Schmoozing with the OmniNetworking Framework

William Garrison
garrison@standardorbit.com

Abstract

Historically, the Cocoa frameworks have not provided integrated support for TCP/IP network programming. The OmniNetworking framework fills this gap by providing Objective-C wrappers around the BSD socket API that integrate well the Cocoa frameworks. This paper presents an introduction to the OmniNetworking framework and its use in developing client and server applications for TCP/IP network communication.

Introduction

Mac OS X provides a variety of APIs for TCP/IP network programming. The Carbon environment uses Open Transport as its native networking API. CoreFoundation provides CFSocket, a C-based API with pseudo object-oriented semantics. Historically, Cocoa (nee OpenStep) has not provided its own object oriented classes for network programming, relying instead on the C-based BSD sockets API to provide this service.

The OmniNetworking framework fills this gap for Cocoa developers by providing an Objective-C framework encapsulating sockets in an object-oriented interface. With a few classes plus NSData and NSString, the Cocoa programmer can use OmniNetworking to create sophisticated TCP/IP network applications in relatively short order.

BSD Sockets API

BSD sockets is a mature and well documented networking API available in all flavors of UNIX, Linux, even Windows (although not without a few Borg enhancements). "Unix Networking Programming" [Stevens90] and "Internetworking with TCP/IP, Vol III" [Comer93] are classic references for writing clients and servers for TCP/IP networking using this API. One or both of these should be on your shelf (and can probably be found at used book sales for bargain prices, given their mature age).

Understanding the OmniNetworking framework is helped by having some familiarity with the concepts and terminology of BSD sockets. The following is an extraordinarily brief introduction. A more complete presentation of BSD sockets can be found in Chapter 6 of Stevens. The real networking geek will want to check out all of Comer, Vol III. The terms "socket" and "inet" provide reasonable starting points for information in the man pages.

Sockets

As the name implies, sockets are the organizing principle in the sockets API. A *socket* represents one endpoint in a network communications channel. A complete network connection involves two sockets — one for the originating end and another for the target — through which two processes on those computers can exchange data with each other.

A socket describes two important elements of the network endpoint: the host address and the port number. The *host address* uniquely identifies the network and the computer where the endpoint is located. In the sockets API, host addresses are represented as 32 bit unsigned integers by the C structure *struct in_addr*. The *port number* maps the network connection to a specific running process that will be sending and/or receiving data from that host address. Port numbers are represented by a 16 bit unsigned integers.

Addressing

Hosts are usually (and more conveniently) referred to by their host names. *Host names* can be expressed in alphanumeric form (e.g. hal9000.ibm.org) or as a "*dotted quad*"[1] (e.g. "10.1.0.40"). A host name has to be resolved

into its corresponding *struct in_addr* host address before it can be used in the socket API. Dotted quad names can be transformed directly into *struct in_addr* form. Alphanumeric *name resolution* usually involves contacting a DNS (domain name server) somewhere on the network, through which the name is mapped into its dotted quad equivalent, then into a *struct in_addr*.

TCP vs UDP

Sockets in the Internet protocol (IP) family usually come in UDP and TCP flavors. *UDP* sockets can send and receive bytes using the User Datagram Protocol, which is intended for connectionless, one-way delivery of datagrams (messages of known length). UDP makes no guarantee that its messages will be delivered reliably or intact. *TCP* sockets use the Transmission Control Protocol to provide reliable, two-way communication of byte streams over the network. TCP is intended for use in connection-oriented communications. Most Internet applications use the TCP protocol.

Using the BSD Socket API

A socket, in either TCP or UDP flavors, is created using the *socket*() routine. This newly created socket must then be mapped to the local host address and a port number using the *bind*() routine.

Client applications use the *connect()* routine to initiate a network connection between its local socket and another, typically located on another computer. A server application uses the *listen()* call to direct its own local socket to listen for incoming connection requests. A server then calls *accept()* to receive and complete those requests.

Once a connection is established between two sockets, data can be sent or received across the connection using the *read*() and *write*() functions. When communications are no longer needed, the *close()*, *shutdown*() or *abort*() routines are used to close the connection and release the local socket from memory.

BSD sockets, given its basis in C, can be used directly in any Cocoa application. The API is simple and is supported by an abundance of documentation in the way of books and freely available source code.

OmniNetworking Framework

The OmniNetworking framework offers two compelling features for the aspiring developer of Cocoa-based network applications: a simple object-oriented interface and convenient integration with the Cocoa frameworks. Two classes, ONHost and ONTCPSocket, handle the most of the heavy lifting in TCP/IP networking. OmniNetworking's network I/O methods are written in terms of NSData and NSString, making it easy to use the network as a data source for other Foundation and AppKit objects in your Cocoa application.

A Lay of the Framework

The OmniNetworking framework contains thirteen classes, but the ones you'll use most frequently for TCP/IP applications are ONHost, ONTCPSocket, and ONSocketStream. ONHost provides host name to address translation methods. ONTCPSocket represents a TCP socket. ONSocketStream takes an ONTCPSocket and provides direct write and buffered read access. I have found that for immediate reference, the source and headers for these four classes are indispensable: ONHost, ONInternetSocket, ONTCPSocket, and ONSocketStream. To get a complete more complete understanding of these classes, you'll need to also look at ONSocket and ONHostAddress.

ONHost

The starting point for using OmniNetworking is ONHost. A client application will need an ONHost object to represent the target for a network connection. A server will have one available identifying the computer from which it has accepted a connection.

```
+ (ONHost*) hostForHostname:
                    (NSString*) aHostname;
- (NSString*) hostname;
- (NSString*) canonicalHostname;
+ (NSString*) localHostname;
```

hostForHostname: is the most frequently used method of ONHost. It takes a host name and

returns an initialized ONHost object to represent it. The host name argument can be specified in dotted quad form, or as an unqualified or fully qualified domain name. ONHost caches name-to-address translations internally and will satisfy subsequent lookups from there before going back out to a DNS. The *hostname:* method returns the name used to initialize an ONHost object. The *canonicalHostname:* method returns the host's canonical DNS name. When the ONHost class is loaded and initialized at runtime, the local host's name is obtained from the system. The *localHostname:* class method returns this as an NSString. *hostForHostname*: will throw an exception if an error occurs doing the name resolution. Refer to the ONHost header for more details or check out the example project, SchmoozingExamples, accompanying this paper.

ONTCPSocket

The socket classes of OmniNetworking are the workhorses of the framework. ONTCPSocket and ONUDPSocket have methods for creating new socket objects, initiating connections, reading and writing data. ONTCPSocket also has methods for establishing server listeners and accepting new connections on a socket. I will discuss ONTCPSocket and its uses in the rest of this paper. ONTCPSocket's methods throw a variety of exceptions when error conditions arise. Refer to the headers for ONTCPSocket and ONInternetSocket, or the SchmoozingExamples project for more details.

Creation method
+ (ONTCPSocket*) tcpSocket;

ONTCPSocket provides the *tcpSocket:* factory method for creating a properly initialized and autoreleased TCP socket object. You will need to retain the ONTCPSocket if it is needed beyond the scope of the autorelease pool in which it was created.

Connection methods
- (void) setLocalPortNumber;
- (void) connectToHost: (ONHost*)host
port: (unsigned short) port;
- (void) abortSocket;
- (void) startListeningOnLocalPort:
(unsigned short)port;
- (void) acceptConnection;
- (ONTCPSocket*)
acceptConnectionOnNewSocket;

With an ONTCPSocket in hand, you can use it to either make or receive a network connection. You can send it the *connectToHost:port:* message, specifying the ONHost representing the target and the port number of the service offers on that host.

For a typical client application, an arbitrary unused port number is used on the local end of the connection. If no port has been explicitly set on the local socket, BSD will bind the local socket to a system-selected port number [2] as a side effect of the *connectToHost*: method. Should a specific local port number be needed, the socket can be configured with one before initiating a connection by sending it the *setLocalPortNumber*: message.

ONTCPSocket automatically takes care of closing a connection in common circumstances. When an ONTCPSocket object is deallocated after receiving its final *release:* message, the underlying BSD socket is gracefully closed. Also, when an end-of-file is detected during a socket read (an indicator that the socket on the other end of the connection has closed), the ONTCPSocket object will also gracefully close its own socket.

An ONTCPSocket can be directed to explicitly close its side of a socket connection sending it the *abortSocket:* message. In response, the ONTCPSocket will immediately shutdown its underlying BSD socket, refusing to send or receive any more bytes.

A server application can configure its ONTCPSocket object to listen for incoming connections by sending it the *startListeningOnLocalPort*: message. A port number must be chosen that will be well-known to all clients [3]. To accept those incoming connections, a server would send *acceptConnectionOnNewSocket:* or *acceptConnection:* to its listening socket. *acceptConnectionOnNewSocket:* is the more commonly used method. It directs a socket to

accept an incoming connection request and returns a new ONTCPSocket object dedicated to handling it, leaving the socket in its listening mode, ready to accept the next connection. The new socket is bound to an arbitrary port on the server, leaving the well-known port available to the listening socket.

Reading and writing methods
- (void) setNonBlocking:
(BOOL) shouldBeNonBlocking;
- (void) setStringEncoding:
(NSStringEncoding) aStringEncoding;
- (void) setReadBufferSize:(int) aSize;
- (void) readData:
(NSMutableData*) dataRead;
- (NSData*) readData;
- (NSString*) readString;
- (void) writeData:(NSData*) someData;
- (void) writeString:(NSString*) aString;
- (void) writeFormat: (NSString*) aFormat, ...;

The convenience with which one can read and write data to a network connection is one of OmniNetworking's most attractive features. Data from the network connection can be read directly into NSData or NSString objects using the *readData*: or *readString*: methods. By default, a 2048 byte buffer is used for reading off the socket. This buffer can be resized using *setReadBufferSize*: method. The corresponding methods for writing to the network are *writeData*: and *writeString:*. Formatted strings can also be written to the network using the *writeFormat:* method. NSStrings are read from or written to the network using ISO Latin-1 as the default character encoding. An ONTCPSocket can be configured to use any encoding available to NSString by way of the *setStringEncoding*: method. Caveat emptor: the *readString:* method's implementation is valid only for single byte character encodings (e.g. ASCII, ISO Latin-1, or UTF-8). When reading from multi-byte encoded character streams, like Unicode, the NSData-based read methods should be used instead.

Status methods
- (BOOL) isConnected;
- (BOOL) didAbort;
- (BOOL) isReadable;
- (BOOL) isWritable;

ONTCPSocket provides methods for determining connection and I/O status. *isConnected:* and *didAbort:* return YES if the socket object is connected or has been disconnected. If you are writing a non-threaded network application, *isReadable*: and *isWritable*: can be useful for constructing network I/O polling loops that keep your application from blocking. If no data is available to be read from the network connection, *isReadable*: returns NO without blocking program execution. If the socket cannot accept more data for writing to the connection, *isWritable*: returns NO, also without blocking.

Attribute methods
- (int) socketFD;
- (const struct sockaddr_in*) localAddress;
- (const struct sockaddr_in*) remoteAddress;
- (ONHost*) remoteAddressHost;
- (NSMutableDictionary*) debugDictionary;

ONTCPSocket provides a number of accessor methods to return the BSD socket level data structures associated with the connection. *socketFD:* returns the file descriptor for the underlying BSD socket. *localAddress:* and *remoteAddress:* return the BSD socket address description (struct sockaddr_in) for the local and remote ends of the connection, from which the Internet host address and port numbers can be accessed. *remoteAddressHost:* returns an ONHost representing the host at the other end of the connection. The *debugDictionary*: method returns a description of the ONTCPSocket object containing useful debugging information consisting of the BSD socket descriptor and the state of the socket object (connected, listening, or aborted).

ONSocketStream

The ONSocketStream class provides a higher level interface to ONTCPSocket that is particularly useful when working with ASCII or ISO Latin-1 based Internet application protocols. ONSocketStream features methods for performing buffered reads on a TCP socket and for reading network data one line at a time.

Creation methods
+ (id) streamWithSocket:(ONSocket *)aSocket;
- (id) initWithSocket:(ONSocket*)aSocket;

An ONSocketStream object layers functionality on top of that provided by ONTCPSocket, so an instance of an ONTCPSocket is a prerequisite for its creation. ONSocketStream throws no specific exceptions of its own, but does relay those passed from its ONTCPSocket object. *streamWithSocket*: returns an allocated, initialized and autoreleased ONSocketStream instance. *initWithSocket*: will initialize a manually allocated ONSocketStream instance.

Reading and writing methods
- (NSString*) readLine;
- (NSString*) peekLine;
- (NSData*) readData;
- (NSData*) readDataOfLength:
(unsigned int)length;
- (NSData*) readDataWithMaxLength:
(unsigned int)length;
- (NSString*) readString;
- (unsigned int) readBytesWithMaxLength:
(unsigned int)length
intoBuffer:(void*)buffer;
- (void) readBytesOfLength:
(unsigned int)length
intoBuffer:(void*)buffer;
- (void) writeData:(NSData*)theData;
- (void) writeString:(NSString*)theString;
- (void) writeFormat:(NSString*)aFormat, ...;

ONSocketStream implements its writing methods by calling through to the corresponding methods on its instance of ONTCPSocket. Its read methods are implemented around an internal NSMutableData object that serves as a read buffer.

The *readLine*: method returns a "line" of data from the network connection as an NSString. The line includes all bytes up to the first newline character ('\r') or carriage return-newline characters ('\r\n'). To get a line of network data without removing it from the buffer, use the *peekLine*: method. *readData*: returns an NSData holding all of the bytes currently in the ONSocketStream's buffer or available from the socket, if the buffer is empty. An arbitrary number of bytes can be read from the socket using the *readDataForLength*: message. To read from the socket up to a maximum number of bytes, send the *readDataWithMaxLength*: message. The *readBytesWithMaxLength*: and *readBytesOfLength*: methods can be used to read into an arbitrary C-based character buffer. The *readString*: method returns the data currently available on the socket as an NSString. As with ONTCPSocket, this *readString:* method is only valid when reading single byte encoded character streams. Multi-byte encoded streams should be read with the NSData-based methods.

Writing A TCP Client

Comer describes the following algorithm for writing a TCP client [Comer93, p. 64]:

1) Find the IP address and port number of the server.
2) Allocate a socket.
3) Specify that the connection needs an arbitrary port on the local server.
4) Connect the socket to the server.
5) Communicate with the server using an application-level protocol (e.g. sending requests and awaiting replies).
6) Close the connection.

Given this basic algorithm, an exceptionally trivial TCP client application using the OmniNetworking framework is shown in Listing 1. Source for a more complete TCP client can be found in the SchmoozingExamples project.

Writing A TCP Server

TCP servers can be characterized by the way they handle multiple connections and state. Servers that handle only one connection at a time are known as iterative servers. Concurrent servers accept and handle multiple connections in parallel. Both iterative and concurrent servers can implement stateless or stateful connections. A stateless server maintains no information from one connection request to the next. A request is received, processed, replied to, then forgotten. A stateful server, on the other hand, maintains

information associated with each client connection. With a stateful server, a connection between client and server would likely encompass several request and reply transactions.

OmniNetworking provides basic networking primitives useful to either iterative or concurrent servers, with or without state. Server implementations, especially state handling, tend to involve logic beyond the scope of OmniNetworking. I'll present the basic algorithms for implementing iterative and concurrent servers with OmniNetworking, leaving the issue of state as another exercise for the reader.

Iterative Servers

Comer describes the algorithm for an iterative TCP server [Comer93, p. 103] as follows:

1) Create a socket and bind it to a well known port number for the server being offered.
2) Establish a listener on the socket for incoming connections.
3) Accept the next connection request on the socket and obtain a new socket for the connection.
4) Repeatedly read the socket for a client request, generate a response, and send the reply back according to the application's protocol.
5) When finished handling a particular client request, close the connection and go back to step 3, accepting the next connection.

Implementing an iterative TCP server with OmniNetworking is straightforward, as Listing 2 illustrates. Refer to the SchmoozingExamples project for a more complete example.

Concurrent Servers

A concurrent server has two roles, a master and slave. The master is responsible for setting up a socket to listen for connections, accept them, and create slaves to process them. The slave's responsibility is to interact with the client connection, leaving the master free to continue accepting new connections.

Comer describes the algorithm for a concurrent TCP server [Comer93, p. 108]:

Master 1) Create a socket for the server and bind it to the well known port for the service being offered.
Master 2) Establishing a listener on the socket.
Master 3) Repeatedly accept incoming requests and create new slaves to handle the response.
Slave 1) Receive a connection request/socket upon creation.
Slave 2) Interact with the client using the socket, reading requests and sending back replies.
Slave 3) Close the connection and exit. The slave exits after handling all requests from one client.

To implement the concurrent processing of connections, the server can implement the slave role with multiple processes or multiple threads.

Concurrent Server Using Processes

The server's initial process implements the three steps of Comer server algorithm's Master role. It creates a socket bound to the port designated for the server being offered, and puts the socket into listening mode, and accepts incoming connections. When a new connection is accepted, a new process is created that will implement the three steps of the Slave role.

The slave process is created using the BSD system call *fork()*. *fork()* splits the original server process into two nearly identical processes. The calling process is referred to as the parent. The newly created process is known as the child. The two processes are duplicate copies of each other, except that each has a unique process identifier and different parent processes. The child process will implement the slave role, processing the client connection, while the parent continues as the master.

The return value of *fork()* is used by each process to determine which course of execution to follow from that point forward. *fork()* returns a zero value to the child process. To the parent process, *fork()* returns the process id of the child. Based on the return value of *fork()*, each process branches execution to the

implementation for their respective roles. The parent process cleans up from the fork and loops back to wait for the next incoming connection. The child process cleans up from the fork and continues on to process the accepted connection. Listing 3 illustrates a skeletal forking server using OmniNetworking.

With forking servers, the parent process must ensure that all forked child processes have been completely terminated. BSD uses the signal mechanism to notify the parent when any of its forked children have exited. To properly exit, a child process must issue the *exit()* system call. When this happens, BSD sends the SIGCHLD signal to the parent. The parent process can arrange to handle the SIGCHLD signal with a routine that ensures that the child processes are completely terminated[4]. This process is known, gruesomely, as "reaping" the children.

The following code can be used as a boilerplate reaping routine for handling the SIGCHLD signal.

```
void reaper() {
  pid_t reaped;
  int exitStatus;
  do {
    reaped = waitpid(-1, &exitStatus,
                WNOHANG | WUNTRACED);
  } while ( reaped > 0 );
  // Loop around until all child
  // processes have been reaped by
  // waitpid().
}
```

Listing 4. A common SIGCHLD signal handler for reaping forked processes.

Concurrent Servers Using Threads

The Cocoa-based server application's main thread implements the master role of Comer's algorithm. A new thread is created for each new connection that is accepted that will implement the slave role. The NSThread *detachNewThreadSelector:* method is used to create the new thread. *detachNewThreadSelector:* works by executing a designated method of some specified object in its own thread. Consequently, connection processing must be implemented in some method of an object in the server application. There are many ways to design a multithreaded server application to satisfy this requirement. I will describe an architecture based on the one implemented in the OmniFTPServer application [5].

This architecture encapsulates the behavior of the server and its client connection in their own classes. Listing 5 depicts the implementation for a connection handling class, *Connection*. Its *initWithConnectedSocket:* method receives the connection socket, implementing Slave Step 1. The *processConnection:* method encapsulates the communication between the server and client, corresponding to Slave Step 2. *processConnection:* would also implement Slave Step 3, closing the client connection after all client requests have been handled. Listing 6 presents a minimal implementation of a threaded concurrent TCP server using the *Connection* class. A more complete example is included in the SchmoozingExamples project.

Alternatives to OmniNetworking

OmniNetworking isn't the only game in town when it comes to writing Cocoa-based Internet applications. Here are some alternative approaches to consider.

BSD Sockets and NSFileHandle

With Mac OS X, the Foundation framework class NSFileHandle now provides direct support for reading and writing data from network sockets as well as files. NSFileHandle objects are tied into a Cocoa application's run loop, making it possible to easily accept network connections and read data asynchronously. NSFileHandle does not handle socket creation and host addressing, so one would continue to use the BSD sockets for this part of the programming.

CFSockets

CFSocket provides a C-based wrapper around BSD sockets using the pseudo object-oriented semantics of the CoreFoundation APIs. CFSocket's primary purpose is to provide a common substrate through which the Carbon and Cocoa frameworks can access underlying networking functionality. CFSockets can also used as sources for run loop events, which can

benefit the writing of multithreaded Cocoa network applications.

CFSockets is a somewhat tedious API, especially when compared to BSD sockets or ONTCPSocket, but the capability for run loop integration is very attractive. Ben Golding has written a subclass of ONTCPSocket that uses CFSocket to enable it as a source for run loop events. The source for his subclass, *TCPCFSocket*, was originally posted to the MacOSX-Dev mailing list on February 21, 2001 and is included, with permission, in the SchmoozingExamples project.

EDInternet

EDInternet is an Objective-C framework by Erik Dörnenburg for Internet programming featuring classes for manipulating MIME, mail, and news messages, as well as a wrapper around the BSD socket library. EDInternet's distinction is that it uses Foundation's NSFileHandle as a base for its socket class, giving it a hook into the Cocoa application run loop, thus enabling asynchronous network I/O. EDInternet is more modern in its design than OmniNetworking in that makes more use of classes provided in the Foundation framework, which did not exist when OmniNetworking was first released. EDInternet can be obtained from <http://www.mulle-kybernetik.com>.

Java Network Classes

Cocoa applications can also be written using Java, which provides its own object-oriented interface to TCP/IP networking via the java.net classes.

Conclusion

OmniNetworking provides a convenient Objective-C framework for programming TCP/IP network applications using Cocoa. It fills a gap in the Foundation framework for an object-oriented interface to Mac OS X's native BSD sockets layer. Networking functionality can be incorporated into a Cocoa application using two classes, ONHost and ONTCPSocket. OmniNetworking implements network I/O in terms of the common Foundation data classes, NSData and NSString, making it easy to use network data with other Foundation and AppKit objects.

The OmniNetworking framework can be downloaded from the Omni Group ftp site at `<ftp://ftp.omnigroup.com/pub/software/Source/>`. OmniNetworking is released to the Cocoa developer community for use under the Omni Source License[6].

Notes

[1] Also known as "dotted decimal". The dotted quad form is a translation of the 32 bit integer host address into four decimal parts, each corresponding to one byte in the address.

[2] The Mac OS X implementation of BSD sockets uses the range from 49152 and 65535 for dynamic port assignments.

[3] For example, HTTP servers listen for TCP connections on the well-known port number 80. The official list of registered port numbers for Internet services is maintained by IANA, the Internet Assigned Numbers Authority `<http://www.iana.org>`. Browsing through the /etc/services file will also reveal a list of services and their well-known port numbers.

[4] An incompletely terminated process is known as a zombie. It has become orphaned from its parent, whose responsibility it was to ensure that the process exits completely. Zombie processes remain on the system until the next reboot.

[5] OmniFTPServer, also from the Omni Group, implements an ftp server as a multithreaded Cocoa application using their OmniNetworking framework. It features an elegant design wherein the ftp server's functionality is implemented with a delegate object via an informal protocol, and the server and connections are modeled with their own classes. OmniFTPServer can be downloaded from the Omni Group's ftp site.

[6] The Omni Source License can be viewed at `<http://www.omnigroup.com/community/developer/sourcecode/sourcelicense/>`. It can be characterized as much closer to the BSD license than the GNU GPL.

Bibliography

[Stevens90] Unix Network Programming. W. Richard Stevens. Prentice-Hall, Inc., Englewood Cliffs, NJ. 1990.

[Comer91] Internetworking with TCP/IP: Principles, Protocols, and Architecture. Douglas E. Comer. Prentice-Hall, Inc., Englewood Cliffs, NJ. 1991.

[Comer93] Internetworking with TCP/IP: Client-Server Programming and Applications. Volume 3. Douglas E. Comer & David L. Stevens. Prentice-Hall, Inc., Englewood Cliffs, NJ. 1993.

```
// TCP Client Using OmniNetworking

NSAutoreleasePool *mainPool;
ONHost *serverHost;
ONTCPSocket *connectionSocket;
NSString *serverHostname = @"www.machack.com";
unsigned short serverPort = 80;

mainPool = [[NSAutoreleasePool alloc] init];

serverHost = [ONHost hostForHostname:serverHostname];
// Step 1. Get the Internet address of the server.

connectionSocket = [ONTCPSocket tcpSocket];
// Step 2. Allocate a socket.

[connectionSocket setLocalPortNumber];
// Step 3. Configure the socket to use an arbitrary local port number.

[connectionSocket connectToHost:serverHost port:serverPort];
// Step 4. Connect to the server.


// Step 5. Communicate with server using some application protocol.

NSString *httpRequest = @"GET / HTTP 1.0\r\n\r\n";
NSMutableString *httpReply = [NSMutableString string];
NSString *someString;

[connectionSocket writeString: httpRequest];
// Send the HTTP request to the server.

[httpReply appendString: [connectionSocket readString]];
// readString: will read up to 2048 bytes from the socket.

while ( [connectionSocket isReadable] )
      [httpReply appendString: [connectionSocket readString]];
// Read any remaining bytes from the socket to get a complete reply.

NSLog(@"%@", httpReply);

[mainPool release];
// Step 6. The connection socket is closed when the pool is released.
```

Listing 1. TCP client using OmniNetworking

```
// An Iterative TCP Server Using OmniNetworking

unsigned short serverPort = 1701;
ONTCPSocket *serverSocket;
NSAutoreleasePool *mainPool;

mainPool = [[NSAutoreleasePool alloc] init];

serverSocket = [ONTCPSocket tcpSocket];
// Step 1. Allocate a socket.

[serverSocket startListeningOnLocalPort: serverPort allowingAddressReuse:YES];
// Step 2. Start listening for connections on the port.

do {
      ONTCPSocket *connectionSocket;
      NSMutableData *clientData;
      NSString *clientRequest;
      NSAutoreleasePool *loopPool;

      loopPool = [[NSAutoreleasePool alloc] init];

      connectionSocket = [serverSocket acceptConnectionOnNewSocket];
      // Step 3. Accept the next connection.

      clientData = [NSMutableData data];
      [clientData appendData: [connectionSocket readData]];
      while ( [connectionSocket isReadable] )
            [clientData appendData: [connectionSocket readData]];

      clientRequest = [[[NSString alloc] initWithData:clientData
                              encoding:NSASCIIStringEncoding] autorelease];
      [connectionSocket writeFormat:@"You sent me: %@", clientRequest];
      [connectionSocket writeFormat:@"I'm sending you this: %@",
                              [clientRequest uppercaseString]];
      // Step 4. Interact with the client.

      [loopPool release];
      // Step 5. Close the connection. When the pool is release, the
      // connection socket will be gracefully closed.

} while (1);
// Loop forever listening for new connections

[mainPool release];
```

Listing 2. An iterative server using OmniNetworking

```
// A Forking Concurrent TCP Server Using OmniNetworking

void myReaper();
ONTCPSocket *serverSocket;
unsigned short serverPort = 1701;
NSAutoreleasePool *mainPool;

signal( SIGCHLD, myReaper  );
// Install myReaper() as the SIGCHLD signal handler.  You'll need to define
// a reaper routine, such as the one defined in the text, to ensure proper
// disposal of all created child processes.

mainPool = [[NSAutoreleasePool alloc] init];

serverSocket = [ONTCPSocket tcpSocket];
// Master Step 1. Allocate a socket.

[serverSocket startListeningOnLocalPort: serverPort];
// Master Step 2. Listen for connections on the bound server port.

do {
      ONTCPSocket *connectionSocket;
      pid_t forkedPID;

      NSAutoreleasePool *loopPool = [[NSAutoreleasePool alloc] init];

      connectionSocket = [serverSocket acceptConnectionOnNewSocket];
      // Master Step 3.

      forkedPID = fork();
      // Fork the server process into two processes.  One will continue
      // to act as the master.  The other will assume the role of the
      // slave, processing the connection.

      if ( forkedPID == 0 ) {
            // The executing process is the forked child process.
            NSMutableString *clientRequest;
            NSString *serverReply;

            // Slave Step 1 is implicitly satisfied.  We have the connection
            // socket and can just start using it.

            [serverSocket abortSocket];
            // Clean up from the fork by closing the child's copy of the
            // listening socket.

            clientRequest = [NSMutableString string];
            [clientRequest appendString: [connectionSocket readString];
            while ( [connectionSocket isReadable] )
                  [clientRequest appendString: [connectionSocket readString];

            serverReply = [clientRequest uppercaseString];
```

```
            [stream writeFormat:@"You sent me this: %@\n", clientRequest];
            [stream writeFormat:@"I'm sending you this: %@\n", serverReply];
            // Slave Step 2. Interact with the client.

            [loopPool release];
            // Slave Step 3. The connection socket is closed when the pool is
            // released.

            break;
            // Break out of the loop so the child process can exit.
      }
      else if ( forkedPID > 0 ) {
            // The executing process is the parent.

            [loopPool release];
            // The accepted connection socket is closed with the pool's release,
            // performing all the fork cleanup that needs to be done on the
            // parent process.
      }

} while (1);

[mainPool release];
```

Listing 3. A forking concurrent server using OmniNetworking.

```
// The Connection class implementation

#import "Connection.h"
@implementation Connection

- initWithConnectedSocket: (ONTCPSocket*) aSocket
{
      // Slave Step 1. Receive the accepted socket
      self = [super init];
      if ( self ) {
            if ( [aSocket isConnected] )
                  mySocket = [aSocket retain];
            else
                  return nil;
      }
      return self;
}

- (void) processConnection
{
      ONSocketStream *stream;
      NSString *inFromClient;

      NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

      stream = [ONSocketStream streamWithSocket: mySocket];

      inFromClient = [stream readLine];
      [stream writeFormat:@"%@: %@ received connection from %@\r\n",
                  [NSCalendarDate calendarDate],
                  [ONHost localHostname],
                  [[mySocket remoteAddressHost] hostname]];
      [stream writeFormat:@"%@\r\n", [inFromClient uppercaseString]];
      // Slave Step 2. Interact with client.

      [mySocket abortSocket];
      // Slave Step 3. Close the connection.  This server is pretty draconian.

      [pool release];
}


- (void) dealloc
{
    [mySocket release];
    [super dealloc];
}
@end
```

Listing 5. A class encapsulating a server's client connection and its processing.

```
// A Threaded Concurrent TCP Server Using OmniNetworking

#import “Connection.h”

ONTCPSocket *serverSocket;
unsigned short serverPort = 1701;
NSAutoreleasePool *mainPool;

mainPool = [[NSAutoreleasePool alloc] init];

serverSocket = [ONTCPSocket tcpSocket];
// Master Step 1. Allocate a socket

[serverSocket startListeningOnLocalPort: serverPort];
// Master Step 2. Establish a listener


while (1) {
      NSAutoreleasePool *loopPool;
      Connection *client;
      ONTCPSocket *connectionSocket;

      loopPool = [[NSAutoreleasePool alloc] init];

      connectionSocket = [serverSocket acceptConnectionOnNewSocket];
      // Master Step 3. Accept new connections

      client = [[Connection alloc] initWithSocket:connectionSocket];
      [client autorelease];
      // Slave Step 1. Receive the connection’s socket in our connection
      // handling object.

      [NSThread detachNewThreadSelector:@selector(processConnection)
                  toTarget:client withObject:nil];
      // Slave Step 2. Interact with the client, in a separate thread,
      // by way of Connection’s processConnection: method.
      // Slave Step 3. Closing the connection, is handled in the spawned thread.

      [loopPool release];
}

[mainPool release];
```

Listing 6. A threaded concurrent server using OmniNetworking