Document number: N3565
Date: 2013-03-15
Reply to: Aleksandar Fabijanic <alex@pocoproject.org>

IP ADDRESS DESIGN CONSTRAINTS

Abstract

This is a summary of IP address class design constraints that surfaced during Portland SG4 meeting [PORTLD] and subsequent [CPPSTDN] discussions. Goal is to determine a direction for the standard C++ IP address class proposal. This document does not aim to be a final verdict on what the design of standard IP address classes should look like; rather, it summarizes the concerns raised in discussions so far, weighing pros and cons of each proposed design. It is meant to serve as a reference for SG4 IP address discussion during Bristol meeting in April 2013.

Table of Contents

IP Address Design Constraints

1. Introduction

An Internet Protocol address (IP address) is a numerical label assigned to a device participating in a network which uses the Internet Protocol for communication.[RFC760] An IP address serves two principal functions: host or network interface identification and location addressing.

The designers of the Internet Protocol defined IP address as a 32-bit number[RFC760]; this protocol, known as Internet Protocol Version 4 (IPv4), is still in widespread use at the time of this writing. Due to the enormous growth of the Internet and the imminent depletion of available addresses, a new version of IP (IPv6), using 128 bits for the address, was developed in 1995.[RFC1883] IPv6 was standardized in 1998.[RFC2460] On February 3rd 2011, each RIR received its last /8 from IANA and the IANA free pool of IPv4 addresses reached 0%.[IANAv6]

2. IP Address Concept

Given the current transition of the IP address versions and complexities arising thereof, the discussion following IP Address [IPADDR] proposal publishing raised a dilemma about the number of classes needed to optimally capture the IP address concept. The forces shaping the design are:

A. simplicity of use (addressed by one class design)
B. space concern (addressed by two-class design)
C. performance concern (addressed by two- and three-class design)

This paper describes different approaches with enumerated pros and cons and a quantitative score for each.

Although it is not mandatory that C++ Standards Committee follows the Request for Comment (RFC) recommendations, given the amount of networking experience and expertise behind those documents, it is prudent to give them due attention. In that context, the relevant quotes recommending use of a single data structure can be found in [R4038-6], [R403861], [R403862] and RFC recommendations are taken into account when weighing pros and cons of each solution.

To grasp the IP address concept, a comparison with the familiar integer concept helps. In a world without memory space and computation speed limitations, a single integer type capable of accommodating infinitely large value would suffice; in reality, the amount of data storage and computing power is limited, mandating multiple integer types of various widths. Furthermore, although the operation set for integer concept is common for all types, there are operations which only make sense for certain types (e.g. comparison 'i < 0' it does not make sense for unsigned 'i').

Same principles apply to the IP address concept; ideally, there would be one IP address version, accommodating infinite number of IP addresses. However, the reality of limited resources mandates IP address versions of different sizes; equivalence with integers also holds in regards to the operation set - there are operations peculiar to an IP address version (e.g. 'is_broadcast()' query does not make sense for an IPv6 address).

In subsequent paragraphs, three distinct modes (with some "sub-modes") of capturing the IP address concept are laid out.

3. IP Address Classes Design Options

The order in which options are listed is based solely on the number of classes a solution proposes. The pros and cons are enumerated and quantified into a numeric score for each solution.

3.1. One IP Address Class

The proposal for a single IP address class is based on (a) RFC recommendations, (b) simplicity of use, (c) future-proof design and (d) an existing practice [POCOCPP]. There are known uses in embedded environments as well as server environments with tens of thousands IP addresses. Furthermore, in this approach, there is a consistency between IP address and socket address classes (there is one of each). On the other side, this approach raises significant size/performance concerns that may preclude its use in extreme size/performance constrained environments.

Pros:

+ one interface

+ future-proof

+ recommended [RFC4038]

Cons:

- size penalty for IPv4

- performance penalty for IPv4

- performance penalty for IPv6

Score: 0

3.2. Two IP Address Classes

The difference between the two approaches described in this paragraph is that the first one proposes two classes of which one is “universal” - IPv4 class covers only IP version 4, while IP class plays the "universal" role of both IPv6 and IPv4 addresses. The second approach proposes two classes as well; however, these are simply two distinct types which can be converted from one to another as needed, if possible.

3.2.1. IPv4/IP

In this variation, IPv4 class provides the IPv4 functionality support. IP, on the other hand, can hold either IPv4 or IPv6, depending on the actual value held and the use context. This requires that IP class exposes an interface that is "union" of IPv4 and IPv6 interfaces; operations that do not apply to certain version return reasonable defaults or predefined invalid values; for example, (a) ::is_broadcast() always returns false for IPv6 while (b) ::scope() always returns 0 for IPv4.

Pros:

+ size-optimized IPv4
+ future-proof

Cons:

- performance penalty for IPv6
- proliferation of types
- discouraged [RFC4038]

Score: -1

NOTE: In this option, there remains one concern regarding IPv4-compatible IPv6 addresses; a concern was raised [CPPSTDN] on whether an IPv4-compatible IPv6 address must itself (as opposed to its use-context) "know" its origin (i.e. whether it originated as IPv4 of IPv4-compatible IPv6). This concern does not apply to IPv4-mapped IPv6 addresses because they have a bit pattern distinguishable from either IPv4 or IPv4-compatible IPv4.

3.2.2. IPv4/IPv6

In this solution, like different integer types, IP address types are implicitly convertible to each other. However, unlike integer types, incompatible "narrowing" conversions from IPv6 to IPv4 must throw at runtime. Conversions from IPv4 to IPv6 always succeed, resulting in IPv4-mapped IPv6 target. In this solution, IP address is not a type but rather a concept that encompasses all present and future IP address types.

Pros:

+ size-optimized

+ no performance penalty

+ future-proof

Cons:

- proliferation of types

- discouraged [RFC4038]

Score: 1

NOTE: The main strength of this approach is that it fits well into the language development direction - concepts as the missing "link" between types and templates, with template generics serving as a "shield" from future changes.

3.3. Three IP Address Classes

3.3.1. IP Interface as Intersection

Asio C++ Library project [ASIOCPP] is an existing practice successfully using this design model; a common IP address class (with IPv4/v6 "intersection" interface, convertible to concrete type) is used as a "generic" IP address class.

Pros:

+ size and speed optimized IPv4

+ size and speed optimized IPv6

Cons:

- proliferation of types

- semi-functional generic interface

- discouraged [RFC4038]

Score: -1

NOTE: The main concern about this approach is a proliferation of classes. With three IP address classes, the questions of (a) other networking classes (should there also be three socket address classes?) and (b) algorithms arise.

3.3.2. IP Interface as Union

This is a variation of 3.3.1 option, with extended third class to "union" interface (as opposed to "intersection" in 3.3.1).

Pros:

+ size and speed optimized IPv4

+ size and speed optimized IPv6

+ fully functional generic interface

Cons:

- proliferation of types

- discouraged [RFC4038]

Score: 1

NOTE: The benefit compared to 3.3.1 option is that the third class can be used generically without conversion to concrete type, alleviating the concerns expressed above; this option is essentially a combination of options 3.1 and 3.2.2.

4. Conclusion

Several options were proposed for the IP address classes design. Options were judged for benefits and downsides, with quantitative score assigned to each. All pros/cons were weighted the same.

5. Acknowledgements

Many people have contributed to this summary, directly or indirectly. In particular, the suggestion by Gabriel Dos Reis [CONCGDR] to think of IP address as a concept was very helpful. Significant contributions came from the existing practices of Asio [ASIOCPP] and POCO [POCOCPP] open source projects as well as numerous standards committee meetings and mailing list[CPPSTDN] discussion participants.

6. References

[PORTLD] "Minutes: SG4 Networking, October 2012" (N3491), http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2012/n3491.htm

[CPPSTDN] c++std-networking mailing list (a.k.a "reflector")

[IPWIKI] "IP address", Wikipedia, http://en.wikipedia.org/wiki/IP_address

[IANAv6] "IPv4 Depletion and IPv6 Adoption Today", https://www.arin.net/knowledge/v4_deplete_v6_adopt.pdf

[RFC760] "DOD STANDARD INTERNET PROTOCOL", http://tools.ietf.org/html/rfc760

[RFC1883] "Internet Protocol, Version 6 (IPv6) Specification", http://tools.ietf.org/html/rfc1883

[RFC2460] "Internet Protocol, Version 6 (IPv6) Specification", http://tools.ietf.org/html/rfc2460

[RFC4038] "Application Aspects of IPv6 Transition", http://www.ietf.org/rfc/rfc4038.txt

[R4038-6] "6. ... dual applications working with both IPv4 and IPv6 are recommended. These applications should avoid IP dependencies in the source code. However, if IP dependencies are required, one of the better solutions would be to build a communication library that provides an IP version - independent API to applications and that hides all dependencies."

[R403861] "6.1. All memory structures and APIs should be IP version-independent. One should avoid structs in_addr, in6_addr, sockaddr_in, and sockaddr_in6."

[R403862] "6.2. The new address independent variants getaddrinfo() and getnameinfo() hide the gory details of name-to-address and address-to-name translations."

[ASIOCPP] Asio C++ Library, http://think-async.com/

[POCOCPP] C++ Portable Components, http://pocoproject.org

[CONCGDR] Gabriel Dos Reis <email removed> via accu.org Jan 17 to c++std-network.

Nevin Liber <email removed> writes:
>> On 17 January 2013 16:05, Gabriel Dos Reis <email removed> wrote:
>> I can understand that this is a very good C design. I am having
>> hard time seeing this as an acceptable C++ design.
> What do you see as an acceptable C++ design?
A typeful design.
More conretely, from what I've seen so far 'ip_address' removes type information. I see 'ip address' more of a concept than an actual datatype you program against.
-- Gaby