NAME

IO::Socket::IP - Family-neutral IP socket supporting both IPv4 and IPv6

SYNOPSIS

```
use IO::Socket::IP;

my $sock = IO::Socket::IP->new(
PeerHost => "www.google.com",
PeerPort => "http",
Type => SOCK_STREAM,
) or die "Cannot construct socket - $@";

my $familyname = ( $sock->sockdomain == PF_INET6 ) ? "IPv6" :
( $sock->sockdomain == PF_INET ) ? "IPv4" :
"unknown";

printf "Connected to google via %s\n", $familyname;
```

DESCRIPTION

This module provides a protocol-independent way to use IPv4 and IPv6 sockets, intended as a replacement for IO::Socket::INET. Most constructor arguments and methods are provided in a backward-compatible way. For a list of known differences, see the IO::Socket::INET INCOMPATIBILITES section below.

It uses the getaddrinfo(3) function to convert hostnames and service names or port numbers into sets of possible addresses to connect to or listen on. This allows it to work for IPv6 where the system supports it, while still falling back to IPv4-only on systems which don't.

REPLACING

IO::Socket DEFAULT BEHAVIOUR IO::Socket DEFAULT BEHAVIOUR By placing `-register` in the import list, IO::Socket uses IO::Socket::IP rather than IO::Socket::INET as the class that handles `PF_INET`. IO::Socket will also use IO::Socket::IP rather than IO::Socket::INET6 to handle `PF_INET6`, provided that the `AF_INET6` constant is available.

Changing IO::Socket default behaviour means that calling the IO::Socket constructor with either `PF_INET` or `PF_INET6` as the `Domain` parameter will yield an IO::Socket::IP object.

```
use IO::Socket::IP -register;

my $sock = IO::Socket->new(
Domain => PF_INET6,
LocalHost => "::1",
Listen => 1,
) or die "Cannot create socket - $@\n";

print "Created a socket of type " . ref($sock) . "\n";
```

Note that `-register` is a global setting that applies to the entire program; it cannot be applied only for certain callers, removed, or limited by lexical scope.

CONSTRUCTORS

$sock = IO::Socket::IP->new(%args)

Creates a new IO::Socket::IP object, containing a newly created socket handle according to the named arguments passed. The recognised arguments are:

PeerHost => STRING
PeerService => STRING

Hostname and service name for the peer to `connect()` to. The service name may be given as a port number, as a decimal string.

PeerAddr => STRING
PeerPort => STRING
: For symmetry with the accessor methods and compatibility with IO::Socket::INET these are accepted as synonyms for `PeerHost` and `PeerService` respectively.

PeerAddrInfo => ARRAY
: Alternate form of specifying the peer to `connect()` to. This should be an array of the form returned by `Socket::getaddrinfo`

  This parameter takes precedence over the `Peer*`, `Family`, `Type` and `Proto` arguments.

LocalHost => STRING
LocalService => STRING
: Hostname and service name for the local address to `bind()` to.

LocalAddr => STRING
LocalPort => STRING
: For symmetry with the accessor methods and compatibility with IO::Socket::INET these are accepted as synonyms for `LocalHost` and `LocalService` respectively.

LocalAddrInfo => ARRAY
: Alternate form of specifying the local address to `bind()` to. This should be an array of the form returned by `Socket::getaddrinfo`

  This parameter takes precedence over the `Local*`, `Family`, `Type` and `Proto` arguments.

Family => INT
: The address family to pass to `getaddrinfo` (e.g. `AF_INET`, `AF_INET6`). Normally this will be left undefined, and `getaddrinfo` will search using any address family supported by the system.

Type => INT
: The socket type to pass to `getaddrinfo` (e.g. `SOCK_STREAM`, `SOCK_DGRAM`). Normally defined by the caller; if left undefined `getaddrinfo` may attempt to infer the type from the service name.

Proto => STRING or INT
: The IP protocol to use for the socket (e.g. `'tcp'`, `IPPROTO_TCP`, `'udp'`,`IPPROTO_UDP`). Normally this will be left undefined, and either `getaddrinfo` or the kernel will choose an appropriate value. May be given either in string name or numeric form.

GetAddrInfoFlags => INT
: More flags to pass to the `getaddrinfo()` function. If not supplied, a default of `AI_ADDRCONFIG` will be used.

  These flags will be combined with `AI_PASSIVE` if the `Listen` argument is given. For more information see the documentation about `getaddrinfo()` in the Socket module.

Listen => INT
: If defined, puts the socket into listening mode where new connections can be accepted using the `accept` method. The value given is used as the listen(2) queue size.

ReuseAddr => BOOL
: If true, set the `SO_REUSEADDR` sockopt

ReusePort => BOOL
: If true, set the `SO_REUSEPORT` sockopt (not all OSes implement this sockopt)

Broadcast => BOOL
: If true, set the `SO_BROADCAST` sockopt

V6Only => BOOL
: If defined, set the `IPV6_V6ONLY` sockopt when creating `PF_INET6` sockets to the given value. If true, a listening-mode socket will only listen on the `AF_INET6` addresses; if false

it will also accept connections from `AF_INET` addresses.

If not defined, the socket option will not be changed, and default value set by the operating system will apply. For repeatable behaviour across platforms it is recommended this value always be defined for listening-mode sockets.

Note that not all platforms support disabling this option. Some, at least OpenBSD and MirBSD, will fail with `EINVAL` if you attempt to disable it. To determine whether it is possible to disable, you may use the class method

```
if( IO::Socket::IP->CAN_DISABLE_V6ONLY ) {
...
}
else {
...
}
```

If your platform does not support disabling this option but you still want to listen for both `AF_INET` and `AF_INET6` connections you will have to create two listening sockets, one bound to each protocol.

MultiHomed
: This `IO::Socket::INET` argument is ignored, except if it is defined but false. See the `IO::Socket::INET` INCOMPATIBILITES section below.

  However, the behaviour it enables is always performed by `IO::Socket::IP`

Blocking => BOOL
: If defined but false, the socket will be set to non-blocking mode. Otherwise it will default to blocking mode. See the NON-BLOCKING section below for more detail.

If neither `Type` nor `Proto` hints are provided, a default of `SOCK_STREAM` and `IPPROTO_TCP` respectively will be set, to maintain compatibility with `IO::Socket::INET` Other named arguments that are not recognised are ignored.

If neither `Family` nor any hosts or addresses are passed, nor any `*AddrInfo`, then the constructor has no information on which to decide a socket family to create. In this case, it performs a `getaddinfo` call with the `AI_ADDRCONFIG` flag, no host name, and a service name of `"0"`, and uses the family of the first returned result.

If the constructor fails, it will set `$@` to an appropriate error message; this may be from `$!` or it may be some other string; not every failure necessarily has an associated `errno` value.

`$sock` = IO::Socket::IP->new(`$peeraddr`)
: As a special case, if the constructor is passed a single argument (as opposed to an even-sized list of key/value pairs), it is taken to be the value of the `PeerAddr` parameter. This is parsed in the same way, according to the behaviour given in the `PeerHost` AND `LocalHost` PARSING section below.

METHODS

As well as the following methods, this class inherits all the methods in IO::Socket and IO::Handle.

(`$host`, `$service`) = `$sock`->sockhost_service(`$numeric`)
: Returns the hostname and service name of the local address (that is, the socket address given by the `sockname` method).

  If `$numeric` is true, these will be given in numeric form rather than being resolved into names.

  The following four convenience wrappers may be used to obtain one of the two values returned here. If both host and service names are required, this method is preferable to the following wrappers, because it will call `getnameinfo(3)` only once.

`$addr = $sock->sockhost`
: Return the numeric form of the local address as a textual representation

`$port = $sock->sockport`
: Return the numeric form of the local port number

`$host = $sock->sockhostname`
: Return the resolved name of the local address

`$service = $sock->sockservice`
: Return the resolved name of the local port number

`$addr = $sock->sockaddr`
: Return the local address as a binary octet string

`( $host, $service ) = $sock->peerhost_service( $numeric )`
: Returns the hostname and service name of the peer address (that is, the socket address given by the `peername` method), similar to the `sockhost_service` method.

  The following four convenience wrappers may be used to obtain one of the two values returned here. If both host and service names are required, this method is preferable to the following wrappers, because it will call `getnameinfo(3)` only once.

`$addr = $sock->peerhost`
: Return the numeric form of the peer address as a textual representation

`$port = $sock->peerport`
: Return the numeric form of the peer port number

`$host = $sock->peerhostname`
: Return the resolved name of the peer address

`$service = $sock->peerservice`
: Return the resolved name of the peer port number

`$addr = $peer->peeraddr`
: Return the peer address as a binary octet string

`$inet = $sock->as_inet`
: Returns a new IO::Socket::INET instance wrapping the same filehandle. This may be useful in cases where it is required, for backward-compatibility, to have a real object of `IO::Socket::INET` type instead of `IO::Socket::IP` The new object will wrap the same underlying socket filehandle as the original, so care should be taken not to continue to use both objects concurrently. Ideally the original `$sock` should be discarded after this method is called.

  This method checks that the socket domain is `PF_INET` and will throw an exception if it isn't.

NON-BLOCKING

If the constructor is passed a defined but false value for the `Blocking` argument then the socket is put into non-blocking mode. When in non-blocking mode, the socket will not be set up by the time the constructor returns, because the underlying `connect(2)` syscall would otherwise have to block.

The non-blocking behaviour is an extension of the `IO::Socket::INET` API, unique to `IO::Socket::IP` because the former does not support multi-homed non-blocking connect.

When using non-blocking mode, the caller must repeatedly check for writeability on the filehandle (for instance using `select` or `IO::Poll` Each time the filehandle is ready to write, the `connect` method must be called, with no arguments. Note that some operating systems, most notably `MSWin32` do not report a `connect()` failure using write-ready; so you must also `select()` for exceptional status.

While `connect` returns false, the value of `$!` indicates whether it should be tried again (by being set to the value `EINPROGRESS`, or `EWOULDBLOCK` on MSWin32), or whether a permanent error has occurred (e.g. `ECONNREFUSED`).

Once the socket has been connected to the peer, `connect` will return true and the socket will now be ready to use.

Note that calls to the platform's underlying `getaddrinfo(3)` function may block. If `IO::Socket::IP` has to perform this lookup, the constructor will block even when in non-blocking mode.

To avoid this blocking behaviour, the caller should pass in the result of such a lookup using the `PeerAddrInfo` or `LocalAddrInfo` arguments. This can be achieved by using Net::LibAsyncNS, or the `getaddrinfo(3)` function can be called in a child process.

```
use IO::Socket::IP;
use Errno qw( EINPROGRESS EWOULDBLOCK );

my @peeraddrinfo = ... # Caller must obtain the getaddinfo result here

my $socket = IO::Socket::IP->new(
PeerAddrInfo => \@peeraddrinfo,
Blocking => 0,
) or die "Cannot construct socket - $@";

while( !$socket->connect and ( $! == EINPROGRESS || $! == EWOULDBLOCK ) ) {
my $wvec = '';
vec( $wvec, fileno $socket, 1 ) = 1;
my $evec = '';
vec( $evec, fileno $socket, 1 ) = 1;

select( undef, $wvec, $evec, undef ) or die "Cannot select - $!";
}

die "Cannot connect - $!" if $!;

...
```

The example above uses `select()`, but any similar mechanism should work analogously. `IO::Socket::IP` takes care when creating new socket filehandles to preserve the actual file descriptor number, so such techniques as `poll` or `epoll` should be transparent to its reallocation of a different socket underneath, perhaps in order to switch protocol family between `PF_INET` and `PF_INET6`.

For another example using `IO::Poll` and `Net::LibAsyncNS` see the *examples/nonblocking_libasyncns.pl* file in the module distribution.

`PeerHost` AND `LocalHost` PARSING

To support the `IO::Socket::INET` API, the host and port information may be passed in a single string rather than as two separate arguments.

If either `LocalHost` or `PeerHost` (or their `...Addr` synonyms) have any of the following special forms then special parsing is applied.

The value of the `...Host` argument will be split to give both the hostname and port (or service name):

```
hostname.example.org:http # Host name
192.0.2.1:80 # IPv4 address
[2001:db8::1]:80 # IPv6 address
```

In each case, the port or service name (e.g. `80`) is passed as the `LocalService` or `PeerService` argument.

Either of `LocalService` or `PeerService` (or their `...Port` synonyms) can be either a service

name, a decimal number, or a string containing both a service name and number, in a form such as

```
http(80)
```

In this case, the name (`http`) will be tried first, but if the resolver does not understand it then the port number (`80`) will be used instead.

If the `...Host` argument is in this special form and the corresponding `...Service` or `...Port` argument is also defined, the one parsed from the `...Host` argument will take precedence and the other will be ignored.

($host, $port) = IO::Socket::IP->split_addr($addr)

Utility method that provides the parsing functionality described above. Returns a 2-element list, containing either the split hostname and port description if it could be parsed, or the given address and `undef` if it was not recognised.

```
IO::Socket::IP->split_addr( "hostname:http" )
# ( "hostname", "http" )

IO::Socket::IP->split_addr( "192.0.2.1:80" )
# ( "192.0.2.1", "80" )

IO::Socket::IP->split_addr( "[2001:db8::1]:80" )
# ( "2001:db8::1", "80" )

IO::Socket::IP->split_addr( "something.else" )
# ( "something.else", undef )
```

$addr = IO::Socket::IP->join_addr($host, $port)

Utility method that performs the reverse of `split_addr`, returning a string formed by joining the specified host address and port number. The host address will be wrapped in `[]` brackets if required (because it is a raw IPv6 numeric address).

This can be especially useful when combined with the `sockhost_service` or `peerhost_service` methods.

```
say "Connected to ", IO::Socket::IP->join_addr( $sock->peerhost_service );
```

IO::Socket::INET INCOMPATIBILITES

- The behaviour enabled by `MultiHomed` is in fact implemented by `IO::Socket::IP` as it is required to correctly support searching for a useable address from the results of the `getaddrinfo(3)` call. The constructor will ignore the value of this argument, except if it is defined but false. An exception is thrown in this case, because that would request it disable the `getaddrinfo(3)` search behaviour in the first place.

TODO

- Investigate whether `POSIX::dup2` upsets BSD's `kqueue` watchers, and if so, consider what possible workarounds might be applied.

AUTHOR

Paul Evans <leonerd@leonerd.org.uk>