Fractal: Automated Application Scaling

Masoud Koleini
University of Nottingham
masoud.koleini@nottingham.ac.uk

Carlos Oviedo
University of Nottingham
carlos.oviedo@nottingham.ac.uk

Derek McAuley
University of Nottingham
derek.mcauley@nottingham.ac.uk

Charalampos Rotsos
University of Lancaster
charalampos.rotsos@lancaster.ac.uk

Anil Madhavapeddy
University of Cambridge
anil.madhavapeddy@cl.cam.ac.uk

Thomas Gazagnaire
University of Cambridge
thomas.gazagnaire@cl.cam.ac.uk

Magnus Skejstad
University of Cambridge
magnus.skejstad@cl.cam.ac.uk

Richard Mortier
University of Cambridge
richard.mortier@cl.cam.ac.uk

ABSTRACT

To date, cloud applications have used datacenter resources through manual configuration and deployment of virtual machines and containers. Current trends see increasing use of microservices, where larger applications are split into many small containers, to be developed and deployed independently. However, even with the rise of the devops movement and orchestration facilities such as Kubernetes, there is a tendency to separate development from deployment. We present an exploration of a more extreme point on the devops spectrum: Fractal. Developers embed orchestration logic inside their application, fully automating the processes of scaling up and down. Providing a set of extensions to and an API over the Jitsu platform, we outline the design of Fractal, and describe the key features of its implementation: how an application is self-replicated, how replica lifecycles are managed, how failure recovery is handled, and how network traffic is transparently distributed between replicas. We present evaluation of a self-scaling website, and demonstrate that Fractal is both useful and feasible.

1 INTRODUCTION

Current datacenters provide the vast pools of computing resource on which today’s modern web infrastructure relies. Application deployment processes have evolved from individual hosts performing several intermingled roles, to use of virtual machines (VMs) and containers to allow hosts to support multiple cleanly separated and managed roles. To support these practices we have seen tools such as Puppet [36], Ansible [4] and Chef [9] developed to automate the process of building and configuring virtual machine images, and more recently, systems such as Autopilot [18], Mesos [27], Borg [37] and Kubernetes [7] to support automated management of resource use within a cluster of machines.

With improved support for orchestration of containers, many are now moving away from building monolithic web applications, to building applications as a set of loosely-coupled microservices: small, purpose-specific web applications. All these systems explicitly separate application logic (typically wrapped up in container images) from deployment configuration (expressed as “glue code”).

We explore an alternative approach, where both application logic and deployment configuration are written directly by the developer in the application’s code, based on her insight into how her application behaves. We provide a simple API that allows her to express the conditions under which replicas of the container instance should be created. This enables more specific application-layer behaviour to be used to manage scale-up and scale-down in a deployment. (§2)

We implement this for a particular type of container on Xen, MirageOS unikernels [24]. Jitsu [23] manages the lifecycle of replicas, and the Irmin [10] datastore manages state associated with each replica and, in particular, to handle merging state back when an application scales down by destroying replicas. The Open vSwitch instance that runs on every Xen deployment, hides all replicas behind the same IP address as the first instance of the service. Although our APIs are presented specifically for MirageOS unikernels, the extensions we provide for Jitsu are generic and would enable any container or VM to make use of Fractal (§3).

We evaluate by using Fractal to create a website that automatically scales itself, load-balancing across replicas appropriately. We show that, as load – defined in terms of requests/second – increases on an instance of this website, it is able to replicate itself to maintain performance targets. Subsequently, when load decreases, the website scales down by having unloaded replicas destroy themselves, merging their website logs back into those of the first instance. We demonstrate that the overheads, in terms of memory footprint, increased response times, and OpenFlow switch rules are negligible. (§4)

Load balancing and cluster resource management are long-standing areas of work, and many such systems have been developed and deployed. Most are constructed to run alongside applications, with policies concerning load metrics, thresholds and so on expressed in glue code or configuration files. In comparison, Fractal allows application developers to specify directly how load should be defined for their application, and how their application should behave under load. (§5)

We conclude with a brief discussion of factors that have yet to be addressed with Fractal. In summary, the contribution of this paper is to demonstrate the feasibility of building self-scaling applications. We do so by presenting the design and implementation of Fractal, a framework in which we build and evaluate a self-scaling website. (§6)

*Corresponding author. This work was originally a submission made in September 2015 to USENIX NSDI 2016.
2 DESIGN

The key distinction between Fractal and existing approaches is to place control of the load balancing mechanisms directly in the application developer’s hands. This enables creation of applications that adapt themselves dynamically, according to an application-specific measure of current demand and desired service. We assume that each application is hosted in a datacenter as a Xen [6] VM, and that the dom0 control domain for each physical Xen instance runs the Open vSwitch [34] softswitch to manage connectivity between local VMs. We augment this standard configuration by further assuming the VM in question is actually a MirageOS unikernel [24], and that the dom0 also runs the Jitsu [23] control stack; we discuss how non-MirageOS applications might use Fractal subsequently (§3.1).

Unikernels can be thought of as the combination of library operating systems with modern VMs. A unikernel VM image is created by compiling and linking application code directly with the various system libraries on which it depends. This results in compact, single-purpose VM images that boot without recourse to a traditional OS such as Linux, reducing attack surface and, in the case of MirageOS unikernels which are written in the OCaml language, leveraging modern programming language features such as type-safety and pervasive modularity [24].

It is particularly notable how compact MirageOS unikernels are in comparison to traditional Linux-based VM images: the complete on-disk image for a MirageOS webserver unikernel supporting TLS is just 2 MB, dramatically smaller than the equivalent using Apache or nginx on Linux. Jitsu is a control stack that takes advantage of this to provide extremely low-latency booting of VMs: around 30 ms for MirageOS unikernels running on Xen/x86 [23]. Fractal provides APIs that enable a VM to request Jitsu either to boot a replica of it or to destroy it, while supporting the target VM in retaining any state it contains.

Jitsu operates in a similar manner to XenStore [13]: it manages a hierarchical, transactional, key-value store. This store managed by Jitsu is an Irmin [10] repository, behaving similarly to a git repository, and is shared among VMs, each VM having an allocated subtree under the parent /jitsu. Jitsu then uses the Irmin watch mechanisms to cause events to be generated when a child of the /jitsu node is modified. Thus, by writing appropriately formatted data into their designated keys under /jitsu, a VM can trigger Jitsu to take action, creating a new replica of a VM, or self-destructing the requesting VM. Self-destruction of a VM takes place in two phases using a similar mechanism: (i) the VM indicates to Jitsu that it is halting causing Jitsu to update Open vSwitch rules to prevent any new network flows being direct to the halting VM; and (ii) when satisfactorily quiesced, the VM indicates that it should now be destroyed.

Applications that scale using Fractal are constrained to fit a particular application-layer topology. The first instance is booted and acts as the central node of a star. Subsequent replicas are booted following requests made to the Jitsu instance local to the requester. The detailed mechanics are described in §3.1, while § 3.4 discusses failure recovery both in the core and the perimeter of the star topology. Client authentication could be supported by, e.g., requiring client-side TLS certificates to connect to Jitsu; unfortunately, although the current release of Jitsu supports HTTPS connections using TLS, it does not yet support verification of client-side certificates. Fractal also relies on Irmin to enable stateful applications to manage per-replica state that must be retained after a particular replica is destroyed. Irmin provides efficient clone and merge operations, enabling git-like push and pull operations to be provided.

As a running example, we will use a simple static website that monitors the number of requests/second it receives, and scales up when a high threshold is exceeded and down when a low threshold is breached. This is an exceedingly simple example: this paper focuses on the overall system support required for such behaviour rather than the specific details of what metrics are best to use for a particular application – the essence of Fractal is that the developer has freedom to determine her own metrics rather than being limited to traditional metrics extracted from the OS running on the VM such as CPU load, memory usage, or ping latencies or other network metrics.

3 IMPLEMENTATION

We next detail three key elements of our implementation: the mechanics of how a running unikernel invokes Fractal (§3.1), how we manipulate the Open vSwitch instance to transparently deflect incoming traffic to the correct replica (§3.2), the subsequent lifecycle management of replicas and their state (§3.3), and finally, the recovery from replica and host failures (§3.4). The mechanisms we present make some assumptions about the datacenter environment in which they operate, but we do not believe these fundamentally limit Fractal’s applicability.

```
| line number | code |
|-------------|------|
| 1           | type error = [ 'Ok' | 'Error of error ] |
| 2           | (\* An invocation response. *) |
| 3           | type handler = response -> unit Lwt.t |
| 4           | (\* An invocation response handler. Consumes a response and returns a schedulable thread. *) |
| 5           | val replicate: handler -> unit Lwt.t |
| 6           | (\* [replicate h ()] asynchronously invokes Jitsu to boot a new copy of this unikernel, calling [h] with any response. *) |
| 7           | val halt: handler -> unit Lwt.t |
| 8           | (\* [halt h ()] asynchronously invokes Jitsu to stop this unikernel receiving new flows, calling [h] with any response. *) |
| 9           | val die: handler -> unit Lwt.t |
| 10          | (\* [die h ()] asynchronously invokes Jitsu to mark this unikernel as dead and ready to be garbage collected, calling [h] with any response. Note that [h] cannot be guaranteed to execute. *) |
```

Figure 2: Fractal API.

We assume that physical hosts all run XenServer, providing the Xen hypervisor and a dom0 management domain with the standard XAPI Xen management toolstack containing running instances of XenStore for storing VM configuration information and Open vSwitch, the OpenFlow softswitch used by Xen to route traffic arriving at the physical NIC to the appropriate virtual NICs in different VMs. Each dom0 also runs an Irmin-backed Jitsu instance. The Jitsu instance is configured with any datacenter policies concerning
resources available to clients, billing information and other administrative details. In the remainder of this paper we assume that each client has permission to access as many resources as are available.

Every XenServer instance has its own pool of MAC addresses it can allocate to VMs at creation time. The Jitsu instance on the physical host with the IP address published for the service name launches the first instance of the application unikernel as normal, in response to a DNS request for that name. It then writes the VM settings into the Irmin store under the subtree /jitsu/vms/NAME as shown in Figure 4. Jitsu allocates a unique identifier to each VM that it launches, and uses it as the key to the relevant subtree in Irmin and as the Group_table identifier for the Open vSwitch instance on every machine hosting the unikernel’s replicas (see §3.2 for details).

3.1 Replication Mechanics

The mechanics of API invocation are straightforward. Xen imposes the constraint that only dom0 can manage VMs, and so the basic mechanism for a unikernel to create and destroy replicas is by invoking the Fractal API (Figure 2) to write appropriate key-value pairs to the Jitsu in its dom0 to carry out the operation on its behalf via the HTTP(S) API for the Jitsu’s Irmin store. Jitsu carries out the request after any necessary authentication has taken place. Jitsu follows a Unix-like access permission scheme. Each entry belongs to a specific service and its replicas have exclusive read/write access permissions. Jitsu instances has read/write access permission to its local Irmin data and to entries on remote Irmin stores that belong to hosted replicas.

The metadata Jitsu maintains for each unikernel is shown in Figure 4. The first instance of each unikernel becomes the centre of a star topology, responsible for directing traffic to all replicas. This metadata includes, for all instances, the state, domain and application ids, what happens when the unikernel is stopped, the IP address of each assigned interface, and whether this is the first instance of that unikernel or not. The first instance also records the numerically lowest assigned MAC address for the unikernel, and any DNS records associated with that service. Replicas use their lowest assigned MAC address to identify the record, in place of the service name, and separately record the DNS name of the host on which the first instance is running. This allows the Jitsu responsible for booting a new replica to install the necessary Open vSwitch rules in the first instance’s dom0 to deflect traffic to the new replica appropriately.

```
let poll_halt = 10 (* period to check if to halt *)
let lo_rps = 100 (* reqs/sec; low threshold *)
let hi_rps = 1000 (* reqs/sec; high threshold *)
let current_rps () =
  if n < poll_halt then replicate replicate_handler >> return rps
  else
    if rps > hi_rps then
      replicate replicate_handler >> return rps
    else
      if rps < lo_rps then
        halt halt_handler ()
      else
        replicate replicate_handler >> return rps
        merge_state () =
          (* ...measure request per second serving... *)
          return ()
```

This metadata includes, for all instances, the state, domain and application ids, what happens when the unikernel is stopped, the IP address of each assigned interface, and whether this is the first instance of that unikernel or not. The first instance also records the numerically lowest assigned MAC address for the unikernel, and any DNS records associated with that service. Replicas use their lowest assigned MAC address to identify the record, in place of the service name, and separately record the DNS name of the host on which the first instance is running. This allows the Jitsu responsible for booting a new replica to install the necessary Open vSwitch rules in the first instance’s dom0 to deflect traffic to the new replica appropriately.

Figure 1: A Fractal topology for a specific application, showing the first instance, local replicas ① and ②, and remote replicas ③ and ④, with traffic deflected to the replicas via Open vSwitch rules.

![Figure 1](image1.png)

Figure 2: Simplified OCaml code extract from webserver used in §4 showing Fractal interactions. >>= and return are the monadic Lwt operators that MirageOS uses for lightweight threading.

![Figure 2](image2.png)

Each Jitsu’s Irmin store also records outstanding RPC invocations to create, halt, or destroy replicas. Jitsu uses Irmin’s watch_tag interface over /jitsu to monitor updates to the request keys whenever they are added or updated. In addition, every Jitsu instance has
a request and a response key on the Irmin stores of all other hosts in the datacenter. When a Jitsu instance runs, it starts watching its response keys on the other hosts using the watch_key interface. That helps in prompt notification of the Jitsu when it expects response from other Jitsu instances.

When a VM wishes to make an invocation, it writes the appropriate value under its /jitsu/requests/INSTANCE/request key. When the request is satisfied, the result code is written under /jitsu/requests/INSTANCE/response, which is watched by the requesting instance to determine whether the request is satisfied or not. Only one outstanding request is permitted, but the system remains responsive under load spikes due to the extremely short boot times achievable when using Jitsu to manage VMs [23].

Finally, although the API presented in Figure 2 is MirageOS specific, the basic functionality is actually available to any VM that wishes to exercise it. All a VM must do is write appropriate key-value pairs into its local Jitsu’s Irmin store. Implementation of the present API could easily be ported to other languages or shell scripts for use by any VM or container.

### 3.2 Managing Traffic

To load balance traffic requires consistent selection of a destination replica on a per-flow basis, with control over the proportion of traffic directed to each replica. We investigate two options for leveraging Open vSwitch to distribute traffic appropriately. The first makes use of a non-standard primitive provided by Open vSwitch with a two table forwarding pipeline. The first table contains exact match rules which forward traffic to a specific replica, plus a low priority default rule that uses the bundle_load command to hash the packet header, store an output port in a packet register, and forward all unmatched packets to the second table. The second table then contains a single learn command, which uses the outcome of the hashing function to install an exact match flow rule in the first table and forward the packet appropriately.

The second uses the standard OpenFlow Group_table abstraction, introduced in OpenFlow v1.1, which aggregates forwarding operations in a Group_table entry, with each group having a type and identifying an ordered list of buckets, sets of actions (i.e., packet modifications and forwarding) to carry out. This approach uses a single SELECT Group_table entry for each replicated service; which selects at random a bucket and applies its actions on the packet. In Open vSwitch, each set of actions in an entry can have a weight associated with it, used to influence the hashing of new flows between available group buckets.

Traffic for each application is then load-balanced among replicas using a single Group_table entry installed in the Open vSwitch running in the dom0 of the original instance. Creation of a replica causes a bucket addition to the group entry that appropriately rewrites header fields (primarily IP and Ethernet destination addresses) and forwards traffic to the switch port to which the destination replica is attached. When a replica is removed, the weight of the corresponding bucket is set to zero initially so that no new flows are directed to the dying replica and the bucket entry is finally removed when all outstanding connections are completed.
As an example, Figure 6 depicts the OpenFlow flow entries used to load balance traffic between a web application’s first-instance, and two replicas, one local and one remote. Incoming traffic is direct to the Group_table entry on the first host (➀) while outgoing traffic is transmitted (➄). The Group_table selects one of three buckets uniformly randomly for each flow, and directs traffic to the relevant unikernel, after rewriting destination IP and MAC addresses (➀). The situation is a little more complex for a remote replica.

To avoid the client receiving traffic from unknown sources when their requests are forwarded to a replica, we must be able to forward packets to a remote replica in compliance with datacenter policy while providing client IP and port details so that the remote replica can respond directly. We use the GRE tunnelling protocol [15] support in Open vSwitch, configuring GRE tunnels between Open vSwitch instances on the original and remote replica hosts through the OVSDB protocol [33]. Each GRE tunnel appears to the Open vSwitch on the first instance’s host as a port to which we forward incoming traffic via appropriate flow entries. To avoid interactions between multiple remote replicas on the same host, we create a unique tunnel for each replica with GRE keys. Our approach is not limited to the GRE tunnelling protocol and could accommodate any other tunneling protocol that can integrate with an OpenFlow switch. With respect to Figure 6, rule ➄ forwards traffic to the GRE tunnel, and flow ➋ then forwards load balanced traffic to the remote replica.

A similar problem occurs for the outgoing traffic from remote replicas. One option would be to force all response traffic flow back via the original instance’s Open vSwitch so it could rewrite IP source address appropriately. However, this would severely limit scaling and increase cross-talk between different VMs by imposing considerable packet forwarding load on the network link and the Open vSwitch of the original instance. Instead, we introduce controlled spoofing of source addresses by installing, in the Open vSwitch on the replica’s host, a flow that matches outgoing response packets by port number and source IP address and rewrites the source IP address to the service IP address before forwarding the packet to the NIC of the host (➋ in Figure 6).

During development we observed that the hashing function was automatically rebalanced during updates to the Group_table table, resulting in delivery of packets of already established flows to the wrong replica. We remedy this using the flow caching policy of Open vSwitch. This maintains an in-kernel flow table, inserting an equivalent exact-match flow in the kernel module when a packet is matched against a select Group_table entry. We modified the ovs-vswitch daemon to not remove this exact-match rule when the hash function is updated, ensuring that ongoing flows are correctly delivered during Group_table updates. The in-kernel cache is sized to support millions of entries, so eviction due to cache pressure is unlikely, though a proper fix would require further minor modifications to the Open vSwitch code.

Comparing the two approaches, the second requires a minor modification to the Open vSwitch implementation, but is standards compliant, potentially allowing offload of traffic deflection rules from the Open vSwitch softswitch to hardware OpenFlow switches. The first, in contrast, does not require modification of the Open vSwitch source code, but is not fully standards compliant, limiting potential future use of OpenFlow hardware switches, and also does not allow dynamic adaptation of the hashing function to the traffic load.

Furthermore, when we evaluate the impact of each deflection policy implementation on our testbed (§4), using the topology in Figure 6 to load balance a static web service, we observe that the second approach exhibits marginally better load distribution across replicas and lower (albeit by just 0.2 ms) mean response latency. Using httperf we generate a traffic load of 2000 page requests/second for 10 minutes, and measure the instantaneous CPU utilisation of the host’s dom0 and each unikernel, as well as the response latency for each page. We configure the two network policies described above: the first uses an OpenFlowGroup_table entry (OF Group), and the second uses the Open vSwitch extension discussed above.

Figure 6: Fractal network configuration when load-balancing a web application among the first instance and two replicas. Flows are evenly distributed, after address translation, among the three instances (➀, ➄). Traffic for the remote replica is forwarded through a GRE tunnel from the first host (➋, ➄). Outgoing traffic from the second host has its source IP address rewritten to match the first host’s (➋, ➋).
while finer grained traffic management policies are applied in the
Table 1: Comparison between OpenFlow policies when load-
degning when a replica needs to be created or destroyed, and
dies (OVS Extension). Table 1 presents the mean (μ) and standard de-
the network core of large datacenter focuses primarily on routing
and fine-grained traffic management policies are applied in the
dom0 and the top-of-rack switches in the network. It might raise
security concerns though. However, when a replica attempts to
generate malicious traffic, it can only use the IP of the first in-
stance of the service it provides, making such attacks easily detectable
and auditable. If a replica tries to spoof traffic to an arbitrary IP
address, the Open vSwitch on the destination host will drop it due
to lack of a suitable flow. Any further network management access
control policies can be easily integrated by inserting appropriate
higher-priority OpenFlow flows into the flow table. For example,
port-based firewall policies can easily integrate with our policy by
installing port-matching flows in the Flow_table with a higher
priority.

### 3.3 Replica Lifecycle Management

Managing an application’s use of datacenter resources requires
taking decisions about when and where replicas should be created
and destroyed, incorporating details about current load on datacen-
ter components, notably servers, and network switches and links.
Fractal makes these decisions collaboratively, with the application
determining when a replica needs to be created or destroyed, and
delegating details of where new replicas should be created to Jitsu.
In turn, the datacenter operator implements policies plus any nec-
essary monitoring infrastructure (e.g., load or topology monitors,
or contractual details for specific customers) for Jitsu to apply to
determining where replicas should be created. We do not address how
to optimally place replicas in this paper, but believe that techniques
used in adaptive resource management, e.g., [19], will be worth
exploring.

The precise metrics that an application developer will wish to
use depend on the application. The essence of Fractal is to place this
decision in the hands of the developer of the specific application in
question, and so we do not make any recommendations here as to
which metrics should be used. The simple website example we use
in our evaluation (§4) uses requests/second as its metric, with two
thresholds, (low, high) = (100, 1000), determining when a unikernel
decides to request creation of a new replica, or to request that itself
be destroyed.

Figure 4 depicts the metadata maintained by Jitsu for each VM.
The first instance of an application VM is allocated a unique name,
passed on to all replicas, and must know its unique name to write its
requests under the correct key. We use the lexicographically lowest
MAC address of each VM’s interfaces for this purpose. Figure 5
shows the Irmin state involved for local and remote invocations of
replicate. The remote request must additionally supply a pathname to
the Xen image to boot (assumed to be available to the remote host,
e.g., via a network file system). Upon receiving the request, Jitsu boots
the unikernel accordingly, and configures Open vSwitch using the
newly booted unikernel’s domain identifier, IP address and MAC
address to address the network traffic between all instances.

For the new VM to be created on a different physical host from
which the request originated, Jitsu first decides upon the target host
according to whatever policies the datacenter operator has in place,
and then writes the request into the target’s Irmin store. Jitsu can
distinguish a remote request from a local request by matching the
requester’s name against the names of running VMs and the list
of hosts. It then defines and creates a new replica, setting the key
/jitsu/vm/hostname/init_xindex to be the name of the requester
(left blank if requester is local). The response is written back to the
requesting host’s Irmin. Requests are not relayed.

When a replica decides that it is no longer required, it calls halt.
In response, the local Jitsu reconfigures the first-instance Open
vSwitch to stop sending traffic to this instance. When the replica is
satisfied it has finished servicing all outstanding requests, and has
pushed any necessary state back to the first-instance, it calls die
which sets the TTL in its Jitsu record to zero. Jitsu has a monitoring
thread that stops VMs whose TTL has reached zero, and so the
replica will be shortly removed (within 5 s by default). The Open
vSwitch on the remote machine must also be reconfigured if the
replica was requested by a remote VM, indicated via the init_xindex
key’s value. Finally, Jitsu deletes all the Irmin keys belonging to the
deleted instance.

### 3.4 Failure Recovery

Failures are common in the datacenter and this section presents
the strategies of Fractal to provide continuous service delivery. In
our failure analysis we consider two failure types: replica and host
failures.

**Replica failure recovery:** In order to detect a replica failure, Frac-
tal exploits the XenStore service, which stores realtime configuration
and state information for each running VM. Effectively, during a
VM failure XenStore removes all associated information from the
database. Fractal monitors the XenStore entries and upon the re-
omoval of a replica VM information, it performs a garbage collection
routine similar to a die call. A replica reboot operation can equally
lead to system inconsistencies, since Xen changes the VM DomId
and the interfaces exposed to the dom0. Fractal can detect DomId
changes for a replica, by comparing its local state with the Xen-
Store state, triggering appropriate updates in the output action, IP
Fractal: Automated Application Scaling

and MAC address translation of the respective Open vSwitch table flows.

**Xen host failure recovery**: Fractal employs a heartbeat mechanism over the Irmin database to detect host failures. The Irmin database on the host of a service first instance stores a timestamp entry for each replica host. On a fixed interval, the Fractal instance on a replica host is responsible to update the respective entry with a timestamp, thus providing a heartbeat mechanism which allows host failure detection. On a host failure, Fractal removes any bucket forwarding traffic to an unavailable host.

To maintain service liveness during a failure of a first instance host we exploit the Xen High Availability (Xen HA) [1] functionality. Xen HA enables the creation of resource pools by federating control of multiple Xen hosts, and provides automated host failure detection and VM migration. In a Xen HA pool, a host is designated as the master host, responsible for the management of VMs running on the pooled hosts. A unique instance of Fractal runs on the master host of the pool, managing service replication between pooled hosts, while an Irmin instance run on each host of the HA pool maintaining strong synchronisation with the master Irmin instance. Upon a non-master host failure, Fractal monitors any replica migration and updates accordingly the Open vSwitch configuration of the live hosts of the pool. On a master host failure, Xen HA elects a new master host, which in turn starts an instance of Fractal triggered by an event hook in the Xen hypervisor. The new master Fractal instance will reset the networking state of the pooled hosts, reboot existing first instances and destroy all replicas. Replicated services can then use the Fractal API to trigger the required scale-up operations in order to meet service delivery requirements.

4 **EVALUATION**

We evaluate the overheads of Fractal using a cluster of 4 machines with single socket 6-core Intel Xeon 2.4GHz processors and 64GB RAM to represent a rack in a datacenter. Three of the machines act as hosts in the datacenter, running XenServer 6.5; the remaining one is our traffic generator, running httperf on Ubuntu 14.04. The prototypical scalable service our unikernel provides is a webserver serving static webpages of sizes drawn from a distribution of total transfer size for pages using statistics for September 2015 reported by HTTP Archive tracker [17]. We created static pages sized to the mid-point of the size bucket in those data, and fed httperf with a seed file of 100 URLs to traverse, with URLs recurring in proportion to the frequency in the reported statistics. We use the XAPI web interface to extract network and CPU utilisation metrics.

**Is running multiple replicas on a single host beneficial?**

The first question we address is the most basic: does our replication architecture improves the aggregate service rate. In this scenario we consider replication in a single Xen host. For each experimental run, we initialise statically Fractal to setup a specific replication degree and measure the CPU utilisation of each unikernel for varying traffic loads.

Figure 7 presents the CPU utilisation per unikernel for a specific traffic rate. We limit our analysis to 4 replicas due to the limited number of CPU cores (We assign two CPU core to the dom0). For each replication degree we run 10 experiments for each traffic rate and report in Figure 7 using errorbars the mean CPU utilisation and two standard deviations from the mean. From the results we highlight that local service replication can achieve linear scalability. A non replicated service can achieve a throughput of 104 MBps, but the addition of a single replica can almost double the achievable rate (203 MBps), while the load is distributed evenly between the two replicas. However, for this configuration, it seems that once there are four instances (3 replicas and the first instance) running locally (the original instance plus three replicas), there is little benefit in creating more replicas locally. The balance between local and remote replication should clearly needs to be managed.

**How does performance compare between local and remote scaling?**

The second question we address is what is the impact on HTTP response times when a replica runs on a remote or on a local host. Figure 8 reports the mean and two standard deviation from the mean of the HTTP response latencies measured using the httperf tool. For this measurement we employ a single web page of size 110 kB and we vary the request rate. We measure the page download latency using four replication strategies: *no replica*, which uses only the first instance of the service, *local replica*, which uses a replica colocated with the first instance, *remote replica*, which uses a replica located on a different host from the first instance, and *local and remote replicas*. **Figure 7**: Average CPU utilisation per unikernel as traffic increases linearly, for different numbers of local replicas.

**Figure 8**: Client latency experienced with no, local, and remote replicas.
and remote replicas, which uses a replica colocated with the first instance and a replica running on a remote host. Figure 8 presents the average page response latency for different traffic loads, with error bars indicating representing two standard errors from the mean. From the results we highlight that the no replica strategy can provide low and predictable response latencies for traffic rates up to 100 MBps, incurring significantly inflated latencies for higher data rates. The addition of a replica achieves lower latencies up to 217 MBps, as suggested by Figure 7. The remote replication strategy incurs a slight latency inflation, which can be explained by the required additional hop in the network path to reach the remote replica. Finally, the local and remote replicas strategy highlights the benefit in response time for HTTP requests, when mixing local and remote replicas.

Is load distributed evenly over multiple local and remote replicas?

Given the observed difference in the experienced latency between local and remote replication strategies, the next question we explore is whether this minor latency inflation has an impacts on the load distribution between replicas. For this experiment we setup Fractal to use 4 unikernels, the first instance and 3 replicas, and we differentiate the number of local and remote replicas. Figure 9 presents, using a stacked bar plot, the average CPU utilisation for each unikernel under constant (high) load. We limit our analysis to 4 unikernels, because of limitations in the number of cores in the Xen host. From the results, we highlight that the aggregate CPU utilisation has small difference between the different replica strategies. We further inspected the number of packets send to each unikernel, as well as, the number of interrupts delivered to each vif and we did not find significant differences. We concluded that this marginal difference between CPU loads is observed when a high number of unikernel is hosted on a single Xen host and it is attributed to scheduling effects.

What is the impact on client response time as the service scales up and down?

Our evaluation shows that clients are benefited during high load circumstances as depicted on Figure 10. This experiment consisted of sending and incrementing the traffic by 400 reqs/sec every 20 seconds using httperf, and unikernels are configured to trigger an scale up event once they hit a threshold value of 1000 reqs/sec rate. The results shows how the utility of the automated system for scaling helps alleviating the latency perceived by clients through the self-replication process. Our experiments have also shown that there is no overhead in terms of the latency perceived by the client when a unikernel that uses a self-replication mechanism but does not replicate over a unikernel that is not able to self-replicate.

5 RELATED WORK

A wide range of load-balancing services are available in commercial public cloud infrastructures. For example, Amazon AWS provides Elastic Load Balancing (ELB) [3], a network-based load balancing service. ELB enables cloud applications to expose multiple service replicas as a single public IP address, and dynamically load balance flows between replicated VMs. Flows are distributed via round-robin for generic TCP flows and last-connection routing for HTTP and HTTPS sessions. The load balancing service uses the Amazon VPC, a generic packet switching service used to interconnect VMs using private networks. While similar in general approach, Fractal hides the complexity of configuring the VPC by producing OpenFlow rules automatically, and through use of Open vSwitch is deployable in any datacenter running Xen.

Another example on a public cloud is Azure’s Ananta [32], a datacenter load-balancing service for the Azure datacenter. Ananta splits the load balancing function in two operations: flow distribution across service replicas and SNAT translation. Flow distribution for incoming traffic is implemented on standalone software switches, MUX, which map each flow to a specific replica. MUX nodes integrate with the datacenter network policy through the routing protocol by advertising each service’s IP reachability, and achieve scale-out capabilities via ECMP routing. SNAT functionality for outgoing traffic is implemented by software agents in the dom0 of virtualised hosts. Unlike Ananta, Fractal reduces the per-packet processing by integrating the per-flow mapping operation with the forwarding decision on the edge of the datacenter network, where the traffic load is lower.

Duet [12], an extension of Ananta’s design, offloads some of the replica assignment processing to hardware switches, thus improving flow latency and scalability. Duet exploits the line-rate per-packet hashing for ECMP mapping and IP-in-IP tunneling capabilities of hardware switches, which remain underutilised in modern datacenters. Nonetheless, the respective lookup tables are limited and cannot store global load balancing state. Duet implements a routing policy which localise traffic of a load balanced IP in a single switch and defines a distribution algorithm that maximise service coverage. Software switches are used for fail-over redundancy and traffic processing when the mapping state for a load balanced IP does not fit in the hardware switch lookup tables.

Application controlled management has been explored in the context of virtualized environments. The Potemkin honeypot farm [38] was a highly available honeypot farm that could spawn new honeypot VMs during an attack and was an early attempt to define an application API for VM management. The authors developed a set of extensions in the Click [29] router that triggered the creation of dedicated honeypots and forwarded traffic to it when an attack was positively detected; during idle periods, those VMs would
be garbage collected. Another example is SnowFlock [21] which provides a Xen-specific pthread-inspired fork API enabling rapid parallel VM migration using network multicast for the distribution of the OS state. While the API is superficially similar, the authors primarily focused on the optimisation of the migration process and did not discuss integration with the complex network policies used in modern multi-tenant datacenters.

Datacenter research has also explored mechanisms to automate cloud application replication. AGILE [28], a notable service replication automation framework, uses wavelets analysis to predict application resource requirements in order to fulfill specific service delivery guarantees and proactively optimize replication strategies. AGILE define a generic black-box performance modeling framework, which uses monitoring agents to collect low-level information, like CPU, memory and network utilization for each VM and predicts future resource requirements. Unlike AGILE, Fractal enables applications to define and run application-centric scaling mechanisms with greater agility.

Several commercial providers offer products with hardware-accelerated load balancing features, e.g., A10networks [2], Brocade [8] and Baracuda [5], though these have high deployment costs and require considerable effort to integrate them with datacenter network policies. Widespread adoption of IaaS services has even motivated development of dedicated load balancing cloud appliances. For example, the VA R20 virtual appliance [22] is a VM image which provides flexible layer-4 to layer-7 load balancing between replicated VMs. Similarly, HAProxy [16] is a popular open-source application used to load balance cloud applications using dedicated Linux VMs. Such cloud appliances make deployment straightforward for self-managed load-balancing services, but exhibit limited scale up capabilities and inflate per-packet latency and resource utilisation, as each client request must traverse at least two VMs. Cloud management platforms, such as the OpenStack project [31], also offer load balancing service management capabilities. OpenStack has developed a generalizable load-balancing control abstraction through the Neutron LBaaS project [30], and provides integration with a wide range of software and hardware platforms. Similarly, the Kubernetes [7] container management platform provides a framework to control the service replication strategy for a service. Specifically, it employs per-cluster replication controllers which execute application-centric health checks, like HTTP status code monitoring, network connectivity, and command executability within the container, in order to maintain a constant replication degree. The Kubernetes replication controller provides a composable primitive which allows scaling and scheduling automation from external controllers through a simple API. For example, Kubernetes provides a beta horizontal autoscaler functionality which allows a user to define a target average CPU utilization during service deployment and the autoscaler will ensure to meet this target by appropriately controlling the replication degree of the service. Fractal proposes an application-centric orchestration, which integrates the application and the orchestration logic.

Load balancing was one of the early application use cases for programmable network control and a number of early network control architectures [14, 39] have explored reactive and proactive OpenFlow-based schemes. These architectures, primarily explored the applicability of control plane programmability to achieve load-balancing functionality, but did not explore in-depth a holistic SDN solution. Furthermore, the introduction of the SDN paradigm has motivated the design of novel load balancing control abstractions. Mekky et al. [26] proposed an enhancement in the OpenFlow protocol for application-specific extensions in the packet processing pipeline, which enables integration of arbitrary packet processing applications in the data plane. Pane [11] is a control architecture which enables control delegation to end-user applications. The framework provides a generalisable mechanism to synthesise individual policies in a single network configuration, while policy conflicts are resolved by direct negotiation with the requesting applications. Fractal exposes a control abstraction focused on load balancing service control, while policy conflicts can be resolved...
by appropriate prioritisation of flows. Finally, Mosyre et al. [29] quantify the VM network performance limitations in 10 GbE datacenters and explore the effectiveness of multiple network operating offloading mechanisms in datacenterers (SR-IOV VF, ToR tunnelling offloading) to improve throughput. We plan to explore similar offloading opportunities to further improve the scalability of our system.

6 CONCLUSIONS

In conclusion, modern cloud service design is heavily dominated by the paradigm of separation between application logic and deployment configuration and management. This approach limits the ability to develop flexible mechanisms that dynamically adapt the service configuration in order to meet application-specific demands and service requirements. This paper presents Fractal, an alternative approach to cloud service management which allows embedding of service management in the application logic of the service.

Fractal extends the Jitsu control stack on the dom0 through a simple RPC API which allows simple, dynamic, programmable creation and destruction of service replicas over the cloud infrastructure from within the service. This enables such decisions to be taken at whatever granularity and based on whatever metrics the developer sees fit. Fractal implements network-based load balancing using the OpenFlow protocol as exposed by the Open vSwitch softswitch running in the dom0 of the XenServer. As a proof of concept of the applicability of Fractal, we have developed a simple self-scaling web service based on a MirageOS webserver. Evaluating this Fractal-enabled unikernel shows that the proposed system does not have significant negative performance impact, while effectively providing flexible load distribution across replicas.

Although we have presented and evaluated Fractal in the context of MirageOS unikernels, it is not limited to either MirageOS or unikernels but can be adopted by other applications. For example, both nginx and Apache define an extensive plugin API which can extract important application measurements that could be used by scripts to invoke Fractal API. However, the introduction of VMs with disks, storage and other less straightforwardly replicated components could affect both the performance and the complexity of the replication and destruction processes. Recent developments using rumpkernels to convert legacy applications into unikernels mitigate these problems however, and may provide an excellent middle ground [25].

The use of Irmin to store application state potentially offers more significant benefits than we have demonstrated here. By taking what would be kernel datastructures in a traditional OS environment, and managing them inside Irmin in the unikernel, it becomes possible to replicate such state. For example, this might enable the state of a running unikernel’s ARP stack to be replicated along with the unikernel itself if the replica was created within the same subnet, reducing both the time it takes for the replica to become active and the potential for ARP storms [35].

In summary, we believe that Fractal provides the basis for a useful and novel service management framework that matches the dynamic nature, short timescale variability and complex configuration needs of modern microservice architectures. Although most powerful when configured for use with modern unikernels and a storage system such as Irmin, Fractal is also applicable in more traditional datacenter contexts. Fractal is available under a BSD licence at https://github.com/koleini/jitsu.

ACKNOWLEDGMENTS

This work was funded in part by UK EPSRC EP/K031724/2, EP/G065802/1, and EP/M02315X/1.

REFERENCES

[1] High availability for Citrix XenServer. www.citrix.com.
[2] A10networks thunder ADC. https://www.a10networks.com/sites/default/files/ A10-DS-15100-EN.pdf.
[3] AWS Elastic Load Balancing. https://aws.amazon.com/elasticloadbalancing/.
[4] Ansible inc. http://www.ansible.com.
[5] Barracuda Load Balancer ADC. https://www.barracuda.com/assets/docs/dms/ Barracuda_Load_Balancer_ADC_DS_US.pdf.
[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In Usenix SOSP, 2003.
[7] E. Brewer. Keymaster and the path to cloud native. In ACM ScCC, 2015. Keynote.
[8] Brocade ADX Series. http://www.brocade.com/en/products-services/ application-delivery-controllers/adx-series.html.
[9] Chef software inc. https://www.chef.io/.
[10] B. Fininets, T. Gaszyn agree, and A. Madhavapeddy. Manageable persistent data structures. In JFLA 2015.
[11] A. D. Ferguson, A. Gauba, J. Place, R. Fonseca, and S. Krishnamurthi. Participatory networking. In Usenix HotICE, 2012.
[12] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In ACM SIGCOMM, 2014.
[13] T. Gaszyn agree and V. Hanquez. OXenstore: an efficient hierarchical and transactional database using functional programming with reference cell comparisons. SIGPLAN Notices, 44(9), 2009.
[14] N. Handigol, S. Sretharan, M. Flajsljik, N. McKeown, and R. Jozari. Pluggable load-balancing web traffic using openflow (demo). 2009.
[15] S. Hanks, D. Meyer, D. Farinacci, and P. Traina. Generic routing encapsulation (GRE). https://tools.ietf.org/html/rfc2784, Mar. 2000.
[16] HAProxy. http://www.haproxy.org/.
[17] HTTP Archive Interesting Stats. http://www.httparchive.org/interesting.php, Sept. 2015.
[18] M. Isard. Autopilot: Automatic data center management. SIGOPS Oper. Syst. Rev., 41(2), 2007.
[19] E. Kalyvianaki, T. Charalambous, and S. Hand. Adaptive resource provisioning for virtualized servers using kalman filters. ACM Trans. Auton. Adapt. Syst., 9(2), 2014.
[20] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. ACM Trans. Comput. Syst., 18(3), 2000.
[21] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudon, and M. Satyanarayanan. Snowball: Rapid virtual machine cloning for cloud computing. In ACM EuroSys, 2009.
[22] loadbalancer cloud products. http://www.loadbalancer.org/uk/products/cloud.
[23] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gaszynagree, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie. Jitsu: Just-in-time summoning of unikernels. In IEEE INFOCOM, 2015.
[24] A. Madhavapeddy, R. Mortier, C. Rosots, D. Scott, B. Singh, T. Gaszynagree, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In ACM ASPLOS, 2013.
[25] A. Madhavapeddy and D. J. Scott. Unikernels: The rise of the virtual library operating system. Commun. ACM, 57(1):61–69, Jan. 2014.
[26] H. Mekky, F. Hao, S. Mukherjee, E. L. Zhang, and T. V. Lakshman. Application-aware data plane processing in SDN. In ACM HotSDN, 2014.
[27] Apache mesos. https://mesos.apache.org/.
[28] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. AGILE: elastic distributed resource scaling for infrastructure-as-a-service. In USENIX ICAC, 2013.
[29] R. Niranjan Mysoor, G. Porter, and A. Vahdat. FastTrak: Enabling express lanes in multi-tenant data centers. In ACM CoNEXT, 2013.
[30] Neutron load balance as a service. https://github.com/openstack/neutron-lbaas.
[31] OpenStack. http://www.openstack.org/.
[32] R. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In ACM SIGCOMM, 2013.
[33] B. Pfaff and B. Dave. The Open vSwitch Database Management Protocol. https://tools.ietf.org/html/rfc7047, Dec. 2013.
Fractal: Automated Application Scaling

[34] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In USENIX NSDI, 2015.

[35] M. Preston, M. Skjegstad, T. Gazagnaire, R. Mortier, and A. Madhavapeddy. Persistent networking with irmin and MirageOS. In Proc. OCaml Workshop at ICFP, Sept. 2015.

[36] Puppet labs. https://puppetlabs.com/.

[37] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale Cluster Management at Google with Borg. In ACM EuroSys, 2015.

[38] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekarft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. SIGOPS Oper. Syst. Rev., 39(5), 2005.

[39] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In USENIX HotICE, 2011.