IPv6 support in Mesos

Benno Evers

September 23, 2016

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Scope . . . 3
- 1.3 Reminder: TCP/IP networking basics . . . 3
- 1.4 Reminder: libprocess networking basics . . . 4

2 Proposal **6**
- 2.1 Proposed changes . . . 6
  - 2.1.1 Interface . . . 6
  - 2.1.2 libstout . . . 6
  - 2.1.3 libprocess . . . 6
  - 2.1.4 Mesos . . . 6
- 2.2 Design considerations . . . 7
  - 2.2.1 Compile-time vs. command-line option . . . 7
  - 2.2.2 Advertised IP addresses . . . 7
  - 2.2.3 Scope Identifier . . . 7

3 Testing and Evolution **9**
- 3.1 Testing . . . 9
- 3.2 Transition paths and future evolution . . . 9
  - 3.2.1 Upgrading from existing installations . . . 9
  - 3.2.2 IPv6 with network isolation . . . 10
  - 3.2.3 Support for dual-stack agents . . . 10
  - 3.2.4 Support for dual-stack masters . . . 10

1 Introduction

1.1 Motivation

The problem of IPv4 address space exhaustion has been known since the 1980's, with the last /8 blocks being assigned in 2011. To address this problem, the IETF published the protocol IPv6 as a successor in 1998. Ever since, it is being adopted around the world.

Especially in datacenters, it is becoming more and more common to run a pure IPv6 network internally and to use NAT64 to speak to the IPv4 world outside during the transition period.

Since Mesos is specifically intended to be run inside datacenters and big clusters, it is important to be able to work inside IPv6-only environment.

1.2 Scope

The objective of this proposal is to give users the ability to run a Mesos installation inside an IPv6-only network. This proposal focuses on a minimal set of changes that need to be implemented to achieve this goal.

Any changes should be done in a backwards compatible manner, not impacting existing installations on an upgrade.

More ambitious plans, including seamless dual-stack support, are not in the scope of this proposal, except for the requirement that they should not be unnecessarily complicated by the proposed changes.

1.3 Reminder: TCP/IP networking basics

The Internet Protocol, better known as IP and sometimes as IPv4, is a packet-oriented network protocol designed during the 1970's and finalized 1981 in RFC 791.

The IPv6 protocol was designed in the 1990's by the IPng working group as a replacement for IPv4, the main objective being a substantial increase in the size of the available address space.

An IPv6 address consists of 16 octets, written in groups of 16-bit words separated by a colon `:`, for example `fe80:0000:0000:0000:6cdf:b4ff:0096:0d95`. To save space, leading zeros can be omitted and up to one consecutive range of zeros can be compressed to a double-colon `::`, shortening the previous example to `fe80::6cdf:b4ff:96:d95`.

TCP is a connection-oriented protocol implemented on top of IP. A TCP connection is identified by the 4-tuple (source ip, source port, destination ip, destination port), where

a *port* is a number between 1 and 65535.

A TCP connection is established by a exchanging a sequence of messages called the TCP 3-way handshake. The initiator of the connection will send a SYN-packet to the receiver. If the receiver wants to accept the incoming connection, it will respond with a SYN-ACK-packet, otherwise it will refuse the connection by sending a RST-packet.

A *socket* is an operating system abstraction which can be used to send and receive data over the network. A socket can be bound to a *socket address* using the `bind()` system call. The form of the socket address is determined by the socket's *address family*. A socket of address family `AF_INET` has a socket address consisting of an IPv4 address and a port, and socket of address family `AF_INET6` has a socket address consisting of an IPv6 address, a port, a scope identifier, and flow information. The flow information is not important for this document, and the scope identifier is briefly described in section 2.2.3 below. The socket address is a system-wide global resource: there can be at most one socket bound to a given combination of IP address and port.

A socket bound to an address can be put to a listening state using the `listen()` system call. Once a socket is in listening state, the kernel will accept incoming connection requests whose destination address matches the socket address. After an incoming connection has been established by the kernel, it can be accepted using the `accept()` system call. This system call returns a new socket which can be used for sending and receiving data.

A TCP connection is defined by both source and destination addresses, so multiple connections can be created from the same listening socket without ambiguity.

A socket can be bound to the special address (`0.0.0.0` for IPv4 or `::` for IPv6) to indicate that it wants to receive inbound connections matching only the port, regardless of their destination IP.

According to RFC 2553, which is respected by all major operating systems, a socket that is bound to the anycast IPv6 address will also receive incoming IPv4 connections to the same port. This behaviour can be configured by setting the socket option `IPV6_V6ONLY`. An IPv4 packet received through an IPv6 socket will have an address of the form `::ffff:xxxx:xxxx`, that is 80 zero-bits, followed by 8 one-bits, followed by the 32-bit IPv4 address.

1.4 Reminder: libprocess networking basics

The C++ library libprocess is a framework to enable a style of asynchronous programming based on the actor model. It is used internally by Mesos to handle all network communication.

Inbound communication in libprocess is received through the global socket `__s__`. At program startup, this socket is bound to the address given by the pair of environment variables `LIBPROCESS_IP` and `LIBPROCESS_PORT`, which in turn are populated from the `--ip` and `--port` command line arguments. The default address is `0.0.0.0:5050`.

The central abstraction of libprocess is the *actor*, who can send and receive messages. Creating a new actor managed by libprocess is done by inheriting from the class

`process::ProcessBase`.

Every `ProcessBase` object has an UPID, which is a string of the form `pid@address:port`. Here, `pid` is a human-readable identifier and `address` is an IPv4 address literal stored in the global variable `__address__`. This address is constructed from the environment variable `LIBPROCESS_ADVERTISE_IP` if specified, else from the environment variable `LIBPROCESS_IP` if specified, and otherwise from a DNS lookup for the return value of `gethostname(2)`.

Since the same operating system process will usually have multiple actors running at the same time, the class `process::SocketManager` is used to multiplex the network traffic such that only one socket per pair of hosts is required, instead of one per pair of actors.

The protocol used by libprocess is HTTP, with the UPID being sent as part of the `User-Agent:` header field.

2 Proposal

2.1 Proposed changes

2.1.1 Interface

We propose to add a new environment variable `LIBPROCESS_IPV6`, and to add a corresponding command-line option `--ipv6` to the mesos binaries to set this variable.

When set, the effect would be to set the default listen address to `::` and explicitly set the `IPV6_V6ONLY` socket option. Additionally, it would enable the use of IPv6 address literals in all places where currently IPv4 literals are supported. All DNS lookups will be made for IPv6 addresses only. Any attempt to use IPv4 literals or hosts with only A-type DNS records would result in an error message.

When not set, the old behaviour will is preserved: the default listen address will be `0.0.0.0`, and DNS lookups will be made for IPv4 addresses only. Any attempt to use IPv6-literals or hosts which only have AAAA-type DNS records would result in an error message.

2.1.2 libstout

The classes `stout::net::IP` and `stout::net::IPNetwork` need to be extended so they can store either an IPv4 or an IPv6 address.

2.1.3 libprocess

The class `process::network::Socket` requires an API change, so that the address family of the socket can be specified on socket construction, and to separate socket construction from establishing an SSL connection. This can be added in a backwards-compatible fashion by deprecating the old constructor `Socket::create` and adding new constructors with the desired functionality.

The parsing code for UPIDs needs to be extended to handle IPv6 literals.

2.1.4 Mesos

As part of determining the main outbound network interface and the default gateway, Mesos queries the system routing table. Currently, it is only looking for IPv4 routes. In IPv6-mode, it has to search IPv6 routes instead.

2.2 Design considerations

2.2.1 Compile-time vs. command-line option

To emphasize the exclusive nature of the IPv6 support, it would be possible to specify an `--enable_ipv6` compile-time option and produce separate binaries `mesos-master6` and `mesos-slave6` which run in IPv6 mode by default.

However, since all existing Mesos installations use IPv4, there is no urgency to enable IPv6 support by default. By having a command-line flag, as opposed to a build-time option, interested users have a low-effort and low-risk way to test the feature in their networks, and report potential issues.

2.2.2 Advertised IP addresses

The options `--advertised_ip` and `--advertised_port` allow the user to manually specify the address and port that are used for constructing the UPID.

According to the principle that the user is always right, we should allow to specify any IP address with this option, even if the specified address has a different address family than the listening socket `__s__`. Allowing this would also be a precondition for primitive support of dual-stack agents, as described in section 3.2.3.

On the other hand, allowing this would greatly increase the chance of a misconfiguration where network traffic between master and agent is silently dropped without the user being notified of the error.

Therefore, we propose to require that the socket address of `__s__` and the advertised IP be of the same address family, and to leave it up to a potential future proposal to relax this rule if required.

2.2.3 Scope Identifier

In addition to an IPv6 address and port, a socket address of family `AF_INET6` also contains a scope identifier. When written in textual form, it appears separated from the IPv6 address by a `%` sign, as in `fe80::1%eth0`.

These identifiers, which are described in RFC 4007, are strictly local to a host. They allow network administrators to disambiguate the network interface to which an address should be bound, in case multiple network interfaces are configured with overlapping IPv6 subnets. This situation is permitted and common for non-global IPv6 addresses, since every network interface will in general have at least one link-local address in the subnet `fe80::/64`.

Since link-local addresses are only guaranteed to be unique per network segment (i.e., they have the same scope as MAC addresses), they are hardly useful for anything but stateless autoconfiguration. Global addresses, localhost, and the address `::` do not have a scope identifier.

Since there are no existing applications depending on the use of scope identifiers, and it is hard to imagine a network layout where this would be necessary, we propose to forbid their usage by showing an error message when a user tries to specify an IPv6 literal

including a scope identifier. This will allow future implementers, should it turn out to be necessary, to use them in any way without having to be careful about preserving backwards compatibility.

3 Testing and Evolution

3.1 Testing

As a proof of concept, a custom version of Mesos with IPv6 support was built. After running this version of Mesos inside an IPv6-only production environment for six months, no network-related issues were encountered. The patch series is available at https://github.com/lava/mesos/tree/bennoe/ipv6.

The diffstat is

```
 3rdparty/libprocess/include/process/address.hpp   |  53 ++++++++++--
 3rdparty/libprocess/include/process/socket.hpp    |  26 +++++-
 3rdparty/libprocess/include/process/ssl/gtest.hpp |   2 +-
 3rdparty/libprocess/src/http.cpp                  |  48 +++++++++--
 3rdparty/libprocess/src/libevent_ssl_socket.cpp   |   2 +-
 3rdparty/libprocess/src/pid.cpp                   |  32 ++++++-
 3rdparty/libprocess/src/poll_socket.cpp           |   2 +-
 3rdparty/libprocess/src/process.cpp               |  30 +++++--
 3rdparty/libprocess/src/socket.cpp                |  69 ++++++++++++----
 3rdparty/libprocess/src/tests/http_tests.cpp      |  16 ++++
 3rdparty/stout/include/stout/ip.hpp               | 195 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 3rdparty/stout/include/stout/net.hpp              |  30 +++++--
 3rdparty/stout/tests/ip_tests.cpp                 |  91 ++++++++++++++++++++++
 src/common/protobuf_utils.cpp                     |   6 +-
 src/linux/routing/route.cpp                       |   6 +-
 src/master/maintenance.cpp                        |   2 +-
 src/master/master.cpp                             |   8 +-
 src/sched/sched.cpp                               |   2 +-
 18 files changed, 503 insertions(+), 117 deletions(-)
```

Note that this patch series does not implement the proposal exactly as written, but it should give an order-of-magnitude approximation of the size of the required code changes.

3.2 Transition paths and future evolution

3.2.1 Upgrading from existing installations

There is no change in behaviour when the `--ipv6` flag and the `LIBPROCESS_IPV6` environment variable are not set, therefore existing installations running in IPv4 are not be

affected.

Since Mesos does not have IPv6 at the moment, there are no existing installations running in IPv6 mode which could be affected.

3.2.2 IPv6 with network isolation

Mesos is a system for executing tasks of some distributed system on a cluster. Since these tasks are provided by the user, Mesos has no control over or knowledge about their behaviour. In particular, it could happen that two tasks are scheduled on the same host that try to use the same system-wide global resource. The most prominent example would be two tasks trying to listen on the same TCP port.

To mitigate this problem, Mesos tries to isolate running tasks from each other to various degrees depending on the agent's run-time configuration. In particular, the class *mesos::internal::slave::PortMappingIsolatorProcess* works by assigning to each task a range of ports and a virtual network interface. A set of routing rules is added to the system routing table which forwards host traffic in this port range to the container.

Both TCP over IPv4 and TCP over IPv6 share the same set of valid port numbers, so this mechanism can be straightforwardly extended to handle IPv6 packets.

Since the IPv6 address space is very large, it would have been possible to avoid port conflicts by assigning a unique, globally routable IPv6 address to every container spawned by Mesos. However, network administrators generally may not allow machines to dynamically self-assign new IP addresses, so this is hardly a practical solution, and some form of NAT will still be necessary even when IPv6 is deployed.

3.2.3 Support for dual-stack agents

It might be desirable to run a slave connected to a master identified by hostname and port without worrying about the underlying transport protocol used for communication.

To enable this behaviour, it would be necessary to decouple the actual network address used by the socket from the network address used as identifier by libprocess, and to add a mechanism to change.

Then, the slave can bind to the IPv6 any-address `::` accepting both IPv4 and IPv6 traffic, and probe which protocol the master expects by using the "Happy Eyeballs" algorithm (RFC 6555).

Note that the master needs to enable and the slave needs to disable the `IPV6_V6ONLY` socket option for this scheme to work correctly.

After determining the expected transport protocol, the slave would set `__address__` to either the public IPv4 or IPv6 address of the slave host.

3.2.4 Support for dual-stack masters

It might also be desirable to run a master that is able to work with both IPv4 and IPv6 slaves simultaneously.

Just exchanging network traffic in a dual-stack environment would be quite easy: the master listens on an IPv6 socket bound to a wildcard address with the socket option `IPV6_V6ONLY` set to false, and it will receive all inbound traffic to that port.

However, most of this inbound traffic will still be refused by libprocess: if a slave on an IPv4-only host connects to, say, `master@10.11.0.1:5050` but receives a reply from `master@[fd11::1]:5050`, it will ignore this message.

Therefore, allowing this kind of scenario would be far more challenging, because it would be necessary to relax the rule that a `process::ProcessBase`'s UPID is one unique value which will also be the source IP address for all received packets.

Note that the problems stemming from the dual role of UPIDs are not specific to IPv6. A proposal to allow a Mesos-master to utilize multiple network interfaces would have to solve the same problem.