Reliable Distributed Programming in C++: The Arjuna Approach

Graham D Parrington

Computing Laboratory,
University of Newcastle upon Tyne,
UK
(Graham.Parrington@uk.ac.newcastle)

ABSTRACT

Programming in a distributed system is fraught with potential difficulties caused, in part, by the physical distribution of the system itself. By making the distribution of the system *transparent* it is hoped that the task becomes comparable with that of programming a more traditional centralised system. Object-oriented programming systems are natural starting points for such an attempt due to the inherent modularisation and encapsulation properties they possess. *Arjuna* is one such system, programmed in C++, which permits the construction of reliable distributed applications in a relatively transparent manner.

Objects in *Arjuna* can be located anywhere in the distributed system and are accessed as if they were purely local to the application. The use of remote procedure calls to perform the actual accesses is hidden by the use of stub generation techniques which operate on the original C++ class descriptions thus furthering the illusion of transparency. Reliability is achieved through the provision of traditional atomic transaction mechanisms implemented using only standard language features.

1. Introduction

Although the physical construction of a distributed system is relatively easy today, the programming of such a system remains a complicated task even for the most accomplished of programmers. In addition to the many additional ways in which distributed systems can fail when compared to their more traditional centralised bretheren, the complexity of the communications protocols required to make use of remote resources is often daunting. Consequently many researchers have attempted to overcome these burdens either by providing the illusion that programs are still executing in a centralised environment by making the distribution of the system *transparent* to the programmer, or alternatively by adapting familiar programming techniques and metaphors to the distributed environment (for example, extending the procedure call notion to that of remote procedure call).

Designing and building large complex applications is itself extremely difficult requiring discipline on the part of both designer and programmer. Since such systems are usually too large to be understood in their entirety by a single person, they must be designed and implemented as a set of smaller pieces, each of which is itself sufficiently small to be comprehensible. Many disciplines, some with formal underpinnings, are available to aid this process of decomposition. One such technique that has gained substantial popularity is the object-oriented programming technique whereby the system is partitioned into a set of logical objects that interact with each other to achieve the required system functionality. Each such logical object is self-contained and provides a well-defined interface that permits the orderly interaction between the object and any other objects in the system. Using this paradigm results in systems that are inherently modular, and since each object is self-contained the object-oriented programming paradigm supports the notions of data abstraction and information hiding directly. It is these properties that make object-oriented systems a natural choice for exploitation in programming distributed systems. To the programmer it should not matter where in the distributed system the actual objects are located, all that is required is a means by which operation invocations can be sent to the correct objects wherever they reside. Thus programming a distributed application should be no more complex than programming a centralised application providing that object location and access can be made transparent.

This paper describes the *Arjuna*[16] programming system currently under development at the University of Newcastle upon Tyne. Implemented in C++[18] *Arjuna* provides a number of flexible and integrated mechanisms for naming, invoking operations on (local or remote) objects, concurrency control, recovery control, object state management, etc. This flexibility is achieved by exploiting the inheritance capabilities of the implementation language.

2. Arjuna

While having similar aims to many other research projects (for example, *Emerald*,[4] *Clouds*,[6] *Avalon*,[7] *Argus*,[12] and *Camelot*[17]), *Arjuna* started with one major premise that differentiates it from them all in that the entire system had to be implementable using only standard compilers and systems. Thus it was decided at the outset that modifying a language or it compiler was not permissible. Since object-oriented languages seemed a natural starting point, the ready availability and portability of C++ was immediately appealing. The system that resulted comprises a stub generation and RPC system for distribution purposes, an object store for the storage of persistent objects, and a hierarchy of classes each of which contributes parts of the functionality required by the system as a whole.

Since the original design was conceived under version 1.1 of C++ the design was based upon the use of single inheritance only. Whether multiple inheritance would be an advantage or not remains open at this time. The decision to restrict the implementation of the system to standard compilers only, however, had a major impact in that it required that the programmer make explicit use of the inherited facilities.

The inheritance capabilities of the language provided a basis for the provision of basic capabilities to handle recovery, persistence and concurrency control to the programmer, while at the same time giving flexibility to the system by allowing those capabilities to be refined as required by the demands of the application. Thus the core class hierarchy of *Arjuna* appears to the programmer as the following:

StateManager
    LockManager
        User-Defined Classes
    Lock
        User-Defined Lock Classes
    AtomicAction
    AbstractRecord
        RecoveryRecord
        LockRecord
        and other management record types
etc.

The following sections describe parts of the *Arjuna* system in more detail.

3. Stub Generation

Stub generation in *Arjuna* is different to that employed in other systems[2, 3, 9, 11, 19] in that it does not require the use of a separate Interface Description Language (IDL). Instead the *Arjuna* stub generator[15] is based upon the philosophy that the interface to an object has already been precisely specified (in C++) when the object was originally designed with a non-distributed implementation in mind. To this end the *Arjuna* stub generator accepts as input the standard C++ header files that would normally be input to the C++ compiler. This naturally enhances the transparency of the system in that the programmer need only produce a single object description as if the system was not distributed and rely on the stub generator to produce the distributed version automatically. Similarly, the programmer need not be concerned about mapping the C++ types used to the types supported by the IDL - in *Arjuna* they are the same.

Stub generation is not without its problems which principally stem from the lack of a shared address space between the caller and the actual object. Potential problems include those of:

- *Machine Heterogeneity*. Different machines may have different representations of various primitive data types. For example, byte ordering, arithmetic precision, etc.
- *Parameter Semantics and Types*. Stub generation usually utilises a copy-in, copy-out style of parameter passing which does not necessarily match the semantics of the local parameter passing semantics. Furthermore, certain types of arguments may be disallowed, for example, procedures.
- *Self-Referential Structures*. Linked data structures (which may even be circular) are an obvious source of potential errors
- *Failures*. Failure of an RPC is far more problematical to handle than failure of a local procedure call since the latter typically only occurs when the entire program fails or the error is expected. A procedure executed remotely can fail completely independently of the caller in unexpected ways.

The *Arjuna* stub generator attempts to compensate for these problems are far as it can automatically but there are cases where assistance from the programmer is required. For example, heterogeneity is handled by converting all primitive types to a standard format understood by both caller and receiver.

3.1. RPC System Interface

The RPC mechanism used in *Arjuna* (a multicasting version of *Rajdoot*[13]) is designed for general purpose use, and is thus not language specific. As a consequence it requires the programmer to convert and pack all arguments and results for a call explicitly into the buffers used by the

RPC mechanism. Furthermore it requires clients to provide binding information which details the location of the remote service. Its primary primitives are:

- *Initiate*. This establishes a binding between the client and the server through the use of a manager process listening on a well-known port at the server's site. This manager process handles binding requests by forking an appropriate process and passing it the communication port to allow it to reply to the client. The server process then creates a new port and returns it to the client establishing a direct client-server connection. The manager process is not involved beyond the creation of the server.
- *Terminate*. This primitive terminates the binding between a client and a server and kills the server process.
- *Call*. This primitive performs the actual RPC. Its main parameters are an opcode indicating which procedure to invoke in the server and a buffer for the call arguments.

The actual interface to the underlying RPC mechanism is provided in *Arjuna* via the classes Client_rpc, Server_rpc and rpc which are C++ interfaces to the underlying RPC mechanism. The precise implementation details of these classes are irrelevant to this paper and so will not be presented here. What is important, however, is the interface they provide to the client and server stubs. The class rpc shown below

```
class RPC
{
        ...             // RPC opcode, buffers, etc.
};

class rpc : public RPC
{
public:
        rpc(long = 0);
        rpc(long, Buffer*);
        ~rpc();

        ...             // several useful operations
};
```

is the primary interface to the underlying RPC mechanism. Instances of this class are created and manipulated by both client and server stubs and are passed through the C++ interface to the RPC mechanism. In fact, the stub generator actually interfaces to the system via the classes ClientAction_rpc and ServerAction_rpc. These classes are derived from Client_rpc and Server_rpc and provide identical interfaces to those classes. They exist to ensure that any atomic action management information is correctly propagated between client and server.

This separation of the details of the actual RPC from the interface seen by the generated stub code is important and has many advantages. In particular, stubs can be generated without regard for the actual RPC mechanism used providing that the RPC mechanism complies with the required interface specification. In fact, as a simple experiment the underlying RPC system was replaced by the SUN RPC[20, 21] mechanism without requiring any change to the stub generator or its output.

3.2. Remote server creation

The RPC mechanism requires an explicit call on its *Initiate* and *Terminate* primitives to create and destroy a remote object server and bind it to the client. The problem is when to perform these operations. Fortunately C++ provides a neat solution through the provision of constructor and destructor operations which are normally used for object initialisation and destruction. For remote objects the stub generator creates constructor and destructor operations that call the underlying RPC primitives. The resulting sequence of events then becomes:

- The (stub) object comes into scope in the application program and automatically invokes the object constructor.
- The stub generated constructor binds client to server using *Initiate*.
- A normal RPC is then made to invoke the real constructor for the object in the server passing any arguments supplied to the stub object constructor. This ensures that the remote object has been constructed at the same point in the application as the stub object and utilising the same arguments.
- All further operations are made as RPCs as operations are invoked in the client.
- When the stub object goes out of scope in the application, the generated destructor operation invokes the real destructor in the server via RPC before cleaning up the RPC connection using *Terminate*. This ensures that the real object is destroyed when the stub object is destroyed.

3.3. Parameter marshalling

Implementing remote procedure calls inevitably requires a mechanism by which arguments and results can be transferred between the client and the server. This typically involves packing the arguments into a buffer used by the underlying RPC transport mechanism for transmission and then unpacking them again at the receiving machine. These operations are frequently referred to as marshalling and unmarshalling.

Since the input to the stub generator is C++, there are several distinct kinds of parameters that may need marshalling:

- *Basic Types*. These are the standard built in types such as *char, int, float, unsigned*, etc.
- *Aggregate Types*. Arrays of other types.
- *Classes*. These are the fundamental units of abstraction and encapsulation in C++ and are the primary types for which stub generation is required.

In *Arjuna*, the C++ interface to the RPC mechanism uses an instance of the class Buffer (shown below)

```
class  Buffer
{
        char *buffer_start;
        ...
public:
        Buffer (long);
        ...
        boolean pack (char);
        boolean pack (int);
        boolean pack (double);
        boolean pack (char*);
        ...
};
```

for parameter and result transmission purposes. This class provides overloaded pack and unpack operations that allow simple types to be marshalled and unmarshalled to and from the buffer with ease. Using this class an individual variable of basic type (say *int*) can be marshalled via the simple statement:

BufferInstance.pack(variable);

Similar statements can be generated to marshall arrays, etc., however, complications arise in marshalling and unmarshalling instances of classes. Recall that class instances are assumed to be encapsulated entities, thus only member or friend functions can access the internal state variables. The stub generator, therefore, augments the operations of any class with the two additional public member functions:

```
virtual void marshall (Buffer&);
virtual void unmarshall (Buffer&);
```

In order to avoid having to generate different code to marshall variables of different types the stub generator exploits the operator overloading capabilities of the language. Traditionally in C++, the operators ‘<<’ and ‘>>’ are overloaded to allow input and output of variables, including instances of classes (given suitable definitions of these operations by the programmer). The stub generator makes use of precisely the same technique, but defines the same operators to be the equivalent of marshall (<<) and unmarshall (>>) when applied to RPC buffers. The nett result of this is that variables of any type can always be marshalled using a statement of the form:

BufferInstance << variable

As an example consider the following simple class definition:

```
class Date
{
        // TRANSMISSIBLE

        int day, month, year;
public:
        Date (int, int, int);
        void set (int, int, int);
};
```

This class represents an object which will not be accessed via RPCs but may be sent as a parameter to some other remote object. When processed by the stub generator the resulting output header and the code generated for the marshall operation and redefinition of the operator << are shown below.

```
class Date
{
        int day, month, year;
public:
        Date (int, int, int);
        void set (int, int, int);
        virtual void marshall (Buffer&);
        virtual void unmarshall (Buffer&);
};

inline  Buffer& operator<<  ( Buffer& rpcbuff, Date& topack )
{
        topack.marshall(rpcbuff);
        return rpcbuff;
}

void Date::marshall ( Buffer& rpc_buff )
{
        rpc_buff << day;
        rpc_buff << month;
        rpc_buff << year;
}
```

As can be seen, the public interface to the class is unchanged beyond the addition of the marshalling operations and instances of this class behave as they would have if the stub generator had not been used.

3.4. Remote objects

The stub generator currently distinguishes between objects designed to be accessed remotely and those that are merely transmitted as arguments. Only in the former case is full stub code generation required (the latter only needs marshalling code generation as shown in the previous section).

For remotely accessible objects code must be generated for all of the public operations supported by the object together with an appropriate server capable of decoding the incoming RPCs and dispatching them to the correct operation in the real object. Thus the stub generator produces two distinct code fragments - one for the client, the other for the server.

For example, consider the simple class definition given below:

```
#include "Date.h"

class Calendar
{
        Date today
        ...
public:
        Calendar ();
        ~Calendar ();

        void setdate (Date);
        void getdate (Date&);
        ...
};
```

This class allows the user to create and manipulate remote calendar instances (for the purpose of arranging a meeting perhaps). Running this class definition through the stub generator yields the following two class definitions

```
class  CalendarClient
{
        ClientAction_rpc *clientclass_rpc;
        Group *clientclass_server;
public:
        CalendarClient (ClientAction_rpc * = 0);
        void setdate (class Date);
        void getdate (class Date &);
        ...
};

class CalendarServer
{
        Calendar *therealclass;
        rpc *Calendar_1002 (Calendar *, rpc *);
        rpc *Calendar_1003 (Calendar *, rpc *);
        rpc *setdate_1004 (Calendar *, rpc *);
        rpc *getdate_1005 (Calendar *, rpc *);
public:
        CalendarServer ();
        void Server (int, char **);
        rpc *Dispatch (Calendar *, rpc *);
};
```

As can be seen, the client class (CalendarClient) provides the same public interface as the original class but its instance variables and the implementation of all of the operations have changed. Suitable name mapping tricks played with the standard preprocessor ensure that the programmer can still use this class under its original name maintaining transparency.

The generated server class (CalendarServer) contains a pointer to the real object which is only assigned when the object is actually created and initialised when an RPC for the constructor arrives. Similarly the object is destroyed when the RPC for the destructor arrives. Sample code generated for the getdate routine is as follows:

```
void CalendarClient::getdate (class Date& Par_0)
{
        rpc *callrpc = 0;
        rpc *result = 0;
        int validresult = 0;

        Buffer& rpc_callbuff = * new Buffer();
        rpc_callbuff << Par_0;

        callrpc = new rpc(1005, &rpc_callbuff);
        result =  clientclass_rpc→Call(clientclass_server, callrpc);

        if ((result != 0) && (result→get_opcode() == S_DONE))
        {
                Buffer& rpc_retbuff = *result!get_buffer();
                rpc_retbuffer >> Par_0;
        }

        if (callrpc != 0)
                delete callrpc;
        if (result != 0)
                delete result;
}
```

3.5. Failure handling

The handling of RPC failures is the major problem in stub generation. This problem is actually exacerbated by the use of C++ since return values may be arbitrary complex objects. Unfortunately, there is no automatic solution to this problem. The stub generator only knows that operations pass and return instances of particular types when invoked and relies on being able to initialise a return value by unmarshalling it from the RPC reply. If the RPC fails the stub generator has no way of automatically producing code to return an error instance of the return type since it has no knowledge of how to construct such an instance. Handling such failures then becomes the task of the programmer. One possible approach, however, is to extend the current mechanism utilised to control the actual stub generation process (via special comments in the class definition) to include details of special error versions of objects which can be returned if an RPC fails. The current implementation takes the cavalier attitude that doing nothing is the safest approach, thus if the RPC call fails output parameters remain unset and nothing is returned

4. Reliability

Reliability of an *Arjuna* application is based upon the use of the well known atomic action (atomic transaction) concept.[10] Atomic actions have the properties of

- *Failure Atomicity*. All of the operations that comprise the action complete successfully or none of them do.
- *Serialisability*. The concurrent execution of actions is equivalent to some serial order of execution
- *Permanence of Effect*. Once completed new system states produced by actions are not lost.

Actions are implemented in *Arjuna* by the class AtomicAction which provides operations that correspond to those that would be familiar to any database programmer - Begin, End, and

Abort.

Instances of AtomicAction must be explicitly declared and used by the programmer as appropriate to the application. Instances of AtomicAction only control when certain events happen on objects, that is, if an atomic action aborts any objects manipulated within it must have their prior state restored and concurrency control information updated. In order to accomplish this AtomicAction instances maintain a list of instances of classes derived from AbstractRecord. Each of these classes manages a certain property, thus RecoveryRecords manage object recovery, LockRecords manage concurrency control information etc. It is thus sufficient for the operations of AtomicAction to run down this list at runtime invoking the appropriate operation (for example, top_level_commit) on each record instance. This record based approach provides complete flexibility in that new record types can be created as required (other record types currently handle persistence, distribution and object scope).

4.1. Recovery and Persistence

The failure atomicity and permanence of effect properties of atomic actions require that objects be both recoverable and persistent. These capabilities are provided by the classes StateManager and ObjectState. StateManager is the root base class of the entire *Arjuna* class hierarchy. As such it provides naming capabilities (in the form of unique identifiers), operations for the activation and deactivation of persistent objects along with state-based recovery. Since recovery and persistence have complimentary requirements (the only difference being where information is stored and for what purpose) the system uses instances of ObjectState for both purposes. When not in use persistent objects are stored in a passive form in an object store as instances of the class ObjectState. When first used they are automatically activated by the system which results in the conversion of the object into its standard runtime form. Deactivation occurs when the top-level action commits at which time the object is again converted back into an ObjectState for storage in an object store. Instances of ObjectState maintained for recovery purposes are held entirely in memory. More precise implementation details can be found in reference (8).

Conversion of objects between their passive and active forms is controlled by the system but uses operations supplied by the programmer. These operations, save_state and restore_state, must be provided (this is enforced by the compiler in R2.0 by making them pure virtual functions) otherwise neither the recovery system nor the persistence mechanisms will function properly.

4.2. Concurrency Control

Concurrency control in *Arjuna* is the heavyweight control required for the provision of the serialisability property of atomic transactions.[1, 14] It is not the lightweight control supplied by such constructs as semaphores or monitors. Concurrency control is provided by the system in the form of strict two-phase locking which by default supports a traditional multiple reader, single writer policy. The concurrency controller is implemented by a class derived from StateManager (the class LockManager) in conjunction with the class Lock. User defined objects are derived from LockManager and thus inherit concurrency control capabilities from it, and recovery, naming and persistence capabilities through it from StateManager.

LockManager is solely responsible for managing requests to set a lock on an object or release a lock as appropriate. It deliberately has no knowledge of the semantics of the actual policy by which the lock requests are granted. Such information is maintained by the Lock class instances which provide operations by which LockManager can determine if two locks conflict or not. This separation is important in that it allows the programmer to derive new lock types from the basic Lock class and by providing appropriate definitions of the conflict operations enhanced levels of concurrency are possible.

5. Exploiting C++ Syntax

One criticism that could be levelled at the original *Arjuna* system is that it required explicit usage by the programmer to be effective. That is, the programmer must declare and use atomic actions appropriately, and make correct calls to the concurrency controller at the appropriate points in the code for an operation. Neglecting to do so was likely to lead to some potentially highly undesirable results. Thus, while the system provided great flexibility it also required careful use to avoid potential problems.

This criticism stemmed from the original design decision to use standard ‘off-the-shelf’ components in building the system. From the outset there was never any intention to either modify the C++ language itself or add features into any particular compiler. *Arjuna* was designed to be portable to any system that had a C++ compiler (regardless of origin) and ran Unix. Thus it was impossible to determine whether any operation modified a particular object without help from the programmer, since determining this fact would require semantic analysis of the particular operation.

However, with the release of R2.0 of the C++ language this criticism need no longer apply since it is possible to generate appropriate calls to the *Arjuna* system automatically as part of the stub generation process using only syntactic analysis of the signature of any particular operation. The key to this feature is the exploitation of the fact that operations can now be marked as *const* which implies that they do not modify the object to which they are applied. Utilising this syntactic marker allows the stub generator to insert calls to the concurrency controller to create read locks automatically when *const* operations are invoked, and to create write locks when non-*const* operations are invoked. In addition operations can have code generated that automatically creates and starts an atomic action when the operation is invoked, and commits or aborts it depending upon the result of the operation.

6. Retrospective

In retrospect C++ has proved to be an adequate language for programming a reliable distributed system like *Arjuna*. Where it has failed it has done so in part because of the actual implementation of the language by the currently available compilers and due to lack of certain facilities at the language level. In the following sub-sections these failures are examined in more detail.

6.1. Distribution and Stub Generation

The ultimate aim of the stub generation system is to take a standard C++ header file defining some class and produce from it an equivalent header containing a replacement stub-class definitions together with the code that implements the client and server operations of the stub classes.

This approach has several potential problems that arise due to a variety of causes. Firstly, typical C++ header files contain not only class definitions but also inline function definitions, macros, manifest constants, various preprocessor directives etc. Secondly, type information about parameters, etc. may be incomplete. For example, declaring a variable to be a pointer to some type does not imply that the variable points to a single instance of that type - it might actually point to an array (particularly if the type is *char* * - that is, a pointer to character which by convention is used to represent a string), or a list, etc. For this reason, the stub generator imposes certain semantics upon pointers such that a pointer to a type always points to only one instance of that type - even for the *char* type. While this may seem to impose restrictions, classes can be written for strings etc. which make this restriction an inconvenience at worst.

Other problems arise due to the fact that, as was noted earlier, remote procedure calls typically provide a restricted set of capabilities over traditional calls. Thus the stub generator imposes the following restrictions upon the form of the class and the operations its supports:

- No public variables. The existence of public variables in a class definition breaks the fundamental encapsulation property assumed by the stub generator, so any such variables are removed automatically (with warnings) as the class definition is processed.
- Variable length argument lists to operations are disallowed since the stub code needs to know exactly the type and number of arguments to each operation for marshalling purposes. This restriction could be removed by allowing the programmer to specify a marshalling routine for such operations explicitly.
- Inline operation definitions are currently discarded as they make no sense to the stub generated version of the class at the client. Future versions of the stub generator will ensure that such inlines are made available in the server.
- Classes cannot contain static members since the stub generator cannot enforce the semantics of such variables correctly.
- Lack of support for operator overloading.

Some of these restrictions are due to weaknesses in the stub generator (for example, inlines and operator overloading) and will be removed in future versions. Others are fundamental to the technique of stub generation (static member variables) and will remain compromising the level of transparency achievable.

Using C++ as both the input and output language of the stub generator has had both benefits and problems. On the benefit side, application level location and access transparency is achieved due to the complete lack of a separate interface definition language. Similarly, exploitation of the operator overloading capabilities of the language in the output code has reduced the complexity of parameter marshalling considerably.

6.2. Reliability

Since atomic actions are not part of the actual implementation language there is a problem related to the scope of objects. Ideally the scope of an atomic action and of any objects it controls should be the same. However, it is easily possible to create programs where this is not true. For example by creating an action in an outer block, and an object in some inner block. In this case the object is automatically destroyed as the inner block is exited despite the fact that its ultimate fate has not yet been decided since the atomic action under whose control its operations have executed is still active. The *Arjuna* system attempts to compensate for this as much as possible by retaining object states until the outcome of the controlling action is known if the object is destroyed prematurely. Similar problems arise if objects are created and deleted on the heap while an atomic action is active. The problem then is how can the deletion of a heap object (and the associated freeing of memory) be undone if the action that did the delete aborts.

The lack of automatic garbage collection of objects is also problematical, since in a distributed system it is very easy to lose track of who is responsible for finally deleting a particular object, the nett result of which is objects are active longer than necessary tieing up potentially valuable system resources.

Finally the lack of an exception handling mechanism remains a current problem (which may be eventually solved if the proposed exception handling mechanism is ever added to the language) which makes the transparent handling of RPC failures difficult if not impossible to implement.

6.3. Recovery and Persistence

The implementation of recovery and persistence is currently based upon the use of programmer supplied save_state and restore_state routines. Although the system uses R2.0 language features to force definition of these operations (pure virtual functions) errors in coding these routines can bring chaos. Such explicitly provided routines are needed because current compilers do not carry over type information about class member variables to execution time. If such information was available standard versions of save_state and restore_state could process it and obviate the usual need for the programmer to provide such routines. However, even then, there would still arise occasions where programmer assistance is required - particularly in the case where objects contain pointers to other objects, in order to determine precisely what constituted the state of any object. Similar techniques as those employed by the stub generator for producing the marshalling operations could be employed (that is save_state is almost equivalent to marshall) but that approach will never be completely automatic and will continue to require programmer assistance in several cases.

7. Conclusions

Object-oriented programming languages and systems still provide an ideal starting point for programming distributed applications and operating systems (for example, *Choices*[5]) due to their inherent modularity and encapsulation properties. However, using standardly available implementations of such systems may not provide sufficient support upon which to build distributed applications. It is in this area that C++ fails principally - current implementations are too biased towards an shared memory model of execution. Instead what may be required is a new implementation of the language such that the compiler can provide extra information not normally provided by the standard implementations - for example additional information about types and how they are laid out in memory, etc. If such a compiler existed and the proposed language extensions for exception handling, etc., were implemented within it, then C++ would be a far more realistic vehicle for implementing a reliable distributed system than it currently is.

8. Acknowledgments

The work reported here has been supported in part by grants from the UK Science and Engineering research Council and ESPRIT project No. 2267 (Integrated Systems Architecture).

References

1. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, 1987.
2. B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo, and M. Schwartz, ‘‘A Remote Procedure Call Facility for Interconnecting Heterogenous Computer Systems,’’ *IEEE Transactions on Software Engineering*, vol. SE-13, no. 8, pp. 880-894, August 1987.
3. A. D. Birrell and B. J. Nelson, ‘‘Implementing Remote Procedure Calls,’’ *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39-59, January 1984.
4. A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, ‘‘Distribution and Abstract Types in Emerald,’’ *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 65-76, January 1987.
5. R. H. Campbell, V. Russo, and G. Johnston, ‘‘Choices: The Design of a Multiprocessor Operating System,’’ *Proceedings of the USENIX C++ Workshop*, Santa Fe, November 1987.
6. P. Dasgupta, R.J. LeBlanc, and E. Spafford, ‘‘The Clouds Project: Designing and Implementing a Fault Tolerant Distributed Operating System,’’ Technical Report GIT-ICS-85/29, Georgia Institute of Technology, 1985.
7. D. Detlefs, M.P. Herlihy, and J.M. Wing, ‘‘Inheritance of Synchronization and Recovery Properties in Avalon/C++,’’ *IEEE Computer*, vol. 21, no. 12, pp. 57-69, December 1988.

8. G. N. Dixon, G. D. Parrington, S. K. Shrivastava, and S. M. Wheater, "The Treatment of Persistent Objects in Arjuna," *The Computer Journal*, vol. 32, no. 4, pp. 323-332, August 1989. (also in Proceedings of the Third European Conference on Object-Oriented Programming ECOOP89, ed. S. Cook, pp. 169-189)

9. P. B. Gibbons, "A Stub Generator for Multilanguage RPC in Heterogeneous Environments," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 77-87, January 1987.

10. J. N. Gray, "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, ed. R. Bayer, R. M. Graham and G. Seegmueller, pp. 393-481, Springer, 1978.

11. M. B. Jones, R. F. Rashid, and M. R. Thompson, "Matchmaker: An Interface Specification Language for Distributed Processing," *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pp. 225-235, January 1985.

12. B. Liskov, "Distributed Programming in Argus," *Communications of the ACM*, vol. 31, no. 3, pp. 300-312, March 1988.

13. F. Panzieri and S. K. Shrivastava, "Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing," *IEEE Transactions on Software Engineering*, vol. SE-14, no. 1, pp. 30-37, January 1988.

14. G. D. Parrington and S.K. Shrivastava, "Implementing Concurrency Control for Robust Object-Oriented Systems," *Proceedings of the Second European Conference on Object-Oriented Programming, ECOOP88*, pp. 233-249, Oslo, Norway, August 1988.

15. G. D. Parrington, "Distributed Programming in C++ via Stub Generation," Technical Report, Computing Laboratory, University of Newcastle upon Tyne, January 1990.

16. S. K. Shrivastava, G. N. Dixon, and G. D. Parrington, "An Overview of Arjuna: A Programming System for Reliable Distributed Computing," *IEEE Software*, to be published.

17. A. Z. Spector, R. Pausch, and G. Bruell, "Camelot: A Flexible, Distributed Transaction Processing System," *Proceedings of CompCon 88*, pp. 432-439, February 1988.

18. B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, 1986.

19. Sun Microsystems Inc., "Rpcgen Programming Guide," in *Network Programming Manual*, 1988.

20. Sun Microsystems Inc., "External Data Representation Standard Protocol Specification," in *Network Programming Manual*, 1988.

21. Sun Microsystems Inc., "Remote Procedure Call Protocol Specification," in *Network Programming Manual*, 1988.