Cells: A Self-Hosting Virtual Infrastructure Service

Conference Paper · November 2012

DOI: 10.1109/UCC.2012.17

CITATIONS
4

READS
37

11 authors, including:

Alistair N. Coles
HP Inc.
11 PUBLICATIONS **170** CITATIONS
SEE PROFILE

Eric Deliot
13 PUBLICATIONS **82** CITATIONS
SEE PROFILE

Patrick Goldsack
HP Inc.
24 PUBLICATIONS **495** CITATIONS
SEE PROFILE

Julio Guijarro
Hewlett Packard Labs
9 PUBLICATIONS **230** CITATIONS
SEE PROFILE

Cells: a Self-hosting Virtual Infrastructure Service

Alistair Coles, Eric Deliot, Aled Edwards, Anna Fischer, Patrick Goldsack, Julio Guijarro, Rycharde Hawkes, Johannes Kirschnick, Steve Loughran, Paul Murray, Lawrence Wilcock

HP Laboratories
HPL-2012-158R1

Keyword(s):

cloud computing; infrastructure as a service

Abstract:

We describe the design and implementation of Cells, a novel multi-tenanted virtual infrastructure service. Cells has the unique property of being self-hosting: it operates its own management system within one of the tenant virtual infrastructures that it manages and therefore benefits from the same security, flexibility and scalability as other tenant services. Cells is also differentiated by its declarative interface for infrastructure configuration, and its fine-grained control of network and storage connections within and between tenant infrastructures.

External Posting Date: February 06, 2013 [Fulltext] Approved for External Publication
Internal Posting Date: February 06, 2013 [Fulltext]
Published in UCC 2012: 5th IEEE/ACM International Conference on Utility and Cloud Computing

Cells: a Self-hosting Virtual Infrastructure Service

Alistair Coles, Eric Deliot, Aled Edwards, Anna Fischer, Patrick Goldsack, Julio Guijarro, Rycharde Hawkes, Johannes Kirschnick, Steve Loughran, Paul Murray†, Lawrence Wilcock
Hewlett-Packard Laboratories
Bristol, United Kingdom
{firstname.lastname}@hp.com, †pmurray@hp.com

***Abstract*—We describe the design and implementation of *Cells*, a novel multi-tenanted virtual infrastructure service. *Cells* has the unique property of being self-hosting: it operates its own management system within one of the tenant virtual infrastructures that it manages and therefore benefits from the same security, flexibility and scalability as other tenant services. *Cells* is also differentiated by its declarative interface for infrastructure configuration, and its fine-grained control of network and storage connections within and between tenant infrastructures.**

Keywords-cloud computing; infrastructure as a service.

I. INTRODUCTION

The evolution of cloud computing in recent years has resulted in a fundamental shift in the provision and consumption of Information Technology (IT). Organizations and individuals that previously satisfied their IT needs by managing services on their own computing infrastructure are now adopting a model in which those needs are partly or wholly met by services available on the Internet. Consumers of cloud computing services benefit from rapid and scalable procurement of IT, while service providers realize economies of scale from their use of shared infrastructure.

The term "cloud computing" is applied to a broad range of Internet based services which have been loosely categorized according to the degree of abstraction they offer with respect to underlying physical infrastructure [1]: Software-as-a-Service (SaaS) describes the provision of IT applications such as email to end-users, typically via a web interface; Platform-as-a-Service (PaaS) providers such as Heroku [2] offer virtual runtime environments in which customers can execute their own application code; Infrastructure-as-a-Service (IaaS) enables customers to deploy low level virtualized infrastructure composed of virtual machines (VMs), virtual networks and volumes onto a shared physical infrastructure.

For an enterprise that is seeking to migrate legacy services to the cloud, IaaS presents the most expedient route; whereas new applications may be freshly coded using languages and interfaces prescribed by PaaS interfaces, or simply procured from a SaaS provider, existing applications are most easily virtualized at the infrastructure level. However, despite the success of IaaS offerings such as Amazon Web Services (AWS) [3], many organizations, in particular large enterprises and government agencies, are reluctant to entrust mission critical applications to a public service shared with and operated by third parties. Among their concerns are the need for reliability, perceived security threats that arise from sharing infrastructure, and regulations governing the location and privacy of data.

There is therefore also significant interest in so-called private clouds that are operated by enterprises for the benefit of their internal users. These require virtual infrastructure management systems that are easy to deploy, manage and interact with. Although privately operated, they must still be capable of isolating multiple tenant infrastructures.

This paper describes *Cells*, a multi-tenanted virtual infrastructure service that we believe is unique in being *self-hosting*: that is, the *Cells* management system is itself virtualized and is a tenant of its own service, operating within one of the virtual infrastructures that it is managing. As a consequence, *Cells*, like any of the virtual infrastructures that it hosts, is secure, easy to configure and deploy, can scale on-demand, is resilient to physical machine failures and can co-exist with other services sharing the same physical infrastructure, including other instances of *Cells*.

Cells has a number of other novel features:

- In contrast to the procedural interfaces of existing solutions that demand piecemeal construction of virtual infrastructure, *Cells* uses a declarative interface for one-stop specification of complete virtual infrastructures;
- *Cells* provides fine-grained controls over network and storage connectivity that enable robust security while permitting controlled connections between tenants;
- The implementation of *Cells* uses new, efficient and scalable network and storage virtualization technologies.

In the next section we introduce the architecture of *Cells*, including the declarative interface and security features, and then describe our implementation in Section III. Section IV expands on the self-hosting nature of *Cells*; failure handling is outlined in Section V. In section VI we discuss related work before identifying further topics of work in section VII.

II. *CELLS* ARCHITECTURE

Cells is a multi-tenanted virtual infrastructure service that presents each tenant with the illusion of its own completely private infrastructure. A cell is a virtual container that encapsulates all of the virtual resources associated with one virtual infrastructure, i.e. virtual machines, subnets and volumes, and acts as a secure management domain that is, by default, isolated from other cells and from the *Cells* management system (see Fig. 1).

A key feature of *Cells* is its use of a comprehensive, persistent declarative model as a management interface to the

Published in UCC 2012: 5th IEEE/ACM International Conference on Utility and Cloud Computing, November 2012, Chicago © IEEE

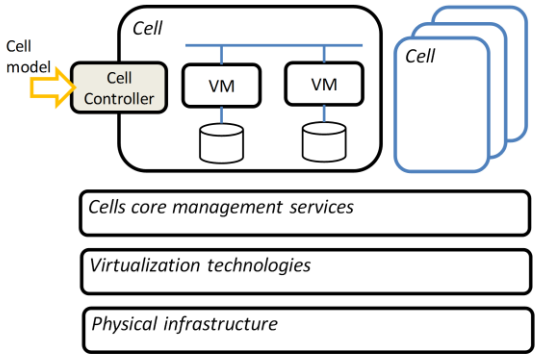

Figure 1. Overview of the *Cells* architecture.

entire contents of a cell. One powerful feature of this model is the ability to specify complex network topologies that mimic their physical counterparts, using fine-grained security rules to restrict traffic between subnets and individual VMs. This is important for enabling the migration of legacy systems to the cloud without extensive re-design or re-coding. Access controls may also be specified to govern connections from VMs to volumes.

The resources within a cell are realized by a set of core management services, with the assistance of novel network and storage virtualization technologies that enable *Cells* to operate on generic, shared physical infrastructure.

A. Cell model interface

Most existing virtual infrastructure services and software stacks provide a procedural interface for the creation of individual resources, but offer no intrinsic support for orchestrating the deployment of complex systems of interdependent resources. Clients must therefore implement their own deployment workflow and handle any failures that may occur during its execution. For example, a client will first need to ensure that storage volumes have been created before then deploying VM's that connect to those volumes; should any operation fail, the client may need to repeat that operation or rollback previous operations in order to reach a stable state. Furthermore, these procedural interfaces do not typically create a persistent model of the deployed system state, so clients must also monitor the health of deployed resources and implement corrective action when resources fail (which is to be expected in large scale systems).

For *Cells* we have taken an alternative approach: the interface to a Cell is via a persistent declarative model which encapsulates all of the cell's resources, their state and their inter-dependencies. A specification for the cell model is submitted in a single transaction to a Cell Controller (CC), which is an autonomous management component that instantiates and connects the specified resources in the appropriate order. Subsequent modifications to the cell model are made by submitting updated specifications (in whole or part) to the CC which infers any changes that are required to the existing cell resources.

The immediate advantage of this approach over a procedural interface is that the client is not burdened by the intricacies of workflow: the desired end-state of an entire system comprising many component resources may be declared "up-front" and the CC is then responsible for orchestrating the realization of those components on the client's behalf. The CC schedules component deployment based on dependencies expressed in the cell model, and retries deployment in the event of temporary resource starvation. When components fail the CC automatically replaces them, and because the cell model is persistent the deployed system can survive transient failures of the management system itself, and can be automatically recovered after a planned or unplanned outage of an entire *Cells* instance, without any further involvement of the client.

The declarative approach has a number of other benefits:

- Specification of the desired cell model is inherently idempotent: this greatly simplifies handling of errors that may occur when interacting with a remote service.
- A model can be checked for consistency, conformity and viability prior to any resources being committed. For example, an "all-or-none" policy is easily applied so that a system is deployed only if sufficient resources are available for all of its components.
- Declarative cell specifications may be re-used as templates for standardized system configurations; with appropriate language support, specification templates are easily extended and parameterized.
- Duplicate parameter entries can be eliminated by cross-referencing within a specification, which along with templating results in fewer configuration errors.
- Declarative virtual machine software configurations may be included within the cell specification, resulting in an integrated configuration of infrastructure and VMs.

A typical cell specification is shown in Fig. 2. Specifications may be expressed using either JavaScript Object Notation (JSON) or the SmartFrog configuration specification language [4], which has inherent support for overriding and extending configurations, making it ideally suited for specifying re-usable templates.

The cell specification schema includes several types of elements: *entities* define virtual resources such as VMs, volumes and subnets; *links* define associations between entities, for example links from a VM to a volume or a subnet; *permissions* define a cell's security policy, governing network visibility and access to volumes. For flexibility, cell specifications may have arbitrary structure, and the relationship between cell elements is declared explicitly by URL references rather than being implicit in the structure of the specification.

A volume element contains attributes that define the size and read/write access of the volume. A volume may also be defined as a copy of another volume by including a reference to that image volume. Volumes will often be referenced by one or more Volume Grant elements that specify which VMs or other cells are permitted to either create copies of the volume or directly connect to it.

Subnet elements specify the size of subnet required and whether its addresses should be internal (i.e. private to the cell) or external (i.e. globally routable).

Volume Connection links reference a Volume and a VM and specify the bus type and address at which the volume

```
params extends { memory 2048; }
mycell extends Cell {
  subnet1 extends Subnet {
    x.size 8;
    x.addressRange "internal";
  }
  vm1 extends VM {
    x.desiredState "on";
    x.restartOnFailure true;
    x.memory "<ref:/params/memory>";
    x.cpus 1;
    connection1 extends VolumeConnection {
      x.busType "ide";
      x.busNumber 1;
      x.busSlot 0;
      x.vm "<ref:..>";
      x.volume "<ref:/mycell/vols/bootvol>";
    }
  }
  vols extends {
    goldenImage extends Volume { x.size 8192; }
    bootvol extends VolumeCopy {
      x.image "<ref:../goldenImage>";
    }
  }
  grant extends VolumeGrant {
    x.volume  "<ref:/mycell/vols/goldenImage>";
    x.cell    "<ref://othercell/";
    x.grant   "copy";
  }
  interface extends VirtualInterface {
    x.vm "<ref:/mycell/vm1>";
    x.subnet "<ref:/mycell/subnet1>";
    x.vifName  "myHostName";
  }
  networkRule extends NetworkRule {
    x.address1 "<ref:../interface>";
    x.address2 "<ref:../subnet1>";
  }
}
```

Figure 2. *Example cell model specification in the SmartFrog language.*

should be attached. Virtual Interface links create VM network interfaces on a specified subnet, and optionally define a MAC address and DNS hostname entry. VMs may have multiple internal and external network interfaces. Both link types also specify dependencies between the VM and the link endpoint e.g. whether the Volume or Subnet must be ready before the VM can be booted.

VM elements specify required resources (e.g. memory), power state and failure handling policy (e.g. whether a failed VM should be automatically restarted). VM elements may include an attribute containing arbitrary structured data which is automatically presented to the running VM inside a mounted file system: we make extensive use of this attribute to configure and customize applications in VMs that are booted from an otherwise standard image.

Any cell specification may also contain a CC element that causes a new child cell to be deployed with its own CC, and also specifies an initial cell model that this child CC uses to populate the newly created child cell. All cells therefore descend from a root cell in a branching hierarchy. The boot volume and configuration of a CC is prescribed by the service provider but it is otherwise handled in the same way as a user-specified VM.

B. Security model

Cells provides a robust security model that addresses both the isolation of resources within the same cell and the isolation of cells from each other and external entities.

1) Intra-cell security

The virtual network infrastructure within a cell can be controlled using fine-grained security restrictions that are equivalent to those used in a physical infrastructure. For example, controls can be applied to network traffic between subnets and between individual virtual machines, as illustrated in Fig. 3. This is achieved by including Network Rule elements in the cell specification that declare permission for traffic to flow between a pair of endpoints. Network Rules are enforced by our network virtualization subsystem, described below, which applies ingress and egress controls to all VM network traffic. Crucially, prohibited traffic may be blocked at source so that a distributed denial of service attack does not result in a destination machine being flooded by unsolicited traffic.

Network Rules' endpoints may be subnets, individual VM network interfaces, specific ports on a VM interface or an IP address. A rule may also be used to permit traffic between a globally addressed interface and the internet. Network Rules are permissive rather than prohibitive, such that no packets are permitted to enter or leave a VM unless explicitly permitted. In contrast to a typical physical network infrastructure, two VMs attached to the same virtual subnet do not, by default, have network connectivity; the virtual subnet is merely a convenient administrative grouping. We can however configure *Cells* with a more relaxed default policy as required (e.g. default subnet connectivity).

The visibility of volumes is similarly protected by Volume Grant elements in a cell specification. These can be used to permit specific VMs to connect to a volume, or to permit a volume's contents to be copied to a new volume.

2) Inter-cell security

A critical feature of a multi-tenanted virtual infrastructure service is a strong guarantee of isolation from other tenants using the same infrastructure service. *Cells* therefore enforces default policies that prohibit any network traffic flowing between cells, and prevent any Volume being connected to a VM in another cell or being copied to a Volume in another cell.

There will, however, be occasions when services have legitimate reasons to communicate and share data with other services operating in different cells. Indeed this will become increasingly common as the cloud computing paradigm encourages an ecosystem in which higher value cloud services are delivered by composing other third party cloud services. Although inter-cell communication can be accomplished by routing packets via the Internet, this is inconvenient and inefficient, since it requires scarce public IP addresses to be provisioned in both participating cells. We therefore allow internal network and data connections to be opened between cells in a tightly controlled fashion as shown in Fig. 3. Crucially, any inter-cell connection requires the cooperation of both participating cells. For example, for a network connection to be opened between a VM in $Cell_x$ and a VM in $Cell_y$, both cells' specifications must include a Network Rule that permits traffic to be routed between their VM's. Similarly, in order to create a Volume in $Cell_x$ that is a copy of a Volume in $Cell_y$, a Volume Grant must be

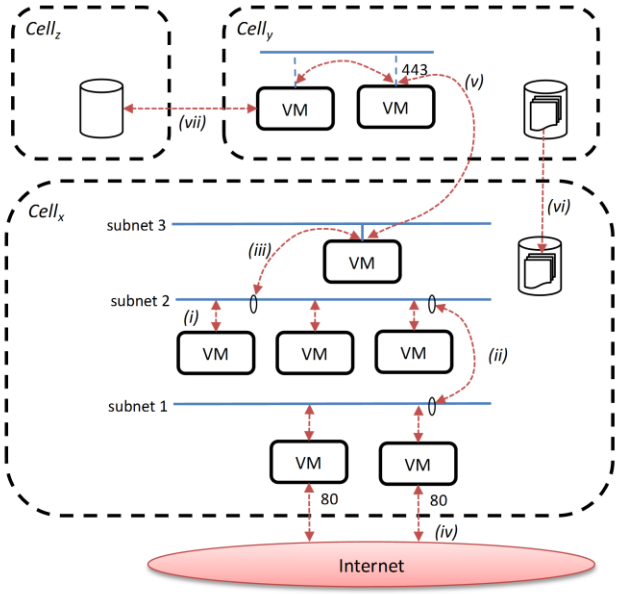

Figure 3. Examples of controlled connections (broken lines): (i) between all VMs on a subnet, (ii) between subnets, (iii) between a specific VM interface and a subnet, (iv) between port 80 on a VM interface and the internet, (v) between VMs in different cells on port 443 only, (vi) to copy a volume between cells, (vii) to attach a volume to a VM in another cell.

declared in the specification for $Cell_y$ that gives $Cell_x$ the right to make the copy.

To enable the declaration of inter-cell connections and permissions, a cell specification can refer to an element in another cell by including the other cell's DNS domain as the authority part of a reference URL. In the example above, a Network Rule in $Cell_x$ can refer to a VM interface in $Cell_y$ by using a URL of the form:

```
ref://y.cells.example.com/a/b/c/interface
```

In keeping with our rigorous policy of cell isolation, this URL will only be resolved if the element of the $Cell_y$ specification having path `a/b/c/interface` has been explicitly declared as being publically visible. One cell's specification cannot be arbitrarily inspected from another.

III. Cells Implementation

Cells is implemented as a loosely coupled distributed system of core management services. Each of these operates autonomously in its own VM (or may be scaled out across a number of VMs) with the exception of a Host Manager service that, uniquely, runs on each physical machine and interacts with storage, network and machine virtualization subsystems to instantiate VMs.

CCs present a REST [5] interface to which cell specifications are pushed using HTTP PUT requests. HTTP GET requests are used to retrieve a model of the cell's current state. To instantiate cell resources, the CC interacts with other core services including a Storage Manager, a VM Placer and an IP Address Manager.

All core services are controlled via a declarative model, following the same pattern as the cell management interface. Each service has a desired state model that CCs modify by applying state specifications, and the service then asynchronously attempts to achieve its desired state while publishing a view of its actual state to any peer that has registered an interest. In this way the system as a whole may continue to progress towards its target state in the face of failures of individual service instances; when instances do fail they are automatically re-started in a new VM.

The exchanges of state between services are facilitated by a State Distribution Service (SDS). This offers a number of modes for publishing state, such as point-to-point, broadcast and subscription based, all of which operate over a choice of transport layers including TCP and gossiping [6]. In each case SDS guarantees to eventually deliver state from a source to its destination(s). SDS has no central message hub; all communication is peer-to-peer, on a need-to-know basis, which results in a more scalable and reliable solution.

In contrast to a message bus, SDS also guarantees that when a state publication source is removed from the system then all of its published state is also removed i.e. all destinations to which the source's state has been published are notified that it is no longer valid. To do this it makes use of a core Presence service described later.

A. Cell model instantiation

The CC forwards Volume and Volume Grant elements to the Storage Manager service, which interacts with a storage virtualization subsystem to allocate physical storage. The Storage Manager also acts as an arbiter for connections to volumes, enforcing basic policies (e.g. single connection to a writable volume) as well as user specified Volume Grants (e.g. prohibiting connection to a volume from a VM in a different cell, or copying of a volume, unless appropriate permission has been granted).

Subnet elements are forwarded to an IP Address Manager which allocates subnet ranges from a global pool. For efficiency and scalability, the allocation of individual subnet addresses to VM interfaces is delegated to the CC. A DNS Manager service creates and resolve records for the IP address allocated to each interface. Hostnames are qualified by a domain name derived from a unique cell name, which is in turn qualified by the domain name of the parent cell.

Each VM element is forwarded to a core Placer which is responsible for determining the physical host on which the VM should be booted, based upon host properties and status registered by Host Managers. By default the Placer matches a VM to a host based on memory, CPU and speed requirements, but more complex placement policies can also be applied. For example, in one application of *Cells* we use extended VM attributes to guide placement of virtualized storage applications to a subset of hosts that have high performance SRIOV connections to physical storage [7].

Once a VM has been allocated a host, its specification, along with associated link specifications, is forwarded to the relevant Host Manager which then realizes the required virtual resources. VMs are created using a standard hypervisor supported by libvirt [8] (e.g. KVM [9]); virtual networks and volumes are created using our own virtualization subsystems.

B. Virtual networking subsystem

The *Cells* virtual networking subsystem is used to create the virtual subnets specified in cell specifications and to enforce the Network Rules described above.

Traditional virtual networking solutions typically rely on hardware network routing elements to convey packets between virtual subnets. Unfortunately, these do not scale to the degree required for *Cells*; for example, VLAN encapsulation [10] is limited to 4096 separate virtual subnets, whereas our goal is to support many tens of thousands of subnets. A hardware-based solution is also hampered by the lack of widely-adopted standard management interfaces for switches and routers, and the reticence of network administrators to allow *Cells* to dynamically reconfigure network hardware that might be shared with other services.

We considered eliminating hardware dependencies by using virtual routers in multi-interface VMs [11], but this approach requires a dedicated routing VM, potentially in every cell, and all packets travelling between subnets or to other cells would pass through at least one such routing VM, which can severely degrade network performance.

We have therefore implemented a novel virtual networking solution that we call Diverter [12]. This replaces routing VMs with a fully-distributed virtualized routing system that facilitates very efficient single-hop communication between network endpoints using only a software module on each physical host. Broadcast, directed broadcast and scoped multicast addresses are supported.

Diverter assumes a flat underlying layer-2 network and therefore requires no configuration of hardware switches or routers. Every virtual network interface is allocated a unique IP address, eliminating any need for Network Address Translation. When a packet is transmitted from a VM, the host Diverter module first checks the packet source IP address as an anti-spoofing precaution, and then checks that Network Rules permit the packet to be sent. Diverter then routes the packet directly to its destination by looking up the MAC address of the physical machine that is hosting the destination VM, and writing this over the packet's destination MAC address. On receiving the packet, the destination Diverter module reinstates the destination VM's MAC address and forwards the packet to the VM.

C. Virtual storage subsystem

The *Cells* storage subsystem (see Fig. 4) is capable of managing many tens of thousands of volumes, each of which is persistent and can be connected almost instantaneously to a VM running on any physical host machine. All volume data is stored on a relatively small number of physical volumes that are statically provisioned on Storage Area Network (SAN) attached disks. We use a virtualization layer to dynamically allocate logical volumes from these physical volumes. This allows us to create a much larger number of volumes than is typically supported by low cost physical storage appliances, and at the same time avoid dependency on proprietary management interfaces to those appliances.

Our storage subsystem makes fast copies of volumes by using a copy-on-write (COW) technique [13]: a copy is performed by creating a new empty logical volume which

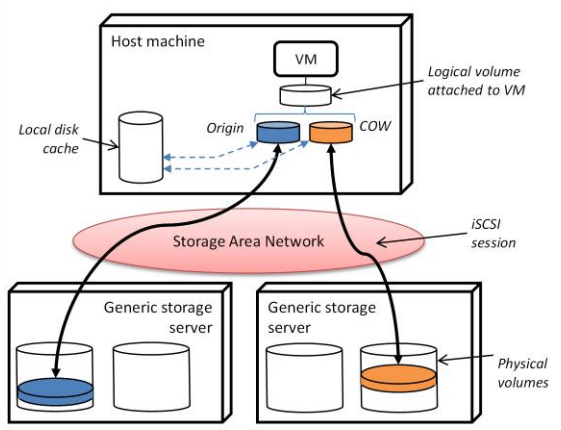

Figure 4. The *Cells* storage subsystem.

acts as a COW overlay for the origin volume; writes to the copy are redirected to the COW overlay, while reads of unmodified data are directed to the origin volume. A volume copy may in turn be copied by applying further COW overlays. The composition of COW overlay(s) and an origin into a single logical volume is implemented by a software module on the host machine on which a VM connects to the volume: this allows us to use generic storage servers and achieves greater flexibility and scalability, since COW overlays and associated origin volumes can be allocated from different physical volumes on different storage servers.

Once a COW overlay has been applied, an origin volume becomes immutable. We can therefore cache origin volume content on host machines with no need for a cache coherency protocol between hosts. This reduces SAN traffic and avoids hotspots in the physical storage, particularly when the origin volume is widely copied such as a popular VM boot image. A further advantage of our approach is that COW overlay volumes typically consume much less space than a full copy, since they only contain changes to the origin.

IV. Self-hosting Management System

A differentiating feature of *Cells* is that its core management services are themselves virtualized and encapsulated within the root cell. Apart from having permission to communicate with Host Managers on each physical host, the root cell is just like any other cell: all core services run in VMs and are capable of being deployed and scaled across the same shared physical infrastructure as tenant cells.

This results in great flexibility when deploying and managing *Cells*, since no physical hardware need be dedicated to the management system. Furthermore, the management system benefits from *Cells*' key features: it can be easily configured using a declarative model, it is automatically deployed by its own Cell Controller, with failed services being automatically re-deployed, and it can be secured using Network Rules.

A. Management System Bootstrap

In order for the management system to be self-hosting we do, of course, need to bootstrap the root cell, which in turn

requires the root CC VM to first be booted under the management of a Host Manager. (This is essential so that the Host Manager can subsequently monitor the health of the root CC.) We achieve this by initially starting a subset of the core services in Java Virtual Machines (JVMs) on a selected bootstrap host: the root CC, a bootstrap Placer that discovers the local Host Manager, and a bootstrap Storage Manager that is configured with pre-created boot volumes and data volumes for the core service VMs. Any persistent state created during bootstrap is stored in the data volumes which are temporarily mounted on the bootstrap host file system.

The JVM-hosted core services interact in the same way as described above for their VM-hosted equivalents. Bootstrap proceeds as follows: first, the CC deploys the bootstrap cell model, which initially contains a single VM configured to host the exact same subset of core services, including the root CC; second, before actually starting this bootstrap VM, the JVM-hosted service instances terminate and their data volumes are connected to the VM; third, the bootstrap VM is started; finally, the remaining core service VMs are deployed by pushing cell specification updates to the root CC, now operating in the VM.

The Placer delays placing further VMs until a quorum of Host Managers have registered. Together with a placement policy that attempts to place root cell VMs on mutually exclusive hosts, this ensures that core services are distributed across physical machines. The core service VMs include additional instances of the Placer and Storage Manager that are used by descendant CCs; further scale out may be achieved by configuring newly created descendant CCs to bind to specific service instances.

B. Management System Security

To protect the core services in the root cell against attacks from other malicious cells or external agents we take advantage of *Cells* own network isolation mechanisms and adopt a defense-in-depth approach (see Fig. 5). First, the core service network interfaces have private IP addresses and therefore cannot be reached directly from outside of the *Cells* private network. Second, the only VMs that can open network connections with the core services are system defined bastion VMs running a CC; packets cannot be routed from any other VM to a core service. Finally, a unique CC VM is provisioned for each cell and this has a private IP address that can only be reached from within its cell.

By dedicating a CC VM to each cell we not only isolate its network interface from other cells, but also isolate each cell's load on the management system in order to prevent denial of service attacks. For example, we can restrict the rate at which VMs may be started in a cell, and by using the CC VM as a DNS forwarder for the cell we can restrict the rate of DNS lookups from VMs in the cell.

Since the CC has a private network address, subsequent modifications to the initial cell model can only be submitted to the CC interface from a VM within the cell. This provides another layer of security. When a cell is to be managed interactively from outside, its initial model must therefore include at least one *gateway VM* that has both public and private network interfaces, as shown Fig. 5. In its simplest form this merely forwards cell specifications from a public interface to the CC, but gateway VMs may also present other external service abstractions as described below. Some cells may have no need for a gateway VM; for example, a cell may have an autonomous load manager VM that adds or removes VMs in response to varying load.

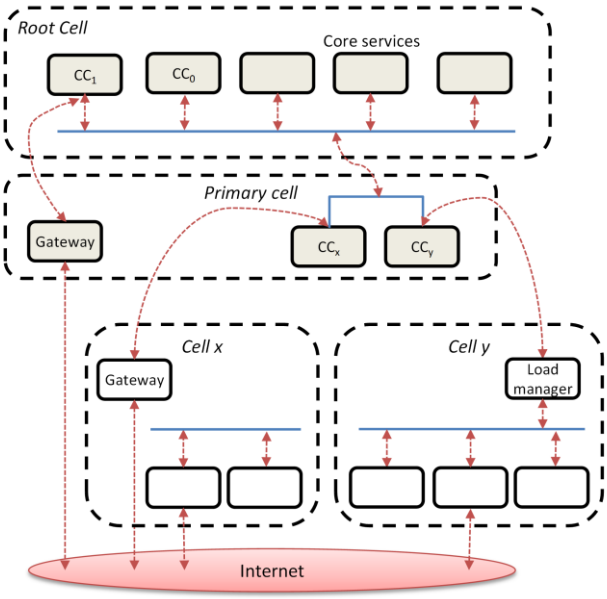

Figure 5. The hierarchy of cells descending from the root cell.

C. Cells Personalities

As a further security measure, the root cell does not include a gateway VM. This avoids having a public network address in the root cell. Instead, a primary cell is statically provisioned as a child of the root cell during the bootstrap process, and this contains a gateway via which further tenant cells may be created. This separation of concerns between the root cell and primary cell protects the integrity and stability of the root cell by minimizing modifications to its model, and allows a variety of primary cell implementations to be deployed, potentially simultaneously, without disturbing the root cell: we refer to these as *personalities*.

Our basic *Cells* personality uses a primary cell containing a simple forwarding gateway. Another personality enables *Cells* to be used as a node in the EU-funded BonFIRE project [14] which operates a multi-site cloud test-bed: for this the primary cell gateway translates OCCI [15] requests into updates to a cell model.

We have also implemented a much richer *Cells-as-a-Service* (*CaaS*) personality which provides a number of higher level services that are required to usefully present *Cells* as a virtual infrastructure service to multiple independent tenants. The first of these is a secure user management service for registration and authentication of tenants [16], who in turn may create and manage sub-tenants to whom they delegate rights to manage subsets of their cells. This hierarchical tenant model enables *CaaS* users to very easily deploy their own multi-tenanted services.

CaaS also provides a *Persistent Volume Manager* (PVM), which enables users to create, clone, modify, delete, and search for *Persistent Volumes* (PVs) in a central repository, and to securely share those volumes between cells and tenants. Unlike volumes created in tenant cells, which are lost when the cell is deleted, PVs transcend the lifecycle of tenant cells because they exist in the primary cell, which is never deleted. PVs are typically used to store "golden images" of operating systems, databases, and other software, or to harvest data from a cell before it is deleted.

Finally, *CaaS* exposes a web front-end and a REST API via which users may create and manage cells and PVs.

V. FAILURE HANDLING

Any *Cells* VM can be configured to be automatically restarted should it fail: failure is detected by a Host Manager and notified to the VM's CC which then re-deploys the VM. In the event of the root CC VM failing, the bootstrap Host Manager restarts it by repeating the bootstrap process.

VMs are monitored by periodically sending a liveness packet to a virtual network interface using a configurable port and protocol; a timely response to this packet verifies the VM's health. For security reasons the Host Manager cannot send packets to a VM interface, nor should its IP address be revealed to a VM. Liveness packets are therefore synthesized by the Diverter module and appear to originate from the virtual interface's subnet gateway.

A basic measure of health may be obtained by using the ICMP protocol to ping the VM. However, this does not necessarily give an indication of the health of applications running in the VM. For our core service VMs we use a UDP liveness packet which is handled by an application level ping responder that is closely coupled to the fate of the management service. Tenant VMs may similarly take advantage of the configurable liveness mechanism to achieve application level health monitoring.

If a Host Manager fails, for example due to a physical host failure, then a fall-back Presence service notifies CCs of associated VM failure. This is based on the SDS gossiping protocol: hosts report a suspected failure if a peer does not make a timely response to a gossiped message. If the number of failure reports exceeds a threshold the Presence service determines that the reported host has failed and notifies any party with a registered interest in that host, e.g. CCs with VMs deployed on the host. If the bootstrap host has failed then a new bootstrap Host Manager is elected. The reliability of the Presence service is achieved by deploying it as a self-monitoring group distributed across a number of machines.

Fig. 6 shows extracts from the operation trace of a *Cells* instance illustrating behaviour during deliberately induced failures. In each case a linearly increasing number of VMs are being deployed into two cells. In Fig. 6(a) the CC process (but not its VM) of one cell is terminated which causes VM creation to be delayed until application liveness monitoring detects the CC failure and a replacement CC VM is started. Note that no existing tenant VMs fail nor is the other cell's operation affected. In Fig. 6(b) a host failure causes some VMs in each cell to fail but these automatically re-start on other hosts within approximately 120 seconds.

VI. RELATED WORK

There are a number of freely available IaaS implementations, including Eucalyptus [17], OpenStack [18], CloudStack [19] and OpenNebula [20]. These all have procedural interfaces as opposed to *Cells* persistent declarative model. None of them encapsulates its own management system in a hosted virtual infrastructure as *Cells* does, and only CloudStack inherently supports automatic recovery of failed management services or hosted VMs.

All employ VLANs for layer 2 network isolation and support traffic filtering between VMs on a virtual network using optional firewall rules. In Eucalyptus and OpenStack inbound rules are defined in *security groups* that may be bound to multiple VMs; by default all traffic is permitted. CloudStack also uses security groups but can restrict both inbound and outbound traffic, with only outbound traffic allowed by default. For *Cells* we rejected the use of VLAN's due to limited scalability and a dependence on underlying hardware, and chose to prohibit network connections unless explicitly permitted by a rule; we can apply rules to both inbound and outbound traffic and can mimic security groups by cross-referencing rule templates in the cell specification.

AWS [3] is arguably the most well-known of the public cloud service offerings. In AWS Elastic Cloud Computing (EC2) all VMs of a tenant by default have network access to each other. As described above, a *security groups* feature can restrict inbound traffic, but outbound traffic is unrestricted.

EC2 does not allow an internal hostname or network address to be specified for a VM. As a result, distributed applications require post-deployment choreography. A static IP address can be rented and assigned to any VM instance, but traffic to that address is billed at a higher rate, meaning that this feature is primarily suited to a small number of external interfaces. *Cells* allows specification of hostnames for both internal and external interfaces; these hostnames can be used to configure applications prior to deployment. *Cells* also allows a MAC address to be specified when fixed addressing is required, e.g. to support legacy license servers.

The AWS Virtual Private Cloud (VPC) premium service offers greater control over tenant networks, specifically the ability to define subnets with predefined IP addresses, and network rules that control outbound as well as inbound traffic. Neither VPC nor EC2 allow broadcast or multicast traffic, both of which are supported in *Cells*.

The interface to EC2 is procedural, but the AWS CloudFormation tool enables declarative configuration of complex systems that are to be deployed onto EC2. As in *Cells*, JSON is used to specify standard or user-defined templates that may include cross-references. Whereas *Cells* references may follow arbitrary paths through structured data, CloudFormation is restricted to a one-level reference structure. CloudFormation can also choreograph system deployment, which addresses the problem of binding services and VMs together in the absence of pre-specified hostnames or IP addresses.

Other third-party tools and services have been developed to assist in the configuration and management of standard applications deployed on EC2, e.g. Apache Whirr [21], and

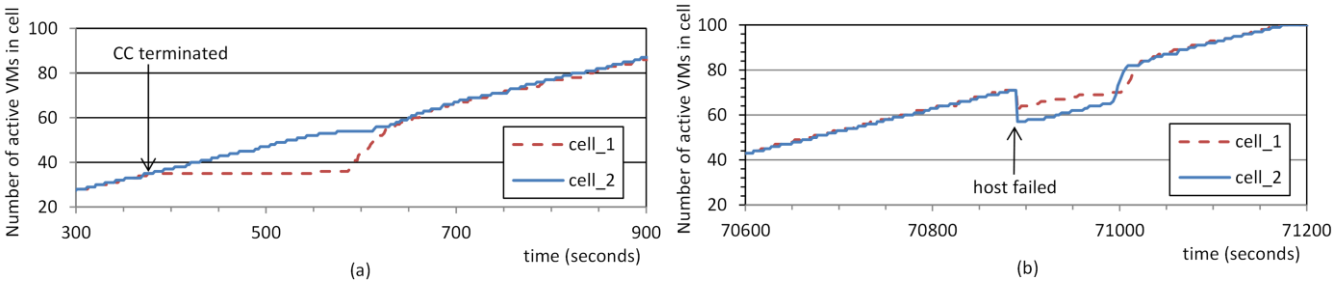

Figure 6. Extracts from a *Cells* operation trace showing behaviour during failure conditions: (a) a CC process is terminated; (b) a host is failed.

to offer higher-level management services such as monitoring the health of VMs and restarting in the event of failure, e.g. RightScale Cloud Management [22].

Microsoft's Azure [23] encapsulates VMs in a *Hosted Service*. Like *Cells*, Azure uses declarative specifications which describe VM roles, the number of instances of each role to be created, and any late-binding configuration parameters; cross-referencing within specifications is supported. Azure persists service specifications, and restarts VMs that fail due to a failure of the underlying physical infrastructure. Azure's network connections do not support broadcast or multicast traffic. In contrast to *Cells*, network connectivity is permitted by default and may be restricted using network rules, but these can only be applied to roles, not to individual VMs.

VII. Status and Future Work

Several independent instances of *Cells* have been operating for many months in support of a number of internal projects and as a node in the EU BonFIRE project. These instances co-exist on a single physical infrastructure that is also shared with other non-*Cells* services.

We have not yet implemented federation of multiple instances to achieve super-scaling and physical dispersal of *Cells*. A first step towards federation would be to enable composition of services hosted in different instances by enabling internal network connections and shared volumes between *Cells*. Beyond this, we would like to be able to disperse a cell across multiple federated instances.

The cell model could be extended to allow users to influence the relative placement of VMs on physical machines, for example to stipulate that two VMs should never be collocated. It may also be desirable for users to distinguish between parameters of a Cell specification that are mandatory versus those that are optional. For example, a particular VM may be specified with “must-have” and “nice-to-have” memory requirements – the latter being provisioned if available. In each case, syntax will be required to express multiple parameter values, or ranges, and to associate weights with each choice.

We would like to enhance *Cells* to take advantage of VM migration. This would enable more intelligent host usage, for example conserving energy by consolidating VMs onto a minimal set of hosts or pre-emptively retiring a host that may be prone to failure. We would also like to complement our strong connectivity isolation with improved performance isolation. We already reserve resources for the management system by using Linux cgroups, but could extend this to enforce quotas for network and processor resource consumption by individual cells and VMs.

References

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," National Institute of Standards and Technology, 800-145 2011.

[2] Heroku. http://www.heroku.com/

[3] Amazon Web Services (AWS). http://aws.amazon.com/

[4] P. Goldsack et al., "The SmartFrog configuration management framework," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 1, pp. 16-25, January 2009.

[5] R.T. Fielding, "Architectural styles and the design of network-based software architectures," University of California, Irvine, ISBN 0-599-87118-0, 2000.

[6] A.M. Kermarrec and M. van Steen, "Gossiping in distributed systems," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 5, pp. 2-7, October 2007.

[7] N. Edwards et al., "High-speed storage nodes for the cloud," in *4th IEEE/ACM International Conference on Utility and Cloud Computing*, Melbourne, 2011.

[8] libvirt: The virtualization API. http://libvirt.org/

[9] KVM. http://www.linux-kvm.org/

[10] IEEE Standard 802.1Q, "Virtual Bridged Local Area Networks," IEEE, ISBN 0-7381-3662-X, 2003.

[11] S. Cabuk, C.I. Dalton, A. Edwards, and A. Fischer, "A Comparative Study on Secure Network Virtualization," HP Labs, Technical Report HPL-2008-57, 2008.

[12] A. Edwards, A. Fischer, and A. Lain, "Diverter: A New Approach to Networking Within Virtualized Infrastructures," in *Workshop: Research on Enterprise Networking (WREN'09/SIGCOMM'09)*, Barcelona, 2009.

[13] A. Coles and A. Edwards, "Rapid Node Reallocation Between Virtual Clusters for Data Intensive Utility Computing," in *IEEE International Conference on Cluster Computing*, Barcelona, 2006.

[14] BonFIRE. http://www.bonfire-project.eu/

[15] Open Cloud Computing Interface. http://occi-wg.org/

[16] J.M. Alcaraz Calero, N. Edwards, J. Kirschnick, L. Wilcock, and M. Wray, "Towards a Multi-Tenancy Authorisation System for Cloud Services," *IEEE Security & Privacy*, pp. 48-55, November 2010.

[17] D. Nurmi et al., "The Eucalyptus Open-Source Cloud-Computing System," in *9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, Shanghai, 2009.

[18] OpenStack. (2011) OpenStack. http://www.openstack.org

[19] CloudStack. http://www.cloudstack.org

[20] J. Fontán, T. Vázquez, L. Gonzalez and R. S. Montero, "OpenNebula: The open source virtual machine manager for cluster computing," in *Open Source Grid and Cluster Software Conference*, San Francisco, 2008.

[21] Apache Whirr. http://whirr.apache.org/

[22] Rightscale Cloud Management. http://www.rightscale.com/

[23] Windows Azure. http://www.windowsazure.com/