2015 7th International Workshop on Science Gateways

Distributed Systems of Microservices Using Docker and Serfnode

Joe Stubbs
Texas Advanced Computing Center
The University of Texas at Austin
Austin, TX 78758-4497
Email: jstubbs@tacc.utexas.edu

Walter Moreira
Texas Advanced Computing Center
The University of Texas at Austin
Austin, TX 78758-4497
Email: wmoreira@tacc.utexas.edu

Rion Dooley
Texas Advanced Computing Center
The University of Texas at Austin
Austin, TX 78758-4497
Email: dooley@tacc.utexas.edu

***Abstract*—We review container technology and the challenge of service discovery in microservice architectures and introduce Serfnode, a fully decentralized open source solution to the service discovery problem, based on the Serf project. Serfnode is a non-intrusive Docker image that composes one or more arbitrary Docker containers. The new images can be deployed into a cluster of Serfnodes, where it advertises itself and provides service discovery mechanisms, monitoring, and self-healing. The resulting cluster is a homogeneous and complete graph, with no master node. We survey existing solutions to the service discovery problem and compare them to Serfnode. As an example of the extensibility of Serfnode, we show the construction of a file system synchronization solution between Docker containers using Git.**

I. Introduction

Container technology, and Docker [1] in particular, is making a profound impact on distributed systems and cloud computing. While virtual machines provide an abstraction at the hardware level, the container model virtualizes operating system calls so that typically many containers share the same kernel instance. This key difference makes for a much more lightweight virtualization package that can be built, modified, rebuilt and shared far more rapidly than its hypervisor-based predecessor. The rise in container popularity coincides with a growing trend away from monolithic software architectures where many components of a software system run in a single process. An alternative that is gaining traction is the so called microservices architecture where systems are composed from many smaller, loosely coupled services. Containers and microservices are a natural pair, helping to realize greater modularity, code reuse, reproducibility and fine-grained scaling of a distributed system. Nevertheless, achieving highly-availability in such a system is not without challenges, the greatest of which perhaps being the so called service discovery problem. In this paper we introduce Serfnode, a fully decentralized open source solution to the service discovery problem based on the Serf project [2]. Serfnode is a non-intrusive Docker image that advertises any Docker container to a dynamically formed cluster of Serfnode containers, and facilitates service discovery and event-based communication.

Any interesting distributed system requires some communication between the components. In a microservice architecture, it is typical for many instances of a particular service to be running at any one time and for these instances to stop and start over time. The problem of service discovery is to enable service consumers to locate service providers in real time to facilitate communication.

Serfnode joins an arbitrary Docker container to an existing cluster of other Serfnode-powered containers and facilitates service discovery and other communications across the cluster without modifying the original container. Serfnode also contains a monitoring and self-healing mechanism based on Supervisor [3] for added resiliency. Serfnode is very lightweight and simple to use. No installation is required beyond downloading the image from the public Docker hub and providing some minimal configuration. The basic idea is that instead of running a given container directly, one runs a Serfnode container instead, passing it a small amount of configuration that includes which container to run and an identifier for the service being provided by that container. The Serfnode container then ensures that the service container is running and maintains a local registry of services in the cluster. While many solutions to service discovery exist in the open source space, Serfnode distinguishes itself as lightweight, platform-agnostic and easy to integrate with an existing system of Docker containers.

The underlying Serf library powers the cluster communication with an event system implemented via a Gossip protocol. In terms of the Consistency, Availability, Partition tolerance (CAP) theorem [4], Serf's implementation of the gossip protocol favors availability over consistency in the presence of partitions, an appropriate trade-off for the service discovery problem.

The rest of the paper is organized as follows. In Section II we begin with an overview of container technologies and microservice architectures before describing the service discovery problem in greater detail. Section III highlights related work specifically within the context of container-based service discovery solutions at the time of this writing. Sections IV and V form the technical heart of the paper and detail the design, use case, and implementation of the Serfnode architecture. Section VI examines the flexibility of Serfnode by looking at how it was used to solve a file system synchronization problem. We conclude the paper in Section VII and address areas of future work.

II. Containers, Microservices and Service Discovery

A. Containers, Jails and Docker

Container technology is not new. With roots in Unix chroot, BSD Jails [5], and Linux LXC [6], container technologies have

978-1-4673-7459-0 2015

DOI 10.1109/IWSG.2015.16

been in wide use for over two decades. Containers are attractive because they enable complete isolation of independent software applications running in a shared environment. Because of this, they can greatly mitigate traditional attack vectors through isolation of the network, file system and resource consumption of each container.

Docker represents a natural evolution in the container technology space. The Docker platform integrates and extends existing container technologies by providing a single, lightweight runtime and API for managing the execution of containers as well as managing the container images themselves. While Docker is still relatively new at the time of this writing, it has infiltrated every type of activity, from system administration to scientific computing.

Perhaps the greatest reason Docker has seen such pervasive adoption is because it allows software applications to prepackage their dependencies into a lightweight, portable runtime requiring less overhead to run than even a standard hypervisor. Docker containers are persisted as a simple tar archive, thus installing an application packaged with Docker requires little more than copying a file to the target machine. Likewise, Docker containers can be used in far more sophisticated ways to reproduce complex systems across different environments with fewer resources and faster turnaround than was ever possible before. It is this latter point that lends itself particularly well to microservice architectures.

B. Microservice Architectures

The last decade brought enormous change for information technology on several fronts. Mobile devices, data proliferation, virtualization and the shift to cloud computing paradigms are just a few of the recent major advances, and modern software systems have been rapidly increasing in complexity as a result. The typical web-based system now has several components: a web front-end, an identity and access management system, a task queue, asynchronous worker processes, multiple databases, a logging system, aggregation and analytics, reporting and more.

Traditionally, web applications were built using a monolithic architecture where nearly all the components of the system ran in a single process. In simple situations this architecture has advantages: deployment and networking are trivial and scaling the system is accomplished by running multiple instances of the monolith behind a load balancer. Nevertheless, as requirements grow in complexity and demand increases, these systems suffer from difficult challenges. Changes to one component can impact seemingly unrelated areas of the application increasing the risk of new development. Individual components cannot be deployed, making wide reuse across the enterprise more difficult. Components designed for reuse gravitate towards verbosity and readability over performance due to conflicting needs and levels of experience in the development team. At the same time, scaling of individual components is not possible, and together these leads to obvious inefficiencies.

The microservices architecture replaces the monolith with a distributed system of lightweight, narrowly focused, independent services. There is no consensus on the precise definition of microservice architecture, and even the distinction between "service" and "component" (in a monolith) is up for debate. We adopt the following definitions: a component is independently replaceable and invoked via in-memory function calls while a microservice is an separate process that uses messages to communicate with other parts of system. Typically, microservices use RPC or web based protocols like TCP or HTTP to communicate, while components in a monolith communicate through shared memory. Services can be developed in disparate languages and tools by teams with different skill sets. They can be deployed, upgraded and scaled individually and their nature in theory leads to greater code reuse. In these ways, microservices overcome many of the shortcomings of the monolithic architecture. Yet, despite all the problems they solve, microservices introduce some particularly challenging new problems.

C. The Service Discovery Problem

Despite the ease in which we can describe microservice architectures, actually developing them is a tremendous challenge. Tasks such as deployment and log aggregation that were once trivial with monolithic and three-tier applications become incredibly complex in this new paradigm. Perhaps the greatest challenge to successfully implementing a microservice architecture is service discovery. While containers can simplify the deployment and distribution of individual components, they do little to address the issue of communication between services over a complex networking layer. This quickly becomes a problem as an application begins to scale. It is typical for redundant instances of individual microservices to be started and stopped based on demand. Because services cannot be counted on to be present at any given moment, consumer services need a mechanism by which they can locate provider services in real-time.

Typically, service discovery is tightly aligned with service monitoring such that a service instance only gains and/or retains its place in a service registry after successfully passing its monitoring tests. Also commonly included in mature service discovery solutions is a mechanism for service consumers to specify a preferred scheme for distinguishing a specific service provider instance when multiple provider instances are available. Typical schemes include minimizing distance (network hops) to the provider, minimizing the total number of providers used by a given consumer, and using the oldest or newest provider available. Before examing how Serfnode solves the service discovery problem, we highlight the most popular container-based service discovery solutions available today.

III. Related Work

Service discovery is a difficult problem that is ubiquitous in field of distributed systems. Perhaps nothing exemplifies that better than the sheer number of projects attempting to solve it. While much progress has been made, Docker introduces variations on these challenges, and some solutions are more fit for certain kinds of deployments than others. In this section we survey some of the existing solutions and compare them to Serfnode.

Consul [7] is a powerful open source solution for service discovery and distributed configuration management. It

is among the best options for many deployments because it bundles service discovery with a strongly consistent key-value store, robust monitoring, and health checking. In some ways, Serfnode might be considered a lightweight alternative to Consul, both in features and complexity, that can be used for services running in Docker containers. The two solutions share the underlying Serf library for cluster management, but Serfnode focuses exclusively on service discovery which ultimately results in a more simplistic system. Consul agents run in one of two modes—"client" or "server"—and Consul relies on a central set of agents running in server mode in each data center to maintain cluster state. In essence, the reliability of the consul cluster is only good as the reliability of the server cluster which adds complexity to deployments. Without a sufficient number of server agents in each data center (an odd number required with 3–5 recommended), data loss can occur. If the central server agents cannot form a quorum, Consul cannot operate. In contrast, every node in a Serfnode cluster has the same set of responsibilities, meaning that a Serfnode deployment is not required to distinguish and separately maintain a sub-cluster within the cluster. If any number of Serfnodes fail, the remaining Serfnodes continue to operate. This simplifies deployments and makes it easier to reason about the availability of any particular service. On the other hand, Consul's Raft-based [8] quorum of server nodes provide strong consistency while Serfnode merely provides eventual consistency, especially when deployed across multiple datacenters. This additional consistency is particularly important for the key-value store but less so for service discovery. Finally, Consul does not support services running in Docker containers out of the box, and in fact entire projects such as `docker-consul` [9] have sprung up to add Docker support.

Synapse [10] is another open source project for service discovery. Unlike Consul, Synapse was designed to support services running in Docker. At the core of Synapse is an HAProxy instance that is used to route requests from a service consumer on the same host to a service provider running in the cluster. Updates to the HAProxy configuration are made by "watchers"—daemons that check for changes to the locations of services. Currently, three types of watchers are supported: "AWS EC2 tags", "Docker", and "Zookeeper"; however, AWS EC2 tag watchers only work on the Amazon platform, and Docker watchers have no way of providing service discovery across Docker hosts. Zookeeper watchers provide platform-agnostic, multi-host discovery in theory, but additional components are needed to actually keep Zookeeper updated. This is a significant undertaking and in fact the focus of entire projects. Moreover, some have questioned the use of Zookeeper as a basis for service discovery since it tends to sacrifice availability for consistency in the presence of partitions, a trade-off more appropriate for coordination, the task for which it was originally designed. Finally, deploying and maintaining a Zookeeper cluster is notoriously difficult, so much so that entire projects have been devoted to mitigating misconfigurations [11].

CoreOS [12] provides first class service discovery as a basis of their operating system through the etcd service. etcd [13] is a distributed, consistent key value store for shared configuration and service discovery with a focus on ease of use and performance. etcd, like Zookeeper, is an implementation of the Raft [8] consensus algorithm. While etcd is a viable, and often times the preferred solution for service discovery on CoreOS, when taken out of that ecosystem, it falls back to simply serving as another key/value store without tight integration into a broader service discovery solution such as the others listed here.

The Docker platform has announced the coming of new offerings Docker Swarm [14] and Docker Compose [15] that, when combined together, may provide a solution to the service discovery problem. Docker Compose already provides a very limited service discovery mechanism, but it currently only works on a single host and it does not update as containers are stopped and restarted. It is unclear exactly what Swarm will enable as details of that feature have yet to be announced.

It is important to distinguish the service discovery problem from "orchestration"—the process of deploying containers on a cluster of machines. CoreOS, Mesos/Mesosphere, OpenShift, CloudFoundry, Kubernetes, Brooklyn/Clocker, Shipyard, and Crane are just a few of the many available orchestration solutions, and some of the larger projects like Kubernetes and Mesosphere include some service discovery features. A full survey and comparison is beyond our scope, but in general, orchestration solutions represent a fundamentally different approach to managing systems built with Docker. Adoption of these solutions is often a very heavy undertaking requiring highly specialized deployment environments.

IV. Serfnode: a Solution to Service Discovery

A. Design

Though flexible and extensible, Serfnode was mainly designed to be lightweight and simple to use, much like Docker itself. The primary goal was to enable production deployments of small to medium-sized systems of Docker containers with minimal installation. For that, we chose a fully decentralized architecture with homogeneous nodes in a completely connected graph, avoiding a central registry, designation of "master" nodes, or a complex network topology. Crucial to the success of our own use of Serfnode was its ability to take an arbitrary Docker container and, without modification, connect it to other members of a cluster in a way that facilitated service discovery and consumption. Additionally, a lightweight and extensible monitoring and self-healing mechanism was included, as these features are essential for any viable service discovery solution.

We fully expect Serfnode to scale to thousands of nodes, though currently its use has been limited to much smaller deployments. Traditional "heart beating" protocols impose network loads that grow quadratically with the cluster size, but the underlying SWIM [16] gossip protocol used by Serfnode overcomes this scaling issue by achieving a message load per member that does not vary with group size. With that said, Serfnode is optimized for use within a single data center, and consistency and convergence rates will be diminished across a WAN. For larger deployments across many data centers, a solution like Consul will likely be more appropriate.

B. Serfnode for Science Gateways

Modern, web-based science gateways are complex distributed systems made of components common in other web

systems. Serfnode was created for the Agave science-as-a-service platform, an ecosystem of open-source hosted developer APIs and tools for building science gateways [17]. Agave is the backbone of NSF funded projects such as the iPlant collaborative with over 15,000 users submitting thousands of jobs and moving over two petabytes of data every month. Production components of Agave run as Docker containers packaged up with Serfnode for service discovery. Additionally, we have open sourced the Agave deployer program, itself a Docker container, that interested parties can use to stand up their own Agave instance—nearly 40 containers in total—across multiple virtual machines with a single command. The only configurations needed for the Agave deployer are the IP addresses of the hosts themselves and SSH connectivity credentials.

V. Structure of a Serfnode

A Serfnode is a Docker image containing a serf agent and a supervisor instance. The mental model to use for Serfnode is a very lightweight envelope around one or more images provided by the user. The wrapped images are called *children*, while the Serfnode image that wraps them is called *parent*. In the most typical case, a Serfnode will wrap a single child container providing the actual service. For instance, a basic system consisting of a Python web service and a Redis database would employ two Serfnodes—one wrapping the web service and one wrapping the database. A Serfnode wrapping a given image `A` behaves exactly as `A` from the point of view of services and storage.

It is useful to identify a Serfnode with the set of images that are being wrapped. One simple way to highlight this association is to create a derived image containing a configuration file describing the children. For example, given an arbitrary image `A`, we can create a customized Serfnode image, say `serf_A`. Then, we can think of `serf_A` as the "serf-aware" version of `A`. Alternatively, we could add the configuration file through a bind-mounted volume at runtime. The choice on which approach to take is largely determined by the overall deployment strategy of the larger application. Building custom Serfnode images is an efficient way to define pods of containers that can be assigned to run together on a host machine. Adding the configuration file at runtime gives a higher degree of freedom, but adds complexity to the deployment process.

Serfnode adds three pieces of functionality to an arbitrary image: cluster membership, monitoring, and event handling. Figure 1 on page 5 illustrates a cluster of Serfnodes. It is important to stress that this extra functionality is obtained without modifying the given images for the children.

A. Cluster membership

A cluster is a non-empty set of Serfnodes. They can run in the same host machine, in separate machines, or in a combination of both situations. The only condition is that each member should be reachable at a defined address from any other member (here address means a pair of IP and port). A new Serfnode joins a cluster by receiving the address of any other node in the cluster at start-up. An event advertising the joining of a new member propagates through all the nodes. Similarly, if a node fails or leaves the cluster, an event propagates to all the other members informing about the exiting member.

Serf propagates events using the gossip protocol and communicates state and membership using the SWIM protocol [16]. The implementation of Serf in Serfnode uses UDP packets to transmit across members of the cluster. In practice, this is fast and reliable when the Serfnodes are spread across a local network. If a cluster is spread over wider networks, say across data-centers, then large rates of packet loss may impact the reliability, making some nodes to connect and to disconnect randomly from the cluster. However, this situation is not a concern in local networks, which is the main environment for a deployment of Serfnodes.

Each member in a cluster has a defined *role*. This is simply a label attached to a Serfnode at start-up time. There can be multiple members with the same role and a member can have multiple roles. This label is used to discover the service or services provided by each member. Serfnode writes the association of roles with IP addresses in the file `/etc/hosts` in every member of the cluster. For standard services, such as a database, this association is usually enough to reach the service. However, in case of non-standard services, or when ports are mapped, the more complete address information (IP and ports) is written in specific files on each member. These files constitute the "developer interface" of the Serfnode.

B. Monitoring

Serfnode executes the children images at start-up, obtaining the parameters for the Docker `run` statement from a configuration file. The configuration file is a YAML file with name `serfnode.yml` mounted at the root of the parent container. Its structure is similar to the files used by Docker Compose. As an example, the following file declares a Serfnode with role `my_role` wrapping two images:

```
---
serfnode:
  role: my_role

children:
  - my_container:
      image: ubuntu
      command: sleep infinity
      volumes:
        - /tmp:/host
      environment:
        MY_VAR: 1
      working_dir: /host

  - my_other_container:
      image: ubuntu
      command: sleep infinity
      volumes_from:
        - my_container
```

This configuration file instructs the Serfnode parent to start two children, roughly equivalent to executing two `docker run` statements with the appropriate command line parameters. The difference between these statements and the Serfnode internal execution scheme is that the parent process takes

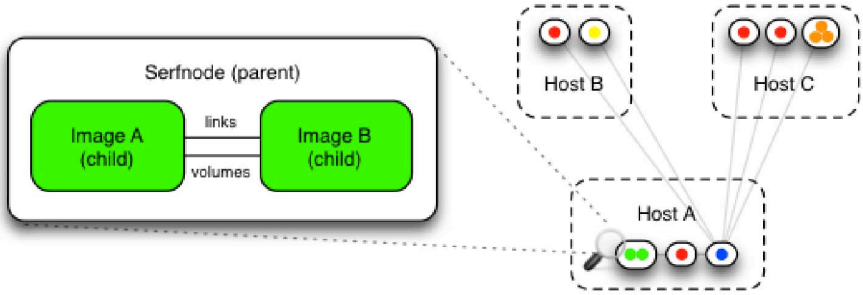

Fig. 1. A cluster of Serfnodes distributed in three hosts. Colors represent roles. The green member has two children, as shown in the magnifier glass. Solid gray lines illustrate how the blue member is connected to every other one. Connections between the rest of the members are *not* displayed to keep the picture simple.

care of making the name of the children globally unique. The renaming preserves the links and any volumes mounted across the children. This feature allows the user to think about the Serfnode as an atomic unit of functionality, without having to think about its individual components.

Once started, each child process is monitored by supervisor and restarted on abnormal exits. Changes of state from supervisor on a given node are also propagated through the cluster. Note that the monitoring works only at the level of individual Serfnodes. When the Serfnode parent process dies, there is no built-in mechanism for restarting the complete Serfnode. However, when the parent process dies an event is propagated in the cluster, allowing other members to take action. Serfnode has intentionally avoided defining a restarting mechanism across the cluster, in an effort to keep a lean and extensible profile. One possible solution to this is leveraging the newly added docker `--restart` flag when starting the Serfnode container to guarantee the container restarts as long as the host exists.

C. Event Handling

There are three type of events in a Serfnode cluster. The first type consists of membership events: join, leave, and fail. Serfnode provides basic handlers for these events, which update the file `/etc/hosts` and update a set of specific files (the developer interface) to reflect the global state of the cluster at any moment. Developers can also add functionality to these handlers to perform extra actions on membership changes.

The second type of event consists of supervisor state changes. When children processes change state, an event communicating the old and new state is propagated through the cluster. These events can be used to implement a global monitoring policy. For example, if a child fails to restart after a given number of attempts, the event propagated to other nodes can be used to take an action on a different node, possibly on a different host computer.

The third type of event consists of user defined events. These are arbitrary labels that can be triggered by a Serfnode parent or by a child process, and they are handled by a user defined function with the same name as the label. Children can generate events by writing to a named pipe connected to the parent. This is also part of the developer interface, available in any child. Section VI shows a useful example of user defined events.

D. Security Considerations

Serfnode requires access to the Docker daemon socket or API in order to spawn child containers. This effectively gives Serfnode extra privileges. However, it is important to note that child containers run in non-privileged mode and they do not have access to the Docker daemon. This means that the security with respect to third party images is as strong as the docker daemon provides.

The network traffic of Serf agents inside the cluster is not encrypted. Access to the serf agents is also not authenticated. Ideally, all Serfnodes would run on hosts within a VPN and they would only accept inbound traffic from a white-list of IP addresses such as a proxy server, load balancer, or a predefined list of hosts. Each individual member of the cluster providing a service to the outside world is expected to implement its own access control.

A cluster of Serfnodes forms a complete graph, which means that every pair of members must be connected. This may require a change in an existing network topology where the cluster is being deployed. This is usually a lightweight requirement when the deployment is within a VPN. For other situations, a Serfnode cluster could be deployed as a hierarchy of sub-clusters joined themselves in a cluster. This kind of deployment can accommodate an arbitrary topology. We are planning as future work an easy API for declaring these cases.

VI. Example: File System Synchronization

As an example of the power and extensibility of this project, we describe a simple solution to the file synchronization problem that is implemented on top of Serfnode. The problem is to maintain a file system (a collection of directories and their files) in sync across many nodes. The solution we present is eventually consistent and performs well when the ratio of read operations to write operations is large. It works on any path within an arbitrary container, and no modifications are needed to the container.

The synchronization solution consists of two docker images: an extension of the base Serfnode image which we call `git-sync-serfnode` and an image called `git-sync` which runs as a child of the `git-sync-serfnode`. Both images are publicly available from Docker Hub. The `git-sync` image is an ssh server combined with a simple event loop based on inotify [18] that is configured to watch a

particular directory in an arbitrary container and issue a serf event which we call "sync" when any create, update or delete event takes place in the directory. This sync event is a message which contains the IP and port of the `git-sync` ssh server. The `git-sync-serfnode` is a basic Serfnode with two extra features. First, it mounts the configured directory from the arbitrary child container into the `git-sync` container (using the Docker "volumes from" command). Additionally, it adds an event handler for the sync event which issues a git fetch from the within the child container, using the IP and port contained in the sync message as the git remote, followed by a git reset to the fetched head. It follows that the contents of the configured directory in the child container match that of the remote after a sync event is processed. The git communication happens over SSH and authentication via SSH keys is handled automatically by the containers.

Therefore, given any Docker container C and directory path p, one can run a `git-sync-serfnode` whose children include a `git-sync` container and C, and configure the `git-sync` child to watch path p in C. All `git-sync-serfnodes` configured in this way will automatically keep the contents of p in sync with each other. Note that no changes whatsoever are required of the original container C.

This is a very simplistic approach to file system synchronization and it is clearly prone to data loss in situations where updates are occurring frequently on many different nodes. A more sophisticated merge strategy could mitigate this to some extent, but this solution is already robust enough to handle many production use cases. It scales to large numbers of nodes as long as updates do not occur frequently, and it is very simple to understand and implement.

VII. Conclusion and Future Work

Distributed systems of microservices can overcome shortcomings of traditional monolithic architectures by allowing independence of development, deployment, updating and scaling of components, but engineering such systems is not without challenges. Container technology, and Docker in particular, can greatly simplify the creation, deployment and maintenance of the individual services, but a system to facilitate communication between the services is still required.

We described the design and structure of Serfnode: an extensible solution to the service discovery problem for microservices running in Docker containers, aimed to be as easy to use as possible. It does not alter the original containers and no special infrastructure is needed to start using it—all that is required is Docker and the Serfnode image. The new images form a decentralized cluster of Serfnodes, where they advertise themselves providing service discovery. Additionally, we showcased Serfnode's extensibility by providing a solution to the file system synchronization problem.

While we fully expect Serfnode to scale to thousands of nodes, it is most appropriate for small to medium deployments within a single data center. Performance is expected to degrade as the network widens.

In the immediate future we plan to increase code coverage for Serfnode including functional and performance tests. We have also have a higher level API planned for common tasks such as spawning new members. Additionally, we plan to add support for recently announced and upcoming Docker tools such as Swarm and Machine.

Acknowledgment

This material is based upon work supported by the National Science Foundation Plant Cyberinfrastructure Program (DBI-0735191), the National Science Foundation Plant Genome Research Program (IOS-1237931 and IOS-1237931), the National Science Foundation Division of Biological Infrastructure (DBI-1262414), the National Science Foundation Division of Advanced CyberInfrastructure (1127210), and the National Institute of Allergy and Infectious Diseases (1R01A1097403).

References

[1] (2014) Build, Ship and Run Any App, Anywhere. Last access: 2015-03-13. [Online]. Available: http://docker.com

[2] (2014) Serf by hashicorp. Last access: 2015-03-13. [Online]. Available: http://www.serfdom.io

[3] (2004) Supervisor: A Process Control System. Last access: 2015-03-26. [Online]. Available: http://supervisord.org

[4] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/564585.564601

[5] P. Henning Kamp and R. N. M. Watson, "Jails: Confining the omnipotent root," in *In Proc. 2nd Intl. SANE Conference*, 2000.

[6] (2014) Linux containers. Last access: 2015-03-24. [Online]. Available: https://linuxcontainers.org

[7] (2014) Consul by Hashicorp. Last access: 2015-03-25. [Online]. Available: https://www.consul.io

[8] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 305–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643666

[9] (2014) Consul Agent in Docker. Last access: 2015-03-13. [Online]. Available: https://github.com/progrium/docker-consul

[10] (2012) Synapse. Last access: 2015-03-13. [Online]. Available: https://github.com/airbnb/synapse

[11] K. Ting, "Building an impenetrable zookeeper." Presented at Strange Loop, 2012. [Online]. Available: http://www.infoq.com/presentations/Misconfiguration-ZooKeeper

[12] (2014) Coreos is Linux for Massive Server Deployments. Last access: 2015-03-24. [Online]. Available: https://coreos.com/

[13] (2014) coreos/etcd. Last access: 2015-03-25. [Online]. Available: https://github.com/coreos/etcd

[14] (2014) docker/swarm. Last access: 2015-03-25. [Online]. Available: https://github.com/docker/swarm/

[15] (2014) docker/compose. Last access: 2015-03-25. [Online]. Available: https://github.com/docker/compose

[16] A. Das, I. Gupta, and A. Motivala, "Swim: Scalable weakly-consistent infection-style process group membership protocol," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 303–312.

[17] R. Dooley, M. Vaughn, D. Stanzione, S. Terry, and E. Skidmore, "Software-as-a-service: The iplant foundation api," in *5th IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, Nov. 2012. [Online]. Available: http://datasys.cs.iit.edu/events/MTAGS12/p07.pdf

[18] (2004) inotify(7) - Linux manual page. Last access: 2015-03-26. [Online]. Available: http://man7.org/linux/man-pages/man7/inotify.7.html