Configuring corosync with kronosnet (knet)

Christine Caulfield (ccaulfie@redhat.com)

v1.0b	16th Jan 2018	initial version
v1.1b	2nd Mar 2018	mention ‘name’ attribute now needed
v1.1c	10th May 2018	fix a couple of small typos
v1.2	25th May 2018	add notes on converting multi-ring clusters
v1.3	25th Feb 2019	explain how ipv6/ipv4 is chosen

Step 1
Change *transport: udp* (or *udpu*) to *transport: knet*

Job done

I hope you enjoyed this tutorial

Wait? You wanted more than that? OK, but the above is all you need to get going with knet, it really is that easy and you should notice no functional difference. In fact if you do notice anything (apart from it being 'better' in some vague way you can't quite put your finger on) then we want to know.

1. Overview

What follows is some the gory details of extra features and options you can tweak when knet is your transport of choice – as it should be. All of these options are documented in the corosync.conf(5) man page, but this should I hope, give you a little more insight into what's happening when you select various options and why you might want to do so.

I'll take the options in the order that most people seem to write their corosync.conf files. If you write yours upside down then I apologise in advance for doing this backwards but really you only have yourself to blame.

This document is specific to corosync, while it discusses some inner workings of knet and what the parameters do, they are always in a corosync context and might or might not be applicable to knet when used in other applications.

2. The totem{} section

The major part of this section is very similar to before. You don't even need to have

```
transport: knet
```

in there for any of this to happen – it's the default.

If you are still using udpu or udp then you are now limited to a single ring (ie one address for each node) in the nodelist. So if you were using udp or udpu for two rings you will now get a startup error from corosync. Only knet allows multiple interfaces (called links) to be used without bonding.

The options bindnetaddr / broadcast / mcastaddr & ttl are not supported under knet as they make no sense.

2.1 crypto

The first different you might in the totem section see is in the crypto options. Knet supports all the crypto options that are allowed in nss (by default) and openssl. Yes, you can choose which library you want to do your encrypting – so if you're a huge fan of openssl, or going out with one of the developers, and want to use that over nss (or vice versa) you now have the choice. You do this with the option:

```
crypto_model: [openssl|nss]
```

The other crypto options (crypto_hash and crypto_cipher) take the same range of options as before, it's just the library implementing them that differs. nss and openssl implementations of the same algorithms should be compatible with each other on the wire.

2.2 links and link modes

Links are a crucial concept to knet, they are slightly analogous to the rings in corosync previously … but only slightly. In fact corosync runs its, now single, ring over however many links you configure in knet. All nodes should be connected to each link and ideally they should be on the same network or with similar latencies between the various nodes on that link. There's nothing new about that I hope you'll agree, I just thought I'd mention it anyway just in case. With

```
link_mode: passive
```

(which is the default) knet will always use the lowest-numbered priority (or number) link for traffic and failover 'upwards' to the next priority/number one if needed. By default links have the same priority number as their link number (which is the 'X' number in ringX_addr in the nodelist. I'll explain this bit a little more later, but this stuff isn't hard. Really it isn't.

```
link_mode: active
```

Means that all of the links will be used for every packet. This is also known as the 'overkill' option but might be useful to some people.

```
link_mode: rr
```

is the Round-Robin option. Messages will be sent on each link in order in, err, “round robin” style. See why we called it that? I'll come clean with you here and say that (at the time of writing) we don't test rr so it might not be as functional as you would hope. So if you do chose to try it, guard your sandwiches.

There is a maximum of eight links in knet, which we hope should be enough. If you only have one link between the nodes then don't waste knet's time by setting link_mode.

Links in knet can have several attributes and can be brought up and down at will. Provided corosync has one link active, you can take down links, reconfigure them, and bring them back up again as much as you please. This is very powerful. You can add new interfaces, move networks around, change protocols etc, all without losing nodes from the cluster. PROVIDED, as I just said (and I'll say it again for effect), corosync has one stable link still there.

That stable link can be a different link for each reconfiguration process though. So (for example) say you have 3 links. You can take down and reconfigure links 1 and 2 provided link 0 stays the

same (and working obvs). *Then* you can reconfigure link 0 to be something else because links 1 & 2 will be there for corosync. Yes, you can TOTALLY CHANGE all of the links in a system. Provided you do it in two steps. And, of course, that you do it consistently on all nodes at the same time – more or less.

2.3 MTU

Knet determines the MTU of all links dynamically so should handle different sizes on each link as well as MTUs that change over time. The 'netmtu' option has no effect on knet at all so you can forget it. I only mention it here as an excuse to show how clever knet is about MTUs. Seriously, we put a LOT of effort into that code.

2.4 Compression

knet supports compression of packets on the wire using whichever compression library is your preference – as with crypto we support personal choice here at knet towers. It needs a few options to get it going, first of which is the library you are going to use. Unlike crypto, the library you choose MUST be available on all nodes as the compression methods are incompatible with each other. You don't actually need to use the same compression mode on each node (the compression type is in the data packet) but I strongly recommend that you do – if only for your own sanity.

```
knet_compression_model: <library>
```

is the option that specifies which library to use. Which libraries are available to you depends on which ones were configured when knet was built. It's likely that zlib and lz4 will be available, but any others might depend on your distributor.

```
knet_compression_level: <number>
```

is a number that is passed to the compression library that should tell it how hard to try and compress the data. I'm not going to elaborate on that here as it's different for each library and I'm not going through all the library docs just for the benefit of this one. You'll have to do some of the work yourself here.

```
knet_compression_threshold: <number>
```

Tells knet to compress only packets larger than <number> bytes. This can save time if you're sending a lot of small packets that there would be no point in compressing. The default is 100 and it's best not to set it any lower than this as corosync itself sends small packets as part of its protocol and you could significantly add to the latency. If you want to see the impact that compression has on performance then have a look at the stats (see section 4). Setting this to 1 will compress everything, and setting it to 0 will restore the default of 100.

To be honest, compression is probably not that useful to corosync. By all means play with it but if it degrades your performance for no real benefit, don't come running to me, I'll just tell you I told you so.

3. interface{}

corosync.conf has interface{} sections inside totem{} that have lots of things you can twiddle to affect knet. If I was serious about my numbering system this would still be in section 2, but as there's a lot of extra detail here and I like being contrary sometimes, it's section 3. Because mummy said so, OK?

By the way, all of these entries are optional. Knet will run quite happily without you messing with the things in here. Maybe even better.

3.1 linknumber

Under the old corosync this was called ringnumber but I've renamed it linknumber to be more consistent with knet. You'll notice (below) that nodes addresses are still labelled ringX_addr rather than linkX_addr. I might add linkX_addr later but mostly it's like that to keep old corosync configs working.

Linknumbers start at 0 and go up to 7. Each link has a priority assigned to it for failover purposes, remember?

3.2 knet_link_priority

is the priority of the link, which overrides the link number when deciding the order to fail over links. Priorities go from 0 to 255 and are only used when link_mode is 'passive'.

3.3 ping options

I'm grouping these under one heading partly because I can't be bothered doing a section on each but mainly because they are all interrelated. Knet does it's own liveness pinging on top of (or 'beneath' I suppose might be more accurate) corosync's ring protocol. This is more frequent and allows it to catch short network outages that could have, in the past, caused corosync token losses. Knet uses *pings* and *pongs* to determine node liveness (I suspect Fabio is a secret table tennis fan), a node must respond to or ignore a certain number of pings with pongs before being declared live or dead.

It is possible to set these parameters individually for each link (hence them being in the 'interface' section) but generally it makes sense to keep them the same for the sanity of the token processing. If you don't then you might find that corosync behaves differently depending on which link is active at the time.

ping_interval is the time (in ms) between pings. By default the ping_interval is set to be the *token_timeout / (pong_count*2)* and the ping_timeout is set to the *token_timeout / pong_count*. This ensures that knet detects node outages before corosync does and only passes actual node losses up to corosync. You can change the values of these manually, but I recommend you don't.

If you do really insist on changing them, then I also insist that you change both knet_ping_timeout and knet_ping_interval together. If you only change one of them, then the other will still be calculated based on the token timeout value and weird things could easily happen. You've seen the film, you know what I mean.

ping_precision has actually nothing to do with the above calculations, confusingly. It's used internally by knet to calculate the latencies between the nodes on each link. Best leave this one too. I don't even know why I mentioned it.

3.4 other interface{} options

Other options in the interface section do just what you might expect.

```
mcastport: <n>
```

tells knet to use that port number <n> for communication,. The default remains the old one of 5405 +linknumber, but you can override it per link here. Even though knet doesn't do actual multicasting the name remains for old time's sake.

```
knet_transport: [udp|sctp]
```

This tells knet which IP protocol to use for this link. The default is UDP. It is quite permissible to have a UDP and an SCTP link on the same network – I'm not sure why you would want to do that but it could be interesting. More usefully they would be on separate networks where perhaps a switch was optimised for those particular protocols. I'm not going to offer advice on which protocol is better - the default is UDP because that's more likely to be allowed by corporate switches, but in testing we have seen very good throughput with SCTP.

4. nodelist{}

In practice, the nodelist is pretty much the same as before, why change a winning formula after all? The main change here is that under knet, eight links (ring[0..7]_addr) are allowed rather than the previous maximum of two. Also, you can freely mix IPv6 and IPv4 addresses in there, you don't need to warn corosync beforehand which you are going to use.

Well, maybe 'freely' mix them is a bit of an overstatement. All of the addresses on a link need to be of the same family (this is checked at startup time, so don't bother messing with us) but you can have, for example, 4 IPv4 links and 4 IPv6 links. Also, remember I said about reconfiguring links in section 2 … when you take a link and bring it back up again it can have a different IP version. So if you are careful, it's possible to upgrade from IPv4 to IPv6 if necessary. And back again when it gets scary of course.

nodelist{} is compulsory with knet clusters by the way. All of this hippy 'oh we'll just create a multicast address and add nodes as we like' stuff has been removed. Nobody liked it anyway. It possibly still works with udp, but … *shrugs*.

One very important point about the links as specified in the nodelist is the Ghostbusters mantra – **you must not cross the streams!** So link 0 on node 1 must go to link 0 on node 2 and link 0 on node 3 etc. The same goes for link 1 etc etc of course, you're not stupid, you got that. Anyway, be careful about it and reality will remain intact.

One requirement that is new here for knet (and I only added this in the 1.1 version of this document because I've just written the code for it) is that you need to assign each node a name. This is in addition to the ringX_addr values. eg:

```
nodelist {
        node {
                ring0_addr: 192.168.1.101
                ring1_addr: 10.0.0.56
                name: mylovelynode
        }
        … etc ...
}
```

The reason for this might not be obvious and I can hear people now shrieking 'control-freakery' at us, but it is very important. Sit in a comfy chair, calm down, get a small brandy and let me explain.

Corosync needs to know what your local node IP address is so that it can assign interfaces to the links. Now that all links are effectively optional, if you identify purely using link0 (as used to happen) and then that link goes away, we then have to look at link 1 or link 2 and where do we stop? And what happens if there's a conflict and the wrong node gets picked (which would be a mistake in making the config file but, hey, these things happen)? Literally all-hell would break loose: nodeids changing, dogs & cats living together, the whole bit.

The way corosync determines which node name corresponds to the local host is taken from cman (in fact I just nicked the code – well, I wrote it originally). I will repeat it here because I'm good to you like that.

1. It looks up $HOSTNAME in the nodelist
2. If this fails it strips the domain name from $HOSTNAME and looks up that in the nodelist
3. If this fails it looks in the nodelist for a fully-qualified name whose short version matches the short version of $HOSTNAME
4. If all this fails then it will search the interfaces list for an address that matches a name in the nodelist

So then. 'name' is now the canonical way of identifying a node, and by good luck it also provides a nice user-friendly thing we can display to users to identify the node too. And as the vast majority of config files will be made using configuration systems this shouldn't be much of a hardship.

There is another, internal technical, reason why we've done this to you. Corosync's internal protocol is a ring. In corosync 1 and 2 the node IP addresses determined the order of nodes in the ring. With the possibility that any of those IP addresses can change this is obviously not a sensible option – if the ring order can change, then we don't necessarily know when a message has been all the way round the ring. So now we use the nodeid to determine ring order which meant quite a lot of internal jiggery-pokery which also had the rather nice side-effect of shrinking the network packets quite a lot too.

You might be yelling at the screen now "BUT YOU SAID I DIDN'T NEED TO CHANGE ANYTHING!". And if you're not then you haven't really been paying attention. Anyway, by special dispensation, you can get away without a 'name' for each node if you get a note from your mum. And also if you are only ever going to use 1 (one) link. This is a given if you are using udpu or udp of course, and that's why I said you don't need to change anything to keep going from corosync 2. If you need to use the new features, then I'm afraid we're going to make you work for them. By 'you' here I mostly mean the nice people who write the GUIs that generate configuration files for you. Though there are adventurous people who do their own. Like, well. me.

So, yeah. Assign a name to your nodes. You know it makes sense.

4.1 IPv4 vs IPv6

Not quite as much fun as *Alien vs Predator* (which I haven't seen TBH), but more likely to cause your data centre to hide under the seat.

We had some exciting times getting corosync to do 'the right thing' when it came to choosing between IPv4 and IPv6 addresses if they have the same DNS name. I won't relate the story here as it's more suitable for bedtime reading (ie. it *will* send to you to sleep), but the end result is that if you have a hostname that resolves to both IPv6 and IPv4 addresses then the IPv6 one will be chosen. We're modern here you see. In more detail: Corosync will call getaddrinfo() on the name,

and look for the first IPv6 address in the list. If found then it will use that. It will then look for an IPv4 address in the list. If it still doesn't find it then it will issue a mild rebuke – at least.

You can change this order with the **ip_version** option in the totem stanza, but be aware that this is global to the whole corosync.conf file and will affect all nodes. If you have some links that are on IPv4 and some that are on IPv6 then I strongly recommend you name them carefully. Something basic like sticking a '4' or a '6' in the name would be helpful. You know the sort of thing

Anyway, the totem.ip_version thing has 4 options that you might (or might not) find helpful.
ipv6-4 is the default and does as I've described above
ipv4-6 does it the other way round
ipv4 and *ipv6* force corosync to always look for one or the other IP version. These options are a bit of a sledgehammer but might get you out of a DNS hole (whatever one of those might be) in some circumstances.

If you're in charge of your node names then I do recommend that you name them so that the ipv4/6 versions are distinguishable, but we do appreciate that not everyone has that luxury, so we hope this option will go some way to making it easier to get the right addresses for your nodes

5. logging

There's nothing much more to say here apart from the fact that knet has its own logging source (called, predictably 'KNET') and enabling debug logging can generate a lot of output.

6 Stats

knet provides loads of stats. And when I say 'loads' I really mean 'LOADS' of stats. You just won't believe how vastly hugely mind-bogglingly big the amount of stats is. There are so many stats available now that we had to move them into their own 'map' away from the normal corosync cmap because they got so big they were crowding out the originals and stealing all the nibbles.

Just to give you a flavour of the sort of thing you can find in the stats map – it contains detailed networking stats from each link to each node – so on a four node cluster with four links that's sixteen tables of information about sent & received packets, latencies and counters. It also tells you the time taken to compress/decompress & encrypt/decrypt packets and how much data was saved by the compression too. And, if that wasn't enough there is also detailed information about the IPC clients. Actually they were there in corosync 2, but I thought it worth mentioning.

If you're using the command-line 'corosync-cmaptool' tool then the default invocation will now just give you the configuration or 'icmap' map. To get the stats map you need to add '-m stats' to the command and your terminal will be flooded with useful (well, ish) numbers. There are two main reasons for this. Firstly it means we don't have to store the stats in memory twice – once in knet and then copy them into corosync's icmap database, but it also keeps the output of cmaptool more sane and thence, hopefully, its user.

If you are using libcmap from a program to access the stats then you can initialise libcmap with cmap_initialize_map(*handle, CMAP_MAP_STATS) and it will give you access to the stats in all their nerdy glory.

There are a couple of 'wrinkles' in using the stats map. Firstly they are all read-only. I'm not sure that's much of a surprise. Inventing your own stats is not a terribly useful thing to do, that's how fake news starts. Also the trackers don't necessarily work how you might initially expect.

Setting up a tracker on any of the transport stats just enables a timer, so you'll get stats updates every 1.5 seconds (ARBITRARY VALUE KLAXON). This is not configurable. If you want more or less frequent stats updates then set up your own timer, it's not hard. Providing a 'proper' stats update would be ridiculous – imagine getting a callback every time a packet is sent, nothing would ever get done, it would be like having a small child demanding ice-cream every 100 nano-seconds. Remember that? Well, we don't want it again.

Trackers on the IPC stats ***do*** work how you would expect them to – well for ADD and DELETE anyway. So if you set up such a tracker on an IPC destination then you'll get a callback when a new client connects or disconnects.

From the command line you can clear the stats with the -C option to corosync-cmapctl, it takes a single argument of *knet, ipc, totem* or *all*. If you are using the libcmap API then you can write to some special destinations in the stats map. They are self-explanatorily (I hope) named:

```
stats.clear.all
stats.clear.knet
stats.clear.ipc
stats.clear.totem
```

Just write anything into those destinations and the respective stats collection will be set to zero.

I'm not going to go into any more detail on the stats here, or I'll be typing well into next year. Just look at them, see what's there and what might be useful to you. If you have a specific question, then ask us on the mailing list or IRC. We're friendly. Really. I've not bitten anyone in AGES.

7. Converting multi-ring clusters

I’ve covered all of this in bits already but I’ll bring it all together here so that things are clearer. Single-ring clusters are easy to upgrade – see section 1, but multi-ring clusters require a little intervention. If you’ve been reading this very closely then either you won’t need to read this bit or you need new glasses. For the rest of you I’ll spell it out a bit more

udp & udpu no longer support more than one ring, so if you try to start an existing multi-ring cluster with corosync 8 you will get this message:

```
error   [MAIN  ] parse error in config: 2 is too many configured interfaces for non-Knet transport.
```

And if you simply change ‘udpu’ for knet you will get this:

```
error   [MAIN  ] parse error in config: No valid address found for local host
```

So, I’m afraid that if you want to update a multi-ring cluster form corosync2 to corosync3 then you’re going to have to at least a small amount of work. That work is simply giving your nodes names as I mentioned in section 4. It’s not a totally seamless upgrade , but it’s not arduous either I hope. Anyway, if a front-end generated the files for you in the first place I would put good money (though perhaps not a large amount of it) on them also helping you upgrade.

8. Example corosync.conf showing some of the new features

This file is deliberately not annotated because you should know what it all means by now.

```
totem {
    version: 2
    cluster_name: capybara
    transport: knet

    knet_compression_model: bzip2
    knet_compression_threshold: 10
    knet_compression_level: 1

    crypto_model: nss
    crypto_hash: sha1
    crypto_cipher: aes256

    interface {
      linknumber: 0
      knet_transport: udp
      knet_link_priority: 0
    }
    interface {
      linknumber: 1
      knet_transport: sctp
    }
    interface {
      linknumber: 2
      knet_transport: udp
      knet_link_priority: 10
    }
}

nodelist {
    node {
           ring0_addr: 192.168.1.101
           ring1_addr: 192.168.10.1
           ring2_addr: 192.168.9.1
           nodeid: 1
           name: mynode1
    }

    node {
           ring0_addr: 192.168.1.102
           ring1_addr: 192.168.10.2
           ring2_addr: 192.168.9.2
           nodeid: 2
           name: mynode2
    }

    node {
           ring0_addr: 192.168.1.103
           ring1_addr: 192.168.10.3
           ring2_addr: 192.168.9.3
           nodeid: 3
           name: mynode3
    }

    node {
           ring0_addr: 192.168.1.104
           ring1_addr: 192.168.10.4
           ring2_addr: 192.168.9.4
           nodeid: 4
```

```
            name: mynode4
    }
}


quorum {
    provider: corosync_votequorum
}

logging {
    to_logfile: yes
    logfile: /var/log/cluster/corosync.log
    to_syslog: yes
    logger_subsys {
        subsys: KNET
        debug: on
    }
}
```