Are you ready for IPv6?

Chris Kohlhoff

Outline:

- IPv4 vs IPv6 – cheat sheet for developers
- IPv6 support in Boost.Asio
- Writing client programs
- Writing server programs

IPv4 vs IPv6 Basics

Address – binary representation

IPv4	IPv6
32 bits	128 bits

Address – string representation

IPv4	IPv6
Dotted decimal address 192.168.1.14	Hex address fec0:0:0:0:129a:ddff:fea7:529e

Address – string representation

IPv4	IPv6
Dotted decimal address 192.168.1.14	Hex address fec0::129a:ddff:fea7:529e

At most one run of 0s may be “compressed”

Endpoint – string representation

IPv4	IPv6
192.168.1.14:12345	[fec0::129a:ddff:fea7:529e]:12345

Loopback address

IPv4	IPv6
127.0.0.1	::1

Any address

IPv4	IPv6
0.0.0.0	::

Sometimes you might also see 0::0

IPv6 Support in Boost.Asio

Principles:

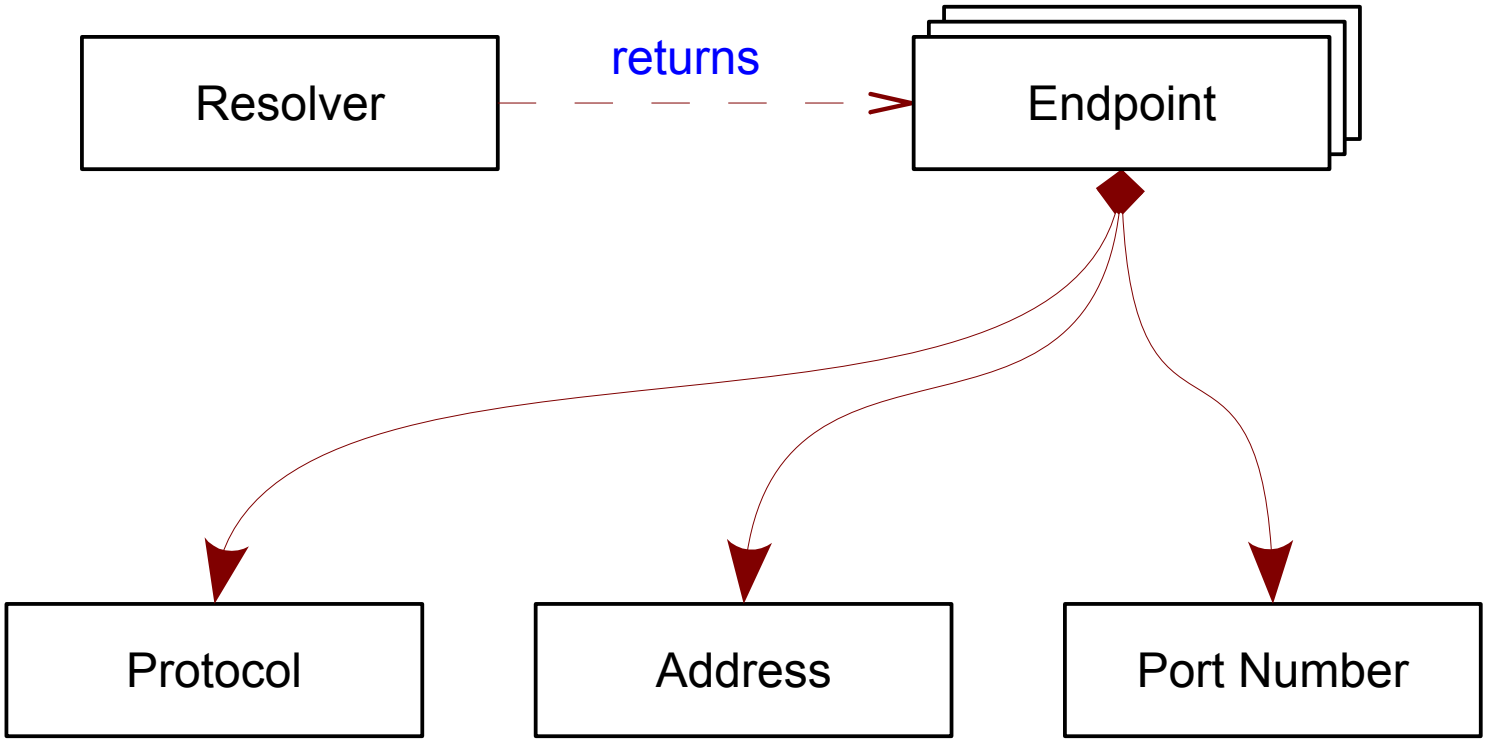
- Keep most user code independent of IP version
- Avoid imposing dynamic memory allocation

Principles:

- Keep most user code independent of IP version
- Avoid imposing dynamic memory allocation

→ Use a variant approach

IPv6 Support in Boost.Asio



Protocol:

- For TCP, use `asio::ip::tcp`
- For UDP, use `asio::ip::udp`

```
udp my_protocol = udp::v4();
```

or

```
udp my_protocol = udp::v6();
```

```
udp my_protocol = udp::v4();
```

or

```
udp my_protocol = udp::v6();

// ...

udp::socket sock(io_service, my_protocol);
```

```
udp my_protocol = udp::v4();
```

or

```
udp my_protocol = udp::v6();

// ...

udp::socket sock(io_service);
sock.open(my_protocol);
```

```
udp my_protocol = udp::v4();
```

or

```
udp my_protocol = udp::v6();

// ...

if (my_protocol == udp::v6())
  ...
```

Address:

- Protocol independent – `asio::ip::address`
- IPv4 – `asio::ip::address_v4`
- IPv6 – `asio::ip::address_v6`

```
const char* ip_address_string = ...;
address my_address =
    address::from_string(ip_address_string);

// ...

std::string s = my_address.to_string();
```

IPv4 dotted decimal
or IPv6 hex address

```
address my_address = address_v4::loopback();

// ...

address my_address = address_v6::loopback();
```

```
address my_address = address_v4::any();

// ...

address my_address = address_v6::any();
```

```
if (my_address.is_loopback())
    ...

if (my_address.is_unspecified())
    ...

if (my_address.is_multicast())
    ...
```

New in Boost 1.47

```
address_v4 my_address_v4 = ...;
address my_address = my_address_v4;


// ...


if (my_address.is_v4())
{
    address_v4 a = my_address.to_v4();
    ...
}
```

```
address_v6 my_address_v6 = ...;
address my_address = my_address_v6;


// ...


if (my_address.is_v6())
{
    address_v6 a = my_address.to_v6();
    ...
}
```

```
const char* ip_address_string = ...;
address_v6 my_address_v6 =
    address_v6::from_string(ip_address_string);


// ...


std::string s = my_address_v6.to_string();
```

IPv6 hex address only

```
address_v6::bytes_type raw_address_bytes;
address_v6 my_address_v6(raw_address_bytes);

// ...

address_v6::bytes_type bytes =
    my_address_v6.to_bytes();
```

array<unsigned char, 16>

```
address_v4 my_address_v4 = ...;
address_v6 my_address_v6 =
    address_v6::v4_mapped(my_address_v4);


// ...


if (my_address_v6.is_v4_mapped())
{
    address_v4 a = my_address_v6.to_v4();
}
```

```
address_v4 my_address_v4 = ...;
address_v6 my_address_v6 =
    address_v6::v4_compatible(my_address_v4);


// ...


if (my_address_v6.is_v4_compatible())
{
    address_v4 a = my_address_v6.to_v4();
}
```

Endpoint:

- Always protocol independent
- For TCP – `asio::ip::tcp::endpoint`
- For UDP – `asio::ip::udp::endpoint`

Endpoint:

- Always protocol independent
- For TCP – `asio::ip::tcp::endpoint`
- For UDP – `asio::ip::udp::endpoint`

Implemented as:

```
template <typename InternetProtocol>
class basic_endpoint;
```

```
udp::endpoint my_endpoint1(udp::v4(), 12345);

// ...

udp::endpoint my_endpoint2(udp::v6(), 12345);

// ...

address my_address = ...;
udp::endpoint my_endpoint3(my_address, 12345);
```

```
udp::endpoint my_endpoint = ...;

// ...

address my_address = my_endpoint.address();

// ...

unsigned short my_port = my_endpoint.port();

// ...

udp my_protocol = my_endpoint.protocol();
```

Resolver:

- Obtain endpoints corresponding to host and service names
- Usually uses DNS

IPv6 Support in Boost.Asio

```
udp::resolver::query my_query("host.name", "daytime");

// ...

udp::resolver resolver(io_service);
udp::resolver::iterator iter = resolver.resolve(my_query), end;

while (iter != end)
{
    cout << iter->endpoint() << std::endl;
    ++iter;
}
```

One query can resolve to both IPv4 and IPv6 endpoints

IPv6 Support in Boost.Asio

```
udp::resolver::query my_query1("host.name", "daytime");

// ...

udp::resolver::query my_query2(udp::v4(), "host.name", "daytime");

// ...

udp::resolver::query my_query3(udp::v6(), "host.name", "daytime");
```

Socket options:

- Mostly protocol independent
- Automatically applies correct option based on socket's protocol

```
udp::socket sock(io_service, my_protocol);

// ...

sock.set_option(ip::unicast::hops(128));
```

For IPv4, uses IPPROTO_IP/IP_TTL

For IPv6, uses IPPROTO_IPV6/IPV6_UNICAST_HOPS

Writing Client Programs

Approaches for TCP clients:

- Keep source code “protocol independent”
- Prefer endpoints over addresses
- Attempt connection to all available hosts

Writing Client Programs

```
tcp::resolver resolver(io_service);
tcp::resolver::query q("www.boost.org", "http");
tcp::resolver::iterator iter = resolver.resolve(q), end;
```

One query can resolve
to multiple endpoints

Writing Client Programs

```
tcp::resolver resolver(io_service);
tcp::resolver::query q("www.boost.org", "http");
tcp::resolver::iterator iter = resolver.resolve(q), end;
```

```
tcp::socket sock(io_service);
error_code ec = asio::error::not_found;
while (iter != end && ec)
{
    sock.close(ec);
    sock.connect(*iter, ec);
    ++iter;
}
if (ec) throw system_error(ec);
```

Try each endpoint
until one works

Writing Client Programs

```
tcp::resolver resolver(io_service);
tcp::resolver::query q("www.boost.org", "http");
```

```
tcp::socket sock(io_service);
asio::connect(sock, resolver.resolve(q));
```

New in Boost 1.47

Writing Client Programs

```
tcp::resolver resolver(io_service);
tcp::resolver::query q("www.boost.org", "http");
vector<tcp::endpoint> endpoints(
    resolver.resolve(q),
    tcp::resolver::iterator());
stable_partition(endpoints.begin(), endpoints.end(), is_ipv4);

tcp::socket sock(io_service);
error_code ec = asio::error::not_found;
for (vector<tcp::endpoint>::iterator iter = endpoints.begin();
    iter != endpoints.end() && ec; ++iter)
{
    sock.close(ec);
    sock.connect(*iter, ec);
}
if (ec) throw system_error(ec);


bool is_ipv4(const tcp::endpoint& endpoint)
{
  return endpoint.protocol() == tcp::v4();
}
```

Reality check: IPv6 is unlikely to be available yet

Writing Client Programs

```
tcp::resolver resolver(io_service);
tcp::resolver::query q("www.boost.org", "http");
vector<tcp::endpoint> endpoints(
    resolver.resolve(q),
    tcp::resolver::iterator());
stable_partition(endpoints.begin(), endpoints.end(), is_ipv4);

tcp::socket sock(io_service);
asio::connect(sock, endpoints.begin(), endpoints.end());
```

New in Boost 1.47

```
bool is_ipv4(const tcp::endpoint& endpoint)
{
  return endpoint.protocol() == tcp::v4();
}
```

Approaches for UDP clients:

- Keep source code “protocol independent”
- Use endpoints or addresses

Writing Client Programs

```
const char* ip_address_string = ...; // IPv4 or IPv6 string representation

udp::resolver resolver(io_service);
udp::resolver::query q(
    ip_address_string, "50555",
    udp::resolver::query::numeric_host);

udp::endpoint endpoint = *resolver.resolve(q);

udp::socket sock(io_service);
sock.open(endpoint.protocol());
sock.send_to(..., endpoint);
```

Writing Client Programs

```
const char* ip_address_string = ...; // IPv4 or IPv6 string representation

ip::address address = ip::address::from_string(ip_address_string);
udp::endpoint endpoint(address, 50555);
```

```
udp::socket sock(io_service);
sock.open(endpoint.protocol());
sock.send_to(..., endpoint);
```

Writing Server Programs

The deployment environment:

- Systems with both IPv4 and IPv6
- Systems with IPv4 only

The deployment environment:

- Systems with both IPv4 and IPv6
  - Dual stack
  - Separate stacks
- Systems with IPv4 only

The deployment environment is complex:

- Systems with both IPv4 and IPv6
  - Dual stack
    - Enabled by default
    - Disabled by default
  - Separate stacks
- Systems with IPv4 only

Approaches for TCP servers:

- Exactly one acceptor
  - Supports dual stack
  - Supports IPv6 only
  - Supports IPv4 only
- Up to two acceptors
  - Supports all environments

Exactly one acceptor:

```
const char* listen_address = ...; // Typically "0.0.0.0" or "::"

ip::address address = ip::address::from_string(listen_address);
tcp::endpoint endpoint(address, 50555);

tcp::acceptor acceptor(io_service, endpoint);
```

Exactly one acceptor:

```
const char* listen_address = ...; // Typically "0.0.0.0" or "::"

ip::address address = ip::address::from_string(listen_address);
tcp::endpoint endpoint(address, 50555);

tcp::acceptor acceptor(io_service, endpoint.protocol());

if (endpoint.protocol() == tcp::v6())
{
    error_code ec;
    acceptor.set_option(ip::v6_only(false), ec);
    // Call succeeds only on dual stack systems.
}

acceptor.bind(endpoint);
acceptor.listen();
```

Up to two acceptors – option 1:

```
error_code ec;
tcp::acceptor acceptor_v4(io_service);
tcp::acceptor acceptor_v6(io_service);

acceptor_v4.open(tcp::v4(), ec);
if (!ec)
{
    acceptor_v4.bind(tcp::endpoint(tcp::v4(), 50555));
    acceptor_v4.listen();
}

acceptor_v6.open(tcp::v6(), ec);
if (!ec)
{
    acceptor_v6.set_option(ip::v6_only(true));
    acceptor_v6.bind(tcp::endpoint(tcp::v6(), 50555));
    acceptor_v6.listen();
}
```

Up to two acceptors – option 2:

```
ip::v6_only v6_only;

acceptor_v6.open(tcp::v6(), ec);
if (!ec)
{
    acceptor_v6.get_option(v6_only);
    acceptor_v6.bind(tcp::endpoint(tcp::v6(), 50555));
    acceptor_v6.listen();
}

if (!acceptor_v6.is_open() || v6_only)
{
    acceptor_v4.open(tcp::v4(), ec);
    if (!ec)
    {
        acceptor_v4.bind(tcp::endpoint(tcp::v4(), 50555));
        acceptor_v4.listen();
    }
}
```

Up to two acceptors – option 3:

```
ip::v6_only v6_only(false);

acceptor_v6.open(tcp::v6(), ec);
if (!ec)
{
    acceptor_v6.set_option(v6_only, ec);
    acceptor_v6.get_option(v6_only);
    acceptor_v6.bind(tcp::endpoint(tcp::v6(), 50555));
    acceptor_v6.listen();
}

if (!acceptor_v6.is_open() || v6_only)
{
    acceptor_v4.open(tcp::v4(), ec);
    if (!ec)
    {
        acceptor_v4.bind(tcp::endpoint(tcp::v4(), 50555));
        acceptor_v4.listen();
    }
}
```

Approaches for UDP servers:

- Exactly one socket
  - Supports dual stack
  - Supports IPv6 only
  - Supports IPv4 only
- Up to two sockets
  - Supports all environments

Exactly one socket:

```
const char* listen_address = ...; // Typically "0.0.0.0" or "::"

ip::address address = ip::address::from_string(listen_address);
udp::endpoint endpoint(address, 50555);

udp::socket sock(io_service, endpoint);
```

Exactly one socket:

```
const char* listen_address = ...; // Typically "0.0.0.0" or "::"

ip::address address = ip::address::from_string(listen_address);
udp::endpoint endpoint(address, 50555);

udp::socket sock(io_service, endpoint.protocol());

if (endpoint.protocol() == udp::v6())
{
    error_code ec;
    sock.set_option(ip::v6_only(false), ec);
    // Call succeeds only on dual stack systems.
}

sock.bind(endpoint);
```

Up to two sockets – option 1:

```
error_code ec;
udp::socket socket_v4(io_service);
udp::socket socket_v6(io_service);

socket_v4.open(udp::v4(), ec);
if (!ec)
{
    socket_v4.bind(udp::endpoint(udp::v4(), 50555));
}

socket_v6.open(udp::v6(), ec);
if (!ec)
{
    socket_v6.set_option(ip::v6_only(true));
    socket_v6.bind(udp::endpoint(udp::v6(), 50555));
}
```

Up to two sockets – option 2:

```
ip::v6_only v6_only;

socket_v6.open(udp::v6(), ec);
if (!ec)
{
    socket_v6.get_option(v6_only);
    socket_v6.bind(udp::endpoint(udp::v6(), 50555));
}

if (!socket_v6.is_open() || v6_only)
{
    socket_v4.open(udp::v4(), ec);
    if (!ec)
    {
        socket_v4.bind(udp::endpoint(udp::v4(), 50555));
    }
}
```

Up to two sockets – option 3:

```
ip::v6_only v6_only(false);

socket_v6.open(udp::v6(), ec);
if (!ec)
{
    socket_v6.set_option(v6_only, ec);
    socket_v6.get_option(v6_only);
    socket_v6.bind(udp::endpoint(udp::v6(), 50555));
}

if (!socket_v6.is_open() || v6_only)
{
    socket_v4.open(udp::v4(), ec);
    if (!ec)
    {
        socket_v4.bind(udp::endpoint(udp::v4(), 50555));
    }
}
```

Summary

Summary

- Prefer protocol-independent approaches
- Remember to try all endpoints
- Choose designs appropriate to target environment