Spawnpoint: Secure Deployment of Distributed, Managed Containers

John Kolb

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-1
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-1.html

January 13, 2018

Acknowledgement

This work would not have been possible without collaborations involving my colleagues in the Building, Energy, and Transportation Sytems (formerly Software Defined Buildings) research group. Michael Andersen developed Bosswave, which has been an essential building block for Spawnpoint, as well as the “Wavelet” Qt libraries that were used to build Spawnpoint’s graphical frontend. Kaifei Chen served as an early user of Spawnpoint and also helped configure his building vision application as a distributed Spawnpoint deployment, presented in Section 6.4. Gabe Fierro worked to deploy Spawnpoint as part of the BETS lab’s Extensible Building Operating Sytems (XBOS) project and has provided insightful feedback for improving and refining the system.

Spawnpoint: Secure Deployment of Distributed, Managed Containers

by

John Kolb

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master's of Science, Plan II

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Randy H. Katz, Chair
Professor David E. Culler

Spring 2018

The thesis of John Kolb, titled Spawnpoint: Secure Deployment of Distributed, Managed Containers, is approved:

Chair _____ Date _____

_____ Date _____

University of California, Berkeley

Spawnpoint: Secure Deployment of Distributed, Managed Containers

Abstract

Spawnpoint: Secure Deployment of Distributed, Managed Containers

by

John Kolb

Master's of Science, Plan II in Computer Science

University of California, Berkeley

Professor Randy H. Katz, Chair

`Spawnpoint` is an infrastructure for deploying and managing distributed software services as execution containers. The heart of the system is `spawnd`, a persistent daemon process that encapsulates the compute resources offered by a specific host. `spawnd` advertises available resources and accepts commands from clients to deploy and manage services. `spawnd` enforces resource reservations, ensuring that a host does not become oversubscribed, and performs admission control to avoid overloading. `Spawnpoint` offers command-line and graphical clients for interacting with `spawnd` instances. A third front-end, `Raptor`, allows users to describe complex distributed applications as a collection of services and specify how these services are deployed through a simple domain-specific language. `Spawnpoint` maintains security for all operations. Only authorized parties may learn of the existence of a `spawnd` instance, deploy a service on a host, monitor service state, or manipulate a running service. This is achieved through the use of Bosswave, a secure syndication network. Experiments show that the CPU, memory, and network costs of running `spawnd` are low and that managed containers have small overhead vs. unmanaged containers. `Spawnpoint` has been used to deploy real systems, such as a vision-based building control application and drivers to secure legacy IoT devices.

Contents

List of Figures

List of Listings

List of Tables

Acknowledgments

This work would not have been possible without collaborations involving my colleagues in the Building, Energy, and Transportation Sytems (formerly Software Defined Buildings) research group. Michael Andersen developed Bosswave, which has been an essential building block for `Spawnpoint`, as well as the "Wavelet" Qt libraries that were used to build `Spawnpoint`'s graphical frontend. Kaifei Chen served as an early user of `Spawnpoint` and also helped configure his building vision application as a distributed `Spawnpoint` deployment, presented in Section 6.4. Gabe Fierro worked to deploy `Spawnpoint` as part of the BETS lab's Extensible Building Operating Sytems (XBOS) project and has provided insightful feedback for improving and refining the system.

Chapter 1

Introduction

The explosion of smaller, more capable, more energy efficient, and better connected computing platforms is beginning to have profound consequences. In particular, computers are becoming increasingly pervasive in the physical world. This is perhaps best exemplified by the emergence of the Internet of Things (IoT), a vision for a rich network of embedded devices that bridges the divide between the physical and virtual worlds. More concretely, computers and embedded devices are being deployed in many physical systems, like buildings [35], cars [26], and the electrical grid [30], to instrument and monitor these systems while also using software to impelement sophisticated and flexible control schemes. However, the devices alone are not enough. Edge devices require the support of external software to ingest and store sensor data, to glean insights from this data, to orchestrate the operation of compositions of devices that together form larger systems, and so on. Visions of highly intelligent cyber-physical systems and ubiquitous computing, e.g., [51], require both devices deployed in the environment as well as reliable and long-lived software services to back them.

This naturally gives rise to the question of where these underlying applications and services will run and how they will be managed. The cloud is often viewed as an attractive complement for edge devices because of its vast computing and storage capabilities. However, the deployment of software on-premises offers its own set of benefits. Local services can control the flow of sensitive information, such as home occupancy data, out of the local area. They allow edge devices to continue operating when disconnected from the cloud and enable low-latency communication for purposes like tight control loops. Finally, local services can act as proxies for legacy devices, encapsulating a fundamentally insecure interface like an HTTP endpoint with a more secure and robust interface. This local tier of services is not a replacement for the cloud; rather, it is an important complement to the cloud with its own set of systems tradeoffs.

This thesis introduces `Spawnpoint`, a container management system that facilitates the deployment and maintenance of software services deployed across heterogeneous and physically distributed hosts, particularly drivers that run on-premises to encapsulate devices in the local environment. `Spawnpoint`'s primary role is wrapping Docker daemons in a secure interface using Bosswave [24] — a publish/subscribe platform in the vein of MQTT [14] and ZeroMQ [8] — elevating Docker containers to first-class citizens on a message bus. Bosswave uses a web of trust model for authenticated transport, enabling decentralized administration and fine-grained permissions that dicate who or what may send, receive, and view specific message streams. `Spawnpoint`'s main component is a daemon that runs on a specific physical host. The high-level architecture of this daemon is shown in Figure 1.1. The daemon advertises the host's computational resources and accepts remote commands for Docker container deployment and manipulation over Bosswave. Using Bosswave's permissions model, `Spawnpoint` can then enforce security guarantees regarding the host's visibility as well as who may push code to that host. `Spawnpoint` emits diagnostic heartbeat messages about the health of a host and the services it is running, which are similarly secured by Bosswave.

Any service that runs on a `Spawnpoint` node must reserve its CPU and memory re-

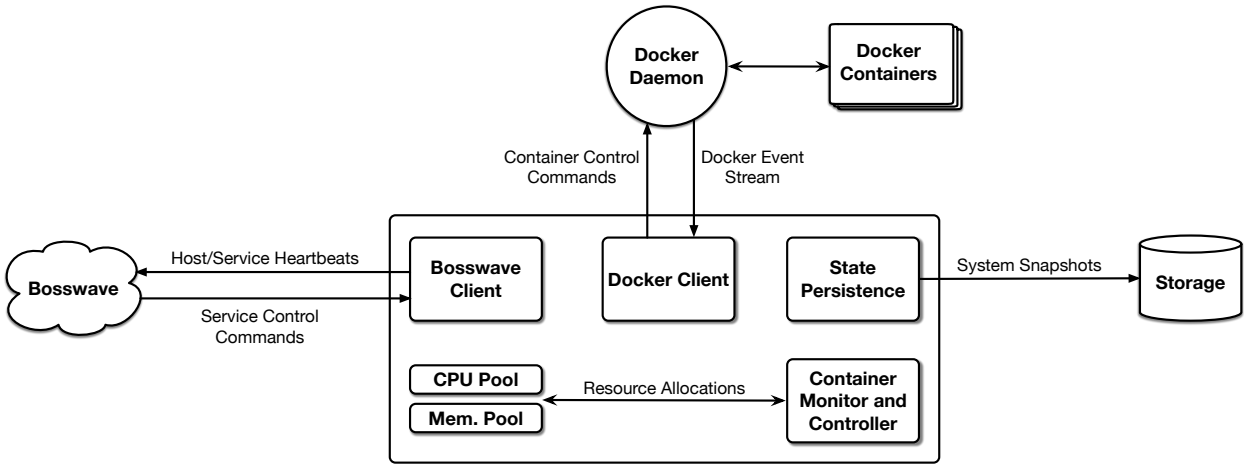


Figure 1.1: High-level design of the `Spawnpoint` host daemon

sources upfront, upon deployment. `Spawnpoint` maintains a global pool of these resources and allocates a slice from both pools to each service when that service is instantiated. `Spawnpoint` leverages Docker to enforce these allocations, ensuring that no single service monopolizes the resources of the host. A `Spawnpoint` daemon will also perform admission control, only accepting new services to a host if it it has sufficient CPU and memory to satisfy the requested allocations. This prevents a host from becoming oversubscribed. The availability of resources is emitted as part of a host's heartbeat messages, allowing developers and schedulers to make informed decisions about where to push new services.

Once a service is running, the `Spawnpoint` daemon manages its execution by monitoring the state of the service's container through the Docker daemon. This allows `Spawnpoint` to control the service's lifecycle, which can be modeled as a finite state machine, and perform useful operations like automatically restarting the container if a failure occurs. Transitions between the states of a service's lifecycle also prompt changes to its resource allocations, and the task of maintaining the integrity of these allocations becomes complex in the face of many simultaneously and independently executing containers that may potentially fail at any time.

`Spawnpoint` takes measures to ensure fault tolerance even in the case of host daemon failures. The execution of each managed container is independent of the execution of `Spawnpoint` itself, allowing a service to continue running when a daemon fails. Moreover, the daemon periodically saves snapshots of system state to persistent storage. When it restarts after a failure, the daemon reconciles this snapshot with its view of the currently running services, reclaiming ownership of containers that have remained running, and restarting containers that have terminated in the meantime. The combination of this decoupling and snapshotting helps minimize the cost of daemon failures.

`Spawnpoint` supports a variety of frontends that are used to issue commands to running daemons on behalf of end users. These include `spawnctl`, a command-line application,

as well as a graphical user interface. A third front-end, `Raptor`, is intended for the deployment and management of complex applications formed through the composition of services running on many Spawnpoint instances, which are potentially distributed. `Raptor` specifies a DSL in which end users express the services they want to deploy, any dependencies between these service containers, and a specification for where each container should run (e.g., on a particular Spawnpoint, on a Spawnpoint satisfying certain constraints, or anywhere).

The remainder of this thesis is organized as follows. Chapter 2 presents a motivating scenario for `Spawnpoint` in the form of a distributed application formed by the composition of many independent software components spanning several tiers of heterogeneous hardware, from the cloud down to on-premises servers. The notions of containers and distributed service management have been well studied, and Chapter 3 discusses the relevant properties of systems that address these issues and uses these properties as a basis for comparison with related work drawn from a variety of domains including container orchestration tools like Kubernetes [13], IoT frameworks offered by companies like Amazon [5] and Google [12], historical distributed systems like Emerald [40] and Ninja [38], and mobile-cloud offloading platforms such as Sapphire [52] and MAUI [34].

Chapter 4 describes `Spawnpoint`'s implementation and presents a refined version of Figure 1.1. The various frontends for interaction with `Spawnpoint` daemons are described in Chapter 5. Chapter 6 presents an evaluation of the `Spawnpoint` daemon, showing that it is lightweight — imposing a very small CPU, memory, and network bandwidth footprint on its host — and that its managed containers add little overhead on top of native Docker containers. This chapter also presents two real-world use cases for the system. Finally, Chapter 7 concludes the thesis.

Chapter 2

Motivating Scenario

Consider the goal of implementing a coordinated and intelligent approach to demand response across a fleet of buildings, perhaps on a corporate or university campus. Demand response [49] is the practice of dynamically reducing energy consumption during periods of high load, typically as a reaction to higher prices or at the request of a utility. This is becoming an increasingly important tool for balancing fluctuations in the load placed on the electrical grid, leading to more efficient and more reliable operation. However, the effort to reduce load cannot come at the expense of safe operation of a building's HVAC system and other appliances, nor at the expense of the comfort and safety of the building's occupants.

Therefore, building managers need to be able to both predict how specific adjustments to a building's HVAC, appliances, and other electrical loads will affect its energy consumption and measure the impact of these adjustments on building conditions. Further, each building within the larger group must be treated as an individual. This is because each building features a unique combination of hundreds to thousands of sensing and actuation endpoints, environmental conditions, occupancy patterns, purposes of use, and so on. For the purposes of demand response, a model must be constructed for each building using the data generated by its resident sensors and actuators that accurately maps perturbations to appliance settings (e.g., lowering the temperature setpoint for a particular zone, turning off all lighting fixtures of a particular type, etc.) to changes in energy consumption. However, there is also value to aggregating data across buildings. Campus officials may want to identify patterns that occur in multiple buildings, compare building performance, or observe the effects of campus-wide phenomena like weather events or holidays on building operation.

This scenario naturally gives rise to a hierarchical organization of devices as well as information, detailed in Figure 2.1. At the lowest level, we have hundreds or thousands of sensors and actuators deployed in each building. To collect data and take actions in response to a DR event, we need to interface with each building's highly heterogeneous collection of devices. Furthermore, these devices must be integrated with the computational resources deployed on the premises, generally in the form of local servers, that make up the next level of the hierarchy. These servers in turn need to cooperate to support campus-wide services and analytics workloads, perhaps using a campus datacenter as a point of coordination. Finally, there is the cloud, which offers immense, but remote, compute and storage capacity. This could serve as a suitable infrastructure for location-agnostic services like web scrapers to interact with sources of weather data or energy prices.

Implementing, deploying, and administering an automated demand-response system on top of this hierarchy involves a number of challenges. First, there is the issue of efficiently using and managing the computational resources available to the system, from edge devices to local machines to robust cloud servers. Second, there is the problem of collecting and processing building data while carefully controlling information flows for efficiency and privacy purposes. There is also the challenge of creating a system that lends itself to future extensions and maintenance. It should be easy to deploy new

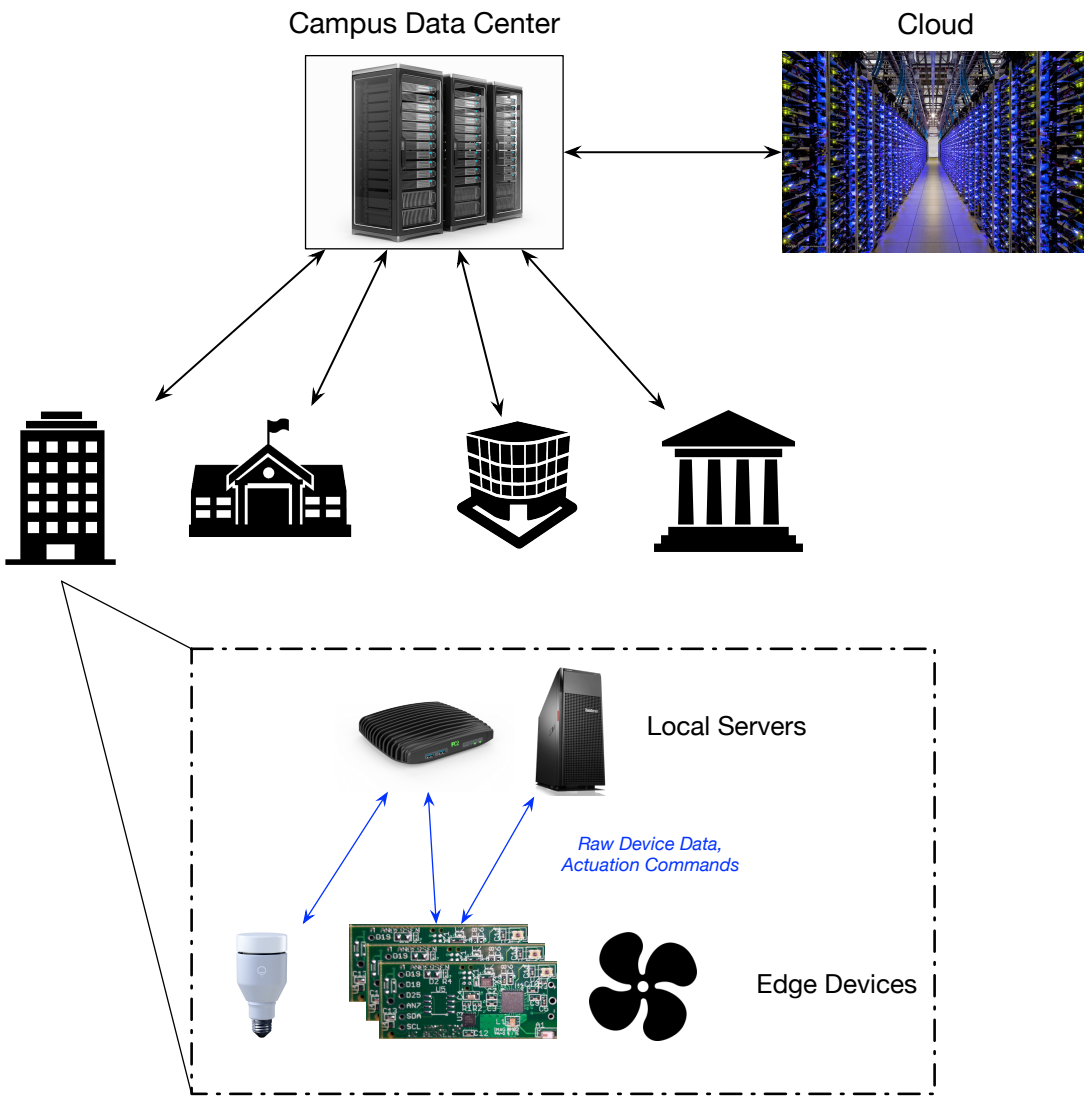


Figure 2.1: The hierarchical organization to a campus-wide automated demand-response application. Each building is an indepenent subdomain containing edge devices and local servers, and all buildings communicate wtih a common campus data center that is potentially augmented by the cloud.

software components and to move a service from one location to another, both within and across layers of the hierarchy. Finally, and perhaps most importantly, there is the issue of security. Campus-wide demand response involves a variety of stakeholders, such as building managers, building occupants, and researchers, each requiring different sets of privileges to perform tasks like installing new devices, interacting with these devices, and prototyping new analytics techniques or building control schemes. Thus, we need to be able to control who may deploy code where and place constraints on where each unit of

software may be executed.

Spawnpoint addresses the challenges identified above in two key ways: by managing the execution infrastructure for the array of services involved in campus-wide automated DR and by giving developers a set of primitives to make these services secure and portable. A slice through an example installation of a campus-wide DR system using Spawnpoint is shown in Figure 2.2. A Spawnpoint instance can encapsulate any host running Linux; thus each host involved in a DR system, from local BMS servers to a powerful cloud-based virtual machines, runs Spawnpoint in order to advertise their respective resources for use by software services. Additionally, each of these services runs inside a portable and isolated container, meaning developers can write code and run it anywhere. Spawnpoint also allows developers to deploy, manage, and update distributed software from a single interface that unifies the heterogeneous tiers of hardware shown in the figure. New components, such as a driver for a newly installed device or new data analysis methods, are easily added to the system by deploying new containers. The system can be easily maintained by updating services or by relocating them to different hosts as necessary.

Security concerns are handled through Spawnpoint's use of Bosswave. Each Spawnpoint instance maintains an independent set of permissions that determine who may push code onto the corresponding host. Analogously, service developers know where their code will be deployed because only the authorized Spawnpoint instance can receive their deployment directives. Each running service is associated with its own set of fine-grained permissions that determine who can inspect or manipulate that service. This gives rise to rich combinations of privileges that can be specifically adapted to different roles in the demand-response system. Building managers might be allowed to deploy and monitor code on any host within their own building or in the cloud, while campus administrators can see and interact with a service running in any building. Demand response researchers might be permitted to consume data from building-specific services but can only deploy code in the cloud. Moreover, Bosswave enables decentralized administration of and interaction with Spawnpoint instances. A user can deploy and manage their containers through any client that speaks Bosswave, and this client is decoupled from any particular Spawnpoint instance.

Finally, the Raptor front-end helps users cope with the complexity of managing multi-container (and potentially multi-host) applications. Raptor defines a domain-specific language for configuring multiple containers, declaring their interdependencies, and specifying where they should be run, all from a single file. This forms a declarative specification of an application's components, and it is submitted to the Raptor tool in order to deploy the application. Raptor will compute the difference between the services that are already deployed and the services included in the specification, only instantiating what is necessary to bring the deployment into the desired state. Raptor files can therefore be resubmitted at a later point in time to ensure that a deployment is in the proper condition. Furthermore, a group of developers can collaborate on a Raptor file and keep it under version control as an executable record of their application's structure. Raptor's approach of treating these files as declarative is inspired by Kubernetes, and much like

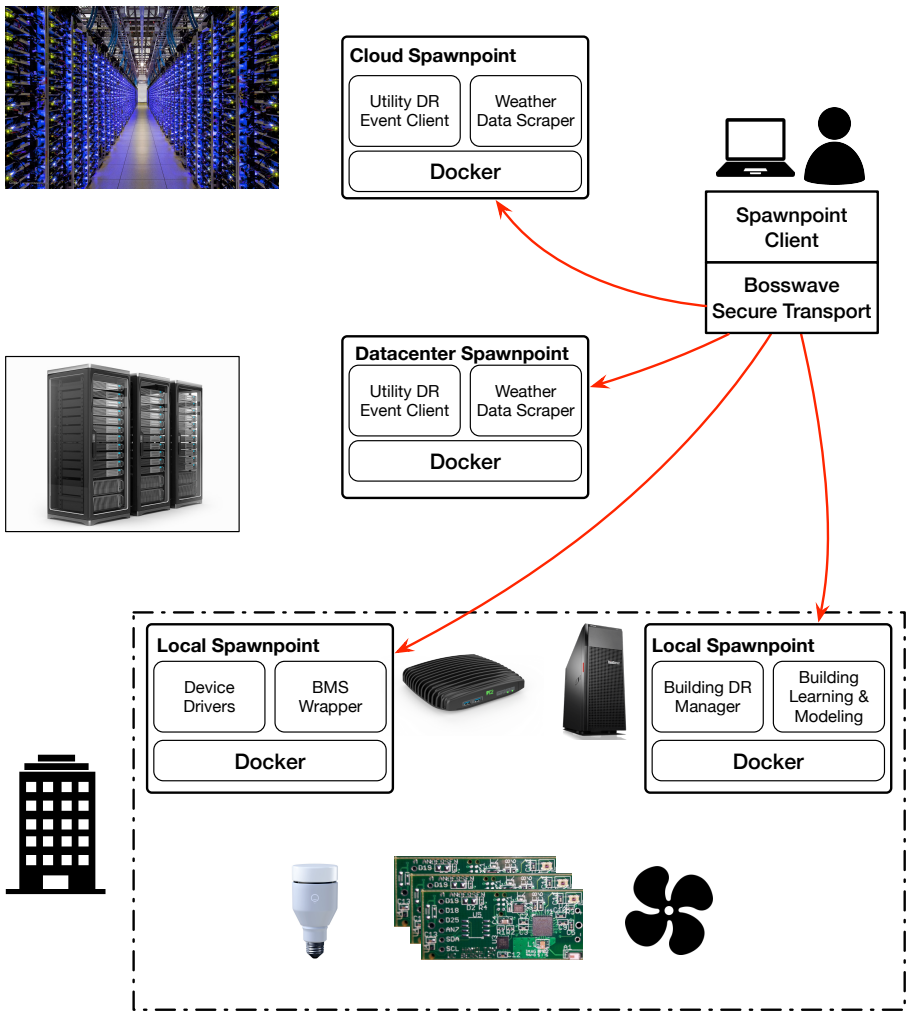


Figure 2.2: An example deployment of a campus-wide DR application. `Spawnpoint` instances encapsulate computational resources deployed locally in buildings, at a shared campus data center, and in the cloud. Building-specific services execute locally, aggregation and coordination occur on campus, and wrappers for external services run in the cloud. However, this is not the only possible arrangement. `Spawnpoint` ensures that changes to the physical placement of services do not affect the application's logical structure.

Kubernetes, a common use case is to edit the file, perhaps to add a new container or update the version on an existing container, and resubmit it to `Raptor` to maintain a previously deployed application.

Chapter 3

Related Work

Services, containers, and management of distributed software systems are hardly new topics. Numerous solutions have been proposed and implemented to help software developers deploy their code on heterogeneous, distributed infrastructure. However, these solutions generally only offer a subset of the system features that are most useful when developing services and applications that are expected to be persistent and robust, execute across multiple tiers of hardware, and interface with cyber-physical systems. For example, IoT frameworks focus on compositions of software interfaces but don't address the deployment and lifecycle management of services. Container management tools like Docker Swarm and Kubernetes offer robust service scheduling, replication, and fault-tolerance but do not offer meaningful support in developing software that spans the multiple hardware tiers that are characteristic IoT applications.

Table 3.1 summarizes the features offered by a collection of related work drawn from container and orchestration systems, frameworks for IoT software development, cloud offloading platforms, and historical distributed systems. Each system is described in more detail in the text below. The features included in this table are as follows:

- **Heartbeats:** Monitors the health and liveness of services, emitting periodic messages with diagnostic information
- **Fine-Grained Permissions:** The ability to precisely control who can deploy, view, and monitor services on each physical host
- **Distributed Apps:** Support for deploying multiple, physically distributed services that are composed to form an application
- **Containers:** Support for software containers as a first-class primitive
- **Cloud:** Support for the deployment of code in the cloud
- **On-Premises:** Support for deployment of meaningful local services
- **Clusters:** Abstracts clusters of machines as a single deployment endpoint
- **Resource Management:** Enforces service-specific resource allocations and performs admission control to avoid overloaded hosts
- **Lifecycle Management:** Reacts to service health information to provide useful primitives like automatic restarting
- **Migration:** The ability to dynamically move the execution of code between hosts
- **Decouple Admin from Use:** Separates the administration of infrastructure from its use, i.e., users do not need to understand the infrastructure's internals

	Heartbeats	Fine-Grained Permissions	Distributed Apps	Containers	Cloud	On-Premises	Clusters	Resource Management	Lifecycle Management	Migration	Decouple Admin from Use
AllJoyn			✓			✓					
AWS IoT			✓		✓						
Azure IoT			✓		✓						
Google Cloud Platform			✓	✓	✓						
Microsoft Beam			✓			✓					✓
Kubernetes	✓			✓	✓	✓	✓		✓		
Docker Compose				✓	✓	✓					
Docker Swarm	✓			✓	✓	✓	✓		✓		
Quilt			✓	✓	✓						
Ansible, Puppet, Chef			✓		✓	✓					
Fog Computing, Cloudlets			✓			✓					✓
HomeOS, BOSS			✓			✓					✓
Ninja			✓		✓	✓	✓				✓
Condor						✓		✓			✓
Sprite, Emerald			✓			✓				✓	✓
Sapphire, MAUI, etc.					✓					✓	✓
Apache Jini			✓		✓	✓					
Tuxedo				✓		✓	✓		✓		✓
`Spawnpoint`	✓	✓	✓	✓	✓	✓		✓	✓		✓

Table 3.1: Summary of Features Offered by Related Systems

3.1 Industrial IoT Frameworks

Technology and electronics companies view the Internet of Things as an important emerging market and opportunity for growth. Thus, companies are not only producing IoT devices but also software frameworks and infrastructure to manage these devices, their

deployment, and the data they produce. The first major class of offerings in this space are frameworks like AllJoyn [1]. At its core, AllJoyn is simply an agreement on a common software communications platform (a message bus built on D-Bus with a standard means of interface advertisement and discovery) that allows devices and services offered by different vendors to freely interoperate. AllJoyn offers language bindings for developers to write services and device controllers that interact with the standardized message bus, enabling compositions of independently developed modules. However, AllJoyn does not deal with the issue of secure service deployment and management. `Spawnpoint` is primarily concerned with instantiating services and device drivers in the local environment and thus is orthogonal to frameworks like AllJoyn. `Spawnpoint` is agnostic to the technologies running inside of its managed containers. Developers may in theory use any language or communications platform. Thus, AllJoyn-compliant services could be deployed and run as `Spawnpoint` containers, giving the developer a solution for both code deployment and interoperation.

The second major class of industrial IoT offerings are application frameworks like Amazon's AWS IoT [5], Microsoft's Azure IoT Suite [6], and the IoT tools of the Google Cloud Platform [12]. These frameworks heavily emphasize the cloud, and are primarily intended to integrate edge devices with backends deployed on each company's respective cloud infrastructure. All three companies offer libraries to develop software for edge devices, but these libraries are mainly concerned with providing publish/subscribe communications primitives to ingest device data into the cloud for information processing, aggregation, and storage. Moreover, these industrial IoT frameworks do not confront the issue of deploying and managing units of software across both local and cloud hardware. Google offers a cloud-based container engine that is tightly integrated with Kubernetes, but these three companies do not provide a solution for managed and physically distributed containers. Finally, while Amazon, Google, and Microsoft all advocate local gateways as in an important component of IoT applications [4, 11, 17], these are regarded as little more than proxies for constrained devices or intermediate points for minimal data processing before that data is ultimately fed into a more capable cloud backend, and it is up to the developer to deploy and maintain any locally-running software.

Much like agreements on common APIs like AllJoyn, the IoT platforms offered by Amazon, Google, and Microsoft are somewhat orthogonal to `Spawnpoint`. Their SDKs and the underlying libraries can integrated into software applications that run inside of `Spawnpoint` containers. However, these industrial frameworks are more opinionated on how IoT applications should be architected, advocating minimal logic in edge devices and local proxies that serve only to publish data into vendor-specific cloud backends. `Spawnpoint`, when combined with Bosswave, instead is intended to facilitate the development and deployment of more capable local device drivers that serve to fully encapsulate local devices and, where appropriate, restrict the flow and visibility of data to the immediate physical environment.

Finally, Beam [47] is an IoT applications platform developed at Microsoft Research that restricts its attention to what the authors call "inference-based apps." Here, an applica-

tions developer expresses the data streams they need at a high level, and Beam handles interfacing with the relevant sensors as well as performing any necessary intermediate processing to produce the desired data. Thus, Beam maintains a dataflow graph with sensors as leaves and computational processes carrying out intermediate transformations as internal nodes. The Beam "engines" on which these processes run are analogous to `Spawnpoint` instances. However, Beam and `Spawnpoint` operate at slightly different levels of abstraction. Beam seeks to hide the underlying computational processes that supply a user with their data, controlling and managing them internally. `Spawnpoint` is centered around explicitly deploying and managing these services as distributed containers, and `Raptor` allows the application developer to easily configure these containers and place constraints upon where they are run. Moreover, Beam is focused on use cases that involve the collection and processing of sensor data, while `Spawnpoint` aims to support more general applications, including IoT applications featuring actuation.

3.2 Containers and Their Orchestration

Containers provide a means of encapsulating a software artifact and its execution environment into a portable and reproducible unit. While the notion of a container and the associated idea of lightweight virtualization is not new, the necessary kernel-provided tools and mechanisms — including PID and networking namespaces, chroot jails, and cgroups — have only recently matured and become widely available. Furthermore, container creation and deployment frameworks like Docker [9] and rkt [20] have made containers an extremely popular primitive for packaging and deploying software in recent years.

As developers began to use containers in the cloud and at large scale, particularly within "microservice" architectures [43] and to implement highly available web services, a need for automated container deployment and management emerged. Kubernetes [13,50] is arguably the foremost container orchestration system in use today. It is a framework for deploying containers on large-scale clusters of machines, potentially provisioned in the cloud, and scheduling containers to run on those machines while managing resource consumption and balancing load across hosts. Users specify how applications are deployed as collections of containers in YAML files, and Kubernetes will schedule and instantiate these containers on the cluster accordingly. Containers can be replicated for scaling, and Kubernetes also monitors container status in order to restart failed containers for a "self-healing" effect.

While Kubernetes is very popular, Docker offers its own tools for container orchestration and management that complement the core Docker container engine. Docker machine [10] is concerned with provisioning and administering end hosts that will support containers, but is unconcerned with the containers themselves. Docker compose [16] is a framework for specifying and running an application as a group of containers. However, it can only manage containers within one host and is intended for testing rather than

production purposes. Finally, Docker Swarm [21], is a tool for deploying containers on clusters. It is much like Kubernetes but has fewer features and a smaller user base.

`Spawnpoint` is similar to Kubernetes and Docker Swarm in that it monitors and manages container lifecycles and uses YAML files to express container configuration and deployment. However, `Spawnpoint` interfaces with only a single Docker daemon in order to manage a specific host rather than a cluster of machines. One could imagine, however, deploying an application across multiple `Spawnpoint` instances, which is exactly what the Raptor frontend is intended to facilitate. Moreover, it is possible to swap out Docker for Kubernetes in `Spawnpoint`'s backend, in which case `Spawnpoint` would support a single Bosswave endpoint that maps to an underlying cluster. That is, `Spawnpoint` could materialize a user's services as a Kubernetes pod — a collection of Docker containers that are scheduled onto nodes of a cluster by Kubernetes. Ultimately, the deployment still assumes the form of Docker containers, but Kubernetes acts as an intermediary between `Spawnpoint` and the Docker daemons running on the constituents of the cluster, handling scheduling, replication, and so on.

Finally, Kubernetes and Docker's various tools are generally intended to be used by system administrators, i.e., people with intimate knowledge of how the underlying infrastructure has been provisioned and maintained. `Spawnpoint`, on the other hand, is intended to be used to cast an application onto potentially unknown infrastructure. `Spawnpoint` users interact only with Bosswave URIs, meaning the underlying hosts are completely opaque to these users. For example, switching a `Spawnpoint` instance's backend from Docker to Kubernetes (and upgrading to a cluster of machines) can be fully transparent to end users.

3.3 Quilt

Quilt [19] is a system also developed at UC Berkeley that confronts similar issues of container deployment and management. Where `Spawnpoint` is focused on applications executing in containers locally, on the cloud, or on a combination of the two, Quilt restricts its focus to Amazon EC2 and Google Cloud Engine. Quilt deploys both the containers and also provisions the hosts on which they run, while `Spawnpoint` exposes existing resources for consumption by containerized applications and services. Furthermore, while both Quilt and `Spawnpoint` support networked containers, `Spawnpoint` is primarily designed for containers that communicate over Bosswave. Quilt does not decouple infrastructure administrators from infrastructure users, i.e., those who want to deploy code on this infrastructure, nor does it perform container health monitoring and lifecycle management like `Spawnpoint`. Finally, where `Spawnpoint` users construct a declarative specification of container deployments using the Raptor DSL, Quilt requires an imperative expression of container and host instantiation in JavaScript.

3.4 Other Orchestration Tools

Several other tools are concerned with specifying the configuration of infrastructure at large scale, but they do not support containers and have been discarded as container-based system architectures and deployments have emerged. These include Ansible [2], Chef [7], and Puppet [18]. Each of these is focused on configuring fleets of machines so that they conform to user specifications regarding which packages are installed, which services are running, etc. HashiCorp's Terraform [22] goes one step further and enables code that specifies how transient infrastructure, like cloud-based virtual machines, are provisioned. Like `Spawnpoint` and Raptor, these tools share the vision of declarative system descriptions, but at different levels of the stack. They focus on establishing static infrastructure configuration invariants rather than on exposing the computational resources provided by this infrastructure to a potentially diverse group of stakeholders. Indeed, one could imagine configuring fleets of `Spawnpoint` hosts using something like Terraform or Ansible, but these tools are not suitable replacements for `Spawnpoint` itself.

3.5 On-Premises Computing

While most container orchestration tools place an emphasis on the cloud data-center setting, `Spawnpoint` instances also run on-premises. Services can be executed locally in order to achieve goals like reducing network latency, controlling the flow of sensitive information to the outside world, and so on. Many other efforts have placed a similar emphasis on local computation. Cisco introduced the vision of fog computing [29], where computing infrastructure normally reserved for the cloud is dispersed into the edge of the network, but did not build a system to realize this vision. Similarly, CMU has proposed provisioning on-premises hardware — "cloudlets" — to run computations on behalf of local users and services [46]. `Spawnpoint` is a software system, and could therefore be deployed on this local hardware to enable service instantiation as connected, managed containers.

Software platforms to facilitate local computation have assumed several forms. The first major category are software systems to interface with physical devices for data collection and automation, such as HomeOS [36], and BOSS [35]. Both of these systems, and many like them, were initially designed before the widespread advent of containers, and we see `Spawnpoint` as the execution engine for the next generation of building and home operating systems. Ninja [38] presented a vision for pervasive deployment of services and a software platform to support interactions between heterogeneous devices well before there was the notion of an Internet of Things. It shares many similarities with `Spawnpoint`, most importantly the notion of a shared infrastructure serving as a substrate for services that are implemented and used by a diverse group of stakeholders. `Spawnpoint` takes advantage of several new technologies that have emerged since the development of Ninja, the most important being lightweight containers to package and distribute software.

3.6 Distributed Systems

Many historical distributed systems also aim to construct a software execution platform that pervades the local area. In Condor [44], resources are "scavenged" from idle workstations to support the execution of background jobs, much like `Spawnpoint` allows services to be cast onto infrastructure within containers. Sprite [37] and Emerald [40] allow computational processes to migrate in order to balance load and achieve fault tolerance. `Spawnpoint` has basic support for migration (it is triggered manually and is meant for stateless services), but one could imagine implementing more sophisticated migration on top of `Spawnpoint`'s primitives.

With the advent of cloud computing, new systems emerged to provide execution platforms where applications running on weaker edge devices could seamlessly offload their demanding computations onto more capable cloud-based servers [32, 34, 45, 52]. These differ from `Spawnpoint` in that services and computations are dynamically cast onto the infrastructure during execution, whereas `Spawnpoint`'s containers are placed and provisioned ahead of time. However, `Spawnpoint` could easily be used in conjunction with these kinds of systems by managing the infrastructure that serves as a target for offloading. For example, an application running on a mobile phone could contact a `Spawnpoint` instance to launch a container that runs backend computations on its behalf.

`Spawnpoint` also has similarities to service-oriented architectures developed in industry. Apache Jini (now Apache River) [3], is a Java platform for creating distributed systems out of a federation of potentially distributed services. However, River is mainly focused on service discovery and communication rather than systematic and secure service deployment. Oracle's Tuxedo [15] is a software framework for deploying services, each within an isolated execution environment, onto a cluster of machines. These services can be transparently replicated for scaling and fault tolerance purposes, much like containers in Kubernetes. In addition, Tuxedo provides an event broker so that services can react to changes in system state and communicate with one another. While `Spawnpoint` offers a similar set of features, it does not restrict itself to clusters, and it features a more decentralized and fine-grained permissions model with respect to the deployment and monitoring of services, enabled by its use of Bosswave.

Chapter 4

Implementation

This chapter begins a detailed discussion of the design and implementation of Spawnpoint. Here, we focus on spawnd, Spawnpoint's host management daemon, and defer the discussion of Spawnpoint front-ends to Chapter 5. spawnd is responsible for advertising the computational resources and current state of a host, accepting commands to deploy and manipulate service containers on that host, and manging these services while they are running. First, we discuss the external interface presented by spawnd through Bosswave, then we detail how service configurations and deployments are specified. Next, we discuss Spawnpoint's use of Docker to create and manage containers. Docker also plays a role in spawnd's fault tolerance, dicussed in the following section. Finally, we detail service lifecycle management and present the overall architecture of spawnd.

4.1 External Bosswave Interface

Spawnpoint exclusively uses Bosswave for external communications. The integration between the Spawnpoint host daemon (spawnd) and Bosswave is depicted in Figure 4.1. On each Spawnpoint host, spawnd operates in tandem with a second persistently running process, a Bosswave agent. spawnd communicates with the agent over a purely local network socket, issuing syndication commands to publish messages and to receive messages issued to specific endpoints. These commands are expressed in a simple, plain-text "out of band" protocol specified by Bosswave. The agent then performs all of the necessary work on behalf of spawnd, signing, encrypting, and sending messages for publication as well as receiving, decrypting, validating, and delivering messages matching spawnd's open subscriptions. Bosswave provides bindings to wrap all interactions with the local agent in the form of a Go library.[1] Interested readers are referred to Bosswave-related papers and repositories for more information [24, 25].

Spawnpoint's integration with Bosswave provides a number of security guarantees to the system without complicating its implementation. First, the Spawnpoint interface provided to external users and services is broken up into a fixed collection of publish/-subscribe topics, expressed as URIs, that each form an independent permissions domain. In other words, operations to send or receive a message on a particular URI are always mediated by Bosswave, which ensures that only authorized entities may carry out such operations, and permissions on a given topic's URI are independent of those for any other topic's URI. In particular, Spawnpoint subscribes to a collection of topics in order to accept remote commands from users and services. Bosswave's security model ensures that only authorized entities may issue a command (by publishing a message to the corresponding URI) and conversely that only the authorized Spawnpoint daemon may receive the command (by subscribing to that URI). Each Spawnpoint instance is configured with a base URI, which is a prefix for all of the topic names that the Spawnpoint daemon subscribes to. Equivalently, when we view URIs as items in a hierarchical namespace, this is the root of the URI subtree that the Spawnpoint daemon uses.

[1] https://github.com/immesys/bw2bind

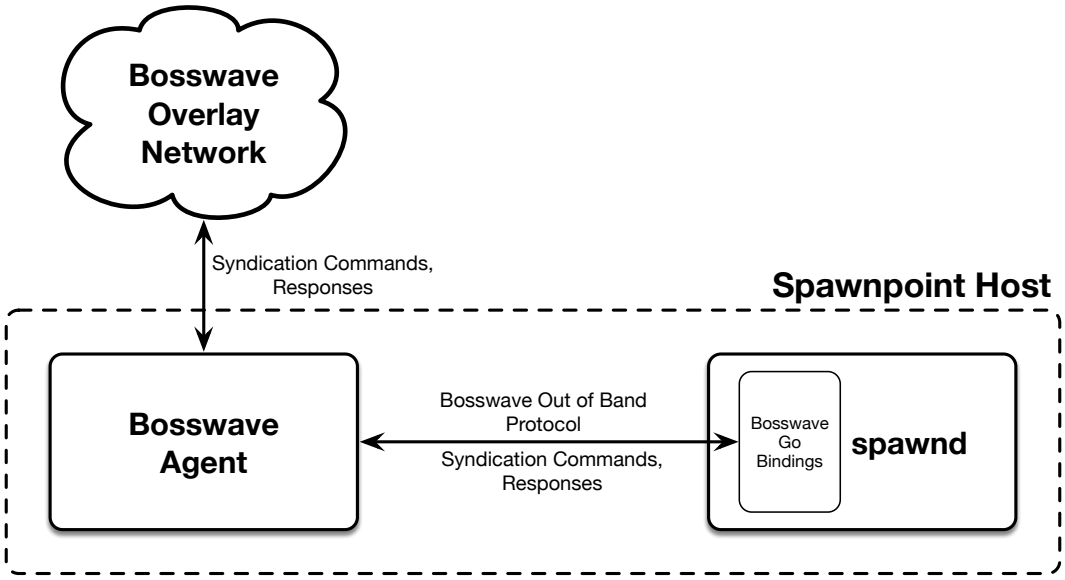


Figure 4.1: `spawnd`'s integration with Bosswave

The set of URI topics that a `Spawnpoint` subscribes to is:

- `bw2://<base_uri>/s.spawnpoint/server/i.spawnpoint/slot/config`
- `bw2://<base_uri>/s.spawnpoint/server/i.spawnpoint/slot/restart`
- `bw2://<base_uri>/s.spawnpoint/server/i.spawnpoint/slot/stop`
- `bw2://<base_uri>/s.spawnpoint/server/i.spawnpoint/slot/logs`

These URIs adhere to a standard set of structural patterns that are meant to make them self-describing and to facilitate interoperation among Bosswave-based services. More concretely, we can see that these URIs are interface endpoints for the `s.spanwpoint` service, and while a service can have multiple interfaces, here we are only concerned with aninterface named `i.spawnpoint`. Topics that are intended to serve as channels for input to a service are dubbed "slots," and here each slot is used to receive an external command to perform a specific operation:

- An entity deploys a new service on a `Spawnpoint` instance by publishing a YAML configuration document (detailed in the next section) to the `config` slot.
- An entity restarts an existing service by specifying its unique name in a message published to the `restart` slot.
- An entity stops an existing service by specifying its unique name in a message published to the `stop` slot.

- An entity retrieves the log output of a service by publishing its unique name and a log start time in a message on the `logs` slot.

Because each of these topics, i.e. slots, forms an independent permissions domain, `Spawnpoint` enables users and services to be assigned rich combinations of access privileges. The ability to perform any `Spawnpoint` operation is simply contingent upon the initiating entity's permissions to publish a message on the corresponding slot. For example, a systems administrator can be granted permissions to publish on any slot, which allows them to deploy, restart, stop and log any service. A regular user might be allowed to restart or log existing services but not to start or stop them. An external logging service could be given the ability to read service logs but not to instantiate or manipulate any services.

Just as slots are Bosswave channels for input to a `Spawnpoint` instance, "signals" are channels for output from the instance. The daemon will periodically publish a heartbeat to the URI:

```
bw2://<base_uri>/s.spawnpoint/server/i.spawnpoint/signal/heartbeat
```

These messages advertise the liveness of the instance to all entities with permissions to subscribe to the signal's URI. Thus, the permissions on this URI can be used to restrict the visibility of a `Spawnpoint` daemon to a specific set of entities. Furthermore, these messages not only establish liveness but also contain information on the current availability of resources at the `Spawnpoint`. This allows users and external services to determine if a given `Spawnpoint` can accommodate the resource requirements of a new service, were it to be deployed there. Additionally, `Spawnpoint` automatically publishes heartbeats for each of the services it is running to signals of the following form:

```
bw2://<base_uri>/s.spawnpoint/server/i.spawnpoint/signal/
    heartbeat/<service_name>
```

Like the heartbeats for the `Spawnpoint` daemon itself, these heartbeats go beyond establishing liveness. They also contain useful diagnostic information about the container hosting the service, such as how much memory and CPU it is consuming and statistics about its IO operations. Also like the daemon's heartbeats, service visibility is controlled by permissions to subscribe to messages on the corresponding heartbeat signal URI. All heartbeat messages, for both the daemon and the services it runs, are serialized using MessagePack.[2] This is similar to JSON but uses a binary format rather than plain text for more compact representations and faster processing.

The complete set of URIs that are involved in routine `Spawnpoint` operations are diagrammed in Figure 4.2. There is one potential issue with this configuration. Each `Spawnpoint` instance features just one slot URI for restarting, stopping, or logging any service. This means that, for an entity to have permission to manipulate any service on

[2] `https://msgpack.org/index.html`

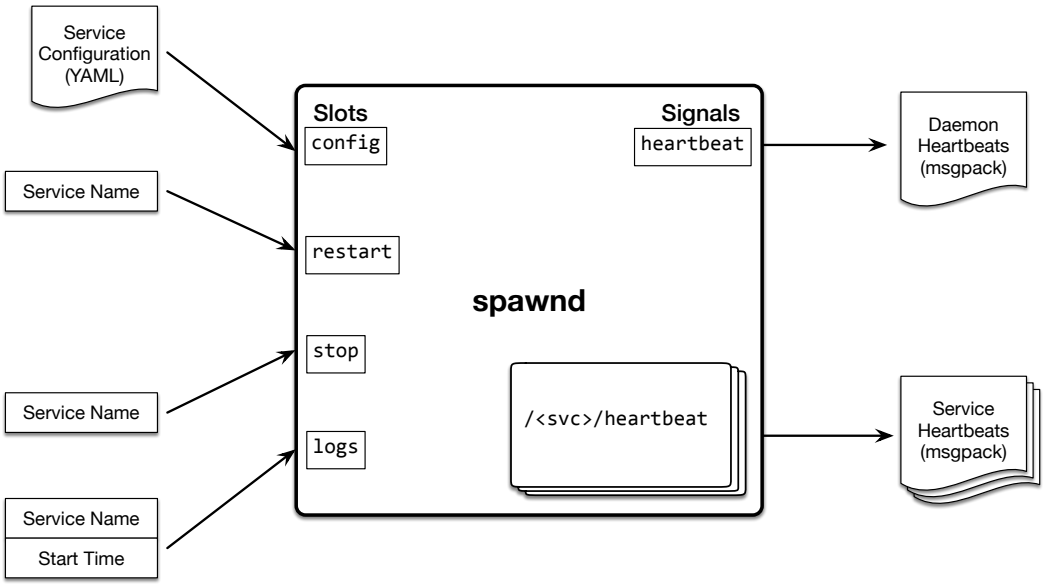


Figure 4.2: The external interface of a `Spawnpoint` daemon, presented through Bosswave

a `Spawnpoint`, it will have permissions to manipulate all services on that `Spawnpoint`. This is not a fundamental limitation of the implementation, but rather a design decision based on the idea that the users of a shared `Spawnpoint` have some degree of mutual trust. However, it would be easy to create unique slots for each service that accept external commands for restarting, stopping, or logging that service. A second approach, which is in fact already a feature of the current `Spawnpoint` implementation, is to create the illusion of multiple `Spawnpoint` instances from a single daemon process simply by subscribing to distinct collections of signals and slots, with different base URIs. Hence, a single `Spawnpoint` daemon can expose distinct domains of trust, and users on each of these domains are not allowed to access services deployed on other domains.

4.2 Daemon and Service Configuration

All `Spawnpoint` configuration settings are specified in YAML files. YAML was chosen because its syntax is flexible and easily readable while still amenable to machine parsing and because it is extensively used by popular tools like Kubernetes. A `Spawnpoint` daemon is described by two YAML files. The first specifies the following configuration parameters:

- `path`: The base URI for all Bosswave slots and signals (as described above)
- `entity`: The `Spawnpoint` daemon's identifying key-pair for all Bosswave operations.
- `localRouter`/`containerRouter` The Bosswave agent to connect to and use for all messaging operations, both for the daemon itself and the containers it hosts.

- `memAlloc`: The total amount of memory to make available to running containers. Each container then allocates a slice of this pool for itself.
- `cpuShares`: The number of CPU cores to make available to running containers, expressed in shares. As with memory, this is a pool that is allocated among the containers.
- `allowHostNet`: A flag for container host networking. If enabled, containers may use the host's networking stack directly. This is intended to be very rare as it represents a security risk.

An example configuration file might look like the following:

```
path: ucberkeley/spawnpoint/alpha
entity: alpha.key
memAlloc: 16G
cpuShares: 8192
localRouter: "172.17.0.1:28589"
containerRouter: "172.17.0.1:28589"
allowHostNet: false
```

The second YAML file for a `Spawnpoint` daemon specifies the metadata key/value pairs that are advertised through Bosswave. For example:

```
arch:     "amd64"
location: "Soda Hall"
owner:    "Allan Turing"
```

Service configuration involves more fields, many of which are optional or most often left to their default values. The most important fields are as follows. Many of the parameters relate to the container build process, which is described in detail in the next section.

- `image`: The Docker image to use to create the service's container. This defaults to an Ubuntu Linux container with tools like Go and Bosswave already installed.
- `entity`: A key pair to copy into the container that will serve as the identifier for any Bosswave operations performed by its corresponding service
- `source`: An optional URL for source code to check out use as part of the container build process
- `build`: A sequence of shell commands to run to initialize the container (e.g. compiling checked out source code)
- `run`: The container's entrypoint, i.e. a sequence of commands to launch the necessary software, such as invoking a binary

- `memAlloc`: The container's required memory, which is allocated from the `Spawnpoint` daemon's pool
- `cpuShares`: The container's required CPU cores, expressed as shares, also allocated from the daemon's pool
- `autoRestart`: When explicitly set, this flag triggers an automatic restart of the container when it terminates (normally or abnormally)
- `restartInt`: Specifies a delay interval that will precede any automatic container restarts, if they are enabled
- `includedFiles`/`includedDirs`: Files and directories to be copied into the container.

A typical service configuration file might be written like so:

```
entity: demoDev.ent
source: "git+http://github.com/jhkolb/demosvc"
build: ["go get -d", "go build -o demosvc"]
run: [./demosvc, 60]
memAlloc: 512M,
cpuShare: 512,
includedFiles: ["params.yml",]
```

Here, we see that a new container will be created from the default, Ubuntu-based image.[3] The service will be instantiated with source code from `http://github.com/jhkolb/demosvc`. Because this is code written in Go, the two commands listed in the `build` parameter are required to compile a binary, which is then invoked (with a command line argument) in `run`. In `Spawnpoint`, service developers must state the memory and CPU requirements of their containers upfront, before they are launched. In this example, the container will be allowed to use up to 512 MiB of memory and 512 CPU shares, which translates to half of a CPU core.

The last line of the configuration specifies that a single file, `params.yml` is copied into the container. Note that this file is not drawn from the container's host, i.e. the `Spawnpoint` daemon's host machine, but rather from whatever machine is running the `Spawnpoint` front-end that initiates the service deployment. `Spawnpoint` front-ends are not coupled to any particular instance of a `Spawnpoint` daemon. Additionally, inclusion of a YAML parameters file is a fairly common idiom of use for services deployed on `Spawnpoint`. This is because a service cannot be completely characterized by its normal configuration file. There are certain parameters, such as API keys and unique device identifiers, that are particular to one instance of a service but not to the service in general. While the service configuration file might be kept under version control and reused, a unique version of a separate parameters file is created for each instance of the service that is deployed.

[3] Available at `https://hub.docker.com/r/jhkolb/spawnpoint`

4.3 Container Management with Docker

Spawnpoint internally relies on Docker for container creation and management. Spawnpoint can thus be viewed as a system to manage and host Docker containers and to elevate them to the status of first-class citizens within a secure message bus, i.e. Bosswave. This arrangement has several advantages. First, it simplifies the implementation of Spawnpoint, as it need only focus on issues like exposing an interface over Bosswave, managing container lifecycles and resource consumption (discussed in the next section), and achieving fault tolerance, rather than providing a fully-featured container system. Moreover, Spawnpoint leverages Docker's ancillary tools and infrastructure. Users who do not wish to use the stock Spawnpoint container image can create their own with Docker's standard, well-documented tools and host these images on Docker's repository hub. Also, Spawnpoint is not permanently tied to Docker, which is treated as an isolated back-end that could easily be swapped out for a different container toolchain. For example, Spawnpoint could work just as well with rkt or even with Kubernetes to serve as an abstraction over a cluster of machines.

Container Initialization

When a Spawnpoint daemon receives a new service configuration from a user or external service, it first verifies that it has sufficient resources to accommodate the new container (discussed below) and then needs to create and start a new container in accordance with the configuration's specifications. First, Spawnpoint obtains the necessary Docker image, which defaults to an Ubuntu-based and Bosswave-enabled container, but can be any image of the user's choosing. Images that are available locally are reused if possible, otherwise they are pulled from a remote repository. Next, Spawnpoint creates a temporary "Dockerfile" — essentially a build file that specifies how the container will be instantiated from its base image. This is where Spawnpoint instructs Docker to check out and incorporate source code from a user-specified repository, copy in the necessary external files, run a specific command to to start the container, and so on. Finally, Spawnpoint passes the Dockerfile to Docker itself and waits either for notification of a successful container build or an error. The full container instantiation process is diagrammed in Figure 4.3.

Container Management

Once a container has been launched, Spawnpoint oversees it through Docker's monitoring capabilities. The Docker daemon will publish events through a REST API on a Unix domain socket, which Spawnpoint connects to and monitors. Because Docker relies on cgroups to track and potentially restrict container resource consumption, it also emits detailed information on the status of running containers. Thus, Spawnpoint's task is to observe the information published by Docker, publish status information about the containers for services it is actively managing, and react appropriately to changes in container

state. `Spawnpoint` periodically samples container container CPU and memory consumption and includes this information in the heartbeats it publishes to establish service liveness. When Docker notifies `Spawnpoint` about a container termination, `Spawnpoint` will either release the corresponding resource allocations back to its central pool, or it will automatically restart the container if this was enabled in the service's configuration.

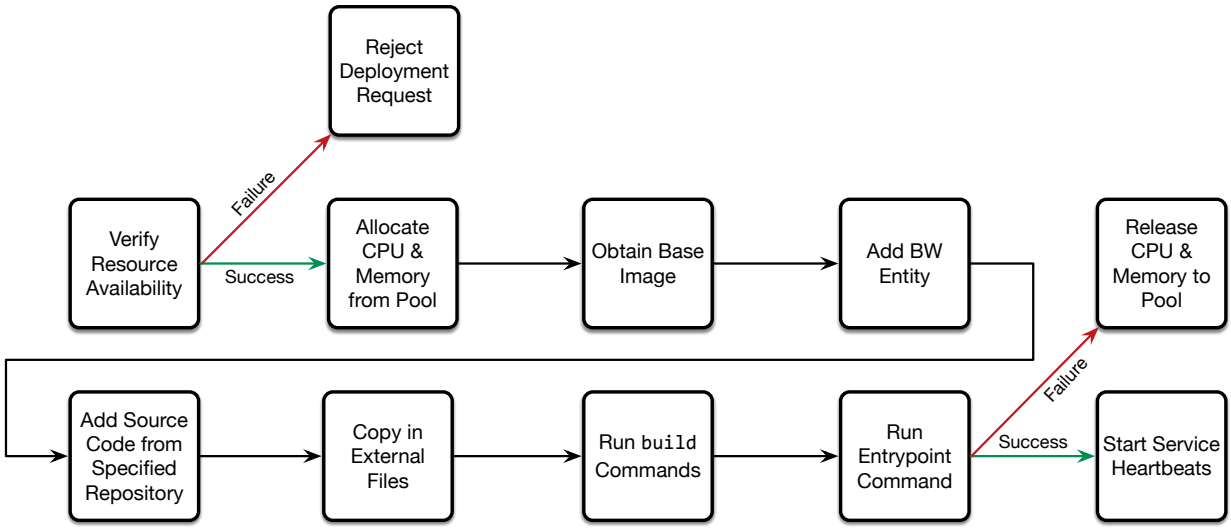


Figure 4.3: The steps to instantiating a new service on `Spawnpoint`

4.4 Fault Tolerance

As the previous section made clear, one way to view `Spawnpoint` is as a Docker client that wraps containers and their deployment in a management layer with a Bosswave interface. Aside from the advantages of simplifying the implementation and achieving compatibility with the plethora of Docker tools in use today, this leads to another important benefit: Docker's daemon serves to decouple `Spawnpoint` from the services that it manages. Importantly, failures in `Spawnpoint` are not tied to failures in its managed services, and vice versa. This is because service containers are independently executing processes that can continue running without `Spawnpoint` present — they simply won't be managed or monitored. In fact, `Spawnpoint` itself can run as a normal Docker container, in a sense putting it on equal footing with the services it manages. The Docker daemon interacts with its clients through a RESTful API over a UNIX domain socket. The `spawnd` container is configured to have the socket's endpoint mapped from the host's file system into its own file system, allowing it to issue commands to the Docker daemon and manage the execution of other containers on the host.

This has important consequences for the design and fault tolerance of the `Spawnpoint` daemon. Because the daemon may fail, it also must recover gracefully. In particular,

when the daemon is restarted, it should cleanly reassume ownership of containers that it instantiated in the past. This breaks down into three distinct cases:

1. A service has continued to run while `Spawnpoint` was down. The `Spawnpoint` should resume management of this container, restore its resource allocation from the central pool, and resume emitting heartbeats for the service.

2. A service that was running when `Spawnpoint` failed has since terminated, and its configuration has automatic restarts enabled. `Spawnpoint` should restore its resource allocation and restart the container. Once the restart is complete, `Spawnpoint` should publish heartbeats for the service.

3. A service that was running when `Spawnpoint` failed has since terminated, but it does not have automatic restarts enabled. `Spawnpoint` should free the service's resource allocation but otherwise take no action.

To achieve this behavior, and because it could potentially fail at any time, the `Spawnpoint` daemon periodically saves a snapshot of its current state to persistent storage. This state includes a detailed manifest for each service currently managed by the `Spawnpoint` instance — with information like the user's specified configuration, Docker's unique identifier for the service's container, and so on — as well as a record of all resource allocations granted by the daemon to its services. When a `Spawnpoint` daemon is started, it consults this snapshot, queries the Docker daemon for information on all running containers, and reconciles the two using the procedures laid out in the list above. This allows `Spawnpoint` to restart cleanly, picking up where it left off as well as possible and leaving long-running services undisturbed by its own failures.

The combination of persisted snapshots and independent service execution dramatically reduces the cost of execution failures in the `Spawnpoint` daemon. The obvious result is that `Spawnpoint` can cleanly restart and recover upon failure, helping it more reliably manage its services and enhancing its availability to end users. However, there are other interesting use cases for this behavior. For example, in-place updates are now possible where a new version of `Spawnpoint` is deployed without disturbing running services. The original `Spawnpoint` daemon is intentionally shut down, the new version of the software is deployed, and the new `Spawnpoint` daemon starts up, reading the snapshot from the older version and resuming management of all hosted services. Note that this does require a consistent format for the persisted data that is understood by both versions of `Spawnpoint`, but this is a small cost to pay compared to the benefits of in-place software updates.

4.5 Resource and Service Lifecycle Management

Service Lifecycle

The lifecycle of a `Spawnpoint` container forms a finite state machine, shown in Figure 4.4. By monitoring events published by the Docker daemon, `Spawnpoint` is made aware when a container transitions to a new state and can then modify the relevant resource allocations. When `Spawnpoint` deploys a new service, it first must take the time to complete the container creation process described above. After it is created, a container transitions from the "Booting" state to the "Running" state, and its execution is independent of the `Spawnpoint` daemon, as described above. Once a running container terminates, the rest of its lifecycle depends on how the service has been configured. If automatic restarts are enabled, the container will be restarted by `Spawnpoint` without any user intervention. This continues until an external user or service sends an explicit request to stop the container. After a container enters the "Stopped" state, either because its underlying process has terminated or in response to a request, it becomes a zombie — it is neither dead nor alive. If `Spawnpoint` receives a request to restart the container, it is brought back up with the same configuration, and there is no need to go through the time-consuming container creation process a second time. However, if a timeout expires (the duration of which is a configuration parameter of the daemon), then the container becomes dead. `Spawnpoint` purges its configuration, and the service must be explicitly deployed again from scratch.

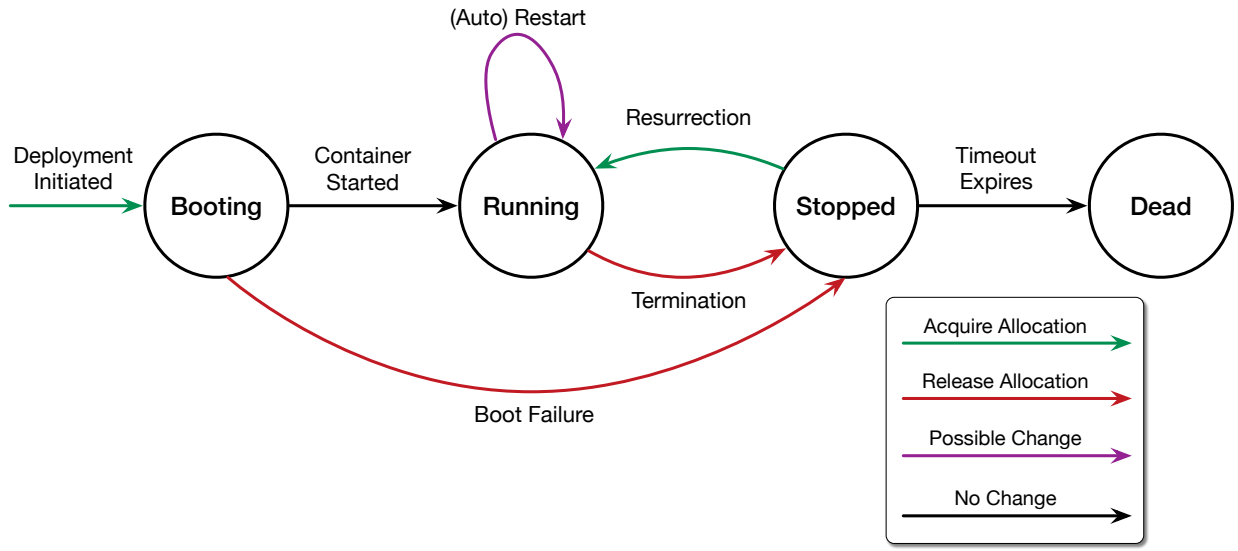


Figure 4.4: A state machine representation of a `Spawnpoint` container's lifecycle and the management of its resource allocations

Restricting Container Resource Consumption

Through its use of `cgroups`, Docker can place limits on container memory and CPU consumption if instructed to do so. This achieves a measure of performance isolation, albeit with much weaker guarantees than a strategy based on full virtualization. More importantly, this prevents containers with faulty code from monopolizing the host's resources. One of the main tasks of the `Spawnpoint` daemon is to use this mechanism to allocate resources to individual containers while ensuring that the host does not become overburdened by the containers in aggregate. A `Spawnpoint` daemon is configured with a cap on the total amount of memory and CPU that its containers may use in total. The memory cap is simply expressed in megabytes or gigabytes, while the CPU cap is expressed in shares. In a naive Docker environment, containers are subject to relative scheduling — their shares dictate what proportion of host CPU resources they may use. With `Spawnpoint`'s cap on the total number of shares however, this becomes absolute scheduling. In terms of both memory and CPU, then, containers are allocated a portion of a shared pool maintained by the `Spawnpoint` daemon, and these resource allocations are acquired, released, and modified in reaction to changes in container state.

`Spawnpoint`'s approach to the management of the shared memory and CPU pools is relatively simple, but it is easy to reason about and can easily be swapped out for more sophisticated approaches in future versions of the system. First, `Spawnpoint` takes a conservative approach to admission control for new services. When a new service is submitted for execution, its configuration must state its resource requirements upfront, and `Spawnpoint` immediately allocates those resources to the new service from the global CPU and memory pools. If either pool contains insufficient resources, then the service deployment is rejected, and it will not be allowed to run on the host.

Managing Resource Allocations

Once a container is started, `Spawnpoint` watches for changes in its state and updates the container's resource allocation accordingly. This involves several subtleties because the `Spawnpoint` daemon must now manage resources allocated to many simultaneously and independently executing containers, each of which can potentially fail at any time, all while preventing race conditions and maintaining consistency in allocations from the shared CPU and memory pools. A service container is immediately allocated the necessary resources when the request for its deployment is submitted to the daemon, i.e., when the container enters the "Booting" state shown in Figure 4.4. If the initialization of a new container fails at any point, its resource allocation is released when the container transitions to the "Stopped" state. Similarly, a container's allocated resources are returned to their respective pools when the container terminates.

Handling container restarts is more nuanced. A restart cannot be treated as a termination and subsequent boot of the same container because it introduces a race condition. If the container's termination were treated normally, its resource allocations would be

released, allowing a new service to acquire them before the container is restarted. Then, the original container's reacquisition of its original allocations would be rejected, preventing it from restarting. Thus, when a container is restarted, either automatically upon termination or when explicitly requested through `Spawnpoint`'s `restart` slot, container termination and booting is treated as an atomic operation, as reflected by the self-loop involving the "Running" state in Figure 4.4's finite state machine.

The situation is further complicated by the fact that a user can request the deployment of a service that is already present on a `Spawnpoint` instance, which essentially becomes a restart. This may be done to prompt an update to the container, either in terms of the code it executes or in terms of its configuration. Therefore, a request can be made to restart a service but with *different* CPU and memory allocations than before. If the allocation needs to shrink, the unused portion of the service's allocation is returned to the pool. However, if an allocation needs to grow, `Spawnpoint` must validate that this is possible before initiating the restart.

4.6 Complete System Architecture

The `Spawnpoint` daemon is expected to perform a number of tasks simultaneously: monitoring the slots of its Bosswave interface for incoming commands, maintaining pools of shared resources, reacting to changes in container state, and publishing heartbeats. `Spawnpoint` is implemented in Go [23], which provides a set of tools that have proven very well suited to the challenge of implementing this kind of system. Concretely, the `Spawnpoint` daemon cleanly decomposes into a set of lightweight userland-schedule coroutines, a first class primitive in Go, which emphasize communication through shared message channels rather than shared memory. Each task within the daemon roughly corresponds to a coroutine or set of coroutines:

- One coroutine handles all communications on each slot of `Spawnpoint`'s Bosswave interface.
- One coroutine interfaces with the host's Docker daemon, forwarding events to other coroutines as necessary.
- A coroutine is devoted to publishing heartbeat messages to advertise the daemon's liveness and current status.
- One coroutine periodically snapshots the daemon's state and saves it to persistent storage to enable clean recovery from restarts.
- Each managed service is backed by three coroutines: one to periodically publish heartbeats, another to tail the container's log entries and publish them to a Bosswave signal, and a third to instantiate the state machine depicted in Figure 4.4.

These coroutines, as well as the communication that occurs among them, are shown in Figure 4.5.

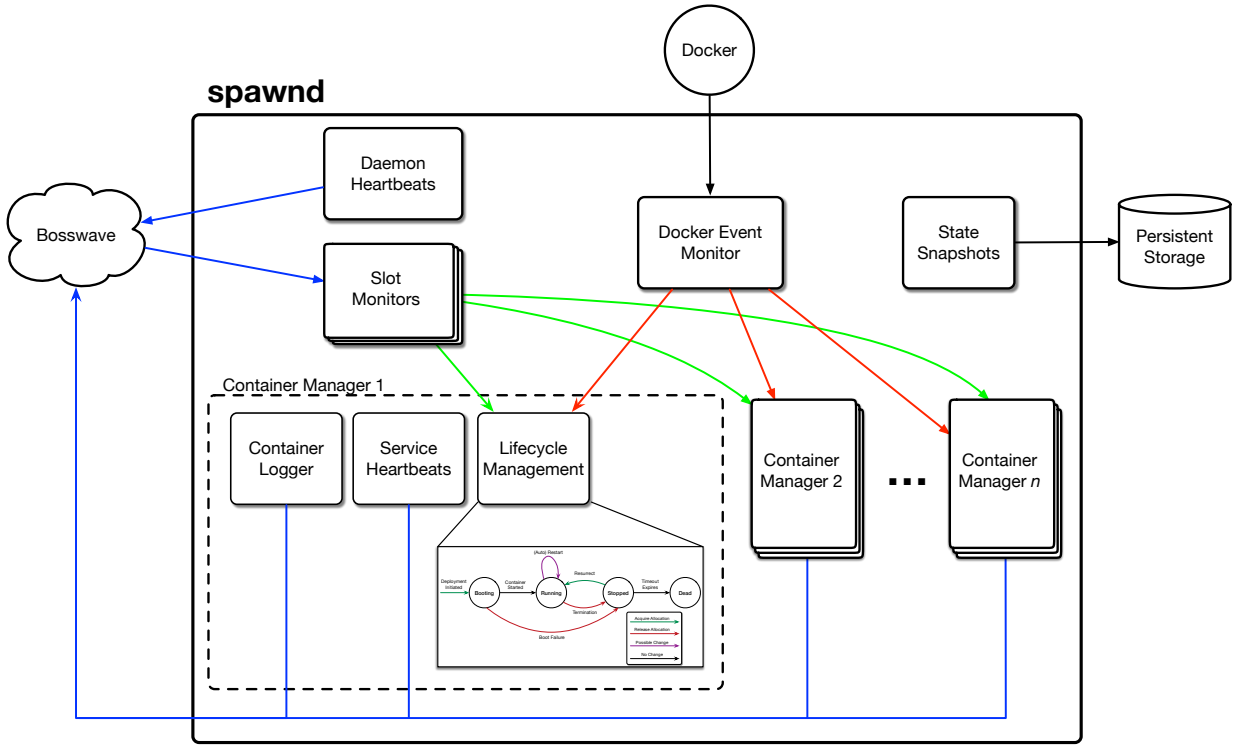


Figure 4.5: The architecture of a `Spawnpoint` daemon, in terms of its underlying coroutines, external software components, and the communication between them

The realization of the state machine from Figure 4.4 as a single coroutine merits further discussion. In a sense, this set of coroutines, with each element corresponding to a running service, is the backbone of the `Spawnpoint` daemon. All other coroutines serve roles that are ancillary to this core set. For example, the coroutine that manages the daemon's `config` slot is tasked with adding new coroutines to this set that will initialize and then manage a new service. The coroutines for the `restart` and `stop` slots simply forward incoming commands to the state machine management coroutine for the relevant service, much like the coroutine to interface with Docker forwards events signalling transitions in a container's lifecycle. This approach has significantly simplified the internal design of the daemon. It centralizes the logic for service management in one place. More importantly, any transition in a service's state, which may correspond to allocating or freeing its associated resources, is prompted by a message sent to its underlying mnagement coroutine. Because all changes in service state and resource allocations occur in one place, they become linearized. A service's management coroutine can enforce much of its consistency semantics internally and without expensive synchronization, as there is no other thread

of execution that manipulates the same collection of state. All that this coroutine needs to do is move between consistent states upon receiving each incoming message.

Chapter 5

System Front-ends

While its host and container management daemon forms the bulk of its complexity, `Spawnpoint` also includes three frontends for interaction with instances of these daemons. The first is `spawnctl`, a command line tool that allows a user to search for and inspect spawnpoints and to deploy and manage services. The second front-end is a "wavelet" — a graphical application built using a modified version of Qt with native Bosswave support — that enables monitoring and interaction with a specific Spawnpoint. Finally, `Raptor` is a front-end intended to facilitate the deployment of complex multi-container and multi-host applications. It consists of a DSL to describe these applications and a command line tool to parse a description written in this DSL and deploy the necessary containers. However, in reality any program that can publish and subscribe using Bosswave can interact with a `Spawnpoint` daemon. A Go library, `spawnclient`, wraps publications and subscriptions in more convenient functions that perform the necessary validation and serialization, and thus enables any Go program to interact with `Spawnpoint`. In fact, `spawnctl` is little more than a command line wrapper for interactive use of this library, and `Raptor` uses the same library after parsing the user's application specification. If more flexibility is required, a program can bypass this library and publish or subscribe manually.

5.1 `spawnctl`

`spawnctl` is the primary means of interacting with running `Spawnpoint` daemons. Its command line interface is meant to be straightforward and flexible. `spawnctl` serves two major purposes. The first is to search for and inspect `Spawnpoint` daemons as well as the services they are hosting. `spawnctl` can scan for any `Spawnpoint` instances that are operating on a particular subtree of Bosswave's URI-based topic space. Frequently, scans are in fact performed on an entire Bosswave namespace. `spawnctl` then provides the user with a list of such `Spawnpoint` daemons, as well as some basic information about them, as shown in Figure 5.1a. `Spawnpoint` users have typically performed scans either to find a suitable host for their services on a particular namespace, or to check the health of running daemons. When a scan finds only one daemon instance on the specified URI subtree, `spawnctl` produces more detailed diagnostic information, including a list of the services it hosts and their respective states, as shown in Figure 5.1b. Additionally, a user may also inspect a specific service to view information about its resource consumption and configuration. As shown in Figure 5.2, the user must specify the URI of the daemon hosting the service as well as the service's unique name.

The second major purpose of the `spawnctl` tool is to deploy or manipulate services from the command line. To deploy a new service, the user must specify the URI of the host `Spawnpoint` daemon, the YAML configuration file for the service, and a unique name for the service. `spawnctl` then tails the daemon's logs for the service as it is starting and as it executes, until the user exits. This allows a user to determine if the service was correctly booted and if not, to diagnose what went wrong. An example service deployment is shown in Figure 5.3. Once a service is running, `spawnctl` can be used to tail its logs

```
jack@cloakroom:~$ spawnctl scan ucberkeley
Discovered 4 SpawnPoint(s):
[showroom] seen 08 Feb 17 03:45 PST (80ms) ago at nvrnSE4pJe4ZMO3WQdb-EPi5iwuzmTVUpk6XNNRGYsc=/eop/spawnpoint/showroom
  Available Memory: 6144 MB, Available Cpu Shares: 2048
[castle] seen 01 Feb 17 11:56 PST (159h48m59.16s) ago at nvrnSE4pJe4ZMO3WQdb-EPi5iwuzmTVUpk6XNNRGYsc=/spawnpoint/castle
  Available Memory: 16384 MB, Available Cpu Shares: 8192
[spawnpoint] seen 08 Feb 17 03:45 PST (20ms) ago at nvrnSE4pJe4ZMO3WQdb-EPi5iwuzmTVUpk6XNNRGYsc=/sasc/spawnpoint
  Available Memory: 2048 MB, Available Cpu Shares: 2048
[alpha] seen 08 Feb 17 03:45 PST (9.44s) ago at nvrnSE4pJe4ZMO3WQdb-EPi5iwuzmTVUpk6XNNRGYsc=/spawnpoint/alpha
  Available Memory: 11264 MB, Available Cpu Shares: 3072
```

(a) Results from a `spawnctl` scan when multiple daemons are found on one URI subtree

```
jack@cloakroom:~$ spawnctl scan ucberkeley/eop/spawnpoint/showroom
Discovered 1 SpawnPoint(s):
[showroom] seen 08 Feb 17 02:59 PST (1.78s) ago at nvrnSE4pJe4ZMO3W
  Available Memory: 6144 MB, Available Cpu Shares: 2048
Metadata:
  • lastalive: 2017-02-08 10:59:47.014198907 +0000 UTC
Services:
  • [venstar] seen 08 Feb 17 02:59 PST (780ms) ago
      Memory: 5.18/512 MB, CPU Shares: ~0/512
  • [lifx] seen 08 Feb 17 02:59 PST (4.73s) ago
      Memory: 6.72/512 MB, CPU Shares: ~0/512
  • [echola] seen 08 Feb 17 02:59 PST (5.77s) ago
      Memory: 7.81/512 MB, CPU Shares: ~0/512
  • [tp-link-plug] seen 08 Feb 17 02:59 PST (1.77s) ago
      Memory: 9.82/512 MB, CPU Shares: ~0/512
```

(b) Results from a `spawnctl` scan when only one daemon is found on the specified subtree

Figure 5.1: Scanning for running daemons with the `spawnctl` tool

or to retrieve old log entries. A service's logs contain anything that the service itself logs or prints to `stdout` and `stderr` as well as status information logged by the daemon regarding the service. Finally, a user may restart or stop a running service with `spawnctl` by specifying the host `Spawnpoint`'s base URI and the unique name of the service. An example of a service restart is shown in Figure 5.4.

5.2 Wavelet

"Wavelets" are graphical applications built around publishing and subscribing to Bosswave channels. They are implemented using a modified version of the Qt framework that includes first-class support for Bosswave. `Spawnpoint` includes one such Wavelet as a graphical frontend. Unlike `spawnctl` however, the user must specify a particular daemon instance when launching the app, which is then bound to that instance for the remainder of its execution. The GUI includes three main tabs, each devoted to a particular function. The first tab, shown in Figure 5.5a, is for viewing the status of the daemon, much like

```
jack@cloakroom:~$ spawnctl inspect -u ucberkeley/eop/spawnpoint/showroom -n lifx
[lifx] seen 08 Feb 17 03:01 PST (3.3s) ago at ucberkeley/eop/spawnpoint/showroom
CPU Usage: 0/512 Shares
Memory Usage: 6.5703125/512 MiB
Original Configuration File:
    entity: lifx.ent
    container: jhkolb/spawnpoint:amd64
    build: [go get github.com/SoftwareDefinedBuildings/bw2-contrib/driver/lifx]
    run: [lifx]
    memAlloc: 512M
    cpuShares: 512
    autoRestart: false
    includedFiles: [params.yml]
```

Figure 5.2: Inspecting a specific service with `spawnctl`

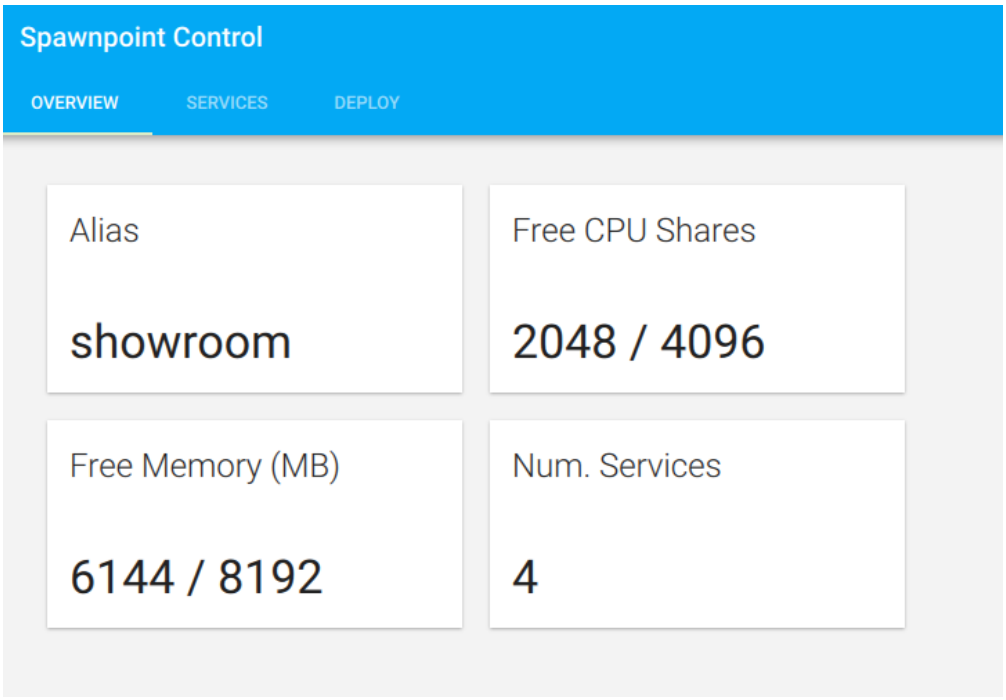
```
jack@cloakroom:~/goWork/src/github.com/immesys/spawnpoint/spawnctl$ ./spawnctl deploy -u jkolb/spawnpoint/alpha -c example.yml -n demosvc
 Deployment complete, tailing log. Ctrl-C to quit.
[05/18 22:00:49] alpha::demosvc > [INFO] Booting service
[05/18 22:00:57] alpha::demosvc > [SUCCESS] Container (re)start successful
[05/18 22:00:59] alpha::demosvc > 1495170059120213585 [Info] Connected to BOSSWAVE router version 2.6.11 'Jansky'
[05/18 22:01:04] alpha::demosvc > Publishing 0
[05/18 22:01:07] alpha::demosvc > Publishing 1
[05/18 22:01:10] alpha::demosvc > Publishing 2
[05/18 22:01:13] alpha::demosvc > Publishing 3
[05/18 22:01:16] alpha::demosvc > Publishing 4
[05/18 22:01:19] alpha::demosvc > Publishing 5
^C
```

Figure 5.3: Deploying a new service with `spawnctl`

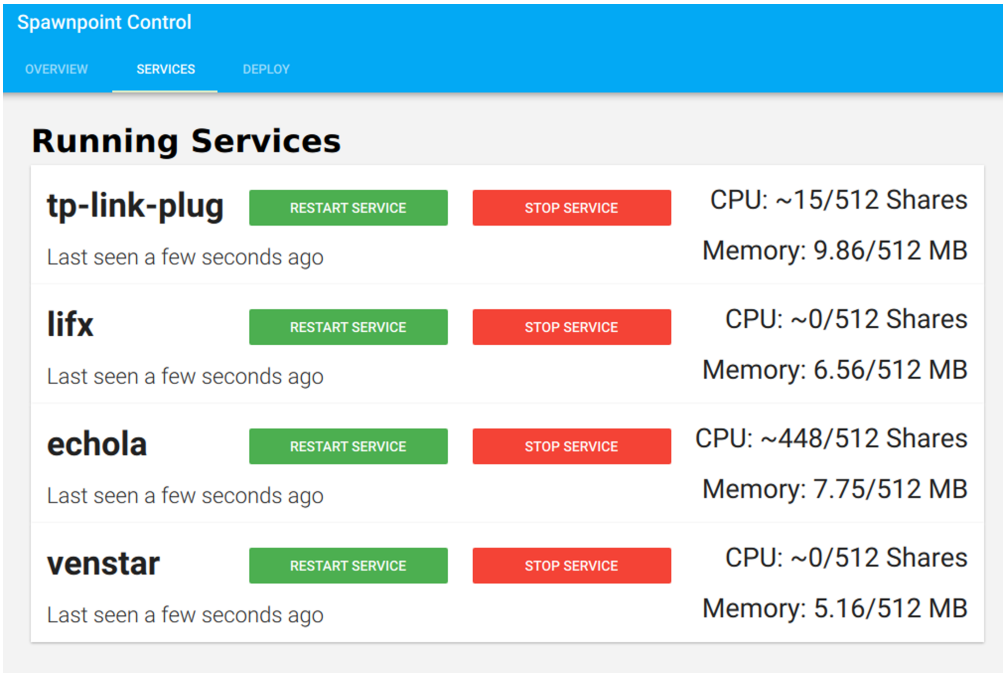
```
jack@cloakroom:~/goWork/src/github.com/immesys/spawnpoint/spawnctl$ ./spawnctl restart -u jkolb/spawnpoint/alpha -n demosvc
Monitoring service log. Press Ctrl-C to quit.
[05/18 22:08:29] alpha::demosvc > [INFO] Attempting restart
[05/18 22:08:31] alpha::demosvc > Publishing 22
[05/18 22:08:34] alpha::demosvc > [INFO] Stopped existing service
[05/18 22:08:34] alpha::demosvc > [SUCCESS] Container restart successful
[05/18 22:08:36] alpha::demosvc > 1495170516168361449 [Info] Connected to BOSSWAVE router version 2.6.11 'Jansky'
[05/18 22:08:41] alpha::demosvc > Publishing 0
[05/18 22:08:44] alpha::demosvc > Publishing 1
[05/18 22:08:47] alpha::demosvc > Publishing 2
[05/18 22:08:50] alpha::demosvc > Publishing 3
^C
```

Figure 5.4: Restarting an existing service with `spawnctl`

a `spawnctl`'s scan function. The second tab, shown in Figure 5.5b, provides a detailed listing of all services running on the daemon, and each service can be stopped or restarted at the push of a button. Finally, the third tab allows the user to launch a new service on the daemon by specifying a YAML configuration file. Note that, while this is a very different interface from the `spawnctl` tool, they are merely different wrappers for the same ultimate primitives: publishing to and subscribing on a specific collection of Bosswave URIs that constitute the interface for a `Spawnpoint` daemon, which itself is oblivious to the nature of its clients as long as they adhere to a common protocol.



(a) An overview of the daemon's status



(b) A detailed listing of the daemon's running services

Figure 5.5: Interfaces provided by the `Spawnpoint` Wavelet

5.3 Raptor

Raptor is Spawnpoint's most advanced frontend. Its role is to facilitate the construction and deployment of complex multi-container, multi-host applications. A real use case of Raptor is discussed in Section 6.4. It consists of two major components. The first is a domain-specific language (DSL) that is specified and parsed with a Scala program. Users describe the containers that form their application, how they are related to each other, and how they should be deployed, in this DSL. The Scala parser reads and validates such a description and converts it into a language-agnostic intermediate form with Google's Protocol Buffer framework.[1] The second component of Raptor, implemented in Go, reads this intermediate form and automatically instantiates the necessary containers according to the user's specifications. Artifacts written in Raptor's DSL are treated as declarative specifications rather than imperative instructions. More precisely, Raptor computes a "delta" between the containers described using the DSL and the containers that are already up and running on the relevant Spawnpoint daemons. Only containers that are not already running will be deployed. Raptor only does the minimum work necessary to bring the relevant collection of Spawnpoint instances into the desired state to support the application.

DSL and Parser

We implemented the parser for Raptor's DSL using Scala's standard parser combinator library.[2] This tool makes it easy to write clean, readable code for the parser that easily doubles as a specification for the DSL itself. An EBNF specification of the DSL is provided in Listing 1.

As can be seen from its EBNF specification, Raptor's DSL is fairly small and simple. A user describes their application from a few basic building blocks:

1. The *spawnpointList* symbol is a sequence of Bosswave URIs that identify the Spawnpoint daemons that will host the application's containers. An item in this list may specify the base URI for a specific spawnpoint, e.g. `jkolb/spawnpoint/alpha`, or it may use wildcards in order to reference any Spawnpoint instances that adhere to a particular URI pattern, e.g. `jkolb/spawnpoint/*` would refer to Spawnpoint instances with base URIs of `jkolb/spawnpoint/alpha`, `jkolb/spawnpoint/sodaHall/beta`, and so on.

2. The optional *dependencySpec* is a sequence of service names that are required by the application and are each expected to be up and running on one of the Spawnpoint instances described above. If any of these services are not found, the user's deployment specification is rejected, and they are notified of the missing services.

[1] https://developers.google.com/protocol-buffers
[2] https://github.com/scala/scala-parser-combinators

```
appDeployment = spawnpointList, [dependencySpec],
                {serviceDeployment | forComprehension}, [svcGraphSpec] ;

spawnpointList = "spawnpoint", "[", {uri, ","}, uri, [","], "]" ;

dependencySpec = "external", "[", {simpleName, ","}, simpleName, [","], "]" ;

serviceDeployment = "container", imageName, "as", (simpleName | varSubstitution),
                    "with", paramSequence, spawnpointSpec ;
paramSequence = "{", simpleName, (":" | "="), value}, "}" ;
spawnpointSpec = ("on", (simpleName | varSubstitution)) |
                 ("where", paramSequence) ;

forComprehension = "for", simpleName, "in", "[", value, ",", {value, ","}, "]" ,
                   "{", serviceDeployment, "}" ;

svcGraphSpec = "dependencies", "{", {svcPath, ","}, "}" ;
svcPath = simpleName, "->", {simpleName, "->"}, simpleName ;

value = ("""", simpleName, """") | number | path | memAllocLiteral | varSubstitution ;
varSubstitution = "$", "{", simpleName, "}" ;
path = {simpleName, "/"}, simpleName ;
uri = path, ["/", ("*" | "+")] ;
simpleName = (digit | letter), {digit | letter | "_"} ;
imageName = path, [":", simpleName] ;
memAllocLiteral = number, ("M" | "m" | "G" | "g") ;

number = ["-"], digit, {digit} , [".", digit, {digit}] ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"
       | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
       | "U" | "V" | "W" | "X" | "Y" | "Z" ;
```

Listing 1: EBNF specification for Raptor's domain-specific language

3. The most important part of a `Raptor` application specification is the sequence of *serviceDeployment* blocks. These may be enclosed within a *forComprehension* in order to avoid repetitive code when specifying services that verify only by a single parameter. It consists of several parts:

   - An *imageName* to identify the Docker image that will be used to instantiate the container
   - A name that will be used to identify the container within `Spawnpoint` and in the remainder of this `Raptor` specification
   - A sequence of configuration parameters, identical to the service configuration parameters described in Section 4.2.
   - A *spawnpointSpec* that indicates which `Spawnpoint` instance to deploy the container to. This may simply identify a specific `Spawnpoint` or it may take the form of a list of key/value pairs that specify constraints on the metadata of the `Spawnpoint` that will host the container. For example, a user may wish to deploy a service to any `Spawnpoint` where the `location` metadata key is `410 Soda Hall`.

4. Finally, a user may specify any dependencies among the containers of the application that should influence the order in which these containers are initialized in the *svcGraphSpec*. This takes the form of specifying paths and links in a directed acyclic graph. `Raptor` will then perform a topological sort and start the containers according to this ordering.

An example specification written in the `Raptor` DSL is provided in Listing 2. We can see that `Spawnpoint` daemons are referenced using two URIs, one with a wildcard. The `vision_driver` service is required to be deployed on one of the `Spawnpoints` referenced by these URIs for the application deployment to proceed. Next, five containers are specified inside of a single for comprehension. They share identical configuration parameters, except for their unique names. The containers can each be deployed on any of the `Spawnpoint` instances referred to by URIs at the beginning of the code snippet, as long as these daemons satisfy the given constraints on their `owner` and `location` metadata tags. Alternatively, the user could have instead identified a specific `Spawnpoint` to host all of the containers. Finally, the DSL specification describes a container dependency DAG with two components. Container `gamma` must be started before container `beta`, which in turn needs to be started before container `alpha` can be deployed. Similarly, the initialization of container `epsilon` must be complete before container `delta` can begin its deployment.

After successfully parsing an application specification, `Raptor`'s Scala component also performs basic validation. Every service that has been described must include parameters specifying its CPU and memory requirements, much like in a YAML file submitted to `spawnctl`, and these parameters must be in the proper form, i.e., both are positive integers and memory allocation must use valid units. Additionally, the service dependency

```
spawnpoints [scratch.ns/spawnpoint/alpha, jkolb.ns/spawnpoint/*]
external [vision_driver,]

for name in [alpha, beta, gamma, delta, epsilon] {
    container jhkolb/spawnpoint:amd64 as ${name} with {
        entity: ~/bosswave/keys/demosvc.ent,
        memAlloc: 512M,
        cpuShares: 1024,
        includedFiles: [params.yml],
    } where {
        owner = "Oski Bear",
        location = "Cloud"
    }
}

dependencies {
    alpha -> beta -> gamma,
    delta -> epsilon,
}
```

Listing 2: An example application specification written in Raptor's DSL

DAG is traversed to ensure that it is self contained — all nodes in the graph must be services described within the specification or explicitly listed as external dependencies. Finally, Raptor returns an error to the user if it encounters any undefined variables in the document. If an app specification passes the validation phase, it is serialized into a language-independent form as a final step.

Container Deployment

Raptor's second component is written in Go and has the task of realizing a user's application specification by deploying the necessary containers onto the underlying Spawnpoint hosts. It does so by taking the following steps:

1. Read the serialized specification in order to understand what containers need to be deployed as part of the application it describes.
2. Scan the Spawnpoint daemons identified by URI in the specification to learn what services they're currently running.
3. Check that each service listed as an external dependency is up and running. If a required service is not present, raise an error.
4. Topologically sort the containers according to their interdependency DAG.

5. Deploy the application's containers in this sorted order, omitting any containers that are already deployed.

6. Tail the logs of the newly deployed services until the user explicitly exits, much like `spawnctl`.

Currently, `Raptor`'s deployment strategy is fairly rudimentary. It deploys all containers in serial, and, if a container deployment fails, `Raptor` simply raises an error and stops at that point. Note that failed deployments are not necessarily a rare occurrence, as they include the situation where a `Spawnpoint` instance does not have enough CPU or memory to accommodate the new container. There are two major improvements that can be made to this strategy, which are left as future work. First, a degree of parallelism can be achieved by instantiating each component of the service DAG in parallel. Second, `Raptor` could endeavor to make application development more "atomic" by cleaning up the containers that are left over after an application deployment encounters an error and is only partially completed.

Step 5 in the list above glosses over an important challenge: determining which `Spawnpoint` daemon will host each container. When the user chooses a daemon for a container explicitly in their specification, this is trivial. However, when the specification only identifies a set of metadata constraints for a container's host, `Raptor` has more freedom in container placement. Currently, `Raptor` features a simple *first fit* scheduler — the first `Spawnpoint` instance that satisfies the metadata constraints and has sufficient CPU and memory to accommodate the new container is chosen to host it. This is another interesting prospect for future work, as more sophisticated schedules might try to enact more interesting placement techniques, such as minimizing the number of inter-service links that traverse machine boundaries in order to reduce network communication.

Chapter 6

Evaluation & Case Studies

This chapter describes both experimental evaluations of the `Spawnpoint` host daemon as well as case studies demonstrating the system's utility in real-world applications. First, we look at `Spawnpoint`'s behavior as it is subjected to a synthetic, steadily increasing workload and are able to establish that `spawnd` imposes minimal load on its host. Next, we examine the latency of `Spawnpoint` operations and observe that managed containers add very little overhead on top of native Docker containers. The discussion then shifts to two case studies based on real `Spawnpoint`-based software deployments. The first case study examines software drivers to encapsulate connected devices characteristic of today's IoT, while the second case study examines a computer vision application composed of several distributed software modules.

6.1 Overhead and Footprint

One of the primary development goals in `Spawnpoint` is to make its host management daemon, `spawnd`, lightweight. To the fullest extent possible, `Spawnpoint` hosts should devote their resources to executing and supporting services, with `spawnd` adding minimal overhead. To quantify this overhead, we measured the CPU consumption, memory footprint, and network traffic incurred by `spawnd` as it was subjected to a synthetically generated load that increases over time. Note that the system load associated with `spawnd` is decoupled from the load associated with the services it manages. That is, the cost of managing a service is independent of the cost of executing that service; `spawnd` maintains the same data structures and executes the same code for all services it manages. There is one important exception to this rule involving logging of container output, discussed below.

As shown below, `spawnd`'s burden on its host system is quite small, even when managing up to 100 services. Thus, it achieves its goal of serving as a lightweight management daemon that leaves the resources of the host available for executing services and avoids interfering with the execution of those services.

Experimental Setup

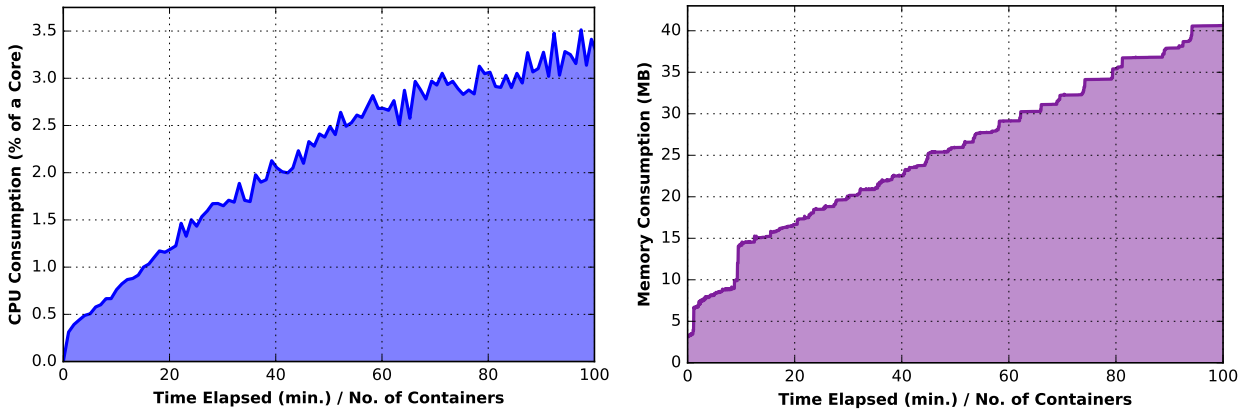
To determine `spawnd`'s overhead, an instance of the daemon was deployed as a container on a desktop PC with an Intel Core i7-6700 CPU featuring 8 logical cores running at 3.40 GHz and 32 GB of RAM. The machine ran Ubuntu 16.04 with Docker version 17.05.0 as its container engine. The `Spawnpoint` daemon was configured to allocate 8192 CPU shares and 16 GB of memory among all running services. Detailed instrumentation of the `spawnd` container was carried out using Google's `cadvisor`[1] tool.

In order to steadily increase load on `spawnd`, 100 instances of a simple service were deployed on the host, using the `spawnctl` frontend, at a rate of 1 service per minute. Each instance of the service was allocated 80 CPU shares and 64 MB of memory. Each

[1] `https://github.com/google/cadvisor`

service instance maintains an internal counter. Once per second, it increments the value of this counter, publishes this value to Bosswave, and writes a notification message to `stdout`. Recall that we can use this kind of simple service in this experiment because the management overhead of a service is independent of its complexity. While a computationally intensive service may increase the degree to which `spawnd` has to contend for system resources, it does not mean that `spawnd` will require more CPU, memory, or network resources to manage the service.

CPU and Memory Consumption



(a) CPU Consumption

(b) Memory Footprint

Figure 6.1: `spawnd`'s CPU and Memory Requirements as More Containers are Deployed

The CPU consumption of `spawnd` as load increases is shown in Figure 6.1a. Note that the x-axis of this graph represents both the number of currently running containers as well as the time that has elapsed since the start of the experiment in minutes. This is because new containers are deployed at a rate of one per minute. As depicted in the figure, `spawnd`'s load on its host increases as it is tasked with managing more containers, and the rate of this increase begins to slow as the total number of containers grows. Even when managing 100 separate containers, `spawnd`'s CPU demands peak at a very modest 3.5% of one core.

The memory footprint of `spawnd` is depicted in Figure 6.1b. The graph's shape is roughly a monotonically increasing step function. This is an artifact of the Go runtime's memory allocation system. As Go is a memory-safe language, it requests memory from the operating system in bulk to amortize the cost of these allocations, and subsequently doles out this memory internally for language objects that are subject to garbage collection. Thus what we are observing in this experiment is the footprint of the `spawnd` process, including free space internally maintained by the Go runtime, rather than the strict memory footprint of the running code. Similarly to CPU load, memory consumption increases with container

count, and this increase begins to slow as more containers are added. When managing 100 containers, `spawnd`'s memory demands are also modest, peaking at roughly 40 MB of space.

Network Traffic

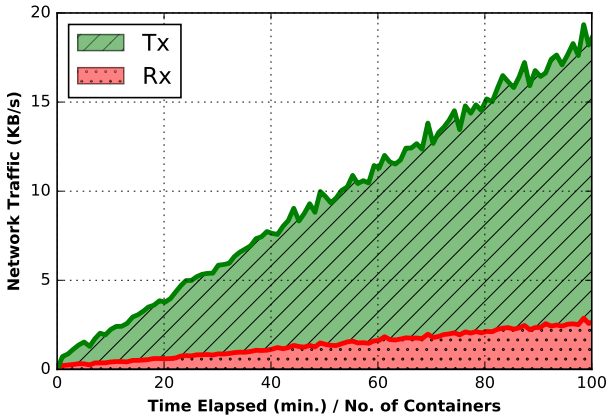


Figure 6.2: `spawnd`'s Induced Network Traffic as More Containers are Deployed

The total quantity of network traffic induced by `spawnd` is depicted in Figure 6.2 and is separated into upstream and downstream portions. Network traffic for both sending and receiving increases linearly with container count and, unlike CPU consumption and memory footprint, the rate of increase remains steady as more containers are deployed. This is the expected result as nearly all of the network traffic is the result of the periodic, service-specific heartbeat messages published to Bosswave by the `Spawnpoint` daemon. These messages are roughly equally sized across different services, producing the linear increase in upstream network traffic as more containers are deployed. The received traffic consists of control messages that are part of the Bosswave protocol and notify `spawnd` of the success or failure of its publication attempts. This is why the growth in upstream traffic is synchronized with the growth in downstream traffic.

Logging

While in general the load associated with a particular service on `spawnd` is independent of that service's complexity and resource demands, there is one important exception. `Spawnpoint` gives users the ability to view or tail a running service's output via Bosswave. That is, the text written to `stdout` or `stderr` is captured by `spawnd` and can be published to Bosswave when requested. In this situation, the load imposed on `spawnd` by a service is proportional to the amount of console output it produces.

To more precisely quantify logging overhead, we reran the experiment described above with one important change. When a new service is deployed on `spawnd`, a request to publish its output to Bosswave is also submitted and remains in effect for the duration of its execution. Memory footprint was generally unaffected by the change, and its peak remained at 40 MB. CPU consumption experienced a clear increase but remained modest, peaking at about 6% of one core. Finally, network traffic increased significantly and reached about 75 KB/sec. when all 100 services were actively running. The bandwidth demands of `spanwd` are thus fairly sensitive to the logging demands of the services it manages, although this has not presented a problem in any `Spawnpoint` deployments to date.

6.2 Efficiency of Container Operations

Here, we evaluate two important aspects of `Spawnpoint`: how quickly it can perform various container management tasks, and how the speed of each operation compares to the corresponding operation for an unmanaged docker container. This second point thus helps determine the cost of running a service as a managed `Spawnpoint` container as opposed to a standard `Docker` container. The results below demonstrate that `Spawnpoint` is both responsive (e.g., it can stop or restart a container in under a second) and that operations on managed containers have comparable latency to operations on unmanaged containers. Together with the results from 6.1, this indicates that the cost of container management is small.

Experimental Setup

All measurements of operational latency were collected on the same hardware used for the experiments described in Section 6.1. The `spawnd` implementation was modified to record the time required to perform the following operations:

1. Stop an existing service container
2. Restart a recently stopped service container
3. Restart a running service container
4. Deploy a new service container

These timing measurements are observed entirely within the `Spawnpoint` daemon and therefore do not capture network latency involved in sending the command from the client or in receiving a response. However, these measurements do capture the time required to parse the incoming Bosswave message as well as the time required to extract and deserialize the contents of the message.

Using `Spawnpoint`'s command-line client and the same simple counter-based service as before, each command listed above was repeatedly performed and profiled. Items (1) through (3) were repeated 100 times, while item (4) was repeated 50 times. A delay was included between each repetition of a command to allow `spawnd` to reach a quiescent state before performing the next test operation.

To perform corresponding operations on unmanaged containers, we used `Docker`'s command line tool to perform operations on an identical container, i.e., a container built from the same image that was used to instantiate the counter service on `Spawnpoint`. The corresponding operations are as follows:

1. `docker stop`
2. `docker start`
3. `docker restart`
4. `docker build` followed by `docker run`

Once again, each operation was repeatedly performed while the time required for completion was measured and recorded, with a delay inserted in between tests. Items (1) through (3) were repeated 100 times, and item (4) was repeated 50 times. As a final note, neither `spawnd` `Docker` were running any other containers for the duration of these experiments.

Results

The results from these timing experiments are shown in Figure 6.3. The height of each bar represents the average of all latency measurements for a given container operation; the error bars represent standard deviation. The cost of stopping a container is roughly identical for both managed and unmanaged containers. A managed `Spawnpoint` container takes longer than a native `Docker` container, although this difference is very small, on the order of 50 msec. Moreover, this is unsurprising as restarting a stopped container requires `spawnd` to perform the necessary admission control operations — checking if sufficient resources for the service are available and allocating those resources if so.

Interestingly, a more significant latency gap exists between `spawnd` and `Docker` when restarting a running container, although averages for both are well under a second. This may be due to the fact that the `spawnd` implementation, for the sake of simplicity, does not optimize heavily for container restarts vs. starting a fresh container. It does omit the process of building the container's image and is careful to avoid unnecessarily releasing and reacquiring the service's resource allocations (which would introduce a race condition), but `spawnd` still performs validation and container setup that is not strictly necessary when restarting an existing service. If this proves to be a bottleneck, the restart logic can be further optimized in the future.

Booting a container requires much more time (an order of magnitude) than the other operations. This is expected as it involves building a new Docker image. `Spawnpoint`

service developers are free to deploy a service with a ready-made image, but currently the common workflow is simply to push new versions of a service to a repository and let `Spawnpoint` update container images accordingly. The build process for the simple service container used in these experiments, for example, involves expensive operations such as downloading code from a source repository, compiling this code, and unpacking a compressed tarball containing items such as a Bosswave key and configuration parameters. While `spawnd`'s boot process does take longer than building and running a new container with `Docker` alone, the overhead is again fairly small, at under a second.

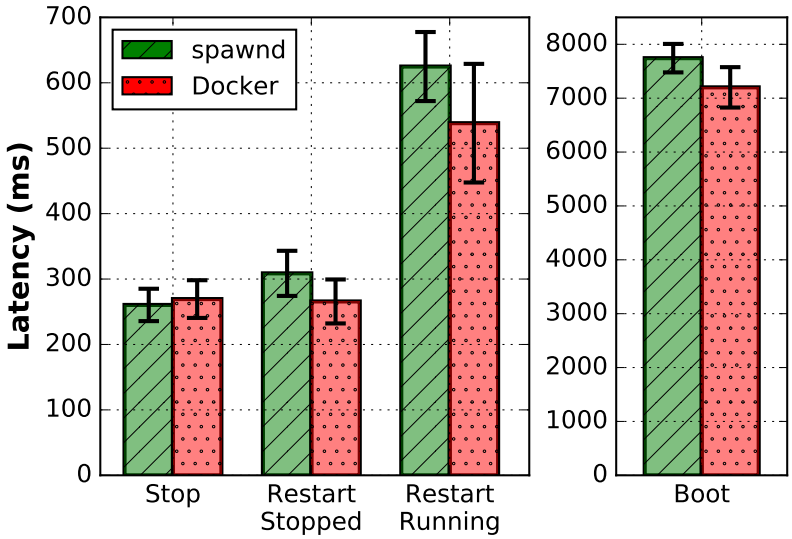


Figure 6.3: Latency for operations on both managed and unmanaged containers

6.3 Case Study: Securing Legacy IoT Devices

As the Internet of Things remains a nascent area, the design of many IoT hardware and software systems fails to adhere to proper security practices. For example, a web-connected camera might employ "hole-punching" to bypass a local NAT and expose itself directly over the Internet, allowing anyone to potentially connect to the camera and access its video feed. This situation not only raises serious concerns about user privacy, but also means that any security vulnerabilities in the device can be exploited by remote attackers. The Mirai Botnet is a famous example in which publicly accessible devices were hijacked simply by logging in using factory-set credentials. While this approach is quite simple, enough devices were compromised to power distributed denial of service (DDOS) attacks at unprecedented scale [41].

This motivates the secure deployment of local proxy services to encapsulate IoT devices and mediate access to such devices. Moreover, Bosswave serves as an excellent means of

securing access to these devices. The device's communications are restricted to the local area network, and a locally-deployed container running Bosswave communicates with the outside world on behalf of the device. Only authorized parties may access the data published by the device or send control commands to the device, and security is enforced through the primitives offered by Bosswave.

The deployment and management of these proxies is naturally accomplished with `Spawnpoint` and its associated tools. A homeowner who wishes to secure access to a legacy IoT device sets up a proxy on their local server running `Spawnpoint`. Because `Spawnpoint` containers can only be instantiated via publication to a Bosswave topic, only the homeowner or people who possess the necessary permissions can instantiate such a proxy. The ability to monitor the proxy is similarly restricted. Furthermore, a device proxy gains several useful abilities by running as a `Spawnpoint` container — automatic restart upon failure, enforced caps on CPU and memory consumption, advertisements of liveness through heartbeats — without complicating the implementation of the proxy itself.

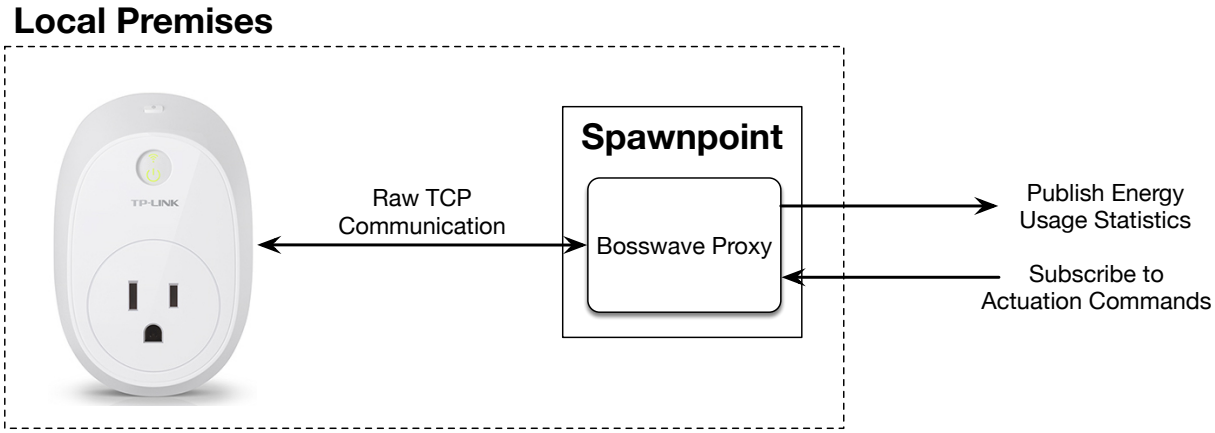


Figure 6.4: Deployment of a Bosswave Proxy for the TP-Link HS-110 on `Spawnpoint`

We implemented such a proxy for the TP-Link HS-100 series of WiFi-enabled electrical plugs.[2] The architecture for this proxy is shown in Figure 6.4. These plugs are fundamentally insecure as they expose a TCP endpoint that can accept remote actuation commands as well as requests for electricity consumption data in a JSON-based format. While the smart plugs do not communicate in the clear, they use a simple autokey cipher with a hard-coded initial key that can be gleaned by decompiling TP-Link's Android application [48]. This makes it trivial to encrypt and decrypt any traffic sent or received by the device. The role of the proxy, therefore, is to wrap the plug's vulnerable TCP interface with a Bosswave interface for sending data and receiving actuation commands. A particularly attractive feature of this proxy is its fine-grained permissions model. Different

[2] `http://www.tp-link.com/us/home-networking/smart-home/smart-plugs`

principals may be granted different combinations of permissions on the various Bosswave URIs exposed by the proxy. For example, the homeowner might have full permissions on all URIs, while guests may have (temporary) permissions to actuate the plug but not to view consumption data.

The configuration for the TP-Link HS-100 proxy is shown in Listing 3. It is fairly straightforward but is also a good illustration of `Spawnpoint` in action. It specifies the following for the proxy's container:

- The proxy will use the entity `evtplink.ent`, which is a file on the deploying host that is automatically included in the container, as its identity for all Bosswave communications.
- The container will be built from the `jhkolb/spawnpoint:amd64` base image.
- As part of the container build process, Go's `get` command line utility will be used to fetch the proxy's code at the specified git source control URL.
- To launch the container, the `tp-link-plugstrip` binary is invoked. This was produced as a result of the `go get` operation specified above.
- The container requires 512 megabytes of memory and half of a CPU core as allocations from the `Spawnpoint` daemon.
- One file is included in the container, a YAML parameters file with instance-specific information like the local IP of the TP-Link plug.

```
entity: evtplink.ent
container: jhkolb/spawnpoint:amd64
build: [go get github.com/SDB/bw2-contrib/driver/tp-link-plugstrip]
run: [tp-link-plugstrip]
memAlloc: 512M
cpuShares: 512
autoRestart: true
includedFiles: [params.yml]
```

Listing 3: Configuration for the TP-Link smart plug proxy container

6.4 Case Study: Building Vision Application

One of the primary purposes of `Spawnpoint` is to enable the vision of smart, software-augmented environments by providing a convenient and robust platform for service execution. `SnapLink` [31] is a smart environment application that relies on `Spawnpoint`

```
entity bosswave/keys/snaplink.ent
spawnpoints [ucberkeley/410/spawnpoint/*]

container kaifeichen/snaplink_dist:latest as front with {
    run: ["SnapLink/server/build/front", "spawnpoint_feature:8081"],
    memAlloc: 64M,
    cpuShares: 1024,
} where {
    location = "Soda Hall"
}
container kaifeichen/snaplink_dist:latest as feature with {
    run: ["SnapLink/server/build/feature", ],
    memAlloc: 512M,
    cpuShares: 1024,
} where {
    location = "Soda Hall"
}
container kaifeichen/snaplink_dist:latest as image_localize with {
    run: ["SnapLink/server/build/image_localize", "data/db/160308-162616.db",
          "data/417/db/160606-224205.db"],
    memAlloc: 1G,
    cpuShares: 2048,
} on castle
container kaifeichen/snaplink_dist:latest as image_project with {
    run: ["SnapLink/server/build/image_project", "data/db/160308-162616.db",
          "data/417/db/160606-224205.db"],
    memAlloc: 1G,
    cpuShares: 2048,
} on castle
container kaifeichen/snaplink_dist:latest as model_build with {
    run ["SnapLink/server/build/model_build"],
    memAlloc: 4G,
    cpuShares: 4096
} where {
    Location = "Cloud"
}

dependencies {
    front -> feature -> image_localize -> image_project -> model_build
}
```

Listing 4: SnapLink's Raptor deployment specification, originally written by its primary developer and modified for clarity

as its execution substrate. It is a system that facilitates human interaction with the many control points dispersed throughout a building. A user captures an image of an appliance using an application on their smartphone, and `SnapLink` identifies the specific appliance contained within the image. The smartphone app presents the appliance's associated control interface (e.g., a slider for controlling the luminosity of a light) on the user's phone. The user can then issue actuation commands through their smartphone, which are forwarded to the appliance, e.g., through a building management system or a Bosswave driver.

`SnapLink`'s backend is naturally implemented as a composition of services. The system architecture is shown in Figure 6.5. The components of `SnapLink` are as follows:

- A front end that interacts with smartphone clients, featuring an HTTP API for uploading query images and retrieving appliance matching results
- A module to extract the most significant features of submitted images using computer vision algorithms
- A module to match a submitted image to a location within a 3D point-cloud model of the building (and thus a location in the physical world) using the image's extracted features
- A module that projects the image onto 3D space based on its physical location, thus identifying the appliance contained in the image
- A module to construct and periodically update the building's 3D model using video of the building's interior captured with an RGB-d camera

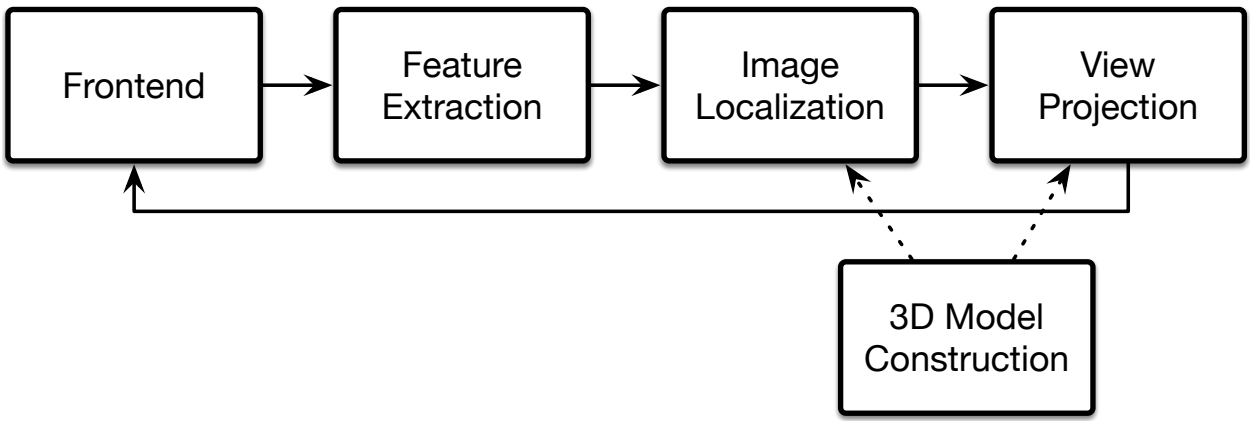


Figure 6.5: `SnapLink` architecture. Note that the arrows represent communication, not dependencies, between components. The dashed arrows indicate that model construction is performed offline.

SnapLink's backend performance is a significant concern because it influences the latency of the appliance identification process, which must be kept small in order to maintain responsiveness and give the user the experience of "instantaneous" interactions with their environment. Moreover, each of the services listed above has its own set of resource demands as well as deployment constraints that need to be met in order to optimize system performance. For example, creating a building's 3D model from video is computationally expensive and requires ample CPU and memory. The image localization and projection components are also somewhat expensive and should be co-located because they reference the same underlying data (i.e., the same 3D model). Finally, the feature extraction process should be run locally (in the building) because it reduces network traffic to external servers or the cloud — it is far cheaper to send the features extracted from an image than the image itself.

Listing 4 shows SnapLink's deployment description for Spawnpoint's Raptor frontend. The DSL is able to express all of the necessary configuration details and constraints. Each of SnapLink's modules is deployed as a container. The frontend and image feature extraction components run on local Spawnpoint instances (more precisely, spawnd instances with the Location metadata tag matching Soda Hall). Image localization and projection services are pinned to a specific Spawnpoint instance with alias castle in order to ensure that they are colocated. Finally, the model building process is constrained to a cloud-based Spawnpoint that is more likely to be able to satisfy its larger CPU and memory allocation requests.

Together with Raptor, Spawnpoint handles all of the deployment process for SnapLink. The developer's role is simple; all they need to do is submit the application specification from Listing 4 using Raptor's command-line tool. The tool will monitor the progress of the application's deployment and provide the user with updates as it proceeds. Internally, Raptor identifies the best Spawnpoint instance to host each container and issues appropriate deployment commands to each of these instances. From the application's deployment specification, we know that the frontend and feature extraction containers will run on a local Spawnpoint instance, the image localization and projection analysis containers run on castle, and model building is carried out on a cloud-based Spawnpoint host, e.g., on Amazon EC2. Currently, Raptor uses a simple first-fit heuristic when there are multiple Spawnpoint instances that both satisfy the user-specified deployment constraints and have sufficient available resources to host a particular service. However, this can easily be changed to a more sophisticated strategy in the future.

Raptor will also deploy services in the proper order, i.e., to satisfy any dependencies between services identified by the user. For example, in Listing 4, we can see that the service dependency graph is a simple path, with model_build at the end and front at the beginning. Therefore, model_build is the first service deployed and, once it is up and running, its predecessor in the graph is deployed, and so on until front is the last service deployed. Finally, recall from the earlier discussion that a Raptor description is declarative. Any services that are already running need not be redeployed. This can be quite useful when updating an application, e.g., to update a service to a newer version or

to add a new component to an existing application.

The specification in Listing 4 was originally written by `SnapLink`'s primary developer. Their experience in using `Raptor` revealed both strengths and weaknesses in the frontend's current implementation. The DSL was not difficult to learn, understand, and write, nor was it difficult to use `Raptor`'s built-in tool to submit a specification for deployment. However, once the application was deployed, `Raptor` did not provide sufficient assistance in monitoring the application's status. The user reverted back to the `spawnctl` client to understand the health of their application. Additionally, as `Raptor` does not leave a persistent record of how a logical specification was instantiated as a physical deployment, it is difficult for a user to remember where some of their services are running, especially when they are returning to work on their application after a long period of time has passed since its deployment. In the future, `Raptor` can be improved to address these issues. First, the system can give the user a persistent record of how their services have been deployed. Second, `Raptor` can provide the user the ability to query the health of every service in their application in one place.

Chapter 7

Conclusion

As computers have become increasingly pervasive in they physical world, they have fueled exciting visions of a future involving ubiquitous computing, smart environments, and a global scale Internet of Things. Fulfilling these visions, however, also requires a plethora of services to control and manage devices, collect and process data, interact with end users, and so on. `Spawnpoint` is a system that handles the deployment, management, and monitoring of these services. It leverages containers as a primitive in order to package service software and its dependencies, to ensure portability across heterogeneous hosts, and to monitor service execution and resource consumption. `Spawnpoint` enforces its security guarantees through its use of Bosswave, a secure publish/subscribe message bus. Only authorized users may deploy or control services, learn what services are running, or monitor service status.

At the heart of `Spawnpoint` is `spawnd`, a persistent daemon process that encapsulates the compute resources offered by a physical host running in the cloud or on premises. `spawnd` advertises the host on Bosswave and accepts commands to deploy new services on the host and to restart or remove existing services. `spawnd` conservatively allocates resources to running services and performs admission control for any potential new service in order to ensure that the host does not become oversubscribed. Services must reserve CPU and memory upfront, and `spawnd` enforces these reservations to ensure that no service monopolizes the underlying host. The daemon monitors the health of all running services, publishing status information on Bosswave, and recovers services from failures when configured to do so. `spawnd` leverages the `Docker` container engine to decouple itself from the services it manages, allowing the daemon to fail and recvoer without disrupting running services.

`Spawnpoint` features three frontends for users to interact with distributed hosts under management by `spawnd` instances. The first is a command-line interface that allows users to search for hosts, deploy a new service to a host, or manipulate an existing service. A graphical application for interaction with a single `Spawnpoint` instance is also available. Finally, `Raptor` is a tool for users to provision and launch complex applications comprised of multiple `Spawnpoint`-based services, potentially running on a distributed collection of hosts, all from one place. A user describes the collection of services and their configurations, constraints on the placement of these services, and the interdependencies between them. `Raptor` determines the final physical placement of each of the services, determines the order in which to deploy them, and issues appropriate commands to each of the `spawnd` instances involved. In short, `Raptor` allows users to describe a distributed application through a logical deployment specification without needing to reason about the application's instantiation on the underlying physical infrastructure.

Experiments demonstrate that `Spawnpoint` is lightweight and imposes minimal overhead on the hosts it encapsulates or the services it manages. When managing 100 services on a single host, `spawnd` requires about 6% of a CPU core, 40 MB of memory, and 20 KB/s of network bandwidth. Primitive operations like launching or restarting a service are efficient and comparable in speed to corresponding operations on unmanaged Docker containers. Moreover, `Spawnpoint` has been deployed as a key component in multiple

software systems, running a composition of services to support SnapLink, a vision-based building control application (Section 6.4), and as a deployment and monitoring framework for Bosswave-based secure device drivers (Section 6.3).

There are a number of interesting prospects for future work involving `Spawnpoint`. First, there are some potential augmentations to the existing system, such as supporting Kubernetes as a container deployment backend. This would enable `Spawnpoint` to run on clusters as well as single machines, which opens the possibility of more sophisticated deployment features such as load-balancing and replication. Support for clusters would also make `Spawnpoint` better suited for computationally intensive jobs and services, such as those involving machine learning. Another possible improvement to `Spawnpoint` is support for migrating services in response to changes in network conditions and resource availability. This would be easier for stateless services or services with soft state, as the main challenge would be a smooth handoff in which client requests are routed to the new instance of the service rather than the old instance. For services with state, more sophisticated migration techniques, such as those used for virtual machine migration [33, 42], would be necessary.

Running services on infrastructure managed by other parties raises important security concerns. Thus far, `Spawnpoint`'s design has assumed that the hosts it manages are trusted. A host does not tamper with a container's code, it does not inspect the container's internal state as it executes, and it does not deny service (e.g. network bandwidth or compute time) to the container. Intel has recently released CPUs that support a special set of instructions called Secure Guard Extensions (SGX) [39]. These instructions enable the creation and execution of secure "enclaves" — essentially encapsulated software that is protected from tampering and inspection by the host OS and/or hypervisor. This technology has already been leveraged in implementations of secure cloud computing [28] and secure containers [27]. However, SGX has not been applied to a distributed service infrastructure like `Spawnpoint`, nor does SGX deal with the issue of denial of service. A future version of `Spawnpoint` that provides both secure execution and guarantees against denial of service would form a powerful tool for distributed application and service developers.

Bibliography

[1] AllJoyn Framework — AllSeen Alliance. https://allseenalliance.org/framework. Accessed 2017-4-12.

[2] Ansible is Simple IT Automation. https://www.ansible.com. Accessed 2017-5-1.

[3] Apache River. https://river.apache.org/. Accessed 5-4-17.

[4] AWS Greengrass – Embedded Lambda Compute in Connected Devices. https://aws.amazon.com/greengrass/. Accessed 2017-4-12.

[5] AWS IoT. https://aws.amazon.com/iot. Accessed 2017-4-12.

[6] Azure IoT Suite — Microsoft Azure. https://azure.microsoft.com/en-us/suites/iot-suite. Accessed 2017-4-12.

[7] Chef: Deploy new code faster and more frequently. https://www.chef.io. Accessed 2017-5-1.

[8] Distributed Messaging – zeromq. http://zeromq.org. Accessed 5-6-17.

[9] Docker – Build, Ship, and Run any App, Anywhere. https://www.docker.com. Accessed 2017-4-29.

[10] Docker Machine. https://docs.docker.com/machine. Accessed 2017-4-30.

[11] Get Started with the Azure IoT Gateway SDK. https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-linux-gateway-sdk-get-started. Accessed 2017-4-12.

[12] Internet of Things – Google Cloud Platform. https://cloud.google.com/solutions/iot. Accessed 2017-4-12.

[13] Kubernetes — Production-Grade Container Orchestration. https://kubernetes.io. Accessed 2017-5-1.

[14] MQTT. http://mqtt.org/. Accessed 5-6-17.

[15] Oracle Tuxedo. http://www.oracle.com/technetwork/middleware/tuxedo/overview/index.html. Accessed 5-4-17.

[16] Overview of Docker Compose. https://docs.docker.com/compose/overview. Accessed 2017-4-30.

[17] Overview of Internet of Things — Solutions. https://cloud.google.com/solutions/iot-overview#gateway. Accessed 2017-4-25.

[18] Puppet - The shortest path to better software. https://puppet.com. Accessed 2017-5-1.

[19] quilt: Deploy and manage containers with JavaScript. https://github.com/quilt/quilt. Accessed 2017-5-1.

[20] rkt, a security-minded, standards-based container engine. https://coreos.com/rkt. Accessed 2017-4-29.

[21] Swarm mode overview. https://docs.docker.com/engine/swarm/. Accessed 2017-5-1.

[22] Terraform by HashiCorp. https://www.terraform.io. Accessed 2017-5-1.

[23] The Go Programming Language. https://golang.org. Accessed 5-18-17.

[24] Andersen, M. P., Kolb, J., Chen, K., Culler, D. E., and Katz, R. Democratizing Authority in the Built Environment. In *Proceedings of The 4th International Conference on Systems for Energy-Efficient Built Environments* (New York, NY, USA, 2017), BuildSys '17, ACM.

[25] Andersen, M. P., Kumar, S., and Kolb, J. GitHub - BOSSWAVE 2 Development. https://github.com/immesys/bw2. Accessed 11-27-17.

[26] Armengaud, E., Tengg, A., Driussi, M., Karner, M., Steger, C., and Weiss, R. *Automotive Embedded Systems*. Springer Netherlands, Dordrecht, 2011, pp. 155–171.

[27] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, 2016), USENIX Association, pp. 689–703.

[28] Baumann, A., Peinado, M., and Hunt, G. Shielding Applications from an Untrusted Cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 267–283.

[29] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing* (New York, NY, USA, 2012), MCC '12, ACM, pp. 13–16.

[30] Brady, K. Uses and Limitations of Micro-Synchrophasor Measurements in Distribution Grid Management. Master's thesis, EECS Department, University of California, Berkeley, May 2016.

[31] Chen, K., Fürst, J., Kolb, J., Kim, H.-S., Jin, X., Culler, D. E., and Katz, R. H. SnapLink: Fast and Accurate Vision-Based Appliance Control in Large Commercial Buildings. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies 1*, 4 (2017), 129:1–129:27.

[32] Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the Sixth European Conference on Computer Systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 301–314.

[33] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2* (2005), NSDI'05, pp. 273–286.

[34] Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2010), MobiSys '10, ACM, pp. 49–62.

[35] Dawson-Haggerty, S., Krioukov, A., Taneja, J., Karandikar, S., Fierro, G., Kitaev, N., and Culler, D. BOSS: Building Operating System Services. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 443–457.

[36] Dixon, C., Mahajan, R., Agarwal, S., Brush, A., Lee, B., Saroiu, S., and Bahl, P. An Operating System for the Home. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 337–352.

[37] Douglis, F., and Ousterhout, J. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Softw. Pract. Exper. 21*, 8 (July 1991), 757–785.

[38] Gribble, S. D., Welsh, M., Behren, R. v., Brewer, E. A., Culler, D., Borisov, N., Czerwinski, S., Gummadi, R., Hill, J., Joseph, A., Katz, R. H., Mao, Z. M., Ross, S., Zhao, B., and Holte, R. C. The Ninja Architecture for Robust Internet-scale Systems and Services. *Comput. Netw. 35*, 4 (Mar. 2001), 473–497.

[39] Intel Corporation. Intel SGX Homepage. `https://software.intel.com/en-us/sgx`. Accessed 11-16-17.

[40] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained Mobility in the Emerald System. *ACM Trans. Comput. Syst. 6*, 1 (Feb. 1988), 109–133.

[41] Krebs, B. KrebsOnSecurity Hit with Record DDOS. `https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos`. Accessed 2017-04-06.

[42] Lagar-Cavilla, H. A., Whitney, J. A., Scannell, A. M., Patchin, P., Rumble, S. M., de Lara, E., Brudno, M., and Satyanarayanan, M. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of the 4th ACM European Conference on Computer Systems* (New York, NY, USA, 2009), EuroSys '09, ACM, pp. 1–12.

[43] Lewis, J., and Fowler, M. Microservices. `https://martinfowler.com/articles/microservices.html`. Accessed 2017-4-30.

[44] Litzkow, M. J., Livny, M., and Mutka, M. W. Condor-a hunter of idle workstations. In *[1988] Proceedings. The 8th International Conference on Distributed* (Jun 1988), pp. 104–111.

[45] Ra, M.-R., Sheth, A., Mummert, L., Pillai, P., Wetherall, D., and Govindan, R. Odessa: Enabling Interactive Perception Applications on Mobile Devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2011), MobiSys '11, ACM, pp. 43–56.

[46] Satyanarayanan, M., Bahl, P., Caceres, R., and Davies, N. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing 8*, 4 (Oct 2009), 14–23.

[47] Shen, C., Singh, R. P., Phanishayee, A., Kansal, A., and Mahajan, R. Beam: Ending Monolithic Applications for Connected Devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 143–157.

[48] Stroetmann, L., and Esser, T. Reverse Engineering the TP-Link HS110. `https://www.softscheck.com/en/reverse-engineering-tp-link-hs110`. Accessed 2017-4-07.

[49] U.S. Department of Energy. Demand Response. `https://energy.gov/oe/services/technology-development/smart-grid/demand-response`. Accessed 5-7-17.

[50] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (Bordeaux, France, 2015).

[51] Weiser, M. Some Computer Science Issues in Ubiquitous Computing. *Commun. ACM 36*, 7 (July 1993), 75–84.

[52] Zhang, I., Szekeres, A., Aken, D. V., Ackerman, I., Gribble, S. D., Krishnamurthy, A., and Levy, H. M. Customizable and Extensible Deployment for Mobile/Cloud Applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 97–112.