High Availability with PostgreSQL and Pacemaker

Author: Shaun Thomas
Date: September 18th, 2012
Venue: Postgres Open 2012

Your Presenter

Shaun M. Thomas
Senior Database Administrator
Peak6 OptionsHouse

I apologize in advance.

High Availability?

- Stay On-line
- Survive Hardware Failures
- No Transaction Loss

High availability means a lot of things, but the primary meaning is contained within the name. Keeping a database available at all times can be a rough challenge, and for certain environments, redundancy is a must. Two of everything is the order of the day. This sounds expensive at first, but a hardware failure at the wrong time can result in millions of dollars in loss for some business cases. This doesn't even account for credibility and reputation lost in the process.

But keeping two machines synchronized is no easy task. Some solutions seek to share a single disk device, mounted by two servers that can replace each other during failures or upgrades. Others opt for duplicating the machines *and* the disks, to prevent a single point of failure such as a SAN or other attached storage. Whatever the case, some external management is necessary to avoid losing transactions during network transmission.

That's really what this whole presentation is about. Just showing one way the problem can be solved. We chose this because everything is an off-the-shelf component, and for larger enterprises, it is a common approach to data survival.

Why Not Replication?

- Trigger Based = Inexact
- Multicast = Inexact
- Log Shipping = Transaction Loss
- Asynchronous = Transaction Loss
- Synchronous …
    - Commit lag
    - Slaves affect master

This is probably the most important part of a high availability choice. There are a lot of ways to do it, and not all are created equal. Some companies like to use trigger based tools such as Bucardo or Slony. Yet these tools have problems with side-effects such as triggers in complex schemas. The same can be said for multicast tools like PGPool. There's a reason Multi-Master replication is hard, and a lot of that has to do with transaction isolation and completion. Both of which are intrinsically violated in these kinds of setup, unless every single table and sequence is replicated.

So what about the PostgreSQL transaction logs? The ones we use for warm and hot standbys, and point in time recovery, and any number of things? The files that presumably allow block-level replay of database activity on all log consumers? They're perfect for the job of making sure data is accurately represented, but do not guarantee it does so in a timely fashion. Log shipping and asynchronous replication will both involve unverified transmission and replay, which means high throughput systems are still at risk.

Which leaves synchronous replication. With Postgresql 9.1, this is a viable solution, but it has a couple important caveats: transmission must be acknowledged before transaction commit, and transmission does *not* guarantee replay. This means the master node can and will hang until a slave both receives the transaction, and says so. If the slave goes down, the master can hang indefinitely. This actually doubles risk of downtime, and in a high availability context, is a steep price to pay for transaction safety.

So if all the built-in and commonly associated tools aren't good enough, what's left?

Why DRBD?

- Networked RAID-1
- Hidden from PG
- Sync writes to both nodes
- Automatic re-synchronization

DRBD stands for Distributed Replicating Block Device and can solve several of the replication problems we mentioned earlier. First and foremost, it's the equivalent of a network-driven RAID-1, but at the block level, existing below the OS filesystem. What does this mean? A default DRBD synchronization setting means a filesystem sync, which PostgreSQL calls at several points to ensure transaction safety and data integrity. A sync must be acknowledged by both nodes before it succeeds. So just like synchronous replication, no committed transaction can be lost, because it's written to both nodes.

But what happens when a node becomes unavailable? While PostgreSQL synchronous replication would need a configuration reload to prevent hanging transactions, DRBD will operate just like a degraded RAID-1. This means it will operate on the remaining disk, and monitoring tools should inform operators that a node is unavailable—if they don't already know. When a node re-appears, DRBD will replay or rebuild until the offline node is again in sync. Again, this is just like a regular disk mirror.

So what have we gained? An offline slave won't cause the master to hang, because indeed, PostgreSQL doesn't even know about DRBD at all. And unlike PostgreSQL synchronous replication, both nodes have exactly the same data at all times, because every block written to disk is faithfully written to both nodes simultaneously. Replication would have to replay the acknowledged transaction on the slave, while DRBD does not.

Why Pacemaker?

- Replaces you
- Automated
- Fast

To complete the picture, we have Pacemaker. It does the job you or a Unix administrator would normally do in the case of a node failure: monitor for failure, switch everything over to the spare node, do so as quickly as possible. This is why having a spare node is so important. While diagnostics and other forensics are performed on the misbehaving system, all services are still available. In most other contexts, a website, financial application, or game platform would simply be inoperable.

There are other tools to do automatic cluster management. Redhat Cluster Suite is another popular choice. SIOS provides LifeKeeper as a commercial solution. So why Pacemaker? It's a fairly popular community-driven tool. While Redhat Cluster Suite isn't entirely tied to Redhad derived Linux flavors, it's definitely developed with them in mind. Pacemaker is a little more platform agnostic, and like Redhat Cluster, free.

So, down to business.

Sample Cluster

- Basic 2-node cluster
- Built on a VirtualBox VM
- Ubuntu 12.04 LTS
- /dev/sdb as "external" device
- PostgreSQL 9.1 from repo
- DRBD + Pacemaker from repo

We're going to keep the system as simple as possible. We have a VM with a popular, recent Linux distribution. We have a block device for DRBD to manage, and represents an internal or external RAID, NVRAM device, or SAN. And we have PostgreSQL, DRBD, and Pacemaker. All the basic building-blocks of a PostgreSQL cluster.

Before we do anything, we'll try to make sure we cover prerequisites. And while we're running this example on Ubuntu, most, if not all of this presentation should apply equally well to CentOS, RHEL, Arch Linux, Gentoo, Debian, or any other distribution. Though we'll make relevant notes where necessary.

LVM and DRBD

- Get LVM to ignore DRBD devices.
- Turn off LVM Caching
- Update Linux RAMfs

We'll be doing all of this as a root user on both nodes!

LVM, the Linux Volume Manager, makes it easier to manage DRBD resources. We're going to use it, but we first have to tell LVM not to directly try and manage the device we're giving to DRBD. Having both pieces trying to claim it causes very odd behavior that will befuddle and frustrate admins of all flavors. So let's fix that by changing a line in */etc/lvm/lvm.conf*:

```
filter = [ "r|/dev/sdb|", "r|/dev/disk/*|", "r|/dev/block/*|", "a|.*|" ]
```

Likewise, the DRBD device is likely to disappear and reappear at will, so we don't want LVM to cache the devices it sees at any time. Why? Because the cache may cause LVM to assume the DRBD device is available when it isn't, and vice versa. One more line in */etc/lvm/lvm.conf*:

```
write_cache_state = 0
```

Finally, LVM is capable of seeing devices before the OS itself is available, for obvious reasons. It's very likely our chosen device of */dev/sdb* was caught by LVM's old filter line, so we need to regenerate the kernel device map. So at the command line:

```
update-initramfs -u
```

Start setting up DRBD

- Install tools
- Configure DRBD device
- Remove DRBD from startup

For all of these steps, we'll again be running as root.

DRBD itself is built into recent Linux kernels in most distributions, But the tools to manage it are not installed by default. So let's go get them:

```
apt-get install drbd8-utils
```

We've already done this for the sake of time. But what we haven't done, is configure the DRBD device. Basically, we need to create a named resource and tell it which nodes are involved. We'll also set a couple of other handy options while we're at it. Note that we're using DRBD 8.3, which is built into Ubuntu 12.04. DRBD 8.4 is only available in the most cutting-edge distribution, so when you're reading DRBD documentation, the recent docs refer to 8.4 which actually has much different syntax than 8.3.

So we'll want a file named */etc/drbd.d/pg.res* on both nodes:

```
resource pg {
  device minor 0;
  disk /dev/sdb;

  syncer {
    rate 150M;
    verify-alg md5;
  }

  on HA-node1 {
    address 192.168.56.10:7788;
    meta-disk internal;
  }
  on HA-node2 {
    address 192.168.56.20:7788;
    meta-disk internal;
  }
}
```

Notice that we gave the entire device to DRBD without partitioning it. This is completely valid, so don't fret. We do however, need to remove DRBD from the list of startup and shutdown services, because pacemaker will do that for us. In Ubuntu, we do this:

```
update-rc.d drbd disable
```

Initialize DRBD

- Create the device
- Turn it on
- Pick your favorite
- Watch it sync

Now that we have a config file and we don't have to worry about DRBD taking over on reboot, we need to actually get DRBD to provision the device for itself. Luckily, that's darn easy:

```
drbdadm create-md pg
drbdadm up pg
```

In most cases, you'll want to start on node 1, just for the sake of convenience. DRBD 8.4 has better syntax for this, but the next step basically involves telling one node to take over as the primary resource, while also telling it to overwrite any data on the other node. Before we do that though, we need to start the actual DRBD service on each node to manage the data transactions:

```
service drbd start
drbdadm -- --overwrite-data-of-peer primary pg
```

Watch DRBD sync for a while:

```
watch cat /proc/drbd
```

Until it looks like this:

```
version: 8.3.11 (api:88/proto:86-96)
srcversion: 71955441799F513ACA6DA60
 0: cs:Connected ro:Primary/Secondary ds:UpToDate/UpToDate C r-----
    ns:511948 nr:0 dw:0 dr:512612 al:0 bm:32 lo:0 pe:0 ua:0 ap:0 ep:1 wo:f oos:0
```

The Primary/Secondary part means the node you're on is the primary, and the "other" node is secondary. The second node will say the opposite: Secondary/Primary. The */proc/drbd* device is the easiest way to get the current state of DRBD's sync.

A Quick Aside

Performance!

- no-disk-barrier
- no-disk-flushes
- no-disk-drain

We just wanted to make sure you knew about these, because on many "big iron" systems with RAID controllers featuring battery backed caches, or NVRAM devices with super-capacitors, or expensive SANs, write barriers and disk flushes are generally optimized by the controllers or drivers. We don't need DRBD doing it too!

However, pay attention to disk-drain. Never, ever disable this unless you *really, really* know what you're doing. Disk-drain ensures no reordering of block writes. Disabling this means writes can happen out of order, and that is almost never a good thing. Seriously, never use the no-disk-drain option.

None of this matters for our VM, of course, but it's something to keep in mind for your own setup.

So, we have a DRBD device… what now? LVM!

Set up LVM

- Create a physical device
- Create a volume group
- Create a logical volume

We highly encourage you to look up LVM when you get a chance. There's a lot of power there, such as breaking a device apart into several resizable partitions, inclusion of new devices into the same volume as a JBOD, device snapshots, and so on. For now, we just need a basic setup. Now, our sample device is 500MB, so we'll create a single device, group, and volume, and save 50MB for snapshots.

Create it all, only on the primary node:

```
pvcreate /dev/drbd0
vgcreate VG_PG /dev/drbd0
lvcreate -L 450M -n LV_DATA VG_PG
```

Note we saved some room for snapshots, if we ever want those. On a larger system, you might reserve more space for byte-stable backups, for instance.

Set up Filesystem

- Create a mount point
- Format LVM device
- Mount!

Start with a mount-point on both systems. Clearly we'll want this around for when Pacemaker switches the services from server to server:

```
mkdir -p -m 0700 /db/pgdata
```

Now, for our example, we're using XFS. With XFS, we need to make sure the xfsprogs package is installed, as it actually contains all the useful formatting and management tools:

```
apt-get install xfsprogs
```

For you RHEL license users, this is a very expensive thing to do. For our VM, it doesn't matter, but you may need to take more into consideration. So, since XFS gives us the ability to have parallel access to the underlying block device, let's take advantage of that ability while formatting the filesystem. We only need to do this on the current primary node:

```
mkfs.xfs -d agcount=8 /dev/VG_PG/LV_DATA
```

These make it easier for parallel accesses of the same filesystem. The default isn't quite high enough. Try to have at least one per CPU. Next, we should mount the filesystem and also change the mounted ownership so PostgreSQL, for obvious reasons. Once again, only on the primary node:

```
mount -t xfs -o noatime,nodiratime,attr2 /dev/VG_PG/LV_DATA /db/pgdata
chown postgres:postgres /db/pgdata
```

```
chmod 0700 /db/pgdata
```

So now we have a mounted, formatted, empty filesystem, ready for PostgreSQL. Instead of jumping right to setting up Pacemaker, let's get PG running so we can see DRBD in action.

Get PG Working

- Initialize the Database
- Fix the Config
- Start Postgres
- Create Sample DB

Make sure we have the essentials for PostgreSQL:

```
apt-get install postgresql-9.1
apt-get install postgresql-contrib-9.1
```

Sounds easy, no? For our Ubuntu setup, it's not quite. Ubuntu added cluster control scripts so multiple installs can exist on the same server. This kind of usage pattern isn't really that prevalent. So we're going to do a couple things first. Let's kill the default config and create a new one for ourselves:

```
pg_dropcluster 9.1 main
pg_createcluster -d /db/pgdata \
    -s /var/run/postgresql 9.1 hapg
```

Oddly enough, this should happen on both nodes so the config files all match up. Afterwards on node 2, delete the actual data files it created, since Pacemaker will need exclusive access. So, on node 2:

```
rm -Rf /db/pgdata/*
```

With any other system, such as CentOS, we could just use initdb directly, and leave the config files in */db/pgdata*. But we're trying to use as many packaged tools as possible. Feel free to ignore the built-in tools on your own setup! While we're configuring, modify */etc/postgresql/9.1/hapg/postgresql.conf* and change *listen_addresses* to '*'. We're going to be assigning a virtual IP, and it would be nice if PostgreSQL actually used it.

Next, we need to ensure PostgreSQL does not start with the system, because pacemaker will do that for us. On both nodes:

```
update-rc.d postgresql disable
```

Finally, on node 1, start PG and create a sample database with pgbench. We're going to only use a very small sample size for the sake of time.

Please note, Ubuntu doesn't put pgbench in the system path, so for the purposes of this demonstration, we added */usr/lib/postgresql/9.1/bin* to the postgres user's path:

```
service postgresql start
su -c 'createdb pgbench' - postgres
su -c 'pgbench -i -s 5 pgbench' - postgres
```

Set up Pacemaker

- Install Corosync
- Install Pacemaker
- Configure Corosync
- Start Corosync
- Watch it Grow

Wait, what's this "corosync" thing? It's the communication layer between nodes in the cluster, and also makes sure Pacemaker is running. It basically bootstraps everything. So make sure both it and Pacemaker are installed:

```
apt-get install corosync pacemaker
```

So how do we set up Corosync? For our purposes, we just have to tell it the location of our network it can bind to for UDP traffic. The defaults are fine, otherwise. So modify */etc/corosync/corosync.conf* on both servers:

```
bindnetaddr: 192.68.56.0
```

Why that address? We set up Virtualbox with a virtual network, and in this case, the bindnetaddr parameter integrates the net mask. That's your actual broadcast address, and all defaults otherwise. Do that on both nodes. It'll be different based on whether or not you have a second NIC. It's pretty common to use a 10G direct link between nodes for this. Ditto for DRBD.

Now set corosync to start, and start it on both nodes:

```
sed -i 's/=no/=yes/' /etc/default/corosync
service corosync start
```

Monitor corosync until both nodes appear. We can do this on any node:

```
crm_mon
```

It'll have some declarative header and this:

```
Online: [ ubuntu-node-2 ubuntu-node-1 ]
```

When that shows up, we're ready to get rockin'.

Pacemaker Defaults

- Disable STONITH
- Disable Quorum
- Enable Stickiness

STONITH stands for: Shoot The Other Node In The Head, and is otherwise known as fencing, to fence off the misbehaving server. It is a mechanism by which nodes can be taken offline following an automatic or managed failover. This disallows the possibility of an accidental takeover or cluster disruption when the other node is returned to service. We're just a VM, so there's really no risk here. On a bigger setup, it's a good idea to evaluate these options. And what options they are! Power switches can be toggled, network switches can permanently disconnect nodes, and any plethora of untimely death can block a bad node from making a

situation worse. But for these circumstances, it's just in the way.

This can be disabled as such:

```
crm configure property stonith-enabled="false"
```

Quorum in the context of Pacemaker is a scoring system used to determine the health of the cluster. Effectively this means *more* than half of a cluster must be alive at all times. For a two-node cluster such as our VM, this means an automatic failover will not occur unless we disable Pacemaker's quorum policy.

So to disable quorum policy:

```
crm configure property no-quorum-policy="ignore"
```

This ensures that the cluster will operate even when reduced to one operational node.

By default, Pacemaker is a transient system, running services on any available node whenever possible, with a slight bias toward the original start system. Indeed, on massive clusters composed of dozens of nodes, services could be spread all over the systems so long as they meet a certain critical threshold. With a two-node cluster however, this is problematic, especially when doing maintenance that returns an offline node into service. Pacemaker may decide to move the resource back to the original node, disrupting service.

What we want is a "sticky" move. Meaning, once a node becomes unavailable, or a failover occurs for any other reason, all migrated services should remain in their current location unless manually overridden. We can accomplish this by assigning a default run score, that gets assigned to every service for the current running node.

crm configure property default-resource-stickiness="100"

With this in place, returning an offline node to service, even without STONITH, should not affect Pacemaker's decision on where resources are currently assigned.

Manage DRBD

- Add DRBD to Pacemaker
- Add Master / Slave Resource

Pacemaker configs are comprised primarily of primitives. With DRBD, we also have a master/slave resource, which controlls which node is primary, and which is secondary. So for DRBD, we need to add both. Luckily, we can utilize crm like we did with the Pacemaker defaults. So, let's create a primitive for DRBD:

```
crm configure primitive drbd_pg ocf:linbit:drbd \
  params drbd_resource="pg" \
  op monitor interval="15" \
  op start interval="0" timeout="240" \
  op stop interval="0" timeout="120"
```

We also need to define a resource that can promote and demote the DRBD service on each node, keeping in mind the service needs to run on both nodes at all times, just in a different state. So we'll now add a master/slave resource:

```
crm configure ms ms_drbd_pg drbd_pg \
  meta master-max="1" master-node-max="1" clone-max="2" \
       clone-node-max="1" notify="true"
```

And that's it. Now we can send commands such as "promote" or "demote" to the master/slave resource, and it will make sure the binary state of DRBD properly follows the "online" node.

Filesystem Related

- Manage LVM
- Manage Filesystem

Next comes LVM, the Linux Volume Manager. Due to how DRBD works, the actual volume will be invisible on the secondary node, meaning it can't be mounted or manipulated in any way. LVM scans following a DRBD promotion fix this problem, by either ensuring the corresponding volume group is available, or stopping the process. We also get the benefit of LVM snapshots, and the ability to move away from DRBD if some other intermediary device comes into play.

Help LVM find DRBD's devices:

```
crm configure primitive pg_lvm ocf:heartbeat:LVM \
  params volgrpname="VG_PG" \
  op start interval="0" timeout="30" \
  op stop interval="0" timeout="30"
```

Now pacemaker will search for usable volumes which live on DRBD devices, and will only be available following a DRBD resource promotion.

Though we use XFS, and the existing DRBD and LVM resources are formatted for XFS with our default settings, it still needs to be mounted. At this point, the primary is just a long-winded way of defining mount options to Pacemaker following a failover:

```
crm configure primitive pg_fs ocf:heartbeat:Filesystem \
  params device="/dev/VG_PG/LV_DATA" directory="/db/pgdata" \
         options="noatime,nodiratime" fstype="xfs" \
  op start interval="0" timeout="60" \
  op stop interval="0" timeout="120"
```

Thankfully, that was the hard part. Getting the base filesystem takes so many steps because we have a fairly large stack. DRBD is under everything, next it's LVM so we get snapshot capabilities and a certain level of isolation to make the device invisible on the "offline" node. And then it has to be mounted.

Now we should be safe to set up PostgreSQL itself.

Manage PostgreSQL

- Fix PostgreSQL Init Script
- Add PostgreSQL Resource

LSB stands for Linux Standard Base. Pacemaker relies on this standard for error codes to know when services are started, if they're running, and so on. It's critical all scripts that handle common services are either built-in to pacemaker, or follow the LSB standard.

As a special note about Ubuntu, its supplied postgresql startup script *is not* LSB compliant. The 'status' command has "set +e" defined because it is meant to check the status of all cluster elements. Due to this, Pacemaker never receives the expected '3' for an offline status on the secondary server. This makes it think the service is running on two nodes, and will cause it to restart the primary running node in attempt to rectify. We strongly suggest you obtain a proper PostgreSQL LSB startup script, or remove the "set +e" line from */etc/init.d/postgresql* and never run more than one concurrent pacemaker-controlled copy of PostgreSQL on that server.

The PostgreSQL Wiki has a handy LSB-compliant script, if the supplied one isn't working right. Whatever the case, make sure your init script supports LSB, or Pacemaker will do all manner of odd things trying to control PostgreSQL. So, add the primitive:

```
crm configure primitive pg_lsb lsb:postgresql \
  op monitor interval="30" timeout="60" \
  op start interval="0" timeout="60" \
  op stop interval="0" timeout="60"
```

This primitive will cause Pacemaker to check the health of PG every 30 seconds, and no check should take longer than 60 due to a timeout. The start and stop timeouts are slightly extended to account for checkpoints during shutdown, or recovery during startup. Feel free to experiment with these, as they're the primary manner Pacemaker uses to determine the availability of PostgreSQL.

Take a Name

- Dedicate an IP Address
- Add VIP Resource

To make sure applications don't have to change their configurations every time a failover occurs, we supply a "virtual" IP address, that can be redirected to the active node at any time. This allows DNS to point to the virtual IP address so a cluster name can be used as well. This is fairly straight forward:

```
crm configure primitive pg_vip ocf:heartbeat:IPaddr2 \
  params ip="192.168.56.30" iflabel="pgvip" \
  op monitor interval="5"
```

We don't specify a separate ARP resource here, because the IPAddr2 agent automatically sends five unsolicited ARP packets. This should be more than enough, but if not, we can always increase this default.

As usual, if you have a secondary adapter used specifically for this, use that. In the case of our VM, we're using eth1 instead of eth0 because we statically assigned these IPs.

Put it Together

- Create a Group
- Bolt DRBD to Group
- Force a Start Order

Three things remain in this Pacemaker config, and all of them relate to gluing everything together so it all works in the expected order. We'll make limited use of colocations, and concentrate mostly on a resource group, and a process order.

First, we need a group collecting the filesystem and all PG-related services together:

```
crm configure group PGServer pg_lvm pg_fs pg_lsb pg_vip
```

Next, we want to associate the group with DRBD. This way, those things can happen in parallel, but follow dependency rules:

```
crm configure colocation col_pg_drbd inf: PGServer \
    ms_drbd_pg:Master
```

The “Master” specification means PGServer, our group, depends on the master/slave setup reporting a Master status, which should only happen on the active node. This also reads a bit backward, but can be interpreted as “PGServer depends on DRBD Master.”

Finally, we need a startup order, so Pacemaker knows which order to stop and start services:

```
crm configure order ord_pg inf: ms_drbd_pg:promote \
    PGServer:start
```

The “:start” appended to the end of PGServer is *very important*. Without it, the state of the group and its contents are ambiguous, and Pacemaker may choose to start or stop the services after choosing targets seemingly at random. This will nearly always result in a broken final state, requiring manual cleanup.

Let’s Break Stuff

- Take a node Offline
- Manual Failover
- Secondary crash
- Primary crash
- Force Data Sync

So now we can explore a little of the payoff. Now that we’ve spent all this time getting DRBD and Pacemaker to work, why not break it? Let’s start with something simple, such as taking the standby node offline:

```
crm node standby HA-node2
```

Now our crm_mon says this:

```
Node HA-node2: standby
Online: [ HA-node1 ]
```

And we can fix it by bringing the node online again:

crm node online HA-node2

That was boring! Let’s invoke a failover while paying attention to crm_mon:

```
crm resource migrate PGServer HA-node2
```

That was pretty fast. Let’s do it again, and also unmigrate the resource afterwards. The migrate command is an absolute. If the target of a migrate becomes unavailable, Pacemaker will still move the cluster to an alternate node, but if it ever returns, Pacemaker will move it, thinking its the preferred target. Why? Migrate basically gives an infinite resource score to the named node, which is way higher than our sticky setting. Unmigrate removes that for us. So, back to node 1:

```
crm resource migrate PGServer HA-node1
crm resource unmigrate PGServer
```

Now let’s be a little more destructive. We’re not even going to shut down the secondary, but kill the VM outright. Keep an eye on crm_mon, and it’ll say this pretty quickly:

```
Online: [ HA-node1 ]
OFFLINE: [ HA-node2 ]

 Master/Slave Set: ms_drbd_pg [drbd_pg]
     Masters: [ HA-node1 ]
     Stopped: [ drbd_pg:1 ]
```

Pacemaker knew almost immediately that the node was dead, and further, that half of DRBD was broken. So let's restart the VM and watch it re-integrate:

```
Online: [ HA-node1 HA-node2 ]

 Master/Slave Set: ms_drbd_pg [drbd_pg]
     Masters: [ HA-node1 ]
     Slaves: [ HA-node2 ]
```

Better! But who cares if the secondary fails? What happens if the primary becomes unresponsive? Again, we'll kill the VM directly:

```
Online: [ HA-node2 ]
OFFLINE: [ HA-node1 ]

 Master/Slave Set: ms_drbd_pg [drbd_pg]
     Masters: [ HA-node2 ]
     Stopped: [ drbd_pg:0 ]
 Resource Group: PGServer
     pg_lvm     (ocf::heartbeat:LVM):   Started HA-node2
     pg_fs      (ocf::heartbeat:Filesystem):    Started HA-node2
     pg_lsb     (lsb:postgresql):       Started HA-node2
     pg_vip     (ocf::heartbeat:IPaddr2):       Started HA-node2
```

So, not only did Pacemaker detect the node failure, but it migrated the resources. Let's alter some of the data by doing a few pgbench runs. This will make the data itself out of sync, forcing DRBD to catch up. Some pgbench on node 2:

```
su -c 'pgbench -T 30 pgbench' - postgres
```

And let's restart node 1, but instead of watching Pacemaker, let's watch DRBD's status. It's pretty interesting:

```
watch -n 2 cat /proc/drbd
```

The part that really concerns us is at the bottom:

```
[====>..............] sync'ed: 25.0% (333636/440492)K
finish: 0:00:06 speed: 53,428 (53,428) K/sec
```

DRBD clearly knew the data wasn't up to sync, and transmitted all missing blocks, so that the remote store matched the primary. Node 2 also remained as the primary, exactly as our sticky setting dictated. Node 1 is now a byte-for-byte copy of node 2, and PostgreSQL has no idea.

Obviously such a sync would take longer with several GB or TB of data, but the concept is the same. Only the blocks that are out of sync are transmitted. Though DRBD does offer such capabilities as invalidating the data on a node so it re-syncs everything

from scratch, or doing a full data validation so all checksums are verified for every block, but that is more of a maintenance procedure.

There's a lot more to explore, and we encourage you to do so!

More Information

- PostgreSQL LSB Init Script
- Pacemaker Explained
- Clusters from Scratch
- OCF Specification
- LSB Specification
- CRM CLI Documentation
- DRBD Documentation
- LVM HOWTO

Questions