Analyzing cman tcpdumps from RHEL-4

Christine Caulfield 2nd December 2010

Introduction

This document describes some rudimentary analysis techniques for looking at tcpdumps for RHEL-4 clusters.

Network packet captures (via tcpdump or wireshark) are a useful diagnostic aid in RHEL-4 clusters. The packet structure is simple, as is the membership protcol, and packets are fairly infrequent (or should be). Also, with cman being in the kernel in RHEL-4 doing other diagnosis on the state of the code can be difficult.

None of this applies to RHEL-5 where packets are large, complex, encrypted and frequent, and the membership protocol is complex.

Despite this be warned: looking at network packet captures is not a task for the faint of heart, the short of time, or the weak of eyesight, so think carefully before you embark on it.

Basics

When asking for, or looking at dumps there are some basic things to remember.

- The simplest is that all cman traffic goes over UDP port 6809, and it is (nearly always) broadcast. This should make it easy to find all the packets you need.
- Don't be afraid to ask for days & days of data. cman packets are not large or frequent so the files will not get too large even if you are collecting weeks worth of traffic
- It's really helpful to get captures from all nodes in the cluster. This can help determine whether packets are being lost on the network

Some commands to get you started:

```
# tcpdump -w $HOSTNAME.pcap port 6809
```

Is a simple command that will capture all cman traffic for one node. It's great for trying things out or getting evidence for an easily reproducible problem.

```
# tcpdump  -G86400 -w$HOSTNAME-%Y%m%d:%H%M%S.pcap port 6809
```

Is a handy command that will capture a day's worth of data to a file, starting a new file every 24 hours. Use this to get evidence for infrequent events, but ensure that you get ALL the files from the customer after an incident, not just the latest ones. Comment#72* applies here.

Packet protocol

The cman protocol is a simple send/ack system without any windowing. This means that for each packet sent (unless the NOACK) flag is sent, an ACK is required from the node that packet was sent to. But remember that cman can send a packet to a single node or all known nodes in the cluster. If a packet is sent to a single node, then just an ACK from that node is required. If a packet is sent to all cluster nodes then an ACK is needed from all nodes before another packet can be sent (apart from NOACK packets). ACKs can be embedded in data packets, they do not have to be in a packet on their own.

All packets have a 16 bit sequence number associated with them, so that cman can identify which packets are being acked and to identify nodes that have missed packets or received them out of sequence and take appropriate action. This is usually as simple as resending the packet. Sequence numbers should wrap around safely. Sequence number 0 is not used.

I'll repeat this in case it wasn't clear the first time, cman can not send another packet that requires an ACK, unless all the ACKs from the previous packet have been seen by the sender. This is important.

Packets are sent to 'ports' on the other nodes. A port is simply an 8 bit identifier for an application to help isolate its traffic from other applications, very similar to ports in IP traffic. Two ports are reserved for cman use, 0 is for internal 'protocol' messages, and 1 is for 'membership' messages. By far the majority of messages you will see will be membership messages so I'll concentrate on them. If things go wrong at a lower level than this you'll need to call Ghostbusters. If they go wrong in port numbers above 2 then it is probably not a cman problem, it's an application problem.

Packet structure

The packet structure is nice and simple and defined by structures and constants that you will find in the source file cman-kernel/src/cnxman-private.h. All 16 & 32 bit numbers are stored little-endian so will look backwards in a packet dump. Here's the header structure:

```
struct cl_protheader {
        unsigned char  tgtport; /* Target port number */
        unsigned char  srcport; /* Source (originating) port number */
        unsigned short seq;     /* Packet sequence number, little-endian */
        unsigned short ack;     /* Inline ACK */
        unsigned short cluster; /* Our cluster number, little-endian */
        unsigned int   flags;
        int            srcid;   /* Node ID of the sender */
        int            tgtid;   /* Node ID of the target or 0 for multicast
                                 * messages */
};
```

and here's how this looks as captured by tcpdump:

```
16:04:22.592517 IP 10.112.70.21.6809 > 10.112.70.63.6809: UDP, length 28
        0x0000:  4500 0038 0000 4000 4011 9981 0a70 4615
        0x0010:  0a70 463f 1a99 1a99 0024 0b59 0101 c498
        0x0020:  0000 f94a 0000 1900 0300 0000 0000 0000
        0x0030:  0402 0400 3b00 0000
```

I've highlighted the useful part in bold. The bits before it are the IP & UDP headers that you should be able to ignore.

From this we can see that the packet is send to and from port 1 (0101), the packet sequence number is c498 (39108 decimal), there is no inline ACK (0), the cluster number is f94a (19193), packet flags are 1900.0000 (I'll explain these later), and that the message is sent from node ID 3 to node ID 0 (the whole cluster).

Starting on the bottom line here (0x0030) are the port-specific parts of the message. As this is a membership message (port 1, remember?) the 04 tells us that this is a HELLO message. These are the messages you are likely to see most of in a working cluster.

Message Flags

These are the flags you might see in cman messages. Most are set by applications (eg membership) and dealt with by the cman sending code, so seeing them in a packet doesn't indicate much. The following two are important to know though:

MSG_NOACK – tells the receiving end that an ACK is not required.

MSG_REPLYEXP – tells the receiving end that a reply is expected so not to generate an automatic ACK packet for this message. If this flag is not set then an ACK will be sent before the message is passed to the application. If the flag is set, then the application is expected to provide an inline ack in its reply message.

```
/* Sendmsg flags, these are above the normal sendmsg flags so they don't
 * interfere */
#define MSG_NOACK     0x010000  /* Don't need an ACK for this message */
#define MSG_QUEUE     0x020000  /* Queue the message for sending later */
#define MSG_MULTICAST 0x080000  /* Message was sent to all nodes in the
                                   cluster */
#define MSG_ALLINT    0x100000  /* Send out of all interfaces */
#define MSG_REPLYEXP  0x200000  /* Reply is expected */
#define MSG_BCASTSELF 0x400000  /* Broadcast message also gets send to us */
```

Membership packets

Here are the command numbers taken from cnxman-private.h, each message has its own structure associated with it which I won't repeat here. It's as easy to have the header file handy when you're analysing packets anyway, but showing the sort of packets that get sent gives you an idea of the range of things that happen in the membership module:

```
#define CLUSTER_MEM_JOINCONF   1   /* Confirmation of JOIN REQuest */
#define CLUSTER_MEM_JOINREQ    2   /* REQuest to join a cluster */
#define CLUSTER_MEM_LEAVE      3   /* Indicate we a leaving the cluster */
#define CLUSTER_MEM_HELLO      4   /* Heartbeat message */
#define CLUSTER_MEM_KILL       5   /* Tell another node to leave the cluster
*/
#define CLUSTER_MEM_JOINACK    6   /* Acknowledge receipt of a JOINREQ */
#define CLUSTER_MEM_ENDTRANS   7   /* Finish a state transition */
#define CLUSTER_MEM_RECONFIG   8   /* Change a configuration parameter */
#define CLUSTER_MEM_MASTERVIEW 9   /* Part of transition, contains the master
node's view of the cluster */
#define CLUSTER_MEM_STARTTRANS 10  /* Start a new state transition */
#define CLUSTER_MEM_JOINREJ    11  /* Reject a JOIN REQuest */
#define CLUSTER_MEM_VIEWACK    12  /* ACK (or NACK) a MASTERVIEW */
#define CLUSTER_MEM_STARTACK   13  /* ACK (or NACK) a STARTTRANS */
#define CLUSTER_MEM_TRANSITION 14  /* not used */
#define CLUSTER_MEM_NEWCLUSTER 15  /* Tentative “is there anyone there with
my cluster ID” message */
#define CLUSTER_MEM_CONFACK    16  /* ACK (or NACK) RECONFIG */
#define CLUSTER_MEM_NOMINATE   17  /* Nominate a new master for transition */
#define CLUSTER_MEM_NODEDOWN   18  /* Tell the other nodes we noticed a node
go down */
```

A quick word here about the state transition process. When a node joins or leaves the cluster the first node to notice initiates a state transition with a STARTTRANS message. This node then becomes the master node for this state transition and will control the process. If this node crashes during the transition (or there is a clash for master nodes at the start) then a new master will be nominated and the transition process restarted from the beginning. It's important to realise that the master node is a temporary concept that only exists during a state transition and the cluster protocol is designed to cope with master failure during this time. This in no way stops the cluster software from being properly symmetrical in architecture. I've seen people fluster about terms such as “master” in a symmetrical cluster and insist that this means it's not symmetrical or that there is a single point of failure. They are idiots and they are wrong. And you can quote me on that.

The membership protocol has it's own ACK messages which are in addition to the protocol ACKs that the lower layers require. This isn't wasteful as the membership messages that require ACKs have the REPLYEXP flag set so the protocol ACK will be embedded in the membership ACK message. Membership does this because its messages can also be NACKed (refused/denied). In the examples below a “protocol ACK” is the terminology used for a separate message containing the low-level ACK on its own.

Although the bulk of a state transition is the verification of membership by all the nodes in the cluster, there is also a “Barrier” at the end of each transition. A barrier is a low-level cman function (hence it's on port 0) that ensures all nodes are synchronised and ready to start running services again. The barrier name used here is TRANSITION.%d where %d becomes the cluster incarnation number which is also printed in syslog in RHEL4.7 and later.

Barriers

Barriers are probably the only non-membership messages you will need to worry about when looking at a tcpdump from cman. They are regarded as a low-level cman operation and thus appear on port 0 of the cman traffic.

The purpose of a barrier is to synchronise operations on all nodes in the cluster. cman itself uses them to finalise a state transition, and they are also used by the service manager (which I don't cover here) to synchronise services such as the DLM and fencing systems.

Barrier function is, like all of cman in RHEL4, quite simple. Barriers have an ASCII name which they register with the cluster and then all nodes wait for all the other nodes to say that they have reached that state. The barrier structure is this:

```
struct cl_barriermsg {
        unsigned char  cmd;     /* CLUSTER_CMD_BARRIER */
        unsigned char  subcmd;  /* BARRIER sub command */
        unsigned short pad;
        unsigned int   flags;
        unsigned int   nodes;
        char name[MAX_BARRIER_NAME_LEN];
};
#define BARRIER_REGISTER 1 /* not used */
#define BARRIER_CHANGE   2 /* rarely used */
#define BARRIER_WAIT     4
#define BARRIER_COMPLETE 5
```

Barriers are best illustrated by an example, so you can see them in operation below when I walk through a real packet capture with you.

Analyzing a tcpdump from a cluster problem

Naturally, problems come in all types and it would be impossible to list all the things that can possibly happen. What you need to do is to work out where the problem seems to lie and then use that information to decide which packets (of the ones yo can see in the capture) are likely to be helpful in diagnosing the problem.

There are a few things that you can look for generally that are helpful though.

Check the sequence numbers increment sanely from all the nodes. If you see sudden changes (other than wrapping) or a node attempting to join with a sequence number not starting at 1 or an odd ACK number then that is grounds for suspicion.

If you see a lot of packets that are sent from one node and never received by another node (or ANY other node) then this could indicate a NIC problem. cman is very bad at handling these and it's quite possible for 1 or 2 bad NICs to bring down a larger cluster, especially if one node is sending but not receiving packets.

Transition restarts are usually a bad thing. It means something happened during the state transition that cman could not handle. This might be as straightforward as a node leaving the cluster during the transition, but it's nearly always worth investigating further unless you know this is the case. There will usually be messages in syslog that will help you here too.

In general, always try to correlate syslog messages (from all nodes) with the packets that you see in the capture files.

Example captures

Here are two annotated captures from a working 3 node cluster.

The first shows a node leaving the cluster after it has failed:

Two heartbeat messages. There is no heartbeat message from node 3 as it has failed

```
15:04:30.121181 IP 192.168.1.202.6809 > 192.168.1.255.6809: UDP, length 28
       0x0000:  4500 0038 0000 4000 4011 b59b c0a8 01ca
       0x0010:  c0a8 01ff 1a99 1a99 0024 b13d 0101 2e00
       0x0020:  0000 3f19 0000 1900 0200 0000 0000 0000
       0x0030:  0402 0300 0400 0000
15:04:31.138307 IP 192.168.1.201.6809 > 192.168.1.255.6809: UDP, length 28
       0x0000:  4500 0038 0000 4000 4011 b59c c0a8 01c9
       0x0010:  c0a8 01ff 1a99 1a99 0024 c13e 0101 1f00
       0x0020:  0000 3f19 0000 1900 0100 0000 0000 0000
       0x0030:  0402 0300 0400 0000
```

The first node to spot the failure sends a KILL message to node 3, just in case the node has been temporarily disconnected or can see messages. This is just a precation really. Fencing will take care of it eventually

```
15:04:31.138427 IP 192.168.1.201.6809 > 192.168.1.255.6809: UDP, length 21
       0x0000:  4500 0031 0000 4000 4011 b5a3 c0a8 01c9
       0x0010:  c0a8 01ff 1a99 1a99 001d d24e 0101 2000
       0x0020:  0900 3f19 0000 0100 0100 0000 0300 0000
       0x0030:  05
```

The node then sends a NODEDOWN message to the cluster tell the other nodes that a node has been spotted down.

```
15:04:31.139584 IP 192.168.1.201.6809 > 192.168.1.255.6809: UDP, length 28
       0x0000:  4500 0038 0000 4000 4011 b59c c0a8 01c9
       0x0010:  c0a8 01ff 1a99 1a99 0024 b71d 0101 2100
       0x0020:  0000 3f19 0000 0800 0100 0000 0000 0000
       0x0030:  1206 0f1d 0300 0000
```

Node 1 nominates itself as master for this transition and sends a STARTTRANS message with the reason REMNODE

```
15:04:31.140063 IP 192.168.1.201.6809 > 192.168.1.255.6809: UDP, length 40
       0x0000:  4500 0044 0000 4000 4011 b590 c0a8 01c9
       0x0010:  c0a8 01ff 1a99 1a99 0030 9f25 0101 2200
       0x0020:  0000 3f19 0000 2800 0100 0000 0000 0000
       0x0030:  0a02 0601 0300 0000 0500 0000 0300 0000
       0x0040:  0000 0000
```

Node 2 acknowledges receipt of the STARTTRANS message and joins in the state transition

```
15:04:31.140244 IP 192.168.1.202.6809 > 192.168.1.255.6809: UDP, length 28
       0x0000:  4500 0038 0000 4000 4011 b59b c0a8 01ca
       0x0010:  c0a8 01ff 1a99 1a99 0024 7f3a 0101 2f00
       0x0020:  2200 3f19 0000 2000 0200 0000 0100 0000
       0x0030:  0d05 0000 0500 0000
```

Node 1 sends a MASTERVIEW message showing it's view of the cluster

```
15:04:31.140458 IP 192.168.1.201.6809 > 192.168.1.255.6809: UDP, length 125
       0x0000:  4500 0099 0000 4000 4011 b53b c0a8 01c9
       0x0010:  c0a8 01ff 1a99 1a99 0085 582a 0101 2300
       0x0020:  0000 3f19 0000 2800 0100 0000 0000 0000
       0x0030:  0903 066a 6f68 616e 6e02 0100 0200 1a99
       0x0040:  c0a8 01ca 0000 0000 0000 0000 0103 0000
       0x0050:  0002 0000 0004 6d61 726b 0301 0002 001a
       0x0060:  99c0 a801 cb00 0000 0000 0000 0001 0300
       0x0070:  0000 0300 0000 066c 7564 7769 6702 0100
       0x0080:  0200 1a99 c0a8 01c9 0000 0000 0000 0000
       0x0090:  0103 0000 0001 0000 00
```

This is another STARTACK message. I'm not sure why, but I'm not going to pretend it didn't happen.

```
15:04:32.139776 IP 192.168.1.202.6809 > 192.168.1.255.6809: UDP, length 28
       0x0000:  4500 0038 0000 4000 4011 b59b c0a8 01ca
       0x0010:  c0a8 01ff 1a99 1a99 0024 9f3a 0101 2f00
       0x0020:  2200 3f19 0000 0000 0200 0000 0100 0000
       0x0030:  0d05 0000 0500 0000
```

Node 2 acknowledges the MASTERVIEW message which means that it has the same view of the cluster as the master node.

```
15:04:32.140400 IP 192.168.1.202.6809 > 192.168.1.255.6809: UDP, length 22
       0x0000:  4500 0032 0000 4000 4011 b5a1 c0a8 01ca
       0x0010:  c0a8 01ff 1a99 1a99 001e a34a 0101 3000
       0x0020:  2300 3f19 0000 0000 0200 0000 0100 0000
       0x0030:  0c01
```

Node 1 closes the transition with an ENDTRANS message

```
15:04:32.140524 IP 192.168.1.201.6809 > 192.168.1.255.6809: UDP, length 40
       0x0000:  4500 0044 0000 4000 4011 b590 c0a8 01c9
       0x0010:  c0a8 01ff 1a99 1a99 0030 c726 0101 2400
       0x0020:  0000 3f19 0000 0800 0100 0000 0000 0000
       0x0030:  0702 0000 0200 0000 0200 0000 0600 0000
       0x0040:  0000 0000
```

These next few messages are on port 0 and are barrier protocol messages. Their purpose is to synchronise the cluster so that the transition ends at the same time on all nodes. This is overkill on what is now a 2 node cluster but becomes important on larger installations.

Node 2 sends a BARRIER_WAIT message to establish the barrier

```
15:04:32.140808 IP 192.168.1.202.6809 > 192.168.1.255.6809: UDP, length 68
       0x0000:  4500 0060 0000 4000 4011 b573 c0a8 01ca
       0x0010:  c0a8 01ff 1a99 1a99 004c aeb4 0000 3100
       0x0020:  0000 3f19 0000 0800 0200 0000 0000 0000
       0x0030:  0504 0000 0000 0000 0100 0000 5452 414e
       0x0040:  5349 5449 4f4e 2e36 00de 14a0 ffff ffff
       0x0050:  0000 0000 0000 0000 4602 0000 0000 0000
```

Node 1 sends a BARRIER_WAIT message, now both nodes know the barrier is registered

```
15:04:32.141066 IP 192.168.1.201.6809 > 192.168.1.255.6809: UDP, length 68
       0x0000:  4500 0060 0000 4000 4011 b574 c0a8 01c9
       0x0010:  c0a8 01ff 1a99 1a99 004c 405e 0000 2500
       0x0020:  0000 3f19 0000 0800 0100 0000 0000 0000
       0x0030:  0504 0000 0000 0000 fbff ffff 5452 414e
       0x0040:  5349 5449 4f4e 2e36 00d0 cd1c 0001 0000
       0x0050:  c8c6 ff1f 0001 0000 4602 0000 0000 0000
```

Node 2 sends a BARRIER_COMPLETE message to complete the operation

```
15:04:32.141306 IP 192.168.1.202.6809 > 192.168.1.255.6809: UDP, length 68
       0x0000:  4500 0060 0000 4000 4011 b573 c0a8 01ca
       0x0010:  c0a8 01ff 1a99 1a99 004c 351f 0000 3200
       0x0020:  0000 3f19 0000 0800 0200 0000 0000 0000
       0x0030:  0505 0000 0000 0000 4602 0000 5452 414e
       0x0040:  5349 5449 4f4e 2e36 00f3 12a0 ffff ffff
       0x0050:  4602 0000 0000 0000 20dd 14a0 ffff ffff
```

Node 1 sends a BARRIER_COMPLETE message to complete the operation. When this is received the cluster will continue operations.

```
15:04:32.141595 IP 192.168.1.201.6809 > 192.168.1.255.6809: UDP, length 68
       0x0000:  4500 0060 0000 4000 4011 b574 c0a8 01c9
       0x0010:  c0a8 01ff 1a99 1a99 004c 4220 0000 2600
       0x0020:  0000 3f19 0000 0800 0100 0000 0000 0000
       0x0030:  0505 0000 0000 0000 4602 0000 5452 414e
       0x0040:  5349 5449 4f4e 2e36 00f3 12a0 ffff ffff
       0x0050:  4602 0000 0000 0000 20dd 14a0 ffff ffff
```

The next capture shows node 3 rejoining the cluster

We start with a couple of HELLO messages from the existing 2 cluster nodes

```
15:26:59.327999 IP 192.168.1.201.6809 > 192.168.1.255.6809: UDP, length 28
       0x0000:  4500 0038 0000 4000 4011 b59c c0a8 01c9
       0x0010:  c0a8 01ff 1a99 1a99 0024 9f3d 0101 3c01
       0x0020:  0000 3f19 0000 1900 0100 0000 0000 0000
       0x0030:  0402 0200 0a00 0000
15:27:03.310728 IP 192.168.1.202.6809 > 192.168.1.255.6809: UDP, length 28
       0x0000:  4500 0038 0000 4000 4011 b59b c0a8 01ca
       0x0010:  c0a8 01ff 1a99 1a99 0024 8c3c 0101 4e01
       0x0020:  0000 3f19 0000 1900 0200 0000 0000 0000
       0x0030:  0402 0200 0a00 0000
```

Node 3 now sends a JOINREQ message to the node who's HELLO message it just saw

```
15:27:03.310837 IP 192.168.1.203.6809 > 192.168.1.202.6809: UDP, length 89
       0x0000:  4500 0075 0000 4000 4011 b592 c0a8 01cb
       0x0010:  c0a8 01ca 1a99 1a99 0061 25d2 0101 0100
       0x0020:  0000 3f19 0000 0100 0000 0000 0000 0000
       0x0030:  0201 0100 0300 0000 0300 0000 0500 0000
       0x0040:  0000 0000 0100 0000 0100 0000 1000 0000
       0x0050:  636f
```

Node 2 accepts that node 3 can be a valid member of the cluster and sends JOINACK

```
15:27:03.311270 IP 192.168.1.202.6809 > 192.168.1.203.6809: UDP, length 22
       0x0000:  4500 0032 0000 4000 4011 b5d5 c0a8 01ca
       0x0010:  c0a8 01cb 1a99 1a99 001e ad7d 0101 4f01
       0x0020:  0000 3f19 0000 0100 0200 0000 0000 0000
       0x0030:  0601
```

Node 2 then starts a state transition with the rest of the cluster and the reason NEWNODE

```
15:27:03.311278 IP 192.168.1.202.6809 > 192.168.1.255.6809: UDP, length 61
       0x0000:  4500 0059 0000 4000 4011 b57a c0a8 01ca
       0x0010:  c0a8 01ff 1a99 1a99 0045 ae57 0101 5001
       0x0020:  0000 3f19 0000 2800 0200 0000 0000 0000
       0x0030:  0a01 0001 0300 0000 0b00 0000 0300 0000
       0x0040:  0100 01c9 0200 1a99 c0a8 01cb 0000 0000
       0x0050:  0000
```

STARTACK from node 1

```
15:27:03.311638 IP 192.168.1.201.6809 > 192.168.1.255.6809: UDP, length 28
       0x0000:  4500 0038 0000 4000 4011 b59c c0a8 01c9
       0x0010:  c0a8 01ff 1a99 1a99 0024 1912 0101 3d01
       0x0020:  5001 3f19 0000 2000 0100 0000 0200 0000
       0x0030:  0d0d 241f 0b00 0000
```

VIEWACK from node 2

```
15:27:03.311857 IP 192.168.1.202.6809 > 192.168.1.255.6809: UDP, length 92
       0x0000:  4500 0078 0000 4000 4011 b55b c0a8 01ca
       0x0010:  c0a8 01ff 1a99 1a99 0064 f149 0101 5101
       0x0020:  0000 3f19 0000 2800 0200 0000 0000 0000
       0x0030:  0903 066c 7564 7769 6702 0100 0200 1a99
       0x0040:  c0a8 01c9 0000 0000 0000 0000 0103 0000
       0x0050:  0001
```

That spare STARTACK message again. Looks like this might be a minor bug!

```
15:27:04.312158 IP 192.168.1.201.6809 > 192.168.1.255.6809: UDP, length 28
       0x0000:  4500 0038 0000 4000 4011 b59c c0a8 01c9
       0x0010:  c0a8 01ff 1a99 1a99 0024 3912 0101 3d01
       0x0020:  5001 3f19 0000 0000 0100 0000 0200 0000
       0x0030:  0d0d 241f 0b00 0000
```

Node 2 sends the MASTERVIEW message. This includes the new node to be added

```
15:27:04.312436 IP 192.168.1.202.6809 > 192.168.1.255.6809: UDP, length 92
       0x0000:  4500 0078 0000 4000 4011 b55b c0a8 01ca
       0x0010:  c0a8 01ff 1a99 1a99 0064 114a 0101 5101
       0x0020:  0000 3f19 0000 0800 0200 0000 0000 0000
       0x0030:  0903 066c 7564 7769 6702 0100 0200 1a99
       0x0040:  c0a8 01c9 0000 0000 0000 0000 0103 0000
       0x0050:  0001
```

Node 1 VIEWACKs the MASTERVIEW message

```
15:27:04.312664 IP 192.168.1.201.6809 > 192.168.1.255.6809: UDP, length 22
       0x0000:  4500 0032 0000 4000 4011 b5a2 c0a8 01c9
       0x0010:  c0a8 01ff 1a99 1a99 001e 6749 0101 3e01
       0x0020:  5101 3f19 0000 0000 0100 0000 0200 0000
       0x0030:  0c01
```

Node 2 Sends a JOINCONF to node 3 to confirm that it is now a member of the cluster

```
15:27:04.312959 IP 192.168.1.202.6809 > 192.168.1.203.6809: UDP, length 92
       0x0000:  4500 0078 0000 4000 4011 b58f c0a8 01ca
       0x0010:  c0a8 01cb 1a99 1a99 0064 1f7e 0101 5201
       0x0020:  0000 3f19 0000 0100 0200 0000 0000 0000
       0x0030:  0103 066c 7564 7769 6702 0100 0200 1a99
       0x0040:  c0a8 01c9 0000 0000 0000 0000 0103 0000
       0x0050:  0001
```

Node 3 acknowledges the JOINCONF message

```
15:27:04.313043 IP 192.168.1.203.6809 > 192.168.1.202.6809: UDP, length 21
       0x0000:  4500 0031 0000 4000 4011 b5d6 c0a8 01cb
       0x0010:  c0a8 01ca 1a99 1a99 001d f281 0101 0200
       0x0020:  0000 3f19 0000 0100 0000 0000 0000 0000
       0x0030:  10
```

Node 2 wraps up the transition with an ENDTRANS message

```
15:27:04.313394 IP 192.168.1.202.6809 > 192.168.1.255.6809: UDP, length 40
       0x0000:  4500 0044 0000 4000 4011 b58f c0a8 01ca
       0x0010:  c0a8 01ff 1a99 1a99 0030 8d26 0101 5301
       0x0020:  0000 3f19 0000 0800 0200 0000 0000 0000
       0x0030:  0700 0000 0200 0000 0300 0000 0c00 0000
       0x0040:  0300 0000
```

This is a protocol ACK for the ENDTRANS

```
15:27:04.313457 IP 192.168.1.203.6809 > 192.168.1.202.6809: UDP, length 24
       0x0000:  4500 0034 0000 4000 4011 b5d3 c0a8 01cb
       0x0010:  c0a8 01ca 1a99 1a99 0020 b07b 0000 0000
       0x0020:  5301 3f19 0000 0100 0000 0000 0000 0000
       0x0030:  0101 00ff
```

Then we have a 3-way barrier that ensures the new cluster is synchronised:
Node 1 Barrier WAIT

```
15:27:04.313632 IP 192.168.1.201.6809 > 192.168.1.255.6809: UDP, length 68
       0x0000:  4500 0060 0000 4000 4011 b574 c0a8 01c9
       0x0010:  c0a8 01ff 1a99 1a99 004c 7537 0000 3f01
       0x0020:  0000 3f19 0000 0800 0100 0000 0000 0000
       0x0030:  0504 0000 0000 0000 0100 0000 5452 414e
       0x0040:  5349 5449 4f4e 2e31 3200 0000 0000 0000
       0x0050:  1000
```

Protocol ACK (3f01) to barrier wait
15:27:04.313681 IP 192.168.1.203.6809 > 192.168.1.201.6809: UDP, length 24

```
       0x0000:  4500 0034 0000 4000 4011 b5d4 c0a8 01cb
       0x0010:  c0a8 01c9 1a99 1a99 0020 c57c 0000 0000
       0x0020:  3f01 3f19 0000 0100 0000 0000 0000 0000
       0x0030:  0100 0000
```

Node 3 Barrier WAIT

```
15:27:04.313797 IP 192.168.1.203.6809 > 192.168.1.255.6809: UDP, length 68
       0x0000:  4500 0060 0000 4000 4011 b572 c0a8 01cb
       0x0010:  c0a8 01ff 1a99 1a99 004c b035 0000 0300
       0x0020:  0000 3f19 0000 0800 0300 0000 0000 0000
       0x0030:  0504 0000 0000 0000 0100 0000 5452 414e
       0x0040:  5349 5449 4f4e 2e31 3200 0000 0001 0000
       0x0050:  0300
```

Node 2 Barrier WAIT

```
15:27:04.313973 IP 192.168.1.202.6809 > 192.168.1.255.6809: UDP, length 68
       0x0000:  4500 0060 0000 4000 4011 b573 c0a8 01ca
       0x0010:  c0a8 01ff 1a99 1a99 004c 17bd 0000 5401
       0x0020:  0000 3f19 0000 0800 0200 0000 0000 0000
       0x0030:  0504 4922 6901 0000 f017 b31f 5452 414e
       0x0040:  5349 5449 4f4e 2e31 3200 f31c 0001 0000
       0x0050:  1000
```

Protocol ACK (5401) to barrier wait

```
15:27:04.314013 IP 192.168.1.203.6809 > 192.168.1.202.6809: UDP, length 24
       0x0000:  4500 0034 0000 4000 4011 b5d3 c0a8 01cb
       0x0010:  c0a8 01ca 1a99 1a99 0020 ad7b 0000 0000
       0x0020:  5401 3f19 0000 0100 0300 0000 0000 0000
       0x0030:  0100 0000
```

Protocol ACK (0100) to barrier wait

```
15:27:04.314151 IP 192.168.1.201.6809 > 192.168.1.203.6809: UDP, length 24
       0x0000:  4500 0034 0000 4000 4011 b5d4 c0a8 01c9
       0x0010:  c0a8 01cb 1a99 1a99 0020 007e 0000 0000
       0x0020:  0300 3f19 0000 0100 0100 0000 0000 0000
       0x0030:  0100 0000
```

Node 1 Barrier COMPLETE

```
15:27:04.314183 IP 192.168.1.201.6809 > 192.168.1.255.6809: UDP, length 68
       0x0000:  4500 0060 0000 4000 4011 b574 c0a8 01c9
       0x0010:  c0a8 01ff 1a99 1a99 004c f716 0000 4001
       0x0020:  0000 3f19 0000 0800 0100 0000 0000 0000
       0x0030:  0505 0000 0000 0000 4602 0000 5452 414e
       0x0040:  5349 5449 4f4e 2e31 3200 12a0 ffff ffff
       0x0050:  4602
```

Protocol ACK (4001) to barrier complete

```
15:27:04.314231 IP 192.168.1.203.6809 > 192.168.1.201.6809: UDP, length 24
       0x0000:  4500 0034 0000 4000 4011 b5d4 c0a8 01cb
       0x0010:  c0a8 01c9 1a99 1a99 0020 c17c 0000 0000
       0x0020:  4001 3f19 0000 0100 0300 0000 0000 0000
       0x0030:  0100 0000
```

Protocol ACK (0300) to barrier wait

```
15:27:04.314406 IP 192.168.1.202.6809 > 192.168.1.203.6809: UDP, length 24
       0x0000:  4500 0034 0000 4000 4011 b5d3 c0a8 01ca
       0x0010:  c0a8 01cb 1a99 1a99 0020 ff7c 0000 0000
       0x0020:  0300 3f19 0000 0100 0200 0000 0000 0000
       0x0030:  0100 0000
```

Node 3 Barrier COMPLETE

```
15:27:04.314485 IP 192.168.1.203.6809 > 192.168.1.255.6809: UDP, length 68
       0x0000:  4500 0060 0000 4000 4011 b572 c0a8 01cb
       0x0010:  c0a8 01ff 1a99 1a99 004c 3116 0000 0400
       0x0020:  0000 3f19 0000 0800 0300 0000 0000 0000
       0x0030:  0505 0000 0000 0000 4602 0000 5452 414e
       0x0040:  5349 5449 4f4e 2e31 3200 12a0 ffff ffff
       0x0050:  4602
```

Node 2 Barrier COMPLETE

```
15:27:04.314701 IP 192.168.1.202.6809 > 192.168.1.255.6809: UDP, length 68
       0x0000:  4500 0060 0000 4000 4011 b573 c0a8 01ca
       0x0010:  c0a8 01ff 1a99 1a99 004c e115 0000 5501
       0x0020:  0000 3f19 0000 0800 0200 0000 0000 0000
       0x0030:  0505 0000 0000 0000 4602 0000 5452 414e
       0x0040:  5349 5449 4f4e 2e31 3200 12a0 ffff ffff
       0x0050:  4602
```

Protocol ACK (5501) to barrier complete

```
15:27:04.314749 IP 192.168.1.203.6809 > 192.168.1.202.6809: UDP, length 24
       0x0000:  4500 0034 0000 4000 4011 b5d3 c0a8 01cb
       0x0010:  c0a8 01ca 1a99 1a99 0020 ac7b 0000 0000
       0x0020:  5501 3f19 0000 0100 0300 0000 0000 0000
       0x0030:  0100 0000
```

Protocol ACK (0400) to barrier complete

```
15:27:04.314922 IP 192.168.1.201.6809 > 192.168.1.203.6809: UDP, length 24
       0x0000:  4500 0034 0000 4000 4011 b5d4 c0a8 01c9
       0x0010:  c0a8 01cb 1a99 1a99 0020 ff7d 0000 0000
       0x0020:  0400 3f19 0000 0100 0100 0000 0000 0000
       0x0030:  0100 0000
```

Protocol ACK (0400) to barrier complete

```
15:27:04.314934 IP 192.168.1.202.6809 > 192.168.1.203.6809: UDP, length 24
       0x0000:  4500 0034 0000 4000 4011 b5d3 c0a8 01ca
       0x0010:  c0a8 01cb 1a99 1a99 0020 fe7c 0000 0000
       0x0020:  0400 3f19 0000 0100 0200 0000 0000 0000
       0x0030:  0100 0000
```

And we all start sending HELLO messages again

```
15:27:04.328114 IP 192.168.1.201.6809 > 192.168.1.255.6809: UDP, length 28
       0x0000:  4500 0038 0000 4000 4011 b59c c0a8 01c9
       0x0010:  c0a8 01ff 1a99 1a99 0024 973d 0101 4101
       0x0020:  0000 3f19 0000 1900 0100 0000 0000 0000
       0x0030:  0402 0300 0c00 0000
```

Footnotes

* Comment#72 (https://bugzilla.redhat.com/show_bug.cgi?id=485026#c72) states

> *“Please, please *always* attach full logs. I'd much rather have 2GB of log files to wade through than 1K of truncated logs that don't show what I'm looking for.*
>
> *I'm very good at filtering log files, it's my job and I've been doing it for a very long time now! And it's quite possible that I might spot something important that looks insignificant to you”*