Socket Layer User Guide

$Rev: 1$

An overview of concepts involved in the Socket Layer, and how they relate to API use.

Contents

Bubblephone Limited
Sussex Innovation Centre
Science Park Square
University of Sussex
Brighton BN1 9SB
Tel +44 (0) 1273 704 535
http://www.bubblephone.com/

Sockets

A **socket** is a communication endpoint, an entity that allows processes (possibly on different machines on the network) to send data to each other. On POSIX systems a socket is denoted by a file descriptor (a small integer) and can be used with nearly every system call that uses file descriptors.

Processes allocate a fresh socket using the `socket` call.

```
int
socket(int protocolfamily, int type, int protocol);
```

The `protocolfamily` argument selects the broad class of protocols the socket is for (e.g., `PF_INET` for IPv4, `PF_OSI` for the OSI protocol family). C-THRU sockets are allocated from a family `PF_CTHRU`. Each protocol family would usually have its own kind of format for its network addresses.

The socket `type` argument determines the characteristics of data flow supported by the socket. This could be one of:

> `SOCK_STREAM` provides reliable, sequenced data transfer. Message boundaries are not preserved. These sockets may support out-of-band data transfer.
>
> `SOCK_DGRAM` provides a connectionless unreliable datagram service. Message boundaries are preserved, but individual packets can get lost and/or be delivered multiple times. In addition, each datagram has a maximum length that cannot be exceeded.
>
> `SOCK_SEQPACKET` provides a connection-oriented and reliable datagram service.

The `protocol` argument that specifies a particular protocol in the protocol family (e.g., TCP, UDP, ICMP, IGMP etc.)

If a protocol family of `PF_CTHRU` is selected the socket layer library returns a valid UNIX file descriptor to the client.

Addresses

Once allocated, a socket usually needs to be bound to a local network address. This is achieved using the `bind` function:

```
int
bind(int s, const struct sockaddr *addr, socklen_t addrlen)
```

Network addresses are represented using a generic structure `struct sockaddr` that is large enough to hold all supported kinds of network addresses.

`PF_CTHRU` sockets can use `AF_INET` addresses as well as C-THRU specific `AF_CTHRU` addresses.

The address of the network endpoints associated with a socket may be retrieved using `getpeername` and `getsockname`.

```
int
getpeername(int s, struct sockaddr * restrict name, socklen_t * restrict namelen);

int
getsockname(int s, struct sockaddr * restrict name, socklen_t * restrict namelen);
```

For C-THRU sockets, these return the addresses associated with the respective endpoints. As mentioned above, these addresses could be using `AF_INET` and `AF_CTHRU` addressing.

Connecting

Stream sockets have to be connected to their remote end point before they may be used. This is achieved using the `connect` function:

```
int
connect(int s, const struct sockaddr *remote, socklen_t remotelen);
```

Datagram sockets do not required to be `connect()`ed before they can be used. There is no “disconnect” interface.

Basic I/O

I/O is done using the regular Unix I/O system calls: `read`, `write`, `readv`, and `writev`.

```
ssize_t
read(int d, void *buf, size_t nbytes);

ssize_t
readv(int d, const struct iovec *iov, int iovcnt);

ssize_t
write(int d, const void *buf, size_t nbytes);

ssize_t
writev(int d, const struct iovec *iov, int iovcnt);
```

In addition, there are interfaces tuned for network communication: `send`, `sendto`, `sendmsg`, `recv`, `recvfrom`, and `recvmsg`.

```
ssize_t
recv(int, void *, size_t, int);

ssize_t
recvfrom(int, void *, size_t, int, struct sockaddr * __restrict, socklen_t * __restrict);

ssize_t
recvmsg(int, struct msghdr *, int);

ssize_t
send(int, const void *, size_t, int);

ssize_t
sendto(int, const void *, size_t, int, const struct sockaddr *, socklen_t);

ssize_t
sendmsg(int, const struct msghdr *, int);
```

These functions allow specification of additional metadata (addresses and flags that control the I/O request) along with the data to be transferred. The `recvmsg` and `sendmsg` interfacess provide scatter-gather I/O to and from the caller's address space.

Non-blocking I/O

Sockets may be placed in non-blocking mode using `fcntl(F_SETFL, O_NONBLOCK)`. When in non-blocking mode I/O operations that would otherwise block waiting for data instead return with an error return of `EAGAIN`.

In addition to I/O operations, `connect` and acceptance of incoming stream connections are allowed to proceed asynchronously. Clients use the `select` call to determine if a given socket is readable, writable, if a `connect` request has gone through, or, for `listen()`ing sockets, when an incoming connection is ready to be `accept()`ed.

C-THRU sockets support non-blocking I/O.

Data I/O Options

The `sendto`, `sendmsg`, `recvfrom` and `recvmsg` operations allow the specification of additional flags that control the semantics of the I/O request. These include:

MSG_OOB	Process out-of-band data, not normal data.
MSG_PEEK	Read but do not dequeue incoming data.
MSG_WAITALL	Wait for a full request (or intervening error).
MSG_EOR	The given I/O request completes a record.

Additionally, other systems may support additional flags such as:

MSG_NOSIGNAL	(non POSIX) do not generate signals
MSG_DONTROUTE	(non POSIX) bypass routing for this request
MSG_DONTWAIT	(non POSIX) do not block.

Listening for Incoming Connections

Setting up `SOCK_STREAM` socket to accept incoming connections requires a different set of interfaces to be called. After the socket is allocated using `socket` and a local address is set using `bind`, the socket must be marked as awaiting incoming connections using the `listen` interface.

```
int
listen(int s, int backlog);
```

After this, incoming connections are retrieved using `accept`:

```
int
accept(int s, struct sockaddr * restrict addr, socklen_t * restrict addrlen);
```

`accept` returns to its caller a “fresh” stream socket that is set up to communicate with its remote end point. The original socket continues to accept new incoming connections in the background.

Socket Options

Additional configuration of sockets may be performed using `setsockopt`. The existing configuration may be retrieved using `getsockopt`.

```
int
getsockopt(int s, int level, int optname, void * restrict optval, socklen_t * restrict optlen);

int
setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
```

Sockets support a number of options that control their behaviour:

`SO_BROADCAST`	Allow transmission of broadcast messages.
`SO_DEBUG`	Record debugging information.
`SO_DONTROUTE`	Bypass normal routing.
`SO_KEEPALIVE`	Periodically probe for connectivity.
`SO_LINGER`	Linger on `close()` if data remains to be sent.
`SO_OOBINLINE`	Out-of-band data is transmitted in line.
`SO_RCVBUF`	Configure the receive buffer size.
`SO_RCVLOWAT`	Configure the receive “low water mark”.
`SO_RCVTIMEO`	Configure the timeout when receiving data.
`SO_REUSEADDR`	Allow local address reuse.
`SO_REUSEPORT`	(non POSIX) allow duplicate port and address bindings
`SO_SNDBUF`	Configure the send buffer size.
`SO_SNDLOWAT`	Configure the send “low water mark”.
`SO_SNDTIMEO`	Configure the send timeout.

The `SO_LINGER` option applies to stream sockets being closed that have unsent data queued. It specifies the amount of time the system should keep retrying to send data in order to keep its promise of “reliable” data delivery.

The `SO_RCVBUF` and `SO_SNDBUF` options control the amount of data that may be buffered inside the socket's buffers.

The `SO_SNDLOWAT` option specifies the minimum amount of data that needs to be present in the socket's send queue before an output operation is attempted. The `SO_RCVLOWAT` option specifies that blocking receive calls should wait untill at least the number of bytes specified are available to be read.

The `SO_RCVTIMEO` and `SO_SNDTIMEO` specify timeouts for I/O operations to complete failing which a “short” read or write would be returned.

Additional configuration on sockets is done using the `fcntl` call. The `F_SETOWN` request to `fcntl` specifies a process id or process group id that is to receive a `SIGIO` or `SIGURG` signal when exceptional conditions arise for the socket.

Select

Client applications use `select` to be informed of socket state changes. Such changes include:

- Data being available for reading on the socket.
- Space being available for writing on the socket.
- A `connect()` operation succeeding on a socket that has non-blocking mode set.
- The arrival of an incoming connection on a socket that is `listen()`ing for incoming connections.

C-THRU sockets support most of these options.

Signals

Sockets that support out-of-band data may be configured to signal processes with a `SIGURG` signal when out of band data is available to be read on the socket. The `fcntl(F_SETOWN)` interface is used to set the process id or process group id that will be so signalled.

Some of the socket related interfaces are interruptible by signals sent to the process. When such calls are interrupted the return with an error of `EINTR`. Application code is then expected to either retry or take other kinds of action.

Out of Band Data

Some kinds of protocols may support the concept of delivery of out-of-band data. Such data is usually urgent in character.

Reading a socket will not mix out-of-band data and normal data. Applications may explicitly ask for out-of-band data using the `MSG_OOB` flag with `recvmsg` or `recvfrom` or may ask for out-of-band data to be delivered inline using the `SO_OOBINLINE` socket option.

Application code may use `sockatmark` to determine if a given socket is at an out-of-band mark.

C-THRU sockets do not currently support out-of-band data.

Shutdown

Once their use is over, sockets are shut down using `close` or `shutdown`:

```
int
close(int d);

int
shutdown(int s, int how);
```

Closing a socket that promises reliable delivery of data may wait untill data transmission completes. This could take a long time, but a timeout is configurable using a socket option (see `setsockopt`).