NAME

AnyEvent::Socket – useful IPv4 and IPv6 stuff. also unix domain sockets. and stuff.

SYNOPSIS

```
use AnyEvent::Socket;

tcp_connect "gameserver.deliantra.net", 13327, sub {
   my ($fh) = @_
      or die "gameserver.deliantra.net connect failed: $!";

   # enjoy your filehandle
};

# a simple tcp server
tcp_server undef, 8888, sub {
   my ($fh, $host, $port) = @_;

   syswrite $fh, "The internet is full, $host:$port. Go away!\015\012";
};
```

DESCRIPTION

This module implements various utility functions for handling internet protocol addresses and sockets, in an as transparent and simple way as possible.

All functions documented without `AnyEvent::Socket::` prefix are exported by default.

`$ipn` = parse_ipv4 `$dotted_quad`

Tries to parse the given dotted quad IPv4 address and return it in octet form (or undef when it isn't in a parsable format). Supports all forms specified by POSIX (e.g. `10.0.0.1`, `10.1`, `10.0x020304`, `0x12345678` or `0377.0377.0377.0377`).

`$ipn` = parse_ipv6 `$textual_ipv6_address`

Tries to parse the given IPv6 address and return it in octet form (or undef when it isn't in a parsable format).

Should support all forms specified by RFC 2373 (and additionally all IPv4 forms supported by parse_ipv4). Note that scope-id's are not supported (and will not parse).

This function works similarly to `inet_pton AF_INET6, ...`.

Example:

```
print unpack "H*", parse_ipv6 "2002:5345::10.0.0.1";
# => 2002534500000000000000000a000001

print unpack "H*", parse_ipv6 "192.89.98.1";
# => 00000000000000000000ffffc0596201
```

`$token` = parse_unix `$hostname`

This function exists mainly for symmetry to the other `parse_protocol` functions – it takes a hostname and, if it is `unix/`, it returns a special address token, otherwise `undef`.

The only use for this function is probably to detect whether a hostname matches whatever AnyEvent uses for unix domain sockets.

`$ipn` = parse_address `$ip`

Combines `parse_ipv4`, `parse_ipv6` and `parse_unix` in one function. The address here refers to the host address (not socket address) in network form (binary).

If the `$text` is `unix/`, then this function returns a special token recognised by the other functions in this module to mean ''UNIX domain socket''.

If the `$text` to parse is a plain IPv4 or mapped IPv4 in IPv6 address (:ffff::<ipv4>), then it will be treated as an IPv4 address and four octets will be returned. If you don't want that, you have to call `parse_ipv4` and/or `parse_ipv6` manually (the latter always returning a 16 octet IPv6 address for mapped IPv4 addresses).

Example:

```
print unpack "H*", parse_address "10.1.2.3";
# => 0a010203
```

`$ipn` = AnyEvent::Socket::aton `$ip`

Same as `parse_address`, but not exported (think `Socket::inet_aton` but *without* name resolution).

($name, `$aliases`, `$proto`) = getprotobyname `$name`

Works like the builtin function of the same name, except it tries hard to work even on broken platforms (well, that's windows), where getprotobyname is traditionally very unreliable.

Example: get the protocol number for TCP (usually 6)

```
my $proto = getprotobyname "tcp";
```

($host, `$service`) = parse_hostport `$string`[, `$default_service`]

Splitting a string of the form `hostname:port` is a common problem. Unfortunately, just splitting on the colon makes it hard to specify IPv6 addresses and doesn't support the less common but well standardised `[ip literal]` syntax.

This function tries to do this job in a better way, it supports (at least) the following formats, where `port` can be a numerical port number of a service name, or a `name=port` string, and the `port` and `:port` parts are optional. Also, everywhere where an IP address is supported a hostname or unix domain socket address is also supported (see `parse_unix`), and strings starting with `/` will also be interpreted as unix domain sockets.

```
hostname:port    e.g. "www.linux.org", "www.x.de:443", "www.x.de:https=443"
ipv4:port        e.g. "198.182.196.56", "127.1:22"
ipv6             e.g. "::1", "affe::1"
[ipv4or6]:port   e.g. "[::1]", "[10.0.1]:80"
[ipv4or6] port   e.g. "[127.0.0.1]", "[www.x.org] 17"
ipv4or6 port     e.g. "::1 443", "10.0.0.1 smtp"
unix/:path       e.g. "unix/:/path/to/socket"
/path            e.g. "/path/to/socket"
```

It also supports defaulting the service name in a simple way by using `$default_service` if no service was detected. If neither a service was detected nor a default was specified, then this function returns the empty list. The same happens when a parse error was detected, such as a hostname with a colon in it (the function is rather forgiving, though).

Example:

```
print join ",", parse_hostport "localhost:443";
# => "localhost,443"

print join ",", parse_hostport "localhost", "https";
# => "localhost,https"

print join ",", parse_hostport "[::1]";
# => "," (empty list)

print join ",", parse_hostport "/tmp/debug.sock";
# => "unix/", "/tmp/debug.sock"
```

`$string` = format_hostport `$host`, `$port`
: Takes a host (in textual form) and a port and formats in unambigiously in a way that `parse_hostport` can parse it again. `$port` can be `undef`.

`$sa_family` = address_family `$ipn`
: Returns the address family/protocol−family (AF_xxx/PF_xxx, in one value :) of the given host address in network format.

`$text` = format_ipv4 `$ipn`
: Expects a four octet string representing a binary IPv4 address and returns its textual format. Rarely used, see `format_address` for a nicer interface.

`$text` = format_ipv6 `$ipn`
: Expects a sixteen octet string representing a binary IPv6 address and returns its textual format. Rarely used, see `format_address` for a nicer interface.

`$text` = format_address `$ipn`
: Covnvert a host address in network format (e.g. 4 octets for IPv4 or 16 octets for IPv6) and convert it into textual form.

  Returns `unix/` for UNIX domain sockets.

  This function works similarly to `inet_ntop AF_INET || AF_INET6, ...`, except it automatically detects the address type.

  Returns `undef` if it cannot detect the type.

  If the `$ipn` is a mapped IPv4 in IPv6 address (:ffff::<ipv4>), then just the contained IPv4 address will be returned. If you do not want that, you have to call `format_ipv6` manually.

  Example:

  ```
  print format_address "\x01\x02\x03\x05";
  => 1.2.3.5
  ```

`$text` = AnyEvent::Socket::ntoa `$ipn`
: Same as format_address, but not exported (think `inet_ntoa`).

inet_aton `$name_or_address`, `$cb->`(@addresses)
: Works similarly to its Socket counterpart, except that it uses a callback. Use the length to distinguish between ipv4 and ipv6 (4 octets for IPv4, 16 for IPv6), or use `format_address` to convert it to a more readable format.

  Note that `resolve_sockaddr`, while initially a more complex interface, resolves host addresses, IDNs, service names and SRV records and gives you an ordered list of socket addresses to try and should be preferred over `inet_aton`.

  Example.

  ```
  inet_aton "www.google.com", my $cv = AE::cv;
  say unpack "H*", $_
     for $cv->recv;
  # => d155e363
  # => d155e367 etc.

  inet_aton "ipv6.google.com", my $cv = AE::cv;
  say unpack "H*", $_
     for $cv->recv;
  # => 20014860a003000000000000000000068
  ```

`$sa` = AnyEvent::Socket::pack_sockaddr `$service`, `$host`
: Pack the given port/host combination into a binary sockaddr structure. Handles both IPv4 and IPv6 host addresses, as well as UNIX domain sockets (`$host == unix/` and `$service ==` absolute

pathname).

Example:

```
my $bind = AnyEvent::Socket::pack_sockaddr 43, v195.234.53.120;
bind $socket, $bind
   or die "bind: $!";
```

($service, $host) = AnyEvent::Socket::unpack_sockaddr $sa

Unpack the given binary sockaddr structure (as used by bind, getpeername etc.) into a $service, $host combination.

For IPv4 and IPv6, $service is the port number and $host the host address in network format (binary).

For UNIX domain sockets, $service is the absolute pathname and $host is a special token that is understood by the other functions in this module (format_address converts it to unix/).

AnyEvent::Socket::resolve_sockaddr $node, $service, $proto, $family, $type, $cb->([$family, $type, $proto, $sockaddr], ...)

Tries to resolve the given nodename and service name into protocol families and sockaddr structures usable to connect to this node and service in a protocol-independent way. It works remotely similar to the getaddrinfo posix function.

For internet addresses, $node is either an IPv4 or IPv6 address, an internet hostname (DNS domain name or IDN), and $service is either a service name (port name from */etc/services*) or a numerical port number. If both $node and $service are names, then SRV records will be consulted to find the real service, otherwise they will be used as-is. If you know that the service name is not in your services database, then you can specify the service in the format name=port (e.g. http=80).

If a host cannot be found via DNS, then it will be looked up in */etc/hosts* (or the file specified via $ENV{PERL_ANYEVENT_HOSTS}). If they are found, the addresses there will be used. The effect is as if entries from */etc/hosts* would yield A and AAAA records for the host name unless DNS already had records for them.

For UNIX domain sockets, $node must be the string unix/ and $service must be the absolute pathname of the socket. In this case, $proto will be ignored.

$proto must be a protocol name, currently tcp, udp or sctp. The default is currently tcp, but in the future, this function might try to use other protocols such as sctp, depending on the socket type and any SRV records it might find.

$family must be either 0 (meaning any protocol is OK), 4 (use only IPv4) or 6 (use only IPv6). The default is influenced by $ENV{PERL_ANYEVENT_PROTOCOLS}.

$type must be SOCK_STREAM, SOCK_DGRAM or SOCK_SEQPACKET (or undef in which case it gets automatically chosen to be SOCK_STREAM unless $proto is udp).

The callback will receive zero or more array references that contain $family, $type, $proto for use in socket and a binary $sockaddr for use in connect (or bind).

The application should try these in the order given.

Example:

```
resolve_sockaddr "google.com", "http", 0, undef, undef, sub { ... };
```

$guard = tcp_connect $host, $service, $connect_cb[, $prepare_cb]

This is a convenience function that creates a TCP socket and makes a 100% non-blocking connect to the given $host (which can be a DNS/IDN hostname or a textual IP address, or the string unix/ for UNIX domain sockets) and $service (which can be a numeric port number or a service name, or a servicename=portnumber string, or the pathname to a UNIX domain socket).

If both $host and $port are names, then this function will use SRV records to locate the real

target(s).

In either case, it will create a list of target hosts (e.g. for multihomed hosts or hosts with both IPv4 and IPv6 addresses) and try to connect to each in turn.

After the connection is established, then the `$connect_cb` will be invoked with the socket file handle (in non-blocking mode) as first, and the peer host (as a textual IP address) and peer port as second and third arguments, respectively. The fourth argument is a code reference that you can call if, for some reason, you don't like this connection, which will cause `tcp_connect` to try the next one (or call your callback without any arguments if there are no more connections). In most cases, you can simply ignore this argument.

```
$cb->($filehandle, $host, $port, $retry)
```

If the connect is unsuccessful, then the `$connect_cb` will be invoked without any arguments and `$!` will be set appropriately (with `ENXIO` indicating a DNS resolution failure).

The callback will *never* be invoked before `tcp_connect` returns, even if `tcp_connect` was able to connect immediately (e.g. on unix domain sockets).

The file handle is perfect for being plugged into AnyEvent::Handle, but can be used as a normal perl file handle as well.

Unless called in void context, `tcp_connect` returns a guard object that will automatically cancel the connection attempt when it gets destroyed – in which case the callback will not be invoked. Destroying it does not do anything to the socket after the connect was successful – you cannot ''uncall'' a callback that has been invoked already.

Sometimes you need to ''prepare'' the socket before connecting, for example, to `bind` it to some port, or you want a specific connect timeout that is lower than your kernel's default timeout. In this case you can specify a second callback, `$prepare_cb`. It will be called with the file handle in not-yet-connected state as only argument and must return the connection timeout value (or `0`, `undef` or the empty list to indicate the default timeout is to be used).

Note to the poor Microsoft Windows users: Windows (of course) doesn't correctly signal connection errors, so unless your event library works around this, failed connections will simply hang. The only event libraries that handle this condition correctly are EV and Glib. Additionally, AnyEvent works around this bug with Event and in its pure-perl backend. All other libraries cannot correctly handle this condition. To lessen the impact of this windows bug, a default timeout of 30 seconds will be imposed on windows. Cygwin is not affected.

Simple Example: connect to localhost on port 22.

```
tcp_connect localhost => 22, sub {
   my $fh = shift
      or die "unable to connect: $!";
   # do something
};
```

Complex Example: connect to www.google.com on port 80 and make a simple GET request without much error handling. Also limit the connection timeout to 15 seconds.

```
tcp_connect "www.google.com", "http",
   sub {
      my ($fh) = @_
         or die "unable to connect: $!";

      my $handle; # avoid direct assignment so on_eof has it in scope.
      $handle = new AnyEvent::Handle
         fh     => $fh,
         on_error => sub {
```

```
                  AE::log error => $_[2];
                  $_[0]->destroy;
               },
               on_eof => sub {
                  $handle->destroy; # destroy handle
                  AE::log info => "Done.";
               };

            $handle->push_write ("GET / HTTP/1.0\015\012\015\012");

            $handle->push_read (line => "\015\012\015\012", sub {
               my ($handle, $line) = @_;

               # print response header
               print "HEADER\n$line\n\nBODY\n";

               $handle->on_read (sub {
                  # print response body
                  print $_[0]->rbuf;
                  $_[0]->rbuf = "";
               });
            });
         }, sub {
            my ($fh) = @_;
            # could call $fh->bind etc. here

            15
         };
```

Example: connect to a UNIX domain socket.

```
   tcp_connect "unix/", "/tmp/.X11-unix/X0", sub {
      ...
   }
```

`$guard` = tcp_server `$host`, `$service`, `$accept_cb`[, `$prepare_cb`]

Create and bind a stream socket to the given host address and port, set the SO_REUSEADDR flag (if applicable) and call `listen`. Unlike the name implies, this function can also bind on UNIX domain sockets.

For internet sockets, `$host` must be an IPv4 or IPv6 address (or `undef`, in which case it binds either to `0` or to `::`, depending on whether IPv4 or IPv6 is the preferred protocol, and maybe to both in future versions, as applicable).

To bind to the IPv4 wildcard address, use `0`, to bind to the IPv6 wildcard address, use `::`.

The port is specified by `$service`, which must be either a service name or a numeric port number (or `0` or `undef`, in which case an ephemeral port will be used).

For UNIX domain sockets, `$host` must be `unix/` and `$service` must be the absolute pathname of the socket. This function will try to `unlink` the socket before it tries to bind to it, and will try to unlink it after it stops using it. See SECURITY CONSIDERATIONS, below.

For each new connection that could be `accept`ed, call the `$accept_cb->($fh, $host, $port)` with the file handle (in non-blocking mode) as first, and the peer host and port as second and third arguments (see `tcp_connect` for details).

Croaks on any errors it can detect before the listen.

In non-void context, this function returns a guard object whose lifetime it tied to the TCP server: If the

object gets destroyed, the server will be stopped and the listening socket will be cleaned up/unlinked (already accepted connections will not be affected).

When called in void-context, AnyEvent will keep the listening socket alive internally. In this case, there is no guarantee that the listening socket will be cleaned up or unlinked.

In all cases, when the function returns to the caller, the socket is bound and in listening state.

If you need more control over the listening socket, you can provide a `$prepare_cb->($fh, $host, $port)`, which is called just before the `listen ()` call, with the listen file handle as first argument, and IP address and port number of the local socket endpoint as second and third arguments.

It should return the length of the listen queue (or `0` for the default).

Note to IPv6 users: RFC-compliant behaviour for IPv6 sockets listening on `::` is to bind to both IPv6 and IPv4 addresses by default on dual-stack hosts. Unfortunately, only GNU/Linux seems to implement this properly, so if you want both IPv4 and IPv6 listening sockets you should create the IPv6 socket first and then attempt to bind on the IPv4 socket, but ignore any `EADDRINUSE` errors.

Example: bind on some TCP port on the local machine and tell each client to go away.

```
tcp_server undef, undef, sub {
   my ($fh, $host, $port) = @_;

   syswrite $fh, "The internet is full, $host:$port. Go away!\015\012";
}, sub {
   my ($fh, $thishost, $thisport) = @_;
   AE::log info => "Bound to $thishost, port $thisport.";
};
```

Example: bind a server on a unix domain socket.

```
tcp_server "unix/", "/tmp/mydir/mysocket", sub {
   my ($fh) = @_;
};
```

`$guard` = AnyEvent::Socket::tcp_bind `$host`, `$service`, `$done_cb`[, `$prepare_cb`]

Same as `tcp_server`, except it doesn't call `accept` in a loop for you but simply passes the listen socket to the `$done_cb`. This is useful when you want to have a convenient set up for your listen socket, but want to do the `accept`'ing yourself, for example, in another process.

In case of an error, `tcp_bind` either croaks, or passes `undef` to the `$done_cb`.

In non-void context, a guard will be returned. It will clean up/unlink the listening socket when destroyed. In void context, no automatic clean up might be performed.

tcp_nodelay `$fh`, `$enable`

Enables (or disables) the `TCP_NODELAY` socket option (also known as Nagle's algorithm). Returns false on error, true otherwise.

tcp_congestion `$fh`, `$algorithm`

Sets the tcp congestion avoidance algorithm (via the `TCP_CONGESTION` socket option). The default is OS-specific, but is usually `reno`. Typical other available choices include `cubic`, `lp`, `bic`, `highspeed`, `htcp`, `hybla`, `illinois`, `scalable`, `vegas`, `veno`, `westwood` and `yeah`.

SECURITY CONSIDERATIONS

This module is quite powerful, with with power comes the ability to abuse as well: If you accept "hostnames" and ports from untrusted sources, then note that this can be abused to delete files (host=`unix/`). This is not really a problem with this module, however, as blindly accepting any address and protocol and trying to bind a server or connect to it is harmful in general.

AUTHOR

Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de