by Lukasz Guminski

Recently I became interested in ContainerPilot - the idea of "self-aware and self-operating" containers promoted by Joyent. I decided to perform a few experiments to see how it behaves on top of Cisco's Mantl platform, which was well suited for my needs. In this blog post I am going to share the results and discuss the potential benefits of the technology.

What is ContainerPilot?

The core concept of ContainerPilot is to have an agent inside each container that deals with changes in infrastructure - for instance, if a microservice A depends on another one (B), the agent inside container A can update the configuration of microservice A whenever B changes location.

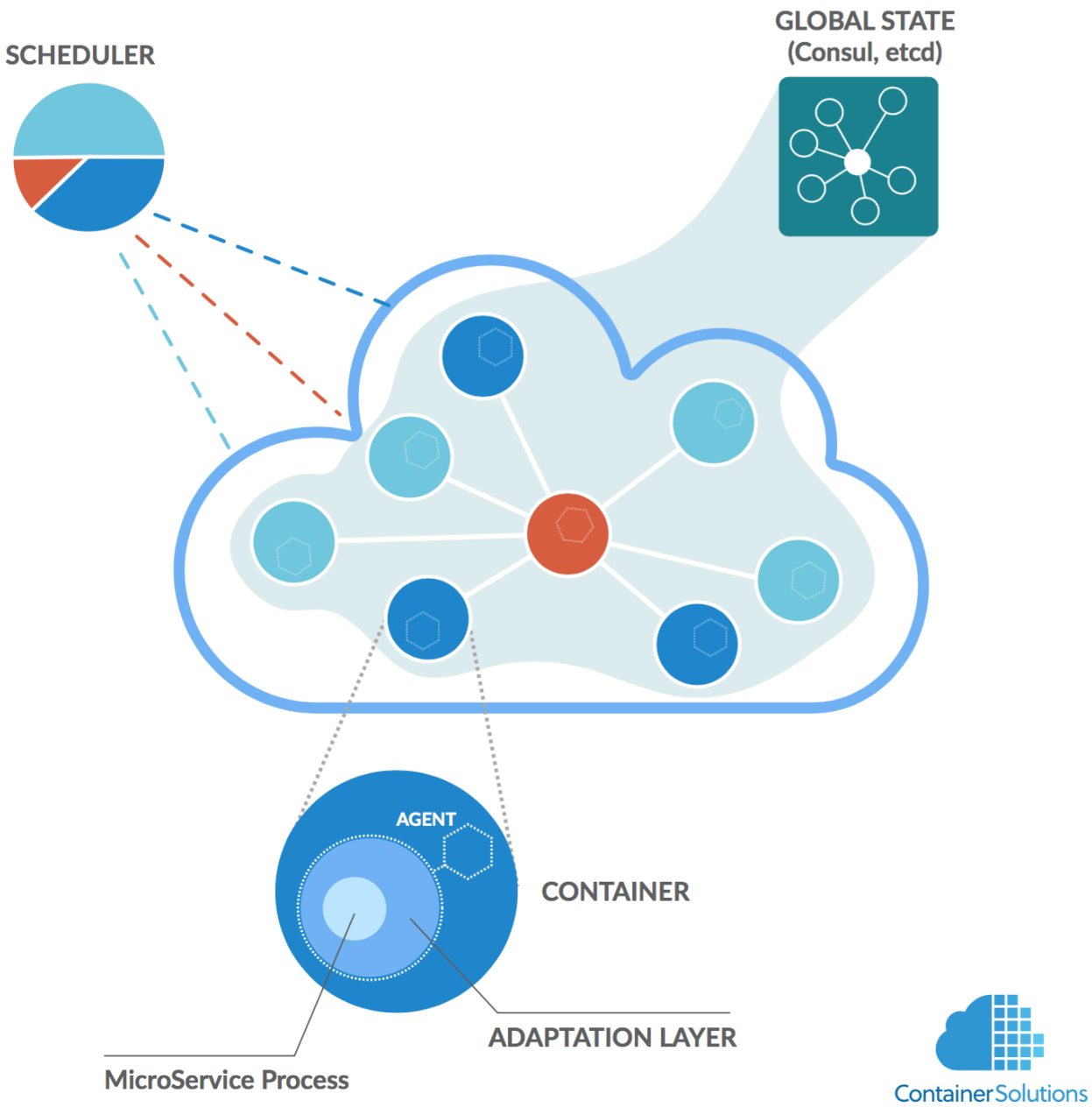
For this simple idea to work, the agent must be embedded inside each container, and start as the first process in the container (PID 1). Effectively this means that it is behaves like an init system, but unlike initd or systemd, it is meant to supervise only one process - the microservice. When the process fails beyond recovery, the agent exits as well, which effectively ends the lifecycle of the container.

The agent itself does not alter configurations, reload processes or anything like that - it delegates the task to handlers. You can use the following hooks:

- preStart - to initialize the microservice (e.g. to create configuration files).
- preStop/postStop - to perform a cleanup right before or after main process stopped.
- health - to periodically check if the application behaves as expected.
- onChange - whenever a dependency changes its state.

For the *onChange* hook to work, each agent registers its microservice in the Service Catalog (Consul or etcd), and then it can use the Catalog to query for information on the state of its dependencies (this is how the agent can then call *onChange* handler to react to the changes published in the catalog).

All the hooks together define the adaptation abilities of the microservice (*Adaptation Layer).*



It's worth to mention that Consul/etcd are not only used for Service Discovery. They are also shared key/value stores with distributed locking mechanisms. This can be used by containers to communicate and coordinate their actions.

In summary, ContainerPilot is a pattern where containers autonomously adapt to changes in their environment and coordinate their actions through a globally shared state maintained by a distributed key-value store (such as Consul or etcd).

Note that the allocation of resources for containers and starting them remains the responsibility of a scheduler.

Behavior

The architecture seems for be a compromise between pure self-organization and global coordination using state-of-the-art key/value stores. I am curious if it brings the benefits of the two: high availability and predictability. Let's see how it works.

Use cases

Joyent provided several examples on how to use it:

- **NGINX load balancer & webapps** (blog post)
  nginx proxy that updates its configuration whenever its backends change addresses.
- **Elasticsearch-Logstash-Kibana (ELK)** (source) - the stack supporting the analysis of logs.

The most advanced example is the **MySQL cluster** (blog post), where MySQL instances self-organize into a cluster with one master, and several replicas.

I decided to adapt the example and test it on Mantl platform, which is Cisco's platform for running microservices on top of Mesos, with additional services that modern microservice systems typically need. In this case I needed the following: Mesos' ability to run containers, Consul for Service Discovery and distributed locking, and Marathon for scheduling.

Preparation

(To skip the preparations and see how it works, jump to Deploying a cluster)

There were two options for implementing this. I could use Mantl in "process mode" - in which containers are merely "wrappers" for processes run on Mesos - or in "IoT mode" - where each container is like a unique device with its own globally routable IPv6 address.

The second option was more aligned with the idea of having autonomous agents, and also more interesting for me, so I went for it. For that I needed to enable Calico virtual networking and allow container to communicate with Consul through its external IPs - so I had to face a very strict security model of Mantl, which requires that communication between Consul and containers to be encrypted, login/password protected, and under the control of Access Control List (ACL) system.

Let me say a few words on preparing an application to work according to ContainerPilot. First each container needs to be injected with a ContainerPilot agent, so that the agent can start within the container's context (its namespaces). The way Joyent recommends extending all application's images by adding */bin/containerpilot* binary. Then the binary needs to be provided with configuration that tells the agent how to react to events (*preStart, pre/postStop, onChange, health)*. In the example, all handlers were defined in a single (pretty large) Python script. So I had to modify the script a little bit to use secure communication with Consul, and finally, I needed to add the Mantl public keys, because by default Mantl uses self-signed certificates for encryption. This is how the Dockerfile looks like (note that it is based on percona:5.6 image).

After I did all of this, it turned out that I need to wait until Docker gets upgraded in Mantl (issue) in order to get Calico to work. In this situation I decided to fall back to "process mode" and enable host networking (*-net=host*) for containers. That was additional work, since I needed to ensure that each container starts MySQL server at a different port in order to avoid conflicts. Luckily Marathon helps a little bit with this, and automatically assigns non-conflicting ports based on what Mesos offers (more on this here), but still I had to adjust all scripts.

Observation

Applying ContainerPilot requires building new Docker images, which is quite an intrusive process. Especially if this is a complex system that consists of various types of containers.

Nevertheless, I understand the rationale. By doing this we know exactly what version of ContainerPilot is used with each application. If this would be injected by mapping external volumes, then we would need a mechanism to ensure that versions are compatible. It would be a challenge as currently there is no configuration management solution in the Docker world that can do that.

Deploying a cluster

The most interesting thing about the pattern is that containers have lots of freedom to self-organize. It's exciting that at the moment of container deployment, I have no idea which one will be elected as leader. It all depends on which one will acquire a Consul lock - other containers will automatically start acting as replicas. This means that the same container can potentially become a master or replica, each with different behavior (polymorphism):

- Master initializes the database and starts to operate. In the background, it periodically exports its databases to Joyent's Manta store.
- Replicas upon startup import the latest backup and load it into their own databases. After that a replica establishes replication connection, and periodically tries to acquire lock (which succeeds only if there is no active master)

Ok. So I requested Marathon to schedule three containers.

Result

One of the containers acquired the lock and initialized as master. Other containers established themselves as replicas and immediately tried to download the latest backup. Unfortunately it was not ready yet (all containers started at exactly the same time), so they failed to initialize and never recovered (issue). After I removed the malfunctioning replicas manually, Marathon automatically recreated them, and then they initialized successfully.

```
Slave I/O thread: connected to master 'replicator@10.0.0.3:4796',replication started in log 'FIRST' at position 4
```

So some of my help was needed, but I finally got a healthy custer!

Observation

It's a small implementation bug, which however shows that an event based architecture with multiple processes running in parallel is exposed to all problems known from distributed programming, including race conditions.

I think this is an important observation. It means that to build systems that are truly reactive, we need to start thinking about infrastructure not like an operator, but more like a developer. ContainerPilot pattern enforces this kind of thinking, which is still very uncommon on the market. I like it.

Losing the Leader

During the deployment I had to manually remove an uninitialized replica. Because of the scheduler, the system recovered very quickly. But what will happen if I remove the master?

First, it is important to know that in microservice architectures the chance of an abrupt termination of a container is very high e.g. any time we ask Marathon to scale the cluster down, it can potentially terminate the current leader (from its perspective all containers are equally (un)important). The risk is built-in into the concept of microservices, and systems should develop strategies to be "antifragile".

So what should happen if the leader suddenly terminates? Something should release Consul lock, but what if ContainerPilot agent has already terminated? (Note that *preStop* and *postStop* handlers are executed only if a container is stopped gracefully) So what else can remove the lock?

Luckily Consul implemented the mechanism of sessions. In the example the lock is acquired by creating a session with TTL (time-to-live) set to 60 seconds.

*When creating a session, a TTL can be specified. If the TTL interval expires without being renewed, the session has expired and an invalidation is triggered (*Consul's documentation*).*

So in case there no leader, which could refresh the session, Consul releases the lock giving replicas a chance to become a leader.

Result

After a few seconds one of the replicas acquired the lock and became the master. A new leader got elected. Unfortunately, replicas did not reconnect, because Consul did not provide the correct information through REST API (despite the fact that in web UI everything looked fine). So the state replicas entered was:

```
Tried replication setup, but primary is set and not healthy.
```

Not good.

Observation

It looks like the mechanism works. But sometimes Consul propagates the state to all its agents too slowly (issue).

What problem does it solve?

I am always more interested in concepts than particular technical implementations. So right after looking at the implementation, I started to think what problem ContainerPilot pattern aims to solve. Is it better than other architectures?

So I think the problem could be defined as follows:

How to build reliable software systems out of many moving parts (microservices)?

If we look at the market for solutions that try to answer the same question, I can currently see two approaches. First, deployments tools (like Puppet, Chef, Ansible etc) are now being adapted to the world of microservices (see Puppet's Blueshift project). The tools already know the structure of systems they deploy, so it's tempting to use them for the coordination of microservices at run-time. The problem is that deployment tools were never designed to operate in real-time, and we need something that can react very fast (microservices can change their state at any moment).

The second approach are schedulers - AWS Auto-scaling groups, Mesos(w/Marathon), Kubernetes, Swarm, etc.. Their knowledge of system structure is typically minimal - they were simply designed to find resources for a new task - but in contrary to deployment tools, they were designed to operate in real-time. That's why for simple systems (e.g. only stateless containers) a scheduler can be sufficient to manage even large clusters.

The two products have one thing in common - they both manage microservices from outside i.e. orchestrate. There is another option: to let them self-organize - i.e. manage from inside.

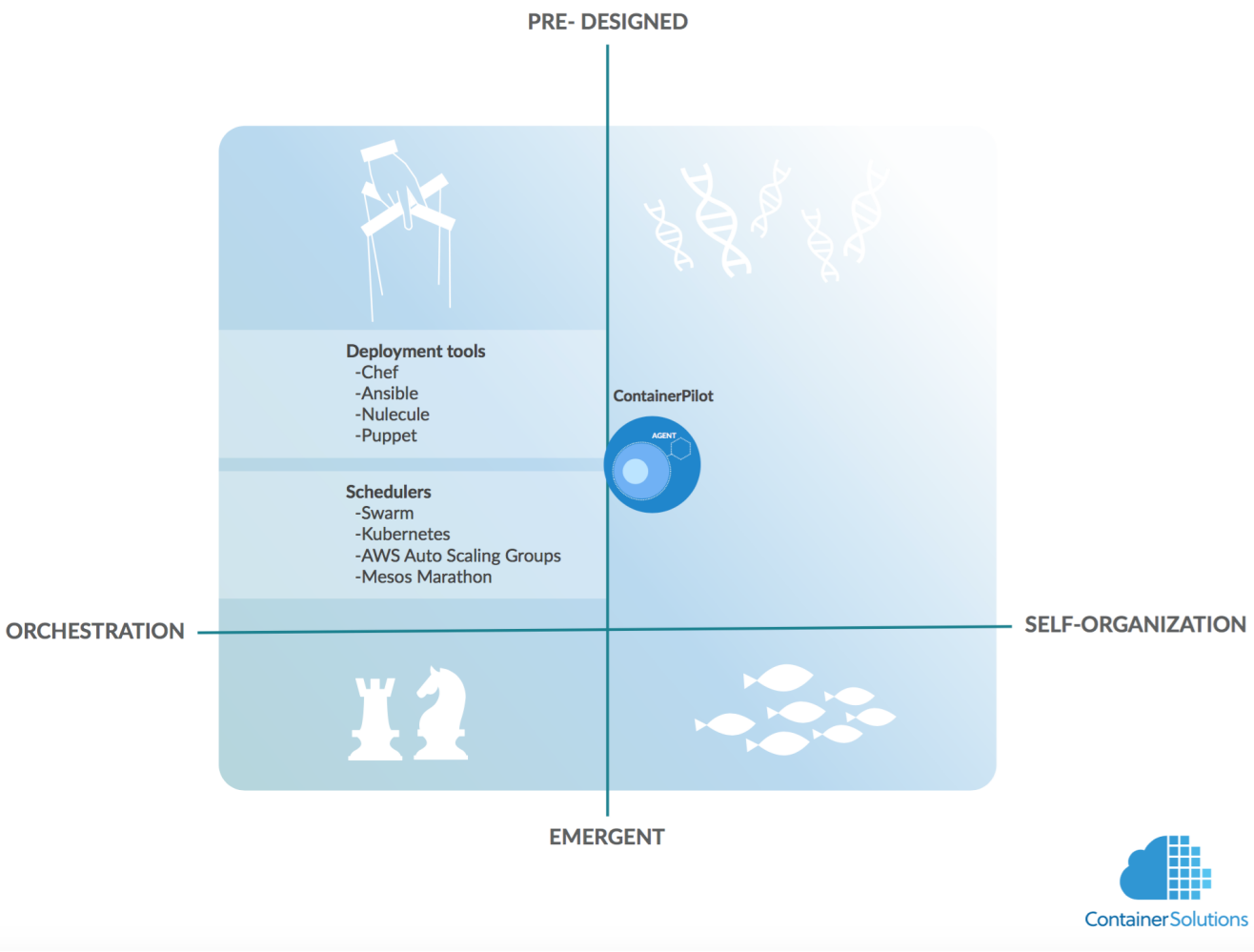
Like I mentioned earlier, self-organization lets you build highly available systems - in a purely self-organizing system, there is no central component, which could become a bottleneck or fail. On the other hand, this kind of systems have a strong tendency to become unpredictable. This is not always good, especially in the business context, which strongly prefers predictability.

However, there are various types of self-organization. Some are more predictable, some are less so. The most popular type of self-organization are "emergent" systems, when a swarm of very simple entities collectively produces a very complex behaviour. This is how ants, bees, fish, and to certain extent humans behave.

There is also another type of self-organizing systems. A good example are biological cells that operate based on DNA. In such systems there is no (radical) emergence The behaviour is mostly pre-designed:

The essence of emergence is the existence of a global behaviour that is novel. [...] When, for example, every agent has a model of the global behaviour that has to be achieved, this behaviour is explicitly present in the parts of the system and thus not novel.1

ContainerPilot is more like DNA-based systems. This means the designer needs to pre-design all behaviours of such systems i.e. to analyze all possible states and for all of them design recovery paths, so that no matter what happens to the system it will be able to return to safety.



Handling all possible states of a system from the perspective of a single microservice is not an easy task, even in the simplest possible case i.e. when all microservices are stateless. The reason for this is that *stateless microservices collectively form a stateful system*. This is clearly visible when a cluster needs to be upgraded. The

migration from an old version to new version needs to be performed gradually, otherwise the state of system might be disrupted. For a stateless system that would not matter.

> “Stateless microservices collectively form a stateful system.”
>
> Tweet This

There are ways to deal with gradual upgrades by self-organizing system - e.g. through attrition: containers of old version randomly commit suicide and get slowly replaced by new ones, however, it is easier to achieve the right speed of the process through orchestration.

In return for the initial design effort, a pre-designed system is much more predictable than emergent one.

Summary

I believe ContainerPilot can help build reliable systems. It is a smart choice between high availability of self-organizing systems and predictability of centrally managed system (owing to the implicit coordination through Consul’s global state). Even though it is in an early stage of development, the potential advantages are quite clear. I think the shift toward self-orchestration is what is coming soon and ContainerPilot paves the way. There are other solutions of this kind appearing on the market (e.g. Percona Orchestrator-agent), but ContainerPilot is a platform- and application-agnostic solution, which makes it more generic.

What comes next? I guess ContainerPilot will move a little bit toward more explicit central coordination, as it already started exposing endpoints for external orchestration (e.g. marking nodes for maintenance, telemetry endpoints). It would be also nice if ContainerPilot agent would be built-in into most popular Docker images as an alternative entrypoint. With all that I think the model could become an attractive option for large scale deployments.

Mantl, with its strict security model, is becoming a suitable platform for this. When only Calico virtual networking will be integrated properly into it, the platform will provide a new quality on top of Mesos.

References

[1] Tom De Wolf and Tom Holvoet, Emergence Versus Self-Organisation: Different Concepts but Promising When Combined, http://www.uvm.edu/~pdodds/teaching/courses/2009-08UVM-300/docs/others/2005/dewolf2005a.pdf, 2005