Abstracting Low-Level Network Programming With ACE, a Pattern-Oriented Network Programming Framework

Michael LeMay
Computer Sci. Department
UW-Eau Claire
2316 80th St.
Eau Claire, WI 54703
lemaymd@uwec.edu
Phone: (715)832-9951
Fax: (715)836-2923

Jack Tan
Computer Sci. Department
UW-Eau Claire
105 Garfield Ave. Box 4004
Eau Claire, WI 54702
tanjs@uwec.edu
Phone: (715)836-2526
Fax: (715)836-2923

Presented By: Michael LeMay

Abstract— ***Network protocols such as the File Transfer Protocol (FTP) have been vital to the success of the Internet. Network operating systems have developed APIs to allow application programmers to easily implement these protocols. The most popular API in usage today is the BSD Sockets API. This API provides low-level operations for performing connection establishment, message transmission and reception and provides ways to configure various low-level network parameters. However, this low-level API forces network programmers to duplicate code in and between applications, lends itself very poorly to common Object-Oriented and Pattern-Oriented software design methodologies and reduces the level of abstraction present in an application, making the application much more difficult to comprehend, extend and maintain. By applying a popular C++ network programming framework, ACE, it is possible to retain all the low-level detail and flexibility offered by the Sockets API while addressing its shortfalls.***

Keywords: pattern, ace, ftp, framework, concurrency

I. Introduction

ACE [3], the ADAPTIVE (A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment) Communication Environment, is an object-oriented network programming framework that implements many core patterns useful in network applications, especially those that make use of concurrency. The target audience of ACE includes developers of high-performance and real-time network applications, but ACE is generally applicable to a wide range of network-programming problems.

Some of the components provided by ACE for reuse in network applications are:

1) C++ wrapper facades that abstract platform-specific operations
2) Framework components that perform a variety of tasks, comprising:
    a) Event handler dispatching
    b) Signal handling
    c) Service initialization
    d) Inter-process communication (IPC)
    e) Message routing
    f) Synchronization

Although ACE can present a steep learning curve, once it is mastered it is an unequaled tool to the network application developer.

In this paper, the design of an efficient FTP protocol is explored and justified. The implementation of high-performance, concurrent client and server software using ACE and conforming to the protocol is then presented. ACE is used as the basis for both pieces of software, and was used to full advantage to provide powerful functionality and support for their features.

The FTP system described herein is based upon a session concept. A single session as viewed by a mirror (server) consists of a single network data connection between the mirror and a client. On the other hand, a session as viewed by the client consists of one or more connections to one or more mirrors, leading to sub-file-level concurrency that can be exploited to attain higher transfer rates by striping a file transfer across multiple connections.

II. Mirror Discovery and Session Negotiation

Clients can utilize an atypical method of connecting to mirrors available on a local network. It is possible for a client, using multicast, to scan the network for beacons, which are network programs that advertise the availability and properties of mirrors. These beacons, upon receiving the client's request for information, transmit a response datagram to the client containing information about the mirrors they represent. Upon receiving that information, the client transmits a datagram to each mirror in which it is interested requesting a number of sessions. Each mirror will then fulfill the session request to the best of its ability by opening TCP or UDP data connections to the client. The protocol does not specify whether UDP or TCP must be used, either will operate correctly.

On the other hand, if multicast mirror beacons are not available, or if the client knows the Internet address of a particular mirror, it may send a session request datagram directly to that mirror. The session negotiation will then proceed as it normally would.

This flexible session negotiation scheme provides a simple way of utilizing maximum possible concurrency within a local network, as the client can automatically detect all currently available mirrors on the local network.

The event handling capabilities provided by ACE were used to implement the multicast and datagram servers. The servers simply register themselves as being able to handle input events originating from either a datagram socket or multicast channel, and instruct ACE to enter an event loop, waiting for those events. Whenever such an event arrives, ACE performs a callback out of the event loop to the handler and allows it to process the data received before reentering the event loop. ACE also provides full support for datagram and multicast sockets. No low-level code was created to perform I/O operations using either type of socket. ACE abstracts socket I/O functions to simple send/recv method calls on objects that automatically manage the underlying connections.

III. Efficient FTP Protocol

Once a session has been established between a client and mirror, a specific FTP protocol is executed to allow remote filesystem manipulation.

This FTP protocol was designed with the objective that it provide basic commands to users to allow them to detect mirrors available on the network, connect with the detected hosts or other manually selected ones, navigate through the mirrors' filesystem, upload new files, download files, and delete/modify existing files. It must support text and binary data.

To that end, operations were defined in the protocol to:

1) detect mirrors
2) connect to mirrors
3) download file
4) disconnect from mirrors
5) change directory
6) change file permissions
7) move file
8) list files in directory
9) remove file
10) create directory
11) upload file

Remote error messages are also supported by the protocol.

For the sake of brevity, the full message formats for each of these commands is omitted, but the packets used to transmit them all conform to a common format, which is discussed here.

The first part of each packet is the header. The header first contains the packet's sequence number encoded as a network byte-order 16-bit unsigned integer. The FTP protocol is defined to operate with user-created flow control schemes, such as sliding-window [13] protocols. These protocols require packet sequence numbers.

The next field in the header specifies the type of the packet, specified as an 8-bit unsigned value. Packets may contain either data or acknowledgments for other packets. Again, acknowledgments are used in flow control protocols.

The final field in the header specifies the length of the packet as a 32-bit network byte-order unsigned integer. This specifies the length in bytes of the payload following the header.

The next portion of the packet contains the actual payload. This is a raw binary structure with a variable length specified in the header. Messages are segmented as they move down the protocol stack towards physical transmission. Each packet's payload consists of one of those low-level segments. The first segment from each message contains a 32-bit command ID that is used to differentiate different types of messages when the message is reconstructed on the receiving end of the protocol.

The final portion of each packet contains a 16-bit checksum value, computed using the Internet Checksum [12] algorithm. This checksum is used to verify the integrity of the packet it is associated with.

A few common message types are specified in the protocol. They represent generally useful values such as character strings and file paths. These basic messages can be aggregated to form larger messages with greater meaning, such as messages representing a command to move or download a file.

An aggregate message can be visualized as a tree structure. The parent (root) message in the tree contains zero or more special attributes, often represented as integers or other simple types, that are transmitted directly in the parent message's packet payload. Then, each message that sits below the parent message in the hierarchy is transmitted in a strictly-defined sequence. On the receiving end of the protocol, the parent message is received first, and as each child message is received, it is used to reconstruct the overall message. This basic pattern may recur arbitrarily many times in a single high-level message.

As an example of packet aggregation, consider the message used to transfer a file listing from a mirror to a client. The first message transmitted simply contains a 32-bit unsigned integer specifying the number n of files represented in the list and identifies the message as a file listing. Following that packet, n file information packets are transmitted. Each file information packet contains a number of attributes associated with the file such as permissions and size information, encoded as integers and converted to network byte-ordering, and is followed by a single file path packet. Each file path packet is followed by a string packet, with contains the actual characters that make up the string.

From this example, it is possible to glimpse the flexibility allowed by this scheme. As will be demonstrated later, ACE makes the implementation of this scheme very straightforward and efficient.

One final feature of these messages is that they may be compressed before being transmitted. If a message exceeds a certain application-defined size threshold, the message is compressed using Zlib [11] and repackaged before being segmented and transmitted. To indicate that compression has occurred, a special escape sequence is inserted in the resulting message. After being reconstructed on the receiving end of the protocol, each message is inspected to see if it contains the escape sequence, and if it does it is uncompressed before being processed further.

IV. Common Design and Implementation

Because of the regular format of the packets described previously, it is possible to construct a common library of classes to deal with packet formatting and I/O. ACE provides a "Streams" framework for the express purpose of simplifying implementation of functionality such as this. An instance of the streams framework is actually modeled as a pair of uni-directional streams. One stream flows up, towards the top (high-level end) of the protocol stack while the other stream flows down, towards the bottom (low-level end) of the stack.

The entities that perform work in each layer of the stack represented by the stream are known as modules. A module consists of a pair of active objects. Each active object contains a procedure that operates independently of all other procedures, in its own context. It simply processes data that is passed to it and subsequently passes the processed data to the next object in the stack.

The unit of communication that is passed between layers in the stack and active objects is called a message block. A message block is any object that represents information to be passed through the stack. In this application, specialized message blocks were created to represent packet headers, payloads and checksums, as well as high-level conceptual packets. Message blocks serve to unify the interfaces between layers, since every layer communicates using message blocks.

Active objects actually operate by interpreting messages from a message queue. They operate in a loop and block on a method invocation that retrieves the next message from a dedicated message queue provided by ACE for each individual object. Other objects communicate with this object by asynchronously placing messages into the queue to be interpreted by the object. This allows a high degree of concurrency between cooperating objects, since their communication is so loosely coupled.

This entire paradigm of passing a common data representation through active objects that process the data and are organized in a linear fashion is known as the "Pipes-and-Filters" architectural pattern. "Pipes-and-Filters" provides a very direct and convenient way of implementing network protocol stacks.

The highest-level layer in the protocol stack is provided by the specific application using the stack. This is because mirrors and clients interpret high-level packets differently. However, there exists some common functionality between both pieces of software that is concerned with handling aggregate messages. Messages that are received are passed up to the high-level message handler for interpretation. This handler maintains a stack of incompletely formed messages that have been received.

The handler invokes a virtual method on each message received that asks if it is completely formed. The definition of this method varies according to message type. If the message is complete, and there is another incomplete message on the stack, it is passed to that message so that it can be aggregated into the incomplete message. That previously incomplete message is then queried, to discover if the new message completed it. If it did, the entire message is popped off the stack and passed to the application for interpretation. Otherwise, the message remains on the stack. If, however, the stack is empty when an incomplete message arrives, it is placed directly onto the stack. In this fashion, deeply recursive and complex aggregate messages can be simply and efficiently handled independent of the type of message.

The second layer down in the protocol stack performs optional compression/decompression of high-level packets. The compression object inspects each packet passed to it and decides whether it is large enough to require compression. If it is, the compression object creates a new message block large enough to hold the compressed message and initializes it with the special escape sequence that notifies the receiving decompressor that the message is compressed, and also includes the original size of the packet. The compression object then compresses the entire old packet into the new message block and sends it down the rest of the stream. If the packet is not large enough to warrant compression, it is propagated down the stream unmodified.

The decompression object on the receiving protocol stack inspects each message passed up through it, searching for the compression escape sequence. If the sequence is located, the decompression object inspects the size field contained after the sequence and creates a new message block large enough for the resultant packet. The decompression object then decompresses the compressed packet into the new message block. The final resultant packet is then propagated up the stack to the application-defined packet interpretation layer.

The third layer down the stack is responsible for segmenting/aggregating packets. The segmentation object inspects a packet passing down through it, and if its size exceeds the limitations of the underlying transport medium, it breaks the packet into smaller packets that can be reassembled on the receiving end of the protocol. The aggregation object is simply responsible for reversing this process before transmitting packets up the protocol stack.

The second layer from the bottom of the stack performs flow control on packet traffic. It implements one of a variety of flow control algorithms and basically regulates how quickly packets are transmitted on the transmitting side of the protocol, and how packets are received and re-sequenced if they arrive out-of-order on the receiving end of the protocol. The transmitting object is also responsible for dealing with acks received from the receiving end of the protocol. Typically, if an ack is not received within a certain time period, the packet served by that ack is resent.

The lowest layer in the protocol stack deals with actually transmitting and receiving packet segments to and from the underlying network transport medium. The transmitting object transmits the segment header, payload and checksum in sequence, data which is then received by the remote receiving object, again in sequence. The receiving object bears the additional responsibility of generating acks for uncorrupted packets received and transmitting those acks to the sender. The sending object decomposes segments into their components, while the receiving object reconstructs a segment from those individual components.

The most important features to notice about this entire system are that the system is easy to analyze and design in components, since the working of each layer is separated from the working of any other layer, and that using individual layers facilitates component interchangeability.

For example, a number of flow control strategies are supported by the stack. Upon session establishment, the mirror sends a configuration message to the client indicating which flow control algorithm is in use. The client uses this information to configure its own protocol stack to be compatible with the mirror's. Furthermore, since each connection between a client and mirror contains its own protocol stack, a client may simultaneously use a variety of flow control mechanisms on different connections within a single client session.

Another example of the modularity and flexibility of this system is the simple fact that it is almost completely reusable between client and mirror software. The only component that changes between them is the highest-level interpretation layer. This is in striking contrast to many other procedural network applications that have completely separate codebases for client and server software.

Layering the protocol stack in this manner also provides a great deal of concurrency. Each object in each layer operates in its own thread (lightweight process) and communicates with surrounding layers only indirectly through message queues. This provides high application performance and scalability, especially when the application is executed on a multi-processor system.

Finally, the abstract data representation afforded by message blocks promotes data-agnosticism throughout the stack. It was pointed out, for example, that compressed and decompressed packets are treated equivalently by all low-level stack components, and are only dealt with specifically by the compression layer. It is conceivable and likely that encryption of data packets would also be desirable in many domains. It would be a trivial task to insert an encryption layer somewhere within the stack, and could be done without affecting any other layer.

Once again, not a single line of low-level sockets programming or synchronization code was written in the entire system. ACE proved to be totally sufficient for implementing the entire system and practically forced a modular, object-oriented and pattern-oriented design on the system.

V. Client Design and Implementation

The client software that lays on top of this stack operates by interactively interpreting user commands and executing them using the sessions established between it and any number of mirrors.

The front end of the software was implemented using the standard GNU [8] tools Flex [14], Bison [15] and Readline [16] to create a simple, familiar command line. Common commands such as `bye`, `mv`, `cd`, `mkdir`, `chmod`, `ls`, `put`, `get`, `rm` and `conn` (similar to open) were provided, making this a fairly full-featured FTP implementation.

Upon receipt of a command, the client software generates messages corresponding to the command to be executed, and pushes these messages into the protocol stack, which then autonomously processes the messages and ensures their delivery. Their is a clear separation between command formulation and command delivery.

Responses to the messages are likewise delivered up through the protocol stack from the mirrors and may be interpreted and displayed to the user if appropriate. An error message is one such message that would be displayed to the user.

On the other hand, many messages have significance only for the client software. For example, to initiate a file download the client software sends a download request message to a mirror, that responds with detailed information about the file. The client software receives this message and uses the information contained within it to generate a number of messages to send to each mirror, instructing them to begin sending file data. In response, each mirror sends the proper chunks of the file to the client software, which proceeds to store it in a local file.

Multi-streaming striped downloads are enabled by the high-degree of application modularity encouraged by ACE's architecture. Being able to encapsulate an entire protocol stack within an ACE stream that manages communication between a single mirror and the client makes it simple to divide work between different communication links.

VI. Server Design and Implementation

The server software differs from the client software mainly in that it operates non-interactively, providing no facility for directly accepting commands from a user. Instead, it interprets commands received though the network from remote clients.

VII. Performance Results

One of the claimed advantages of the ACE framework is its high-performance. The reasons for this are obvious, since it enables such a high degree of concurrency and promotes quality application designs. Despite the fact that the network available for testing the application was heavily degraded due to poor configuration and overloaded network devices, the performance results were encouraging. The advantages offered by striped file transfers and the cross-platform flexibility inherent in the application are made apparent by these results:

Multi-streaming Configuration Using Selective Repeat Flow Control:
1 connection from x86 cluster to x86 multiprocessor
1 connection from x86 cluster to dual G4 Macintosh
Adaptive acknowledgment timeout
Window size: 100
8192 byte packets
120MB file

Results:
Throughput from x86 multiprocessor: 588551B/s (575KB/s)
Throughput from G4 Macintosh: 354490B/s (346KB/s)
Throughput total: 920.938kB/s

VIII. Conclusion

In this paper, an efficient binary FTP protocol was presented and discussed. The object-oriented ACE framework was also presented as a tool that can be quickly and easily used to write portable, flexible and maintainable network applications that make use of well-established network programming patterns and paradigms. Finally, performance results for one implementation of the FTP protocol using ACE were presented and analyzed.

References

[1] Carnegie Mellon S.E.I., "Object-Oriented Analysis", http://www.sei.cmu.edu/str/descriptions/ooanalysis.html
[2] Schmidt, Douglas C.; "Design Patterns, Pattern Languages, and Frameworks", http://www.cs.wustl.edu/ schmidt/patterns.html
[3] Schmidt, Douglas C.; "The ADAPTIVE Communication Environment", http://www.cs.wustl.edu/ schmidt/ACE.html

[4] Schmidt, Douglas C.; Stal, Michael; Rohnert, Hans; Buschmann, Frank; “Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects”, 2000

[5] Syyid, Umar; “A Tutorial Introduction to the ADAPTIVE Communication Environment (ACE)”

[6] Schmidt, Douglas C.; “The ADAPTIVE Communication Environment, An Object-Oriented Network Programming Toolkit for Developing Communication Software”, June 14-17, 1993

[7] Schmidt, Douglas C.; “An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit”

[8] “Philisophy of the GNU Project”, http://www.gnu.org/philosophy/philosophy.html

[9] Postel, J.; Reynolds, J.; “FILE TRANSFER PROTOCOL (FTP)” http://www.ietf.org/rfc/rfc0959.txt, Oct 1985

[10] “BSD Sockets”, http://www.ecst.csuchico.edu/ chafey/prog/sockets/sinfo1.html

[11] “Zlib Compression Library”, http://www.gzip.org/zlib/

[12] Braden, R.; Borman, D.; Partridge, C.; “RFC 1071 – Computing the Internet Checksum”, http://www.faqs.org/rfcs/rfc1071.html

[13] “Sliding Window Protocols”, http://www.freesoft.org/CIE/Course/Section4/5.htm

[14] “GNU Flex”, http://www.gnu.org/software/flex/

[15] “GNU Bison”, http://www.gnu.org/software/bison/

[16] “The GNU Readline Library”, http://cnswww.cns.cwru.edu/php/chet/readline/rltop.html