the various features of this implementation are :

1. The facility to write distributed applications using object-oriented programming techniques.
2. The ability to define remote classes and instantiate remote objects using the syntax and semantics defined for local objects.
3. User transparent invocation of remote class methods.
4. That the user can control the location of remote object instances.
5. That security from malicious users is ensured by giving each user a logically distinct process and address space on the server.
6. That consistency with C++ data types is maintained except for a few restrictions.
7. That inheritance is provided for remote classes.

This implementation can be used as a starting point to develop a tool for constructing reliable and portable distributed applications based on the object paradigm.

5.2 Recommendations for future work

Various design issues have been addressed and decisions were made depending on the complexities of the issues involved. The current implementation needs work to be done in a few areas to make it more robust and complete. These are :

1. Binding Mechanism : a location independent class naming mechanism needs to be developed. This will free the programmer of requiring to know the location of exported interfaces.

2. Authentication: this issue has not been addressed in this thesis. The current implementation makes it possible for any user on the distributed system to access any interface exported on the system. A user authentication mechanism when developed would allow only the intended users of an interface to access a particular interface.

3. Error recovery in the event of crashes : in this implementation the client and server processes have to be restarted all over in the event of node or network crashes. A robust mechanism would provide atomic transactions which would possibly store enough information to start the programs from their crash points when the system or network is restarted after a crash.

4. Object mobilty : in certain cases object migration would be more efficient

than performing operations over the network. A file object is one such case, in which transferring the file to the client site is more efficient than executing multiple read and write operations over the network.

Bibliography

[1] Acetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M., "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of USENIX 1986 Summer Conference*, 1986.

[2] Almes, G. T., "The Impact of Language and System on Remote Procedure Call Design," *6th International Conference on Distributed Computing Systems*, May 1986.

[3] Andrews, G. R., Olsson, R. A., Coffin, M., Elshoff, I., Nilsen, K., Purdin, T., and Townsend, G., "An Overview of the SR Language and Implementation," *ACM Trans. on Prog. Lang. and Sys.*, Vol. 10, No. 1, January 1988.

[4] Bennett, J. K., "The Design and Implementation of Distributed Smalltalk," *Proc. of the Second ACM Conference on Object-Oriented Prog. Sys., Lang., and Applications*, Oct 1987.

[5] Birrel, A. D., and Nelson, B. J., "Implementing Remote Procedure Calls," *ACM Trans. on Computer Systems*, Vol. 2, 1984.

[6] Birrel, A.D., Levin, R., Needham, R. M., and Schroedrer, M. D., "Grapevine: An Exercise in Distributed Computing," *Comm. of the ACM*, Vol. 25, No. 4, 1982.

[7] Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L., "Distribution and Abstract Types in Emerald," *IEEE Trans. on Soft. Engr.*, Vol. SE-13, No. 1, Jan 1987.

[8] Cheriton. D. R., "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, Vol. 1, No. 2, 1984.

[9] Coulouris, G. F., and Dollimore, J., "Distributed Systems: Concepts and Design," *Addison-Wesley Publishing Company*, 1988.

[10] Decouchant, D., "Design of a Distributed Object Manager for the Smalltalk-80 System," *Proc. of the First ACM Conference on Object-Oriented Prog. Sys., Lang., and Applications*, Oct 1986.

[11] Decouchant, D., Krakowiak, S., Meysembourg, M., Riveill, M., and Rousset de Pina, X., "A Synchronization Mechanism for Typed Objects in a Distributed System," *Proceedings of the ACM SIGPLAN Workshop on Object-based Concurrent Programming*, San Diego, Sept 26-27, 1988.

[12] Gibbons, P. B., "A Stub Generator for Multilanguage RPC in Heterogenous Environments," *IEEE Trans. on Software Engr.*, Vol SE-13, No. 1, Jan 1987.

[13] Gifford, D. K., and Glasser, N., "Remote Pipes and Procedures for Efficient Distributed Communication," *ACM Trans. Comput. Sys.*, Vol. 6, No. 3, August 1988.

[14] Heuser, L., Schill, A., and Muhlhauser, M., "Extensions to the Object Paradigm for the Development of Distributed Applications," *Proceedings of the ACM SIGPLAN Workshop on Object-based Concurrent Programming*, San Diego, Sept 26-27, 1988.

[15] Hindel, B., "An Object-Oriented Programming Language for Distributed Systems: HERAKLIT," *Proceedings of the ACM SIGPLAN Workshop on Object-based Concurrent Programming*, San Diego, Sept 26-27, 1988.

[16] Jazayeri, M., "Objects for Distributed Systems," *Proceedings of the ACM SIGPLAN Workshop on Object-based Concurrent Programming*, San Diego, Sept 26-27, 1988.

[17] Jones, M. B., and Rashid, R., J., "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems," *ACM Sigplan Notices*, Vol. 21, No. 11, 1986.

[18] Leach, P. J., Levine, P. H., Douros, B. P., Hamilton, J. A., Nelson, D. L., and Stumpf, B. L., "The Architecture of an Integrated Local Network," *IEEE Journal on Selected Areas in Communications*, Vol. SAC-1, No. 5, 1983.

[19] Liskov, B., "Primitives for Distributed Computing," *Proceedings of the 7th ACM Symposium on Operating System Principles*, Pacific Grove, CA, 1979.

[20] Liskov, B., and Sheifler, R. W., "Guardian and Actions: Linguistic Support for Robust Distributed Programs," *ACM Trans. Prog. Lang. and Sys.*, Vol. 5, No. 3, 1982.

[21] Liskov, B., and Shira, L., "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems," *SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988.

[22] Marinescu, D. C., "Modeling of Programs with Remote Procedures," *7th International Conference on Distributed Computing Systems*, Sept 1987.

[23] Mellor, P. V., Dubery, J. M., and Whitehead, D. G., "Adapting Modula-2 for Distributed Systems," *Software Engr. Journal*, Vol. 1, No. 5, Sept 1986.

[24] Mitchell, J. G., "File Servers," *Local Area Networks: An Advanced Course*, Lecture Notes in Computer Science, No. 184, Springer-Verlag, 1985.

[25] Morris, D. S., and Wheeler, T., "Distributed Program Design in Ada: An Example," *IEEE Computer Society Second International Conference on ADA Applications and Environments*, Miami Beach, Florida, April 1986.

[26] Mullender, S. J., and Tannenbaum, A. S., "The Design of a Capability-Based Distributed Operating System," *Computer Journal*, Vol. 29, No. 4, 1986.

[27] Nelson, B. J., "Remote Procedure Call," *Technical Report CSL-81-9*, Xerox Palo Alto Research Center, 1981.

[28] Otway, D., and Oskiewicz, E., "REX: A Remote Execution Protocol for Object-Oriented Distributed Applications," *7th International Conference on Distributed Computing Systems*, Sept 1987.

[29] Panzeiri, F., and Shrivastava, S. K., "Reliable Remote Calls for Distributed Unix: An Implementation Study," *Proceedings of the Second Symposium on Reliability in Dist. Soft. and Database Systems*, July 1982.

[30] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudison, G., and Thiel, G., "LOCUS: A Network Transparent, High Reliability Distributed

System," *Proc. of the Eighth SOSP*, Pacific Grove, CA, 1981.

[31] Rozier, M., and Martins, L., "The Chorus Distributed Operating System : Some Design Issues," *Distributed Operating Systems. Theory and Practice*, NATO ASI Series, Vol. F28, Springer-Verlag, 1987.

[32] Sheets, K. B., and Lin, K. J., "A Kernel Level Remote Procedure Call Mechanism," *Proceedings of the Eleventh Annual International Computer Software and Applications Conference*, Oct 1987.

[33] Shrivastava, S. K., and Panzeiri, F., "The Design of a Reliable Remote Procedure Call Mechanism," *IEEE Trans. Comput.*, Vol. C-31, No. 7, July 1982.

[34] Souza, R. J., and Miller, S. P., "UNIX and Remote Procedure Calls : A Peaceful Coexistence?," *6th International Conference on Distributed Computing Systems*, May 1986.

[35] Stroustrup, B., "The C++ Programming Language," *Addison-Wesley Publishing Company*, 1986.

[36] Sun Microsystems "Network Programming Reference Manual," *Sun Microsystems, Inc.*, Mountain View, CA, May 1988.

[37] Teitelman, W., "A Tour Through Cedar," *IEEE Software*, Vol. 1, No. 2, 1984.

[38] Tripathi, A. R., and Wang, P., "An Object-Oriented Design Model for Reliable Distributed Systems," *Proceedings of the Third Symposium on Reliability in Dist. Soft. and Database Systems*, Nov 1983.

[39] Wilbur, S., and Bacarisse, B., "Building Distributed Systems with Remote Procedure Calls," *Software Engr. Journal*, Vol. 2, No. 5, Sept 1987.

[40] Wing, J., Herlihy, M., Clamen, S., Detlefs, D., Kietzke, K., Lerner, R., and Ling, S., "The Avalon/C++ Programming Language (Version 0)," *Computer Science Department, Carnegie Mellon University*, Pittsburgh, PA, Dec 1988.

[41] Xerox Corporation, "Courier: The Remote Procedure Call Protocol," *Xerox Systems Integration Standards*, Stamford, Connecticut, 1981.

[42] Yap, K. S., Jalote, P., and Tripathi, S., "Fault Tolerant Remote Procedure Call," *8th International Conference on Distributed Computing Systems*, June 1988.

Appendix A

Distributed C++ Language (DCL) Syntax Summary

This DCL syntax summary is intended more for aiding comprehension than an exact statement of the language. It describes the language for input to the DCL compiler. The syntax described is left recursive as used in the Yacc specification to generate the parser.

Type definitions

A DCL file consists of a series of type definitions

```
definition_list
        : definition
        | definition_list definition
```

Eight types of definitions are recognized

```
definition
        : typedef_definition
        | constant_definition
        | enum_definition
        | structure_definition
        | discriminated_union_definition
        | class_definition
        | rclass_definition
        | portno_definition
```

In addition single line C++ preprocessor directives are recognized and are echoed unchanged to the stub header files generated. Nested definitions are not allowed by the compiler.

Typedef definition

A DCL typedef is defined as follows :

typedef_definition
: **typedef** type_name declarator ;

Constant definitions

Symbolic constants are defined as follows :

constant_definition
: **const** identifier = value ;

The value assigned must be an integer.

Enumeration type definitions

DCL enumerations have the same syntax as C++ enumerations

enum_definition
: **enum** identifier {
enum_list
};

enum_list
: enumerator
| enum_list , enumerator

enumerator
: identifier
| identifier = value

The assigned value may be either an integer or a symbolic constant.

Structure Definitions

Structure definitions have the following syntax :

struct_definition
: **struct** identifier {
struct_decl_list
};

```
struct_decl_list
        : struct_declaration
        | struct_decl_list struct_declaration

struct_declaration
        : type_name struct_declarator_list ;

struct_declarator_list
        : declarator
        | struct_declarator_list , declarator
```

A structure cannot have functions as members, which is permitted in C++.

Discriminated unions

The discriminated union type is adapted from the XDR [36] protocol. These are different from C++ unions and are more analogous to Pascal variant records. The syntax is a cross between a C++ union declaration and C++ switch statement.

```
union_definition
        : union identifier switch ( discrim_decl ) {
                case_list
          };

case_list
        : case case_identifier : type_name declarator ;
        | default : type_name declarator ;
        | case_list case case_identifier : type_name declarator ;

discrim_decl
        : type_name declarator
```

The discriminant declarator must be an enumeration type.

Class definitions

Classes of object instances local to the machine are defined using the keyword **class** as in C++. The syntax is similar to C++ class declarations with some restrictions.

```
class_definition
        : class_head { class_list } ;

class_head
        : class identifier
        | class identifier : identifier
        | class identifier : public identifier
```

Remote class definitions

Remote classes are defined using the keyword **rclass**. The syntax of a remote class definition is similar to that of a class definition.

```
rclass_definition
        : rclass_head { class_list } = class_id ;

rclass_head
        : rclass identifier
        | rclass identifier : identifier
        | rclass identifier : public identifier

class_list
        : member_list
        | class_list member_list

member_list
        : private : mem_list
        | protected : mem_list
        | public : mem_list

mem_list
        : mem_decl
        | mem_list mem_decl

mem_decl
        : data_declaration
        | function_declaration

data_declaration
        : type_name declarator ;

function_declaration
        : type_name function_name ( arg_list ) ;
```

```
        | function_name ( arg_list ) ;
        | ~ function_name () ;

    function_name
        : fct_name
        | virtual fct_name
        | friend fct_name

    fct_name
        : identifier
        | operator op

    arg_list
        : EMPTY
        | arg_list , argument

    argument
        : type_name
        | type_name declarator
```

The *class_id* is an integer value or a symbolic constant.

DCL declarations

Data types recognized by the compiler are

```
    type_name
        : simple_type
        | enum identifier
        | struct identifier
        | union identifier

    simple_type
        : [ unsigned ] int
        | [ unsigned ] short
        | [ unsigned ] long
        | [ unsigned ] char
        | unsigned
        | float
        | double
        | void
        | typedef_name
```

typedef_name
: identifier

Declarators are declared using the following syntax :

declarator
: identifier
| * declarator
| (declarator)
| declarator [const_expr]

const_expr
: integer
| identifier

The following operators are recognized

```
+   -   *   /   %   ^   &   |   !   =   <   >   ~
+=  -=  *=  /=  %=  ^=  &=  |=  !=  ==  <=  >=
<<  >>  <<=  >>=  &&  ||  ++  --  []  ()  new  delete
```

Remote class functions cannot have pointer arguments. Also, pointer data members are not permitted in class definitons. These restrictions are imposed as the remote procedure call mechanism does not permit pointer data types.

Portno definition

The communication port number for the server is defined using the keyword **portno**.

portno_definition
: **portno** = const_expr ;

Appendix B

Sample Program Listing

```
// point.dc
//
// DCL sample program.
// This file contains the definition for the rclass Point.


portno = 55000;

rclass Point {
     int xvalue;
     int yvalue;
   public:
     Point();
     Point (int xValue, int yValue);

     int operator< (Point aPoint);
     int operator<= (Point aPoint);
     int operator>= (Point aPoint);
     int operator== (Point aPoint);
     int operator!= (Point aPoint);
     Point operator* (int scale);
     Point operator+ (Point delta);
     Point operator- (Point delta);
     Point operator/ (int scale);
     Point abs();
     float dist (Point aPoint);
     Point max (Point aPoint);
     Point min (Point aPoint);
     Point transpose();
     int x();
     void x (int aValue);
     void xy (int xValue, int yValue);
     int y();
     void y (int aValue);
} = 1000;
```

```
// run_clnt.C
//
// This file contains the client application program.
// It is supplied by the user and is linked with the client and packet
// stub generated by the DCL compiler.

#include <stream.h>
#include "point_clnt.h"

main() {

    Server peach("ma", point_PORTNO);      // instantiate a server

    cout << "Creating point A with no initial values\n";
    Point a(&peach);
    cout << "Creating point B with initial values (10, 10)\n";
    Point b(&peach,10, 10);
    cout << "Creating point C with initial values (-15, -25)\n";
    Point c(&peach, -15, -25);
    cout << "Creating point D with no initial values\n";
    Point d(&peach);

    char response;

    cout << "Enter q to continue : ";
    cin >> response;

    cout << "Retrieving (x,y) of B\n";
    int bx = b.x();
    int by = b.y();
    cout << "bx : " << bx << " by : " << by << "\n";

    cout << "Enter q to continue : ";
    cin >> response;

    cout << "Getting absolute of C and setting it to A\n";
    a = c.abs();

    cout << "Retrieving (x,y) of A\n";
    int ax = a.x();
    int ay = a.y();
    cout << "ax : " << ax << " ay : " << ay << "\n";

    cout << "Enter q to continue : ";
    cin >> response;
```

```
    cout << "Adding A and B and setting sum to D\n";
    d = a + b;

    cout << "Retrieving (x,y) of D\n";
    int dx = d.x();
    int dy = d.y();
    cout << "dx : " << dx << " dy : " << dy << "\n";


    cout << "Enter q to continue : ";
    cin >> response;

    peach.disconnect();

}
```

```
// point_clnt.h
//
// This file contains the client definition for rclass Point.
// It is generated by the compiler

#ifndef POINT_CLNTH
#define POINT_CLNTH 1

#include <dc++.h>

#define point_PORTNO 55000

#define Point_ID    1000
class Point : public RemoteObj {
    public :
      Point(Server*);
      Point(Server*, int xValue, int yValue);
      int operator<(Point aPoint);
      int operator<=(Point aPoint);
      int operator>=(Point aPoint);
      int operator==(Point aPoint);
      int operator!=(Point aPoint);
      Point operator*(int scale);
      Point operator+(Point delta);
      Point operator-(Point delta);
      Point operator/(int scale);
      Point abs();
      float dist(Point aPoint);
      Point max(Point aPoint);
      Point min(Point aPoint);
      Point transpose();
      int x();
      void x(int aValue);
      void xy(int xValue, int yValue);
      int y();
      void y(int aValue);
};

#define Point_DESTR      0
#define Point_Point_1    1
#define Point_Point_2    2
#define Point_operator_3 3
#define Point_operator_4 4
#define Point_operator_5 5
#define Point_operator_6 6
```

```
#define Point_operator_7   7
#define Point_operator_8   8
#define Point_operator_9   9
#define Point_operator_10  10
#define Point_operator_11  11
#define Point_abs_12       12
#define Point_dist_13      13
#define Point_max_14       14
#define Point_min_15       15
#define Point_transpose_16         16
#define Point_x_17  17
#define Point_x_18  18
#define Point_xy_19        19
#define Point_y_20  20
#define Point_y_21  21


#endif
```

```
// point_clnt.C
//
// This file contains the client stub member functions for the rclass Point.
// This is generated by the DCL compiler.

#include <stream.h>
#include "point_clnt.h"
#include "point_pkt.h"

extern vector(PtrServer)* SvrTable;

Point::Point(Server* s) {

    point_Packet RetPacket(8);
    int retval;
    if (s->execRemProc(Point_ID, Point_Point_1, 0, 0, &RetPacket) == -1) {
      cerr << "Error in executing remote function Point\n";
      exit(1);
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval);
    RemoteObj::setMyId(retval);
    RemoteObj::setMyServerId(s->svrId());
}

Point::Point(Server* s, int arg_1, int arg_2) {

    point_Packet RetPacket(8);
    int retval;
    point_Packet ArgPacket(8);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_Point_2_args(&arg_1, &arg_2);
    if (s->execRemProc(Point_ID, Point_Point_2, 0, &ArgPacket, &RetPacket) == -1) {
      cerr << "Error in executing remote function Point\n";
      exit(1);
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval);
    RemoteObj::setMyId(retval);
    RemoteObj::setMyServerId(s->svrId());
}

int Point::operator<(Point arg_1) {
```

```
    point_Packet RetPacket(8);
    int retval;
    int arg_1_id = arg_1.RemoteObj::myId();
    point_Packet ArgPacket(4);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_operator_3_args(&arg_1_id);
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_operator_3, Re
      cerr << "Error in executing remote function operator\n";
      return retval;
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval);
    return (retval);
}

int Point::operator<=(Point arg_1) {

    point_Packet RetPacket(8);
    int retval;
    int arg_1_id = arg_1.RemoteObj::myId();
    point_Packet ArgPacket(4);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_operator_4_args(&arg_1_id);
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_operator_4, Re
      cerr << "Error in executing remote function operator\n";
      return retval;
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval);
    return (retval);
}

int Point::operator>=(Point arg_1) {

    point_Packet RetPacket(8);
    int retval;
    int arg_1_id = arg_1.RemoteObj::myId();
    point_Packet ArgPacket(4);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_operator_5_args(&arg_1_id);
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_operator_5, Re
      cerr << "Error in executing remote function operator\n";
```

```
        return retval;
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval);
    return (retval);
}

int Point::operator==(Point arg_1) {

    point_Packet RetPacket(8);
    int retval;
    int arg_1_id = arg_1.RemoteObj::myId();
    point_Packet ArgPacket(4);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_operator_6_args(&arg_1_id);
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_operator_6, Re
        cerr << "Error in executing remote function operator\n";
        return retval;
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval);
    return (retval);
}

int Point::operator!=(Point arg_1) {

    point_Packet RetPacket(8);
    int retval;
    int arg_1_id = arg_1.RemoteObj::myId();
    point_Packet ArgPacket(4);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_operator_7_args(&arg_1_id);
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_operator_7, Re
        cerr << "Error in executing remote function operator\n";
        return retval;
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval);
    return (retval);
}

Point Point::operator*(int arg_1) {
```

```
    point_Packet RetPacket(8);
    int retval_id;
    RemoteObj retval;
    point_Packet ArgPacket(4);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_operator_8_args(&arg_1);
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_operator_8, Re
      cerr << "Error in executing remote function operator\n";
      return (*(Point *) (&retval));
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval_id);
    retval.setMyId(retval_id);
    retval.setMyServerId(RemoteObj::myServerId());
    return (*(Point *) (&retval));
}

Point Point::operator+(Point arg_1) {

    point_Packet RetPacket(8);
    int retval_id;
    RemoteObj retval;
    int arg_1_id = arg_1.RemoteObj::myId();
    point_Packet ArgPacket(4);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_operator_9_args(&arg_1_id);
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_operator_9, Re
      cerr << "Error in executing remote function operator\n";
      return (*(Point *) (&retval));
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval_id);
    retval.setMyId(retval_id);
    retval.setMyServerId(RemoteObj::myServerId());
    return (*(Point *) (&retval));
}

Point Point::operator-(Point arg_1) {

    point_Packet RetPacket(8);
    int retval_id;
    RemoteObj retval;
    int arg_1_id = arg_1.RemoteObj::myId();
```

```
    point_Packet ArgPacket(4);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_operator_10_args(&arg_1_id);
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_operator_10, Re
      cerr << "Error in executing remote function operator\n";
      return (*(Point *) (&retval));
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval_id);
    retval.setMyId(retval_id);
    retval.setMyServerId(RemoteObj::myServerId());
    return (*(Point *) (&retval));
}

Point Point::operator/(int arg_1) {

    point_Packet RetPacket(8);
    int retval_id;
    RemoteObj retval;
    point_Packet ArgPacket(4);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_operator_11_args(&arg_1);
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_operator_11, Re
      cerr << "Error in executing remote function operator\n";
      return (*(Point *) (&retval));
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval_id);
    retval.setMyId(retval_id);
    retval.setMyServerId(RemoteObj::myServerId());
    return (*(Point *) (&retval));
}

Point Point::abs() {

    point_Packet RetPacket(8);
    int retval_id;
    RemoteObj retval;
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_abs_12, Remote
      cerr << "Error in executing remote function abs\n";
      return (*(Point *) (&retval));
    }
    RetPacket.setMode(PKT_DECODE);
```

```
    RetPacket.pkt_int(&retval_id);
    retval.setMyId(retval_id);
    retval.setMyServerId(RemoteObj::myServerId());
    return (*(Point *) (&retval));
}

float Point::dist(Point arg_1) {

    point_Packet RetPacket(8);
    float retval;
    int arg_1_id = arg_1.RemoteObj::myId();
    point_Packet ArgPacket(4);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_dist_13_args(&arg_1_id);
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_dist_13, Remot
      cerr << "Error in executing remote function dist\n";
      return retval;
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_float(&retval);
    return (retval);
}

Point Point::max(Point arg_1) {

    point_Packet RetPacket(8);
    int retval_id;
    RemoteObj retval;
    int arg_1_id = arg_1.RemoteObj::myId();
    point_Packet ArgPacket(4);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_max_14_args(&arg_1_id);
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_max_14, Remot
      cerr << "Error in executing remote function max\n";
      return (*(Point *) (&retval));
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval_id);
    retval.setMyId(retval_id);
    retval.setMyServerId(RemoteObj::myServerId());
    return (*(Point *) (&retval));
}
```

```
Point Point::min(Point arg_1) {

    point_Packet RetPacket(8);
    int retval_id;
    RemoteObj retval;
    int arg_1_id = arg_1.RemoteObj::myId();
    point_Packet ArgPacket(4);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_min_15_args(&arg_1_id);
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_min_15, Remot
      cerr << "Error in executing remote function min\n";
      return (*(Point *) (&retval));
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval_id);
    retval.setMyId(retval_id);
    retval.setMyServerId(RemoteObj::myServerId());
    return (*(Point *) (&retval));
}

Point Point::transpose() {

    point_Packet RetPacket(8);
    int retval_id;
    RemoteObj retval;
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_transpose_16, R
      cerr << "Error in executing remote function transpose\n";
      return (*(Point *) (&retval));
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval_id);
    retval.setMyId(retval_id);
    retval.setMyServerId(RemoteObj::myServerId());
    return (*(Point *) (&retval));
}

int Point::x() {

    point_Packet RetPacket(8);
    int retval;
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_x_17, RemoteO
      cerr << "Error in executing remote function x\n";
      return retval;
    }
```

```
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval);
    return (retval);
}

void Point::x(int arg_1) {

    point_Packet RetPacket(4);
    point_Packet ArgPacket(4);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_x_18_args(&arg_1);
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_x_18, RemoteO
      cerr << "Error in executing remote function x\n";
    }
}

void Point::xy(int arg_1, int arg_2) {

    point_Packet RetPacket(4);
    point_Packet ArgPacket(8);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_xy_19_args(&arg_1, &arg_2);
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_xy_19, Remote
      cerr << "Error in executing remote function xy\n";
    }
}

int Point::y() {

    point_Packet RetPacket(8);
    int retval;
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_y_20, RemoteO
      cerr << "Error in executing remote function y\n";
      return retval;
    }
    RetPacket.setMode(PKT_DECODE);
    RetPacket.pkt_int(&retval);
    return (retval);
}

void Point::y(int arg_1) {

    point_Packet RetPacket(4);
```

```
    point_Packet ArgPacket(4);

    ArgPacket.setMode(PKT_ENCODE);
    ArgPacket.Point_y_21_args(&arg_1);
    if (((*SvrTable)[RemoteObj::myServerId()])->execRemProc(Point_ID, Point_y_21, RemoteO
      cerr << "Error in executing remote function y\n";
    }
}
```

```
// point_svr.C
//
// This file contains the server stub generated by the compiler.

#include <stream.h>
#include "point_svr.h"
#include "point_pkt.h"

main () {

    Client point_client(point_PORTNO);
    int run_point_server(int);
    const int CHILD = 2;

    for ( ; ; ) {
        point_client.listen();
        point_client.accept();
        if (point_client.serve() == CHILD) {
          run_point_server(point_client.getConnId());
          break;
        }
    }
}


int run_point_server(int fd) {
   extern clnt_snd(int, char*, unsigned);
   extern clnt_rcv(int, char*, unsigned);
   Rlist ObjList;
   int curr_objid = 0;
   Packet HdrPkt(16);
   int procid;
   int classid;
   int objid;
   int pktsize;
   int brcvd;
   int success = 0;
   int failure = -1;

   while ((brcvd = clnt_rcv(fd, HdrPkt.getBuffer(), 16)) >= 0) {

        HdrPkt.setMode(PKT_DECODE);
        HdrPkt.pkt_int(&classid);
        HdrPkt.pkt_int(&procid);
        HdrPkt.pkt_int(&objid);
```

```
HdrPkt.pkt_int(&pktsize);
HdrPkt.setPos(0);

switch (classid) {

  case Point_ID : {

      switch(procid) {

        case Point_Point_1 : {
          point_Packet RetPacket(8);
          int retval;
          RetPacket.setMode(PKT_ENCODE);
          Point* objp = new Point;
          if (objp) {
            retval = int(objp);
            ObjList.insert(objp, retval);
            RetPacket.pkt_int(&success);
            RetPacket.pkt_int(&retval);
          }
          else
            RetPacket.pkt_int(&failure);
          clnt_snd(fd, RetPacket.getBuffer(), 8);
        }
        break;

        case Point_Point_2 : {
          point_Packet ArgPacket(8);
          point_Packet RetPacket(8);
          int retval;
          int arg_1;
          int arg_2;
          clnt_rcv(fd, ArgPacket.getBuffer(), 8);
          ArgPacket.setMode(PKT_DECODE);
          ArgPacket.Point_Point_2_args(&arg_1, &arg_2);
          RetPacket.setMode(PKT_ENCODE);
          Point* objp = new Point(arg_1, arg_2);
          if (objp) {
            retval = int(objp);
            ObjList.insert(objp, retval);
            RetPacket.pkt_int(&success);
            RetPacket.pkt_int(&retval);
          }
          else
            RetPacket.pkt_int(&failure);
```

```
    clnt_snd(fd, RetPacket.getBuffer(), 8);
  }
  break;

  case Point_operator_3 : {
    Point* objp = (Point*) ObjList.inlist(objid);
    point_Packet ArgPacket(4);
    point_Packet RetPacket(8);
    int retval;
    int arg_1_id;
    clnt_rcv(fd, ArgPacket.getBuffer(), 4);
    ArgPacket.setMode(PKT_DECODE);
    ArgPacket.Point_operator_3_args(&arg_1_id);
    Point* arg_1 = (Point*) ObjList.inlist(arg_1_id);
    RetPacket.setMode(PKT_ENCODE);
    if (objp) {
      retval = objp->operator<(*arg_1);
      RetPacket.pkt_int(&success);
      RetPacket.pkt_int(&retval);
    }
    else
      RetPacket.pkt_int(&failure);
    clnt_snd(fd, RetPacket.getBuffer(), 8);
  }
  break;

  case Point_operator_4 : {
    Point* objp = (Point*) ObjList.inlist(objid);
    point_Packet ArgPacket(4);
    point_Packet RetPacket(8);
    int retval;
    int arg_1_id;
    clnt_rcv(fd, ArgPacket.getBuffer(), 4);
    ArgPacket.setMode(PKT_DECODE);
    ArgPacket.Point_operator_4_args(&arg_1_id);
    Point* arg_1 = (Point*) ObjList.inlist(arg_1_id);
    RetPacket.setMode(PKT_ENCODE);
    if (objp) {
      retval = objp->operator<=(*arg_1);
      RetPacket.pkt_int(&success);
      RetPacket.pkt_int(&retval);
    }
    else
      RetPacket.pkt_int(&failure);
    clnt_snd(fd, RetPacket.getBuffer(), 8);
```

```
}
break;

case Point_operator_5 : {
  Point* objp = (Point*) ObjList.inlist(objid);
  point_Packet ArgPacket(4);
  point_Packet RetPacket(8);
  int retval;
  int arg_1_id;
  clnt_rcv(fd, ArgPacket.getBuffer(), 4);
  ArgPacket.setMode(PKT_DECODE);
  ArgPacket.Point_operator_5_args(&arg_1_id);
  Point* arg_1 = (Point*) ObjList.inlist(arg_1_id);
  RetPacket.setMode(PKT_ENCODE);
  if (objp) {
    retval = objp->operator>=(*arg_1);
    RetPacket.pkt_int(&success);
    RetPacket.pkt_int(&retval);
  }
  else
    RetPacket.pkt_int(&failure);
  clnt_snd(fd, RetPacket.getBuffer(), 8);
}
break;

case Point_operator_6 : {
  Point* objp = (Point*) ObjList.inlist(objid);
  point_Packet ArgPacket(4);
  point_Packet RetPacket(8);
  int retval;
  int arg_1_id;
  clnt_rcv(fd, ArgPacket.getBuffer(), 4);
  ArgPacket.setMode(PKT_DECODE);
  ArgPacket.Point_operator_6_args(&arg_1_id);
  Point* arg_1 = (Point*) ObjList.inlist(arg_1_id);
  RetPacket.setMode(PKT_ENCODE);
  if (objp) {
    retval = objp->operator==(*arg_1);
    RetPacket.pkt_int(&success);
    RetPacket.pkt_int(&retval);
  }
  else
    RetPacket.pkt_int(&failure);
  clnt_snd(fd, RetPacket.getBuffer(), 8);
}
```

```
break;

case Point_operator_7 : {
  Point* objp = (Point*) ObjList.inlist(objid);
  point_Packet ArgPacket(4);
  point_Packet RetPacket(8);
  int retval;
  int arg_1_id;
  clnt_rcv(fd, ArgPacket.getBuffer(), 4);
  ArgPacket.setMode(PKT_DECODE);
  ArgPacket.Point_operator_7_args(&arg_1_id);
  Point* arg_1 = (Point*) ObjList.inlist(arg_1_id);
  RetPacket.setMode(PKT_ENCODE);
  if (objp) {
    retval = objp->operator!=(*arg_1);
    RetPacket.pkt_int(&success);
    RetPacket.pkt_int(&retval);
  }
  else
    RetPacket.pkt_int(&failure);
  clnt_snd(fd, RetPacket.getBuffer(), 8);
}
break;

case Point_operator_8 : {
  Point* objp = (Point*) ObjList.inlist(objid);
  point_Packet ArgPacket(4);
  point_Packet RetPacket(8);
  Point* retval = new Point;
  int retval_id;
  int arg_1;
  clnt_rcv(fd, ArgPacket.getBuffer(), 4);
  ArgPacket.setMode(PKT_DECODE);
  ArgPacket.Point_operator_8_args(&arg_1);
  RetPacket.setMode(PKT_ENCODE);
  if (objp) {
    *retval = objp->operator*(arg_1);
    retval_id = int(retval);
    ObjList.insert(retval, retval_id);
    RetPacket.pkt_int(&success);
    RetPacket.pkt_int(&retval_id);
  }
  else
    RetPacket.pkt_int(&failure);
  clnt_snd(fd, RetPacket.getBuffer(), 8);
```

```
}
break;

case Point_operator_9 : {
  Point* objp = (Point*) ObjList.inlist(objid);
  point_Packet ArgPacket(4);
  point_Packet RetPacket(8);
  Point* retval = new Point;
  int retval_id;
  int arg_1_id;
  clnt_rcv(fd, ArgPacket.getBuffer(), 4);
  ArgPacket.setMode(PKT_DECODE);
  ArgPacket.Point_operator_9_args(&arg_1_id);
  Point* arg_1 = (Point*) ObjList.inlist(arg_1_id);
  RetPacket.setMode(PKT_ENCODE);
  if (objp) {
    *retval = objp->operator+(*arg_1);
    retval_id = int(retval);
    ObjList.insert(retval, retval_id);
    RetPacket.pkt_int(&success);
    RetPacket.pkt_int(&retval_id);
  }
  else
    RetPacket.pkt_int(&failure);
  clnt_snd(fd, RetPacket.getBuffer(), 8);
}
break;

case Point_operator_10 : {
  Point* objp = (Point*) ObjList.inlist(objid);
  point_Packet ArgPacket(4);
  point_Packet RetPacket(8);
  Point* retval = new Point;
  int retval_id;
  int arg_1_id;
  clnt_rcv(fd, ArgPacket.getBuffer(), 4);
  ArgPacket.setMode(PKT_DECODE);
  ArgPacket.Point_operator_10_args(&arg_1_id);
  Point* arg_1 = (Point*) ObjList.inlist(arg_1_id);
  RetPacket.setMode(PKT_ENCODE);
  if (objp) {
    *retval = objp->operator-(*arg_1);
    retval_id = int(retval);
    ObjList.insert(retval, retval_id);
    RetPacket.pkt_int(&success);
```

```
      RetPacket.pkt_int(&retval_id);
    }
    else
      RetPacket.pkt_int(&failure);
    clnt_snd(fd, RetPacket.getBuffer(), 8);
  }
  break;

  case Point_operator_11 : {
    Point* objp = (Point*) ObjList.inlist(objid);
    point_Packet ArgPacket(4);
    point_Packet RetPacket(8);
    Point* retval = new Point;
    int retval_id;
    int arg_1;
    clnt_rcv(fd, ArgPacket.getBuffer(), 4);
    ArgPacket.setMode(PKT_DECODE);
    ArgPacket.Point_operator_11_args(&arg_1);
    RetPacket.setMode(PKT_ENCODE);
    if (objp) {
      *retval = objp->operator/(arg_1);
      retval_id = int(retval);
      ObjList.insert(retval, retval_id);
      RetPacket.pkt_int(&success);
      RetPacket.pkt_int(&retval_id);
    }
    else
      RetPacket.pkt_int(&failure);
    clnt_snd(fd, RetPacket.getBuffer(), 8);
  }
  break;

  case Point_abs_12 : {
    Point* objp = (Point*) ObjList.inlist(objid);
    point_Packet RetPacket(8);
    Point* retval = new Point;
    int retval_id;
    RetPacket.setMode(PKT_ENCODE);
    if (objp) {
      *retval = objp->abs();
      retval_id = int(retval);
      ObjList.insert(retval, retval_id);
      RetPacket.pkt_int(&success);
      RetPacket.pkt_int(&retval_id);
    }
```

```
    else
      RetPacket.pkt_int(&failure);
    clnt_snd(fd, RetPacket.getBuffer(), 8);
  }
  break;

  case Point_dist_13 : {
    Point* objp = (Point*) ObjList.inlist(objid);
    point_Packet ArgPacket(4);
    point_Packet RetPacket(8);
    float retval;
    int arg_1_id;
    clnt_rcv(fd, ArgPacket.getBuffer(), 4);
    ArgPacket.setMode(PKT_DECODE);
    ArgPacket.Point_dist_13_args(&arg_1_id);
    Point* arg_1 = (Point*) ObjList.inlist(arg_1_id);
    RetPacket.setMode(PKT_ENCODE);
    if (objp) {
      retval = objp->dist(*arg_1);
      RetPacket.pkt_int(&success);
      RetPacket.pkt_float(&retval);
    }
    else
      RetPacket.pkt_int(&failure);
    clnt_snd(fd, RetPacket.getBuffer(), 8);
  }
  break;

  case Point_max_14 : {
    Point* objp = (Point*) ObjList.inlist(objid);
    point_Packet ArgPacket(4);
    point_Packet RetPacket(8);
    Point* retval = new Point;
    int retval_id;
    int arg_1_id;
    clnt_rcv(fd, ArgPacket.getBuffer(), 4);
    ArgPacket.setMode(PKT_DECODE);
    ArgPacket.Point_max_14_args(&arg_1_id);
    Point* arg_1 = (Point*) ObjList.inlist(arg_1_id);
    RetPacket.setMode(PKT_ENCODE);
    if (objp) {
      *retval = objp->max(*arg_1);
      retval_id = int(retval);
      ObjList.insert(retval, retval_id);
      RetPacket.pkt_int(&success);
```

```
      RetPacket.pkt_int(&retval_id);
    }
    else
      RetPacket.pkt_int(&failure);
    clnt_snd(fd, RetPacket.getBuffer(), 8);
  }
  break;

  case Point_min_15 : {
    Point* objp = (Point*) ObjList.inlist(objid);
    point_Packet ArgPacket(4);
    point_Packet RetPacket(8);
    Point* retval = new Point;
    int retval_id;
    int arg_1_id;
    clnt_rcv(fd, ArgPacket.getBuffer(), 4);
    ArgPacket.setMode(PKT_DECODE);
    ArgPacket.Point_min_15_args(&arg_1_id);
    Point* arg_1 = (Point*) ObjList.inlist(arg_1_id);
    RetPacket.setMode(PKT_ENCODE);
    if (objp) {
      *retval = objp->min(*arg_1);
      retval_id = int(retval);
      ObjList.insert(retval, retval_id);
      RetPacket.pkt_int(&success);
      RetPacket.pkt_int(&retval_id);
    }
    else
      RetPacket.pkt_int(&failure);
    clnt_snd(fd, RetPacket.getBuffer(), 8);
  }
  break;

  case Point_transpose_16 : {
    Point* objp = (Point*) ObjList.inlist(objid);
    point_Packet RetPacket(8);
    Point* retval = new Point;
    int retval_id;
    RetPacket.setMode(PKT_ENCODE);
    if (objp) {
      *retval = objp->transpose();
      retval_id = int(retval);
      ObjList.insert(retval, retval_id);
      RetPacket.pkt_int(&success);
      RetPacket.pkt_int(&retval_id);
```

```
    }
    else
      RetPacket.pkt_int(&failure);
    clnt_snd(fd, RetPacket.getBuffer(), 8);
  }
  break;

  case Point_x_17 : {
    Point* objp = (Point*) ObjList.inlist(objid);
    point_Packet RetPacket(8);
    int retval;
    RetPacket.setMode(PKT_ENCODE);
    if (objp) {
      retval = objp->x();
      RetPacket.pkt_int(&success);
      RetPacket.pkt_int(&retval);
    }
    else
      RetPacket.pkt_int(&failure);
    clnt_snd(fd, RetPacket.getBuffer(), 8);
  }
  break;

  case Point_x_18 : {
    Point* objp = (Point*) ObjList.inlist(objid);
    point_Packet ArgPacket(4);
    point_Packet RetPacket(4);
    int arg_1;
    clnt_rcv(fd, ArgPacket.getBuffer(), 4);
    ArgPacket.setMode(PKT_DECODE);
    ArgPacket.Point_x_18_args(&arg_1);
    RetPacket.setMode(PKT_ENCODE);
    if (objp) {
      objp->x(arg_1);
      RetPacket.pkt_int(&success);
    }
    else
      RetPacket.pkt_int(&failure);
    clnt_snd(fd, RetPacket.getBuffer(), 4);
  }
  break;

  case Point_xy_19 : {
    Point* objp = (Point*) ObjList.inlist(objid);
    point_Packet ArgPacket(8);
```

```
    point_Packet RetPacket(4);
    int arg_1;
    int arg_2;
    clnt_rcv(fd, ArgPacket.getBuffer(), 8);
    ArgPacket.setMode(PKT_DECODE);
    ArgPacket.Point_xy_19_args(&arg_1, &arg_2);
    RetPacket.setMode(PKT_ENCODE);
    if (objp) {
      objp->xy(arg_1, arg_2);
      RetPacket.pkt_int(&success);
    }
    else
      RetPacket.pkt_int(&failure);
    clnt_snd(fd, RetPacket.getBuffer(), 4);
  }
  break;

  case Point_y_20 : {
    Point* objp = (Point*) ObjList.inlist(objid);
    point_Packet RetPacket(8);
    int retval;
    RetPacket.setMode(PKT_ENCODE);
    if (objp) {
      retval = objp->y();
      RetPacket.pkt_int(&success);
      RetPacket.pkt_int(&retval);
    }
    else
      RetPacket.pkt_int(&failure);
    clnt_snd(fd, RetPacket.getBuffer(), 8);
  }
  break;

  case Point_y_21 : {
    Point* objp = (Point*) ObjList.inlist(objid);
    point_Packet ArgPacket(4);
    point_Packet RetPacket(4);
    int arg_1;
    clnt_rcv(fd, ArgPacket.getBuffer(), 4);
    ArgPacket.setMode(PKT_DECODE);
    ArgPacket.Point_y_21_args(&arg_1);
    RetPacket.setMode(PKT_ENCODE);
    if (objp) {
      objp->y(arg_1);
      RetPacket.pkt_int(&success);
```

```
                    }
                    else
                      RetPacket.pkt_int(&failure);
                    clnt_snd(fd, RetPacket.getBuffer(), 4);
                  }
                  break;

              }
            }

        }
    }
    return 0;
}
```

```
// point_svr.h
//
// This file contains the server definition for rclass Point.
// It is generated by the compiler.

#ifndef POINT_SVRH
#define POINT_SVRH 1

#include <dc++.h>


#define point_PORTNO 55000

#define Point_ID    1000
class Point : public RemoteObj {
    private :
      int xvalue;
      int yvalue;
    public :
      Point();
      Point(int xValue, int yValue);
      int operator<(Point aPoint);
      int operator<=(Point aPoint);
      int operator>=(Point aPoint);
      int operator==(Point aPoint);
      int operator!=(Point aPoint);
      Point operator*(int scale);
      Point operator+(Point delta);
      Point operator-(Point delta);
      Point operator/(int scale);
      Point abs();
      float dist(Point aPoint);
      Point max(Point aPoint);
      Point min(Point aPoint);
      Point transpose();
      int x();
      void x(int aValue);
      void xy(int xValue, int yValue);
      int y();
      void y(int aValue);
};


#define Point_DESTR      0
#define Point_Point_1    1
```

```
#define Point_Point_2       2
#define Point_operator_3    3
#define Point_operator_4    4
#define Point_operator_5    5
#define Point_operator_6    6
#define Point_operator_7    7
#define Point_operator_8    8
#define Point_operator_9    9
#define Point_operator_10   10
#define Point_operator_11   11
#define Point_abs_12        12
#define Point_dist_13       13
#define Point_max_14        14
#define Point_min_15        15
#define Point_transpose_16          16
#define Point_x_17  17
#define Point_x_18  18
#define Point_xy_19         19
#define Point_y_20  20
#define Point_y_21  21


#endif
```

```
// spoint.C
//
// This file contains the functions for the rclass Point.
// These are supplied by the user.

#include <stream.h>
#include <math.h>
#include "point_svr.h"

// Constructor function, creates a point at the default position (0,0)
Point::Point() {

   xvalue = 0;
   yvalue = 0;
}

// Constructor function, creates a point at the specified position
// (xValue, yValue)
Point::Point(int xValue, int yValue) {

   xvalue = xValue;
   yvalue = yValue;
}

// Less than operator
// Returns TRUE if this point is less than aPoint
int Point::operator< (Point aPoint) {
   return((xvalue < aPoint.xvalue) && (yvalue < aPoint.yvalue));
}

// Less than or equals operator
// Returns TRUE if this point is less than or equal to aPoint
int Point::operator<= (Point aPoint) {

   return((xvalue <= aPoint.xvalue) && (yvalue <= aPoint.yvalue));
}

// Greater than or equals operator
// Returns TRUE if this point is greater than or equal to aPoint
int Point::operator>= (Point aPoint) {

   return((xvalue >= aPoint.xvalue) && (yvalue >= aPoint.yvalue));
}

// Equal to operator
```

```
// Returns TRUE if this point is equal to aPoint
int Point::operator== (Point aPoint) {

   return((xvalue == aPoint.xvalue) && (yvalue == aPoint.yvalue));
}

// Not equal to operator
// Returns TRUE if this point is not equal to aPoint
int Point::operator!= (Point aPoint) {

   return((xvalue != aPoint.xvalue) && (yvalue != aPoint.yvalue));
}

// Multiplication operator
// Multiplies this point by factor scale - returns new point
Point Point::operator* (int scale) {

   Point temp(xvalue * scale, yvalue * scale);
   return temp;
}


// Addition operator
// Adds the point delta to this point
Point Point::operator+ (Point delta) {

   Point temp(xvalue + delta.xvalue, yvalue + delta.yvalue);
   return temp;
}

// Subtraction operator
// Subtracts the point delta from this point - returns new point
Point Point::operator- (Point delta) {

   Point temp(xvalue - delta.xvalue, yvalue - delta.yvalue);
   return temp;
}

// Division operator
// Divides this point by factor scale - returns new point
Point Point::operator/ (int scale) {

   Point temp(xvalue / scale, yvalue / scale);
   return temp;
}
```

```
// Absolute value function
// Returns point at position = image of this point in the first quadrant
Point Point::abs() {

   Point temp(::abs(xvalue), ::abs(yvalue));
   return temp;
}

// Function dist
// Returns the distance between this point and aPoint
float Point::dist(Point aPoint) {

   float temp1 = float(xvalue - aPoint.xvalue);
   float temp2 = float(yvalue - aPoint.yvalue);
   return (sqrt(temp1 * temp1 + temp2 * temp2));
}

// Function max
// Returns the maximum of this point and aPoint
Point Point::max(Point aPoint) {

   Point temp;

   temp.xvalue = xvalue > aPoint.xvalue ? xvalue : aPoint.xvalue;
   temp.yvalue = yvalue > aPoint.yvalue ? yvalue : aPoint.yvalue;
   return temp;
}

// Function min
// Returns the minimum of this point and aPoint
Point Point::min(Point aPoint) {

   Point temp;

   temp.xvalue = xvalue < aPoint.xvalue ? xvalue : aPoint.xvalue;
   temp.yvalue = yvalue < aPoint.yvalue ? yvalue : aPoint.yvalue;
   return temp;
}

// Function transpose
// Returns a new point with x and y values of this point transposed
Point Point::transpose() {

   Point temp(yvalue, xvalue);
   return temp;
```

```
}

// Function x
// Returns x coordinate of this point
int Point::x() {

   return xvalue;
}

// Procedure x
// Sets x coordinate of this point to aValue
void Point::x(int aValue) {

   xvalue = aValue;
}

// Procedure xy
// Sets x and y coordinates of this point to xValue and yValue respectively.
void Point::xy(int xValue, int yValue) {

   xvalue = xValue;
   yvalue = yValue;
}

// Function y
// Returns y coordinate of this point
int Point::y() {

   return yvalue;
}

// Procedure y
// Sets y coordinate of this point to aValue
void Point::y(int aValue) {

   yvalue = aValue;
}
```