Subject: Re: Network virtualization/isolation
Posted by ebiederm on Mon, 04 Dec 2006 15:35:42 GMT
View Forum Message <> Reply to Message

jamal <hadi@cyberus.ca> writes:

> On Mon, 2006-04-12 at 05:15 -0700, Eric W. Biederman wrote:
>> jamal <hadi@cyberus.ca> writes:
>>
>
>> Containers are a necessary first step to getting migration and
> checkpoint/restart
>> assistance from the kernel.
>
> Isnt it like a MUST have if you are doing things from scratch instead of
> it being an after thought.

Having the proper semantics is a MUST, which generally makes those
a requirement to get consensus and to build the general mergeable
solution.

The logic for serializing the state is totally uninteresting
for the first pass at containers. The applications inside the
containers simply don't care.

There are two basic techniques for containers.
1) Name filtering.
   Where you keep the same global identifiers as you do now, but
   applications inside the container are only allowed to deal with a
   subset of those names. The current vserver layer 3 networking
   approach is a handy example of this. But this can apply to process
   ids and just about everything else.

2) Independent namespaces. (Name duplication)
   Where you allow the same global name to refer to two different
   objects at the same time, with the context the reference comes
   being used to resolve which global object you are talking about.

Independent namespaces are the only core requirement for migration,
because the ensure when you get to the next machine you don't have
a conflict with your global names.

So at this point simply allowing duplicate names is the only
requirement for migration. But yes that part is a MUST.

>> > 2) the socket level bind/accept filtering with multiple IPs. From
>> > reading what Herbert has, it seems they have figured a clever way to
>> > optimize this path albeit some challenges (speacial casing for raw

>> > filters) etc.
>> >
>> > I am wondering if one was to use the two level muxing of the socket
>> > layer, how much more performance improvement the above scheme provides
>> > for #2?
>>
>> I don't follow this question.
>
> if you had the sockets tables being in two level mux, first level to
> hash on namespace which leads to an indirection pointer to the table
> to find the socket and its bindings (with zero code changes to the
> socket code), then isnt this "fast enough"? Clearly you can optimize as
> in the case of bind/accept filtering, but then you may have to do that
> for every socket family/protocol (eg netlink doesnt have IP addresses,
> but the binding to multiple groups is possible)
>
> Am i making any more sense? ;->

Yes. As far as I can tell this is what we are doing and generally
it doesn't even require a hash to get the namespace. Just an appropriate
place to look for the pointer to the namespace structure.

The practical problem with socket lookup is that is a hash table today,
allocating the top level of that hash table dynamically at run-time looks
problematic, as it is more than a single page.

>> > Consider the case of L2 where by the time the packet hits the socket
>> > layer on incoming, the VE is already known; in such a case, the lookup
>> > would be very cheap. The advantage being you get rid of the speacial
>> > casing altogether. I dont see any issues with binds per multiple IPs etc
>> > using such a technique.
>> >
>> > For the case of #1 above, wouldnt it be also easier if the tables for
>> > netdevices, PIDs etc were per VE (using the 2 level mux)?
>>
>> Generally yes. s/VE/namespace/. There is a case with hash tables where
>> it seems saner to add an additional entry because hash it is hard to
> dynamically
>> allocate a hash table, (because they need something large then a
>> single page allocation).
>
> A page to store the namespace indirection hash doesnt seem to be such a
> big waste; i wonder though why you even need a page. If i had 256 hash
> buckets with 1024 namespaces, it is still not too much of an overhead.

Not for namespaces, the problem is for existing hash tables, like the
ipv4 routing cache, and for the sockets...

>> But for everything else yes it makes things
>> much easier if you have a per namespace data structure.
>
> Ok, I am sure youve done the research; i am just being a devils
> advocate.

I don't think we have gone far enough to prove what has good performance.

>> A practical question is can we replace hash tables with some variant of
>> trie or radix-tree and not take a performance hit. Given the better scaling
> of
>> tress to different workload sizes if we can use them so much the
>> better. Especially because a per namespace split gives us a lot of
>> good properties.
>
> Is there a patch somewhere i can stare at that you guys agree on?

For non networking stuff you can look at the uts and ipc namespaces
that have been merged into 2.6.19. There is also the struct pid work
that is a lead up to the pid namespace.

We have very carefully broken the problem by subsystem so we can do
incremental steps to get container support into the kernel.

That I don't think is the answer you want I think you are looking
for networking stack agreement. If we had that we would be submitting
patches at the moment.

The OpenVz and Vserver code is available.

I have my proof of concept git tree up on kernel.org which has a terribly
messing history but it's network stack is largely the L2 we are talking about.

>> Well we rather expect to bash heads until we can come up with something
>> we all can agree on with the people who more regularly have to maintain
>> the code. The discussions so far have largely been warm ups, to actually
>> doing something.
>>
>> Getting feedback from people who regularly work with the networking stack
>> is appreciated.
>
> I hope i am being helpful;

I thought that was what I just said! Yes you are helpful.

> It seems to me that folks doing the different implementations may have
> had different apps in mind. IMO, as long as the solution caters for all

> apps (can you do virtual bridges/routers?), then we should be fine.
> Intrusiveness may not be so bad if it needs to be done once. I have to
> say i like the approach where the core code and algorithms are
> untouched. Thats why i am humping on the two level mux approach, where
> one level is to mux and find the namespace indirection and the second
> step is to use the current datastructures and algorithms as is. I dont
> know how much more cleaner or less intrusive you can be compared to
> that. If i compile out the first level mux, I have my old net stack as
> is, untouched.

As a general technique I am in favor of that as well. The intrusive
part is that you have to touch every global variable access in the
networking stack. It is fairly clean and fairly non-intrusive, and
certainly makes it easy to that the patch does nothing nasty. But you
do have to touch a lot of code.

It occurs to me that what we really want as a first step to this
is simply to implement noop versions of the accessors functions we
are going to need. That way we can merge the intrusive bits before
we do the actual implementation.

If we could get a reasonably clear design with how to do the variable
accesses and the incremental approach to changing all of them, I think
we would see a lot less resistance to the L2 work. Simply because
it would go from a mammoth patch to a relatively small patch.

Thinking out loud. As far as I have seen there are two paths for
looking up the network namespace.

- Packets transmitted out of the kernel.
- Packets being received by the kernel.

For out going packets we can look at the socket to find the
network namespace. For in coming packets we can look at the
network device. In neither case do we actually have to tag
the packets themselves.

In addition there are a couple of weird cases with things like
ARP. Where the kernel generates the reply packet without the
packet really coming all of the way into the kernel, that
I recall having to special case.

Is that roughly what you were thinking with respect to finding
the current network namespace?

Where and when you look to find the network namespace that applies to
a packet is the primary difference between the OpenVZ L2
implementation and my L2 implementation.

If there is a better and less intrusive while still being obvious method I am all for it. I do not like the OpenVZ thing of doing the lookup once and then stashing the value in current and the special casing the exceptions.

For me I'm just trying to keep everyone from going in really insane directions, while I work through the really non-obvious bits like how do we successfully handle sysctl, and sysfs. Those things that must be refactored to be able to cope with multiple instances of some of the primary kernel data structures.

Eric
