Highly Available Service Environments

Introduction

This paper gives a very brief overview of the common issues that occur at the network, hardware, and application layers, as well as possible solutions, that may be faced by an organization required to address availability and performance issues of a service. It will provide this information in a clear and concise manner without the use of marketing buzzwords. Reading this paper will not make a person an expert on high availability systems nor allow them to go out and start creating one. It instead will get them to actively think about the issues that are involved not only with their services but also with high availability systems and provide a starting point for when they are ready to go get more information.

Network

The network is often the last piece of infrastructure service designers look at when planning out a service. There is a misconception that as long as a service's connection to the network is not slow or close to max capacity that there is nothing else to address on the network side. Presented below is a description of common network configurations, the problems that can occur in these configurations, and methods for addressing these problems.

Standard Network Environment

The standard configuration of a network, pictured below, is to have one or more routers which make up the core of the network. These routers establish the internal addressing scheme (subnets) and connect the network to a service provider, like AT&T or Verizon. Switches then connect to these routers in order to provide connections to individual systems. Depending on the size of the network there may be multiple layers of routers and switches in order to distribute some of the work of handling the network traffic.

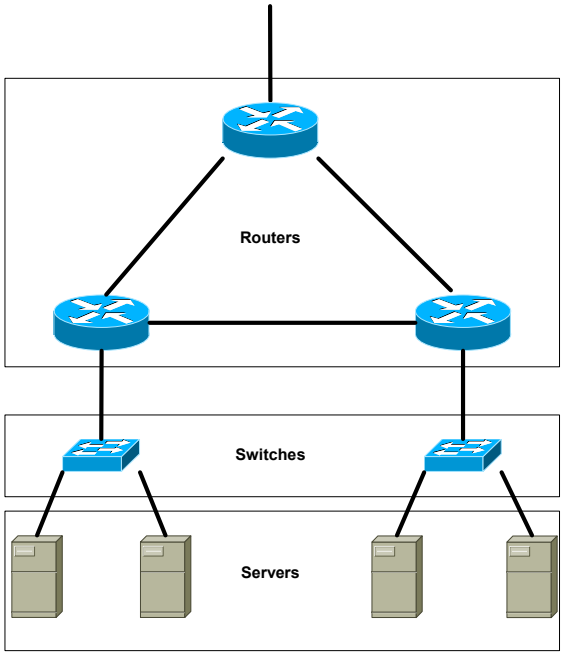

Figure 1: Standard Network Environment

While this network configuration is very useful in many circumstances and easy to maintain because of its simplicity it does have some problems if the goal is to create a highly available environment for services.

The first problem lies with the routers themselves. In most network configurations if a router fails anything connected to that router is isolated. Until recently the only way to address this problem was to purchase routers that included as much redundancy as possible. These routers contain things like redundant network interfaces, power supplies, and routing engines. While this definitely helps the problem there is still a chance that a router can fail and leave a segment of the network isolated, and routers with redundant hardware are often far more expensive than those without.

To address this issue a special network protocol called Virtual Routing Redundancy Protocol (VRRP) was created. VRRP allows one router to take over the operations of another router in the event of a failure. This allows a company to invest in hardware that does not have redundant features, spending less money, and still eliminate the router as a single point of failure in a network.

The next problem lies with the router to switch connection and is especially acute if the routers are using VRRP. While VRRP can allow one router to take over for a failed router it can not move the physical connection from the failed router to the operational one. Therefore the switches must now have connections to all of the routers that are set up in this redundant fashion. For example, if the two bottom routers in the diagram above employ VRRP then each switch pictured will need to be connected to both routers so that, in the event of a failure, the remaining router will still be able to contact the switch and servers connected to it. This is true, even if the routers are not using VRRP, but instead of whole router failures a single network interface failure will isolate a segment of the network.

Now that the router hardware and router/switch connections are not single points of failure only one major problem remains. Like the router/switch connections the switch/server connections could fail and isolate a server. This failure could be caused by a single switch interface failure, or by a complete switch failure. The very nature of switches allows computers to be hooked up to more than one, likely without any additional configuration.* Therefore, to protect against interface or complete switch failures, each server should be connected to two or more switches.

Highly Available Network Environment

Employing all the suggestions above creates a network, often termed a multi-homed network, without a single point of failure which is very desirable for hosting services that need to be available the vast majority of the time. The diagram below shows what a small highly available network might look like.

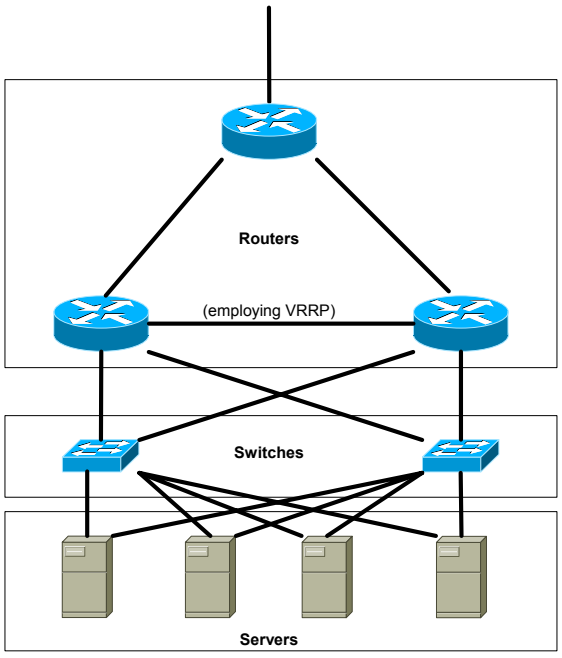

Figure 2: Highly Available Network Environment

In addition to being highly available this network may provide other benefits depending on the capability of the hardware used. Many servers and switches can be setup to take more than one network connection and treat them as a single virtual interface, an approach sometimes called binding. This allows them to make use of all links, when they are operational, to gain additional bandwidth which in turn could improve the response time of services hosted in this environment.

* Some switches which provide advanced features may require that some configuration be done to the switches in order to handle this type of setup.

Server Hardware

When it comes to server hardware there are two approaches often taken to ensure the availability of the server. One approach is to pack a single server with as much redundant hardware as possible. The other method is to purchase multiple, often commodity, machines and create some mechanism to have working machines take over for failed machines. These approaches are analogous to the approaches, discussed above, to ensure the availability of the network.

Redundant Hardware Method

Many companies that produce servers for mission critical applications, like IBM, HP, and Sun, sell hardware that contain at least two of every piece of hardware inside the system and an operating system that can reroute work away from hardware that fails, a breed of hardware that these companies often refer to as enterprise class systems. Each company has different mechanisms for doing this which succeed, in varying degrees, to providing fault-tolerant hardware.

Usually these enterprise class systems are far more expensive than a system, or set of systems, which provide the same amount of computing power but don't offer the internal redundant hardware. For the extra expense however a customer usually gets not only a greater mean time between complete system failure but also better support and quicker turn around time for hardware failures, and arguably better engineered hardware. Together these things can go a long way to making sure servers and the services running on them are available for the maximum amount of time.

Extra hardware costs aside there can be some very real problems with this approach. First, some enterprise class systems have a controlling piece of hardware for every set of redundant hardware. If this controlling hardware fails, the system goes down. For example a server has 4 CPUs (C1-C4) and C1 is the controlling hardware for all the CPUs. If C1 fails, all the CPUs become unusable and the server goes down. If C2, C3, and/or C4 fail things continue as they normally would have, except for perhaps performing operations a bit slower. This sort of architecture is definitely something to look out for and avoid.

Another problem with this approach is that no matter how it is approached there is still only one machine. If that machine needs to be taken down for maintenance then all the services running on that machine will be unavailable. This could result in a loss of business, a situation most companies would prefer to avoid.

The last problem is one of resources. Most enterprise class systems run a special operating system. These operating systems require skills to maintain that people in an organization may not possess. Also, because of the unique operating system, certain applications may not be available for that system.

Load Balanced Commodity Hardware

The second approach to ensuring the high availability of a server, as stated above, is to take a collection of commodity hardware and set them up in a manner that active hardware will take over for failed hardware. Such a setup is usually called a cluster with

each system in the cluster known as a node. The mechanism used to provide the fail-over capability is usually a process called load balancing.

To implement this approach a designer will need not only the two or more servers but also two or more network switches and load balancing network appliances such as the F5's BIG-IP or Foundry's ServerIron systems. Each server is then connected to each of the switches which are then connected to each load balancer. Each load balancer should then be connected to two switches on the external network, the same switches the servers where connected to in the highly available network environment described above.

In this setup an instance of a service is deployed on each server, known as a node, in the load balanced cluster of machines. The load balancers will treat all operational machines as one virtual machine, addressable by one or more IP addresses and host names. Incoming requests are then forwarded to one of the servers based on some policy, such as round robin, in the load balancers. The load balancers periodically scan the cluster to make sure all servers are up. If one or more servers are down no requests are sent to them until they come back online. Most load balancers will also detect, when forwarding a request, a server which is not operational but has not yet been marked as such. It will then mark the server as offline and forward the request to another server.

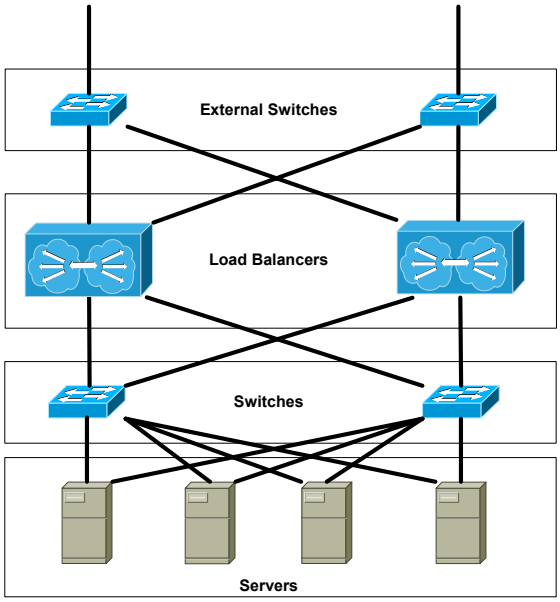

Figure 3: Load Balanced Server Environment

This approach to highly available servers has many benefits over the single system approach. First it is often much less costly, even taking into account the additional hardware (switches and load balancers) required. Second, because the servers are commodity hardware, a company will likely already have the resources to administer the machines. Third, because there is more then one server running a service at one time, servers can be taken down for maintenance without taking down the service. Lastly, this solution is often times easier to scale and usually offers better performance for the price.

This approach is, however, not without problems. First, commodity hardware often comes with commodity support. This could mean slower turn around times when a system does fail. Second, the mean time between failures for each system will be less

because each system lacks any sort of internal redundancy. Third, while adding more systems to the load balanced cluster will increase performance this is only true in certain cases and can quickly become an exercise in diminishing returns. Lastly, as will be discussed later, some applications are not designed to work in a load balanced environment which can lead to problematic behavior when they are placed one.

Highly Available Server Environment

The best approach to creating a highly available server environment is to use both of the methods discussed above. This will maximize the strengths of each approach and minimized most of their weaknesses. This approach creates a load balanced setup, as described above but, instead of using commodity hardware, low-end enterprise class hardware is used for the servers. Usually this means the server has redundant CPUs, hard drives, and power supplies.

This solution tends to offer greater performance gains then either solution individually as well as offer greater mean time between failures and better support. It still allows individual machines to be taken down for maintenance without losing the service but also increases the benefits of adding new machines to the cluster. The only drawbacks to this approach is that it is more costly than the load balanced commodity hardware approach, though it is usually still cheaper then the single-system approach, and it still presents a problem for applications not designed to run in load balanced environments.

Services

When it comes to deploying a service most service designers assume the service will be deployed on a single machine. If it is then deployed in a highly available environment, as described above, odd and indeterministic behavior can be exhibited. There are at least three broad categories of issues that can cause this behavior and any service may demonstrate one or more of them. These issues, described next, can often times be avoided with little trouble during the design of the service but it becomes far harder to fix them after development, especially if the service is an off-the-shelf application.

Persistent Connections

When a client communicates to a service the most costly operation, time-wise and sometimes resource-wise, is creating the communication channel between the two. Therefore most applications will, if the communication protocol being used allows it, try to create persistent connections. Persistent connections are connection that are created once and used many times, that is they persist between each usage. This allows a client to only incur the cost of creating the connection once.

A few problems can arise if the client is connecting to a service that is placed in a highly available environment. First, if the client is unable to reach the same instance of the service it had originally communicated with the clients persistent connection will fail and the client may report the service as down, which may not be the case. This situation

can occur either because the node to which the client originally communicated with went down, or because the load balancer sent the client's request to another node.

The easiest way to avoid this problem is, of course, not to use persistent connections. Such a solution, however, is rarely practical, especially if the client is another application that makes very frequent requests to a service. Instead a better way to handle this problem is to make sure the client gracefully handles persistent connection failures and, when a failure is encountered, tries to re-establish the connection.

Most load balancers can also be set up to allow a client to communicate with the same node, assuming the node is still available, that it originally communicated with. This setup, often referred to as sticky sessions or session affinity, coupled with the client gracefully handling failed connections, as mentioned above, greatly reduces the likelihood of encountering a problem when using persistent connections.

Local Storage of Data

Many applications store things like configuration files, internal state information, or other data on the local file system. If this data is only ever used for the internal functionally of the service it is unlikely that problems will arise, except for the head ache administrators will get having to change configuration files on every node when a configuration change is needed. If, however, the service stores, and operates on, information locally and then returns it to a client problems will arise in a highly available environment.

The problem is that this information will not be available to all the other nodes. For example consider a message board service which stores messages in a file on the local file system. A client initiates a request and ends up at Node1, the client posts a message, and leaves. Later the client comes back, to view the message board, ends up at Node2, and does not see their message. This is not because the service did not receive the message and properly store it but instead because the message is stored on the local file system of Node1.

The solution to this problem is, like the previous problem, fairly easy. Do not store information locally, instead use some central data store (e.g. a database). This allows all instances of a service to read and act on the same data store. If this option is used though there are two things the data store will need to do. First, it has to have mechanisms in place to handle multiple write requests at the same time, otherwise it may become corrupt. Second, it needs to be redundant so that a single point of failure is not introduced into the highly available environment.

In-memory information

Every service deployed stores some amount of information in memory and like information stored on the local file system if this information is only used internally by the application there is no issue. Most services, however, employ techniques such as caching and in-memory session management which can lead to problems in a highly available environment.

If a service uses in-memory caching, and a vast majority of them do, there are a few things that can be done, at design time, to alleviate or diminish problems that may

arise. First, if it is okay to return old data that may have changed on another node, nothing needs to be done, if this is not acceptable however then the caches between nodes must be synchronized. Such a synchronization process, at the very least, will need to invalidate a cached object from all caches when it is invalidated or changed in one cache. It may also replicate the addition of objects to the cache as well as changes that occur to cached objects, instead of just invalidating those objects when a change occurs. This type of caching system is sometimes referred to as a lateral caching system. So, if a service requires cache consistency some sort of lateral caching will need to be employed. Unfortunately lateral caching libraries are not available for most programming languages and those libraries that are available are immature.

Another common area where the use of in-memory artifacts may cause problems is when a service employs some sort of session management. Sessions often store, in memory, information specific to each client. If the client arrives at a different node then the one that initiated the session this information will not be available which may lead to odd behavior or at the very least inconvenient behavior for the client. The most common type of services that do this are web applications.

If a service does employ session management there are a couple things that can be done to address problems that occur if the service is placed in a highly available environment. First, a lateral cache could be used to replicate session information to each node. Second, if the service is a web application, many enterprise web application servers automatically replication sessions if they are aware that they are in a highly available environment.

A load balancer employing sticky sessions can also go a long way to addressing these issues as well. If a client always communicates with the same node these in-memory issues go away so long as the node never becomes inactive. Since this should be a very rare occasion it is often acceptable to use sticky sessions to solve this problem. In cases where session management is critical, for example in financial transactions, it may still be necessary to recover from a node failure which means sessions information will still need to be replicated.

Conclusion

While highly available service environments require a great deal of thought in order to plan and operate well most organization that do a large amount of electronic transaction will eventually encounter a need for them. It is vitally important to note however that like all technologies these environments are not magic panaceas that will fix availability and performance problems in every situation. They are simply a useful tool to consider when faced with such issues. Highly available service environments can pound out a lot of issues, the trick is to not view everything as a nail.