Notes on building applications for the NeCTAR cloud.

These notes are a list of some of the points that you should be aware of when putting an application up into the NeCTAR cloud. There are recommendations sprinkled around these points, but like all things in life, they should be considered in the context of your application: where they might not be appropriate.

The most important issue to be aware of is that the NeCTAR cloud is shared by many other users and that its resources are dynamically reallocated according to demand. Thus the NeCTAR environment is very different to the static environment offered by a dedicated server.

It is for this reason that the idea of cloud servers being treated more like cattle than puppies (http://www.networkworld.com/article/2165267/cloud-computing/why-servers-should-be-seen-like-cows--not-puppies.html) is taking hold.

Hopefully this set of notes offers some guidance as to that more dynamic bovine environment.

Infrastructure

Images and instances

An image is a copy of the disk of a server with a basic operating system installed on it.
Essentially an image is the prototype for an instance.
You can create and upload your own, or use one of the NeCTAR provided images. If you make your own, OpenStack expects certain packages to be installed on it.
Instances are virtual machines, booted from a cloned copy of an image.

Images and instances rules of thumb

You should have an image management strategy!
As instance are virtual, watch out for I/O issues: for example a 'noisy neighbour', running on the same host machine, might be hogging the available bandwidth.

Network

Currently each instance is allocated a public static IP address on boot that will be reallocated to another instance on shutdown.
At some future date you will possible be able to set up an internal on cloud network, but at the moment you only have the public static IP addresses to work with.
Security Groups allow you to create and apply rules that allow incoming traffic to an instance to be filtered by type and origin.
All outgoing traffic from an instance is allowed.

Storage

When an instance is terminated on the NeCTAR cloud, it, and its associated hard drive, vanish. All software and configuration changes on that instance that might have occurred since the instance was launched is lost.
With this in mind, NeCTAR offers three types of storage:

- Transient storage is tied to the instance and dies with the instance. You will never be able to access transient storage for an instance once the instance is gone.
- Block storage is network attached storage that persists beyond the life of the instance (Cinder).
- Object storage (Swift) that persists beyond the life of the instance.

Transient Storage

The 10Gig root file system of the instance is transient.
There is also additional transient storage made available to the instance. Some operating systems, such as Ubuntu, will have this additional storage automatically formatted and mounted for you (/mnt). Other, less friendly operating systems, leave you to do the hard work of formatting and mounting this extra transient storage yourself.

Block Storage

A volume backed by block storage can only be connected to one instance at any give time.
If block storage is not flushed before being disconnected from an instance writes can be lost.

Object Storage

Object storage is not a filesystem! It is slow compared to local storage, (check!), requires a special API to access, and objects written to it must be less than 5 Gig in size.
Objects larger than 5 Gig are handled by the API, but are broken up into chunks.

Storage Summary Table

	Object	**Transient**	**Block**
Speed	Low for write, reasonable for read (is this true?)	Unpredictable	High (is this true?)
Durability	Super high (uses replication to guard against data loss)	Super low - lost whenever the instance is lost	High - persists beyond the life of the instance
Flexibility	Low (basically supports uploads and downloads into containers)	Medium (a full file system, but tied to the duration of the associated instance)	High (a durable full file system that can be serially shared amongst instances)
Complexity	High (in that software/scripts have to be written/installed to work with it)	Low	Medium (in that it has to created and mounted etc.)

Strength	Disaster recovery, backups, static assets	Transient data	Operational data
Weakness	Operational data	Non Transient data	Lots of small I/O

Storage rules of thumb

If an instance dies or is terminated you will loose any data on it that hasn't been backed up. So don't store persistent data on transient storage if you can help it...
An instance will, however, preserve its transient storage across reboots.
As mentioned, block storage can become corrupted if not properly disconnected! So when disconnecting it, flush all buffers, sync it, unmount it, then detach it. Note that a dying instance can cause corruption of block storage. So snapshot block storage regularly, back it up, and then discard the snapshot (see snapshot rules of thumb).
Given the 10Gig size limit on the root disk of an instance some applications may require that the application directories are copied to the extra ephemeral volume (and possibly be aliased to directories on the on-instance volume if the application can't be modified to point to the copied directories).
Move log files onto the extended transient storage, if possible.

Snapshots

A snapshot is a copy of a block volume or of the primary ephemeral drive of an instance. They are useful for backup and for replication.

Snapshot rules of thumb

You should stop databases before making a snapshot and flush caches to disk.
(more on snapshots here: http://docs.openstack.org/openstack-ops/content/snapshots.html)
Image snapshots are not incremental, as they are written to the object store.
Some block storage snapshots implementations (LVM) choose to do their snapshots as an incremental operation. In this scenario the more snapshots you take, the slower the storage becomes.
If only snapshotting an instance only, shut it off (*do not terminate it*), then make your snapshot. Don't forget to restart your instance!

Software

Licenses

It is important to know the and understand the software licences of the applications and code that is being used!
Open Source licences are probably the simplest and least risky to deal with.
Also know how the software being used audits licences. Some license audit techniques can break in the virtual environment of the cloud.
Know that if the software requires access to licence server it probably will not easily be run on the NeCTAR cloud...

Databases

The are several different ways of speeding up or optimising databases that aren't coping.

Stored procedures

Stored procedures might look attractive. But they:

- Aren't portable
- Need an understanding of database programming
- Can make applications more complex

Sharding

In sharding you segment the data across a number of servers so that only a subset of the data is on a given server.
Sharding can offer dramatic performance improvements.
However, sharding is complicated to setup and manage.
If you lose a shard, your life can become very traumatic, very quickly.
So avoid sharding if at all possible.

Clustering

In clustering two or more instances appear to be a single instance to the clients that they serve.
This arrangement provides fault tolerance, reliability and load balancing between the instances.
However, clustering is complicated to setup and manage. It also introduces latency issues if the cluster spans different zones.
Only cluster a database if you have a DBA on hand.

Replication

In replication, one database is set up as a master and others, termed slaves, slavishly copy everything that occurs in the master database.
Replication is not as reliable as clustering. Slaves can either fall behind, or the master can propagate corrupt data.
However, replication is a cheaper and simpler alternative to set up when compared to clustering or sharding.
Database exports/backups can be taken off of the slave database. If your backup frequency is too high, be aware that the slave might not properly track the master database...
If a slave fails, simply start a new one and it point to the master.
If the master fails you can:

1. Try restart from the master image and mount the old masters block storage device (easiest to do, but the block storage may have been corrupted)
2. Promote a slave to master and start new slave (fastest to do)
3. Build a new master and take its data from the slave

If both fail, rebuild the whole setup from the backups.

Database rules of thumb

If possible, use block storage for your database storage.
Don't rely on databases auto increment function for primary keys. This makes it harder to split your database up onto multiple machines at a later date.
Automate backups.
Regularly test that your backups do in fact restore!
If backing up a database via snapshots of a filesystem, don't forget to stop the database writes whilst snapshotting (lock the database, flush the buffers, sync the file system, take the

snapshot, unlock the database). If snapshotting block storage, backup the snapshot once done, then release it.
Block storage snapshots aren't portable, so if using block storage snapshots as your primary backup strategy, make sure that you do full database exports via a dump at regular intervals.
Copy the database backups to Swift: then to your own off site location. Make sure that the offsite location is totally independent of NeCTAR's infrastructure.
In a master slave environment you have to ensure that you never write to a slave! So make slaves read only (this does make slave promotion more difficult).

Disaster recovery

All software and hardware will fail at some point. So you should have disaster recovery plan to cover what to do when something breaks.
With regard to your plan:

- Document it!
- Automate it!
- Test it!

In documentation, specify:

- How much data you are willing to lose if there is a disaster (the Recovery Point Objective). Specify this in hours or days of data. Know that the less that is acceptable, the more you are going to invest in preparing against disaster.
- How much downtime is acceptable when recovering (the Recovery Time Objective). Again, specify in hours and again, know that a smaller figure will require a greater investment.
- The types of data (fixed/transient/configuration/persistent) and your policy toward them. e.g. Fixed: can rebuild with Puppet, no need to backup. Transient: don't care. Configuration, can be backed up only when changed, so put in version control. Persistent: the application state, so daily via database dumps.

Automation, using tools such as Packer, Puppet/Chef/Ansible and **Heat** etc. give you the power to reliably and repeatedly rebuild your infrastructure without much human intervention. This means that when your servers die you should have far less downtime. Automation also makes it easier for you to scale by adding extra servers to the mix.

Disaster recovery rules of thumb

Automate if possible!
You can have a slave system in a different zone and switch to it in the event of failure… If you do this be aware of possible network issues.
If you have another cloud provider as a recovery option, make sure you have the required infrastructure in place on that provider (images, etc), and that it is kept up to date. Also test by firing up your junk on that provider regularly!
You need to test both the parts and the whole of your disaster recovery regularly.

Monitoring

Monitoring is important: you can't take action to fix any failures if you don't know that there has in fact been a failure.
You need to monitor:

- The cloud state (is the NeCTAR cloud OK?) Ceilometer helps at this level.
- Instance state (are your instances in good order?) Ceilometer can help at this level as well, but is limited.
- Application state (are the applications still working as expected)

Be aware that on instance monitoring software might not be aware it is running in a virtualized environment - and report everything is 100% fine when the instance is being paged a lot…

Monitoring rules of thumb

Try to have at least one monitoring system that is independent of the NeCTAR cloud.
The most common stress points are:

- Application server CPU
- Application server RAM
- Database disk I/O
- Actual storage and disk usage (running out of storage/disk can be very painful)

You should monitor disk I/O on the database server. If this is the cause of speed issues, adding more application servers is simply going to slow the system down further.
If you automate your disaster recovery, the servers that do the monitoring should be able to kick off/control the process if you don't want to disturb your beauty sleep.
Regularly inspect your log files.

Security

There is no perimeter – so do not rely on perimeter security: instead focus on instance security.

Security rules of thumb

Read and follow the NeCTAR security guide.
Open as few ports as is possible, and close any ports not in use: this is easily done via security groups. So for example, disable ssh access on your security group when you don't need to use it.
Have a unique ssh key for each project you belong to.
Try to avoid a rule of 0.0.0.0/0 in your security groups as much as possible.
Use security groups to mimic layered perimeter zones, though. Do this by limiting traffic to specific security groups or IP addresses/ranges.
Be negative: assume your data is vulnerable.
If you have sensitive data you should zero out all ephemeral and block storage devices when you are finished with them.
Leverage encrypted file systems.
Encrypt sensitive data before placing it in Swift.
Encrypt database, backups and all network communications.
If encrypting backups is chewing up CPU time, you can simply spin up another server to do the encryption.
If you have sensitive data, then, if possible, partition the data so that if an individual data server is compromised, the data is useless unless other data servers are also compromised.
Put more sensitive data on its own server with security group restricting access to only the application servers. Try to make sure the application server and the sensitive data server run different software (i.e.: don't have common attack vectors).
Consider putting a reverse proxy on the server. It might impact performance, but it adds an extra layer between the world and the application server.
Keep patching your servers! Set them up to patch themselves automatically.

Use Host Intrusion Detection, such as OSSEC.
You should never have user account passwords on a server. Certificate access only!
Never let your private keys go onto the cloud.
Never put your keys/security credentials into version control. Laugh, but people do this…
Never hard code your keys/security credentials into your code. Again, laugh...
Make sure you will never loose your private keys. If need be, you can print the keys out on paper and put them in a safety deposit box.
Don't keep backup decryption keys with the backups. Sound obvious, but...
Make sure more than one person can and knows how to decrypt anything (and that they know how to get the keys needed)
Have your monitoring server poll your instances rather than have your instances report to your monitoring server (you can thus firewall the monitoring server to restrict incoming traffic)
Turn on SELinux

- $ sestatus # shows the SE linux status
- $setenforce enforcing # turns it on.

Review log files regularly
Store log files on Swift. That way you can always recover them if your instance goes down - and they don't take up space on your instance.

When Compromised

Snapshot the disk and block storage for later analysis.
Shutdown/turn off the compromised instance
Bring up a new server (remember, it will still has the attack vector, but it is not yet compromised)
Then build an isolated server and mount the compromised volumes and do a forensic examination.
Once you've worked out the attack vector patch the new server or rebuild it with fixes applied.

Design

Shared resources

Be aware that memory locks don't work outside of a single server environment. Hence applications that use memory locks to control access to shared resources can't scale.
There are several ways to control access to shared resources: use clusters, cross server shared memory, or simply make the database the authority via database locks or timestamps.

Timestamps

If your application does a lot of read only requests, database locks might slow the system down. In this scenario using timestamps to control access might be a superior approach.
To use timestamps simply add a last updated timestamp column to database tables. Then read it, and use the read value in the where clause of an update. If another client updates in the meantime, the update will fail. Don't forget to update the timestamp value in the update part of the query!
If using this technique beware of deadlocks if you have complex operations. You can work around this by introducing a locking table that controls access to a group of tables.
Also be aware that high conflict rates will result in worse performance than a locking system.

Scaling

The ability to easily and rapidly scale is one of the main draw cards of the cloud.
You can scale manually (ugh) or dynamically.
In dynamic scaling the software adjusts resources automatically. It can be:

- Proactive: scaling that is done in anticipation of demand.
- Reactive: scaling that is done in response to demand.

You can scale both horizontally (add extra resources - also known as scaling out) or vertically (move to a bigger server - also known as scaling up).
So know:

- What demand will be placed on your application
- Its expected usage patterns
- How the application will respond to load
- When adding capacity adds value, and when it doesn't (e.g: is the unexpected load user demand or a denial of service attack?)

Scaling rules of thumb

Automation makes it easier for you to scale dynamically.
Run load tests to see how and where you need to scale.
Plan for the expected, but be able to recognise the unexpected and know how to react (how to scale appropriately). For example, adding another application server to a system that is failing because of an I/O bound database will exacerbate the problem.
Reactive scaling can be bad when suffering a denial of service attack - you are just giving the attackers a more responsive target.
Don’t forget: if you scale up you’ll also want to scale down!

State

In moving to the cloud, state management will be your main issue to wrestle with.
The simplest solution is probably to put all state data into a database, which should be backed by block storage.
To scale software load balancers can be put in front of either a cluster or independent instances. Independent instances are easier to set up and scale, but can't share state – so with independent instances, state must be scoped to the immediate request. Clusters are harder to set up, and don't scale that well: but do share state.
Sticky sessions are ones whereby the load balance will send a user to the same instance each time. The instance holds session state (the instance is stateful). The instance thus becomes a possible point of failure. The sessions might also become unevenly distributed amongst instances, thus leading to some instances being heavily used whilst others are ignored.
An instance can be kept stateless by storing the session in a cookie (must be small), or by putting a session key in a cookie and retrieving the session from a store via the key. Stateless instances allow a load balancer to do round robin load balancing.

Queue Centric Workflow

One way of delivering responsive systems is use queues: you kick off long running tasks by placing a message onto a queue and returning to the user straight away. At some later time, the message is picked off of the queue by a receiver, and the requested task is performed. The receiver should only delete the message once the requested task is performed, as the queue marks the message as invisible to other tasks for a certain amount of time. Hence if the task doesn't delete it, it will re-appear in the queue. This gives a guarantee that the task is performed at least once.
The code that handles the task will have to guard against partially completed tasks. If there is a message that breaks the application (the message can never be handled - it is ‘poison’) it can be dealt with by tracking the dequeue count and only allowing a certain number of tries before refusing to handle the message and deleting it. If this isn't done, the 'poisoned' messages can bring the system to a halt...
The length of time it takes to handle messages is a good metric to understand system load.

Design rules of thumb

Converting existing applications to run on the cloud can be **expensive**…
Try not store application state on your instances transient storage. Some people do this by simply moving the applications data directories onto block storage. Be aware that this is a quick fix that will stop you from scaling out.
Design applications so logic can go across many servers, and can be run against the same database without communication between the servers.
If replicating databases, run reads against slaves and writes against the master. Read heavy applications can get big benefits!
Make sure your database is indexed appropriately.
You have many cores available: use them via threading!
Know your application's performance choke points.
Do not rely on Swift for fast write access.
Move static content to Swift, though (writes slow, reads fast). Then allow Swift to serve it.
Try to minimise traffic between zones.
To simplify backups and restores, binary data can be kept in the database as well, but if done it should be cached by the application server for performance…

Image management strategy

You have to trust your base image (you don't want one that might have trojans and/or backdoors in it). So avoid images of unknown provenance and base images that are kitchen sink images...
Avoid having unnecessary software installed on your image: it expands the attack surface. So it should only have software needed on it and nothing more!
If you build your own image, don't have your private keys embedded in it!
And of course, avoid having any sensitive data on an image.

Author:: Martin Paulo

Make sure you don't have passwords in configuration files on your image.
And of course avoid any stateful data on the image. Because you are doing this you might not want applications to start automatically when the instance boots (Heat is helpful here), but you still want them to be gracefully shut down on system shut down. So you need to know how the applications you are using startup and shutdown.
Test images once you've built them…
To harden an image:

Avoid having unnecessary services running: it expands the attack surface.

```
# /sbin/chkconfig --list |grep '3:on'
```

lists the services running on runlevel 3 (use `rcconf` in debian)

```
# chkconfig serviceName off
```

disables any you don't want...
Remove unneeded accounts (allow only specific users by editing the `/etc/sshd_conf` configuration file: `AllowUsers username )`
Run all services in a role if possible, don't just default to the root accounts
Run services in a restricted jail, if possible.
Verify system services have proper permissions.
Know what ports are being used

```
# netstat -tulpn
```

shows open ports and associated programs.
Change the default ssh port to another higher level one...

```
# vi /etc/ssh/sshd_config
```

Make sure root login is disabled
In `/etc/sshd_conf` configuration file: `PermitRootLogin no`

Keep your base image updated with the latest releases and patches...

Finally

Read up on the pros and cons of Microservices (and they do have cons!)

Author:: Martin Paulo