The road to high availability: solutions built on open source

Dejan Muhamedagic, dejan@suse.de

v1.0

Abstract

This paper is about high availability and failover clusters. High availability has always been a difficult and costly business and the road bumpy and time consuming. But in today's highly computerized society, many organizations want their services to be available 24 hours a day and are willing to spend money and time to achieve it.

The text is divided into three parts: The first part discusses high availability in general. The second lists the most important concepts and issues related to failover clusters. The third part is devoted to Pacemaker (also known as Heartbeat v2), the open source cluster resource manager. Most solutions in this area were based on costly proprietary hardware and software. In recent years in particular, open source products did make good progress and today open source high availability products are on par with commercial offers. When combined with generally available off-the-shelf computers, they make a very competitive product.

1 Introduction to high availability

We want our machines to function. If they do not, then we look for remedies. If it cannot be fixed, then we get angry. That is the procedure in general. The pencil you use runs out of ink—you reach for another one on your desk. Pencils are cheap and it is easy to have spares around. A car has four wheels, but there is also a spare wheel hidden somewhere in the trunk. If one of the tires goes flat, the driver stops, takes out the spare wheel and tools, and replaces the wheel. Not as easy as taking another pencil, but still bearable.

Let us try to analyze these two examples from everyday life. You may wonder what is there to analyze—the actions we take are simple and straightforward. Still, in both cases the same process is involved. First, we detect faults. The pencil produces no marks on the paper anymore. The car starts swerving, so we stop to check and see that one tire is flat. Then we proceed to recover from the fault. We may first try to shake the pencil a bit, to see if there was perhaps some air stopping the ink. If that has no effects within reasonable time (until we get impatient), then we take another pencil. Replacing the tire itself is a complicated process, not for the layman, which is why cars are equipped with a whole extra wheel.

When it comes to computing, things get complicated, as always. Why is that? Well, both fault detection and fault recovery are actually complex processes. There are some fault-tolerant systems which are capable of sustaining hardware faults, but they are beyond reach for most budgets. And when it comes to software, all bets are off. Once software goes nuts, it never recovers. You must have seen that happen. At least once? Anybody? Some faults seem to happen in software, but actually they originate in hardware. Some are intermittent and happen only under certain conditions, say when a component is overheated. The bottom line is that it is difficult to detect many kinds of faults. And we have not even started thinking about how to recover from them.

The fault which the computer could not contain causes a failure. It usually manifests itself by utter incapability of a system to carry out any kind of useful operation. Sometimes also called crash or hang. You might have experienced that too.

Now let us imagine that the computer could pinpoint a fault. How to recover from it? Thinking about it, most faults can be addressed only by a complete system restart hoping that the fault is indeterministic. In fact, it turns out the most of recoverable faults are such. Sometimes, a reboot does not help either, in case an essential component went south (or anywhere else for that matter). Software bugs, and there are always some, are extremely difficult to contain. Just ask NASA.

As we apparently painted ourselves into a corner, let us try to recover from that. Since a single computer system cannot easily recover we can put another one to keep it company. Then we duplicate the functionality and if the first computers fails, just start the second one. So far, so good. We can now make the two machines "watch" each other and then take appropriate action if the other system seems to experience problems. Of course, things are not as simple as they appear, but this is a good starting point.

The concept of making two or more computers cooperate and jointly provide a service is called a cluster. Failover clusters are essential to most high availability solutions and one of the main topics of this paper.

Availability can be measured using the following formula:

$$\text{Availability} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

where MTBF stands for Mean Time Between Failures and MTTR for Mean Time To Recover. Higher values are obviously achieved by increasing the MTBF and decreasing the MTTR.

After choosing the hardware and the software, the MTBF is not within our power (cf. Murphy for more details). We will make it our business to reduce the recovery time.

1.1 The number of nines

The availability, as calculated in the formula above and expressed as percentage, contains certain number of leading nines (if it does not, we want to see your solution). This number of nines is how high availability systems are classified. For instance, one may hear that some system has *four nines*, i.e. availability of 99.99% or at most about one hour of downtime per year. Catchy, but not very useful. It simply does not tell much. Is that downtime coming from planned or unplanned outages? Does it happen all at once or in several occurrences and how many? Why does it happen at all? How is availability actually affected?

1.2 Single point of failure (SPOF)

A SPOF is a component of the system which has no spare to take its place should it malfunction. That brings us to one definition of a high availability system (Pfister 1998):

1. No replaceable piece is no single point of failure.
2. The system is sufficiently reliable that you are overwhelmingly likely to be able to repair or replace a broken part before something else breaks.

For instance, if you need a pencil for an exam, you will be really glad to find an extra one in your bag in case the one you are writing with runs out of ink. That is making at least that part of the problem redundant (though it would be nice to get there brain-wise as well, but that part is not yet replaceable). It may help looking around and into your daily life, you will be surprised to find quite a few things with redundancy built-in (explicitely or implicitly) and then some without.

Identifying single points of failure is one of the crucial tasks while devising a high availability solution. With some one must learn to live: it is simply too expensive to provide redundancy for all. That is unless the service you are protecting costs even more, but in that case you would probably want to look into systems which are designed for fault tolerance.

1.3 Availability requirements

This is where the "how high" is defined. Though in no precise terms, but one should at least know to which ball-park a solution is supposed to belong. There are several classifications to choose from (one must love the variety out there). This one is adapted from an oldish book about one venerable cluster system (Digital 1994):

Conventional computing

No particular availability requirements. The system may go down at any time and stay out of service for longer periods. These systems are not considered to be highly available for any definition of the term and are not of much interest here.

Highly available computing (during specific time periods)

These systems are very important, but only during certain times, typically business hours. For example, an accounting application server for a solicitor's office.

Highly available computing (365x24x7)

These systems are very important and should be available at all times, with just a few short outage periods for the maintenance or perhaps failover times. Some of these outages may be offset by good planning and good maintenance practice, but may not in general be completely prevented.

Mission critical

Highest possible availability. With devices used in medical care or aviation for instance. Typically fulfilled by fault-tolerant systems. These systems are few and cost dearly both to develop and to own. If you are looking into this kind of computing then you will not find anything of use in this paper.

Disaster tolerant

Some of the above mentioned systems may as well need to be disaster tolerant. This topic, though of great importance, will only be mentioned in this paper.

We are interested in the second and third class or the highly available computing. Those could also be defined as high availability with lower end computers—we expressly go for *normal* species. Mind, that does not mean that one should be cheap.

1.4 What (not) to expect?

It is unreasonable to look for 100% availability. Things do go wrong in a myriad of ways and then commonly with preference for nasty ones. Your shiny brand new cluster setup might inspire a tad too much confidence—try not to succumb to that kind of temptation. The technology we deploy to protect important services is not perfect in a sense that disruptions are not noticeable. Sometimes, depending on the actual application design, the disturbance may even be rather bad. And sometimes the failover, which is what we ultimately depend on, may fail as well. Apparently that happens more often than one would expect.

2 Planning and implementation

One legendary book about software engineering says to plan to throw one away, because you will anyhow (Brooks 1995). This is not about software and we would not go that far anyway, but there is a good point: It is hard to get things right the first time. And it is also easy to get them very wrong. In particular if the problem is not well understood or if you rush to provide a solution based on your favourite choice of products. The solution must fit the problem you are trying to solve as close as possible. Afterwards, you should continuously analyze and evaluate the design, deployment, and testing results. Finally, complexity is an enemy of availability. Try to keep it simple. Do not add that nifty peace of software just for the sake of it, unless really necessary. Remember that there are eventually going to be others handling the system, perhaps not as diligent or proficient as you.

For the most part, this section is relevant to just about any project. Even to those which have nothing to do with computing. It is also a huge subject about which serious books are written, far beyond the scope of this paper. Still, it is good to point out those issues inherent to high availability solutions. What follows is not going to encompass every aspect of good project management and planning, but stress the issues which are of most importance.

2.1 People

No amount of magic may replace good people on the project. This is the foundation of every successful enterprise. Unfortunately, companies and management often give only lip service instead of really taking proper care of their human resources. If your employees and colleagues are discontent, the project will just as well hobble instead of making decent progress.

Knowledge is an asset which cannot be replaced by technology, no matter how hard one tries (though some do not even try very hard). Obviously, not everybody can know everything, but sharing knowledge is still often neglected. You should try to organize workshops and presentations. Reward speakers and demonstrators; they do deserve it for teaching and presenting knowledge is rather demanding (trust me, I know).

When it comes to high availability, one should also stress the level of experience. Some high availability issues may be understood only by extensive use, testing, and experimentation. If people have never before worked in this area, you may be in for some unpleasant surprises.

2.2 Customer requirements

Reading and interpreting requirements is an art. If you fail here, it is sure to hunt you until the end of the project (which may be a rather long time). The better the requirements are understood, the better fit the future solution is going to be. And you do not need a misfit here.

More often than not, the customer is either not sure what they want or not able to express it precisely. It is your task to educate them and bring the requirements out in the clear. Furthermore, high availability may appear to hold too much promise. Again, it is up to you to right your customer's (high) expectations. High availability is no magic which may escape the gravity law. Obviously, these are all tough propositions and you probably will not enjoy this part of the engagement, but persevering will bring abundant reward. Even when, in the end, you are not really aware of it.

The users of the system are sometimes represented in the customer delegations, sometimes they are not. That probably depends on the company's organization. Even though they may not have a chance to provide input for the requirements, you should make sure that their voice is heard. And you should listen to it. Do not forget that it is the users who are going to judge the solution in practice in the first place.

2.3 Hardware

Hardware should be robust and reliable. It should also be proven in practice and simple in design. Redundancy on the hardware level is important. After all, it comes with much lower price tag than software. That is what we ultimately rely on.

Choosing hardware is not easy (as if there is anything easy), but should be easier than the rest of our project. It is important that the hardware fits applications it is to run. Computing requirements are diverse: just compare a relational database and a tomcat/java application.

Some hardware components are more prone to failures. Usually those with moving parts: disks and fans. Plan to make those redundant. Hardware and software vendors come a long way in this respect: there are redundant power supplies and RAIDs. Things which come in multitude, and exactly for that reason, also fail often. Make sure that those parts either have built-in redundancy (e.g. memory with ECC) or that the system can survive their failures. An eight-way server is not going to be a net contributor to availability if it cannot survive a single CPU failure.

2.4 Software

Software, at least in part, usually comes with the customer requirements and is not subject to change. That still does not mean that one should not analyze it and see how it fits in the high availability landscape. It may also be the case that a particular application was not designed with high availability in mind or does not play well with failover clusters, which is usually the main technology to protect services. In such cases it is necessary to discuss the matter with the customer and the application vendor. Note that the customer is usually not willing to look into details in case of an unplanned outage—they payed for a high availability solution and expect to have one. Whether the problem is in application which has not much to do with you may at times be difficult to prove or explain.

Finally, there is also clustering software which is at the core of the highly available solution. There are quite a few to choose from. Not all are well suited to all needs, though most are fairly general and even tend to converge over time.

2.5 Environment

Physical environment is always part of the high availability solution and not something to be neglected. If the air-conditioning system failures are not accounted for, you may end up having all computers shut themselves down which certainly will save hardware from melting, but will not improve the availability.

Physical security deserves special attention, not only because one should guard assets from malicious people, but also to prevent accidents such as pulling wrong cables (now, that is already legendary). Environment risks must also be considered, for instance checking roof for leaks.

2.6 Finding single points of failure

Once the solution is laid out, one should search for all the ways it can fail. This activity cannot be emphasized enough. The more ways you find to disrupt the service, the better you will be able to protect it. Brain storming sessions may help. Getting into a particularly mean mood also.

2.7 Testing, one, two, testing

It is not certain how much testing is enough testing, but you will do well to err on the plus side. If the systems survive the tests then something is seriously wrong with them. With the tests that is. The primary purpose of testing is to try to break things. Then see how and if the computers recover from it. Then file bugs. Seriously though, there is no way testing (or your computers) can be stressed enough. The more different ways you find to break them the better. OK, you should refrain from excessive violence, using axes, or spilling good coffee over a good disk.

3 Failures and outages

Since it is our main business to recover from failures and prevent or lessen outages we should get ourselves acquainted with them. Not all faults result in a failure. Some faults must be contained by the system, the hardware in particular. Typical examples are power supply and disk faults, and some limited network outages.

Failures may be broadly classified in two categories: failures from which it is possible to recover automatically and those which need manual intervention. The latter, resulting in longer outages, require disaster recovery. That usually involves some rather involved and, hopefully, well documented processes such as restoring data from backup or moving applications to a backup site. It is important to know which failures have such grave consequences. Firstly, all single points of failures belong to this category. Shared storage is one typical example. If it fails due to a hardware problem or firmware bug, quick recovery is not possible. Secondly, shared data/filesystem corruption have disastrous effects. Thirdly, logical errors in shared data structures or serious defects in software applications have big impact also. If there is a possibility for an application to leave data in an inconsistent state, that could be a serious problem. Cluster software failures can also cause a big outage. Further, if you are unfortunate enough to have two or more failures happen in quick succession, that is also usually not a candidate for automated recovery. “Quick” is faster than you can replace or repair the component which failed first. The morale of the story is that it is essential to identify SPOFs in your high availability solution.

Failures from which the clustering software can recover automatically also result in an outage, though the outage is short and should not incur a big disruption of service. That, however, depends to a large extent on the application which provides the service. One of the main issues is how the application deals with the session data. If the application must be migrated, then obviously all sessions are going to be interrupted and, usually, lost. That typically means that all users have to login again. In case there were many and the failover happens during prime time (which is quite likely), then the server may be faced with the effect of a thundering herd, i.e. all users trying to login at the same time. In some cases, clients (as in software) may help by keeping the session data locally. The most common case is web browsers and cookies.

This is an excerpt, representative though, from the collection of possible problems. They will also depend on particular circumstances in the project. You have to identify and account for all possibilities in this respect and document them in the service level agreement along with the projected outage time and the probability of occurrence over a certain period of time.

Now, likeliness of some event is often difficult to estimate. Apart from future being consistently elusive, such data from hardware vendors is hard to obtain. Do not even think about asking software people how often their application is expected to fail within a year. And it is obviously impossible to predict when a system administrator is going to be tired or distracted enough to bring down a wrong instance or type that fatal `rm` command. The question is not really *if* there will be a failure, only *when.* The best one can do is to rely on experience, yours and that of other people. Though the latter should of course be taken with a grain of salt.

4 Failover clusters

A failover cluster is the essential part of most high availability solutions. That is the thing which makes automatic recovery possible. It is also the technology mostly discussed in this paper.

The idea behind failover clusters is simple: application from a failing node migrates (fails over) to a healthy node. Simple indeed. Unfortunately, not so simple when it comes to implementation or deployment. One of the tricky parts is making sure that there are no false negatives, i.e. to establish with certainty that that node has really failed. If two nodes get to try to control a resource which has not been designed for such usage, and most of them are not, the service will be gone at best and at worst your data will be shredded.

Let us review the most important cluster concepts.

Nodes

A node is a computer. It is just another name, but you are almost sure to hear cluster people talking exclusively about nodes and never saying server or host or even computer. There is nothing really special about nodes. They do run some special software and, being members of a cluster, have some special communication requirements.

Membership

Nodes are members of a cluster. It is one important function of the cluster to ensure that some computer may be a member of the cluster and to accept all eligible nodes as members.

Quorum

Due to communication problems a cluster may be partitioned. If that happens, it is essential that exactly one partition (set) of nodes forms a cluster which may run resources. That partition is said to have quorum. Quorum is a measure which makes a set of nodes entitled to form a cluster which can provide services. Typically, nodes vote (one node, one vote) and the partition having more than half of total possible number of votes has quorum.

Resources

A resource is a piece of software which provides some kind of service, either directly to users or to another resource. For a cluster, it is crucial that the service can be started, stopped, and checked (monitored). If one of these things cannot be done, then the service cannot be a resource. Typical examples of resources are a database, a filesystem, or an IP address.

Fencing

Fencing is an exercise in preventing a node to own some or all resources. It is necessary in case software on the node misbehaves in certain manner or the node is not accessible (which often boils down to the same thing). Fencing comes in two kinds: resource level and node level. The former is just about "building a fence" around some resource for the target node. The latter is about stopping the target node completely, usually by pulling the power plug. In one clustering product the process thread doing fencing is called *grim reaper* (Fafrak et al. 2003). You get the idea.

Split brain

A vivid term for a partitioned cluster. The most dreaded event. Best to be avoided. If at all possible. When it happens, somebody usually gets killed (I mean nodes, not people, though one never knows). But it is really, really for the best to do all reasonably possible to prevent it from occurring. And that is not even that difficult, just provide two or more communication paths and make sure that they are reliable.

4.1 How failover clusters work?

As we have implied before, a cluster consists of a set of nodes. We also mentioned that those nodes are just computers though with certain quirks regarding communication. Now, communication is extremely important for clusters. Nodes which cannot talk to each other cannot make a cluster and present a denial of its own existence as cluster nodes. Lame computers are of no use. Hence, please do make sure that the nodes can hear each other loud and clear.

Now that the communication is out of our way, or, better, within our way, let us see what nodes talk about. First and foremost, they have to exchange credentials. A cluster is an elitist club and no node is allowed to join the party unless they show an invitation (and are dressed according to the code). This is what one also refers to as authentication. Only authenticated nodes may be members of a cluster. How do they authenticate? Usually with a shared secret key. This key is replicated by the system administrator to all nodes.

After saying "How do you do", all of them, the nodes do a complex dance which allows them to form a society or, if you will, a cluster. How exactly this is done, you do not really want to know (me neither). Note that it is not the same thing as a bunch of people meeting in a room. You may try to simulate it for fun, just in order to appreciate the complexity. This is what we call cluster membership.

Membership established, the members count themselves and if the count is sufficient, they proclaim themselves a quorate cluster. This is important, because nothing is going to happen unless there is a cluster with quorum. Having quorum, the nodes proceed to elect the master of the parade. It does not really matter which one it is going to be, but it does matter that there is only one. Have two—and the cluster is in for a mess. Three is right out. One is the number. This node, the one which is more equal than the others, will be making all decisions in the future. All the others shall obey.

Now the cluster will start the applications and we may all go home, right? Not so soon. Before dealing with resources in any way, the master node checks his membership files in order to see if any members are missing. If there are, he produces a big hammer and hits each of them hard on the head. Just in case. Joke aside, as we did not hear from those missing nodes, heaven knows what they may be up to. Better safe than sorry. That is what we call fencing. This time, more precisely, the startup fencing. If you wonder how come that the nodes are missing but one can still hit them, the answer is that fencing simply must work when we need it. It is up to you to make sure that the big hammer is present and regularly oiled.

Right, you're being a bore and let's start the applications now. Well, still not there, but almost. The first principle of clusters is to do no harm. That is absolutely the worst case in terms of availability. A cluster always makes sure that there is always exactly one copy of a resource running. Imagine what would happen if a filesystem, designed to run on a single computer, gets mounted on two or more nodes. Pfister vividly compares that to two people at the same time holding the steering wheel of a car on a busy road (Pfister 1998). So, the chief node first orders all nodes to check if there is any of resources running out there and collects all results. Normally, all resources should be stopped (down), because the gentleman agreement is that you shall have no other masters but me (i.e. the cluster). If the cluster finds any of them running it is certain to complain loudly about it, traces of which you will find in the logs.

OK, we can start the applications now. Resource interdependencies are read, user preferences observed, and all resources started in order which does not violate any given or implied rules. That means that some resources are put together on some node and furthermore that they are started in certain order. Some, again, may be running on different nodes, but still comply with some startup order. And some, those which do not depend on each other, may be started in parallel. All these rules are applied by the cluster, but defined by a human. Let us just hope that they understood each other.

At this point in time, the cluster reaches a stable state. But it does not yield to lethargy: the nodes keep exchanging health information with each other and the hardware and resources get monitored. To the user, however, the cluster seems dormant and that is the way it should be. All the activity serves only to catch faults. Before one attempts recovery, it is a good idea to identify a fault first.

Faults, well, there are many to choose from. Node level faults are rectified by grotesque yet efficient means of fencing. Of course, all resources running there are migrated (moved) elsewhere. On resource level faults, the resource is either restarted or failed over to another node. That depends on user preferences or perhaps the node's record (history) running the resource. Resources which depend on the one to be moved follow suite.

Some resource level faults are considered to be unrecoverable and also result in fencing. It may sound harsh, but if the resource fails to stop it is impossible to establish its state anymore and the only way to rectify that is to reboot the node.

If you managed to get through the narrative, it is fair to note that this description may not match all failover clusters, but the operation should not vary too much. Allowed quantity of white lies included. Do not try this at home.

5 Two-node clusters

Though we do not have hard data, these are probably the most common clusters in use today. On the one hand, they represent the basic idea of redundant computing. On the other, it supports well vast majority of applications which typically run in *active-passive* configurations such as big relational databases.

For us, humans, the concept is only straightforward: one server handles the load while the other just hangs around, then if the first one goes under the second one kicks in and takes over its duties. Indeed what could be more simple. It is just that computers are not humans and have different perception of reality. The reason in this case is simple: node A may not know what is happening with the node B. While the two nodes can communicate everything is fine, but in the case of split brain no partition is eligible to run resources. Because there can be only two partitions and each have just half of votes: not enough for the quorum. Note that split brain is not the only possibility: the other node may simply be down.

Anyway, since two-node things are out there in great numbers, we have to deal with them somehow. One possibility is to use a tiebreaker. That could be a quorum disk or external quorum server. The tiebreaker normally sits idle, but in case of membership changes both partitions will try to contact it in order to gain enough votes. It is then up to the tiebreaker to yield to just one side thus resolving the dilemma. You can imagine the kind of calamity which would ensue if it were to yield to both sides.

Another solution is to simply ignore quorum which is of no use with the total of two votes anyway, and use fencing in its stead. In case of split brain or a node outage, one party is just going to fence the other instead of seeking quorum. Now, there may be a death-match, both sides trying to pull the trigger at about the same time. Still, only one is going to survive, because the probability that both execute the `stonith` command at exactly the same time is very small. You should make sure that your fencing device really resets a node immediately and does not give it chance to do a reset itself.

5.1 How many nodes?

If two nodes are no good, you may wonder how many nodes are. The answer is, as so often, that it depends. First, it depends on your applications. If they require big computers, then it will not make financial sense to have more than two of them. Also, if an application cannot run on more than one node in parallel and there is only one application, again two nodes is the only reasonable option. Note also that maintenance of extra nodes does not come for free.

Still, if you can go with more than two nodes, it is better to have odd number of them. Clusters with say four and five nodes respectively have exactly the same quorum value (three), but the latter cluster can survive if two nodes go bust whereas the four-node cluster cannot. The reason: the remaining two nodes would have only two votes and that is not enough for the quorum.

6 Cluster maintenance

Good maintenance practice is always important. Even more so for clusters. First, they are part of the high availability solution and it has been shown that sloppiness reduces availability. Second, though clusters present an illusion of a single-system image to the users, the system administrators still have to face multiple computers. It really helps if all nodes of a cluster are kept in sync in terms of software releases and configuration. Unfortunately, experience shows that computers tend to drift apart in both respects. In worst case, these differences may prevent an important application from running in case of failover.

It is beneficial to use a unified management solution for clusters. I understand that there are not many out there and that they are expensive and often difficult to use. Well, simple tools such as `clusterssh`, `dsh`, `pdsh` or similar may go a long way if used with care. If used without care they may bring all of your cluster down. Similarly, `csync2` or `cfengine` may keep your configuration files synchronized. Again, if you screw one setup, your tool will happily do the same to the other nodes. Hopefully you get the point: `root` access is dangerous, simultaneous `root` access more, and when it is to a high availability cluster still more.

Software upgrades also present a challenge. Whereas it is obviously advantageous to run identical computers, monoculture can be a serious impediment to the vitality of the whole system. A single software bug, if placed strategically, say in a network device driver, and then replicated, may result in complete service denial. For example, it may be a good strategy to install network cards which use different drivers. The conclusion: test all changes before infesting the whole cluster with a condition which may take time to cure.

System and application logs are important for reporting or troubleshooting. Clusters, consisting of more than one computer, benefit from centralized logging for at least two reasons: a) the logs may be examined even if a cluster node ceased to function and b) the logs are combined and better represent cluster actions. It is also important to have clocks in sync within the cluster, otherwise you may go crazy trying to relate events coming from different nodes. `syslog` is the ubiquitous unix logging application which supports so-called log hosts: a host which accepts and logs messages coming from various hosts on the TCP/IP network. Standard `syslogd` daemon is antiquated and does not handle particularly well large number of log messages sent within a small period. It is better to use a newer syslog implementation such as `syslog-ng`.

6.1 Cluster aided management

Now that everybody is properly frightened ("We scare because we care!", Monsters Inc.), let us examine the advantages clusters bring to system administrators.

For obvious reasons of duplication, the system management becomes much easier. If you need to update one system which is running the application, just migrate it beforehand to another node, do the update, and migrate the application back.

Pacemaker in particular has some interesting features in this area. Apart from the usual node standby, where a node is not allowed to run any resources, it is also possible to place a resource in the *unmanaged* mode. Application upgrades may be done then without cluster getting into our way. There is also an option to put the whole cluster into maintenance mode. Note that for the duration of maintenance mode resources are not protected by failover mechanism.

7 Applications and failover clusters

As we already have said, applications are termed resources when managed by a cluster. That already hints that cluster software involvement makes a difference in application handling.

Applications are usually within the realm of other specialists. However, it is important to know how they behave, in particular under stress (high load) and on failovers. Though the application itself may not be susceptible to crashes, these may happen due to many other, unrelated faults. The application data must remain in a consistent state. Certain unusual faults may create circumstances which the application designers did not foresee. Even though we may not be in situation to fix them, such weak points must be brought out and discussed with the customer and application vendor.

Cluster sees applications through a special device which makes them all look exactly the same. That may sound scary, but this black-box feature ultimately makes failover clusters so versatile. The device is called a resource agent (RA). It is typically just a shell script which may start, stop, and test an application. Something like an LSB (Linux Standard Base) init script. In fact, it is possible to deploy an init script as a resource agent. Use of LSB init scripts should, however, be discouraged in favour of resource agents which are distributed with Heartbeat and Pacemaker. The latter are implemented with more care and designed to work with failover clusters. The former are often lax if not outright sloppy, simply because it is expected that system administrator is going to nurse them in case of problems. With clusters, resource agents must wrestle autonomously with all possible application vagaries.

7.1 Timeouts and false positives

One of the tricky areas of cluster configuration is to make a cutoff line between application working and not working. Obvious faults are easy to detect, but processing of any kind should be finished within a decent time-frame. If it does not then it may be the same as service failure or even worse than that. The trouble is how to define decent. Even more trouble is to know whether it is realistic to do any better under given circumstances. All this must sound like a lot of hot air, but it is in fact a matter to be seriously considered. For clusters, these considerations translate to defining timeouts for operations on resources, in particular for monitors. The point is, if the software and hardware are doing their best, but in spite of that the monitor operation times out, there will be a failover triggered.

That is going to make things only worse and may lead to a serious service degradation. Note how in this case it is actually the cluster, more precisely the resource agent, which aggravates the problem and eventually reduces availability.

One cannot give any definite advice on how to set timeouts as that depends on many factors which are specific to the application and environment, among other things. Just note that false positives, or failure detection which should not have happened, are a really bad thing. On the whole, try not to make timeouts too tight. Thorough testing may also help find out how application behaves and to get a general feeling of what should be considered a failure.

Start and stop operations deserve similar considerations. This time it should be easier to get a good estimate simply because starts and stops are mostly happening at calm sea and not in troubled waters. However, a high-load induced failover may make conditions completely different and if these operations do not behave in such circumstances that may cause an outage.

Resource agents and applications they control are the foundation of a cluster. Dealing with them may be boring, but is certainly worth every effort as it seems like significant number of cluster issues stem from this area.

8 The Pacemaker/CRM cluster

The pacemaker (formerly known as Heartbeat/v2) is at the core of the open source failover cluster product. Another name is CRM which stands for Cluster Resource Manager and that is exactly what it does: managing resources cluster-wide.

8.1 Components

Figure 1 shows the major subsystems of our cluster stack.

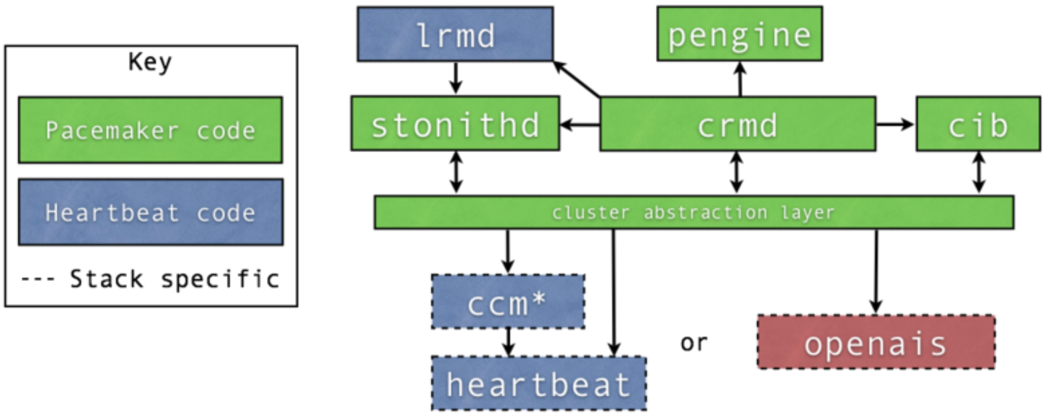
`crmd`

: `crmd` is in the center of the cluster. It talks to and coordinates all other subsystems. It also receives events from other components, such as membership changes or resource failures and initiates actions accordingly.

Membership and communication layer

: Until recently, CRM was used exclusively with the Heartbeat communication layer. Today it also supports OpenAIS. The Heartbeat communication layer is supplanted by CCM (Consensus Cluster Membership) which provides membership information. OpenAIS has the membership combined with the communication layer. Heartbeat has quorum plugins to provide quorum information. With OpenAIS the CRM provides the quorum service itself.

Figure 1: Subsystems



Pengine

Policy engine is the "thinking part" of the Pacemaker. It makes decisions about resource placement and action ordering. Actions are resource management actions and fencing operations.

CIB (Cluster Information Base)

CIB is a distributed application which manages the cluster configuration. The configuration is in the XML format and kept synchronized on all cluster members.

LRM (Local Resource Manager)

The LRM manages resources on a single node by invoking resource agents (RA). Every supported application has a resource agent which knows how it is to be started, stopped, and monitored.

Stonith

Stonith is the fencing subsystem. Its sole role is to carry out fencing requests sent by the CRM.

Management tools

Two high level management tools are available: the GUI (`hb_gui`) and the command line management and configuration utility (`crm`). Both are capable of configuring and managing the cluster and resources.

8.2 Configuration elements

The CIB is represented as a file which is distributed to all cluster members. It is in the XML format and consists of several sections which may each contain one or more elements. The meat of the configuration are resources and constraints. Resources represent applications and constraints user preferences and application relations (or dependencies). While the cluster is running, the CIB may not be edited directly using an editor, but only via the management tools.

Primitive

This is the basic building block and represents a resource. A resource is described by defining attributes such as configuration file or logical volume name.

Group

A group is a set of primitives. It is a useful shortcut to represent a group of resources which must run together and be started in certain order.

Clone

It is possible to run several copies of a resource using clones. Usually one per node, though it does not have to be that way. One example is a parallel (cluster) filesystem mount.

Master (multi state)

Some applications have three rather than two states. The two running states are usually called master/slave or primary/secondary. Using Master it is possible to represent such resources.

Location constraint

This constraint serves to enumerate nodes allowed or preferred to run the given resource. Depending on the given expression, a score is calculated and assigned to referenced nodes. The node with the highest score gets to run the resource.

Collocation constraint

Collocation expresses a spatial relation between resources. Resources may be required to run on the same node or on different nodes. It may also sometimes be used to prevent a resource from running at all (for example, a development database instance).

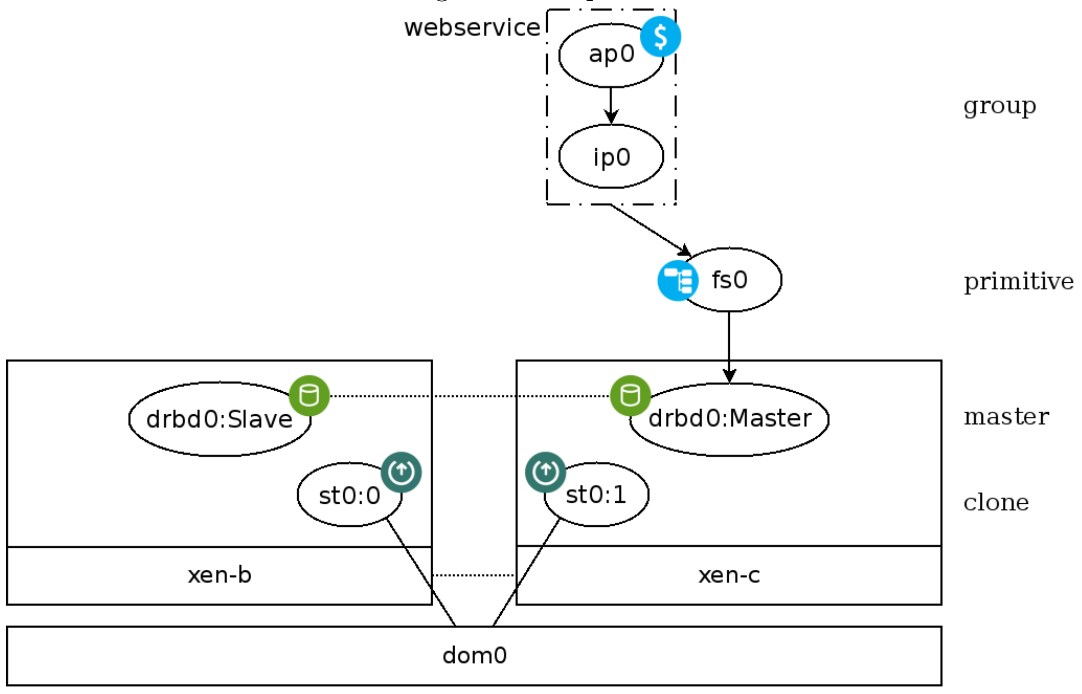
Order constraint

Order expresses a temporal relation between actions for resources. Dependant resources must be started and stopped in certain order.

9 Pacemaker/CRM in practice

For demonstration purpose we will show here a small cluster configuration consisting of the drbd shared storage and the Apache web server (as shown in Figure 2). The cluster consists of two Xen virtual machines: `xen-b` and `xen-c`. The Xen, drbd, and Apache

Figure 2: Sample cluster



specific configurations are not going to be shown. The usual setup should suffice. We also skip OpenAIS or Heartbeat configuration details.

All examples are displayed in the `crm` shell notation and should be entered at the `configure` level. Note that the changes do not take effect until you issue the `commit` command.

9.1 Cluster properties

OpenAIS does not provide quorum service, so `crmd` should be informed about the number of nodes:

```
property expected-quorum-votes="2"
```

Note that in this case it is not going to matter, but for bigger clusters you should set this property. Since this is a two-node cluster there is no quorum, so we set the corresponding cluster property:

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

9.2 Fencing (`stonith`)

We are going to need fencing, so it is just as well to immediately deal with it. More, it is easier to test fencing while the cluster configuration is empty.

The `external/xen0` stonith plugin utilizes the dom0 `xm destroy` command with `ssh` and authorized keys. Stonith is configured like any other resource using the `primitive` element:

```
primitive st0 stonith:external/xen0 \
    params dom0="xen-host" hostlist="xen-b xen-c"
```

The stonith resource serves as fencing agent, so we have to run it on both nodes. Since `external/xen0` stonith plugin accepts multiple nodes in the hostlist, we can simply clone the resource:

```
clone c-fence st0
```

Cloning makes one instance of `st0` resource running on each of nodes. "Running" should not be taken literally: the fencing agent sits still until somebody makes a fencing request. We should also occasionally check if it works:

```
monitor st0 2h:30s
```

Every two hours the `st0` resource is going to run the monitor operation. That sets the fencing for our virtual cluster.

To test it, just login to one of the nodes and remove the main communication process without giving it a chance to sign off (`heartbeat` for Heartbeat, `aisexec` for OpenAIS):

```
# killall -9 heartbeat
# killall -9 aisexec
```

The other node is then going to reset the node which cannot talk anymore.

9.3 Shared storage (drbd)

It is time to define the drbd based shared storage:

```
primitive drbd0 ocf:heartbeat:drbd \
    params drbd_resource="r0"
```

That makes the basic resource, which represents the `r0` disk defined in the drbd configuration file (usually `/etc/drbd.conf`). The drbd disk is a master-slave application:

```
ms ms-drbd0 drbd0 \
    meta notify="true"
```

Just like `clone`, `ms` (or `master`) creates more than one instance of the resource in the cluster. `ms` also *promotes* one of the instances of the `drbd0` resource to the master state. The `notify` meta attribute is an instruction to CRM to send notifications to the `drbd` resource agent before and after operations which are going to change state. Needless to say, the `drbd` resource agent is rather complicated.

It is good time to see how our cluster looks like after these changes:

```
# crm resource status
Master/Slave Set: ms-drbd0
        Masters: [ xen-c ]
        Slaves: [ xen-b ]
Clone Set: c-fence
        Started: [ xen-b xen-c ]
```

Now that the shared storage is running, we can put some files on it. After making the `ext3` filesystem and mount points on all nodes, let us create a resource which will run the filesystem:

```
primitive fs0 ocf:heartbeat:Filesystem \
    params device="/dev/drbd0" directory="/mnt/drbd" fstype="ext3"
```

The filesystem however can be started only after `drbd0`:

```
order drbd-before-fs0 mandatory: ms-drbd0:promote fs0:start
```

This order constraint tells CRM that the start operation of `fs0` can only be executed after promoting the drbd resource. The order is symmetrical: stop and demote operations are run in the reverse order.

We also must make sure that both the filesystem and the master drbd instance run on the same node:

```
collocation fs0-with-drbd inf: fs0:Started ms-drbd0:Master
```

With this constraint we make sure that `fs0` in the `Started` state runs only together with `ms-drbd0` in the `Master` state. The order of the resource-states pair in a `collocation` is significant. The resource on the left side follows the resource on the right side. Or: the left resource is collocated *with* the right one.

9.4 The web server (Apache)

We have the data now, which can failover between the nodes, but it is not of much use unless we run some application to present it. In this case, we will offer a web service using the Apache web server:

```
primitive ap0 ocf:heartbeat:apache \
    params configfile="/mnt/drbd/conf/httpd.conf"
primitive ip0 ocf:heartbeat:IPaddr \
    params ip="10.2.13.100" cidr_netmask="24"
group webservice ip0 ap0
```

The `group` shortcut makes sure that the resources run always on the same node and that they are started and stopped in certain order.

Obviously, the Apache web server has to start after the filesystem and to run on the same node. Another two constraints take care of that:

```
colocation web-with-fs0 inf: webservice fs0
order fs0-before-web mandatory: fs0 webservice
```

This time we can leave out states and actions. They default to `Started` and `start`.

Finally, we should also monitor the web service:

```
monitor ap0 30s:30s
```

If the Apache server fails it is going to be restarted. If you want to have it failover to another node after certain number of attempts, set the `migration-threshold` meta attribute:

```
primitive ap0 ocf:heartbeat:apache \
    params configfile="/mnt/drbd/conf/httpd.conf" \
    meta migration-threshold="2" \
    op monitor interval="30s" timeout="30s"
```

Note that the monitor `crm` command actually results in the monitor operation which is part of the resource definition.

This example configuration shows most of the CRM features. Virtual machines are cheap and easy to get: it is good to experiment as much as possible with such virtual clusters.

10 Bibliography

Brooks, Frederick P. *The Mythical Man-Month: Essays on Software Engineering*, Second edition, Addison-Wesley, 1995

Digital Equipment Corporation. *Building Dependable Systems: The OpenVMS Approach*, Part number: AA-PV5YB-TE (pdf), 1994

Fafrak, Scott, Jim Lola, Dennis O'Brien, Greg Yates, and Brad Nichols. *TruCluster Server Handbook*, Digital Press, 2003

Marcus, Evan and Hal Stern. *Blueprints for High Availability*, Second edition, Wiley, 2003

Pfister, Gregory F. *In Search Of Clusters: The ongoing battle in lowly parallel computing*, Second edition, Prentice Hall, 1998

Schmidt, Klaus. *High Availability and Disaster Recovery: Concepts, Design, Implementation*, Springer, 2006