Writing a Client-Server Application in C++

Paulo Guedes*

OSF Research Institute
1 Cambridge Center
Cambridge, MA 02142
pjg@osf.org

Daniel Julin

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
dpj@cs.cmu.edu

Abstract

Applications based on the client-server model place a special emphasis on the specification of interfaces, the separation of interface and implementation and on the support for multiple implementations of the same interface. The class hierarchy of such an application has to be designed while taking these issues into account.

In this paper we present a model for writing client-server applications in C++, based in our experience with the Mach 3 multi-server system. Interfaces are defined by C++ abstract classes, from which implementations are derived. Implementations generally use multiple-inheritance to inherit functionality from other implementation classes.

We describe how this simple model was applied to the construction of the clients and servers that compose the Mach 3 multi-server, using standard C++. We discuss how multiple-inheritance simplified the design of the system and the need for run-time, type-safe pointer conversion. Finally, we give an overview of our class library and relevant aspects of the remote object invocation subsystem and summarize our experience with C++ in this project.

1 Introduction

A major aspect of a system composed of cooperating clients and servers is how the interactions between them are specified and implemented.

Traditionally, the interface to a server is composed of a set of routines and is specified with an Interface Definition Language (IDL). A compiler, called the stub generator, generates the stubs that are linked to the clients, isolating their code from the details of the communication mechanisms.

In an object-oriented environment the natural evolution is to encapsulate the interface to the server in one or more classes, define their interface in a language-independent way with an IDL and generate stub classes to link with clients and servers. Several projects have implemented stub generators for C++[4, 11].

However, there are a number of limitations with this simple approach. One of the major concerns of a designer of a distributed application is the optimization of the interactions between clients and servers, as they amount to a very significant percentage of the costs involved. A common technique to deal with this problem is to use proxy objects[13]. Such objects are stubs with special hooks to allow the programmer to optimize or even avoid the communication between the client and the server while keeping the structure of the application clean.

*Author's current address: INESC/IST, R. Alves Redol 9, 1000 Lisboa, Portugal (pjg@sabrina.inesc.pt)

Multiple implementations of the same interface are the rule and not the exception in client-server applications. There are at least always two implementations for each interface, the client-side and the server-side implementation, but often systems need a number of different implementations to handle various communication mechanisms.

Another concern is the extensibility of the system to allow a newer version of a server to be able to work with existing clients, and conversely to allow clients to work with old versions of the servers. Subtyping of object-oriented languages is a major tool to handle this problem, but it has to be used carefully and effectively. Moreover, the precise subtyping rules of the implementation language do not usually match those of the IDL.

In this paper we describe how we approach these issues in the Mach 3 multi-server system[5]. This system is composed of a number of servers running on top of the Mach 3 kernel, together with emulation libraries executing in the address space of the user programs. Together, they provide the complete services of an operating system. Both the servers and the emulation libraries are programmed as a set of C++ objects that communicate by invocation of C++ functions, independent of their location. Clients and servers are constructed from a library containing a number of common C++ classes. Both the interfaces and their implementation are specified in C++. Several interfaces have multiple client and server-side implementations. Servers may be modified without requiring changes to existing clients.

2 Interface and Implementation

This section describes how we specify the interfaces between clients and servers and how we support different client-side and server-side implementations.

2.1 Interface

The interfaces between clients and servers are defined with C++ abstract classes. For the sake of clarity we shall ignore for the moment the details of the RPC subsystem and concentrate on the programmer's view of the C++ class hierarchy. When writing a server, the programmer must first design the classes that are externally visible by clients and then define the C++ abstract classes that constitute the interface. These abstract classes are the only server classes seen by the client's code. For example, if the server defines classes `naming` and `file` as:

```
class file {
      public:
        virtual int read(char*, int) =0;
        virtual int write(char*, int) =0;
};

class naming {
      public:
        virtual int open(char*, file**) =0;
};
```

a client would use them as follows:

```
        file* f;
        char  buf[1024];
        int error = fileserver->open("myfile", &f);
        int bytesread = f->read(buf, 1024);
```

(assuming for the moment that variable fileserver is of type naming and has been previously initialized). The important thing to note is that this code is completely independent of the implementation of classes naming or file, of the mechanism used to communicate with the server and even of the location of the naming and file instances. The designer of the system has complete freedom to choose the location of the various objects and to use the communication mechanisms most appropriate to any given configuration. In particular, if the object being referenced happens to be co-located with the client, it is accessed directly as in a normal C++ program. The system can be conceived in terms of classes, or services, instead of being decomposed in address spaces and servers. That decomposition may be postponed to a latter phase, when servers are assembled out of the library of classes in the configuration that better fits the hardware or any other external constraints (e.g. security, performance).

2.2 Implementation

A client-side implementation class or proxy is always a subclass of the interface class that it implements. In the file example, a simple implementation that sends a RPC to the server for each request would be as follows:

```
class rpc {
        mach_port_t server_port;
      public:
        virtual do_rpc(int method_id, ...) { /* some implementation */ }
};

class client_rpc_file: public virtual file, public virtual rpc {
      public:
        virtual int read(char* buf, int len) {
                return do_rpc(method_id(read), buf, len);
        }
        virtual int write(char* buf, int len) {
                return do_rpc(method_id(write), buf, len);
        }
};
```

where class rpc is a generic class that implements remote procedure calls and uses server_port as the communication handle to talk to the server.

When the open function is called, the RPC subsystem creates an instance of an implementation class, client_rpc_file in this case, and returns it to the client. This is possible because client_rpc_file is a subclass of client and therefore can be used wherever a client is expected. Class client_rpc_file also uses inheritance to reuse the implementation of class rpc. Inheritance is used here with two of the three meanings described by[15], interface inheritance and implementation inheritance.

If there are multiple implementations of the same interface, the RPC subsystem has to decide which one to instantiate. In our system this is determined by the server, by sending the name of the client-side implementation that should be created.

Server-side implementations follow the same pattern: each implementation class is a subtype of the interface that it implements and also inherits code from the implementation class of the level above.

2.3 Why Multiple-Inheritance

The advantage of using multiple-inheritance becomes apparent when we consider *extending* the system by deriving new interfaces from the existing ones.

Consider, for example, that we define a new class `seek_file` that extends the interface of `file` with a new operation `seek`:

```
class seek_file: public virtual file {
      public:
        virtual unsigned long seek(unsigned long, int) =0;
};
```

The client-side implementation of `seek_file` using class `rpc` would look like the following:

```
class client_rpc_seek_file: public virtual seek_file,
                            public virtual client_rpc_file {
      public:
        virtual unsigned long seek(unsigned long position, int direction) {
                return do_rpc(method_id(seek), position, direction);
        }
};
```

The implementation class has to be a subtype of the interface class for the whole scheme to work, so deriving `client_rpc_seek_file` from `seek_file` offers no controversy. The debatable choice is whether it should also inherit from `client_rpc_file` or instead use it as a member. Let us look at this second alternative:

```
class client_rpc_file_SI: public file {
      public:
        rpc parent_impl;
        virtual int read(char* buf, int len) {
                return parent_impl.do_rpc(method_id(read), buf, len);
        }
        virtual int write(char* buf, int len) {
                return parent_impl.do_rpc(method_id(write), buf, len);
        }
};

class client_rpc_seek_file_SI: public seek_file {
      public:
        client_rpc_file_SI parent_impl;
        virtual unsigned long seek(unsigned long position, int direction) {
                return parent_impl.parent_impl.do_rpc(method_id(seek),
                                                      position, direction);
        }
        virtual int read(char* buf, int len) {
                return parent_impl.read(buf, len);
        }
        virtual int write(char* buf, int len) {
                return parent_impl.write(buf, len);
        }
};
```

This second alternative basically forces us to repeat in each subclass all the methods of the base classes with trivial implementations as shown above. This may be acceptable for a small system with a reduced number of classes and methods, but is clearly undesirable in a large system with many classes and several levels of subtyping. Each time a new interface class is derived, its implementation class has to copy all the methods from the whole class hierarchy, each of them calling the implementation of the level above. Even worse, each time an interface class changes, because a parameter is added or deleted in a function, that change must be propagated by hand to all the implementation classes that derive from that interface. We feel that in practice this is equivalent to implementing multiple inheritance by hand, instead of letting the compiler do it for us. It is interesting to note that this example uses only single-inheritance in the interface classes and still leads to the use of multiple-inheritance.

2.4 Consistency Between Clients and Servers

The consistency between clients and servers is guaranteed by the lattice of interface classes. Changing an interface, by modifying an interface class, affects the implementations on both sides. During development this is easily detected with tools such as `make` that re-compile all the necessary files.

Traditionally systems have maintained consistency between clients and servers at the RPC level, requiring both sides to be modified if the messages exchanged between them change. We maintain this consistency at a higher level of abstraction that we call the service layer. Clients may remain unchanged as long as the interface classes are not modified, independent of the format of the messages exchanged with the server. If the message format changes, clients have to be re-linked (possibly dynamically) with new proxies, but their compiled code remains unchanged.

In some situations, we found it useful to define private interfaces between certain client and server side implementations. One example is the implementation of mapped files using the Mach external memory management facility. This implementation is fairly complex and requires close cooperation between the client proxy and the server object. By defining a private interface class visible only by these two implementations we are able to keep them consistent while still shielding the client's code from them.

A private interface class is just like an ordinary interface class, but it is visible only by the server and its proxies. It extends the public interface by defining new methods that are available only to the proxy. The proxy inherits the private interface and the implementation of the default proxy for the public interface.

2.5 Run-Time Pointer Conversion

In the previous example, a client that receives an object of type `seek_file` from `open` cannot use it as such unless the reference is converted to type `seek_file`. This case is of special importance in our system because most operations start by looking up an object by name and then request operations on it. The type of the object is not known until it is looked-up in the directory, but depending on it different operations may be performed. For example, the system contains files, directories, symbolic links, mount points, pipes, various types of connection endpoints or sockets, all of which can be looked-up with the same operation, yet files and connection-less sockets require different protocols to read and write data.

For this reason, we use a type-safe run-time pointer conversion mechanism, similar to the one described in[2]. All classes contain a `castdown(class_id)` function that returns the pointer to the object cast to class `class_id` if `class_id` is a base class of the object (otherwise it returns 0).

Clients use it as follows: the name lookup routine returns a generic type that is the least common denominator of all possible types and an indication of the (interface) class of the object. Based on this information the client then casts the reference to the real interface type of the object. For example, if a client had called `open` and an object of interface `seek_file` and implementation `client_rpc_seek_file` had been returned, the reference would be cast from `file` to `seek_file` and used as such. A pointer conversion failure generally indicates an error in the operation of the system.

We only use such pointer conversions in the interface hierarchy. In the example above, the real type of the object is `client_rpc_seek_file`, but we convert it from the interface type we know at compile time (`file`) to the the one we detect at run-time (`seek_file`).

We rejected the alternate method of defining all functions in the base class with an implementation that returns an error because it would force us to modify the base classes whenever a new class is added. In our case this is not even possible, as one of our requirements is to not change existing clients and servers when new services, and hence new classes, are introduced.

3 Overview of the Class Library

The class library contains the C++ classes from which servers are assembled. All operating system entities such as processes, files, sockets, are represented by objects and are implemented by these classes. The current prototype provides a self-hosting implementation of Unix BSD 4.3. Most of these classes were written from scratch, but a large amount of existing C code is reused (e.g. BSD Unix File System).

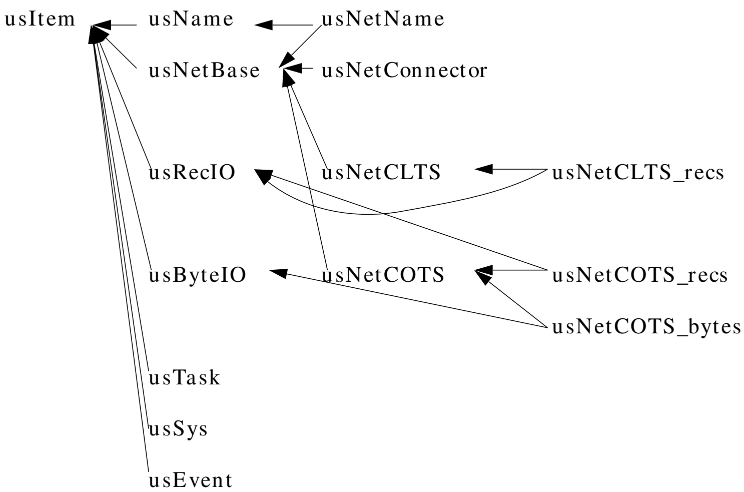

Figure 1: Interface classes.

The hierarchy of interface classes is represented in Fig. 1. All classes have a common base `usItem` that defines a protocol available to all objects. The next level defines the interfaces for most of the objects. For example, `usName` is used for directories, `usByteIO` for files, `usRecIO` for connectionless sockets, `usTask` for process control, `usSys` for configuration management and `usEvent` for signals. Class `usNetbase` and all the other interface classes are used to describe the networking interfaces, which may be connectionless and record oriented (e.g. UDP sockets), connection oriented and record oriented, or connection oriented and byte-stream (e.g. TPC/IP sockets).

Interface class `usByteIO` describes the protocol for byte-stream input/output. It has four different server-side implementations for files, pipes, ttys and sockets. There are two different proxies for this interface, a simple proxy that sends messages for each operation and is used by pipes, tty and sockets, and a complex one that implements mapped files. From the client's point of view all these implementations are absolutely equivalent.

Interface class `usName` defines a naming protocol to lookup objects by name, insert and delete them in a directory, mount one naming hierarchy on another and link two naming hierarchies. There are three server-side implementation classes that implement volatile directories, mount points and symbolic links, and several others that implement persistent directories in the Unix File System. Class `usName` is also a base class for the networking naming class `usNetName`.

Interface class `usTask` defines the protocol for process management. It currently has one server-side implementation and three proxies, the default one that sends messages for each operation and two others that cache certain characteristics of the process (e.g. pid, session identifier) to avoid contacting the server.

The complete description of all the interfaces and implementations is beyond the scope of this paper. We currently have the fifteen interface classes represented in the figure, eighteen proxy classes, about thirty server-side implementations and over twenty other classes internal to the servers. Several others are still being developed.

4 The Remote Object Invocation Mechanism

The remote object invocation subsystem consists of an RPC package extended with a layer that handles object references. An unusual characteristic of our RPC is that there are no compiled stubs. Instead, each function is described by a data structure that is parsed at run-time by a single pair of routines.

This section describes how the type information necessary to format messages is specified and the mechanics of transferring object references in messages, instantiating proxies and distributed garbage collection.

4.1 RPC Interface

The abstract classes described in the previous sections completely define the C++ interface between clients and servers. If no actual communication took place, they would provide all the information necessary for clients and servers to interact. As this is not the case, we need to specify the RPCs exchanged between clients and servers.

In our library the abstract class defining an interface also specifies the RPCs that can be exchanged between a client of the class and the server implementing it. Each member function of an abstract class may correspond to a message. The abstract class must therefore contain the information describing the parameters to each member function so that messages can be generated. Implementation classes inherit these descriptions, so by default they are able to communicate through RPC. They may redefine or ignore this information if their message interface happens to be different.

Since we have no other tools available, we require the programmer to specify the RPCs using a set of macros and functions in both the interface declaration and implementation files. They generate a data structure with type information for the class in the form of a table containing for each function the function identifier, the description of the parameters and the address of the function. This type information is parsed at run-time by the RPC package to generate messages and to dispatch incoming calls. There are no compiled stubs in the traditional sense, instead a single pair of routines scan the type description of the function and pack or unpack the parameters accordingly.

The complete version of class `naming` presented in section 2 would now be:

```
// file naming.h
class naming {
      public:
        virtual int open(char*, file**) =0;
};
EXPORT_METHOD(open);

// file naming.cc
#include <naming.h>
DEFINE_ABSTRACT_CLASS(naming);
DEFINE_METHOD_ARGS(open, "rpc: IN string; OUT * object<file>;");
```

File `naming.h` contains the abstract class definition, as seen before, and the declaration of the functions exported to the RPC package. Macro `EXPORT_METHOD` declares a structure with the method descriptor that uniquely identifies and describes function `open` in class `naming`. File `naming.cc` contains the description of the RPCs. Macro `DEFINE_ABSTRACT_CLASS` initializes the static member that contains the class' type information (e.g. class name, `typeid` used for pointer conversion). Macro `DEFINE_METHOD_ARGS` initializes the method descriptor for `open` with a string describing the type of its parameters. We still have to initialize the method descriptor with the address of the function and enter it in the per-class table. As functions may be redefined in subclasses, this cannot be done statically. Instead, implementation classes have a function `init_class` that initializes all exported functions as follows:

```
// file naming_proxy.h
#include <naming.h>
class naming_proxy: public virtual naming, public virtual rpc {
      public:
        virtual int open(char*, file**);
};

// file naming_proxy.cc
#include <naming_proxy.h>
DEFINE_CLASS(naming_proxy);
void naming_proxy::init_class(void)
{
        // initialize base classes if any
        BEGIN_SETUP_METHOD_WITH_ARGS(naming_proxy);
        SETUP_METHOD_WITH_ARGS(naming_proxy,open);
        END_SETUP_METHOD_WITH_ARGS;
}

int naming_proxy::open(char *name, file **f)
{
        return do_rpc(method_id(open), name, f);
}
```

The call to macro SETUP_METHOD_WITH_ARGS initializes the various fields of the method descriptor for open: a per-process unique value to be used as identifier, a parsed version of the string that is easier and faster to manipulate at run-time and the pointer to the function. It also enters the method descriptor in the per-class function table, overriding any that might already be there. Base classes are initialized first, so that the function associated with the function identifier is the one of the most derived class, which is the desired behavior for virtual functions. For this reason we only support remote invocation of virtual functions.

The function do_rpc, that is used by the client-side classes to invoke a remote operation, is defined in a base class and handles all outgoing messages. The first parameter method_id(open) is really the address of the method descriptor for open. do_rpc parses the method descriptor, extracts the parameters from the stack, constructs the message, performs the RPC, extracts the output parameters from the message and returns them on the stack. For the server-side the RPC does a similar set of operations. No server-side stub is necessary since the description in the method descriptor is enough to unpack the incoming messages and pack the replies.

4.2 RPC Implementation

The remote invocation mechanism depends heavily on the underlying Mach 3.0 kernel[1]. The relevant Mach abstractions for this discussion are Mach ports, port rights and messages. Mach ports are unidirectional communication channels between processes. Port rights are capabilities allowing specific rights (e.g. send, receive) of access to a port. They have 32 bit names unique in an address space. Messages are typed collections of data passed between processes. Messages are sent to ports and can carry port rights, along with other basic types such as integers and characters.

Each object exported to clients is associated with a Mach port. The server that creates the object holds a receive right to that port, and each client that is given access to the object holds a send right to it. Initially, objects are created in the server with no associated ports. The remote invocation subsystem creates a Mach port for each object the first time that this object is returned as an output parameter in a remote invocation.

Clients send messages to the port that identifies the server object. A message contains a string specifying the name of the function being invoked, its parameters and a port indicating where the reply should be sent by the server. Simple types are passed by value. Objects are represented by a tuple composed of the Mach port associated with the object and the name of the proxy class to be instantiated at the destination if the object does not exist there.

Typically, messages sent from clients to servers pass basic types as input parameters and receive objects as output parameters. Let us use the open function is class naming with implementation naming_proxy as an example. The proxy class simply formats the open message using the information stored in the method descriptor and sends it to the port stored in its instance data (let us assume that it had been initialized to reference the directory service). On the server the object identified by the port is obtained from the table port_to_object_table, the function name is searched and its address is obtained. The stack frame is constructed as described earlier and function open at the server is called.

When this function returns, the remote invocation subsystem creates a port to represent the return parameter and uses a callback to store it in the object. It uses another callback to obtain the name of the proxy class and sends the tuple (port, proxy class) in the reply. It also stores the port in the table port_to_object_table for later lookups.

At the client, the message is scanned and the return frame is constructed. When the object reference is found, the (unique) port name is used to locate the object in table port_to_object_table.

If the object exists, its pointer is returned; if not, the name of the proxy class is used to search another table called `class_map` and obtain a pointer to an object of the proxy class. This instance is used as a "factory" to create a new instance of the proxy class. The proxy object is initialized, s-toring the Mach port in its instance data, and its pointer is returned to the client. Table `class_map` is set-up statically at initialization time and contains one instance of each proxy class, associated with the class name.

When the client later invokes an operation on the proxy, it simply formats a message and sends it to the associated Mach port, repeating the process just described. The port representing the root of the directory service is obtained by clients at initialization time using a well-known port name.

The use of multiple-inheritance together with our run-time parser complicates this implementation because pointers to objects received as parameters have to be cast to the type specified by the interface. This cast is done once again using the pointer conversion mechanism described previously. In the method descriptor, the description of the parameters contains the type to which the object should be cast. At initialization time we store there the `typeid` of that class and use it to perform the pointer conversion at run-time.

4.3 Garbage Collection

Garbage collection is greatly facilitated by the use of the Mach IPC system. Mach generates a notification message when there are no more send rights outstanding for the port, thereby allowing the server to destroy the associated object. Clients and servers garbage collect their objects locally using reference counting. When a Mach port is first associated with a server object its reference count is incremented to account for all remote proxies. When the reference count on a proxy object comes to zero, the associated Mach port is deallocated and the proxy is destroyed. When the last port associated with the object is destroyed, Mach sends a notification message to the server that then decrements the reference count on the object. If it is the last reference the object is destroyed, otherwise the object will be deallocated when there are no more local references to it.

4.4 Performance

The motivation to use a run-time parser to generate messages is to make each class as self-contained and lightweight as possible, so that clients and servers can be easily assembled from a class library without having to carry extra baggage. This is particularly attractive in a system composed of communicating objects where each class may be visible remotely.

The run-time parser is composed of a loop that for each parameter enters a switch statement depending on the type of the parameter. Each branch of the switch statement contains code similar to what a compiled stub would contain for the same type of parameter.

`void f(int*)` with MiG (simple server)	338	μs
`void f(int*)` with MiG (complex server)	490-590	μs
`void f(int*)` with C++ package	596	μs
`void f(file**)` with C++ package	1156	μs

Table 1: Remote invocation times measured on a 25MHz i386 HP Vectra with the client and the server on the same machine

Table 1 shows the total elapsed times for calling a simple C function returning one integer, for a similar C++ virtual function and for a C++ virtual function returning an object. The first row was obtained with a trivial client and server using stubs generated by the the Mach Interface Generator (MiG). It represents the minimum time it takes to pack and unpack the parameters and perform the RPC. The C++ package is more complex since each invocation includes acquiring and releasing several mutexes, checking if there are still threads available to service other requests, checking the sequence number and other fields in the incoming message to prevent races relative to garbage-collection, and finally unpacking the message and performing the invocation (row 3). When similar operations are successively added to the MiG-based server, closer values are obtained (row 2). The last row shows the time to invoke a remote virtual function that returns an object. The additional costs in this case are the callbacks at the server, the longer time it takes Mach to send a message that contains a port, and the cost of instantiating the proxy object.

These values show that the performance of our RPC package is roughly equivalent to a MiG based system that does a similar amount of work, thus confirming our assumption that the cost of parsing the messages at run-time has a negligible effect on the overall performance. They also show that our C++ remote invocation subsystem is more complex and has more run-time overhead than a simple RPC based server, but the same level of complexity and overhead must be present in more realistic servers. It should be noted that these measurements are only approximate in the sense that it is very hard to compare systems that do different things.

Our 18 proxy classes have a total of 85 methods and consume about 70 Kilobytes of text space and inititalized data. A MiG file with 87 similar definitions produces 30 Kilobytes for the user-side stubs file and 63 Kilobytes for the server-side stubs file. We were expecting more dramatic space savings by not using compiled stubs. Our proxies are larger than we expected because method `init_class` is generally very large and proxy classes contain several methods that have to be present in all classes of the library. In a 2.9K proxy, these two factors account for about 2K of the total text size. We never made any effort to optimize these values.

5 Related Work

The characteristics of our system forced us to put a much bigger emphasis on the support of different and customized client-side implementations of the same interface than most of the previous work on this field. This fact influenced heavily our main design choices. On the other hand, our system is more tightly integrated in the sense that our clients and servers are system entities that are not supposed to be coded or directly used by user programs. This allowed us to impose some restrictions that otherwise would have been undesirable, such as not considering mixing servers written in very different programming languages.

The Object Request Broker Architecture and Specification from OMG[10] provides a framework to define interfaces and specify the mechanisms by which objects can interoperate in a heterogeneous environment. The specification is language and even implementation independent and aims at being used by all kinds of applications. It defines an IDL that is a subset of C++ and supports a model for separating interface and implementation similar to[9]. We could have used their IDL and compiler to generate our abstract classes but they were not available when we started the project[3].

A number of other works addressed the issue of distributed C++ applications by proposing extensions to the language[14, 8]. In our case this was not an option. One of our requirements was the ability to compile the system with existing compilers, so that other people could reuse our work and we could not afford the effort of writing our own C++ pre-processor or compiler.

Extended C++[12] extends C++ with constructs for distributed programming. It is implemented as a pre-processor that generates standard C++ code. Our goal was not to define a distributed C++. Instead, we focused on using the language as it is to program a distributed system. The features in the language that are inadequate or irrelevant to our goals were simply not used. This allowed us to ignore all the hard issues about handling or disallowing certain constructs that do not apply to the distributed case.

Several systems have implemented RPC generators for C++ [4, 11] but they do not support several client-side implementations for the same interface.

The SOS project[13] introduced the notion of proxies and provided one of the earlier implementations of a system composed of a number of communicating C++ objects.

Choices[7] is another operating system written in C++. It provides a library of C++ classes that may be customized to construct different instances of an operating system. However, Choices is from the C++ point of view a monolithic application, i.e. it lives all in the same address space, not addressing the issues of client-server interaction.

6 Retrospective

This section evaluates the use of C++ in our project and some of the design decisions presented earlier.

6.1 Using C++

The Mach 3 multi-server was initially written in MachObjects[6], an object-oriented environment based on C macros and library routines that provides a programming model similar to Objective-C with some extensions. A large amount of BSD Unix C code is also reused with minimal changes.

The decision to use C++ instead of MachObjects was mostly motivated by the perceived need to use a well-known language instead of an arcane and virtually unknown environment like MachObjects.

Our major concerns with using C++ were the adequacy of static type checking in a dynamic system such as ours and the impact of having to use a different compiler.

The first problem was adequately addressed with our model for specifying interfaces and deriving the implementation classes and the additional mechanism for run-time pointer conversion. We think that one of the most positive aspects of using C++ was the better expressiveness of typed interfaces and the type-checking provided by the compiler.

The biggest challenge was the level of stability and maturity of the GNU C++ tools (compiler, debugger, linker) and their integration with our build environment. This is only a small part of the bigger problem of introducing a new compiler in an existing organization. We experienced many problems with poor or no support for virtual base classes and multiple-inheritance in the compiler and debugger. Some of these problems have been overcome with more recent versions of these tools; others have not.

6.2 C++ vs. an Interface Definition Language

One of the uncommon characteristics of this system is that it does not use an IDL and instead relies only on C++ to define RPCs.

There are two layers at which we could have used an IDL: the service layer where interfaces are specified in terms of classes and objects, and the RPC layer where we define the messages to be interchanged. In simple systems there is a one-to-one correspondence between these two layers, but in more complicated ones like ours that is not necessarily the case.

At the service layer our solution is basically to coalesce the IDL and C++ by defining the interfaces using the subset of C++ that "makes sense". The main drawback of this approach is that we have no way to automatically check the constructs allowed by C++ that are invalid in the distributed case (e.g. pointers, public data). That task is left to the programmer, who has to carefully design these abstract classes using only a small subset of the language. In our system these interfaces are a crucial part of the design and their definition involved discussions between several people of different groups, which minimized the number of mistakes made by misuse of the language.

Defining the service interfaces with an object-oriented IDL and from there directly generating the C++ implementation classes does not support the notion of multiple implementations for the same interface. Since there are no C++ classes that describe the interface, clients have to define variables of the C++ implementation types generated by the IDL compiler, which defeats our goal of supporting different proxies for the same interface.

We could have used an object-oriented IDL to automatically generate the C++ abstract classes, instead of writing them manually as we did (if such IDL and compiler were available to us). A hierarchy of interface classes would be defined in IDL and the IDL compiler would generate our hierarchy of C++ abstract classes. One of the drawbacks of this approach is that it forces the designer of the system to define the interfaces in one language but then use the code generated by the IDL compiler to write the implementations. It might also be hard to establish a mapping between the subtyping rules of the IDL and those of C++. It has the advantage of having the IDL compiler checking constructs that make no sense in the distributed case and being more language independent, provided that the IDL compiler can generate more than one target language.

At the RPC level we require the programmer to specify the interfaces using macros basically because it was relatively easy to implement and allowed us to experiment with the run-time parsing of messages, but this is clearly unacceptable as a general solution.

A better solution would be to have an RPC compiler that automatically generates the necessary routines and data structures. Using C++ to specify the interfaces, as we did, keeps the system very homogeneous, but it is impossible to specify all the different parameter attributes of the communication subsystem, such as in-line or out-of-line data and port manipulation supported by Mach IPC. A specialized IDL adequately addresses these problems, at the expense of introducing yet another language the programmer has to deal with.

The issue of interfacing clients and servers written in different languages can be solved at the message level by using the same RPC package on both sides and defining RPCs that are bit-for-bit compatible. At a certain point of our development a fileserver still written in MachObjects was perfectly able to communicate with C++ clients using the complex mapped files proxy, just because the RPCs between them had remained unchanged when the clients were converted to C++.

7 Conclusions

We present a model to write a client-server application in C++ that uses C++ abstract classes to specify the interfaces and multiple inheritance to construct the implementations. This model cleanly supports different client-side and server-side implementations of the same interface. The lattice of interface classes, composed of C++ abstract classes using mostly single inheritance, is visible to both clients and servers and guarantees the consistency of the implementations.

We believe that the model presented here to define the interfaces and to construct the implementations is a good way to write client-server applications in C++ and can be widely used in the C++ community.

8 Acknowledgements

Many people at CMU and at the Research Institute of OSF have participated in the design and implementation of the system. Beside the authors, they are: J. Mark Stevenson, Jonathan J. Chew, Paul Neves, Paul Roy, Robert Baron, Alessandro Forin, Jeffrey Heller, Michael Jones, Keith Loepere, Douglas Orr, Richard Rashid, Franklin Reynolds and Richard Sanzi. John LoVerso, Franklin Reynolds and the anonymous reviewers provided helpful comments on early versions of this paper.

References

[1] Richard P. Draves. A Revised IPC Interface. In *Proceedings of Mach Usenix Workshop*, Burlington, Vermont, October 4-5 1990.

[2] K. Gorlen, S. Orlow, and P. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons, 1990.

[3] P. Guedes. Use of Object-Oriented Technology in the Implementation of a Distributed Operating System. In *Addendum to the Proceedings of OOPSLA/ECOOP*, Ottawa, Canada, Oct 1990. Sigplan Notices Special Issue.

[4] J. Heliotis and P. Mansey. Remote Object Invocation and its Implementation in C++. In *Proc. C++ at Work*, C++ Report and the Wang Institute of Boston University, 1989.

[5] D. Julin, J. Chew, P. Guedes, P. Neves, P. Roy, and M. Stevenson. Generalized Emulation Services for Mach 3.0 - Overview, Experiences and Current Status. In *Proceedings of Usenix Mach Symposium*, Monterey, CA, Nov 20-22 1991.

[6] D. Julin and R. Rashid. Machobjects. Technical report, Mach Project, Carnegie Mellon University, 1989.

[7] P. Madany, D. Leyens, and V. Russo. A C++ Class Hierarchy for Building UNIX-like File Systems. In *Proc. 1988 Usenix C++ Conference*, Denver, CO (USA), Oct 1988.

[8] José Alves Marques and Paulo Guedes. Extending the Operating System to Support an Object-Oriented Environment. In *Proceedings of OOPSLA 89*, New Orleans, 2-6th October 1989.

[9] B. Martin. The Separation of Interface and Implementation in C++. In *Proc. 1991 Usenix C++ Conference*, Washington, DC (USA), Apr 1991.

[10] OMG. The common object request broker: Architecture and specification. Technical Report OMG Document Number 91.12.1, Revision 1.1, Object Management Group, December 1991.

[11] G. Parrington. Reliable Distributed Programming in C++: The Arjuna Approach. In *Proc. 1990 Usenix C++ Conference*, San Francisco, CA (USA), Apr 1990.

[12] Robert Seliger. Extended C++. In *Proc. 1990 Usenix C++ Conference*, San Francisco, CA (USA), Apr 1990.

[13] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th. International Conference on Distributed Computer Systems*, pages 198–204, Cambridge, Mass. (USA), May 1986. IEEE.

[14] M. Tieman. Wrappers: Solving the RPC Problem in GNU C++. In *Proc. 1988 Usenix C++ Conference*, Denver, CO (USA), Oct 1988.

[15] Jim Waldo. Controversy: The Case for Multiple Inheritance in C++. *Computing Systems*, 4(2), Spring 1991.