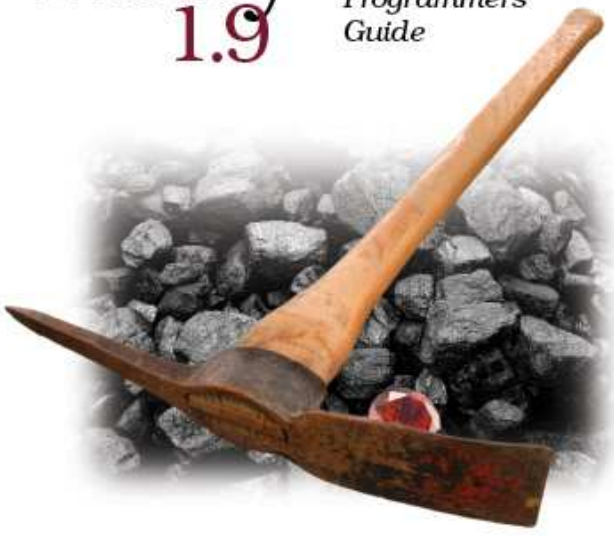
The Pragmatic Programmers

10th Anniversary

Updated for Ruby 1.9.2

Programming Ruby 1.9

The Pragmatic Programmers' Guide



Dave Thomas
with Chad Fowler and Andy Hunt

The Facets of Ruby Series

Ruby 1.9 Socket Library

Dave Thomas

with Chad Fowler
Andy Hunt

The Pragmatic Bookshelf
Raleigh, North Carolina Dallas, Texas

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at http://www.pragprog.com.

Printed in the United States of America.

ISBN-10:

ISBN-13:

Printed on acid-free paper.

1.0 printing, November 2010

Version: 2010-11-11

Contents

1 Introduction **5**

2 Socket Library **6**

Addrinfo: 7 BasicSocket: 14 Socket::Constants: 18 Socket: 20 IPSocket: 28
TCPSocket: 29 SOCKSSocket: 30 TCPServer: 31 UDPSocket: 32
UnixSocket: 34 UnixServer: 36

A Bibliography **37**

Chapter 1

Introduction

This material was, for 10 years, an appendix in *Programming Ruby* [TFH08].[1] However, as of the Tenth Anniversary printing in November, 2010, I've decided to extract this appendix from the printed book and make it freely available online.

Back when the first edition of Programming Ruby appeared, knowing the low-level socket API was important—it was how you'd communicate across a network. But as Ruby matured, so did its libraries, both built-in and external. Today, you're unlikely to be grovelling around at the socket level. Instead you'll use one of the higher level libraries or frameworks to get the job done.

So rather than kill a bunch a trees by printing thousands of copies of an appendix that few people would need, I decided to remove it from the book. But rather than simply discard the material, I've updated it for Ruby 1.9.2 and made it available in electronic form (PDF, mobi, and epub) for free.

Dave Thomas, November 2010

1. http://pragprog.com/titles/ruby3

Chapter 2

Socket Library

The socket and network libraries are such important parts of integrating Ruby applications with the Internet and other communications-based environments. However, the chances are pretty good that you'll never need to code down at this level—if you're writing a web applications, tools such as Rack abstract the communications layers away. If you want to write a socket-based server, the GServer library will keep you away from the messy details. So this documentation is primarily of interested to those hardy, dedicated folks who write the frameworks and libraries that the rest of us use.

The socket classes form a hierarchy based on class IO.

```
IO
  BasicSocket
    IPSocket
      TCPSocket
        TCPServer
      UDPSocket
    Socket
    UNIXSocket
      UNIXServer
```

Because the socket calls are implemented in a library, you'll need to remember to add the following line to your code:

```
require 'socket'
```

Class Addrinfo < Data

The socket classes used to manipulate addresses using something called a *struct sockaddr*, which is effectively an opaque binary string. As of Ruby 1.9.2, the library now uses Addrinfo objects to represent addresses. For now, both the opaque string and an Addrinfo object are accepted whereever an address is expected. *1.9.2*

Socket-based programming spans a range of communications protocols, addressing schemes, and transport mechanisms. The interested reader should have a look at *Unix Network Programming, Volume 1: Networking APIs: Sockets and Xti* [Ste98] by the late W. Richard Stevens for the definitive description of how this addressing works.

The Addrinfo class captures the *protocol family* (also called the *communications domain*), the *socket type*, the *protocol*, and the *socket address*. Between them, these four fields uniquely identify a socket endpoint.

The socket address (often called a *sockaddr*) has its own internal structure. Just to make things interesting, that structure varies depending on the protocol family of the socket. A PF_INET socket, representing a TCP or UDP protocol, will need an IP address and a port, whereas a PF_LOCAL socket (sometimes called PF_UNIX) needs a path to a local file.

You construct a sockaddr as either an array or as a binary string. The array form is most commonly used when people create the sockaddr, and the binary form when it is returned by API calls such as Socket#sockaddr_in.

For PF_INET and PF_INET6 socket, the sockaddr array should contain

```
[ family, port, name, address ]
```

family: The protocol family, expressed as an integer (Socket::PF_INET) or a string with or without the leading PF_ ("PF_INET", "INET", "INET6").

port: is the numeric port number.

name: Is not used in address manipulation—it is used as a documentation field when creating *addr*.inspect.

address: The IP address as a string (a dotted quad for INET and a colon separated set of hex digits for INET6).

The address array for Unix domain sockets looks like

```
[ family, path ]
```

The family is Socket::PF_LOCAL (or "PF_LOCAL" or "LOCAL") and the path is a locat filesystem path.

Class methods

foreach

Addrinfo.foreach(*nodename*, *service*) { |addr| ... } → [*addr*...]
Addrinfo.foreach(*nodename*, *service*, *family*) { |addr| ... } → [*addr*...]
Addrinfo.foreach(*nodename*, *service*, *family*, *socktype*) { |addr| ... } → [*addr*...]
Addrinfo.foreach(*nodename*, *service*, *family*, *socktype*, *protocol*) { |addr| ... } → [*addr*...]
Addrinfo.foreach(*nodename*, *service*, *family*, *socktype*, *protocol*, *flags*) { |addr| ... } → [*addr*...]

Calls Addrinfo#getsockinfo with the given parameters, then passes each of the returned *addr* objects to the block. Returns the array returned by getaddrinfo.

getaddrinfo

Addrinfo.getaddrinfo(*nodename*, *service*) → [*addr*...]
Addrinfo.getaddrinfo(*nodename*, *service*, *family*) → [*addr*...]
Addrinfo.getaddrinfo(*nodename*, *service*, *family*, *socktype*) → [*addr*...]
Addrinfo.getaddrinfo(*nodename*, *service*, *family*, *socktype*, *protocol*) → [*addr*...]
Addrinfo.getaddrinfo(*nodename*, *service*, *family*, *socktype*, *protocol*, *flags*) → [*addr*...]

Returns all possible Address objects for the given *nodename* and *service*. The result set may be constrained to addresses that have a particular family, socket type, protocol. The flags may be a bitwise OR of the Socket::AI_*xxx* values.

```
require 'socket'
puts Addrinfo.getaddrinfo('localhost', 80).map(&:inspect)
```

produces:

```
#<Addrinfo: [::1]:80 UDP (localhost)>
#<Addrinfo: [::1]:80 TCP (localhost)>
#<Addrinfo: [fe80::1%lo0]:80 UDP (localhost)>
#<Addrinfo: [fe80::1%lo0]:80 TCP (localhost)>
#<Addrinfo: 127.0.0.1:80 UDP (localhost)>
#<Addrinfo: 127.0.0.1:80 TCP (localhost)>
```

ip

Addrinfo.ip(*host*) → *addr*

Returns an Addrinfo with the address portion only filled in. The given host is looked up, and the address is extracted from the first sockaddr returned. The protocol, socket type, and port fields of the address are left as zero.

```
require 'socket'
Addrinfo.ip("127.0.0.1") # => #<Addrinfo: 127.0.0.1>
Addrinfo.ip("localhost") # => #<Addrinfo: ::1 (localhost)>
```

new Addrinfo.new(*sockaddr* ⟨ , family ⟨ , socktype ⟨ , protocol ⟩ ⟩ ⟩ → *addr*

Creates an Addrinfo object for the given Unix or INET6 *sockaddr*. The format of *sockaddr* is described at the start of this section. Normally the *family*, *socktype*, and *protocol* can be inferred from the *sockaddr*—if specified they override the *sockaddr* information. The family and socktype can be specified as integers (using the constants defined in class Socket) or as symbols or strings. The protocol can only be specified as an integer.

```
require 'socket'
p Addrinfo.new(["LOCAL", "/tmp/control_socket"])
p Addrinfo.new(["INET", 80, "dave.local", "127.0.0.1"])
```

produces:

```
#<Addrinfo: /tmp/control_socket SOCK_STREAM>
#<Addrinfo: 127.0.0.1:80 (dave.local)>
```

tcp Addrinfo.tcp(*host*, *port*) → *addr*

Returns a TCP Addrinfo object for the given host and port.

```
require 'socket'
Addrinfo.tcp('localhost', 80)    # => #<Addrinfo: [::1]:80 TCP (localhost)>
Addrinfo.tcp('localhost', 'www') # => #<Addrinfo: [::1]:80 TCP (localhost:www)>
Addrinfo.tcp('127.0.0.1', 'www') # => #<Addrinfo: 127.0.0.1:80 TCP (:www)>
```

udp Addrinfo.udp(*host*, *port*) → *addr*

Returns a UDP Addrinfo object for the given host and port.

```
require 'socket'
Addrinfo.udp('localhost', 'ntp') # => #<Addrinfo: [::1]:123 UDP (localhost:ntp)>
```

unix Addrinfo.unix(*path*, *socktype="SOCK_STREAM"*) → *addr*

Returns a PF_LOCAL Addrinfo object for the given path.

```
require 'socket'
Addrinfo.unix('/tmp/mysock')              # => #<Addrinfo: /tmp/mysock
                                          # .. SOCK_STREAM>
Addrinfo.unix('/tmp/mysock', :SOCK_DGRAM) # => #<Addrinfo: /tmp/mysock
                                          # .. SOCK_DGRAM>
```

Instance methods

Predicates

addr.ip? → true or false
addr.ipv4? → true or false
addr.ipv4_loopback? → true or false
addr.ipv4_multicast? → true or false
addr.ipv4_private? → true or false
addr.ipv6? → true or false
addr.ipv6_linklocal? → true or false
addr.ipv6_loopback? → true or false
addr.ipv6_mc_global? → true or false
addr.ipv6_mc_linklocal? → true or false
addr.ipv6_mc_nodelocal? → true or false
addr.ipv6_mc_orglocal? → true or false
addr.ipv6_mc_sitelocal? → true or false
addr.ipv6_multicast? → true or false
addr.ipv6_sitelocal? → true or false
addr.ipv6_unspecified? → true or false
addr.ipv6_v4compat? → true or false
addr.ipv6_v4mapped? → true or false
addr.unix? → true or false

Returns a boolean depending on the value of the given property.

Properties

addr.afamily → *integer*
addr.ip_port → *integer*
addr.pfamily → *integer*
addr.protocol → *integer*
addr.socktype → *integer*
addr.unix_path → *string*

Returns the given property of *addr*.

bind

addr.bind → *sock*
addr.bind { |sock| ... } → *obj*

Binds a socket to the address and protocol given by *addr*. With no block returns the socket object. With a block, invokes it with the socket, closes the socket when the block returns, and returns the value of the block.

canonname

addr.canonname → *string* or nil

If the address was created with the Socket::AI_CANONNAME option, return the actual host name, resolving any CNAMEs.

```
require 'socket'
addr = Addrinfo.getaddrinfo("pragprog.com", 80, :INET, :STREAM, nil, Socket::AI_CANONNAME)
addr.first.canonname # => "pragprog.com"
addr = Addrinfo.getaddrinfo("wiki.pragprog.com", 80, :INET, :STREAM, nil, Socket::AI_CANONNAME)
addr.first.canonname # => "pragprog.com"
addr = Addrinfo.getaddrinfo("wiki.pragprog.com", 80, :INET, :STREAM, nil)
addr.first.canonname # => nil
```

connect

addr.connect → *sock*
addr.connect { |sock| ... } → *obj*

Creates a socket connection to the address and protocol given by *addr*. With no block returns the socket object. With a block, invokes it with the socket, closes the socket when the block returns, and returns the value of the block.

```
require 'socket'
addr = Addrinfo.tcp('localhost', 80)
addr.connect do |socket|
  socket.puts "GET / HTTP/1.0\r\n\r\n"
  3.times { puts socket.gets }
end
```

produces:

```
HTTP/1.1 200 OK
Date: Thu, 11 Nov 2010 19:01:27 GMT
Server: Apache/2.2.14 (Unix) mod_ssl/2.2.14 OpenSSL/0.9.8l DAV/2
```

connect_from

addr.connect_from(⟨local_addr⟩*) → *sock*
addr.connect_from(⟨local_addr⟩*) { |sock| ... } → *obj*

Works like Addrinfo#connect, but binds the local end of the connection to any interface or port given as parameters. The parameters can be an Addrinfo object, the same parameters taken by Addrinfo.getaddrinfo if *addr* is an PF_INET object, or a path if *addr* is a PF_LOCAL object.

connect_to

addr.connect_to(⟨remote_addr⟩*) → *sock*
addr.connect_to(⟨remote_addr⟩*) { |sock| ... } → *obj*

Works like Addrinfo#connect_from, but *addr* specifies the local end and the parameters specify the remote end of the connection.

family_addrinfo

addr.family_addrinfo(⟨args⟩*) → *new_addr*

Constructs a new Addrinfo with the same protocol family as *addr* but with a different address.

```
require 'socket'
addr = Addrinfo.tcp('127.0.0.1', 80)
addr.family_addrinfo('google.com', 'ftp') # => #<Addrinfo: 74.125.45.99:21 TCP
                                          # .. (google.com:ftp)>
```

getnameinfo

addr.getnameinfo(*options*=0) → [*node*, *service*]

Extract the node name (or address) and the service name (or port number) from the sockaddr help in *addr*. The options are a bitwose OR of the Socket::NI_*xxx* constants.

```
require 'socket'
a = Addrinfo.tcp('pragprog.com', 80)
a                                    # => #<Addrinfo: 209.251.185.98:80 TCP
                                     # .. (pragprog.com)>
a.getnameinfo                        # => ["209.251.185.98", "http"]
a.getnameinfo(Socket::NI_NUMERICHOST) # => ["209.251.185.98", "http"]
a.getnameinfo(Socket::NI_NUMERICSERV) # => ["209.251.185.98", "80"]
```

inspect_sockaddr

addr.inspect_sockaddr → *string*

Inspect just the sockaddr portion of *addr*.

ip_unpack

addr.ip_unpack → [*host*, *port*]

Returns the numeric host and port for an AF_INET Addrinfo.

```
require 'socket'
Addrinfo.tcp("pragprog.com", "www").ip_unpack # => ["209.251.185.98", 80]
```

ipv6_to_ipv4

addr.ipv6_to_ipv4 → *ipv4_addr* or nil

If *addr* is an IPV4-mapped IPV6 address, return a new Addrinfo containing the corresponding IPV4 address.

```
require 'socket'
Addrinfo.ip("::ffff:192.0.2.128").ipv6_to_ipv4 # => #<Addrinfo: 192.0.2.128>
Addrinfo.ip("::1").ipv6_to_ipv4                # => nil
```

listen

addr.listen(*backlog*=5) → *sock*
addr.listen(*backlog*=5) { |sock| ... } → *obj*

Binds a socket to *addr* and then issues a listen on it. With no block, returns the socket. With a block, passes the socket as a parameter, closes the socket at the end, and returns the block's value.

to_sockaddr

addr.to_sockaddr → *binary_string*

Returns the sockaddr component of *addr* as a packed binary string. (For the layout, see the Unix documentation for *inet(4)* and *unix(4)*.

Class BasicSocket < IO

BasicSocket is an abstract base class for all other socket classes.

Class methods

do_not_reverse_lookup

BasicSocket.do_not_reverse_lookup → *true* or *false*

Returns the value of the global reverse lookup flag.

do_not_reverse_lookup=

% BasicSocket.do_not_reverse_lookup = *true* or *false*

Sets the global reverse lookup flag. If set to true, queries on remote addresses will return the numeric address but not the host name.

Previously this flag defaulted to false, which caused the occasional performance problem. It now defaults to true. *1.9.2*

for_fd

BasicSocket.for_fd(*fd*) → *sock*

Wraps an already open file descriptor into a socket object.

Instance methods

close_read

sock.close_read → nil

Closes the readable connection on this socket.

close_write

sock.close_write → nil

Closes the writable connection on this socket.

connect_address

sock.connect_address → *addr_info*

Return the address that should be used to connect to this socket. Normally this is the same as local_address, but the IPV4 and IPV4 unspecified addresses are replaced by their corresponding loopback addresses. *1.9.2*

```
require 'socket'
listening_socket = Addrinfo.tcp('::', 0).listen
listening_socket.local_address   # => #<Addrinfo: [::]:56676 TCP>
listening_socket.connect_address # => #<Addrinfo: [::1]:56676 TCP>
```

getpeereid

sock.getpeereid → [*euid*, *egid*]

Return the effective user ID and effective group ID of the socket. *1.9.2*

getpeername

sock.getpeername → *string*

Returns the struct sockaddr structure associated with the other end of this socket connection.

getsockname

sock.getsockname → *string*

Returns the struct sockaddr structure associated with *sock*.

getsockopt

sock.getsockopt(*level*, *optname*) → *sockopt*

Returns the value of the specified option as a Socket::Option object. The *level* is an integer, string, or symbol drawn from the SOL_*xxx* constants, and the option is an integer, symbol, or string drawn from the SO_*xxx* constants.

```
require 'socket'
sock = Socket.new(Socket::PF_INET, Socket::SOCK_STREAM) # => #<Socket:fd 3>
opt = sock.getsockopt(:SOL_SOCKET, :SO_DEBUG)           # => #<Socket::Option:
                                                        # .. INET SOCKET DEBUG
                                                        # .. 0>
opt.bool                                                # => false
opt = sock.getsockopt(:SOL_SOCKET, :SO_RCVBUF)          # => #<Socket::Option:
                                                        # .. INET SOCKET RCVBUF
                                                        # .. 262140>
opt.int                                                 # => 262140
```

local_address

sock.local_address → *addr_info*

Return the address information for the local end of a socket. *1.9.2*

```
require 'socket'
s = Socket.tcp('google.com', 80)
s.local_address  # => #<Addrinfo: 192.168.1.17:56677 TCP>
s.remote_address # => #<Addrinfo: 74.125.45.99:80 TCP>
```

recv

sock.recv(*len*, ⟨ , *flags* ⟩) → *string*

Receives up to *len* bytes from *sock*.

recvmsg *sock*.recvmsg(*max_data_len*=nil, *flags*=0, *max_control_len*=nil, *options*={}) → [*data*, *sender_addr*, *flags*, *controls*]

Uses the *recvmsg(2)* call to receive a message from a socket. One use of this is to pass open file descriptors between processes. The *ancillary* parameter can be the three-element array [*cmsg_level*, *cmsg_type*, *cmsg_data*] or a Socket::AncillaryData object. *1.9.2*

recvmsg_nonblock *sock*.recvmsg_nonblock(*max_data_len*=nil, *flags*=0, *max_control_len*=nil, *options*={}) → [*data*, *sender_addr*, *flags*, *controls*]

Nonblocking version of recvmsg. *1.9.2*

recv_nonblock *sock*.recv_nonblock(*len*, ⟨ , *flags* ⟩) → *string*

Receives up to *len* bytes from *sock* after first setting the socket into nonblocking mode. If the underlying recvfrom call returns 0, an empty string is returned. *1.9*

remote_address *sock*.remote_address → *addr_info*

Return the address information for the remote end of a socket. *1.9.2*

```
require 'socket'
s = Socket.tcp('google.com', 80)
s.local_address  # => #<Addrinfo: 192.168.1.17:56678 TCP>
s.remote_address # => #<Addrinfo: 74.125.45.99:80 TCP>
```

send *sock*.send(*string*, *flags*, ⟨ , *to* ⟩) → *int*

Sends *string* over *sock*. If specified, *to* is a struct sockaddr or an Addrinfo specifying the recipient address. *flags* are the sum of one or more of the MSG_ options (listed on page 18). Returns the number of characters sent. *1.9.2*

sendmsg *sock*.sendmsg(*data*, *flags*=0, ⟨ , *to* ⟨ , ancillary ⟩* ⟩) → *int*

Uses the *sendmsg(2)* call to send a message with optional access rights data to another socket. One use of this is to pass open file descriptors between processes. The *ancillary* parameter can be the three-element array [*cmsg_level*, *cmsg_type*, *cmsg_data*] or a Socket::AncillaryData object. *1.9.2*

sendmsg_nonblock *sock*.sendmsg_nonblock(*data*, *flags*=0, ⟨ , *to* ⟨ , ancillary ⟩* ⟩) → *int*

Nonblocking version of sendmsg. *1.9.2*

setsockopt *sock*.setsockopt(*level*, *optname*, *optval*) → 0

Sets a socket option. *level* is one of the socket-level options (listed on the next page). *optname* and *optval* are protocol specific—see your system documentation for details.

shutdown *sock*.shutdown(*how*=2) → 0

Shuts down the receive (*how* == 0), sender (*how* == 1), or both (*how* == 2), parts of this socket.

Module

Socket::Constants

Defines the constants used as options and parameters throughout the socket library.

Constants are available only on architectures that support the related facility.

Types

SOCK_DGRAM, SOCK_PACKET, SOCK_RAW, SOCK_RDM, SOCK_SEQPACKET, SOCK_STREAM

Protocol families

PF_APPLETALK, PF_ATM, PF_AX25, PF_CCITT, PF_CHAOS, PF_CNT, PF_COIP, PF_DATAKIT, PF_DEC, PF_DLI, PF_ECMA, PF_HYLINK, PF_IMPLINK, PF_INET, PF_INET6, PF_IPX, PF_ISDN, PF_ISO, PF_KEY, PF_LAT, PF_LINK, PF_LOCAL, PF_MAX, PF_NATM, PF_NDRV, PF_NETBIOS, PF_NETGRAPH, PF_NS, PF_OSI, PF_PACKET, PF_PIP, PF_PPP, PF_PUP, PF_ROUTE, PF_RTIP, PF_SIP, PF_SNA, PF_SYSTEM, PF_UNIX, PF_UNSPEC, PF_XTP

Address families

AF_APPLETALK, AF_ATM, AF_AX25, AF_CCITT, AF_CHAOS, AF_CNT, AF_COIP, AF_DATAKIT, AF_DEC, AF_DLI, AF_E164, AF_ECMA, AF_HYLINK, AF_IMPLINK, AF_INET, AF_INET6, AF_IPX, AF_ISDN, AF_ISO, AF_LAT, AF_LINK, AF_LOCAL, AF_MAX, AF_NATM, AF_NDRV, AF_NETBIOS, AF_NETGRAPH, AF_NS, AF_OSI, AF_PACKET, AF_PPP, AF_PUP, AF_ROUTE, AF_SIP, AF_SNA, AF_SYSTEM, AF_UNIX, AF_UNSPEC

Send/receive options

MSG_COMPAT, MSG_CONFIRM, MSG_CTRUNC, MSG_DONTROUTE, MSG_DONTWAIT, MSG_EOF, MSG_EOR, MSG_ERRQUEUE, MSG_FIN, MSG_FLUSH, MSG_HAVEMORE, MSG_HOLD, MSG_MORE, MSG_NOSIGNAL, MSG_OOB, MSG_PEEK, MSG_PROXY, MSG_RCVMORE, MSG_RST, MSG_SEND, MSG_SYN, MSG_TRUNC, MSG_WAITALL

Socket-level options

SOL_ATALK, SOL_AX25, SOL_IP, SOL_IPX, SOL_SOCKET, SOL_TCP, SOL_UDP

Socket options

SO_ACCEPTCONN, SO_ACCEPTFILTER, SO_ALLZONES, SO_ATTACH_FILTER, SO_BINDTODEVICE, SO_BINTIME, SO_BROADCAST, SO_DEBUG, SO_DETACH_FILTER, SO_DONTROUTE, SO_DONTTRUNC, SO_ERROR, SO_KEEPALIVE, SO_LINGER, SO_MAC_EXEMPT, SO_NKE, SO_NOSIGPIPE, SO_NO_CHECK, SO_NREAD, SO_OOBINLINE, SO_PASSCRED, SO_PEERCRED, SO_PEERNAME, SO_PRIORITY, SO_RCVBUF, SO_RCVLOWAT, SO_RCVTIMEO, SO_RECVUCRED, SO_REUSEADDR, SO_REUSEPORT, SO_SECURITY_AUTHENTICATION, SO_SECURITY_ENCRYPTION_NETWORK, SO_SECURITY_ENCRYPTION_TRANSPORT, SO_SNDBUF, SO_SNDLOWAT,

SO_SNDTIMEO, SO_TIMESTAMP, SO_TIMESTAMPNS, SO_TYPE, SO_USELOOPBACK, SO_WANTMORE, SO_WANTOOBFLAG

Qualtity-of-service options

SOPRI_BACKGROUND, SOPRI_INTERACTIVE, SOPRI_NORMAL

Multicast options

IP_ADD_MEMBERSHIP, IP_ADD_SOURCE_MEMBERSHIP, IP_BLOCK_SOURCE, IP_DEFAULT_MULTICAST_LOOP, IP_DEFAULT_MULTICAST_TTL, IP_DONTFRAG, IP_DROP_MEMBERSHIP, IP_DROP_SOURCE_MEMBERSHIP, IP_FREEBIND, IP_HDRINCL, IP_IPSEC_POLICY, IP_MAX_MEMBERSHIPS, IP_MINTTL, IP_MSFILTER, IP_MTU, IP_MTU_DISCOVER, IP_MULTICAST_IF, IP_MULTICAST_LOOP, IP_MULTICAST_TTL, IP_ONESBCAST, IP_OPTIONS, IP_PASSSEC, IP_PKTINFO, IP_PKTOPTIONS, IP_PMTUDISC_DO, IP_PMTUDISC_DONT, IP_PMTUDISC_WANT, IP_PORTRANGE, IP_RECVDSTADDR, IP_RECVERR, IP_RECVIF, IP_RECVOPTS, IP_RECVRETOPTS, IP_RECVSLLA, IP_RECVTOS, IP_RECVTTL, IP_RETOPTS, IP_ROUTER_ALERT, IP_SENDSRCADDR, IP_TOS, IP_TTL, IP_UNBLOCK_SOURCE, IP_XFRM_POLICY

TCP options

TCP_CORK, TCP_DEFER_ACCEPT, TCP_INFO, TCP_KEEPCNT, TCP_KEEPIDLE, TCP_KEEPINTVL, TCP_LINGER2, TCP_MAXSEG, TCP_MD5SIG, TCP_NODELAY, TCP_NOOPT, TCP_NOPUSH, TCP_QUICKACK, TCP_SYNCNT, TCP_WINDOW_CLAMP

getaddrinfo error codes

EAI_ADDRFAMILY, EAI_AGAIN, EAI_BADFLAGS, EAI_BADHINTS, EAI_FAIL, EAI_FAMILY, EAI_MAX, EAI_MEMORY, EAI_NODATA, EAI_NONAME, EAI_OVERFLOW, EAI_PROTOCOL, EAI_SERVICE, EAI_SOCKTYPE, EAI_SYSTEM

ai_flag values

AI_ADDRCONFIG, AI_ALL, AI_CANONNAME, AI_DEFAULT, AI_MASK, AI_NUMERICHOST, AI_NUMERICSERV, AI_PASSIVE, AI_V4MAPPED, AI_V4MAPPED_CFG

Class Socket < BasicSocket

Class Socket provides access to the operating system socket implementation. It can be used to provide more system–specific functionality than the protocol-specific socket classes but at the expense of greater complexity.

Class methods

accept_loop Socket.accept_loop(*sockets...*) { |socket, client_addr_info| ... }

Takes a list of listening sockets or arrays of listening sockets. When a connection arrives on any, accepts it and invokes the block, passing in the new socket and the client address. The block is invoked serially—if you need to handle multiple concurrent connections, you'll need to do your own threading in the block (or simply use listen, accept, and select yourself. *1.9.2*

```
require 'socket'
Socket.getaddrinfo('www.microsoft.com', 'http').each do |addr|
  puts addr.join(", ")
end
```

produces:

```
AF_INET, 80, 207.46.170.123, 207.46.170.123, 2, 2, 17
AF_INET, 80, 207.46.170.123, 207.46.170.123, 2, 1, 6
AF_INET, 80, 207.46.170.10, 207.46.170.10, 2, 2, 17
AF_INET, 80, 207.46.170.10, 207.46.170.10, 2, 1, 6
```

getaddrinfo Socket.getaddrinfo(*hostname*, *port*, ⟨ , *family* ⟨ , *socktype* ⟨ , *protocol* ⟨ , *flags* ⟨ , rlookup ⟩ ⟩ ⟩ ⟩ ⟩) → *array*

Returns an array of arrays describing the given host and port (optionally qualified as shown). Each subarray contains the address family, port number, host name, host IP address, protocol family, socket type, and protocol. The *rlookup* parameter overrides the default reverse name lookup option. *1.9.2*

```
require 'socket'
Socket.getaddrinfo('www.microsoft.com', 'http').each do |addr|
  puts addr.join(", ")
end
```

produces:

```
AF_INET, 80, 207.46.170.123, 207.46.170.123, 2, 2, 17
AF_INET, 80, 207.46.170.123, 207.46.170.123, 2, 1, 6
AF_INET, 80, 207.46.170.10, 207.46.170.10, 2, 2, 17
AF_INET, 80, 207.46.170.10, 207.46.170.10, 2, 1, 6
```

gethostbyaddr Socket.gethostbyaddr(*addr*, *type*=AF_INET) → *array*

Returns the host name, address family, and sockaddr component for the given address.

```
require 'socket'
a = Socket.gethostbyname("221.186.184.68")
res = Socket.gethostbyaddr(a[3], a[2])
res.join(', ') # => "carbon.ruby-lang.org, 68.184.186.221.in-addr.arpa,
               # .. 68.64.184.186.221.in-addr.arpa, 2, \xDD\xBA\xB8D"
```

gethostbyname

Socket.gethostbyname(*hostname*) → *array*

Returns a four-element array containing the canonical host name, a subarray of host aliases, the address family, and the address portion of the sockaddr structure.

```
require 'socket'
a = Socket.gethostbyname("63.68.129.130")
a.join(', ') # => "63.68.129.130, , 2, ?D\x81\x82"
```

gethostname

Socket.gethostname → *string*

Returns the name of the current host.

```
require 'socket'
Socket.gethostname # => "wide-boy"
```

getnameinfo

Socket.getnameinfo(*addr* ⟨ , *flags* ⟩) → *array*

Looks up the given address, which may be either a string containing a sockaddr, a Addrinfo, or a three- or four-element array. If *addr* is an array, it should contain the string address family, the port (or nil), and the host name or IP address. If a fourth element is present and not nil, it will be used as the host name. Returns a canonical host name (or address) and port number as an array. *1.9.2*

```
require 'socket'
puts Socket.getnameinfo(["AF_INET", '23', 'www.ruby-lang.org'])
```

produces:

```
carbon.ruby-lang.org
telnet
```

getservbyname

Socket.getservbyname(*service*, *proto*='tcp') → *int*

Returns the port corresponding to the given service and protocol.

```
require 'socket'
Socket.getservbyname("telnet") # => 23
```

getservbyport

Socket.getservbyport(*port*, *proto*='tcp') → *string*

Returns the port corresponding to the given service and protocol. *1.9*

```
require 'socket'
Socket.getservbyport(23) # => "telnet"
```

ip_address_list

Socket.ip_address_list → [*addr*...]

Retruns the addresses of the local network interfaces. *1.9.2*

```
require 'socket'
puts Socket.ip_address_list.map(&:inspect)
```

produces:

```
#<Addrinfo: ::1>
#<Addrinfo: fe80::1%lo0>
#<Addrinfo: 127.0.0.1>
#<Addrinfo: fdd7:b0e5:d31f:2e70:225:4bff:feb8:f12c>
#<Addrinfo: fe80::225:ff:fe44:ac61%en1>
#<Addrinfo: 192.168.1.17>
#<Addrinfo: fe80::225:4bff:feb8:f12c%en2>
#<Addrinfo: 169.254.97.62>
```

new

Socket.new(*domain*, *type* ⟨ , *protocol* ⟩) → *sock*

Creates a socket using the given parameters. If missing, the protocol parameter is inferred from the other two. *1.9.2*

open

Socket.open(*domain*, *type*, *protocol*) → *sock*

Synonym for Socket.new.

pack_sockaddr_in

Socket.pack_sockaddr_in(*port*, *host*) → *str_address*

Given a port and a host, returns the (system dependent) AF_INET sockaddr structure as a string of bytes.

```
require 'socket'
addr = Socket.pack_sockaddr_in(80, "pragprog.com") # Pragprog.com is 65.74.171.137
addr.unpack("CCnC4")        # => [16, 2, 80, 209, 251, 185, 98]
```

pack_sockaddr_un

Socket.pack_sockaddr_un(*path*) → *str_address*

Given a path to a Unix socket, returns the (system dependent) sock_addr_un structure as a string of bytes. Available only on boxes supporting the Unix address family.

```
require 'socket'
sock = UNIXServer.open("/tmp/sample")
addr = Socket.pack_sockaddr_un("/tmp/sample")
addr[0,20]                              # => "\x00\x01/tmp/sample\x00\x00\x00\x00\
                                        # .. x00\x00\x00"
```

pair

Socket.pair(*domain*, *type* ⟨ , *protocol* ⟩) → *array*
Socket.pair(*domain*, *type* ⟨ , *protocol* ⟩) { |sock1, sock2| ... } → *obj*

Returns an array containing a pair of connected, anonymous Socket objects with the given domain, type, and protocol. If omitted, the *protocol* parameter is inferred from the other two. If a block is given, it is passed the two sockets, and the first socket is closed when the block exits. *1.9.2*

socketpair

Socket.socketpair(*domain*, *type*, *protocol*) → *array*

Synonym for Socket.pair.

sockaddr_in

Socket.sockaddr_in(*port*, *host*) → *str_address*

Synonym for pack_sockaddr_in. *1.9*

sockaddr_un

Socket.sockaddr_un(*path*) → *str_address*

Synonym for pack_sockaddr_un. *1.9*

socket_pair

Socket.socket_pair(*domain*, *type*, *protocol*) → *array*

Synonym for Socket.pair.

tcp

Socket.tcp(*host*, *port* ⟨ , *local_interface* ⟨ , *local_port* ⟩ ⟩) { |socket| ... } → *obj*
Socket.tcp(*host*, *port* ⟨ , *local_interface* ⟨ , *local_port* ⟩ ⟩) → *socket*

Create a TCP connection to the given host and port, optionally setting the local interface and port to use. If given a block, pass it the socket, and close the connection, and return the block's value; otherwise return the open socket. *1.9.2*

tcp_server_loop Socket.tcp_server_loop(*host*=nil, *port*) { |socket, client_addr_info| ... }

Accepts connections on all the interfaces for the given port (and optionally host). When a connection arrives, call the block, passing in the connected socket and an Addrinfo structure describing the client. Connections are serialized through the block, so you'll need add concurrently yourself (for example, using threading). In reality, this is a bad idea unless you can control the rate at which clients connect—you're probably better off using listen and accept directly in these cases. In all cases, your code is responsible for closing the socket passed to the block. *1.9.2*

tcp_server_sockets Socket.tcp_server_sockets(*host*=nil, *port*) → [*socket*...]
Socket.tcp_server_sockets(*host*=nil, *port*) { |sockets| ... } → *obj*

Opens a listening socket on each on the interfaces for the host, using the given port or a dynamically assigned port if *port* is zero. If a block is given, passes the array of sockets to it and closes them when the block exits; otherwise returns the array of sockets. The list of sockets is effectively that given by calling *1.9.2*

```
Addrinfo.foreach(host, port, nil, :STREAM, nil, Socket::AI_PASSIVE).map(&:listen)
```

udp_server_loop Socket.udp_server_loop(*host*=nil, *port*) { |msg, source_addr| ... }

Invokes the block for every message that arrives on the given UDP port, passing in the message (a string) and the address of the sender (a Socket::UDPSource object). *1.9.2*

```
# From the internal documentation...
# UDP/IP echo server.
Socket.udp_server_loop(9261) do |msg, msg_src|
  msg_src.reply msg
end
```

udp_server_loop_on Socket.udp_server_loop_on(*sockets*=nil) { |msg, source_addr| ... }

Takes an array of sockets (probably created using udp_server_sockets), and invokes the block repeatedly for each message that arrives on any of them, passing in the message (a string) and the address of the sender (a Socket::UDPSource object). *1.9.2*

udp_server_sockets Socket.udp_server_sockets(*host*=nil, *port*) → [*socket*...]
Socket.ucp_server_sockets(*host*=nil, *port*) { |sockets| ... } → *obj*

Opens a UDP socket on each on the interfaces for the host, using the given port or a dynamically assigned port if *port* is zero. If a block is given, passes the array of sockets to it and closes them when the block exits; otherwise returns the array of sockets. *1.9.2*

unit

Socket.unit(*path*) { |socket| ... } → *obj*
Socket.unit(*path*) → *socket*

Create a domain socket connection on the given path. If given a block, pass it the socket, and close the connection, and return the block's value; otherwise return the open socket. *1.9.2*

unix_server_loop

Socket.tcp_server_loop(*path*) { |socket, client_addr_info| ... }

Accepts connections on all the the Unix domain socket identified by *path*. When a connection arrives, call the block, passing in the connected socket and an Addrinfo structure describing the client. Connections are serialized through the block, so you'll need add concurrently yourself (for example, using threading). In reality, this is a bad idea unless you can control the rate at which clients connect—you're probably better off using listen and accept directly in these cases. In all cases, your code is responsible for closing the socket passed to the block. *1.9.2*

unix_server_socket

Socket.unix_server_socket(*path*) → *socket*
Socket.unix_server_socket(*path*) { |socket| ... } → *obj*

Create a domain socket on the given path (first deleting any existing socket if it is owned by the caller) If a block is given, passes the socket to it and closes and deletes the socket when the block exits; otherwise returns the socket. *1.9.2*

unpack_sockaddr_in

Socket.pack_sockaddr_in(*string_address*) → [*port*, *host*]

Given a string containing a binary addrinfo structure, return the port and host.

```
require 'socket'
addr = Socket.pack_sockaddr_in(80, "pragprog.com")
Socket.unpack_sockaddr_in(addr)        # => [80, "209.251.185.98"]
```

unpack_sockaddr_un

Socket.pack_sockaddr_un(*string_address*) → *path*

Return the path for an AF_LOCAL socket.

```
require 'socket'
addr = Addrinfo.unix("/tmp/socket")
Socket.unpack_sockaddr_un(addr)        # => "/tmp/socket"
```

Instance methods

accept

sock.accept → [*socket*, *caller_address*]

Accepts an incoming connection returning an array containing a new Socket object and an Addrinfo object containing the address of the caller. *1.9.2*

accept_nonblock

sock.accept_nonblock → [*socket*, *caller_address*]

Puts the listening socket into nonblocking mode and then accepts an incoming connection. Throws an exception if no connection is pending. You'll probably use this in conjunction with select. *1.9*

bind

sock.bind(*addr*) → 0

Binds to the given addr, contained in a struct sockaddr string or a Addrinfo object. *1.9.2*

connect

sock.connect(*addr*) → 0

Connects to the given addr, contained in a struct sockaddr string or a Addrinfo object. *1.9.2*

connect_nonblock

sock.connect_nonblock(*addr*) → 0

Connects to the given addr, contained in a struct sockaddr string or a Addrinfo object. The non-blocking option O_NONBLOCK is set on the underlying file descriptor. *1.9.2*

ipv6-only!

sock.ipv6_only!

Set the SO_IPV6_ONLY option on the socket if supported by the underlying operating system. Equivalent to: *1.9.2*

```
def ipv6only!
  if defined? Socket::IPV6_V6ONLY
    self.setsockopt(:IPV6, :V6ONLY, 1)
  end
end
```

listen

sock.listen(*int*) → 0

Listens for connections, using the specified *int* as the backlog.

recvfrom *sock*.recvfrom(*len* ⟨ , *flags* ⟩) → [*data*, *sender_addr*]

Receives up to *len* bytes from *sock*. *flags* is zero or more of the MSG_ options. The first element of the result is the data received. The second element contains an Addrinfo object containing the address of the sender. 1.9.2

recvfrom_nonblock *sock*.recvfrom_nonblock(*len* ⟨ , *flags* ⟩) → [*data*, *sender_addr*]

Receives up to *len* bytes from *sock* in nonblocking mode. *flags* is zero or more of the MSG_ options. The first element of the result is the data received. The second element contains an Addrinfo object containing the address of the sender. 1.9 1.9.2

sysaccept *sock*.sysaccept → [*socket_fd*, *address*]

Accepts an incoming connection. Returns an array containing the (integer) file descriptor of the incoming connection and an Addrinfo object containing the address of the caller. 1.9.2

Class IPSocket < BasicSocket

Class IPSocket is a base class for sockets using IP as their transport. TCPSocket and UDPSocket are children of this class.

Class methods

getaddress IPSocket.getaddress(*hostname*) → *string*

Returns the dotted-quad IP address of *hostname*.

```
require 'socket'
IPSocket.getaddress('www.ruby-lang.org') # => "221.186.184.68"
```

Instance methods

addr *sock*.addr(⟨ *rlookup* ⟩) → *array*

Returns the domain, port, name, and IP address of *sock* as a four-element array. If the *rlookup* parameter is absent, the global do_not_reverse_lookup flag determines if the host address is returned as an address or a name. If the parameter is present, a value of true or :hostname causes a name to be returned; false or :numeric causes a number to be returned. *1.9.2*

```
require 'socket'
u = UDPSocket.new
u.bind('localhost', 8765) # => 0
u.addr                    # => ["AF_INET", 8765, "127.0.0.1", "127.0.0.1"]
u.addr(:numeric)          # => ["AF_INET", 8765, "127.0.0.1", "127.0.0.1"]
u.addr(:hostname)         # => ["AF_INET", 8765, "localhost", "127.0.0.1"]
BasicSocket.do_not_reverse_lookup = false
u.addr                    # => ["AF_INET", 8765, "127.0.0.1", "127.0.0.1"]
```

peeraddr *sock*.peeraddr(⟨ *rlocal* ⟩) → *array*

Returns the domain, port, name, and IP address of the peer. If the *rlookup* parameter is absent, the global do_not_reverse_lookup flag determines if the host address is returned as an address or a name. If the parameter is present, a value of true or :hostname causes a name to be returned; false or :numeric causes a number to be returned. *1.9.2*

recvfrom *sock*.recvfrom(*len* ⟨ , *flags* ⟩) → [*data*, *sender*]

Receives up to *len* bytes on the connection. *flags* is zero or more of the MSG_ options (listed on page 18). Returns a two-element array. The first element is the received data, and the second is an array containing information about the peer. On systems such as my Mac OS X box where the native recvfrom() method does not return peer information for TCP connections, the second element of the array is nil.

```
require 'socket'
t = TCPSocket.new('127.0.0.1', 'ftp')
data = t.recvfrom(40)
data      # => ["220 127.0.0.1 FTP server (tnftpd 2008092", nil]
t.close   # => nil
```

Class TCPSocket < IPSocket

```
require 'socket'
t = TCPSocket.new('localhost', 'ftp')
t.gets    # => "220 ::1 FTP server (tnftpd 20080929) ready.\r\n"
t.close   # => nil
```

Class methods

gethostbyname TCPSocket.gethostbyname(*hostname*) → *array*

Looks up *hostname* and returns its canonical name, an array containing any aliases, the address type (AF_INET), and the dotted-quad IP address.

```
require 'socket'
TCPSocket.gethostbyname('ns.pragprog.com') # => ["pragprog.com", [], 2,
                                           # .. "209.251.185.98"]
```

new TCPSocket.new(*hostname*, *port*) → *sock*

Opens a TCP connection to *hostname* on the *port*.

open TCPSocket.open(*hostname*, *port*) → *sock*

Synonym for TCPSocket.new.

Class SOCKSSocket < TCPSocket

Class SOCKSSocket supports connections based on the SOCKS protocol.

Class methods

new

SOCKSSocket.new(*hostname*, *port*) → *sock*

Opens a SOCKS connection to *port* on *hostname*.

open

SOCKSSocket.open(*hostname*, *port*) → *sock*

Synonym for SOCKSSocket.new.

Instance methods

close

sock.close → nil

Closes this SOCKS connection.

Class TCPServer < TCPSocket

A TCPServer accepts incoming TCP connections. Here is a web server that listens on a given port and returns the time:

```
require 'socket'
port = (ARGV[0] || 80).to_i
server = TCPServer.new('0.0.0.0', port)
while (session = server.accept)
  puts "Request: #{session.gets}"
  session.print "HTTP/1.1 200/OK\r\nContent-type: text/html\r\n\r\n"
  session.print "<html><body><h1>#{Time.now}</h1></body></html>\r\n"
  session.close
end
```

Class methods

new TCPServer.new(⟨ *hostname*, ⟩*port*) → *sock*

Creates a new socket on the given interface (identified by *hostname* and port). If *hostname* is omitted, the server will listen on all interfaces on the current host (equivalent to an address of 0.0.0.0).

open TCPServer.open(⟨ *hostname*, ⟩*port*) → *sock*

Synonym for TCPServer.new.

Instance methods

accept *sock*.accept → *tcp_socket*

Waits for a connection on *sock* and returns a new tcp_socket connected to the caller. See the example on page ??.

Class **UDPSocket** < IPSocket

UDP sockets send and receive datagrams. To receive data, a socket must be bound to a particular port. You have two choices when sending data: you can connect to a remote UDP socket and thereafter send datagrams to that port, or you can specify a host and port every time you send a packet. The following example is a UDP server that prints the message it receives. It is called by both connectionless and connection-based clients.

```
require 'socket'

PORT = 4321

server = UDPSocket.open
server.bind(nil, PORT)
server_thread = Thread.start(server) do |server|  # run server in a thread
  3.times { p server.recvfrom(64) }
end


# Ad-hoc client
UDPSocket.open.send("ad hoc", 0, 'localhost', PORT)

# Connection based client
sock = UDPSocket.open
sock.connect('localhost', PORT)
sock.send("connection-based", 0)
sock.send("second message", 0)
server_thread.join
```

produces:

```
["ad hoc", ["AF_INET", 60665, "127.0.0.1", "127.0.0.1"]]
["connection-based", ["AF_INET", 55041, "127.0.0.1", "127.0.0.1"]]
["second message", ["AF_INET", 55041, "127.0.0.1", "127.0.0.1"]]
```

Class methods

new UDPSocket.new(*family* = AF_INET) → *sock*

Creates a UDP endpoint, optionally specifying an address family.

open UDPSocket.open(*family* = AF_INET) → *sock*

Synonym for UDPSocket.new.

Instance methods

bind

sock.bind(*hostname*, *port*) → 0

Associates the local end of the UDP connection with a given *hostname* and *port*. As well as a host name, the first parameter may be "<broadcast>" or "" (the empty string) to bind to INADDR_BROADCAST and INADDR_ANY, respectively. Must be used by servers to establish an accessible endpoint.

connect

sock.connect(*hostname*, *port*) → 0

Creates a connection to the given *hostname* and *port*. Subsequent UDPSocket#send requests that don't override the recipient will use this connection. Multiple connect requests may be issued on *sock*: the most recent will be used by send. As well as a host name, the first parameter may be "<broadcast>" or "" (the empty string) to bind to INADDR_BROADCAST and INADDR_ANY, respectively.

recvfrom

sock.recvfrom(*len* ⟨ , *flags* ⟩) → [*data*, *sender*]

Receives up to *len* bytes from *sock*. *flags* is zero or more of the MSG_ options (listed on page 18). The result is a two-element array containing the received data and information on the sender. See the example on page ??.

recvfrom_nonblock

sock.recvfrom_nonblock(*len* ⟨ , *flags* ⟩) → [*data*, *sender*]

Receives up to *len* bytes from *sock* in nonblocking mode. *1.9*

send

sock.send(*string*, *flags*) → *int*
sock.send(*string*, *flags*, *hostname*, *port*) → *int*

The two-parameter form sends *string* on an existing connection. The four-parameter form sends *string* to *port* on *hostname*.

Class

UnixSocket < BasicSocket

A UNIXSocket supports interprocess communication using the Unix domain protocol. Although the underlying protocol supports both datagram and stream connections, the Ruby library provides only a stream-based connection.

```
require 'socket'

SOCKET = "/tmp/sample"

sock = UNIXServer.open(SOCKET)
server_thread = Thread.start(sock) do |sock|      # run server in a thread
  s1 = sock.accept
  p s1.recvfrom(124)
end

client = UNIXSocket.open(SOCKET)
client.send("hello", 0)
client.close

server_thread.join
```

produces:

```
["hello", ["AF_UNIX", ""]]
```

Class methods

new

UNIXSocket.new(*path*) → *sock*

Opens a new domain socket on *path*, which must be a path name.

open

UNIXSocket.open(*path*) → *sock*

Synonym for UNIXSocket.new.

Instance methods

addr

sock.addr → *array*

Returns the address family and path of this socket.

path

sock.path → *string*

Returns the path of this domain socket.

peeraddr

sock.peeraddr → *array*

Returns the address family and path of the server end of the connection.

recvfrom

sock.recvfrom(*len* ⟨ , *flags* ⟩) → *array*

Receives up to *len* bytes from *sock*. *flags* is zero or more of the MSG_ options (listed on page 18). The first element of the returned array is the received data, and the second contains (minimal) information on the sender.

Class UnixServer < UnixSocket

Class UNIXServer provides a simple Unix domain socket server. See UNIXSocket for example code.

Class methods

new UNIXServer.new(*path*) → *sock*

Creates a server on the given *path*. The corresponding file must not exist at the time of the call.

open UNIXServer.open(*path*) → *sock*

Synonym for UNIXServer.new.

Instance methods

accept *sock*.accept → *unix_socket*

Waits for a connection on the server socket and returns a new socket object for that connection. See the example for UNIXSocket on page 34.

Appendix A

Bibliography

[Ste98] W. Richard Stevens. *Unix Network Programming, Volume 1: Networking APIs: Sockets and Xti*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1998.

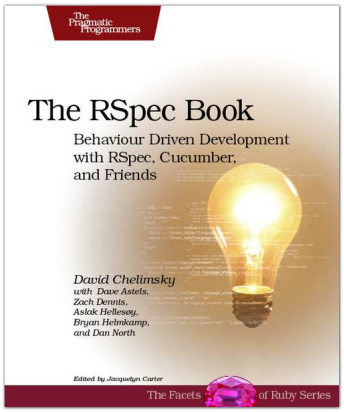
[TFH08] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, third edition, 2008.

More Ruby, More Rails

The RSpec Book

RSpec, Ruby's leading Behaviour Driven Development tool, helps you do TDD right by embracing the design and documentation aspects of TDD. It encourages readable, maintainable suites of code examples that not only test your code, they document it as well. *The RSpec Book* will teach you how to use RSpec, Cucumber, and other Ruby tools to develop truly agile software that gets you to market quickly and maintains its value as evolving market trends drive new requirements.

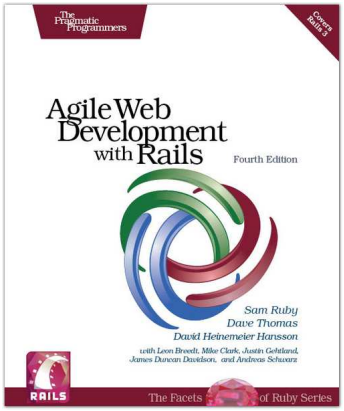
The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends
David Chelimsky, Dave Astels, Zach Dennis, Aslak Hellesøy, Bryan Helmkamp, Dan North
(450 pages) ISBN: 978-1-9343563-7-1. $42.95
http://pragprog.com/titles/achbd



Agile Web Development with Rails

Rails just keeps on changing. Rails 3 and Ruby 1.9 bring hundreds of improvements, including new APIs and substantial performance enhancements. The fourth edition of this award-winning classic has been reorganized and refocused so it's more useful than ever before for developers new to Ruby and Rails. This book isn't just a rework, it's a complete refactoring.

Agile Web Development with Rails: Fourth Edition
Sam Ruby, Dave Thomas, and David Heinemeier Hansson, et al.
(500 pages) ISBN: 978-1-93435-654-8. $43.95
http://pragprog.com/titles/rails4



The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Programming Ruby 1.9
http://pragprog.com/titles/ruby3
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/ruby3.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)