C++ Class Libraries for Interprocess Communication

Introduction

The demand for efficient, portable, and easy to program interprocess communication (IPC) mechanisms has increased as more developers attempt to build distributed applications. Common IPC tasks include:

- Connection establishment and termination, addressing and access to services in the network;
- Communication between processes that may reside on different hardware platforms (e.g., Intel or PowerPC) , operating systems, (e.g., UNIX, OS/2, or Win32), and networks (e.g., Ethernet or ATM);
- Flexible access to various IPC mechanisms (e.g., Sockets, Named Pipes, RPC, shared memory, etc.).

One way to meet these requirements is to use high-level communication toolkits such as Sun RPC [Sun:94], OSF DCE [OSF:95], OMG CORBA [OMB:95], or Microsoft's Network OLE [Brockschmidt:95]. However, these solutions are often inefficient, non-portable, complex, or expensive.

A common way to workaround the drawbacks of higher-level communication toolkits is to program applications using lower-level mechanisms such as Sockets and System V TLI. These mechanisms are widely available under a variety of operating systems. Programming at this level provides the benefit of flexibility and performance, while shielding developers from the details of network and transport layer protocols such as TCP, UDP, or IPX/SPX. Sockets are particularly appealing since they are available under most Unix derivatives and also on many PC platforms, including the operating systems in the Windows family and OS/2.

Unfortunately, there are still some portability limitations with using Sockets since different implementations are subtly incompatible. Often the header files, error codes, and initialization strategies differ across operating systems. For instance, the Socket library must be explicitly initialized on Win32 platforms, whereas it's implicitly initialized on Unix. Likewise, under Unix, Socket endpoints are represented by integral file descriptors, so that read() or write() are legal calls, which is not the case under Win32, where Socket endpoints are represented as pointer handles.

Portability aside, it’s still difficult to program conventional IPC mechanisms due to the inherent non-type safety of the low-level C APIs. As mentioned above, Socket descriptors on Unix are implemented as integral types and as void pointers on Win32. Therefore, at compilation time the compiler cannot assist the programmer in checking the correctness of the application code by identifying invalid operations on Socket endpoints. This makes it all too easy to use the wrong operation at the wrong time (e.g., trying to “accept” a connection on a Datagram Socket).

A time honored way [Schmidt:92] of solving these programming problems is to create class libraries that encapsulate low level IPC APIs with C++ wrappers. This approach offers the following advantages:

- *Portability and heterogeneity*: The inconsistencies between different IPC implementations remain hidden from the developer. Depending on the sophistication of the C++ wrappers, different mechanisms such as System V TLI and Sockets can be integrated to form a consistent interface.

- *Ease of programming*: The high level of abstraction and additional functionality provided by C++ wrappers can make developing network applications easier by hiding low-level details of the C-level IPC APIs.

- *Reuse*: Routine tasks and repetitive procedures can be handled by the library. This abstraction makes the application code shorter, more readable, and easier to maintain.

- *Type-safety and robustness*: C++ wrappers can eliminate entire categories of common errors. One example is the use of Internet Socket addresses. The C-level sock API requires the developer to initialize the allocated memory - e.g., using the memset() call - after defining a variable of type sockaddr_in. If the programmer forgets to perform this initialization, the program behavior is unpredictable. Unfortunately, these types of errors occur at runtime and are hard to diagnose without tedious debugging sessions. In contrast, a C++ class library can eliminate these common errors. For instance the initialization of the sockaddr_in structures can be hidden in the constructor of a network address class.

Although C++ wrappers solve many problems for application programmers, it is time consuming to develop these types of class libraries from scratch. Developers of network programming wrappers must have a deep understanding of many topics including networking, object-oriented design, and C++. Therefore, most organizations can’t afford to devote the resources necessary to reinvent, debug, document, and maintain these wrappers internally.

Fortunately, there are now several options available to developers. For instance, two C++ IPC class libraries (Socket++ and the Adaptive Communications Environment (ACE)) can be freely obtained via anonymous ftp or the WWW. There are also several commercially supported C++ IPC class libraries from vendors such as Rogue Wave's (Net.h++) and ObjectSpace's (Systems<Toolkit>). This article compares the functionality and performance of three of these C++ IPC class libraries (Socket++, ACE, and Net.h++). Our test environment consisted of Sun SPARCstations running Solaris 2.4, using Sun CC 4.0.1 to compile Net.h++ and ACE, and GNU gcc 2.7.0 to compile Socket++.

Net.h++

In the C++ community, RogueWave has established itself as a world leader in class libraries, all of which end with .h++. The flagship product of Rogue Wave is Tools.h++. This widely used component library consists of collection classes and reusable data structures and algorithms designed for general-purpose applications. Quite logically, Rogue Wave used this library to implement Net.h++, making the installation of Tools.h++ (version 6.1) a precondition for working with this library.

RogueWave designed Net.h++ to allow independence from several operating systems and networking protocols. The present version 1.0 supports the software platforms Unix (Solaris, SunOS, HP-UX), Windows (Winsocks by Microsoft, NetManage Chameleon, Trumpet and Wollongong), and Windows NT. AIX, SGI, OS/2 and Windows95 have been planned for the future. The library is available in source code form for an additional cost.

System Architecture

The architecture of Net.h++ is subdivided into four layers shown in Figure 1 (the modules not contained in Version 1.0 are marked in grey). The lowest layer of Net.h++ provides the basic services of Tools.h++. The next layer is the Communication Adapters Layer. It contains an encapsulation of Sockets and TCP/IP addresses that provides a thin veneer atop the existing C-APIs. Unfortunately, this short list contains all the communication mechanisms supported by the current version. For future releases, RogueWave plans to include extensions for TLI, (Named) Pipes, RPC, CORBA, and FTP.

One layer up is the Portal Layer. This layer provides a more abstract C++ programming API than the Communication Adapters Layer. The Portal Layer presents a uniform, transport independent view of the different underlying communication mechanisms to applications. In addition, it provides functionality not found in the underlying layer, such as sending or receiving a specific amount of data.

The highest level of abstraction in the Net.h++ architecture is the Communication Services Layer. At this level programmers can treat communication channels as C++ IOStreams. By using the overloaded functions operator<<() and operator>>(), programmers can send complex objects without changing their morphology (i.e., their internal structure). This illustrates the strong integration with Tools.h++, which provides a mechanism known as “Virtual Streams” that send objects in binary, ASCII or XDR encoded formats. As a possible extension to the Communication Services Layer, RogueWave proposes an implementation of the Acceptor pattern [Schmidt:95] that could simplify the development of servers for different purposes.

One of the tutorials, called “The Receptionist,” outlines how such an Acceptor could work. The basic idea is to separate (1) the communication semantics of server applications from (2) the handling of connection requests. While the former cannot be done in a general way (since this depends on the application’s communication protocol), the latter can be generic. Just like an office receptionist handles the phone traffic without knowing the topics of conversation, the Receptionist classes implement strategies for answering connection requests and dispatching them. The tutorial describes classes that can implement sequential, concurrent, and event driven receptionist strategies.

Installation and Documentation

The installation procedure for Net.h++ is well documented in the Installation Guide. I was able to install it without much difficulty, though the manual did contain a minor mistake. Instead of using the command "cd C:\rogue", which clearly does not make any sense under Unix, you have to change into the directory above (!) the Tools.h++ home directory. Then the library is written onto the disk using tar and uncompress. Unfortunately, the manual does not indicate that afterwards the tar file can be deleted again. The second installation step consists in compiling the source code. The necessary adjustments to Makefile are performed by the script config_make - with exemplary convenience. Afterwards, the compilation of the library and the accompanying example programs proceeded smoothly in the test.

The examples chosen are easily understood and provide a good survey of the essential features of Net.h++. For instance, the 'hostinfo' example provides information on an Internet machine, 'greet' sends a message to an Internet address and outputs the answer, 'wwwget' loads a URL-specified HTTP page.

The Net.h++ manual contains about 130 pages (without appendices) covering the user's guide, tutorials and the class reference. The user's guide provides a good survey of the essential concepts and allows the user to become familiar with how to program applications using Net.h++. Users who are not familiar with OOD, however, must learn the OMT style diagrams since the syntax is not fully explained. The tutorial supplements the examples with an extensive sample program that also explains some general difficulties encountered in interprocess communication.

Sample Program

Listing 1 shows the code for “netfinger,” illustrating a use-case example of Net.h++. The program implements a simplified “finger” client, which is a standard Internet service that provides information about a user. Unlike the finger program found on Unix machines, this version does not directly retrieve any information itself, but gets the whole report from a finger daemon. The same example will be used for Socket++ and ACE to show how the various approaches differ.

In line (1), the Winsock DLL is initialized, if the program is compiled under Windows. Although Unix does not require such an initialization, this statement is necessary for portability reasons. For our Solaris Unix tests this call is conditionally compiled to nothing. The object 'info' is generated on the stack as an automatic variable; the destructor of the class RWWinSockInfo performs the necessary cleanup when leaving the main() function.

Under (2) the command line arguments are checked. The program should be called using netfinger user@host.

Under (3) we see Net.h++’s usage of C++ Exception Handling for errors. The benefit of this approach is that the application’s handling of errors is kept separate from the usage of the library. This improves the readability of the code and can increase the programs ability to react to errors. For instance C++ Exceptions allow applications to react to exceptions raised in constructors - which will create problems with traditional methods of error treatment.

(4) demonstrates the use of RWInetAddr. This class contains the host IP address, a port number, the Socket type (Stream, Datagram or raw), and the transport protocol to use. As always, our C++ solution is more robust and easier to program than a C solution due to the use of constructors and default parameters . In this example, the port is identified by the service name 'finger', the host name is indicated by the user in the program request, the default protocol is TCP, and the Socket type is SOCK_STREAM.

Specifying the port number and host by strings results in a call to the constructors of RWInetPort and RWInetHost that take string parameters. These classes internally call the C-API functions getservbyname() and gethostbyname() to initialize their private data structures. (This is, however, not done in the class constructors, but when the information is actually needed.)

At (5) the connection is set up. In case of an extension of Net.h++, the variable fingerd of the RWPortal type could incorporate also a (fictitious) RWTLIPortal (another subclass of RWPortal using the communication channel System V TLI) instead of an RWSocketPortal. This change can be made without requiring any code modifications. In the line marked (6) the inquiry is addressed to the finger server. This inquiry simply consists of the name of the user, on which the information shall be obtained, and is completed with a carriage return and a line feed. The member function RWPortal::sendAtLeast() guarantees that the complete inquiry will be sent, even if this requires several calls of the lower C-API to be made.

The finger server's reply is received in line (7). For illustration purposes, an object from the C++ IOStream category is used. The characters are read individually with get() to prevent the suppression of blanks, tabs and line feeds. The code used is not too inefficient since the RWPortalIStream, like any IOStream, performs its own buffering. Ostream cout, however, is also buffered, so that we synchronize its output to screen with the flush manipulator.

An alternative to using an IOStream is shown at line (8). This time the element function RWPortal::recv() is used. The output is closed as before under (7) after the finger server has completed its answer with EOF and closed down the connection.

Overall, the performace of Net.h++ was satisfactory although the current version of the library does not cover many issues related to developing distributed applications (such as event loop integration, naming and location services, etc.). However, the basic tools provided worked smoothly and without error.

Socket++

Unlike Net.h++ the Socket++ library by author Gnanasekaran Swaminathan can be obtained free of charge from the Internet. At the time of our test, the version of Socket++ was 1.10. The basic idea behind Gnanasekaran's library is to provide access to Socket communication channels using the C++ IOStream interface. Although this approach is similar to Net.h++'s usage of IOStreams, Socket++ is more closely tied to the use of IOStream interfaces. This reduces the learning curve for programmers who are already familiar with IOStreams and Sockets. Another advantage of

Socket++ is the elegant and type-secure access to Sockets through operator<<() and operator>>(), which allow applications to send and receive objects while maintaining morphology.

System Architecture

Figure 2 illustrates the Socket++ class hierarchy. The class sockbuf has been derived from streambuf (which is not part of the illustration since it belongs to the IOStream library) and is used to buffer Socket I/O. The actual reading from and writing to the Socket takes place in this class. The sockunixbuf and sockinetbuf subclasses offer specializations of the Unix or Inet communication domains, respectively. The iosockstream class has been derived from iostream and provides SOCK STREAM semantics. The classes isockstream or osockstream cover the read-only or write-only access. For Sockets and Unix Pipes the respective subclasses are available. The descendants of the abstract class sockAddr provide the encapsulation of network addresses. In addition, there are classes that implement standard Internet protocols like Echo, FTP and SMTP. Fork implements the fork() system request.

Installation and Documentation

Not surprisingly, a free library is rarely as comprehensive as a commercial product. For Socket++ this applies to the number of compilers and operating systems the library has been ported to and tested with. Unfortunately, a port to Windows is not included. Moreover, due to the peculiarities of the available Winsock implementations (e.g., Sockets endpoints are pointer handles rather than integers), major changes would be required to port the available source code to Windows.

The easiest way to compile Socket++ is to use the GNU gcc (version 2.4.0 and higher), which is available on a variety of operating systems. Though I did not attempt to use the Sun CC 4.0.1, with a little effort, this compiler could also be used.

Socket++ is not a "library for the poor". Quite on the contrary, it offers a surprisingly rich number of features. This can already be seen at the installation, which uses the autoconf GNU mechanism and consequently, could hardly be easier. Even an automatic selftest is performed. The utility classes for the Echo, FTP and SMTP protocols have already been mentioned. Thus, for example, it is possible to implement an Echo server with less than ten lines of code. Although this is not very practical (since the Echo server is already a standard Internet service), it serves as a good example for your own implementations.

The manual is available in Postscript format and contains about 50 pages. Unfortunately it belongs to version 1.6 and is thus not quite up to date. You will need some time to get used to the combined reference section/user's guide that is interspersed with many examples. Supports of the GNU project will be pleased to hear that the manual is available in Texinfo format, too.

Sample Program

Listing 2 demonstrates the netfinger example applied to Socket++. Part (1) shows the treatment of the call arguments (the same as for Listing 1). In (2) a Socket from the Internet domain with IOStream encapsulation is provided for reading and writing (class iosockinet). Evidently, in contrast to Net.h++, error treatment is performed internally in the library and not with Exception Handling procedures. However, this is quite sufficient for our simple example. Anyone who nevertheless insists on Exception Handling should use the older version 1.8 containing this mechanism - it has been dropped afterwards because several compilers do not yet support it. Rather than omitting this feature altogether, a better alternative would have been to use different mechanisms via conditional compilation.

The connection is set up with a connect() call (4) that implicitly uses a sockinetaddr. The port to be addressed is identified by the service description 'finger'. (5) illustrates the use of operator<<() to send an inquiry to the server. Under (6) the answer is read out (as in Listing 1).

The assessment regarding runtime behavior and stability given for Net.h++ also applies to Socket++. I didn't find any serious problems.

ACE (ADAPTIVE Communication Environment)

As with Socket++, ACE can be freely obtained via anonymous FTP or the WWW. Douglas C. Schmidt from Washington University is the author of most parts of ACE. ACE was originally designed as a framework to simplify Doug's research on design patterns and performance bottlenecks in parallel communication software. Over time, the library has matured and is now used in many commercial projects at companies like Bellcore, Ericsson, Motorola, Kodak and Siemens.

Like the two other libraries, ACE is designed as a toolkit for object-oriented network programming. The primary difference between ACE and the other libraries I evaluated is its scope, which goes far beyond that of Socket++ and Net.h++. This scope becomes evident when downloading the program from the FTP server. The overall release occupies around 1.4 megabytes of source code, tests, and example applications (not including the documentation, which is available separately). After compilation, the entire release occupies over 90 megabytes, due largely to the aggressive use of C++ templates and method inlining in ACE (most of the inlining can be conditionally compiled out).

System Architecture

Figure 3 outlines the ACE system architecture. The encapsulation of Unix and Winsock Sockets is called SOCK_SAP. A common interface with the System V TLI encapsulation (TLI_SAP) enables programs to be written independently of the communication channel used. Furthermore ACE contains C++ wrappers for many other Unix IPC mechanisms including System V Unix STREAM Pipes and Named Pipes, Threads, memory-mapped files, and System V IPC mechanisms such as Shared Memory, Semaphore and Message Queues.

The encapsulation of IPC mechanisms represents only one of ACE's many features. As shown in Figure 3, ACE also includes C++ components for event demultiplexing (Reactor), dynamic linking and service configuration (Service Configurator), multi-threading and concurrency control (Synch wrappers and Thread Manager), shared memory management (Shared Malloc), connection management (Connector and Acceptor), CORBA integration (CORBA Handler), layered service management (ASX), and distributed services like naming, locking, logging, and routing (which are not shown in the figure). Although it is beyond the scope of this article to cover these components you can find out more about them by examining the technical papers and documentation distributed with ACE.

Installation and Documentation

Although ACE is farther away from being a finished product than Socket++ and Net.h++, its installation runs smoothly if you use one of the compilers supported. For the ACE 3.3 version tested in our study these include Sun C++ 3.x and 4.x under SunOS 4.1.x and Solaris 2.x. More recent versions of ACE have now been ported to other OS platforms such as SCO Unix, HP-UX, SGI, OSF/1, AIX, Linux, Windows 95 and Windows NT.

Documentation is the biggest problem with ACE. The documention available consists mainly of articles in Postscript format that have been prepared by the author for technical journals and conferences. Each paper addresses certain aspects of ACE, but there is no comprehensive programmers guide. Although, ACE provides manual pages in nroff and HTML formats that are generated from the classes' header files, reading this material is insufficient to familiarise yourself with the complex relations between the different frameworks and components in ACE. Users have to dive into the source code and examples to figure it all out. This can be hard since the interaction between the ACE framework and application code uses callbacks and other forms of decoupled component integration. Application developers will need to have a good understanding of the structure and behavior of the framework tp use it effectively. The payoff can be considerable, but so is the effort required to understand the entire framework. Fortunately, ACE is designed in modular fashion, so it's possible to use subsets of its functionality quite easily without getting mired in the swamp of complexity.

Sample Program

Listing 3 shows the implementation of our netfinger example under ACE. From the multitude of functions available only the SOCK_SAP component will be used, which more or less corresponds to Net.h++ and Socket++, although it doesn't provide the higher-level C++ IOStream integration available with the other libraries.

Part (1) stays the same, under (2) a Socket object of the SOCK_STREAM type is generated. This example shows that ACE provides individual subclasses for each type of Socket, whereas Net.h++ and Socket++ provide a member variable that represents the Socket type. ACE's approach offers more type security, but potentially leads to code that is harder to change. (3) also illustrates that ACE spreads its functionality among more classes than the other two libraries. For instance, the ACE_SOCK_Stream class does not set up the connection itself but uses the service of the ACE_SOCK_Connector class. (4) shows ACE error treatment by way of checking return values and by using preprocessor macros. In (5) and (6) the lack of the familiar IOStream interface under ACE becomes apparent. The function send_n(), however, guarantees that the desired number of bytes will be sent. Finally in (7) the explicit closing of the Socket connection is shown.

Assessing the runtime efficiency of ACE is as difficult as with the other two C++ IPC libraries we've reviewed. However, ACE provides some potential performance advantages due to its extensive use of inlining and templates. A disadvantage of these optimizations, of course, is that the size of the object code can be quite large, depending on how the compiler implements templates and how ACE is conditionally compiled.

Assessment

Any application developer who is primarily interested in a stable Socket encapsulation in conjunction with a comfortable interface and professional support will appreciate Net.h++. I hope that RogueWave increases the number of communications channels supported and also offers frameworks for programming clients and servers at a higher abstraction level.

Socket++ may be an alternative solution, if you are willing to accept the limitations of a noncommercial product. Moreover, many developers have found that programmers of free software react very cooperatively to the "customer's requests."

Managing ACE requires the willingness on the part of developers to deal intensively with the program package. However, it offers quite a lot of functionality and has an active user community to provide support. Therefore, ACE is the right choice if you're looking for more than just interprocess communication support and if you can afford the effort needed to familiarise yourself with the framework.

References

[Brockschmidt:95] Kraig Brockschmidt, "Inside OLE, 2nd Edition", Microsoft Press, 1995.

[OMG:95] "The Common Object Request Broker Architecture and Specification," version 2, Object Management Group, July 1995

[OSF:95] OSF (Ed.), "Introduction to OSF DCE", Prentice Hall, 1995.

[Schmidt:92] Douglas C. Schmidt, "IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services," C++ Report, Volume 4, Number 9, 1992.

[Schmidt:95] Douglas C. Schmidt, "Design Patterns for Establishing Network Connections," C++ Report, Volumn 7, Number 9, 1995

[Sun:94] "Network Interface Programmer's Guide", Sun Microsystems Inc., 1994.

About the authors

Prof. Dr.-Ing. habil. Wilhelm Dangelmaier is Professor for Computer Integrated Manufacturing at the Heinz Nixdorf Institute of Paderborn University, Germany. His research interests include modeling of distributed manufacturing systems and tools to plan, schedule, monitor, and control these systems.

Dipl. Ing. Sören Henkel (soeren@hni.uni-paderborn.de) is a PhD student and scientific employee at the Heinz Nixdorf Institute of Paderborn University, Germany. His research focuses on a program system for model-based distributed production control.

Listings

Listing 1: netfinger with Net.h++

```
// netfinger: request to a  finger daemon

#include <stdlib.h>
#include <string.h>
#include „rw/net/inetaddr.h"  // Internet addresses
#include „rw/net/sockport.h"  // sockets
#include „rw/net/portstrm.h"  // socket IOStreams
#include „rw/net/winsock.h"   // for portability
#include „rw/net/neterr.h"    // exception handling

void usage(char* prog)
{
	cerr << „Usage: „ << prog << „ [user]@[host]" << endl;
	exit(1);
}

main(int argc, char **argv)
{
	// (1) for portability: initialize Winsock DLL
	RWWinSockInfo info;

	// (2) simple argument parsing
	if (argc!=2)
		usage(argv[0]);
	char* user=argv[1];
	char* host=strrchr(user,'@');
	if (!host)
		usage(argv[0]);
	*host++='\0';

	// (3) exception handling
	try
	{
		// (4) addressing
		RWInetAddr addr(„finger", host);

		// (5) connection establishment
		RWPortal fingerd=RWSocketPortal(addr);

		// (6) request
```

```
		fingerd.sendAtLeast( RWCString(user)+“\r\n“ );

		// (7) answer
		RWPortalIStream answer(fingerd);
		char c;
		while(answer.get(c))
			cout << c;
		cout << flush;
	//	(8) alternative:
	//	RWCString packet;
	//	while ( !(packet=fingerd.recv()).isNull() )
	//		cout << packet;
	}
	catch (const RWxmsg& x)
	{
		cerr << „Error: „ << x.why() << endl;
	}

	return 0;
}
```

Listing 2: netfinger with Socket++

```
// netfinger: request to a  finger daemon

#include <stdlib.h>
#include <string.h>
#include „sockinet.h"// socket stream in Internet domain

void usage(char* prog)
{
	cerr << „Usage: „ << prog << „ [user]@[host]" << endl;
	exit(1);
}

main(int argc, char **argv)
{
	// (1) simple argument parsing
	if (argc!=2)
		usage(argv[0]);
	char* user=argv[1];
	char* host=strrchr(user,'@');
	if (!host)
		usage(argv[0]);
	*host++='\0';

	// (2) iosockinet encapsulates a socket with IOStream semantics
	// (3) error handling: internally
	iosockinet fingerd(sockbuf::sock_stream);

	// (4) connection establishment
	fingerd->connect(host,"finger");

	// (5) request
	fingerd << user << „\r\n" << flush;

	// (6) answer
	char c;
	while(fingerd.get(c))
		cout << c;
	cout << flush;

	return 0;
}
```

Listing 3: netfinger with ACE

```
// netfinger: request to a  finger daemon

#include <stdlib.h>
#include <string.h>
#include „ace/SOCK_Stream.h"// socket of type SOCK_STREAM
#include „ace/INET_Addr.h"// address  in  Internet domain
#include „ace/SOCK_Connector.h"// active connection establishmnt

void usage(char* prog)
{
	cerr << „Usage: „ << prog << „ [user]@[host]" << endl;
	exit(1);
}

main(int argc, char **argv)
{
	// (1) simple argument parsing
	if (argc!=2)
		usage(argv[0]);
	char* user=argv[1];
	char* host=strrchr(user,'@');
	if (!host)
		usage(argv[0]);
	*host++='\0';

	// (2) ACE_SOCK_Stream encapsulates a socket of type SOCK_STREAM
	ACE_SOCK_Stream fingerd;

	// (3) connection establishment using an ACE_SOCK_Connector
	ACE_SOCK_Connector connector(fingerd,
		ACE_INET_Addr(„finger",host));
	// (4) Fehlerbehandlung: Rueckgabewerte und Makros
	if (fingerd.get_handle()==-1)
		LM_ERROR_RETURN((LOG_ERROR,
			„connection to host \"%s\" failed%p\n",host,"„),1);

	// (5) request
	char buf[1024];
	sprintf(buf,"%s%s",user,"\r\n");
	if (fingerd.send_n(buf,strlen(buf))==-1)
		LM_ERROR_RETURN((LOG_ERROR,"%p\n","send_n"),1);
```

```
	// (6) answer
	while(fingerd.recv(buf,1024)>0)
		cout << buf;
	cout << flush;

	// (7) explicitly close the connection
	if (fingerd.close()==-1)
		LM_ERROR_RETURN((LOG_ERROR,"%p\n","close"),1);

	return 0;
}
```

Figures

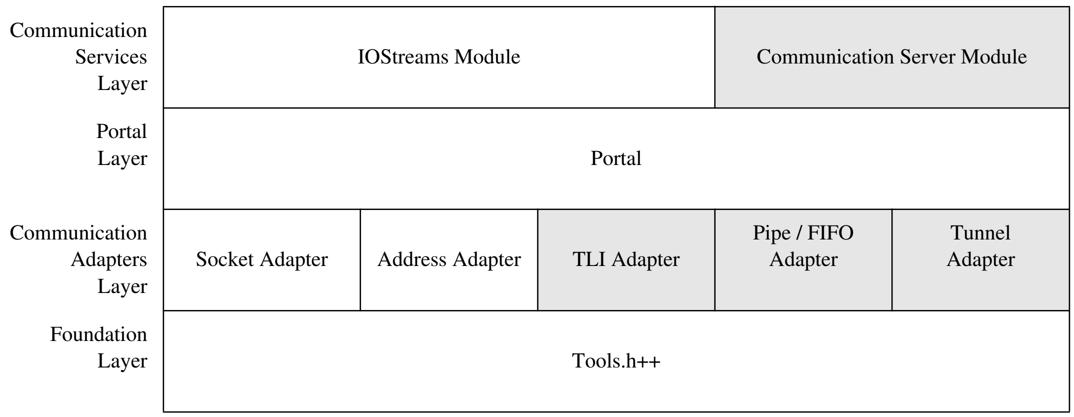

Figure 1: Net.h++ layered architecture

Figure 2: Socket++ class hierarchy

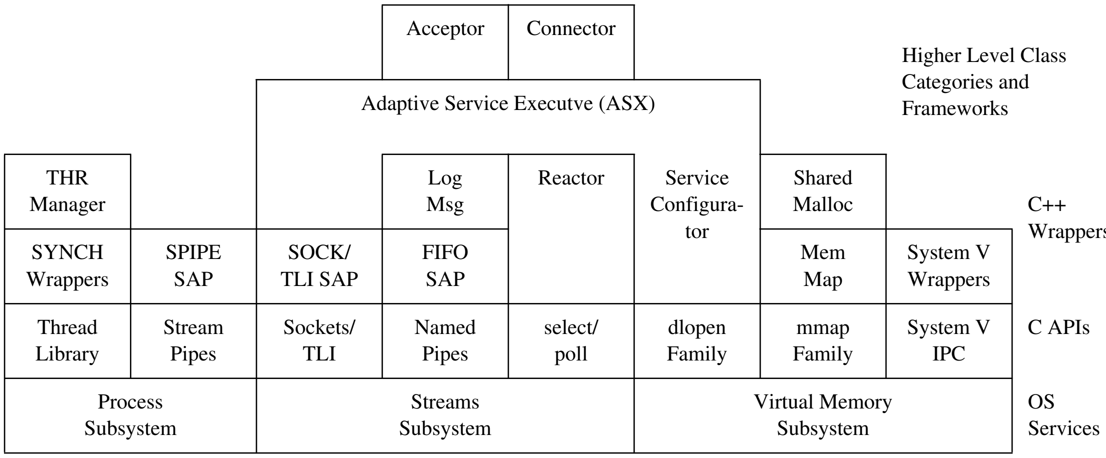

Figure 3: Components of ACE

Boxes

Box 1: Abstract

- Net.h++, Socket++, and ACE are object-oriented encapsulations of interprocess communication channels like Sockets, System V TLI, or Unix Pipes.
- Net.h++ is a commercial product, Socket++ and ACE are freely available software.
- Common to the three tools is comfortable support of Sockets and Socket addresses
- The tools differ in their basic philosophy and in their power. ACE offers the widest range of functionality, but also has the highest learning curve.

Box 2: What is important?

Platforms supported: The number of operating systems the product is offered or tested for should be as large as possible. This is, however, not the only criterion for assessing the possible range of applications. The number of communication channels supported, e.g. Sockets, System V TLI or the different Pipe mechanisms, is of almost equal importance. Their support should be extensive, since the implementations offered by the different suppliers (e.g. for Winsocks) can differ. Furthermore the number of compilers tested plays a certain role, because not every compiler supports all C++ features. If the library is not available in source code, incompatible object codes may also give rise to problems. The ideal class library should allow the development of a code which is independent from the biggest possible number of operating systems, communication channels, implementations of different manufacturers and compilers. The minimum requirement results from the application planned for the near future and from your own system environment.

Layered Architecture: The library should permit a relatively free selection of the degree of abstraction and aggregation during programming. This aim is backed by an appropriate inheritance hierarchy: the more abstract classes implement common features, the leaves of the inheritance tree realize the differences between differing communication concepts. In order to guarantee the highest possible degree of flexibility, access to the lower layers which are closer to the system must not be blocked. The syntax of this lower level should be largely based on the original C-API in order to make use of the programmer's empirical knowledge. At a high abstraction level complex program sequences should be formulated as simple as possible. In this context a comfortable handling is required which can be provided by an appropriate combination of single operations. It would be desirable to have access to communication channels using the C++IOStream interface. Heterogeneity can be improved using the transparent support granted by Sun XDR. Useful utility classes, e.g. for a presentation of addresses and services, round off the picture. Wherever possible the interface should offer useful default parameters - e.g. the local machine for describing a host or SOCK_STREAM for specifying a Socket type.

Extension Capacity: The library's class hierarchy should perform clear categorisations. This will make it easier for the programmer to integrate his own additions and at the same time increases the probability of reuse. If, for example, a communication mechanism is to be used, which has not been supported so far, the structure of the class hierarchy should give clear indications on how to integrate the addition in such a way that the new classes interact smoothly with the old ones. Individual extensions to the library are particularly easy when using C++ Templates.

Efficiency: With a view to interprocess communication the runtime behavior is of central importance. However, this behavior is difficult to measure, because the implementation of the library is only one of many variables. Other parameters include the specific application, the degree of network utilisation and the respective hardware or software platforms. For that reason, we have not focused on the run-time efficiency of the libraries in this article.

Error Treatment: Today Exception Handling provides the most flexible mechanism for treating errors or exceptions under C++. Exception Handling is particularly suitable for use in libraries, because it allows to separate error detection from error treatment. Unfortunately not all compilers support this mechanism - this will be different in the near future, however. As the minimum standard the user should be informed on the current error status by way of returned values or global variables, and the library should offer an appropriate default behavior for unanticipated conditions.

Other (general) requirements may include the documentation, the support offered or the compatibility with other existing libraries. The investment protection, which the manufacturer is able to offer, may be another assessment criterion.

Box 3: Abbreviations used

ACE	Adaptive Communication Environment
API	Application Programmatic Interface
CORBA	Common Object Request Broker Architecture
DCE	Distributed Computing Environment
FAQ	Frequently Answered Questions
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
IPC	Interprocess Communication
IPX/SPX	Internetwork Packet Exchange / Sequenced Packet Exchange
OSF	Open Software Foundation
OLE	Object Linking and Embedding
OMG	Object Management Group
OMT	Object Modeling Technique
OOD	Object-oriented Design
RPC	Remote Procedure Call
SAP	Service Access Point
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
TLI	Transport Layer Interface
UDP	User Datagram Protocol
URL	Uniform Resource Locator
XDR	eXternal Data Representation
WWW	World Wide Web

Box 4: +/- Net.h++

+: good architecture, intelligible interface

+: integration with Tools.h++

+: Exception Handling

+: good portability, many platforms

-: currently supports Sockets only

-: commercial product, expensive

Box 5: +/- Socket++

+: Free software

+: consistent implementation of C++ IOStream philosophy

+: supported by Unix Pipes, utility classes for fork(), Echo, FTP and SMTP

-: little support for several compilers

-: Unix only, no Winsocks

Box 6: +/- ACE

+: Free software

+: vast scope of components and functionality, many types of communication channels are supported

+: supports CORBA and Sun RPC

-: for many platforms only partially tested

-: little documentation for programmer, much experience required

Box 7: Summary

Library:

Number of classes

Net.h++:	50, of which API: 19/internal: 13/Exceptions: 18
Socket++:	32
ACE:	IPC_SAP: approx. 50, total: more than 300 (including examples)

Multiple inheritance

Net.h++:	yes
Socket++:	no
ACE:	yes

Scope of functions

Net.h++:	comfortable
Socket++:	good, some additional utilities
ACE:	vast

Communication channels supported

Net.h++:	Sockets (version 1.0)
Socket++:	Sockets, Unix (Named) Pipes
ACE:	Sockets, TLI, Pipes, Named Pipes, RPC, CORBA, Shared Memory

Ease of use

Net.h++:	very good
Socket++:	good
ACE:	should be improved, much experience required

Inlining

Net.h++:	no
Socket++:	no
ACE:	intensive (can be controlled by conditional compilation)

Support of XDR

Net.h++:	yes, via Tools.h++ Virtual Streams
Socket++:	no
ACE:	limited

Error treatment

Net.h++:	Exception Handling
Socket++:	returned values, internal
ACE:	returned values, macros

Documentation:

Programming Manual

Net.h++:	printed, approx. 140 p.
Socket++:	Postscript file, approx. 50 p.
ACE:	no

Tutorial:

Net.h++:	yes
Socket++:	no
ACE:	no

Test and example programs

Net.h++:	yes
Socket++:	yes
ACE:	yes

Man pages

Net.h++:	no
Socket++:	no
ACE:	yes

GNU Texinfo files

Net.h++:	no
Socket++:	yes
ACE:	no

Distribution:

Version

Net.h++:	1.0
Socket++:	1.10
ACE:	3.3

Source

Net.h++:	RogueWave Software, Inc. P.O. Box 2328 Corvallis, OR 97339 (503) 754-3010, (800) 487-3217
Socket++:	ftp://ftp.virginia.edu/pub/socket++-1.x.tar.gz, ftp://ftp.th-darmstadt.de/pub/programming/languages/C++/ class-libraries/networking/socket++-1.x.tar.gz
ACE:	http://www.cs.wustl.edu/~schmidt/ACE.html, ftp://ftp.th-darmstadt.de/pub/programming/languages/C++/

class-libraries/ACE

Price

Net.h++:	Object Code $594, Source Code $1794, additionally $474 for Tools.h++ (if not yet purchased)
Socket++:	Freeware
ACE:	Freeware

Platforms/compilers

Net.h++:	Unix (Solaris, SunOS, HP-Unix), Windows (Winsocks by Microsoft, NetManage Chameleon, Trumpet and Wollongong), Windows NT. Planned: AIX, SGI, OS/2, Windows95
Socket++:	Unix, gcc 2.4 and higher or cfont 3.0 and higher
ACE:	SunOS 4.1.x or Solaris 2.x. with Sun CC 3.x or 4.x, SCO, HP-UX, SGI, OSF/1, AIX, Linux, Windows95 and NT

Source Code

Net.h++:	yes, for extra charge
Socket++:	yes
ACE:	yes