A Stub Generation System for C++

Graham D. Parrington

The University of Newcastle upon Tyne

ABSTRACT: This paper describes the implementation of a Stub Generation system targeted specifically at C++. It enables distributed C++ applications to be constructed in a straightforward manner with minimal programmer assistance. The system does not require the use of an auxiliary interface definition language but instead processes existing C++ header files to maximize transparency. The generated code exploits the power of C++ by using operator overloading for parameter marshalling and constructors and destructors to drive the remote binding process. The system described here is fully implemented and can be obtained as part of the Arjuna programming system.

1. Introduction

Modern computing and networking hardware makes the physical interconnection of many computers relatively simple to achieve. However, programming an application to take even limited advantage of the interconnection is still considered difficult even for the most accomplished of programmers. Much notable research effort has concentrated upon methods by which the underlying distribution of the system can be hidden. This attempt to achieve what is termed *distribution transparency* effectively reduces the burden on the programmer to that involved in programming a more traditional centralized application. However, this transparency has generally been achieved by the creation of entirely new programming languages or systems (for example: Emerald [Black et al. 1987], Clouds [Dasgupta et al. 1985], Avalon [Detlefs et al. 1988], Argus [Liskov 1988], and Camelot [Spector et al. 1988]). Although such languages and systems often include many other useful or desirable capabilities in addition to pure distribution (such as transaction mechanisms), real world demands insist that applications need to be programmed in an existing, preferably widely available, language.

Regrettably, most widely available languages (that is, available in the commercial arena) have little or no direct support for programming a distributed application. The primary reason for this focus is that existing languages have been developed without the demands of distribution in mind and thus possess one or more features that are either impossible, or at least impractical, to support in a distributed execution environment. Thus, although full distribution transparency is typically impossible to achieve in existing languages, a partial form of transparency is both achievable and highly desirable. Computing platforms and architectures that support this philosophy are emerging from several sources (for example: Integrated Systems Architecture (ISA) [APM 1991], Open Network Computing (ONC) [Sun 1988.24], the Open Software Foundation Distributed Computing Environment (OSF/DCE) [OSF 1991], and the Object Management Group Common Object Request Broker Architecture (CORBA) [OMG 1991]).

These latter systems tend to be targeted primarily towards applications written in the C language and must therefore use a separate *Interface Definition Language* to describe the interface to the remote service. In contrast, the system described

here is targeted solely at the language C++ [Stroustrup 1986]. The choice of C++ was motivated by its use in the Arjuna system [Shrivastava et al. 1991], however, the stub generator was deliberately designed to operate in a standalone fashion and to distribute transparently as much of C++ as was possible (with some restrictions which will be described in a later section). It was felt that the use of C++ might obviate the need for a separate IDL by using class declarations as descriptions of the interface to remote objects thus promoting increased transparency. Hence, the input language to the stub generator is C++ and it produces C++ as its output. This has some advantages and allows the full power of the language to be used in the generated stub code, leading to elegant and flexible marshalling code for example. Furthermore, in keeping to the philosophy of the C++ language, the system deliberately does not address issues such as concurrency and synchronisation which are deemed to be the responsibility of higher level services. For example, when used in the Arjuna system the latter's transaction mechanism supplies appropriate concurrency control.

The remainder of the paper first describes the important design decisions made followed by a description of the fundamental technique of stub generation (which can be skipped by seasoned practitioners). Following that is a description of the code the stub generator produces, showing how it copes with features of the language that are not often necessary in the other languages distributed by this technique such as inheritance (both single and multiple), templates, and exceptions. The paper closes with some conclusions on the effectiveness of this approach and the use of C++ as an interface definition language.

2. *Principle Design Decisions*

Several important decisions governed the design and implementation of the system, some of which have been alluded to already:

1. The stub generator should process standard C++ header files that would normally be acceptable to any C++ compiler. Thus any stub generation specific information would have to be placed within comments normally ignored. Its output would also be standard C++ (preferably acceptable to the majority of existing compilers thus precluding the use of some of the newer features of the language).
2. The unit of distribution was to be the C++ class. This would allow individual objects to be accessed remotely. Only public operations would be remotely invocable.

3. The stub system would only handle those problems associated with distribution; that is principally parameter passing and operation invocation. Other issues such as service sharing and synchronisation are considered to be concerns best handled in other sub-systems.
4. Client code should remain unaltered if at all possible. That is, the original client application should compile with either the original class declarations or with the stub produced declarations without source changes.
5. Stub generation should be modular. That is, if a base class header changes it should only be necessary to regenerate the stubs for that class. Stubs already in existence for derived classes should be unaffected.

3. *Principles of Stub Generation*

Within a distributed environment, applications may access services that are either local to, or remote from, the node upon which the application is currently executing. Given that the service can only be accessed by invoking one of its operations, then access to a remote service requires a mechanism that supports the remote execution of an operation. This effectively requires the implementation of a communications protocol responsible for the orderly transfer of control between the invoker of the operation (the caller or *client*) and some remote *server* that will actually execute the operation. Besides the transfer of control a means must also be provided for the transfer of any arguments required by the called operation and the return to the client of any results. This mechanism is termed *remote procedure call* (RPC [Bershad et al. 1987, Birrell & Nelson 1984]) and represents a natural extension of the traditional notion of procedure call to the distributed environment.

Conceptually, a distributed application consists of several distinct fragments split between the original calling process (client) and a remote server process responsible for executing the requested operations locally. These fragments are known as: the client, the client stubs, RPC transport, the server stubs, and the server. Both the client and server are typically designed and implemented as if the application was to execute in a traditional centralised environment. It is the function of the client and server stubs to hide the underlying distribution to as great a degree as possible. Since production of these stubs can be tedious and complicated the process can be automated by the use of a *Stub Generator* [Gibbons 1987, Jones et al. 1985, Parrington 1992, Sun 1988.23]. This parses a description of the interface between the client and the server, written in some *Interface Definition Language* (IDL), and produces the required stub code in a language compatible

with both. In many systems this interface description language has a different syntax and semantics to the language in which the application is programmed. By way of comparison the stub generation system described here is designed to work with a single language and thus requires no separate IDL. Instead it processes the original C++ header files that would normally be read by the standard compiler.

Each type of IDL has its advantages and disadvantages. An IDL that is programming language independent may allow different parts of the distributed application to be programmed in a different language suitable to the task. Furthermore, since such IDLs are typically more constraining they can force a programmer to be aware of the difference between local and remote processing and forbid use of certain potentially dangerous constructs normally available in the programming language. In addition, they may also augment the language with functionality not normally provided; for example, by providing new types such as strings and dynamic arrays, or new constructs such as exception handling. However, the disadvantages are that the programmer is required to map the original interface description from the host language to the IDL before the stub generator can operate, thus losing transparency; and also it becomes possible for the IDL description not to match what is actually implemented if each is updated independently of the other. Furthermore, many IDL systems actually require that the programmer write substantially different code for the server (by renaming the functions, for example) to the code that would be written if the application was not distributed.

On the other hand a programming language specific IDL aids transparency and the interface and implementation should not diverge as easily as with a separate IDL. Unfortunately, such transparency can lull the programmer into a false sense of security since the whole language will typically not be distributed and the inherent costs of distribution are not apparent.

In general, stub generation is not without its problems that principally stem from the lack of a shared address space between the caller and the actual service being manipulated. Potential problems include those of:

Machine Heterogeneity. Different machines may have different binary representations of various primitive data types. For example, different byte orderings and floating point number representations; different arithmetic precision (16 vs. 32 bit); unusual pointer representation; etc. The commonest solution to this problem requires the client and server stubs to convert the native format to some common format (for example, ASN.1 or XDR) prior to transmission. Such conversion is potentially costly and unnecessary between machines of the same type. However, the simplicity of the approach often outweighs this problem in a true heterogeneous environment.

Parameter Passing Semantics and Types. Different languages have different semantics governing parameter passing, such as call by value and call by reference to name but two. Remote procedure call usually enforces a copy-in, copy-out style of parameter passing which does not necessarily match the semantics of the local parameter passing mechanism. Furthermore, certain types of arguments may have to be disallowed entirely, for example, procedures.

Self-Referential Structures. Most modern programming languages allow the creation of linked data structures whereby a given data structure contains pointers to other data structures. This facility provides the programmer with a very flexible mechanism but may cause problems for stub generation systems that must usually marshall the entire structure if a single element of it is passed as a parameter. Circular data structures are potentially even more problematic to handle.

Failures. Failure of an RPC is far more problematic 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 many unexpected ways.

4. Implementation

4.1. Assumptions

Since the stub generator is only aimed at distributing C++ programs it tries to exploit the capabilities of the language to the full. In particular the C++ constructor and destructor notions provide a convenient handle on remote server creation and destruction. Furthermore, operator overloading is used in the marshalling and unmarshalling of parameters. The input is assumed to be a syntactically correct C++ header file (that is, the stub generator is not intended to replace the compiler as a method of error detection) describing one or more classes.

The unit of distribution is the C++ class, enabling individual C++ objects to be distributed over the system. The interface the distributed objects support is the public *operation* set of the class—all other public entities are removed (with appropriate warnings).

4.2. Primary Operation

The stub generator reads a standard C++ header the name of which is supplied as a command line argument. Since this file typically contains many pre-processor

directives (#include etc.) it must first be processed to remove these. This is handled by feeding the input to the standard C++ compiler (the actual compiler used is the compiler used to compile the stub generator itself) but instructing it to preprocess the file only. The resulting output is then read, parsed and the appropriate stubs produced.

The standard compiler is used to ensure that any special directives normally passed by it (usually in the form of special definitions of system variables) are used to produce the stub generator's input. If the standard compiler cannot be found, several alternatives are tried in sequence. These alternatives have been ascertained based on experience porting the stub generator to various UNIX implementations.

4.3. Interfacing to the Underlying RPC

In order to be as portable as possible the stub generator places as few demands on the underlying RPC system as it can. In particular it requires only the ability to initiate a connection to some remote server, a means of making actual calls, and a method of breaking the connection. 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.

The stub generated code uses only three classes as its interface to the RPC mechanism: `ClientRpcManager`, `ClientRpc`, and `ServerRpc`. As expected `ClientRpc` represents the client side view of the RPC mechanism and provides operations to initiate an RPC connection (`initiate`), perform a remote call (`call`), and break the RPC connection (`terminate`). The operation `initiate` should establish a binding between the client and the server through whatever mechanism the underlying RPC mechanism provides using the information provided by the `ServiceName` object supplied as a parameter. Naturally, `terminate` breaks the binding between a client and a server, while `call` performs the actual RPC. The main parameters to `call` are an opcode indicating which operation to invoke in the server and buffers for the call arguments and returned results, together with two status flags.

```
class ClientRpc
{
public:
    //
    // Fundamental generic operations provided by
    // the RPC interface.
    //
```

```
    ClientRpc (ServiceName* Name);
    ClientRpc (const char* serviceName = 0);
    virtual ~ClientRpc();

    RPC_Status initiate();
    RPC_Status call(Int32 opCode, RpcBuffer& callBuff,Int32& errCode,
                            RpcBuffer& result);
    virtual RPC_Status terminate();
};
```

Similarly, the server side of the connection is handled by `ServerRpc` which provides operations to receive an incoming request (`getWork`) and return some results (`sendResult`).

```
class ServerRpc
{
public:
    ServerRpc ();
    virtual ~ServerRpc ();

    int initialise (int argc, char *argv[]);
    void getWork (Int32& opCode, RpcBuffer& call);
    void sendResult (Int32 errCode, RpcBuffer& result);
};
```

Normally the stub generated code does not invoke any of the client side operations directly. Instead this is handled by the third class `ClientRpcManager`. The constructor for this class invokes `initiate`, while its destructor invokes `terminate`. Naturally it exports the `call` operation unmodified. This approach ensures that client/server connection and disconnection is handled simply by creating and deleting instances of the control class.

```
class ClientRpcManager
{
public:
    ClientRpcManager (ServiceName *);
    ClientRpcManager (ClientRpc * = 0);
    ClientRpcManager (const char *);
    virtual ~ClientRpcManager ();

    RPC_Status call (Int32, RpcBuffer&, Int32&, RpcBuffer&);
    void rpcAbort () const;
    rpcAbortHandler setHandler(rpcAbortHandler);
```

```
private:
    int initiated;
    ClientRpc *rpcHandle;
};
```

4.4. Primary Mechanisms

4.4.1. 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.

The default RPC mechanism used in testing the stub generator is a version of Rajdoot [Panzieri & Shrivastava 1988]. Rajdoot is designed for general purpose use, and is thus not language specific. As a consequence it requires the programmer to convert and pack all parameters and results for a call explicitly into the buffers used by the RPC mechanism. C++ operator overloading is used to simplify considerably the code required to marshall (encode) and unmarshall (decode) arguments to and from these underlying RPC buffers. In particular, the operators >> and << have been adopted for this purpose (similar to their use in the C++ I/O system). Thus << is used to marshall arguments into the buffers used by the RPC mechanism, and >> to unmarshall arguments from the buffers regardless of the actual type of the argument. The RPC buffer class (`RpcBuffer`) provides a set of operations that permit the marshalling and unmarshalling of all of the basic types of C++ (int, char, etc.). The marshalling of more complex structures is simply achieved by breaking the structure up into its component parts and marshalling each independently. The actual encoding scheme currently used is the same as that used by the persistence mechanisms in Arjuna that enable a C++ object to be stored on disk (that is `RpcBuffer` is derived from the class `Buffer` and uses its `pack` and `unpack` operations directly). There is, however, no reason why some other scheme (say XDR or ASN.1) could not also be used.

```
/*
 * Class to handle RPC buffering. Based upon the standard Arjuna
 * Buffer class
 *
 */

class RpcBuffer : public Buffer
```

```
{
public:
    /* Constructors and destructor */

    RpcBuffer ();
    RpcBuffer (Int32 initVal);
    RpcBuffer (const RpcBuffer& copyFrom);
    RpcBuffer (const RpcBuffer& copyFrom, Int32 initVal);
    virtual ~RpcBuffer ();

    /* Standard marshalling operations */

    RpcBuffer& operator<< (char);
    RpcBuffer& operator<< (unsigned char);
    RpcBuffer& operator<< (short);
    RpcBuffer& operator<< (unsigned short);
    . . .

    /* Standard unmarshalling operations */

    RpcBuffer& operator>> (char&);
    RpcBuffer& operator>> (unsigned char&);
    RpcBuffer& operator>> (short&);
    RpcBuffer& operator>> (unsigned short&);
    RpcBuffer& operator>> (double&);
    . . .

};
```

Since all C++ objects are treated as encapsulated entities, the stub generator ensures that suitable definitions exist for these marshalling operators for all objects passed as arguments—even class objects which must have their public operation set augmented by the inclusion of the operations for (un)marshalling.

Arguments passed by pointer or reference require special handling. By default these are treated as in/out parameters and are both sent in the call and assumed to be returned as part of the result. This behaviour can be modified in two ways. Firstly, if the argument is declared to be `const` then it is automatically treated as input only. Secondly, the programmer can augment the declaration of an argument with stub generation specific commands to guide the process explicitly. This topic will be discussed and expanded further in a later section.

Thus the following class declaration and marshalling code is one sample output:

```
class AnAppointment
{
public:
```

```
    AnAppointment ();
    ~AnAppointMent ();

    // Ignore other operations here for clarity
    ...

    // These are the added marshalling operations
    void marshall (RpcBuffer&) const;
    void unmarshall (RpcBuffer&);
private:
    time_t start;
    time_t end;
    String description;
    Boolean confirmed;
};

// Overload << to marshall instance into buffer
inline RpcBuffer& operator<< ( RpcBuffer& rp,
                               const AnAppointment& topack)
{
    topack.marshall(rp);
    return rp;
}

// Marshall each variable in turn
void AnAppointment::marshall ( RpcBuffer& rpc_buff ) const
{
    rpc_buff << start;
    rpc_buff << end;
    rpc_buff << description;
    rpc_buff << confirmed;
}

// Unmarshalling operations are similar only using the operator >>
```

4.4.2. Client and Server Classes

For each class declaration that it reads from its input file the stub generator will (when appropriate) generate three new class definitions. These class definitions represent:

1. The replacement class for use by the programmer in the client application. This is the mechanism whereby transparency is achieved since the replacement class has the same set of public operations as the original and can thus be substituted for it without the programmer's knowledge.

2. The server stub class responsible for decoding an incoming RPC request, unmarshalling any incoming parameters, invoking the required operation, and marshalling and returning any output values prior to returning control to the caller.
3. A renamed version of the original input class that is instantiated in the server as required.

For example, this class definition:

```
#include "AppointMent.h"

// The following stub specific commands are actually the default
// @Remote, @NoMarshall
class Diary : public LockManager
{
public:
    Diary(ServiceName AN);
    ~Diary();

    String WhereIs(time_t now, String user);

    AnAppointment GetNextAppointment(time_t now);
    int AddAppointment(AnAppointment entry);
    int DelAppointment(time_t when);

    virtual Boolean save_state(ObjectState&, ObjectType);
    virtual Boolean restore_state(ObjectState&, ObjectType);
    virtual const TypeName type() const;

private:
    String   user_name;
    AnAppointment *appts;
};
```

would result in the generation of the definitions and supporting code shown in the following sub-sections.

4.4.3. Client Interface

```
class RemoteDiary : public RemoteLockManager
{
public:
    RemoteDiary (ServiceName, ClientRpcManager *crpc = 0);
    ~RemoteDiary ();

    String WhereIs (time_t, String );
    AnAppointment GetNextAppointment (time_t );
```

```
    int AddAppointment (AnAppointment );
    int DelAppointment (time_t );
    virtual Boolean save_state (ObjectState&, ObjectType );
    virtual Boolean restore_state (ObjectState&, ObjectType );
    virtual const TypeName type () const ;

protected:
    RemoteDiary(const ClientRpcManager&, const RpcBuffer&);

private:
    ClientRpcManager _clientHandle;
    RpcBuffer _myHashVal;
};
```

Simple renaming tricks played using the standard pre-processor enable this class to be transparently used under its original name in the programmer's application code.

This generated client stub class has the same set of public operations as the original (although any constructors have had an extra argument added to them, this is effectively invisible and the code written to use instances of the original class will still compile). Public instance variables, however, are deliberately not included in the generated class for reasons that will be explained in a later subsection. Internally the implementation of the class is totally different. Firstly, only variables pertinent to the establishment and maintenance of the RPC connection are present. Secondly, all of the operations are re-implemented to perform the appropriate parameter (un)marshalling and RPC invocation. Thirdly, some additional operations are introduced including an additional protected constructor which is used to ensure that certain information only pertinent to the RPC system is correctly propagated to the stub generated versions of all base classes (if any).

4.4.4. Client Side Code

The generated client stub code for each operation follows a standard pattern: marshall arguments, send invocation, await reply, unmarshall results, and return to caller. This pattern is illustrated below.

```
AnAppointment RemoteDiary::GetNextAppointment (time_t now)
{
    /* call and return buffers */
    RpcBuffer rvBuffer;
    RpcBuffer callBuffer(_myHashVal);
    RpcBuffer replyBuffer;
    RPC_Status rpcStatus = OPER_UNKNOWN;
    Int32 serverStatus = OPER_INVOKED_OK;
    AnAppointment returnedValue;
```

```
    /* marshall parameter */
    callBuffer << now;
    /* do call */
    rpcStatus = _clientHandle.call(31096804, callBuffer, serverStatus,
                                   replyBuffer);
    if (rpcStatus == OPER_DONE)
    {
         switch (serverStatus)
         {
         case OPER_INVOKED_OK:
              replyBuffer >> rvBuffer;
              rvBuffer >> returnedValue;
              break;
         default:
              _clientHandle.rpcAbort();
         }
    }
    else
         _clientHandle.rpcAbort();
    return (returnedValue);
}
```

The client stub code produced exploits the C++ constructor and destructor notions to ensure that the real (user) objects in the server have lifetimes that match the lifetime of the (stub) objects in the client. At the point that the stub object enters scope in the client (and thus the constructor operation of the object is automatically executed) then binding of client to server is accomplished using the supplied `ServiceName` (how this is handled is RPC system specific). Furthermore, the first RPC sent over the connection corresponds to the invocation of the constructor for the real object and is passed the arguments presented to the stub by the client application. Similarly, when the stub object is destroyed in the client, the generated destructor causes an RPC request to be sent to the server causing the execution of the remote object destructor before the connection to the server is itself terminated. Precisely how server processes are created is a matter for the underlying system.

4.4.5. Server Sharing and Concurrency

From the above paragraphs it can be seen that the binding of client to server is driven by the construction and destruction of the `ClientRpcManager` class instances. These may be either created implicitly when the stub object is created or explicitly by the programmer prior to stub object construction. The actual server creation process is considered to be independent of the stub generation process, in

that the underlying RPC system can freely create a new server process or share an existing process at its discretion.

However, the generated server code must contain some limited mechanisms needed to support server sharing. In particular, since the first call transmitted to a newly initiated server will be to invoke the object constructor (to preserve C++ semantics), there is the possibility that the object will already have been constructed in a shared server by some other client. Solving this requires either invoking a constructor that does nothing (impractical since which constructor is invoked is driven by the client) or supporting a form of multiple construction and destruction. The generated code supports this latter policy in that if a constructor invocation is received by the server and no object exists then the constructor is obeyed and the incoming arguments, returned results and an indication of which constructor was invoked are saved. If another constructor call arrives then providing that it would duplicate the actions of the prior constructor (i.e., it is the same constructor with the same arguments) then the saved results are returned. Otherwise, an error indication is returned.

4.4.6. Server Side Interface

The generated server class has operations that primarily correspond to those of the original input class except that each is responsible for parameter (un)marshalling and calling the equivalent operation on the real object. In addition this server class has operations for server initialisation and two operations that implement the code that determines from the incoming call which server operation to actually call (the so-called operation dispatch code).

```
class ServerDiary : public ServerLockManager
{

public:
    ServerDiary ();
    ~ServerDiary ();

    void Server (int, char **);
    Int32 DispatchToClass (LocalDiary*, Int32, RpcBuffer&, RpcBuffer&);

private:
    // Main server dispatch operation
    Int32 DispatchToOper (LocalDiary *, Int32, RpcBuffer&, RpcBuffer&);

    // Operations corresponding to those callable in the client
    Int32 Diary119360965(LocalDiary *, RpcBuffer&, RpcBuffer&);
    Int32 Diary262355078(LocalDiary *, RpcBuffer&, RpcBuffer&);
    Int32 WhereIs186673735(LocalDiary *, RpcBuffer&, RpcBuffer&);
```

```
    Int32 GetNextAppointment31096804(LocalDiary *, RpcBuffer&, RpcBuffer&);
    Int32 AddAppointment101964452(LocalDiary *, RpcBuffer&, RpcBuffer&);
    Int32 DelAppointment222961300(LocalDiary *, RpcBuffer&, RpcBuffer&);
    Int32 save_state140478901(LocalDiary *, RpcBuffer&, RpcBuffer&);
    Int32 restore_state9807781(LocalDiary *, RpcBuffer&, RpcBuffer&);
    Int32 type117319830(LocalDiary *, RpcBuffer&, RpcBuffer&);

    // Pointer to real object
    LocalDiary *theRealObject;
    . . .
};
```

Each routine in the server class effectively has the same set of arguments. The first is a pointer to the object to be manipulated which is passed to ensure that the semantics of multiple inheritance are obeyed. The second is an `RpcBuffer` that contains all of the call information (incoming parameters, for example), and the third is an `RpcBuffer` into which the results (if any) can be placed. All operation names in this class are generated by combining the original name with a hash value computed from the original full operation signature (class name, operation name, and types of all parameters). This scheme ensures that operations overloaded in the original class can be correctly resolved in the server (otherwise the standard overloading mechanism in the compiler would not be able to tell them apart). This computed hash value is also used in the server dispatch code when determining which operation in the server to actually call.

4.4.7. Server Side Code

```
Int32 ServerDiary::GetNextAppointment31096804
        (LocalDiary *theObject, RpcBuffer& work, RpcBuffer& result)
{
    RpcBuffer rvBuffer;
    Int32 errCode = OPER_INVOKED_OK;
    /* unpack incoming argument */
    time_t now = -1;
    work >> now;
    /* perform the real call */
    AnAppointment returnedValue = theObject->GetNextAppointment(now);
    /* send back result */
    rvBuffer << returnedValue;
    if (rvBuffer.length() > 0)
        result << rvBuffer;
    return errCode;
}
```

4.5. Coping with RPC Failure

Failure of an RPC is far more problematic 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 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 (even though it knows the signature of all of the object's constructors, they may not necessarily be accessible due to C++ access rules).

The most likely additional causes of failure in a distributed system over those found in a non-distributed one will be caused by failure of the RPC system for some reason. RPC failure typically comes from two sources. Firstly the RPC itself fails for some reason (that is, the server does not respond to the client request for a variety of reasons including crashed server machine or process, network partition, or server overload). Secondly, the RPC succeeds (in the sense that the call is delivered) but the server process rejects it as invalid. This latter case can be caused by mismatches between the client and server interfaces for example. In either case a call to the `ClientRpcManager` operation `rpcAbort` is made by the generated stub code. This routine checks to see if the programmer has established a handler routine for RPC failures through this particular connection and if one exists then it is called. If no such handler exists then the global `rpcAbort` operation is called. This routine determines if the programmer has established a global handler for all RPC failures and calls it if it exists. Otherwise if a global handler has not been established then an exception is raised (until the proposed C++ exception handling mechanism is available this is simulated using UNIX signals).

4.6. Coping with Inheritance

To ensure that the stub code for each class can be compiled independently from any of its parents and so that a change in a base class need not necessarily force a regeneration and recompilation of the stub code for any derived class, the stub generator preserves the inheritance properties of the input classes in its output classes. That is, the server dispatch code (implemented in this example by the

generated routine `ServerDiary::DispatchToOper`) will only directly invoke the operations defined in the `Diary` class—not any operations from any class from which `Diary` might have been derived. If an operation inherited from some base class needs to be invoked the request is passed to the appropriate base class by the routine `ServerDiary::DispatchToClass` (in this example).

```
Int32
ServerDiary::dispatchToOper ( LocalDiary *theObject,Int32 funcCode,
                              RpcBuffer& work, RpcBuffer& result)
{
    switch (funcCode)
    {
    case 119360965:
         return Diary119360965(theObject, work, result);
    case 262355078:
         return Diary262355078(theObject, work, result);
   case 186673735:
         return WhereIs186673735(theObject, work, result);
    case 31096804:
         return GetNextAppointment31096804(theObject, work, result);
    case 101964452:
         return AddAppointment101964452(theObject, work, result);
    case 222961300:
         return DelAppointment222961300(theObject, work, result);
    case 140478901:
         return save_state140478901(theObject, work, result);
    case 9807781:
         return restore_state9807781(theObject, work, result);
    case 117319830:
         return type117319830(theObject, work, result);
    default:
         return DISPATCH_ERROR;
    }
}

Int32
ServerDiary::dispatchToClass ( LocalDiary *theObject, Int32 funcCode,
                               RpcBuffer& work, RpcBuffer& result)
{
    Int32 classCode;

    work >> classCode;

    switch (classCode)
    {
    case -1:
```

```
        return dispatchToOper
                                (theObject, funcCode, work, result);
    case 53946306:
        return ServerLockManager::dispatchToClass
                                (theObject, funcCode, work, result);
    default:
        return DISPATCH_ERROR;
    }
}
```

4.6.1. Multiple Inheritance Complications

The above code at first seems unnecessarily complex, however, this is because it must also cope with the complications introduced by multiple inheritance. Had C++ been limited to single inheritance then the `DispatchToClass` operation would not have been required and the default action of `DispatchToOper` would have been to propagate the call to the immediate parent class if one existed or return an error otherwise (note that this latter condition should never occur in practice).

The `DispatchToClass` routine is responsible for resolving the potential ambiguities on which routine to call in the server that can arise when multiple inheritance is used (the ambiguity cannot exist in the original client code otherwise the C++ compiler would have rejected it). It does this using information built when the client object is constructed and which is transmitted as part of each call. Consider the following trivial example of multiple inheritance:

```
class Base
{
public:
    int f ();
};

class Derived1 : public Base {};

class Derived2 : public Base {};

class MostDerived : public Derived1, public Derived2
{
public:
    void anOp();
}
```

In this example any instance of the class `MostDerived` will contain two instances of the `Base` class—one in `Derived1`, and another in `Derived2`. From the

application programmer's point of view this is potentially harmless and complications only arise if `f()` is called from an operation in `MostDerived` (say `anOp`) since the compiler cannot determine unaided upon which of the sub-objects `f()` should be invoked. In this scenario the programmer must explicitly qualify the call (for example as `Derived1::f()`). This explicit qualification uniquely identifies which of the `Base` sub-objects are being operated upon in the client. Naturally it is the job of the stub generator to ensure that this qualification is also reflected in the server by the generated stub code.

However, since the stub generator treats all of these classes as independent then the operation `Base::f()` will be assigned only one operation code (in this particular example 59307398) and it is this value that will be sent in the RPC message. Without an auxiliary mechanism the server dispatch code cannot decide based solely on operation code upon which sub-object the operation should be invoked. To solve this problem the stub generator associates a hash code with each class. As a client stub object is constructed a list of these codes is dynamically built and stored at each level of the hierarchy as each constructor in the hierarchy is invoked. The list for any given class thus consists of the class's own code combined with the list of codes passed as an argument to the class's constructor. This list is prepended to all outgoing calls and effectively acts as a routing map allowing the server dispatch code to navigate the inheritance hierarchy.

Thus in the preceding example, the dynamically built hashcode lists are:

```
MostDerived : -1
    Derived1:210816977:-1
         Base:210816977:297109:-1
    Derived2:210816978:-1
         Base:210816978:297109-1
```

Thus, if `Derived1::f()` is invoked, then the list `210816977:297109:-1` is sent in the RPC buffer along with the code representing the operation (`59307398`). The server's `DispatchToClass` routine extracts this list one entry at a time. On reading the first element (`210816977`) it calls `ServerDerived1::DispatchToClass`, which extracts the code `297109` and then in turn calls `ServerBase::DispatchToClass`. This extracts the code `-1` and thus invokes the `DispatchToOper` routine which finally calls the operation requested on the correct sub-object.

4.7. Template Classes

At first sight simple template classes seem to cause surprisingly few complications. On reading a template class the stub generator proceeds exactly as it would

for any other class, only instead of producing normal classes to replace the original, it produces template classes and the appropriate templated operations. For example, the trivial class:

```
template <class T> class Test
{
public:
    Test ();
    T *example (const T*);
};
```

produces the following client side class:

```
template <class T> class RemoteTest
{
public:
     RemoteTest (ClientRpcManager * = 0);
     T *example (const T*);
private:
     . . .
};
```

and this example implementation for the operation `example()`.

```
template <class T>
T *RemoteTest<T>::example (const T* _par_0_)
{
    RpcBuffer rvBuffer;
    RpcBuffer callBuffer(_myHashVal);
    RpcBuffer replyBuffer;
    RPC_Status rpcStatus = OPER_UNKNOWN;
    Int32 serverStatus = OPER_INVOKED_OK;
    T *returnedValue = 0;

    callBuffer << now;

    rpcStatus = _clientHandle.call(181144212, callBuffer, serverStatus,
                                   replybuffer);
    if (rpcStatus == OPER_DONE)
    {
         switch (serverStatus)
         {
              case OPER_INVOKED_OK:
                   replyBuffer >> rvBuffer;
                   rvBuffer >> returnedValue;
                   break;
```

```
            default:
                _clientHandle.rpcAbort();
        }
    }
    else
        _clientHandle.rpcAbort();
    return (returnedValue);
}
```

There is, however, one complication and that lies in the area of server class instantiation. For non-template classes creation of an instance of a client class `clientX` will always cause the creation of a matching class `serverX` in the generated server code. However, for a template class, the actual instantiated type of the class in the client (and thus the server) is not determined until the client code is compiled. Since the stub generator only reads header files and not client code it cannot determine what server type to instantiate. In this situation the programmer must write the code to instantiate the server type explicitly.

4.8. Exception Handling

Although few compilers currently support the proposed exception handling system, its adoption by the standards committee seems assured and the syntax and semantics appear to have been frozen. Thus it is possible for the stub generator to provide *some* support for true C++ exceptions despite the lack of compilers for the generated code. As will be shown this support cannot be complete due to the lack of information available to the stub generator.

Currently, the generation of exception handling code is conditional. If any member function signature contains an exception specification then exception handling code is generated for *all* member functions of the class. If no member function has an exception specification then the code generated is as described earlier.

Exceptions generated by a member function are treated exactly like normal returnable parameters and have the same restrictions; that is, they must be marshallable so that they can be transmitted back to the client as part of the return RPC. When received in the client, any thrown exception is unmarshalled and then (re)thrown so that the caller observes the correct behaviour. Thus extending the earlier `Diary` example with exceptions gives the following client code:

```
AnAppointment RemoteDiary::GetNextAppointment
                            (time_t now) throw (anException)
{
    RpcBuffer rvBuffer;
```

```
    RpcBuffer callBuffer(_myHashVal);
    RpcBuffer replyBuffer;
    RPC_Status rpcStatus = OPER_UNKNOWN;
    Int32 serverStatus = OPER_INVOKED_OK;
    AnAppointment returnedValue;
    callBuffer << now;
    rpcStatus = _clientHandle.call(31096804, callBuffer,
             serverStatus,replyBuffer);
    if (rpcStatus == OPER_DONE)
    {
         switch (serverStatus)
         {
              case OPER_INVOKED_OK:
                   replyBuffer >> rvBuffer;
                   rvBuffer >> returnedValue;
                   break;
              case EXCEPTION_RAISED:
              {
                   int exceptionNumber;
                   replyBuffer >> rvBuffer;
                   rvBuffer >> exceptionNumber;
                   switch (exceptionNumber)
                   {
                        case 0:
                        {
                             anException eType;
                             rvBuffer >> eType;
                             throw eType;
                        }
                   }
                   break;
              }
              default:
                   _clientHandle.rpcAbort();
         }
    }
    else
         _clientHandle.rpcAbort();
    return (returnedValue);
}
```

From this it can be seen that exceptions are returned as a pair indicating which exception was thrown, followed by the encoded exception itself. The `rvBuffer` contains either the correct returned value or the exception, with the returned `serverStatus` indicating which is valid.

Similar code is produced on the server side:

```
Int32
ServerDiary::GetNextAppointment31096804
             ( LocalDiary *theobject, RpcBuffer& work, RpcBuffer& result)
{
    RpcBuffer rvBuffer;
    Int32 errCode = OPER_INVOKED_OK;
    time_t now = 0;
    work >> now;

    try
    {
         AnAppointment returnedValue =
              theobject->GetNextAppointment(now);
         rvBuffer << returnedValue;
    }
    catch (anException _Ex_0_)
    {
         rvBuffer << 0;
         rvBuffer << _Ex_0_;
         errCode = EXCEPTION_RAISED;
    }
    result << rvBuffer;
    return errCode;
}
```

Here, a handler is established for each exception that the function has declared it may throw, and the returned buffer is constructed appropriately.

Problems exist with functions that do not have any exception specification, since the language specifies this to mean that the function can throw *any* exception. From the stub generation point of view this is akin to having an exception specification of the (currently illegal) form `throw (...)`. Since exceptions have to obey the same rules as parameters this cannot be handled in the same manner as declared exceptions. Instead the server code establishes a generic handler (`catch (...)`) which simply sends the status of `UNEXPECTED_EXCEPTION` back to the client. When the client code detects this return status it passes it to the caller via the `rpcAbort` mechanism outlined earlier. An alternative approach might be to throw a stub specific exception, however, it is impossible to (re)throw the exception that was actually thrown at the server due to the complete lack of information regarding the exception that was actually thrown. The stub generator produces warning messages whenever it produces this code to alert the programmer to the possible problems.

The semantics that may be associated with exceptions is equally hard to mimic since it depends entirely upon the actual code of the member function to

which the stub generator has no access. For example consider the following example signature:

```
long Aclass::doSomething (aParameter& p1) throw (anException);
```

The generated server code can easily establish a handler for the exception `anException`, however, if that exception is thrown by the called function the signature alone is insufficient to determine whether the parameter `p1` is valid or not. By contrast, the return type is assumed to be invalid since the function did not return normally. In this situation the generated code makes the following assumptions:

1. The return type is invalid and is not returned.
2. All other parameters that would have been returned if the exception had not been thrown are valid and are returned as normal.
3. The exception itself is encoded and returned.

4.9. The Stub Generation Process

The stub generator normally generates a *set* of files for *each* class it detects in the input file. However, this behaviour can be modified by inclusion of stub generation specific directives in the input file.

4.9.1. Output Files

For a given input file `Input.h` that contains a class called `TestClass`, the stub generator will, by default, produce the following files:

`Input_stub.h`. Replacement header file—should be included in source code in place of `Input.h`. The contents of this file are similar to the original input file except that class definitions will have been removed into separate header files that will be automatically included. Certain other constructs may also be removed—in particular, inline function definitions. Inline definitions and/or extern declarations for marshalling operations may also have been inserted.

`TestClass_stubclass.h`. This file contains the definitions for the new classes that replace the original class `TestClass`. These classes are named `LocalTestClass`, `RemoteTestClass`, and `ServerTestClass`. C++ preprocessor directives attempt to hide this name change from the user.

`TestClass_client.cc`. Client side code that provides new implementations of all public operations of the original class as RPC calls to the server.

`TestClass_server.cc`. Server side code that decodes incoming RPC requests, calls the original class operations, and returns the results to the caller.

`TestClass_servermain.cc`. Simple main program that creates an instance of the server class and causes it to wait for incoming requests.

`TestClass_marshall.cc`. Automatic marshalling/unmarshalling code for instances of the class `TestClass`. Marshalling code for pointers and references may also be present in this file.

4.9.2. Directives

To ensure that the header file used as input is still acceptable to standard C++ compilers, stub generation directives are hidden in one of two ways—either within comments or as pragmas. When used in comments they are treated as declaration specifiers by the grammar (that is, like `static` or `const`, etc.) and thus should immediately precede the declaration to which they apply. The pragma form of directive only needs to appear earlier in the file. More than one directive may be given at a time, though some conflict with each other. The current set is:

`@Remote`. Indicates that the following class will be accessed remotely so the stub generator should attempt to produce client and server code and definitions to accomplish this.

`@NoRemote`. The negation to the above. In this case no RPC code will be generated for this class. However its public interface may still be augmented with the addition of marshalling code depending upon the setting of the following options.

`@AutoMarshall`. Attempt to generate marshalling code for this class automatically. This enables instances of the class to be passed as arguments in RPC calls.

`@UserMarshall`. Assume that the class already contains appropriate definitions for the operations marshall and unmarshall but still generate definitions of the marshalling operators.

`@NoMarshall`. No marshalling is allowed on instances of this class.

`@Delete`. Applicable only within a class declaration and only to member functions. This option is provided to allow more explicit control over the

de-allocation of memory in the server. Consider some operation that returns a pointer as a result. This pointer may have several possible semantics associated with it with regard to memory allocation. Firstly, it may simply be a copy of an internal pointer that remains valid only while the object upon which the operation was performed exists. Secondly, it may be a copy of an internal pointer but by returning it the object has passed responsibility for freeing the object it points at to the caller. Finally, it may point to a freshly allocated object allowing independent deletion by both parties. Both caller and callee must know which policy is in effect if the application is not either to fail (the wrong party deletes what the pointer points at) or to use memory inefficiently. However, even if both parties follow the same semantics, the separation of client and server into disjoint address spaces can cause the server to use memory inefficiently since the automatically generated code is unaware of which semantics are in effect. It thus errs on the side of caution and will not delete memory unless this option is in effect.

The following directives are only applicable in the declaration of a parameter list for a member function. They modify the default parameter passing behaviour appropriately.

`@In`. Mark parameter as input only. This is the default for all parameters except pointers, references and arrays unless they are marked `const`.

`@Out`. Mark parameter as output only. Never a default, but useful in those cases where the parameter is set by the called operation but invalid before the call (so an attempt to transmit it in the call could fail).

`@InOut`. The default for pointers, references and arrays. Causes such parameters to be passed by value result (copy-in, copy-out) as an approximation to call be reference.

The default options for a class are `@Remote`, `@NoMarshall`. Note that due to limitations in the current implementation, classes that can be accessed remotely cannot be marshalled and passed as parameters.

4.9.3. Command Line Options

The stub generator supports a small set of command line arguments. All command line arguments that start with a minus sign (for example `-I`, `-D`, etc.) are passed directly to the pre-processor unaltered, while those that start with a plus sign (+) affect the execution of the stub generation process in some way. The current set of options includes:

`+S` Indicates that the pre-processed output should be saved in a file suffixed by `.i`. This file can later be used as input to the stub generator directly.

`+w` Enable the printing of warning messages. By default only error messages are printed.

`+idl` Generate OMG style IDL instead of C++ stub code.

5. *Processing Caveats*

Initially, the input file must be pre-processed before parsing commences to remove all C++ pre-processor directives. By default this pre-processing is done by invoking the compiler used to compile the stub generator itself (as specified at configuration time) and passing it the `-E` (pre-process only flag). However, some compiler driver programs produce no output when given header files (`.h` files) as input (Cfront-based systems typically suffer from this). In an attempt to overcome this the stub generator *links* a temporary file that has a `.C` suffix to the original header and pre-processes this file instead. The link is broken when the stub generator terminates. If compilation with the default compiler fails the stub generator attempts to compensate further by running an alternative pre-processor (such as `/lib/cpp`) on the input file in this case. While this should produce compilable output it may not necessarily be correct if the driver program normally invoked the C pre-processor with extra arguments (typically `-D` and `-U` flags).

This pre-processing can also cause problems if, for example, array sizes were defined using pre-processor `#define` directives rather than C++ language facilities since the generated output file will no longer contain such directives.

6. *Marshalling Complex Structures*

So that complicated data structures that contain pointers (such as lists and trees) can be (un)marshalled automatically the routines that do the real work of encoding the data attempt to keep track of whether a pointer has been packed into the buffer already, in which case it is not packed again. Instead a special flag is inserted that the unpacking routines can recognise and can thus compensate appropriately. This helps to ensure that arguments get encoded and decoded only once. Note that since by default the generated marshalling routines encode entire objects this is equivalent to a so-called *deep copy* of any object that contains pointers to

other objects. Furthermore, the detection of whether a pointer has been packed already is not based solely upon the value of the pointer but also upon a calculated checksum of what the pointer actually points at. This ensures that if the contents change for whatever reason during the encoding process a new copy will be made.

The same pointer tracking technique also helps to ensure that pointer aliasing is correctly preserved. For example, consider the following simple (but contrived) code fragment:

```
char *s1, *s2;

s1 = "hello";
s2 = s1;
anObj->oper(s1, s2);
```

Here, in the call to `anObj->oper` the two pointers actually point at the same area of memory in the client and it is important that when the operation is invoked at the server this situation is preserved. The parameter encoding and decoding routines ensure this.

To ensure that C++ reference semantics are obeyed the stub generator implicitly converts references into pointers at the client and marshalls the resulting pointer. Server code ensures that the user is still passed a reference as it expects. Note that this implicit conversion only occurs if the reference is to a fundamental type (int, char, class instance, etc.). Other references (for example, reference to pointer) are passed unmodified.

Finally, the programmer can suppress the automatic generation of marshalling code and provide alternative implementations if required through a command (`@UserMarshall`) embedded in the header file describing the class.

7. *Problems with Stub Generation*

Stub generation is not without its problems caused primarily by the lack of a shared address space between the client and the server. For example, the semantics of procedure call may be different (stub generation usually utilises a copy-in, copy-out process for arguments which may have a different effect upon application execution). Furthermore, certain types of parameters may be disallowed altogether (procedure type parameters, for example). Such problems are not particular to the stub generation system described here but are inherent in the stub generation process and affect all conventional languages distributed this way. Additionally, since C++ was not designed for distributed programming some of its constructs are not amenable to stub generation techniques and have to be disallowed. Examples of such constructs include:

Variable length argument lists. These cannot be marshalled automatically since the stub generator cannot determine at the time it processes the header file how many arguments will need to be marshalled on any given call.

Public variables and friends. These break the assumed encapsulation model and allow potentially unconstrained access to the internal state of an object. Since that object may now be remote from the client application such variables will typically not exist or at least not be accessible in the same address space.

Static class members. C++ semantics state that only a single copy of a static class variable exists regardless of the number of instances of the class in existence. These semantics cannot be enforced in a distributed environment since there is no obvious location to site the single instance, nor any way to provide access to it.

No pointers to functions. Allowing pointers to functions requires the ability for the server to call back to the client to execute the required code. The current RPC implementation does not provide the required functionality.

Limited operator overloading. This is caused by the fact that such operations typically take instances of the same class as arguments and would thus require object mobility (the individual instances could be on different remote nodes).

Parameter Semantics. Pointer and reference arguments are currently implemented as value-result. This approach has several potential problems that arise due to a variety of causes. 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. This is especially true if the type is `char *`—that is, a pointer to character that by convention is used to represent a string, or a list. Therefore, the stub generator imposes certain semantics upon pointers such that a pointer to a type is always considered to point to only one instance of that type—except for the character pointers that retain their traditional meaning.

Failures. As described in an earlier section, failure of an RPC is far more problematic to handle than the 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 provided mechanism based upon the use of an `rpcAbort` operation is sufficient but inelegant. Once the majority of compilers implement exception handling an alternate approach may be feasible.

All of these problems have the effect of lowering the overall access transparency to the programmer; however, and this is the important gain, not completely to zero. With care applications can be written that are fully location and access transparent, while others require only minimal additional programmer assistance. However, the point remains that stub generation relieves the programmer of a significant proportion of the burden involved in the distribution of applications.

8. Current Status and Experience

The system as described in this paper is functional and forms one of the key components of the Arjuna distributed programming system. As such it has been used to support the implementation of a University-wide student registration system which was used by the administration of the University to register all students in the University in October 1994. Additionally it has been used as part of the implementation of a distributed database system (Stabilis [Buzato & Calsavara 1992]) and also in research on providing fault-tolerant parallel programming over a network of workstations. Finally, the system is in use by students in a variety of projects and as part of a course on distributed systems. In general it has proved a useful tool, although as might be expected it tends to be more useful when the original code that it processes has been written with stub generation in mind. Processing old code can be a little problematic depending upon how old the code is and how many of the unsupported features are used.

Due to the evolving nature of C++ the stub generation system itself continues to evolve. However, it still makes relatively few demands on the underlying C++ compiler, particularly in its use of the newer features of the language. In particular, templates could be used in several places in the generated code but since some compilers still have problems with templates they are not used.

Additionally, since the heart of the system is a C++ parser, it has been used to implement a prototype C++ class to OMG IDL definition translator. The aim here being to aid in the process of converting existing code into code that will work in an OMG CORBA compliant system.

9. Related Work

The task of using C++ as a transparent distributed programming language has exercised the minds of many researchers over the years since the language

was first introduced. Their approaches can be classified into two broad approaches.

The first usually extends the language in some way by adding extra keywords and then either provides a pre-processor for the extended language or modifies an available compiler to implement the required functionality. The latter option is a thankless task that ties users into the particular compiler that was modified (undesirable since that compiler may not be available on a particular hardware platform). Early implementations of SOS [Shapiro et al. 1989] used this approach. Alternatively using a pre-processor means that standard compilers can be used; however, depending upon how the additional keywords are presented to the programmer transparency may be compromised (the modified source might be unacceptable as C++ unless pre-processed (for example, C** [Cahill et al. 1990]). While the stub generation approach adopted here is a pre-processor it is basically non-intrusive in that the augmented source remains valid C++ throughout. Furthermore, if the default actions are acceptable the source remains unaltered.

The second approach effectively takes the C++ memory model of a shared address space and spreads it across the network (the so-called Distributed Shared Memory (DSM) approach). This has the advantage that much more of the language becomes distributable, however, the cost is that most DSM approaches assume a homogeneous environment to avoid numerous complications in pointer and data layout, etc. Furthermore, the adoption of a global virtual address space has potential scalability problems due to the scarcity of virtual addresses. Performance can also be a problem in such systems. Systems that basically follow this approach include PANDA [Assenmacher et al. 1993], Amber [Chase et al. 1989], and Cool [Habert et al. 1990].

Some systems are combinations of both of these basic approaches (for example, PANDA augments C++ with keywords such as `persistent` and thus requires a pre-processor but uses DSM as an underlying mechanism).

10. Conclusions

The transparent distribution of an arbitrary C++ program is impossible to achieve in practice due to the inherent assumptions about the underlying system built into the language itself. However, the stub generation technique described here goes a long way in automating the process and relieves the programmer of much of the burden involved. For example, the Stabilis [Buzato & Calsavara 1992] distributed database system is built on top of Arjuna using the stub generator described here.

Many of the problems stem from the fact that the stub generator only reads header files which regrettably often do not contain enough semantic information

to guide the generation process—hence the need for programmer assistance in the form of stub generator commands in the header files. Nonetheless it still achieves a high degree of transparency with minimal programmer intervention. As such it is a useful tool in any programmer's toolkit.

Acknowledgments

The work reported here has been supported in part by grants from the UK Ministry of Defence, Engineering and Science Research Council (Grant Number GR/H81078) and ESPRIT project BROADCAST (Basic Research Project Number 6360).

References

1. APM, *ANSA Reference Manual*, Cambridge, UK, 1991. (Available from APM Ltd., Poseidon House, Cambridge, UK.)
2. H. Assenmacher, T. Breitbach, P. Buhler, V. Hubsch, and R. Schwarz, PANDA—Supporting Distributed Programming in C++, *Proceedings of ECOOP93*, LNCS 707, pages 361–383, Kaiserslautern, Germany, Springer, July 1993.
3. B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo, and M. Schwartz, A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems, *IEEE Transactions on Software Engineering*, vol. SE-13, no. 8, pages 880–894, August 1987.
4. A. D. Birrell and B. J. Nelson, Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems*, vol. 2, no. 1, pages 39–59, January 1984.
5. 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, pages 65–76, January 1987.
6. L. E. Buzato and A. Calsavara, Stabilis: A Case Study in Writing Fault-Tolerant Distributed Applications Using Persistent Objects, *Proceedings of the Fifth International Workshop on Persistent Objects*, San Miniato, Italy, September 1–4, 1992.
7. V. Cahill, C. Horn. A. Kramer, M. Martin, and G. Starovic, C** and Eiffel**: Languages for Distribution and Persistence, Technical Report, Distributed Systems Group, Trinity College Dublin, 1990.
8. J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield, The Amber System: Parallel Programming on a Network of Multiprocessors, *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, 1989.
9. 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.
10. D. Detlefs, M. P. Herlihy, and J. M. Wing, Inheritance of Synchronization and Recovery Properties in Avalon/C++, *IEEE Computer*, vol. 21, no. 12, pages 57–69, December 1988.
11. P. B. Gibbons, A Stub Generator for Multilanguage RPC in Heterogeneous Environments, *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pages 77–87, January 1987.
12. S. Habert, L. Mossieri, and V. Abrossimov, Cool: Kernel Support for Object-Oriented Environments, *Proceedings of the Joint ECOOP/OOPSLA*, pages 269–277, Ottawa, Canada, October 1990
13. 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*, pages 225–235, January 1985.

14. B. Liskov, Distributed Programming in Argus, *Communications of the ACM*, vol. 31, no. 3, pages 300–312, March 1988.
15. OMG, The Common Object Request Broker: Architecture and Specification, Object Management Group, Cambridge, Mass., December 1991.
16. OSF, OSF DCE V1.x Requirements, OSF DCE Reliable Computing Group, 1991.
17. 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, pages 30–37, January 1988.
18. G. D. Parrington, Programming Distributed Applications Transparently in C++: Myth or Reality?, *Proceedings of the OpenForum 92 Technical Conference*, pages 205–218, Utrecht, November 1992.
19. M. Shapiro, Y. Gourhant, S. Habert, L. Mossieri, M. Ruffin, and C. Valot, SOS: An Object-Oriented Operating System—Assessment and Perspectives, *Computing Systems*, vol. 2, No. 4, December 1989.
20. S. K. Shrivastava, G. N. Dixon, and G. D. Parrington, An Overview of Arjuna: A Programming System for Reliable Distributed Computing, *IEEE Software*, vol. 8, no. 1, pages 63–73, January 1991.
21. A. Z. Spector, R. Pausch, and G. Bruell, Camelot: A Flexible, Distributed Transaction Processing System, *Proceedings of CompCon 88*, pages 432–439, February 1988.
22. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.
23. Sun, Rpcgen Programming Guide, Network Programming Guide, Sun Microsystems Inc., 1988.
24. Sun, Network Services, Network Programming Guide, Sun Microsystems Inc., 1988.