
Swift Documentation

Release 1.0.2

Swift Team

July 19, 2010

CONTENTS

1	The Rings	3
1.1	Ring Builder	3
1.2	Ring Data Structure	3
1.3	Building the Ring	5
1.4	History	5
2	The Account Reaper	7
2.1	History	7
3	The Auth System	9
3.1	History and Future	9
4	Replication	11
4.1	DB Replication	11
4.2	Object Replication	12
5	Development Guidelines	13
5.1	Coding Guidelines	13
5.2	Documentation Guidelines	13
5.3	License and Copyright	13
6	SAIO - Swift All In One	15
6.1	Instructions for setting up a dev VM	15
6.2	Debugging Issues	23
7	Partitioned Consistent Hash Ring	25
7.1	Ring	25
7.2	Ring Builder	25
8	Proxy	27
8.1	Proxy Server	27
9	Account	29
9.1	Account Server	29
9.2	Account Auditor	29
9.3	Account Reaper	29
10	Container	31
10.1	Container Server	31
10.2	Container Updater	31

10.3	Container Auditor	31
11	Account DB and Container DB	33
11.1	DB	33
11.2	DB replicator	33
12	Object	35
12.1	Object Server	35
12.2	Object Replicator	35
12.3	Object Updater	35
12.4	Object Auditor	35
13	Developer's Authorization	37
13.1	Auth Server	37
14	Misc	39
14.1	Exceptions	39
14.2	Constraints	39
14.3	Utils	39
14.4	Auth	39
14.5	WSGI	39
14.6	Client	39
14.7	Direct Client	44
14.8	Buffered HTTP	44
14.9	Healthcheck	44
14.10	MemCacheD	44
15	Indices and tables	47
	Python Module Index	49

Overview:

THE RINGS

The rings determine where data should reside in the cluster. There is a separate ring for account databases, container databases, and individual objects but each ring works in the same way. These rings are externally managed, in that the server processes themselves do not modify the rings, they are instead given new rings modified by other tools.

The ring uses a configurable number of bits from a path's MD5 hash as a partition index that designates a device. The number of bits kept from the hash is known as the partition power, and 2 to the partition power indicates the partition count. Partitioning the full MD5 hash ring allows other parts of the cluster to work in batches of items at once which ends up either more efficient or at least less complex than working with each item separately or the entire cluster all at once.

Another configurable value is the replica count, which indicates how many of the partition->device assignments comprise a single ring. For a given partition number, each replica's device will not be in the same zone as any other replica's device. Zones can be used to group devices based on physical locations, power separations, network separations, or any other attribute that would lessen multiple replicas being unavailable at the same time.

1.1 Ring Builder

The rings are built and managed manually by a utility called the ring-builder. The ring-builder assigns partitions to devices and writes an optimized Python structure to a gzipped, pickled file on disk for shipping out to the servers. The server processes just check the modification time of the file occasionally and reload their in-memory copies of the ring structure as needed. Because of how the ring-builder manages changes to the ring, using a slightly older ring usually just means one of the three replicas for a subset of the partitions will be incorrect, which can be easily worked around.

The ring-builder also keeps its own builder file with the ring information and additional data required to build future rings. It is very important to keep multiple backup copies of these builder files. One option is to copy the builder files out to every server while copying the ring files themselves. Another is to upload the builder files into the cluster itself. Complete loss of a builder file will mean creating a new ring from scratch, nearly all partitions will end up assigned to different devices, and therefore nearly all data stored will have to be replicated to new locations. So, recovery from a builder file loss is possible, but data will definitely be unreachable for an extended time.

1.2 Ring Data Structure

The ring data structure consists of three top level fields: a list of devices in the cluster, a list of lists of device ids indicating partition to device assignments, and an integer indicating the number of bits to shift an MD5 hash to calculate the partition for the hash.

1.2.1 List of Devices

The list of devices is known internally to the Ring class as `devs`. Each item in the list of devices is a dictionary with the following keys:

<code>id</code>	<code>integer</code>	The index into the list devices.
<code>zone</code>	<code>integer</code>	The zone the devices resides in.
<code>weight</code>	<code>float</code>	The relative weight of the device in comparison to other devices. This usually corresponds directly to the amount of disk space the device has compared to other devices. For instance a device with 1 terabyte of space might have a weight of 100.0 and another device with 2 terabytes of space might have a weight of 200.0. This weight can also be used to bring back into balance a device that has ended up with more or less data than desired over time. A good average weight of 100.0 allows flexibility in lowering the weight later if necessary.
<code>ip</code>	<code>string</code>	The IP address of the server containing the device.
<code>port</code>	<code>int</code>	The TCP port the listening server process uses that serves requests for the device.
<code>device</code>	<code>string</code>	The on disk name of the device on the server. For example: <code>sdb1</code>
<code>meta</code>	<code>string</code>	A general-use field for storing additional information for the device. This information isn't used directly by the server processes, but can be useful in debugging. For example, the date and time of installation and hardware manufacturer could be stored here.

Note: The list of devices may contain holes, or indexes set to `None`, for devices that have been removed from the cluster. Generally, device ids are not reused. Also, some devices may be temporarily disabled by setting their weight to 0.0. To obtain a list of active devices (for uptime polling, for example) the Python code would look like: `devices = [device for device in self.devs if device and device['weight']]`

1.2.2 Partition Assignment List

This is a list of `array('I')` of devices ids. The outermost list contains an `array('I')` for each replica. Each `array('I')` has a length equal to the partition count for the ring. Each integer in the `array('I')` is an index into the above list of devices. The partition list is known internally to the Ring class as `_replica2part2dev_id`.

So, to create a list of device dictionaries assigned to a partition, the Python code would look like:

```
devices = [self.devs[part2dev_id[partition]] for part2dev_id in self._replica2part2dev_id]
```

`array('I')` is used for memory conservation as there may be millions of partitions.

1.2.3 Partition Shift Value

The partition shift value is known internally to the Ring class as `_part_shift`. This value used to shift an MD5 hash to calculate the partition on which the data for that hash should reside. Only the top four bytes of the hash is used in this process. For example, to compute the partition for the path `/account/container/object` the Python code might look like: `partition = unpack_from('>I', md5('/account/container/object').digest())[0] >> self._part_shift`

1.3 Building the Ring

The initial building of the ring first calculates the number of partitions that should ideally be assigned to each device based on the device's weight. For example, if the partition power of 20 the ring will have 1,048,576 partitions. If there are 1,000 devices of equal weight they will each desire 1,048.576 partitions. The devices are then sorted by the number of partitions they desire and kept in order throughout the initialization process.

Then, the ring builder assigns each partition's replica to the device that desires the most partitions at that point, with the restriction that the device is not in the same zone as any other replica for that partition. Once assigned, the device's desired partition count is decremented and moved to its new sorted location in the list of devices and the process continues.

When building a new ring based on an old ring, the desired number of partitions each device wants is recalculated. Next the partitions to be reassigned are gathered up. Any removed devices have all their assigned partitions unassigned and added to the gathered list. Any devices that have more partitions than they now desire have random partitions unassigned from them and added to the gathered list. Lastly, the gathered partitions are then reassigned to devices using a similar method as in the initial assignment described above.

Whenever a partition has a replica reassigned, the time of the reassignment is recorded. This is taken into account when gathering partitions to reassign so that no partition is moved twice in a configurable amount of time. This configurable amount of time is known internally to the RingBuilder class as `min_part_hours`. This restriction is ignored for replicas of partitions on devices that have been removed, as removing a device only happens on device failure and there's no choice but to make a reassignment.

The above processes don't always perfectly rebalance a ring due to the random nature of gathering partitions for reassignment. To help reach a more balanced ring, the rebalance process is repeated until near perfect (less 1% off) or when the balance doesn't improve by at least 1% (indicating we probably can't get perfect balance due to wildly imbalanced zones or too many partitions recently moved).

1.4 History

The ring code went through many iterations before arriving at what it is now and while it has been stable for a while now, the algorithm may be tweaked or perhaps even fundamentally changed if new ideas emerge. This section will try to describe the previous ideas attempted and attempt to explain why they were discarded.

A "live ring" option was considered where each server could maintain its own copy of the ring and the servers would use a gossip protocol to communicate the changes they made. This was discarded as too complex and error prone to code correctly in the project time span available. One bug could easily gossip bad data out to the entire cluster and be difficult to recover from. Having an externally managed ring simplifies the process, allows full validation of data before it's shipped out to the servers, and guarantees each server is using a ring from the same timeline. It also means that the servers themselves aren't spending a lot of resources maintaining rings.

A couple of "ring server" options were considered. One was where all ring lookups would be done by calling a service on a separate server or set of servers, but this was discarded due to the latency involved. Another was much like the current process but where servers could submit change requests to the ring server to have a new ring built and shipped back out to the servers. This was discarded due to project time constraints and because ring changes are currently infrequent enough that manual control was sufficient. However, lack of quick automatic ring changes did mean that other parts of the system had to be coded to handle devices being unavailable for a period of hours until someone could manually update the ring.

The current ring process has each replica of a partition independently assigned to a device. A version of the ring that used a third of the memory was tried, where the first replica of a partition was directly assigned and the other two were determined by "walking" the ring until finding additional devices in other zones. This was discarded as control was lost as to how many replicas for a given partition moved at once. Keeping each replica independent allows for moving

only one partition replica within a given time window (except due to device failures). Using the additional memory was deemed a good tradeoff for moving data around the cluster much less often.

Another ring design was tried where the partition to device assignments weren't stored in a big list in memory but instead each device was assigned a set of hashes, or anchors. The partition would be determined from the data item's hash and the nearest device anchors would determine where the replicas should be stored. However, to get reasonable distribution of data each device had to have a lot of anchors and walking through those anchors to find replicas started to add up. In the end, the memory savings wasn't that great and more processing power was used, so the idea was discarded.

A completely non-partitioned ring was also tried but discarded as the partitioning helps many other parts of the system, especially replication. Replication can be attempted and retried in a partition batch with the other replicas rather than each data item independently attempted and retried. Hashes of directory structures can be calculated and compared with other replicas to reduce directory walking and network traffic.

Partitioning and independently assigning partition replicas also allowed for the best balanced cluster. The best of the other strategies tended to give $\pm 10\%$ variance on device balance with devices of equal weight and $\pm 15\%$ with devices of varying weights. The current strategy allows us to get $\pm 3\%$ and $\pm 8\%$ respectively.

Various hashing algorithms were tried. SHA offers better security, but the ring doesn't need to be cryptographically secure and SHA is slower. Murmur was much faster, but MD5 was built-in and hash computation is a small percentage of the overall request handling time. In all, once it was decided the servers wouldn't be maintaining the rings themselves anyway and only doing hash lookups, MD5 was chosen for its general availability, good distribution, and adequate speed.

THE ACCOUNT REAPER

The Account Reaper removes data from deleted accounts in the background.

An account is marked for deletion by a reseller through the services server's `remove_storage_account` XMLRPC call. This simply puts the value `DELETED` into the status column of the `account_stat` table in the account database (and replicas), indicating the data for the account should be deleted later. There is no set retention time and no `undelete`; it is assumed the reseller will implement such features and only call `remove_storage_account` once it is truly desired the account's data be removed.

The account reaper runs on each account server and scans the server occasionally for account databases marked for deletion. It will only trigger on accounts that server is the primary node for, so that multiple account servers aren't all trying to do the same work at the same time. Using multiple servers to delete one account might improve deletion speed, but requires coordination so they aren't duplicating effort. Speed really isn't as much of a concern with data deletion and large accounts aren't deleted that often.

The deletion process for an account itself is pretty straightforward. For each container in the account, each object is deleted and then the container is deleted. Any deletion requests that fail won't stop the overall process, but will cause the overall process to fail eventually (for example, if an object delete times out, the container won't be able to be deleted later and therefore the account won't be deleted either). The overall process continues even on a failure so that it doesn't get hung up reclaiming cluster space because of one troublesome spot. The account reaper will keep trying to delete an account until it eventually becomes empty, at which point the database reclaim process within the `db_replicator` will eventually remove the database files.

2.1 History

At first, a simple approach of deleting an account through completely external calls was considered as it required no changes to the system. All data would simply be deleted in the same way the actual user would, through the public ReST API. However, the downside was that it would use proxy resources and log everything when it didn't really need to. Also, it would likely need a dedicated server or two, just for issuing the delete requests.

A completely bottom-up approach was also considered, where the object and container servers would occasionally scan the data they held and check if the account was deleted, removing the data if so. The upside was the speed of reclamation with no impact on the proxies or logging, but the downside was that nearly 100% of the scanning would result in no action creating a lot of I/O load for no reason.

A more container server centric approach was also considered, where the account server would mark all the containers for deletion and the container servers would delete the objects in each container and then themselves. This has the benefit of still speedy reclamation for accounts with a lot of containers, but has the downside of a pretty big load spike. The process could be slowed down to alleviate the load spike possibility, but then the benefit of speedy reclamation is lost and what's left is just a more complex process. Also, scanning all the containers for those marked for deletion when the majority wouldn't be seemed wasteful. The `db_replicator` could do this work while performing its replication scan, but it would have to spawn and track deletion processes which seemed needlessly complex.

In the end, an account server centric approach seemed best, as described above.

THE AUTH SYSTEM

The auth system for Swift is based on the auth system from an existing architecture – actually from a few existing auth systems – and is therefore a bit disjointed. The distilled points about it are:

- The authentication/authorization part is outside Swift itself
- The user of Swift passes in an auth token with each request
- Swift validates each token with the external auth system and caches the result
- The token does not change from request to request, but does expire

The token can be passed into Swift using the `X-Auth-Token` or the `X-Storage-Token` header. Both have the same format: just a simple string representing the token. Some external systems use UUID tokens, some an MD5 hash of something unique, some use “something else” but the salient point is that the token is a string which can be sent as-is back to the auth system for validation.

The validation call is, for historical reasons, an XMLRPC call. There are two types of auth systems, type 0 and type 1. With type 0, the XMLRPC call is given the token and the Swift account name (also known as the account hash because it’s usually of the format `<reseller>_<hash>`). With type 1, the call is given the container name and HTTP method as well as the token and account hash. Both types are also given a service login and password recorded in Swift’s `resellers.conf`. For a valid token, both auth system types respond with a session TTL and overall expiration in seconds from now. Swift does not honor the session TTL but will cache the token up to the expiration time. Tokens can be purged through a call to Swift’s services server.

How the user gets the token to use with Swift is up to the reseller software itself. For instance, with Cloud Files the user has a starting URL to an auth system. The user starts a session by sending a ReST request to that auth system to receive the auth token, a URL to the Swift system, and a URL to the CDN system.

3.1 History and Future

What’s established in Swift for authentication/authorization has history from before Swift, so that won’t be recorded here. It was minimally integrated with Swift to meet project deadlines, but in the near future Swift should have a pluggable auth/reseller system to support the above as well as other architectures.

REPLICATION

Since each replica in swift functions independently, and clients generally require only a simple majority of nodes responding to consider an operation successful, transient failures like network partitions can quickly cause replicas to diverge. These differences are eventually reconciled by asynchronous, peer-to-peer replicator processes. The replicator processes traverse their local filesystems, concurrently performing operations in a manner that balances load across physical disks.

Replication uses a push model, with records and files generally only being copied from local to remote replicas. This is important because data on the node may not belong there (as in the case of handoffs and ring changes), and a replicator can't know what data exists elsewhere in the cluster that it should pull in. It's the duty of any node that contains data to ensure that data gets to where it belongs. Replica placement is handled by the ring.

Every deleted record or file in the system is marked by a tombstone, so that deletions can be replicated alongside creations. These tombstones are cleaned up by the replication process after a period of time referred to as the consistency window, which is related to replication duration and how long transient failures can remove a node from the cluster. Tombstone cleanup must be tied to replication to reach replica convergence.

If a replicator detects that a remote drive is has failed, it will use the ring's "get_more_nodes" interface to choose an alternate node to synchronize with. The replicator can generally maintain desired levels of replication in the face of hardware failures, though some replicas may not be in an immediately usable location.

Replication is an area of active development, and likely rife with potential improvements to speed and correctness.

There are two major classes of replicator - the db replicator, which replicates accounts and containers, and the object replicator, which replicates object data.

4.1 DB Replication

The first step performed by db replication is a low-cost hash comparison to find out whether or not two replicas already match. Under normal operation, this check is able to verify that most databases in the system are already synchronized very quickly. If the hashes differ, the replicator brings the databases in sync by sharing records added since the last sync point.

This sync point is a high water mark noting the last record at which two databases were known to be in sync, and is stored in each database as a tuple of the remote database id and record id. Database ids are unique amongst all replicas of the database, and record ids are monotonically increasing integers. After all new records have been pushed to the remote database, the entire sync table of the local database is pushed, so the remote database knows it's now in sync with everyone the local database has previously synchronized with.

If a replica is found to be missing entirely, the whole local database file is transmitted to the peer using rsync(1) and vested with a new unique id.

In practice, DB replication can process hundreds of databases per concurrency setting per second (up to the number of available CPUs or disks) and is bound by the number of DB transactions that must be performed.

4.2 Object Replication

The initial implementation of object replication simply performed an rsync to push data from a local partition to all remote servers it was expected to exist on. While this performed adequately at small scale, replication times skyrocketed once directory structures could no longer be held in RAM. We now use a modification of this scheme in which a hash of the contents for each suffix directory is saved to a per-partition hashes file. The hash for a suffix directory is invalidated when the contents of that suffix directory are modified.

The object replication process reads in these hash files, calculating any invalidated hashes. It then transmits the hashes to each remote server that should hold the partition, and only suffix directories with differing hashes on the remote server are rsynced. After pushing files to the remote server, the replication process notifies it to recalculate hashes for the rsynced suffix directories.

Performance of object replication is generally bound by the number of uncached directories it has to traverse, usually as a result of invalidated suffix directory hashes. Using write volume and partition counts from our running systems, it was designed so that around 2% of the hash space on a normal node will be invalidated per day, which has experimentally given us acceptable replication speeds.

Development:

DEVELOPMENT GUIDELINES

5.1 Coding Guidelines

For the most part we try to follow PEP 8 guidelines which can be viewed here: <http://www.python.org/dev/peps/pep-0008/>

There is a useful pep8 command line tool for checking files for pep8 compliance which can be installed with `easy_install pep8`.

5.2 Documentation Guidelines

The documentation in docstrings should follow the PEP 257 conventions (as mentioned in the PEP 8 guidelines).

More specifically:

1. Triple quotes should be used for all docstrings.
2. If the docstring is simple and fits on one line, then just use one line.
3. For docstrings that take multiple lines, there should be a newline after the opening quotes, and before the closing quotes.
4. Sphinx is used to build documentation, so use the restructured text markup to designate parameters, return values, etc. Documentation on the sphinx specific markup can be found here: <http://sphinx.pocoo.org/markup/index.html>

5.3 License and Copyright

Every source file should have the following copyright and license statement at the top:

```
# Copyright (c) 2010 OpenStack, LLC.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
```

```
# implied.  
# See the License for the specific language governing permissions and  
# limitations under the License.
```

SAIO - SWIFT ALL IN ONE

6.1 Instructions for setting up a dev VM

This documents setting up a virtual machine for doing Swift development. The virtual machine will emulate running a four node Swift cluster. It assumes you're using *VMware Fusion 3* on *Mac OS X Snow Leopard*, but should give a good idea what to do on other environments.

- Get the *Ubuntu 10.04 LTS (Lucid Lynx)* server image from: <http://cdimage.ubuntu.com/releases/10.04/release/ubuntu-10.04-dvd-amd64.iso>
- Create guest virtual machine:
 1. *Continue without disc*
 2. *Use operating system installation disc image file*, pick the .iso from above.
 3. Select *Linux and Ubuntu 64-bit*.
 4. Fill in the *Linux Easy Install* details.
 5. *Customize Settings*, name the image whatever you want (*SAIO* for instance.)
 6. When the *Settings* window comes up, select *Hard Disk*, create an extra disk (the defaults are fine).
 7. Start the virtual machine up and wait for the easy install to finish.
- As root on guest (you'll have to log in as you, then *sudo su -*):
 1. *apt-get install python-software-properties*
 2. *add-apt-repository ppa:swift-core/ppa*
 3. *apt-get update*
 4. *apt-get install curl gcc bzip2 memcached python-configobj python-coverage python-dev python-nose python-setuptools python-simplejson python-xattr sqlite3 xfsprogs python-webob python-eventlet python-greenlet*
 5. Install anything else you want, like screen, ssh, vim, etc.
 6. *fdisk /dev/sdb* (set up a single partition)
 7. *mkfs.xfs -i size=1024 /dev/sdb1*
 8. *mkdir /mnt/sdb1*
 9. Edit */etc/fstab* and add */dev/sdb1 /mnt/sdb1 xfs noatime,nodiratime,nobarrier,logbufs=8 0 0*
 10. *mount /mnt/sdb1*
 11. *mkdir /mnt/sdb1/1 /mnt/sdb1/2 /mnt/sdb1/3 /mnt/sdb1/4 /mnt/sdb1/test*

12. `chown <your-user-name>:<your-group-name> /mnt/sdb1/*`
13. `mkdir /srv`
14. `for x in {1..4}; do ln -s /mnt/sdb1/$x /srv/$x; done`
15. `mkdir -p /etc/swift/object-server /etc/swift/container-server /etc/swift/account-server /srv/1/node/sdb1 /srv/2/node/sdb2 /srv/3/node/sdb3 /srv/4/node/sdb4 /var/run/swift`
16. `chown -R <your-user-name>:<your-group-name> /etc/swift /srv/[1-4]/ /var/run/swift` – **Make sure to include the trailing slash after /srv/[1-4]/**
17. Add to `/etc/rc.local` (before the `exit 0`):

```
mkdir /var/run/swift
chown <your-user-name>:<your-user-name> /var/run/swift
```

18. Create `/etc/rsyncd.conf`:

```
uid = <Your user name>
gid = <Your group name>
log file = /var/log/rsyncd.log
pid file = /var/run/rsyncd.pid

[account6012]
max connections = 25
path = /srv/1/node/
read only = false
lock file = /var/lock/account6012.lock

[account6022]
max connections = 25
path = /srv/2/node/
read only = false
lock file = /var/lock/account6022.lock

[account6032]
max connections = 25
path = /srv/3/node/
read only = false
lock file = /var/lock/account6032.lock

[account6042]
max connections = 25
path = /srv/4/node/
read only = false
lock file = /var/lock/account6042.lock

[container6011]
max connections = 25
path = /srv/1/node/
read only = false
lock file = /var/lock/container6011.lock

[container6021]
max connections = 25
path = /srv/2/node/
read only = false
lock file = /var/lock/container6021.lock
```

```
[container6031]
max connections = 25
path = /srv/3/node/
read only = false
lock file = /var/lock/container6031.lock
```

```
[container6041]
max connections = 25
path = /srv/4/node/
read only = false
lock file = /var/lock/container6041.lock
```

```
[object6010]
max connections = 25
path = /srv/1/node/
read only = false
lock file = /var/lock/object6010.lock
```

```
[object6020]
max connections = 25
path = /srv/2/node/
read only = false
lock file = /var/lock/object6020.lock
```

```
[object6030]
max connections = 25
path = /srv/3/node/
read only = false
lock file = /var/lock/object6030.lock
```

```
[object6040]
max connections = 25
path = /srv/4/node/
read only = false
lock file = /var/lock/object6040.lock
```

19. Edit the following line in `/etc/default/rsync`:

```
RSYNC_ENABLE=true
```

20. *service rsync restart*

- As you on guest:

1. *mkdir ~/bin*
2. Create *~/bazaar/bazaar.conf*:

```
[DEFAULT]
email = Your Name <your-email-address>
```

3. If you are using launchpad to get the code or make changes, run *bzr launchpad-login <launchpad_id>*
4. Create the swift repo with *bzr init-repo swift*
5. Check out your bzr branch of swift, for example: *cd ~/swift; bzr branch lp:swift trunk*
6. *cd ~/swift/trunk; sudo python setup.py develop*
7. Edit *~/bashrc* and add to the end:

```
export PATH_TO_TEST_XFS=/mnt/sdb1/test
export SWIFT_TEST_CONFIG_FILE=/etc/swift/func_test.conf
export PATH=${PATH}:~/bin
```

8. `./bashrc`

9. Create `/etc/swift/auth-server.conf`:

```
[auth-server]
default_cluster_url = http://127.0.0.1:8080/v1
user = <your-user-name>
```

10. Create `/etc/swift/proxy-server.conf`:

```
[proxy-server]
bind_port = 8080
user = <your-user-name>
```

11. Create `/etc/swift/account-server/1.conf`:

```
[account-server]
devices = /srv/1/node
mount_check = false
bind_port = 6012
user = <your-user-name>
```

```
[account-replicator]
vm_test_mode = yes
```

```
[account-auditor]
```

```
[account-reaper]
```

12. Create `/etc/swift/account-server/2.conf`:

```
[account-server]
devices = /srv/2/node
mount_check = false
bind_port = 6022
user = <your-user-name>
```

```
[account-replicator]
vm_test_mode = yes
```

```
[account-auditor]
```

```
[account-reaper]
```

13. Create `/etc/swift/account-server/3.conf`:

```
[account-server]
devices = /srv/3/node
mount_check = false
bind_port = 6032
user = <your-user-name>
```

```
[account-replicator]
vm_test_mode = yes
```

```
[account-auditor]
```

```
[account-reaper]
```

14. Create */etc/swift/account-server/4.conf*:

```
[account-server]
devices = /srv/4/node
mount_check = false
bind_port = 6042
user = <your-user-name>
```

```
[account-replicator]
vm_test_mode = yes
```

```
[account-auditor]
```

```
[account-reaper]
```

15. Create */etc/swift/container-server/1.conf*:

```
[container-server]
devices = /srv/1/node
mount_check = false
bind_port = 6011
user = <your-user-name>
```

```
[container-replicator]
vm_test_mode = yes
```

```
[container-updater]
```

```
[container-auditor]
```

16. Create */etc/swift/container-server/2.conf*:

```
[container-server]
devices = /srv/2/node
mount_check = false
bind_port = 6021
user = <your-user-name>
```

```
[container-replicator]
vm_test_mode = yes
```

```
[container-updater]
```

```
[container-auditor]
```

17. Create */etc/swift/container-server/3.conf*:

```
[container-server]
devices = /srv/3/node
mount_check = false
bind_port = 6031
user = <your-user-name>
```

```
[container-replicator]
vm_test_mode = yes
```

```
[container-updater]
```

```
[container-auditor]
```

18. Create */etc/swift/container-server/4.conf*:

```
[container-server]
devices = /srv/4/node
mount_check = false
bind_port = 6041
user = <your-user-name>
```

```
[container-replicator]
vm_test_mode = yes
```

```
[container-updater]
```

```
[container-auditor]
```

19. Create */etc/swift/object-server/1.conf*:

```
[object-server]
devices = /srv/1/node
mount_check = false
bind_port = 6010
user = <your-user-name>
```

```
[object-replicator]
vm_test_mode = yes
```

```
[object-updater]
```

```
[object-auditor]
```

20. Create */etc/swift/object-server/2.conf*:

```
[object-server]
devices = /srv/2/node
mount_check = false
bind_port = 6020
user = <your-user-name>
```

```
[object-replicator]
vm_test_mode = yes
```

```
[object-updater]
```

```
[object-auditor]
```

21. Create */etc/swift/object-server/3.conf*:

```
[object-server]
devices = /srv/3/node
mount_check = false
bind_port = 6030
user = <your-user-name>
```

```
[object-replicator]
vm_test_mode = yes
```

```
[object-updater]
```

```
[object-auditor]
```

22. Create `/etc/swift/object-server/4.conf`:

```
[object-server]
devices = /srv/4/node
mount_check = false
bind_port = 6040
user = <your-user-name>
```

```
[object-replicator]
vm_test_mode = yes
```

```
[object-updater]
```

```
[object-auditor]
```

23. Create `~/bin/resetswift`:

```
#!/bin/bash

swift-init all stop
sleep 5
sudo umount /mnt/sdb1
sudo mkfs.xfs -f -i size=1024 /dev/sdb1
sudo mount /mnt/sdb1
sudo mkdir /mnt/sdb1/1 /mnt/sdb1/2 /mnt/sdb1/3 /mnt/sdb1/4 /mnt/sdb1/test
sudo chown <your-user-name>:<your-group-name> /mnt/sdb1/*
mkdir -p /srv/1/node/sdb1 /srv/2/node/sdb2 /srv/3/node/sdb3 /srv/4/node/sdb4
sudo rm -f /var/log/debug /var/log/messages /var/log/rsyncd.log /var/log/syslog
sudo service rsyslog restart
sudo service memcached restart
```

24. Create `~/bin/remakerings`:

```
#!/bin/bash

cd /etc/swift

rm *.builder *.ring.gz backups/*.builder backups/*.ring.gz

swift-ring-builder object.builder create 18 3 1
swift-ring-builder object.builder add z1-127.0.0.1:6010/sdb1 1
swift-ring-builder object.builder add z2-127.0.0.1:6020/sdb2 1
swift-ring-builder object.builder add z3-127.0.0.1:6030/sdb3 1
swift-ring-builder object.builder add z4-127.0.0.1:6040/sdb4 1
swift-ring-builder object.builder rebalance
swift-ring-builder container.builder create 18 3 1
swift-ring-builder container.builder add z1-127.0.0.1:6011/sdb1 1
swift-ring-builder container.builder add z2-127.0.0.1:6021/sdb2 1
swift-ring-builder container.builder add z3-127.0.0.1:6031/sdb3 1
swift-ring-builder container.builder add z4-127.0.0.1:6041/sdb4 1
swift-ring-builder container.builder rebalance
swift-ring-builder account.builder create 18 3 1
swift-ring-builder account.builder add z1-127.0.0.1:6012/sdb1 1
swift-ring-builder account.builder add z2-127.0.0.1:6022/sdb2 1
swift-ring-builder account.builder add z3-127.0.0.1:6032/sdb3 1
swift-ring-builder account.builder add z4-127.0.0.1:6042/sdb4 1
swift-ring-builder account.builder rebalance
```

25. Create `~/bin/startmain`:

```
#!/bin/bash

swift-init auth-server start
swift-init proxy-server start
swift-init account-server start
swift-init container-server start
swift-init object-server start
```

26. Create `~/bin/startrest`:

```
#!/bin/bash

swift-auth-recreate-accounts
swift-init object-updater start
swift-init container-updater start
swift-init object-replicator start
swift-init container-replicator start
swift-init account-replicator start
swift-init object-auditor start
swift-init container-auditor start
swift-init account-auditor start
swift-init account-reaper start
```

27. `chmod +x ~/bin/*`

28. `remakerings`

29. `cd ~/swift/trunk; ./unittests`

30. `startmain` (The Unable to increase file descriptor limit. Running as non-root? warnings are expected and ok.)

31. `swift-auth-create-account test tester testing`

32. Get an `X-Storage-Url` and `X-Auth-Token`: `curl -v -H 'X-Storage-User: test:tester' -H 'X-Storage-Pass: testing' http://127.0.0.1:11000/v1.0`

33. Check that you can GET account: `curl -v -H 'X-Auth-Token: <token-from-x-auth-token-above>' <url-from-x-storage-url-above>`

34. Check that `st` works: `st -A http://127.0.0.1:11000/v1.0 -U test:tester -K testing stat`

35. Create `/etc/swift/func_test.conf`:

```
auth_host = 127.0.0.1
auth_port = 11000
auth_ssl = no

account = test
username = tester
password = testing

collate = C
```

36. `cd ~/swift/trunk; ./functests`

37. `cd ~/swift/trunk; ./probetests` (Note for future reference: probe tests will reset your environment)

If you plan to work on documentation (and who doesn't?!):

1. `sudo apt-get install python-sphinx`

2. `python setup.py build_sphinx`

6.2 Debugging Issues

If all doesn't go as planned, and tests fail, or you can't auth, or something doesn't work, here are some good starting places to look for issues:

1. Everything is logged in `/var/log/syslog`, so that is a good first place to look for errors (most likely python tracebacks).
2. Make sure all of the server processes are running. For the base functionality, the Proxy, Account, Container, Object and Auth servers should be running
3. If one of the servers are not running, and no errors are logged to syslog, it may be useful to try to start the server manually, for example: `swift-object-server /etc/swift/object-server/1.conf` will start the object server. If there are problems not showing up in syslog, then you will likely see the traceback on startup.

Source:

PARTITIONED CONSISTENT HASH RING

7.1 Ring

7.2 Ring Builder

PROXY

8.1 Proxy Server

ACCOUNT

9.1 Account Server

9.2 Account Auditor

9.3 Account Reaper

CONTAINER

10.1 Container Server

10.2 Container Updater

10.3 Container Auditor

ACCOUNT DB AND CONTAINER DB

11.1 DB

11.2 DB replicator

OBJECT

12.1 Object Server

12.2 Object Replicator

12.3 Object Updater

12.4 Object Auditor

DEVELOPER'S AUTHORIZATION

13.1 Auth Server

MISC

14.1 Exceptions

14.2 Constraints

14.3 Utils

14.4 Auth

14.5 WSGI

14.6 Client

Cloud Files client library used internally

```
exception swift.common.client.ClientException (msg, http_scheme='', http_host='',  
                                               http_port='', http_path='', http_query='',  
                                               http_status=0, http_reason='',  
                                               http_device='')
```

Bases: `exceptions.Exception`

```
class swift.common.client.Connection (authurl, user, key, retries=5, preauthurl=None, preauthto-  
                                       ken=None, snet=False)
```

Bases: `object`

Convenience class to make requests that will also retry the request

```
delete_container (container)  
    Wrapper for delete_container
```

```
delete_object (container, obj)  
    Wrapper for delete_object
```

```
get_account (marker=None, limit=None, prefix=None, full_listing=False)  
    Wrapper for get_account
```

```
get_container (container, marker=None, limit=None, prefix=None, delimiter=None,  
                full_listing=False)  
    Wrapper for get_container
```

get_object (*container, obj, resp_chunk_size=None*)
Wrapper for get_object

head_account ()
Wrapper for head_account

head_container (*container*)
Wrapper for head_container

head_object (*container, obj*)
Wrapper for head_object

post_object (*container, obj, metadata*)
Wrapper for post_object

put_container (*container*)
Wrapper for put_container

put_object (*container, obj, contents, metadata={}, content_length=None, etag=None, chunk_size=65536, content_type=None*)
Wrapper for put_object

`swift.common.client.delete_container` (*url, token, container, http_conn=None*)
Delete a container

Parameters

- **url** – storage URL
- **token** – auth token
- **container** – container name to delete
- **http_conn** – HTTP connection object (If None, it will create the conn object)

Raises ClientException HTTP DELETE request failed

`swift.common.client.delete_object` (*url, token, container, name, http_conn=None*)
Delete object

Parameters

- **url** – storage URL
- **token** – auth token
- **container** – container name that the object is in
- **name** – object name to delete
- **http_conn** – HTTP connection object (If None, it will create the conn object)

Raises ClientException HTTP DELETE request failed

`swift.common.client.get_account` (*url, token, marker=None, limit=None, prefix=None, http_conn=None, full_listing=False*)

Get a listing of containers for the account.

Parameters

- **url** – storage URL
- **token** – auth token
- **marker** – marker query
- **limit** – limit query

- **prefix** – prefix query
- **http_conn** – HTTP connection object (If None, it will create the conn object)
- **full_listing** – if True, return a full listing, else returns a max of 10000 listings

Returns a list of accounts

Raises ClientException HTTP GET request failed

```
swift.common.client.get_auth(url, user, key, snet=False)
```

Get authentication credentials

Parameters

- **url** – authentication URL
- **user** – user to auth as
- **key** – key or passowrd for auth
- **snet** – use SERVICENET internal network default is False

Returns tuple of (storage URL, storage token, auth token)

Raises ClientException HTTP GET request to auth URL failed

```
swift.common.client.get_container(url, token, container, marker=None, limit=None,
                                  prefix=None, delimiter=None, http_conn=None,
                                  full_listing=False)
```

Get a listing of objects for the container.

Parameters

- **url** – storage URL
- **token** – auth token
- **container** – container name to get a listing for
- **marker** – marker query
- **limit** – limit query
- **prefix** – prefix query
- **delimiter** – string to delimit the queries on
- **http_conn** – HTTP connection object (If None, it will create the conn object)
- **full_listing** – if True, return a full listing, else returns a max of 10000 listings

Returns a list of objects

Raises ClientException HTTP GET request failed

```
swift.common.client.get_object(url, token, container, name, http_conn=None,
                               resp_chunk_size=None)
```

Get an object

Parameters

- **url** – storage URL
- **token** – auth token
- **container** – container name that the object is in
- **name** – object name to get

- **http_conn** – HTTP connection object (If None, it will create the conn object)
- **resp_chunk_size** – if defined, chunk size of data to read

Returns a list of objects

Raises ClientException HTTP GET request failed

`swift.common.client.head_account` (*url, token, http_conn=None*)
Get account stats.

Parameters

- **url** – storage URL
- **token** – auth token
- **http_conn** – HTTP connection object (If None, it will create the conn object)

Returns a tuple of (container count, object count, bytes used)

Raises ClientException HTTP HEAD request failed

`swift.common.client.head_container` (*url, token, container, http_conn=None*)
Get container stats.

Parameters

- **url** – storage URL
- **token** – auth token
- **container** – container name to get stats for
- **http_conn** – HTTP connection object (If None, it will create the conn object)

Returns a tuple of (object count, bytes used)

Raises ClientException HTTP HEAD request failed

`swift.common.client.head_object` (*url, token, container, name, http_conn=None*)
Get object info

Parameters

- **url** – storage URL
- **token** – auth token
- **container** – container name that the object is in
- **name** – object name to get info for
- **http_conn** – HTTP connection object (If None, it will create the conn object)

Returns a tuple of (content type, content length, last modified, etag, dictionary of metadata)

Raises ClientException HTTP HEAD request failed

`swift.common.client.http_connection` (*url*)
Make an HTTPConnection or HTTPSConnection

Parameters

- **url** – url to connect to

Returns tuple of (parsed url, connection object)

Raises ClientException Unable to handle protocol scheme

`swift.common.client.post_object` (*url, token, container, name, metadata, http_conn=None*)

Change object metadata

Parameters

- **url** – storage URL
- **token** – auth token
- **container** – container name that the object is in
- **name** – object name to change
- **metadata** – dictionary of object metadata
- **http_conn** – HTTP connection object (If None, it will create the conn object)

Raises ClientException HTTP POST request failed

`swift.common.client.put_container` (*url, token, container, http_conn=None*)

Create a container

Parameters

- **url** – storage URL
- **token** – auth token
- **container** – container name to create
- **http_conn** – HTTP connection object (If None, it will create the conn object)

Raises ClientException HTTP PUT request failed

`swift.common.client.put_object` (*url, token, container, name, contents, metadata={}, content_length=None, etag=None, chunk_size=65536, content_type=None, http_conn=None*)

Put an object

Parameters

- **url** – storage URL
- **token** – auth token
- **container** – container name that the object is in
- **name** – object name to put
- **contents** – file like object to read object data from
- **metadata** – dictionary of object metadata
- **content_length** – value to send as content-length header
- **etag** – etag of contents
- **chunk_size** – chunk size of data to write
- **content_type** – value to send as content-type header
- **http_conn** – HTTP connection object (If None, it will create the conn object)

Returns etag from server response

Raises ClientException HTTP PUT request failed

`swift.common.client.quote` (*value, safe=''*)

Patched version of urllib.quote that encodes utf8 strings before quoting

14.7 Direct Client

14.8 Buffered HTTP

14.9 Healthcheck

14.10 MemCached

Lucid comes with memcached: v1.4.2. Protocol documentation for that version is at:

<http://github.com/memcached/memcached/blob/1.4.2/doc/protocol.txt>

```
class swift.common.memcached.MemcacheRing (servers, connect_timeout=0.29999999999999999,  
                                           io_timeout=2.0, tries=3)
```

Bases: object

Simple, consistent-hashed memcache client.

delete (*key*)

Deletes a key/value pair from memcache.

Parameters

- **key** – key to be deleted

get (*key*)

Gets the object specified by key. It will also unpickle the object before returning if it is pickled in memcache.

Parameters

- **key** – key

Returns value of the key in memcache

get_multi (*keys, server_key*)

Gets multiple values from memcache for the given keys.

Parameters

- **keys** – keys for values to be retrieved from memcache
- **server_key** – key to use in determining which server in the ring is used

Returns list of values

incr (*key, delta=1, timeout=0*)

Increments a key which has a numeric value by delta. If the key can't be found, it's added as delta.

Parameters

- **key** – key
- **delta** – amount to add to the value of key (or set as the value if the key is not found)
- **timeout** – ttl in memcache

set (*key, value, serialize=True, timeout=0*)

Set a key/value pair in memcache

Parameters

- **key** – key

- **value** – value
- **serialize** – if True, value is pickled before sending to memcache
- **timeout** – ttl in memcache

set_multi (*mapping, server_key, serialize=True, timeout=0*)

Sets multiple key/value pairs in memcache.

Parameters

- **mapping** – dictionary of keys and values to be set in memcache
- **server_key** – key to use in determining which server in the ring is used
- **serialize** – if True, value is pickled before sending to memcache
- **timeout** – ttl for memcache

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

S

`swift.common.client`, 39
`swift.common.memcached`, 44