Design & Implementation of BigV

Matthew Bloch

December 4, 2012

Abstract

BigV is a new hosting platform designed for "self-service" virtual machines (or VPSs or "cloud" servers). It is based around Linux's KVM virtualisation program and allows customers to rent access to high-performance hardware while segregating their systems. The software has been in development since August 2010, and running live machines since August 2011, and taking money since October 2012. Its designer will talk through decisions made during its specification, and how successive versions have been developed, deployed and rolled out to customers with little down time. Bytemark have built several similar smaller VM systems over the years and Matthew will show how Free software, a little scripting, and a clued-up system administrator can perform tricks that a commercial solutions can't.

1 What is BigV and what can it do now?

BigV is a set of software components for supervision, segregation and selling of virtual machines.

It's built on top of the KVM virtualiser and is designed for a commercial purpose: selling hosting in bulk, and exposing as many of qemu's useful features as possible to the customer through an open source command-line interface. BigV defines a flexible model of a virtual machine designed for typical hosting tasks, but is not tied to any abstractions or supervisor frameworks. The draw for the user is a close representation of a dedicated server, but with the flexibility of a virtual platform exposed through a consistent user interface.

For us (the hosting company), the aim was to allow us to purchase and manage servers that customers could reliably provision themselves. We had to be able to buy memory and storage separately and offer different grades of storage. It was a further design goal to use qemu's live migration feature so that we could swap out hardware for maintenance without disrupting our users. Finally we wanted to allow users to be able to install systems from CD-ROM, both locally and remotely.

This paper presents the technical design and some of the implementation lessons from BigV, with some lessons for anyone needing to implement something similar.

We start from the point of view that Linux has a buffet of virtual machine management options, ranging from those that are simple and work immediately, to those which need months of up-front planning and provisioning, i.e.

- don't virtualise at all (cheap, often the right choice);
- launch KVMs from /etc/inittab on a single server;
- (our existing Virtual Machine scripts - ugly, but working for 10 years and using two virtualisers[1]);
- Bytemark's pairvm (GPL) - uses DRBD and exactly two machines for high availability[2];
- libvirt - Redhat's "Xen insurance", general-purpose insulation from any one local virtualiser;

[1] `http://www.ukuug.org/events/winter2004/speakers+abstracts.shtml#id2717047`
[2] `http://projects.bytemark.co.uk/projects/pairvm`

- libcloud - Apache project doing the same thing for providers;
- Eucalyptus / OpenStack / Aeolus: layers, layers, layers for your own deployment.

BigV leans towards the end of this list. It recognises that it doesn't need multi-layered abstractions to achieve its aims, and this has been an important reason the to implementation has been quite quick. Virtual machine management systems need to be defined around a particular shape, and one supervisor won't be all things to all possible users.

So BigV is built around qemu, the network block device protocol, a custom internal messaging protocol, lots of Ruby conventions and an assumption of close control of the layer 2 and layer 3 network around it. That makes it quite complicated, but it's built to our company's sales model; we can still change anything that will help us fit that model better, rather than being dictated to by a supervisor framework.

Our first BigV installation (uk0.bigv.io) is running around 2TB of virtual machines, and is being stealthily launched.

2 Server and network organisation

There are three different types of physical server that make up a BigV cluster:

Heads — are full of RAM and as many CPU cores as we can buy, and host the Virtual Machines (qemu processes), as well as their consoles (serial port & VGA);

Tails — are full of hard drives, organised into pools by storage speed, and subdivided into logical Disks for presentation to the Virtual Machines over Network Block Device protocol;

Brains — keep the master list of virtual machines and present a public interface based on HTTP, JSON and a layered privilege system to control access. Cluster users talk directly to the brain through an open source command-line interface.

Tails and Heads are currently deployed in approximate a 1:4 ratio, and there are a fixed number of brains.

Our aim is that the user can create a virtual machine that has as much RAM as the largest Head in the cluster, and to attach up to 8 permanent discs to each VM, each of which can fill a storage pool. So a set of running VMs look like this, split between heads and tails:

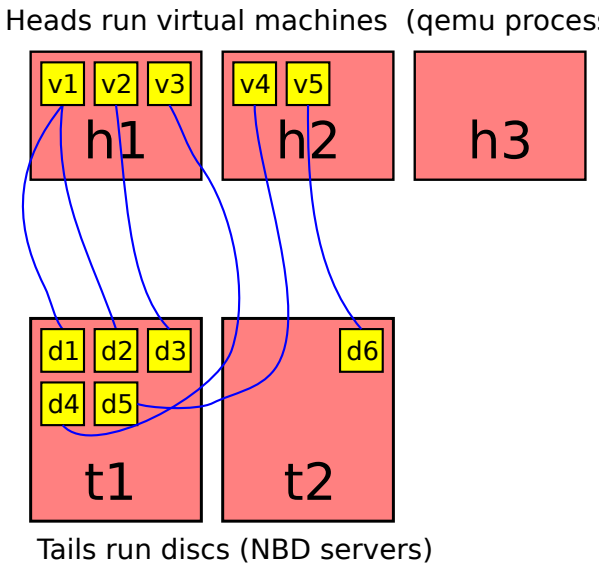

Figure 1: Relationship of Virtual Machines (v), Heads (h), Tails (t) and Discs (d)

There are multiple brain servers in the cluster, but only one is active at a time. They run a warm fail over protocol, that is to say, we require operator intervention to cause one brain to take over from another.

The Brain has control over the Heads and Tails by means of a "spine" protocol. The spine allows heads & tails to report what resources they have available (i.e. RAM and discs). They also receive orders from the brain telling them to create, update or destroy VMs or Discs. The heads and tails can also report changes as they happen, whether the brain initiated those changes or not.

When a server disconnects from the brain its resources become unavailable immediately - for Heads, the brain can work around this by restarting affected VMs on other Heads. If a tail server goes down, VMs have to wait for their discs.

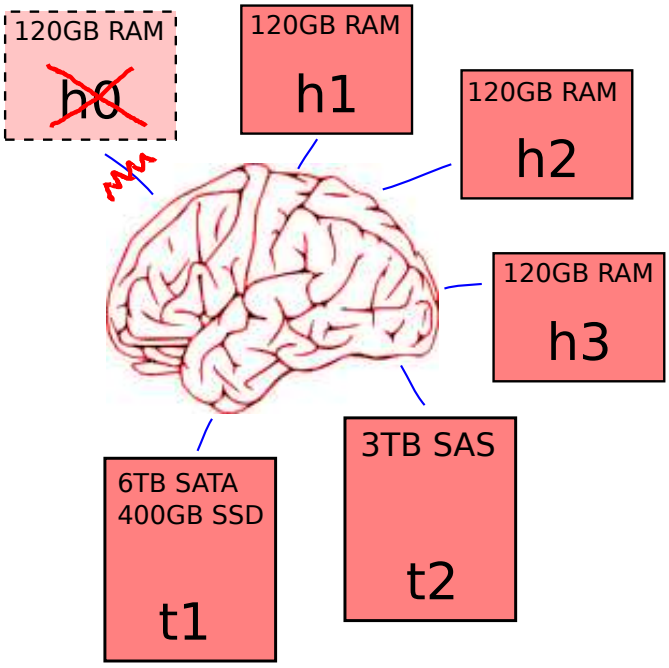

Figure 2: The brain connects directly to every head and tail in the cluster via the spine protocol. A disconnected server's resources become unavailable immediately.

Heads and Tails are connected together through a 10Gbps storage network; we rely completely on switching and predictable IPv6 addressing of every member to attach any Disk to any Virtual Machine. This LAN is also used for live migrations of Virtual Machines.

The heads also need one LAN for presenting "management" addresses (i.e. their consoles) and a set of public LANs for allocating IP addresses to the VMs they are running. So this leads to at least five LANs being necessary for a BigV cluster:

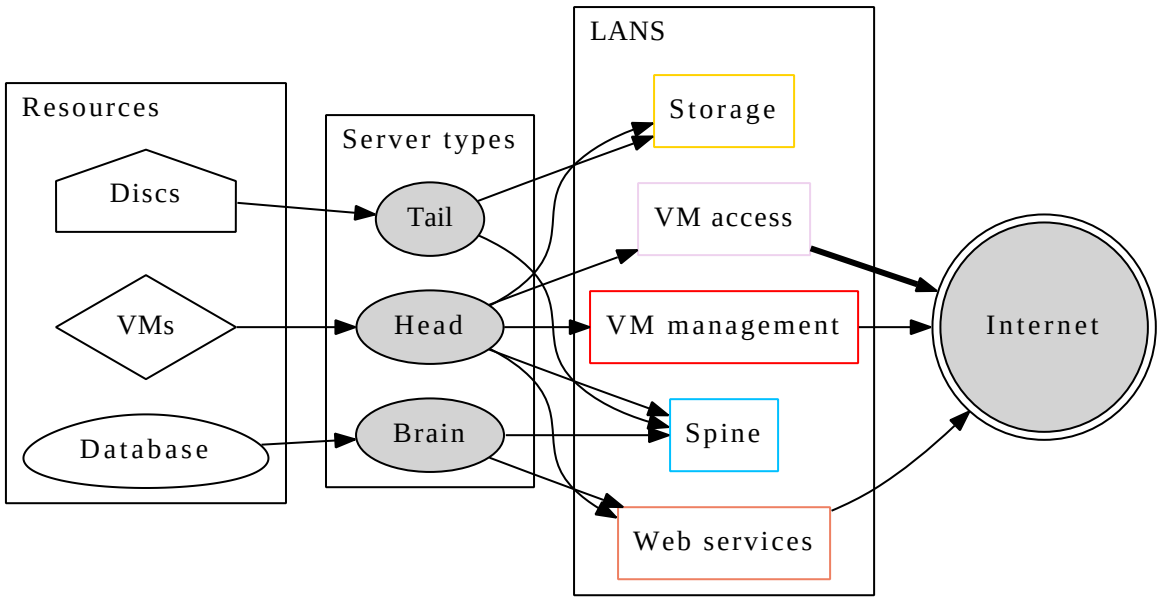

Figure 3: How resources map to server types, and the five types of network in a BigV cluster.

3 How the brain, database and spine interact

The brain stores the whole cluster state in a PostgreSQL database. The crucial information is:

- the list of every Virtual Machines in the cluster - everything from RAM size to more fleeting attributes like what virtual CD-ROM is inserted;
- which Virtual Machines live on which Heads (or are switched off);
- which Disks are connected to which Virtual Machines;
- which Disks live on which Storage Pools; and
- which Storage Pools live on which Tails.

There is a also a lot of data handling authentication, sorting of VMs into groups and accounts for access control. There is some online documentation on how this works[3] but it's not relevant to the way the virtualisation is arranged.

The database gets changed in response to any of the following:

- API interactions with customers typing at the command line (e.g. "bigv vm new ...")
- timed actions, e.g. deleted VMs get purged after 45 days;
- "spine" interactions with the heads and tails.

[3] `http://www.bigv.io/support/howto/privilege/`

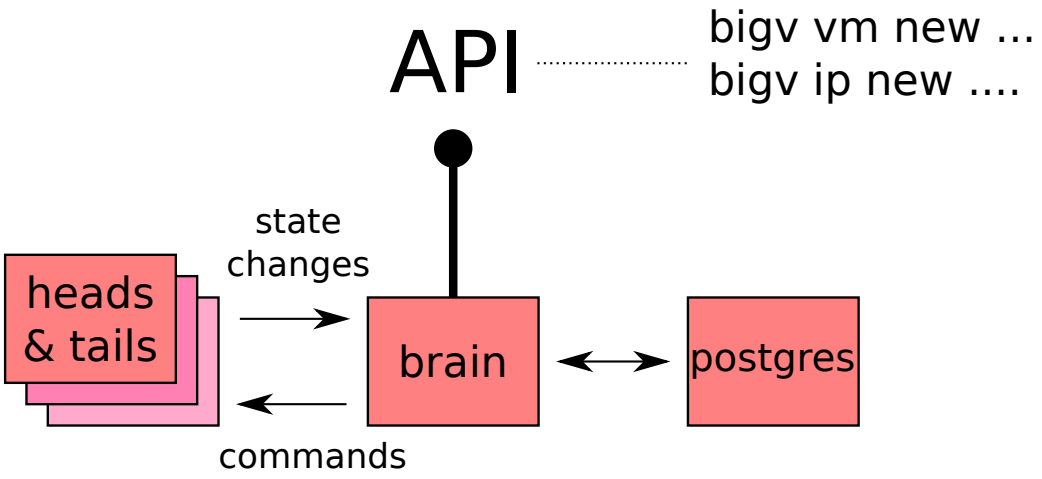

Figure 4: The brain connects directly to every head and tail in the cluster, as well as receiving commands from users.

Spine connections happen automatically, and are initiated by heads and tails on a separate trusted network (2). That means we can quickly add new servers when we need to expand. The brain uses established spine connections to effect the changes it has already written to the database, so e.g.

- when the user creates a new virtual machine, the brain writes a record to the database first. Then it notices on its spine, that there is an inconsistency between the state of the database, and the state of the spine - that VM doesn't exist. So the brain picks a head, and using the spine protocol, asks for the VM to be started there.
- if a VM crashes, the head will tell the brain via the spine that a VM has stopped. The brain will then notice an inconsistency, because that VM is meant to be switched on. So it will treat this exactly the same as if the user had just asked for it to be created, i.e. it will pick a head and "create" the VM process again.
- if a head reconnects to the brain after several minutes of down time, the head may start up by declaring that it has lots of VMs still running. If the brain has restarted those VMs on other heads, it will use the spine protocol to destroy the "rogue" VM instances.

4 The spine protocol

The spine protocol is one we invented ourselves. It is a little like IMAP because it's asynchronous, and supports command / responses based based on tags, as well as spontaneous notifications. It's a bit like BGP in that a set of resources are dumped, and tied to the presence of a TCP connection (i.e. if it breaks, they disappear). The protocol works as follows:

- A head or tail is hard wired to connect to the Brain's IP address on a well-known TCP port, and is "subservient", i.e. it does what it's told and the protocol is asymmetric. A head or tail can't tell the brain to do anything, only notify it of events.
- When the brain answers a TCP connection, the head or tail sends its first "notifications"; these will be:
  - a "hello" message, declaring an identity, so the brain knows when the same head connects twice (or again after a disconnection);
  - an "updated" message declaring that a "resource" has been created (the head or tail will initially send lots of these);
  - a "ready" message declaring that the head is ready for service.

The brain is expected to keep perfect track of every "created" object, and so the first thing a head or tail will do after issuing a "hello" is to dump all the resources it has as "created" messages, then issue a "ready" message. At that point the brain can start to consider those resources available for its use.

A resource is just a named key/value pair, e.g. here are a few common ones:

Name	Description
global.load	a three-element array describing system load
global.memory	the number of megabytes of free RAM for virtual machines [heads only]
global.storage_pools	a mapping of storage pools to free space [tails only]
vm.X	a description of a running virtual machine called X
disc.sata.Y	a description of a disc device numbered Y, which is of storage grade 'sata'

We map a series of names to "resources", which are represented as JSON hashes. Resources can be created, updated and destroyed at any time by the brain, and can do the same spontaneously.

The following is a complete example of a VM resource that might be stored on a host as "vm.swimmingpool". A VM resource maps closely to a KVM process - if it goes away, the VM is dead and it's down to the brain to recreate it. There's no state stored on the head, part of the brain's job is to trigger reboots when they are required by an unexpected machine exit.

This is what a VM resource looks like. In conjunction with the head's configuration, it maps directly to qemu command line arguments:

```
{ "memory"         : 1024,
  "cores"          : 2,
  "balloon_on"     : true,
  "cdrom_url"      : null,
  "imager_options" : null,
  "display_name"   : "swimmingpool.tardis.doctor",
  "keymap"         : "en-gb",

  "network_interfaces" : [
    { "mac"     : "11:22:33:44:55:66",
      "vlan_id" : 950,
      "model"   : "virtio",
      "ips"     : ["1.2.3.4", "2001:ba8:1af::1234"],
      "dhcp"    : { "network"    : "1.2.3.0/24",
                    "router"     : "1.2.3.1",
                    "hostname"   : "swimmingpool",
                    "domain"     : "tardis.doctor.vortex.uk0.bigv.io"
                  }
    },
    { "mac"     : "11:22:33:44:55:68",
      "vlan_id" : 10000,
      "model"   : "virtio",
      "ips"     : ["2.3.4.5", "2.3.4.6", "2.3.5.0/24"]
    }
  ],

  "discs" : [
    { "address" : "nbd:44.44.44.44:4777",
      "model"   : "virtio",
      "serial"  : "USER-SET-NAME" },

    { "address" : "nbd:44.44.44.45:4777",
      "model"   : "virtio" }
  ]

  "management_address" : "44.0.0.1",

  "management_ssh_keys" : [
    "-----BEGIN␣DSA␣PRIVATE␣KEY-----\n....␣etc.␣",
    "-----BEGIN␣RSA␣PRIVATE␣KEY-----\n....␣etc.␣",
  ]

  "console_users"      : ["thedoctor", "amy"],

  "user_authentication" : {

    "thedoctor" : {
      "authorized_keys" => "ssh-rsa␣AAAA␣....␣etc.␣...␣doctor@tardis\n",
      "password" => "26062010"
    },

    "amy" : { "password" => "R0ry" }
  }
}
```

Figure 5: An example of a spine resource named "vm.swimmingpool", representing a running virtual machine on a head.

Unlike the representation of the same data in our PostgreSQL database, this data is completely "unpacked" and stands alone as instructions to the head, without it needing access to the database. This results in some pretty verbose messages but it ensures the heads are stateless slaves, rather than needing to check back with the brain for essential information.

The resources on tails are much smaller and simpler - a complete disc definition only defines the type of storage, its size, and which heads may connect to it. Note that this prevents data corruption through only allowing the "correct" head to connect to a disc.

```
{
 "size" => 10000,
 "storage_pool" => "sata",
 "access_control_list" => ["44.43.0.1"],
 "address" => "44.44.44.45"
}
```

Figure 6: An example of a spine resource representing a disc.

The spine protocol only defines the startup options for both Virtual Machines and Discs. The brain can't directly read data from a disc or memory from a live VM, unless we chose at a later date to expose some other attributes through the resource definition. e.g. we can detect that a machine is running Windows, which requires licensing, by looking for known signatures of the running kernel in memory. That might be exposed as a new key in the VM's resource representation ("detected_kernel" ?), and the memory scan performed by the head. This is a crazy, largely unexplored idea - but makes the point that these resource abstractions are summaries, and might contain any information that fits our administrative model.

The final two operations on the spine are:

- *update* which allows the brain to change one or more attributes of a resource, e.g. changing *size* causes a disc device to be enlarged or shrunk, and changing *cdrom_url* causes a CD-ROM image to be presented onto a VM's emulated CD-ROM interface.
- *remove* allows the brain to stop a VM running on a particular head, or permanently delete a disc.

5 How discs are served

Tail servers present a large number of discs to the storage LAN, each disc running an NBD server on its own IPv6 address. A running qemu process is just given the IPv6 address of the right disc, and continually (re)connects. This means we can do live migration of storage in quite a robust fashion.

The tail server's discs are arranged like this then:

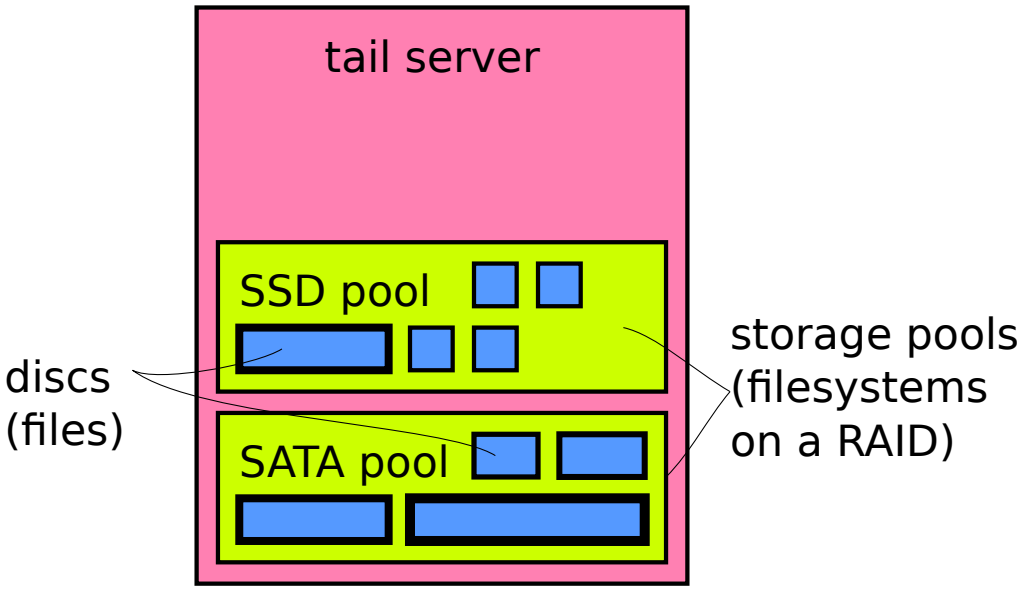

Figure 7: Tails host storage pools, storage pools host discs.

The tail hardware consists of conservative Linux systems, with a storage pool for each "grade" of storage - we currently provide pools based on SATA, SAS and SSD-based discs - RAID1 for the expensive SSDs and RAID10 for everything else.

The individual discs are represented as sparse files on ext3 filesystems. This allows immediate creation and temporary overcommitting of storage space (this is useful because we allocate about 2.5 times more space than is actually used). Given that we can do live migration of storage, even a few percent extra can be useful as space gets tight. The overcommit ratio is adjustable through an interface on the brain.

5.1 flexnbd

We rely on flexnbd[4] to serve discs to qemu.

NBD is a nice simple protocol[5] for serving block devices across a network, and this is how we present "hard discs" to our virtual machines. flexnbd is our own free server implementation. It has some special features:

- it can migrate its contents to another flexnbd server, and exchange IPs seamlessly so that the data can move from one host to another;
- it preserves "sparseness" when serving half-empty files, which avoids allocating space on the host where possible;

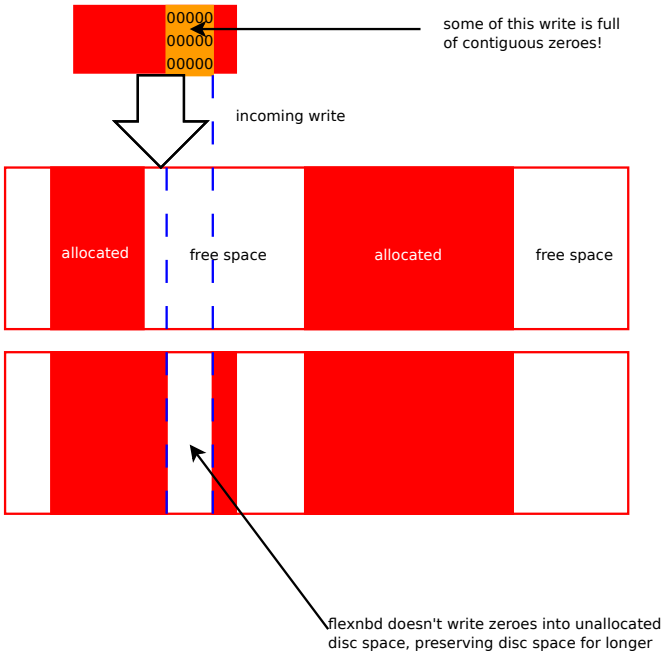

Figure 8: Preserving sparseness in a block file

- its access control list can be reconfigured without taking the server down, so the brain can isolate the "wrong" heads from connecting to the disc;
- it uses Linux's *mmap* & *splice* calls for faster data transfer.

So the tail server controls a group of flexnbd processes, starting and stopping them according to instructions from the brain.

6 How VMs are supervised

The heads present a list of resources to the brain, each resource representing a running VM as shown in full above4. Heads do not remember Virtual Machine information beyond a single run - if the qemu crashes, it is forgotten about, and it's the brain's job to restart it (a VM doesn't have to be restarted on the same Head of course).

Each VM resource corresponds to *three* running processes and a UNIX user id, created on the fly:

- the qemu-kvm process itself runs the virtual machine;

[4]https://projects.bytemark.co.uk/projects/flexnbd-c
[5]http://lkml.indiana.edu/hypermail/linux/kernel/0309.1/0152.html

- a custom DHCP server answers requests for this VM; and
- an SSH server provides:
  - the serial console,
  - the VGA console (via VNC), and
  - a reverse proxy for the VM to access a CD-ROM image on the user's computer.

The SSH server is a patched version of dropbear[6] with a new authentication mechanism, allowing us complete control over the authentication (more control than PAM offers). This allows us to present an SSH interface without it being tied to UNIX users, e.g. to allow multiple BigV users to a single virtual machine console. This hack should be merged back up stream in the next 3-4 weeks.

The network configuration on the heads is complicated because VMs can be attached to any VLAN on our network - we need code to build Linux bridges, VLAN interfaces and firewalls every time a VM comes up.

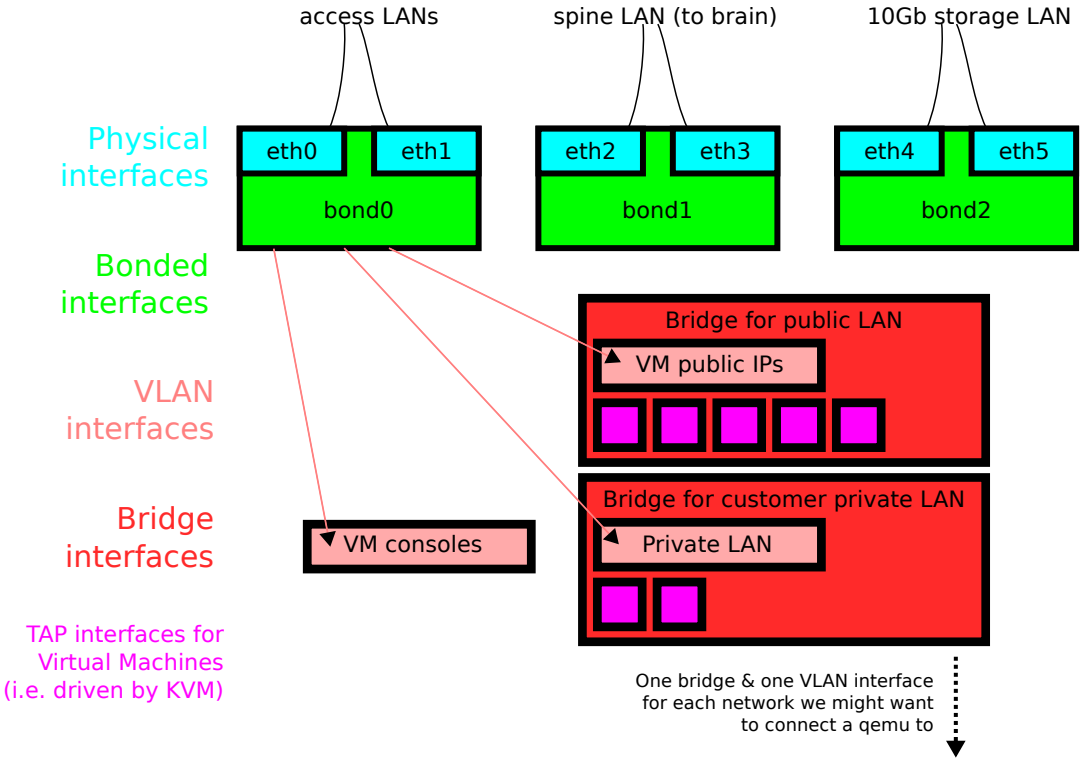

Figure 9: How network interfaces are arranged on a BigV "head" server (one box per ethernet interface)

The LANs for access networks, and their bridges, are created and deleted automatically, and rely on the head servers having trunk ports suitably configured at the switch.

We are very much looking forward to trying OpenVSwitch[7] which has just been merged into Linux 3.3 and should make this massive stack of interfaces less complicated.

7 On the brain

The brain runs a JRuby-based supervisor process. This presents a web service to the user, and backs on to a PostgreSQL database. It runs a special administrative telnet interface for performing

[6] `https://matt.ucc.asn.au/dropbear/dropbear.html`
[7] `http://openvswitch.org/`

surgery on the cluster, e.g. adding or marking down new heads and tails, initiating a live evacuation of a particular head or tail. However cluster administration functions are being pushed into the web service.

All the data about VMs is contained in the database, and the web service manages updates to this database. As development progresses, we are pushing more services out of the brain into separate processes. Currently, these are:

- bgpfeeder[8], which pushes routing information out to our core routers. e.g. when a user asks for a new IP address, the brain can make an instant allocation from the database, alter bgpfeeder's configuration file, and the new IP address(es) are working immediately, and can be configured on the VM. FIXME: draw diagram.

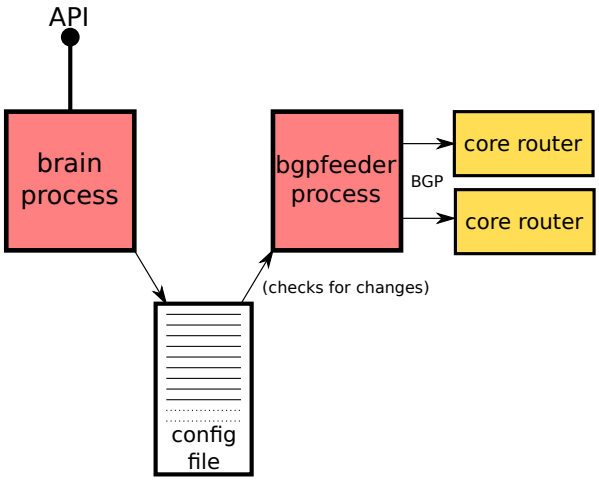

Figure 10: bgpfeeder runs at arms' length from the brain, feeding our core routers with routing data.

- PowerDNS[9], with a custom back-end, serves all the DNS for the uk0.bigv.io domain. This domain automatically reflects the state of the VMs in the cluster, so e.g. inkling.default.matthew.uk0.bigv.io points to a machine called 'inkling' in the 'default' group in the account named 'matthew'.

Both of these services are safe to run on a warm standby server, such that routing and DNS information isn't lost if the main brain process (or server) goes down.

8 Allocation policies and migration

We've gone through two allocation policies for VMs, and will continue to keep it as simple as possible.

The first is "fill the emptiest head" - i.e. resulting in a gradual and even use of all heads. The problem is that you almost immediately rule out the possibility that if your largest server has 120GiB RAM, and you have 15 of them, your whole cluster loses its ability to host a 120GiB VM after only 15 people have requested machines, however small those machines might be.

[8] `https://projects.bytemark.co.uk/projects/bgpfeeder`
[9] http://www.powerdns.com/

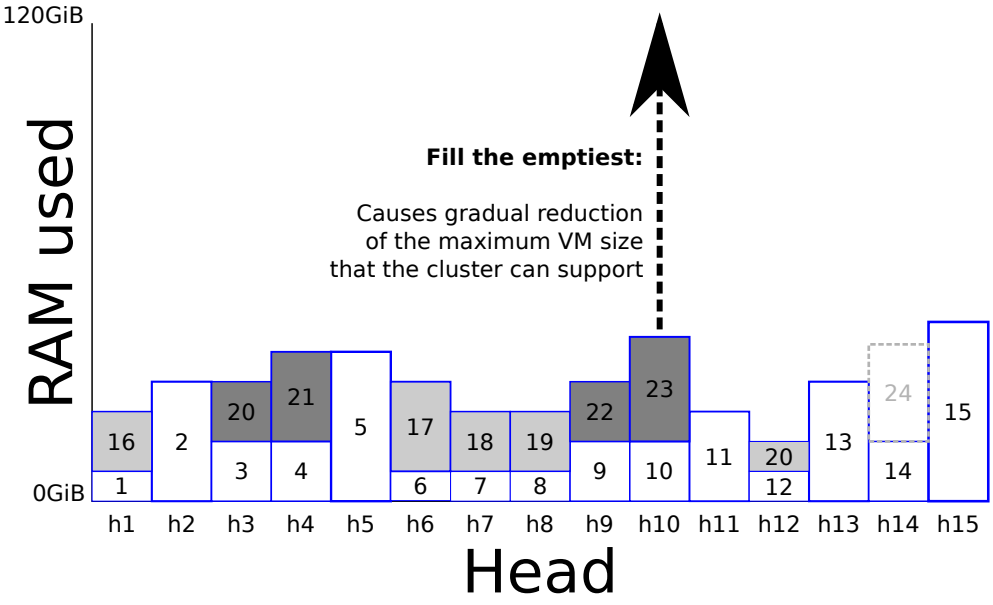

Figure 11: "Fill the emptiest" VM allocation policy

The current policy is "fill the fullest" - i.e. resulting in one host filling up, then another, then another. This means you are never short of space for an unexpectedly large VM until the cluster gets really full. The disadvantage is that VMs created within a short space of time are very likely to be on the same piece of hardware. So if all those VMs are created by the same user, and there is a failure of that piece of hardware, more of their machines will reboot unexpectedly. It has also the advantage of being more power-efficient, and is likely to be the one we pursue for that reason.

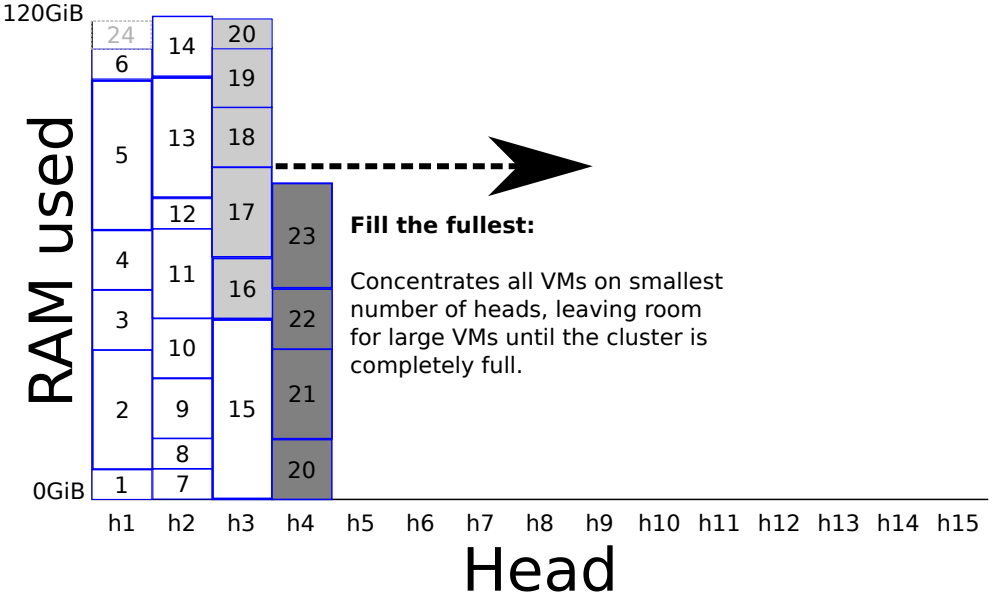

Figure 12: "Fill the fullest" VM allocation policy

For both policies we keep a number of head servers in reserve, just in case we can't restore a particular head, and need to start them all again elsewhere.

Tails have many of the same concerns, and are still running on "fill the emptiest". That's because we can overcommit and migrate them around we get into trouble, while gaining all the benefits of spreading Discs between as many storage pools as possible.

9 Thinking about uptime

It is quite hard to build a distributed system that can exceed a single server's usual up time, once you factor in operational mistakes caused by the extra complication.

This is one of our backup servers doing a very ordinary function, serving thousands of NFS and rsync shares:

```
-bash-3.00$ uptime
  2:15am  up 1240 day(s), 15:13,  1 user,  load average: 3.40, 3.43, 3.54
```

And this is one of our current generation virtual machines hosts:

```
$ ssh peri.bytemark.co.uk uptime
15:03:02 up 590 days, 23:46,  0 users,  load average: 2.73, 2.42, 1.84
```

A system built up of server-grade components running a polished OS and without overcommitting its RAM will run for *years* without intervention. In 10 years of writing virtualisation systems and running dedicated servers, it's hard to compete with the reliability of a single server without an enormous investment in development processes, operations staff and cutting-edge techniques. If you can run everything on a single server, or pair, with some warm/hot failover we think this is the best hosting solution.

So BigV is aiming high, for reliability *at the virtual machine level*, i.e. seeing VMs as analogous to dedicated servers, and not to be rebooted or disposed of lightly if that's the design the customer wants. This is really hard work, as a VM must tolerate:

- live migration working every few days, and between versions of the virtualiser;
- sharing very low-level resources and variable performance;
- potential compatibility problems with a virtual hardware platform;
- maintenance of network-attached disks.

None of these affect a dedicated server. KVM and Xen have gone through such frequent and major software revisions that 5 or 10-year up times are currently impossible to be sure of. The advantages of live migration are tantalising - a system that lives beyond the lifetime of a piece of hardware - but very few "clouds" actually use it.

We've made the following conservative design decisions to try to maximise up time of VMs while getting the advantages of live migration:

- aim for the software development to be "finished" in 2012, and further services (load balancing, firewalls, managed databases, or whatever) can be built on top of the core features;
- tails are just simple NBD servers running on server-grade hardware, on top of ext3 and hardware RAID. Other cloud storage failures have proved that "clever" block storage layers are too big a risk. Complete RAID server failures are extremely rare compared to over-clever replication and/or management systems that are too complicated for operations staff to understand;
- qemu processes try to reconnect indefinitely to a failed disc, meaning that any outage of a tail results in an I/O interruption, not a crash [arguably an I/O stall of more than a few minutes is as good as a crash, but in a lot of circumstances it's not, and helps us cover problems].
- we've so far held off automatic reprovisioning of VMs if a head falls over. We rely instead on a responsive ops team to make the decision between permanently retiring a dead server or reprovisioning. This closes off a predictable avenue for a cascading failure;
- upgrades of head software are handled one head at a time, and *require* that all VMs are migrated off the head before the supervisor can be restarted. This means less state, and that migrations get practised *a lot;*
- VMs will still run if the brain is dead, though no changes can be made, and reboots won't happen.

10 Development & deployment process

The BigV project is a "reboot" for Bytemark's development team and development processes. In order to make progress, we decided to build a completely self-contained system - our "legacy" systems did not get a look-in, even though we were re-implementing bits of them (IP allocation, account creation and so on). Developers get complete ownership of code, and a fresh start for a project that really deserved one.

This has meant we got results quite fast - the first code was committed in October 2010, and we were running VMs and engaging our first beta testers in May 2011. We have had the odd forced reboot during development, but it has now been 144 days since our last forced reboot.

Some development practices that have helped:

- a focus on fast results - we wanted to see a head, tail & command line running as quickly as possible.
- unit and functional tests - to make sure that individual components worked, even if they weren't complete.
- ambitious integration tests and one-shot deployment scripts from the start - so we can deploy a single host test system for manual tests, and our production system is built and distributed via normal Debian packages.
- a free software client has resulted in several important contributions from beta testers.
- stopping me contributing code after the first few months.

There are different risks to (re)deploying different parts of the code.

- the brain can of course shut down / delete VMs by accident, but it has to take positive action to do so! Just deploying a broken version won't necessarily cause problems to currently-running systems;
- the tails must reliably supervise NBD servers - if they don't, or can't restart properly, VMs' discs stop working. This is presented to the VM kernel as an IO "pause" rather than a failure, so they can recover transparently from an NBD server restart.
- the heads are the riskiest - if the supervisor software crashes with VMs under its supervision *it kills all the VMs* (or worse, for some bugs - it leaves them freewheeling). This was a deliberate decision to reduce the already-complex head daemon. This also has an interesting effect on the upgrade process for head software - it means we have to upgrade one server at a time like this:

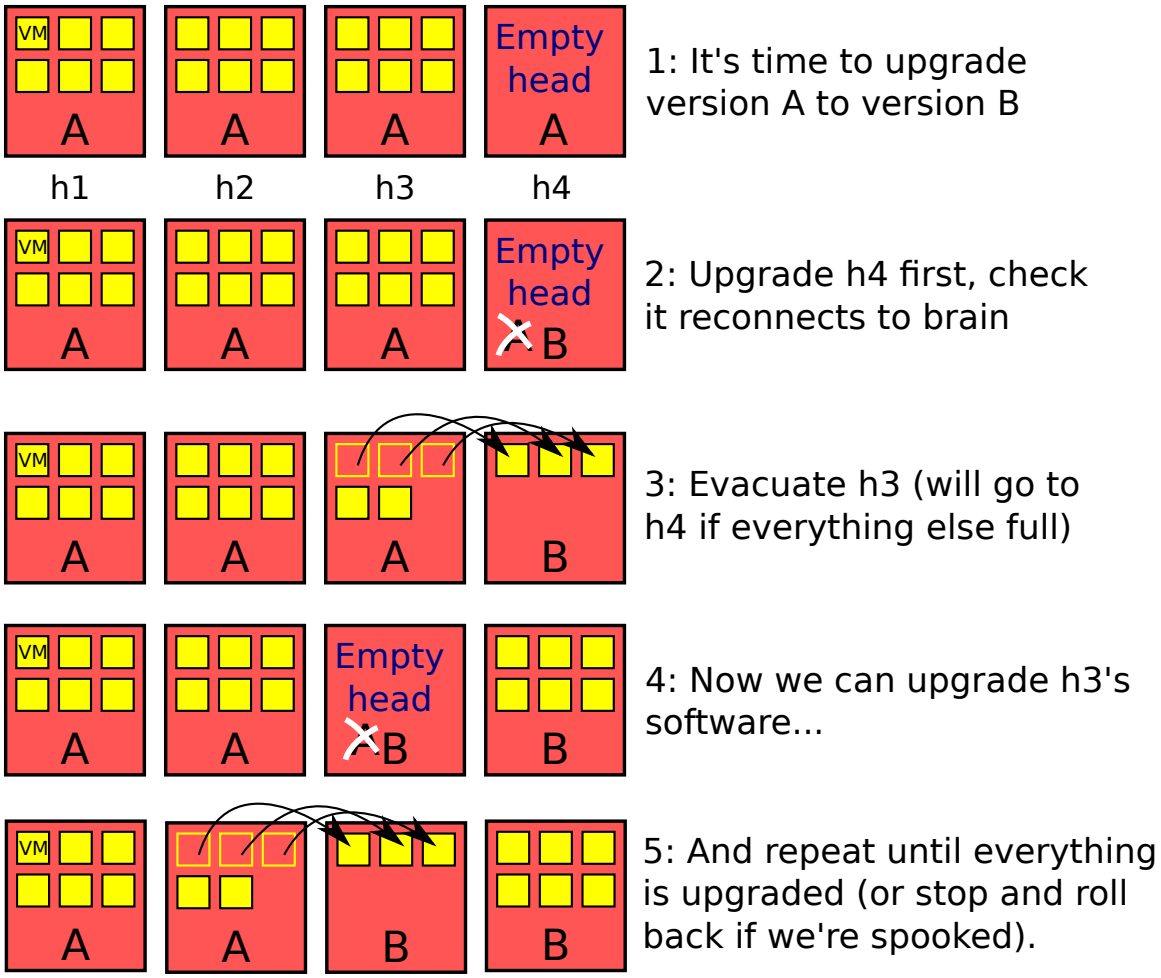

Figure 13: How we upgrade the software on heads

So we have to do an upgrade slowly, and gradually across the whole cluster. This work is currently manual, but we are streamlining the system administration gradually, and will are planning to move to a model where heads and tails are completely diskless, and are PXE-booted via their "Spine" LAN. That would allow the brain to initiate and manage software upgrades by simply rebooting a given server.

11 Criticisms

There are some of the obvious criticisms that can be levelled at the system as-is:

- We didn't want the software to be "too clever" - so it isn't. A failure of one tail server pauses many VMs until the server is brought back up. Our defence: we are quick and experienced at fixing broken servers. Distributed systems should be rightly terrifying.
- We invented our own cluster synchronisation (spine) protocol centred on a single host whereas e.g. Openstack uses a queuing system. How many TCP connections can one server process at a time? We'll find out before we move (and will have quite a lot of money to solve the problem too).
- The head servers run an SSH daemon for serving the console - when VMs are migrated, the sshd gets killed, and users connected to consoles will be kicked off. This could be made slicker either by 1) boxing up the console servers in another VM and migrating that too, or 2) centralising the console service.

- BigV is implemented using two separate Ruby implementations - MRI (for close kernel & system integration) and JRuby (for running a modern, multithreaded server efficiently). Each has advantages the other implementation can't provide, but it makes testing hard (or "thorough" depending on your perspective). JRuby packaging in Debian is not ideal - we use an old JRuby on squeeze, and it involves gems.
- The brain isn't very asynchronous yet - API calls take longer than they ought to (a big update is in the works to fix this).
- "head" and "tail" code still mostly written by me - probably needs quite a few more lines of code for easier testing. The head is a multithreaded monster. Having to migrate all the VMs around once for a software upgrade is weird but tests an important part of the infrastructure more often than we otherwise might. We have a new design to mitigate this but it's as good as a rewrite and will need a few careful months to write and deploy.

12 Coming soon

This paper was first presented in March 2012, and we made a to-do list. Some of it hs

- ~~Billing~~ - could be more granular, but connected up to our existing billing system reliably.
- ~~Disc migrations~~ - we built flexnbd to make the discs work exactly the way we wanted.
- Disc snapshots - are simple enough to implement clumsily - just use cp and wait. This is quite slow with ext3, but not useless. btrfs works better in this regard, but has terrible synchronous performance (which is crucial for reliable VM storage).
- Allocation and management of private VLANs - the back-end functionality works; we already integrate BigV with our various managed customers' private networks and dedicated servers. But we want new customers to be able to allocate their own from the command line - this needs a little more integration with our layer 3 network.
- More efficient management of heads and tails - right now we are ramping up our ability to buy and install servers (even 120GiB ones). This will likely outstrip our ability to manually install each one the old way. Our design document specified that the brain would manage PXE-booting of all head & tail systems, distributing the whole OS and joining them to the cluster automatically. Deploying new heads is currently a single command on top of a normal Debian installation, but as the cluster grows towards 10x its current size, fully automated deployment will be essential. Tail hardware is less homogeneous, so deployment of new tails is going to be harder to automate, involving careful RAID configuration, device naming and so on.
- A helpful follow-on feature will be power efficiencies from switching excess servers on and off automatically via SNMP.
- Option for disc-level replication to work around tail failures. Bytemark don't have much trust in complex disc layers, and we'd be reluctant to force one on our customers, especially one they're not aware of. Guest-side software RAID will work but software RAID over a multi-tenant shared network mount ought to degrade to the level of the slowest disc. The more discs you have, the slower it gets.