Sockets programming in Ruby

Explore Ruby's fundamental sockets interfaces for networking applications

Skill Level: Intermediate

M. Tim Jones (mtj@mtjones.com)
Senior Principal Software Engineer
Emulex Corp.

11 Oct 2005

This tutorial shows how to develop sockets-based networking applications using the Ruby language. The author presents some Ruby basics and walks through the most important classes for sockets programming, followed by working chat application that illustrates these fundamentals. The tutorial finishes by exploring the higher-level classes that make it easy to build dynamic Web servers, mail servers and clients, and other application-layer protocols.

Section 1. Before you start

About this tutorial

Ruby is an object-oriented scripting language that is simple, elegant, and dynamic. Ruby originated in Japan, but it's now gaining popularity in the U.S. over traditional scripting languages such as Python and Perl.

This tutorial demonstrates how to use the Ruby language, with emphasis on network programming. It explores the fundamental socket interfaces for Ruby, the higher-level classes that make it easy to build dynamic Web servers and mail servers and clients, and other application-layer protocols.

Objectives

In this tutorial, you put Ruby to the test by constructing a chat server application that

can serve a large and scalable number of clients. By the end of this tutorial, you will know how to use the standard Sockets class to build both client and server applications, as well as how to use classes that simplify sockets programming, including TCPSocket and TCPServer. You will also have tried your hand at building a working, scalable chat server, and you will be familiar with several classes for manipulating application-layer protocols, including HTTP, SMTP, and POP3.

Prerequisites

You should have a basic knowledge of Ruby and basic familiarity with BSD-style sockets.

System requirements

To run the examples in this tutorial, you need version 1.8 of Ruby, which is available from the Ruby Web site (see Resources for a link). To build the Ruby interpreter, you need the GNU C compiler (gcc) and configure/make utilities, which are part of any standard GNU/Linux distribution.

Section 2. Introducing Ruby

For those new to Ruby, this section offers a brief overview.

What is Ruby?

Ruby's syntax and construction borrow from a number of languages, such as Eiffel, Python, and Smalltalk-80. Because its syntax is similar to these (as well as C), it's easy to learn and is intuitive in its form.

Ruby is an interpreted language, which makes it perfect to quickly try out ideas. Ruby is also a dynamic language, so, like Scheme, you can construct programs within a Ruby program and then execute them.

Ruby is object oriented and includes a large number of classes that can simplify application development. Some interesting classes deal with such functionality as networking, XML, SOAP, multithreaded programming, and database integration. Ruby also includes exception handling as part of the language.

But enough with the hand waving. Let's look at some examples of Ruby in action.

A taste of Ruby

Although you can use Ruby to write large programs, part of Ruby's strength as an interpreted scripting language is the ability it gives you to quickly try things out. Let's look at a few examples to get a better feel for the Ruby language:

Listing 1. Some Ruby samples to try

```
# Open a socket to a local daytime server, read, and print.
require 'socket'
print TCPSocket.open("192.168.1.1", "daytime").read


# Create an array, iterate through each element, and multiply by 2
#    (creating a new array as a result, [2, 4, 6, 8, 10]).
arr = [1, 2, 3, 4, 5]
arr.map{|x| x*2}


# Read text from standard-in, and print out in reverse character
#    order.
print STDIN.read.reverse


# Get the method for * as applied to 2 (an object). Calling this new
#    method with a new value results in 2*x (here the result is 20).
double = 2.method(:*)
double.call(10)


# Create a class, inherit it by another, and then instantiate it and
#    invoke its methods.
class Simple
  def initialize(name)
    @name = name
  end
  def hello
    printf("%s says hi.\n", @name)
  end
end

class Simple2 < Simple
  def goodbye
    printf("%s says goodbye.\n", @name)
  end
end

me = Simple2.new("Tim")
me.hello
me.goodbye
```

Why use Ruby?

Since Ruby is an object-oriented interpreted scripting language like Python and Perl, why should you invest your time in learning a new language? It comes down to a matter of taste, but Ruby does offer advantages over Perl and Python.

First, Ruby's syntax is easy to understand and learn and is highly readable. The language has a large and intuitive class library.

For the pedantic reader, Ruby is a pure object-oriented language. Everything in the language is an object, as is the case with Smalltalk. In fact, Ruby was designed from the ground up as an object-oriented language; it supports singleton methods,

multiple inheritance, overloading, and exception handling.

From a maintenance standpoint, Ruby has an advantage -- it's easy to read and understand and, therefore, easy to maintain even if you weren't an application's original author.

Now, let's dig into sockets programming in Ruby.

Section 3. Ruby sockets classes

Ruby socket class hierarchy

Ruby has a rich and diverse set of sockets classes. These classes range from the standard sockets class (which mimics the BSD Sockets API) to more refined classes that focus on a particular protocol or application type. The sockets class hierarchy is shown in Figure 1.

Figure 1. Class hierarchy of the base sockets classes

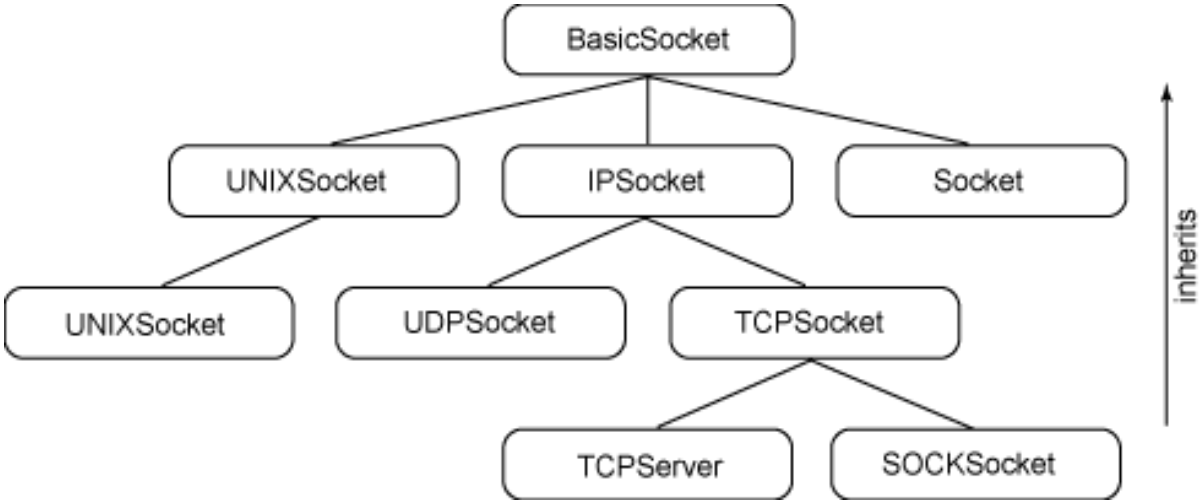


Table 1 provides a short description for each of the classes. This section touches on the important classes to help you understand what they are and the methods they provide. The next section demonstrates how you can use a few of the classes to create, destroy, and communicate with Ruby sockets.

Table 1. Ruby base sockets classes

Class	Description
`BasicSocket`	Abstract base class for all socket classes
`UNIXSocket`	Class providing IPC using the UNIX domain

	protocol (AF_UNIX)
`UNIXServer`	Helper class for building UNIX domain protocol socket servers
`IPSocket`	Base class for protocols using the Internet Protocol (AF_INET)
`UDPSocket`	Class for User Datagram Protocol (UDP) sockets
`TCPSocket`	Class for Transmission Control Protocol (TCP) sockets
`TCPServer`	Helper class for building TCP socket servers
`SOCKSSocket`	Helper class for building SOCKS-based sockets applications
`Socket`	Base socket class that mimics that BSD Sockets API

This tutorial omits the lesser-used classes and focuses on the classes you'll find the most useful. The Resources section provides more information about these omitted classes.

Socket class (class methods)

The `Socket` class is fundamentally the BSD Sockets API in class form. In this class, you can find standard functions such as `accept`, `bind`, `listen`, `connect`, and others. Table 2 lists the `Socket`class's class methods and Table 3 lists the instance methods. (An instance method requires an instance object of the class, whereas a class method does not.) Due to the number of methods available in this class, these tables show a representative, but not exhaustive, list.

Table 2. Class methods for the Socket class

Class method	Description
`Socket::for_fd(fd)`	Returns a `Socket` object for the passed file descriptor
`Socket::gethostbyaddr(addr[, type])`	Resolves the IP address to a fully qualified domain name
`Socket::gethostbyname(name)`	Resolves the host name to an IP address
`Socket::gethostname`	Returns a string representing the hostname
`Socket::new(domain, type, proto)`	Creates a new socket

Socket class (instance methods)

Table 3. Instance methods for the Socket class

Instance method	Description

`sock.bind(addr)`	Binds the socket to the packed address string
`sock.listen(backlog)`	Places the socket into the listening state, with a number of outstanding requests set to `backlog`
`sock.accept`	Accepts a new connection (returns a new socket)
`sock.connect(addr)`	Connects the socket to the host defined by the packed address string
`sock.recvfrom(len[, flags])`	Receives data (up to `len` in size) from the socket, and returns a string containing the data and peer address

Note that an instance of class `Socket` inherits the methods of class `BasicSocket`. Therefore, an instance of `Socket` can invoke such methods as `setsockopt` (part of the `BasicSocket` class).

UDPSocket class

The `UDPSocket` class implements the connectionless User Datagram Protocol. UDP is an unreliable protocol, but it is important and can be found in the Domain Name Service (DNS) protocol, the Dynamic Host Configuration Protocol (DHCP), the Service Location Protocol (SLP), the Network Time Protocol (NTP), and the Trivial File Transport Protocol (TFTP).

UDP's lack of reliability makes it fast (because there's no congestion control and retry); it's perfect for protocols that can easily retransmit after some timeout period.

Table 4 shows the `UDPSocket`class's class method and Table 5 lists the instance methods.

Table 4. Class method for the UDPSocket class

Class method	Description
`UDPSocket::new`	Creates a new `UDPSocket` object

Table 5. Instance methods for the UDPSocket class

Instance method	Description
`sock.bind(host, port)`	Binds the UDP socket instance to the port on the host
`sock.connect(host, port)`	Connects the UDP socket to the port on the host
`sock.send(mesg, flags[, dest])`	Sends data (`mesg`) on the UDP socket to the destination

The `send` method has a couple of variations, as you'll see in the next section.

TCPSocket and TCPServer classes

The `TCPSocket` class supports sockets for the connection-based, reliable Transmission Control Protocol. A helper class for the creation of TCP server sockets is also available in the `TCPServer` class. Table 6 lists the class methods for both the `TCPSocket` and `TCPServer` classes, and Table 7 shows the `TCPSocket` and `TCPServer` instance methods.

Table 6. Class methods for the TCPSocket and TCPServer classes

Class method	Description
`TCPSocket::new(host, service)`	Creates a new `TCPSocket` object
`TCPServer::new(host, service)`	Creates a new `TCPServer` object

Table 7. Instance methods for the TCPSocket and TCPServer classes

Instance method	Description
`tcpsock.gethostbyname(host)`	Returns an array containing the canonical name, aliases, protocol domain, and dotted IP address for the resolved hostname
`tcpsrvsock.accept`	Awaits a client connection, and returns a new `TCPSocket` object

Let's see how you can use these APIs to develop networking applications for clients and servers that are both connection-based (TCP) and connectionless (UDP).

Section 4. Sockets programming in Ruby

Preliminaries

This section demonstrates the Ruby networking APIs. You can see this demonstration interactively using Ruby's `irb`, an interactive Ruby interpreter in which you can command Ruby a line at a time and see the result of each command. Consider the following example, which uses the `IPSocket` class's `getaddress` method to resolve a hostname to a quad-dotted IP address (in string format):

Listing 2. Using irb to demonstrate getaddress

```
[mtj@plato ~]$ irb
irb(main):001:0> require 'socket'
```

```
=> true
irb(main):002:0> IPSocket::getaddress('www.ibm.com')
=> "129.42.19.99"
irb(main):003:0>
```

After starting `irb`, you load the socket library with the `require` command. This command returns "true," indicating that the library was successfully loaded. Next, use the `getaddress` method of the `IPSocket` class to resolve the hostname `www.ibm.com` to an IP address. You specify no target for the command (the statement has no `lval`), but you can see the result as `irb` prints the result of each command invoked.

In addition to `irb`, the Ruby interpreter is invoked with `ruby`. You can debug Ruby programs with the command

```
ruby -r debug prog.rb
```

This tutorial focuses on `irb` and the `ruby` interpreter. The Resources section links to other sources of Ruby tools.

Creating and destroying sockets

The concept of a socket is universal among scripting languages such as Ruby and compiled languages like C. The primary differences are the helper classes and methods that Ruby provides.

Let's start with the construction of a socket in Ruby. You can create sockets a number of ways; you'll begin with the standard method that's most familiar to C programmers. The following two examples create a stream socket (TCP) and a datagram socket (UDP).

Listing 3. Creating stream and datagram sockets

```
myStreamSock = Socket::new( Socket::AF_INET, Socket::SOCK_STREAM, 0 )
myDatagramSock = Socket::new( Socket::AF_INET, Socket::SOCK_DGRAM, 0 )
```

Either of these sockets can be used for client or server applications (which we'll explore in the server and client sockets sections). But Ruby provides a number of other ways to create sockets.

To create a stream socket and connect it to a server, the `TCPSocket` class can be used. The following (identical) examples create a stream socket and then connect it to the HTTP server on host www.ibm.com:

Listing 4. Connecting a stream socket to an HTTP server

```
webSocket = TCPSocket::new( "www.ibm.com", 80 )
```

```
webSocket = TCPSocket::new( "www.ibm.com", "http" )
```

When the class method completes, the `webSocket` contains a socket that is connected to the host and port as defined.

You can create a stream server socket using the `TCPServer` class. To this method, you provide the address to which you'll bind the port. In most cases, you bind to the `INADDR_ANY` ("0.0.0.0"), which lets you accept connections from any of the available interfaces on the host. This can be done in the following ways (binding to port 23000):

Listing 5. Creating a stream server socket

```
servSock = TCPServer::new( "0.0.0.0", 23000 )
servSock = TCPServer::new( "", 23000 )
```

Finally, you can close a socket using the `close` method. This method is inherited from the `IO` class, but it's available for each of the socket types:

Listing 6. Closing a socket

```
myStreamSock.close
servSock::close
```

Socket addresses in Ruby

One of the interesting advantages of Ruby over C is that an IP address can be expressed either as a quad-dotted string or as a hostname that is automatically resolved for you. For example, the following two are equivalent:

Listing 7. Two ways of addressing

```
mySocket = TCPSocket.new( 'www.ibm.com', 80 )
mySocket = TCPSocket.new( '129.42.16.99', 80 )
```

You can resolve a hostname to an IP address using the `getaddress` method of the `IPSocket` class:

Listing 8. Resolving a hostname to an IP address

```
strIPAddress = IPSocket::getaddress('www.ibm.com')
```

Ruby also supports the C `sockaddr` style of addresses. But to use them you need

to pack them into a binary string. The following example illustrates this process:

Listing 9. Using sockaddr-style addresses

```
sockaddr = [Socket::AF_INET, 80, 127, 0, 0, 1, 0, 0].pack("snCCCCNN")
```

This example creates an IP-based address (AF_INET) for port 80 and the local address "127.0.0.1". The last two zeros shown in the array are there for packing purposes. The `pack` method is applied to the array which packs it into a binary string. The string template is defined as `s` for short (16 bits), `n` for big-endian short (the port number), `C` for unsigned char (the four elements of the IP address), and `N` for big-endian longs. The total length of the `sockaddr` is then 16 bytes (which you can determine in Ruby using `sockaddr.length`). The following section looks at these methods in use.

Stream and datagram server sockets

When you create a server socket, you're typically making a service available to clients. To do this, you create a socket and bind it to one or all of the interfaces that are on the host. Ruby provides a few ways to do this. This section shows first the traditional way this is done for both stream and datagram sockets and then how Ruby helps make the process easier.

The traditional form follows the C model in which you create a socket and then bind it to an address. This applies to both stream (TCP) and datagram (UDP) sockets, as demonstrated here:

Listing 10. Creating a server socket the traditional way

```
streamSock = Socket::new(Socket::AF_INET, Socket::SOCK_STREAM, 0)

# Address "0.0.0.0", port 23000
myaddr = [Socket::AF_INET, 23000, 0, 0, 0, 0, 0, 0).pack("snCCCCNN")

streamSock.bind( myaddr )

streamSock.listen(5)
```

Note that this example also calls the `listen` method, which makes the socket available for client connections (places it in the listening state).

But Ruby provides a simpler way. The `TCPServer` class provides an easy way to create a TCP server-side socket. The following example is synonymous with the previous socket-creation example (`""` means `INADDR_ANY`, or bind to all host interfaces):

Listing 11. Creating a server socket the Ruby way

```
streamSock = TCPServer::new( "", 23000 )
```

Note that the `bind` and `listen` are done for you as part of the `TCPServer` socket creation.

Binding an address to a datagram socket is also simple using a less complicated `bind` method. This example binds to all local interfaces and port 23000:

Listing 12. Binding a datagram socket

```
dgramSock = UDPSocket::new
dgramSock.bind( "", 23000 )
```

That's all there is to creating server sockets. Let's now look at the client side of the socket.

Stream and datagram client sockets

The methods for creating and binding server sockets are similar to those used for client sockets. As with server sockets, this section discusses the standard mechanism first and then looks at Ruby methods that make client sockets simpler.

The first method will be familiar if you've used the BSD Sockets API. In this mode, you create a socket and an address structure (identifying the target host and port number) and then connect to the peer using the `connect` method. This example connects to port 25 on the host 192.168.1.1:

Listing 13. Connecting to a host the traditional way

```
streamSock = Socket::new(Socket::AF_INET, Socket::SOCK_STREAM, 0)
# Address 192.168.1.1, port 25
myaddr = [Socket::AF_INET, 25, 192, 168, 1, 1, 0, 0).pack("snCCCCNN")
streamSock.connect( myaddr )
```

Ruby provides a simpler set of methods for connecting to a peer by not requiring an address structure to be created. Instead, the `connect` methods can reference the hostname and port number directly. The following example illustrates connecting to a peer socket for the stream and datagram sockets:

Listing 14. Connecting to a host the Ruby way

```
dgramSock = UDPSocket::new
dgramSock.connect( "192.168.1.2", 23000 )
streamSock = TCPSocket::new( "192.168.1.2", 23000 )
```

This is much simpler than the previous `Socket` class mechanism.

You need to be aware of an important aspect of the `connect` method for the `UDPSocket` class. Because UDP is a connectionless protocol, no connection exists for the datagram socket. Instead, the address provided here is cached with the socket; when sends are performed through the socket, this address is the target for those sends. A datagram socket can call `connect` at any time to respecify the target for communication.

Socket input and output

Let's now look at how I/O is performed with stream and datagram sockets. The socket classes provide a number of methods (`send`, `recv`, `recvfrom`), but you can also use the `IO` class methods such as `read` and `write`, `puts` and `gets`, and `print`. This section demonstrates classes for both stream and datagram sockets.

The first example opens a new stream socket client, sends a short message, and then prints the response:

Listing 15. Stream socket I/O

```
streamSock = TCPSocket::new( "192.168.1.2", 23000 )
streamSock.send( "Hello\n" )
str = streamSock.recv( 100 )
print str
streamSock.close
```

For connectionless datagram sockets, you can specify the destination address with the `send` call. The `recvfrom` call is also unique in that, in addition to the data being read, the source address from which the data came is also returned. Both of these concepts are demonstrated here:

Listing 16. Datagram socket I/O

```
dgramSock = UDPSocket::new
dgramSock.bind( "", 23000 )
reply, from = dgramSock.recvfrom( 100, 0 )
dgramSock.send( "Hello\n", from[2], from[1] )
```

When you create a new UDP socket, you bind to all interfaces (`INADDR_ANY`) and port 23000. You then wait for a message from a peer which upon receipt returns the message (`reply`) and also the source of the message (`from`). The `from` array contains the protocol domain (`AF_INET`), the port number, the hostname, and the IP address. You use the hostname (`from[2]`) and port number (`from[1]`) here to return the message to the originator.

Socket options

Socket options allow you to modify the way a socket works (as well as tweak the TCP and IP layers for the socket). Using socket options, you can resize the I/O buffers, join a multicast group, or send broadcast messages (to name just a few possibilities).

This section demonstrates using the `setsockopt` and `getsockopt` methods to give you an idea how options are used in Ruby.

To read or write socket options, you need to manipulate binary strings. This is simple given the `pack` and `unpack` methods of the `Array` class. The following example passes an integer (`i`) to `setsockopt` to enable the SO_REUSEADDR socket option (and then reads it back with `getsockopt`):

Listing 17. Demonstrating socket options

```
opt = [1].pack("i")
streamSock.setsockopt( Socket::SOL_SOCKET, Socket::SO_REUSEADDR, opt )

newopt = streamSock.getsockopt( Socket::SOL_SOCKET, Socket::SO_REUSEADDR )
print newopt.unpack("i")
```

Ruby supports the socket options that are available from the underlying operating system. See the Resources section for more information.

Asynchronous I/O

The final aspect of Ruby sockets that I'll explore before digging into an application is *asynchronous I/O*, I/O that is nonblocking in nature and occurs asynchronously to the flow of the application. An application requests that an event be generated when activity occurs on a file descriptor. You can request this for a large number of descriptors, including the specific types of events that you're interested in (such as read data available, write data possible, and exception).

The `select` method (of the `IO` class) is the most common method of achieving asynchronous I/O. You provide arrays for `select` identifying the `IO` objects of interest for read, write, and exception. You can also provide a timeout value (in seconds) to force `select` to return if no event has occurred during that time. The `select` return is a three-element array of `IO` objects (one array for read, one for write, and one for exception). Any non-zero array indicates that an object is available for that type of operation. Consider the following code segment:

Listing 18. Asynchronous I/O via the select method

```
select( [STDIN], nil, nil )
```

This example instructs Ruby to return when read data is available on standard-in. The fourth argument (`timeout`) is absent, which indicates that no timeout should be performed. When the return from `select` is for timeout, a `nil` is returned.

Otherwise, an array of three arrays is returned. A `nil` for any of these arrays indicates that no event of this type occurred.

Let's look at one more example that uses an `IO` object:

Listing 19. Another look at select

```
strmSock1 = TCPSocket::new( "192.168.1.1", 100 )
strmSock2 = TCPSocket::new( "192.168.1.1", 101 )

# Block until one or more events are received
result = select( [strmSock1, strmSock2, STDIN], nil, nil )

for inp in result[0]

  if inp == strmSock1 then
    # data avail on strmSock1
  elsif inp == strmSock2 then
    # data avail on strmSock2
  elsif inp == STDIN
    # data avail on STDIN
  end

end
```

The application begins by creating two `TCPSocket` objects and connecting them to a peer device. You then call `select` with an array for the read event. The array contains the two `TCPSocket` objects just created and the `STDIN` IO object. The write and exception arrays are passed as nil, meaning you aren't interested in them.

Once the `select` call returns, an event has occurred. An array containing three arrays is returned, indicating which of the `IO` objects need to be investigated. You create a `for` loop to walk through the first array in result (the read events) and check to see which object is returned. When you find an object match, you'd perform a read on the object, but here you just indicate what has occurred.

If you were interested in the write events from `select`, you'd test `result[1]`; if you were interested in exceptions, you'd check `result[2]`.

This is a powerful mechanism for dealing with lots of `IO` objects asynchronously. It's so important that it's demonstrated again in the sample chat server in the next section.

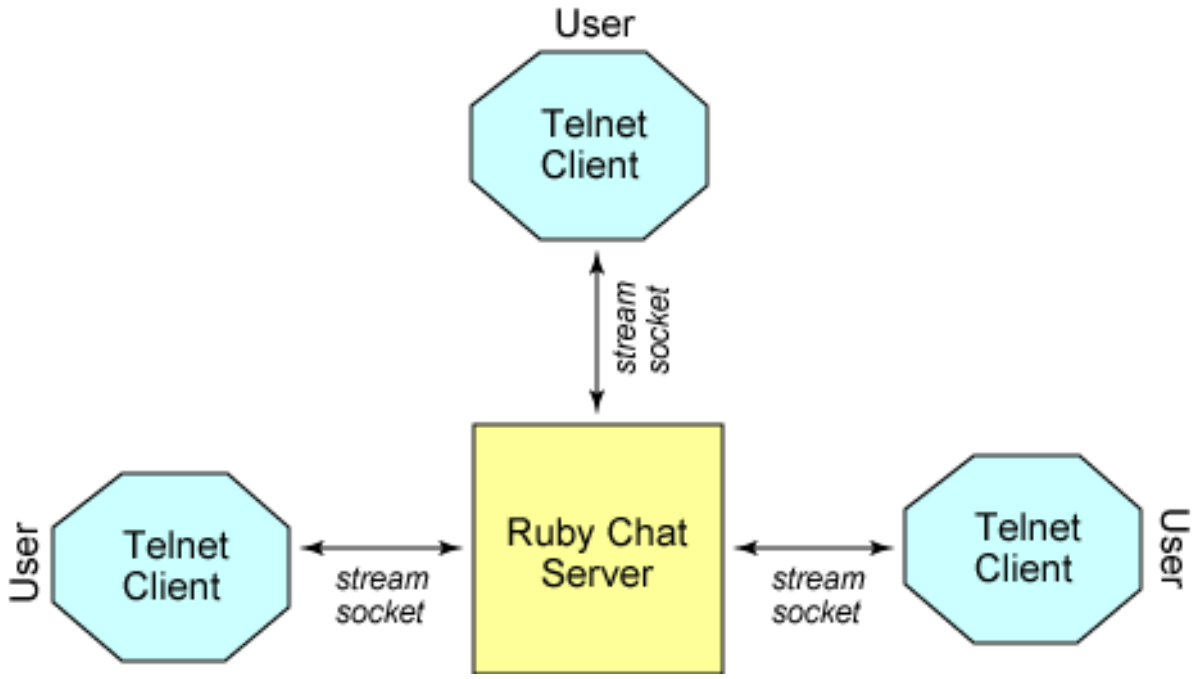
Section 5. Building a Ruby chat server

The goal

Let's put this knowledge to use. In this section, you'll build a simple chat server. Clients can access the chat server through a telnet client. This allows clients to type

messages to the server and also receive messages from the server and other clients (see Figure 2).

Figure 2. Conceptual architecture of the simple Ruby chat server



An overriding requirement of this server is that you want it to be scalable so that you can support an arbitrary number of client connections. All communication is broadcast to all connected clients, so when a client types a message, it's seen by everyone. Finally, you'll use stream (TCP) sockets for communication.

For simplicity and scalability, you'll use an aspect of the `select` method that lets you mix client sockets and the server socket. The `select` method will determine when a client has something to say (a read event from the server's perspective). But you'll also use the `select` method to identify when a new client is connecting. Both are classified as *read* events, so you'll note when the server socket has an event and use the `accept` method for the incoming client connection.

Let's dig into the source to see how Ruby simplifies the development of this application.

ChatServer class

Let's begin by looking at the skeleton class for the `ChatServer`. Listing 20 shows the class and `initialize` method (called when you create a new `ChatServer` instance), the instance variables, and the other methods that make up the server.

Listing 20. ChatServer class and initialize method

```
require "socket"

class ChatServer

  def initialize( port )
    @descriptors = Array::new
    @serverSocket = TCPServer.new( "", port )
    @serverSocket.setsockopt( Socket::SOL_SOCKET, Socket::SO_REUSEADDR, 1 )
    printf("Chatserver started on port %d\n", port)
    @descriptors.push( @serverSocket )
  end # initialize

  def run
  end

  private

  def broadcast_string( str, omit_sock )
  end

  def accept_new_connection
  end # accept_new_connection

end #server
```

This code works as follows:

1. Because you're using sockets classes in the server, you import the `socket` class.
2. You declare the `ChatServer` class and the methods that are available. These are split into two sections:
   - The public methods (`initialize` and `run`)
   - Private methods that aren't visible outside of the class (`broadcast_string` and `accept_new_connection`)
3. The `initialize` method is invoked when a new `ChatServer` is instantiated (via `ChatServer::new(port)`). It works as follows:
   - The caller provides a port number to which you bind your server.
   - For initialization, you create a number of instance variables (which are local to this instantiation of the class).
   - The `descriptors` array keeps track of the sockets that exist for the server.
   - The `serverSocket` is the server socket of type `TCPServer`; it allows connections from any interface and the caller's defined port. To reuse the address (for rapid restarts of the server), you enable the SO_REUSEADDR socket option.
   - You emit a message to `stdout` indicating that the server has started, and you add the server's socket to the `descriptors` array using the `push` method.

The `run` method implements the chat functionality. The private functions are used internally for sending a message to all clients and accepting a new client connection.

The run method

The `run` method is the server code for the chat server (see Listing 21).

Listing 21. The run method

```
def run

  while 1

    res = select( @descriptors, nil, nil, nil )

    if res != nil then

      # Iterate through the tagged read descriptors
      for sock in res[0]

        # Received a connect to the server (listening) socket
        if sock == @serverSocket then
          accept_new_connection
        else

          # Received something on a client socket
          if sock.eof? then
            str = sprintf("Client left %s:%s\n",
                          sock.peeraddr[2], sock.peeraddr[1])
            broadcast_string( str, sock )
            sock.close
            @descriptors.delete(sock)
          else
            str = sprintf("[%s|%s]: %s",
                          sock.peeraddr[2], sock.peeraddr[1], sock.gets())
            broadcast_string( str, sock )
          end

        end

      end
    end
  end

end #run
```

When the `run` method is invoked, it proceeds as follows:

1. The method enters an infinite loop.
2. The method blocks on the `select` method, awaiting a read event (note that descriptors are passed as the read `IO` object array). When the server starts, the only descriptor present is the server socket. Therefore, you're waiting for a client to connect.
3. When `select` returns, you check that a `nil` wasn't returned (if it was, you ignore it and wait again at `select`). If an array was returned, you loop through the elements of the first array (`res[0]`) which represent descriptors for read events. Inside the loop, you check to see if one of three things happened for each descriptor:
   - Whether the descriptor is for the server socket; and, if so, whether a

new client is attempting to connect. For this event, you call `accept_new_connection` to connect to the client (more on this function under "Helper methods," below).

- Whether the socket is at the end-of-file. For this test, you use the `eof` method; in other words, you check whether the client has disconnected. If the peer has disconnected, you generate a message for the client and broadcast it to all other clients that are connected (using the `broadcast_string` helper method). You then close the server's end of the socket and remove it from the descriptors array using the `delete` method.
- If the socket wasn't the server socket and it wasn't disconnected, then the read event indicates that data is available to be read. In the `else` portion of the end-of-file check, you read in the data from the socket using the `gets` method and format it into a string (using the source socket's peer information) using `sprintf`. This string is then broadcast to the connected clients with the `broadcast_string` method.

Helper methods

Let's now look at the helper methods for the chat server. There are two; their jobs are to accept a new client connection for the server and to send messages to the connected clients.

The first method is `accept_new_connection`, which is used to bring another client into the chat as follows:

Listing 22. Bringing a new client into the chat using accept_new_connection

```
def accept_new_connection

  newsock = @serverSocket.accept
  @descriptors.push( newsock )

  newsock.write("You're connected to the Ruby chatserver\n")

  str = sprintf("Client joined %s:%s\n",
                newsock.peeraddr[2], newsock.peeraddr[1])

  broadcast_string( str, newsock )

end # accept_new_connection
```

The first task is to accept the client's connection using the `accept` method. The result of this function is a socket that is your end of the connection to the client. You immediately add this socket to your `descriptors` array using the `push` method and then provide a salutation to this new client, welcoming them to the chat with the `write` method. Next, you construct a string to notify others that a new client is connected and then send this message to everyone using the local `broadcast_string` method.

The final local helper method is `broadcast_string`. This method does little more than broadcast a string, something that has been omitted thus far. The inputs to this method are the string to broadcast and a socket object (`omit_sock`):

Listing 23. Broadcasting a string using broadcast_string

```
def broadcast_string( str, omit_sock )

  @descriptors.each do |clisock|
    if clisock != @serverSocket && clisock != omit_sock
      clisock.write(str)
    end
  end

  print(str)

end # broadcast_string
```

The `omit_sock` argument is a socket that you want to omit from the broadcast. For example, if a client sends a message, you want this message to go to all sockets but not back to the originating client. Therefore, in the loop you check for the server socket (which is in the listening state and won't read any data) and the `omit_sock`. For all sockets that aren't one of these two, you send the message through the socket using the `write` method. You also print the message to stdout so the chat server has a running list of all communication.

Note the special iterator that is used for `broadcast_string`. Using this Ruby construct, you loop for each object found in the `descriptors` array. Each object found in the array is given the name `clisock` so it can be referenced. This simple method for iterating over an object is a powerful and simple mechanism in Ruby.

Instantiating a new ChatServer

You've now seen the entire chat server written in four methods in Ruby. Let's next look at how to instantiate a new `ChatServer` object in Ruby.

This section demonstrates two methods for instantiating a new `ChatServer`. Both methods are readable, but one can make programs simpler and easier to read.

With the first method, you create a new `ChatServer` object and then use the object to call its `run` method:

Listing 24. Creating a ChatServer object and calling run

```
myChatServer = ChatServer.new( 2626 )
myChatServer.run
```

This is easy to understand, but there's another way. In Ruby, you can pull methods together into a single string. They're evaluated from left to right, so the following instantiation is identical to the first:

Listing 25. Creating a ChatServer object and calling run, the short version

```
myChatServer = ChatServer.new( 2626 ).run
```

Next, let's look at the `ChatServer` class in action.

Demonstrating the chat server

To demonstrate the `ChatServer` class in action, this section shows a sample dialog between two clients. User-entered text is shown in bold to differentiate it from other text. The server and the first client both exist on the host *plato* (IP address 192.168.1.1). The second client is on the host *camus* (IP address 192.168.1.2). This section is that of the `ChatServer`:

Listing 26. ChatServer demo: On the host

```
[plato]$ ruby chatsrvr.rb
Chatserver started on port 2626
Client joined plato.mtjones.com:50417
Client joined 192.168.1.1:1442
[192.168.1.1|1442]: Hello, is anyone there?
[plato.mtjones.com|50417]: Yes, I'm here.
[192.168.1.1|1442]: Oh, it's you. Goodbye.
Client left 192.168.1.1:1442
```

The following text is the first client's session:

Listing 27. ChatServer demo: On the first client

```
[plato]$ telnet localhost 2626
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
You're connected to the Ruby chatserver
Client joined 192.168.1.1:1442
[192.168.1.1|1442]: Hello, is anyone there?
Yes, I'm here.
[192.168.1.1|1442]: Oh, it's you. Goodbye.
Client left 192.168.1.1:1442
```

The following text is the second client's session:

Listing 28. ChatServer demo: On the second client

```
[mtj@camus]$ telnet 192.168.1.2 2626
Trying 192.168.1.2...
Connected to 192.168.1.2.
Escape character is '^]'.
You're connected to the Ruby chatserver
Hello, is anyone there?
[plato.mtjones.com|50417]: Yes, I'm here.
Oh, it's you. Goodbye.
```

```
                          ^]
telnet> close
Connection closed.
[mtj@camus]$
```

As shown, the server provides the dialog generated by each connected client as well as client connect and disconnect messages. In less than 50 lines of Ruby code, you constructed a working chat server that can support an arbitrary number of clients.

But wait, there's more! Ruby also provides a set of high-level networking classes that implement application layer protocols. The next section reviews -- the HTTP, POP3, and SMTP classes.

Section 6. High-level networking classes

Net::HTTP

Implementing the client side of HTTP is useful when you're building Web robots or site-scraping utilities. The Ruby `Net::HTTP` class provides a simple set of methods that can do just that.

The following example grabs the index file for the developerWorks page and loads it into a string. It's printed here, but it could instead be searched for links to follow, images to download, and so on:

Listing 29. Demonstrating Net::HTTP

```
require 'net/http'

httpclient = Net::HTTP::new("www-130.ibm.com")
resp, data = httpclient.get("/developerworks/index.html")
print data
```

Methods are also provided to submit a HEAD request (which is used to retrieve information about the page but not the page itself) and also to post to the page.

Net::SMTP

Ruby makes it easy to send e-mail from an application. The `Net::SMTP` class provides a simple way to generate an e-mail, including destination, source, subject, and body, and then send it on to the target mail server:

Listing 30. Demonstrating Net::SMTP

```
require 'net/smtp'

user_from = "mtj@mtjones.com"
user_to   = "you@mail.com"

smtpclient = Net::SMTP::new( '192.168.1.1' )
the_email = "From: mtj@mtjones.com\nSubject: Hello\n\nHi, this works.\n\n"

smtpclient.start
smtpclient.sendmail(the_email, user_from, user_to)
smtpclient.finish
```

Instead of text, HTML can also be encoded within the e-mail string.

Net::POP3

The Post Office Protocol, version 3 (POP3) is a standard server-side mail protocol. You can use the Ruby `Net::POP3` class to retrieve mail from a POP3 server. Ruby can also work with authenticated POP3 and Internet Message Access Protocol (IMAP, with the `Net::IMAP` class).

The following example connects to a POP3 mail server, prints the number of e-mails that are available, and then iterates through each e-mail, displaying the e-mail (which includes the SMTP headers and e-mail body):

Listing 31. Demonstrating Net::POP3

```
require 'net/pop'

popclient = Net::POP3::new("192.168.1.1")
popclient.start('username', 'password')

printf("Number of messages available %d\n", popclient.n_mails)

popclient.mails.each {|new_email|
  print new_email.pop
}

popclient.finish
```

This example works as follows:

1. A new POP3 instance is created to the server at 192.168.1.1.
2. You authenticate yourself to the server using the `start` method (which provides the username and password).
3. The `n_mails` method returns the number of e-mails that are present for the user.
4. The `mails.each` methods iterate through the available e-mails, using the `pop` method to grab the e-mail header and body text.
5. The POP3 session is closed with the `finish` method.

Section 7. Summary

Summary

This tutorial has explored the classes that make networking application development possible in Ruby. It discussed the basic classes for sockets programming (such as the `Sockets` class) and also the classes that help to simplify sockets programming in Ruby, such as `TCPSocket` and `TCPServer`.

After demonstrating the basics of sockets programming in Ruby, the tutorial presented a scalable chat server in Ruby that requires fewer than 50 lines of Ruby script. Finally, it presented some of the higher-level classes that implement application layer protocols such as SMTP and POP3.

I hope you find Ruby to be a simple and elegant language for networking development. Ruby continues to evolve and now finds itself on par with the more visible object-oriented scripting languages, such as Perl and Python. Languages are mostly a matter of taste, but with Ruby's simplicity and intuitiveness, it's a useful tool in your engineering toolbox.

Resources

Learn

- The Ruby Garden has a variety of articles and tutorials on the Ruby language.
- Ruby Central is another great site for information about the Ruby language.
- Avoid the common sockets programming mishaps outlined in Five pitfalls of Linux sockets programming, by M. Tim Jones.
- Learn more about sockets programming in these developerWorks tutorials:
  - Sockets programming in Python (also by M. Tim Jones)
  - Programming Linux Sockets, Part 1
  - Programming Linux Sockets, Part 2
- Discover network programming in a variety of languages (including Ruby) in *BSD Sockets Programming from a Multi-Language Perspective* by M. Tim Jones (Charles River Media, 2003).
- Find more resources for Linux developers in the developerWorks Linux zone.

Get products and technologies

- Download the latest version of Ruby from the official Web site.
- Order the no-charge SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Build your next development project on Linux with IBM trial software, available for download directly from developerWorks.

Discuss

- Get involved in the developerWorks community by participating in developerWorks blogs.

About the author

M. Tim Jones
M. Tim Jones is a senior principal software engineer with Emulex Corp. in Longmont, Colorado, where he architects and designs networking and storage products. Tim's design activities have ranged from real-time kernels for communication satellites to networking protocols and embedded firmware. He is the author of many articles on subjects from artificial intelligence (AI) to application-layer protocol development. He has also the author of *AI Application Programming* (now in its second edition), *GNU/Linux Application Programming, BSD Sockets Programming from a Multilanguage Perspective*, and *TCP/IP Application Layer Protocols for Embedded*

Systems (all through Charles River Media).