4 | Design of the Apate system

We have created a product, Apate, to solve the problems as described previously. Apate is a system of considerable size, with several moving parts. This section serves to give a brief overview of the system, its architecture and why these choices were made. Section 4.1 describes the ways in which Apate can emulate Kubernetes components. Afterwards, section 4.2 will give a high level overview of Apate, its components and the data flow within the system. Finally, section 4.3 explains how users can receive insights into the results of their experiments.

4.1 Emulation methods

Apate makes use of emulation in order to run the Kubernetes cluster using few resources, as discussed in 2.3. However, in order to be useful to users, Apate needs to be highly configurable.

In order to provide the user with options as to how to perform emulation, we provide two types of emulation: planned emulation - in the form of scenarios - and direct emulation. The types will be discussed in sections 4.1.1 and 4.1.2, respectively.

Both types of emulation have one common characteristic: they change the state of chosen nodes or pods. This state is used internally to determine how Apate should emulate the components of Kubernetes. The difference is in what way they allow users to change the state.

State

The state of a node or pod consists of a set of flags, the building blocks of Apate. Each flag signifies a certain response for a certain aspect of a pod or node. For example, there is a flag which determines how a node should handle status updates, and changing this flag might result in the node no longer sending status updates to Kubernetes.

This state can be updated in two ways. First off, one can directly update these flags. This provides the full range of control over the emulated pod or node. Besides directly updating flags, we have grouped some flags into *Events*. Events enable the user to modify the state, without needing to understand the use of each flag. For instance, one can introduce network latency or make some nodes in the cluster fail, without having the knowledge which flags are responsible for said events. In a way events are preprogrammed sets of flags, abstracting away the flags themselves.

4.1.1 Planned emulation

Our primary method of emulation is called *Planned emulation*. Planned emulation can best be described as a set of pre-programmed state changes. These state changes are described by a scenario, which consists of a list of tasks that are triggered at a certain timestamp. The fact that these state changes are planned makes planned emulation a good pick when multiple state changes are called for. Planned emulation also provides a certain amount of reproducibility. However, for simple scenarios (e.g. a single task) planned emulation might be uncalled for.

Tasks

A task is a state update performed at a certain point in time. This timestamp is defined relatively - more on that later. At this given timestamp, all state updates in a set are applied at once. Some tasks take precedence over others, as is the case in a real scenario. For example, adding network latency to a failed node will have no measurable effect. This priority is incorporated into the provider, which will be discussed in more depth in section 5.4.

Scenario

A set of tasks forms a scenario. A scenario has an absolute starting time. All relative timestamps in the tasks are based on this absolute start time.

An important difference of a scenario with respect to a task is that a scenario is the first thing that is not defined by the user. A scenario is merely an abstraction for all tasks combined, since in reality only the start time of the scenario is passed internally and all tasks are fetched dynamically from Kubernetes. The details of this process will be discussed later in section 5.2.

4.1.2 Direct emulation

Instead of using our scenario with all its components, a user can also directly change the state of nodes and pods. This change of state is no longer tied to a timestamp of any sorts. This form of emulation is just as powerful as using planned emulation, but it no longer forces the user to use our scenario. Besides that, it enables the user to more easily build their own system to control the emulation, and experiment with it in real time.

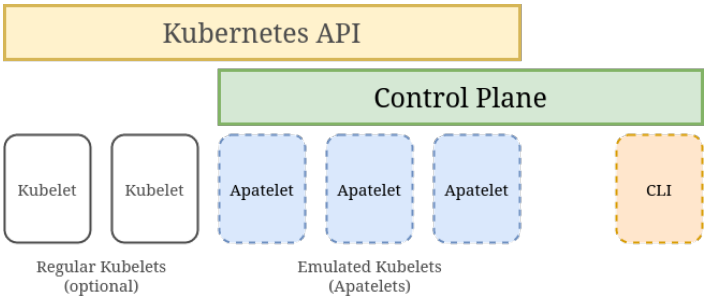

Figure 4.1: Apate architecture overview.

4.2 System architecture

Apate consists of three separate components: the control plane, the emulated Kubelets - Apatelets - and the CLI (Command-Line Interface). A simplified overview of the entire system can be found in figure 4.1. In this report we will refer to the CLI as the third component, but please note that this can be any client which uses our API.

4.2.1 Control plane

The control plane is the brain of Apate. The control plane is aware of all Apatelets, and continuously tracks them to ensure the Apate cluster is healthy. Its main responsibility is managing all Apatelets. Besides that, it is able to create a Kubernetes cluster to test with, or connect to an already existing cluster.

Managing Apatelets

There are several aspects to managing Apatelets: creating new Apatelets, destroying old ones, registering new Apatelets and keeping track of their health.

In order to create and destroy Apatelets, the control plane continuously compares the desired amount of Apatelets stored in Kubernetes to the actual amount of Apatelets, and tries to minimize the difference. New Apatelets are created using the configured runner, and old ones are destroyed by notifying them after which they shutdown themselves.

Nevertheless, failures are bound to occur, which means not all Apatelets might survive their startup: no free ports, not enough resources, temporary network issues, or some might stop during runtime. In order to have a representative experiment, the failed Apatelets should be purged from the cluster and new ones should take their place. This is something the control plane achieves using health checks and the watchdog; the former continuously updates the status of Apatelets, and the latter purges failing Apatelets from the cluster.

Managing the scenario

In case the user wants to use our planned emulation, the control plane will initialize the scenario by generating an absolute timestamp and notifying all Apatelets. Even when a scenario is already running, all other control plane processes will continue running. This ensures the scenario is dynamic, making it a very powerful tool.

4.2.2 Apatelets

The Apatelet is the component doing the heavy lifting. The Apatelet continuously retrieves tasks from Kubernetes, parses them and changes its internal state accordingly. It consists of four main components: the store, the scheduler, the informer, and the provider.

The store is the central part of the Apatelet. It stores all internal state, which all other components read or mutate.

The *scheduler*, a piece of code executing a function at a certain point in time, and *informer*, a Kubernetes concept which 'informs' one of updates in resource configurations, work in tandem to achieve their common goal: mutating the store according to the user's configuration. To that end, the informer retrieves tasks from Kubernetes, parses them, and puts them in the store. Once the parsed tasks are in the store, the scheduler takes care of mutating the state at the right time.

Finally, the provider is used once Kubernetes starts sending requests to the Apatelet. Its responses to said requests depend on the internal state in the store. The provider also keeps track of any pods scheduled by Kubernetes on the node. These Apate pods are very lightweight, as they are merely objects in a map, which are retrieved when Kubernetes for example asks for status updates for this specific pod. This leads to the fact that this emulated pod has no real limitations, it just requires one to implement the expected behavior.

4.2.3 CLI

The CLI is simply a consumer of the API exposed by the control plane. It uses the API to control the Apate system. It is also able to create a new control plane by interfacing with the Docker API. Besides that, it can retrieve the kubeconfig and start the scenario.

The CLI is a small component in the entire Apate system, but an important one. Without it, a user would need to call our API manually, therefore greatly reducing the accessibility of our system.

4.2.4 Data flow within Apate

Apate's components rely on data being processed and being moved around the system. There are two incoming streams of data: one from the API, and one from Kubernetes itself.

Figure 4.2 visualizes the data flows within Apate. Each cogwheel represents a service or smaller component, and each cylinder represents a data store.

Kubernetes

Our main input from Kubernetes is in the form of informers. These components monitor Kubernetes' state for changes, and more specifically for changes in our node configuration or pod configuration.

The control plane monitors the node configuration for changes in replicas. If there is a change it will either create new Apatelets, or stop old ones, depending on the difference. When there are not enough Apatelets, it will create new hardware configurations and hand them out to new Apatelets once they have started. If there are too many Apatelets, it will select just enough Apatelets and notify them, after which the selected Apatelets will shut themselves down.

The Apatelets monitor both node and pod configurations, in order to detect changes in tasks (see section 4.1.1). If there are any changes, the store will be updated accordingly, to allow the scheduler to pick up on these changes.

API

The consumers of the API can interact with Apate, albeit limited. The API can query the status of the Apate cluster, it can request information about the Kubernetes cluster, and it can start a scenario.

Output

As the dataflow within Apate is limited, we do not have an enormous amount of output data. The API can request information about the cluster, we expose metrics and the provider component answers requests from Kubernetes.

4.3 Gathering data from Kubernetes

Since Apate has the ability to connect to any Kubernetes cluster, the user is free to install any monitoring tool they want. The Apatelets also expose a metrics service, where users can query usage statistics of the emulated resources. There is only one restriction to the monitoring software users need to keep in mind: the software must not rely on a pod running on the emulated nodes, as this pod will just be emulated by Apate.

Even though the user has a lot of freedom, we optionally provide a small monitoring stack based on Prometheus [20] and Grafana [21] as will be discussed more in depth in section 5.7. This stack can be installed on the built-in Kubernetes cluster, or on any custom Kubernetes cluster, which provides instrumentation for both Apatelets and their emulated pods.

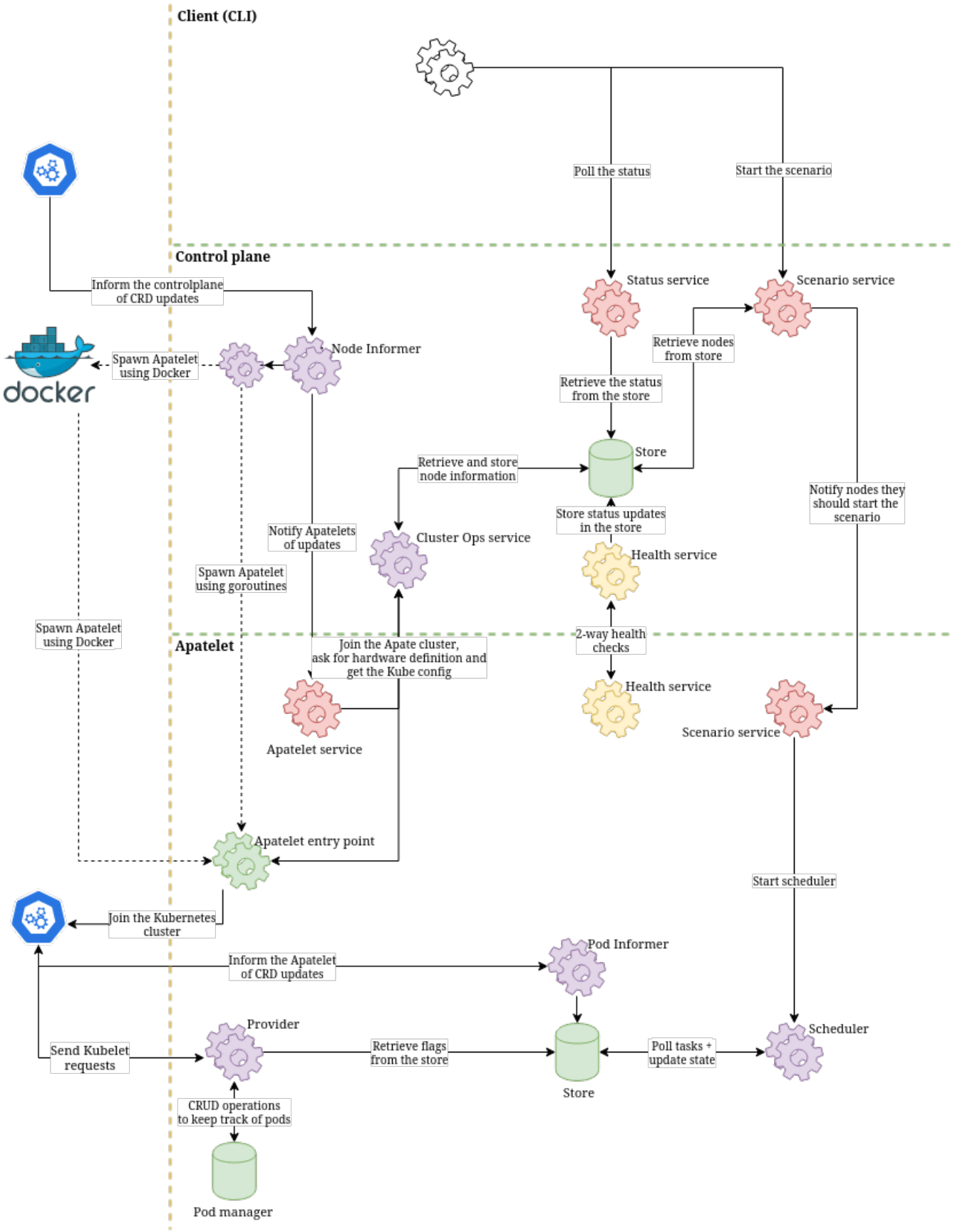

Figure 4.2: Data flow within Apate visualized.

5 | Implementing Apate

In implementing Apate, many implementation decisions have been made resulting in a relatively complex piece of software. This chapter aims to explain these decisions and to explain how some of the internals of Apate function. To start off, section 5.1 showcases the products utilized in order to build Apate. In section 5.2, the use of CRDs [1] will be extensively discussed. Continuing in sections 5.3 and 5.4 two very important parts of software are highlighted. Then, section 5.5 will illustrate how one can extend Apate to alter parts of its functionality. Section 5.6 will provide a walkthrough through our scenario. Subsequently, section 5.7 establishes how our monitoring stack functions. Section 5.8 concludes by explaining how we ensure the quality of the software.

5.1 Tools, languages and frameworks

This section aims to explain the tools we have used to create Apate and what their role is in the application.

5.1.1 Docker runtime

Docker is a tool which is used all throughout Apate. It is used to provide a Kubernetes cluster to test with, which will be further elaborated in section 5.1.5. Docker is also used to run the control plane and to optionally run the Apatelets. Lastly, we use Docker to run our end to end test suite and our continuous integration pipeline.

Advantages of Docker

There are a few reasons to choose for Docker over running a binary directly. First and foremost, it provides us the possibility to fully control the runtime environment. This means we can make sure the right versions of dependencies are installed, certain files are in place an we know up front what system utilities we can use. Secondly, many developers already use Docker[22]. This means that in many cases, one does not have to install anything to get up and running with Apate. Lastly, there is only a slight CPU and memory overhead of a Docker container with respect to running the binary directly[23].

5.1.2 Golang

The programming language in which Apate is written is Go. There are several reasons for doing so. The most prevalent of which is the fact that Docker and Kubernetes and the vast majority of third-party libraries for them are written in Go. Kubernetes and Docker also provide APIs which can be called directly from Go. Another reason for choosing Go is that multithreading is effortless and the goroutines[2] are very lightweight. Besides these, we think Go provides a combination of ease of writing, ecosystem, features and performance which is suitable for this project.

5.1.3 Virtual Kubelet

As mentioned in section 4.1, we emulate Kubernetes pods on virtual kubelets. To achieve this purpose, we need to act as a kubelet which is accepted by Kubernetes but does not run any actual pods. While we could implement this ourselves, there is a library doing exactly that: Virtual Kubelet. This popular, active and almost feature complete library provides a couple of interfaces to be implemented, which can control how the Kubelet acts towards Kubernetes. Virtual Kubelet takes care of the rest. This saves us a lot of time and since Virtual Kubelet has an extensive test suite, we can focus on testing our own software instead of also testing our interaction with Kubernetes.

[1] A Custom Resource Definition is a concept offered by Kubernetes, which allows third party software to add new definitions to Kubernetes

[2] A goroutine is a function that can be run concurrently. It can be viewed as a lightweight thread.

Alternatives to Virtual Kubelet

Before opting to go with Virtual Kubelet, we have considered a few alternatives, which will be discussed in this section.

Container Runtime Interface This solution revolves around the Container Runtime Interface, or CRI for short. In a nutshell, the container runtime, or a shim[3], can expose a gRPC service, which the Kubelet can use to spin up sandboxes. Figure 5.1 shows a simple overview of this architecture.

In this solution we would create a CRI shim for OpenDC, which would use OpenDC to emulate the sandboxes, instead of actually allocating resources and running the pods. This allows us to 'run' a high load using minimal resources. In theory, this can also be extended to a Man in the Middle[4] approach, where we can proxy requests to the actual CRI (like Docker or CRI-O) based on namespace. That would allow for running fake pods next to real pods.

At first glance this solution might look the easiest and least invasive, however it also has some disadvantages. For example, we are able to control pods, but we cannot easily control the node itself, so simulating node failure is a non-trivial task with this setup.

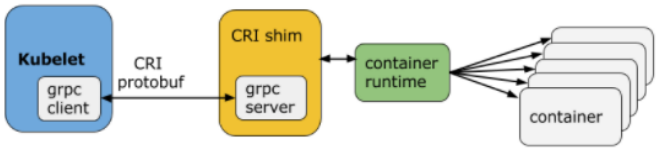

Figure 5.1: Kubelet architecture

Custom Kubelet One lever higher than the CRI is the Kubelet, which receives requests from the Control Plane and then relays these requests to the CRI which will run the actual containers. This Kubelet runs on every node as well. Kubernetes has the ability to taint Kubelets, i.e. mark them as unsuitable for certain workloads. This means we can make the scheduler run certain emulated, fake pods on certain nodes: the nodes which have the custom Kubelet that emulates those pods. Using a custom Kubelet might be easier than proxying the CRI as it is on a slightly higher level, and allows for more types of events like simulating node failure. Virtual Kubelet is an existing implementation of a Custom Kubelet, which allows easy alterations of this Kubelet through interfaces.

[3]Shim: A library that transparently intercepts API calls and changes the arguments passed, handles the operation itself or redirects the operation elsewhere

[4]Intercepting and modifying incoming and outgoing requests

Disadvantages

Although the functionality of Virtual Kubelet suited our needs perfectly, it has a significant downside: its internal Kubernetes version lacks behind the Kubernetes version. At the time of writing, the difference is 3 minor versions, which is about a year of updates. Luckily, Kubernetes has backwards compatibility which still allows this version difference, but we do miss out on new APIs as long as Virtual Kubelet is not upgraded.

Fork

Another more minor downside is that using the barebones Virtual Kubelet library requires a lot of boilerplate. The Virtual Kubelet `node-cli` project is a wrapper around Virtual Kubelet which implements all this boilerplate. Even though it is not recommended to be used in production, we opted to use this library given that Apate will only be used for testing and experimentation purposes, and is unable to serve any production applications. We have forked `node-cli` to improve its performance and memory usage when creating a good deal of nodes.

5.1.4 gRPC & protobuf

The API endpoints of both the control plane and the Apatelets are exposed using gRPC and Protocol Buffers, or protobuf in short. There are various reasons to choose a gRPC and protobuf API over a REST and JSON API. First of all, gRPC provides two-way communication using streams, which is very helpful when dealing with health streams and eliminates the need for polling. Second, protobuf provides a well-defined and statically typed request structure, something that JSON does not provide out-of-the-box.

5.1.5 KinD

Setting up an entire Kubernetes cluster can be time consuming, so Apate can provide a cluster to test and experiment with. This cluster is set up using KinD, which is a Kubernetes cluster inside of a Docker container. Even though this brings a bit of overhead, we think this is the easiest solution, only requiring Docker

to be installed and thus not relying on a certain virtualization software.

Although the overhead is relatively low, it may still cause performance tests using the KinD cluster to be skewed. In order to use a different cluster than the integrated KinD cluster, one can provide the kubeconfig for another cluster to connect to in the environment variables of the control plane.

5.2 Custom Resource Definition

An integral part of Apate is the use of Custom Resource Definitions (CRD). This section aims to illustrate why and how we make use of CRDs.

5.2.1 Concept

Original scenario

In the first iteration of Apate, the concept of scenarios was central to the application. These scenarios were protobuf definitions which almost exclusively contained a list of tasks, similarly to how tasks are currently handled. Clients were expected to send these scenarios to the control plane, which would parse, validate and translate them into a different format which was understood by the Apatelets. This translation was required because we wanted the end user to have a pleasing API to work with, instead of the "desugared" internal API we were using for Apatelets. All-in-all, we were using a non-standard format to work with scenarios. A drawback of this is that changing this scenario required changes in many parts of the application, reducing our extendability. Having the ability to update the current state of an Apatelet directly, without using tasks, would also require significant changes to our entire application.

Switch to CRD

We eventually made the decision to drop our custom scenarios entirely, and made the switch to Kubernetes' CRDs. CRDs are custom resource definitions which one can use however they wish. Pods are built-in examples of CRDs. There are numerous advantages to using CRDs over our custom scenario, the most important of which is the fact that they are standardized: we define a CRD which can be read by tools that support CRDs. The configurations made by the end user, which can be seen as "instances" of a CRD, can then be applied using any such tool. This configuration will then be parsed and validated by Kubernetes, after which it is distributed to listeners of that specific CRD. This means CRDs have more functionality than our original scenario, have a standard format, and can much more easily be changed.

5.2.2 Implementation

For both emulated nodes and pods we have made CRDs, which have a similar structure: on the top level of the `spec`, one can directly apply state, as alluded to in 4.1.2. Another field on this top level is field containing a list of tasks with their respective timestamps. These timestamps are duration strings, such as `10s`. We have opted for this format since it is easy to directly apply new state and also easy to update tasks.

Applying a pod configuration

A design goal of CRDs was that one is able to start pods as they would do normally, without having to interact with Apate. When plainly creating pods, without applying any pod configuration, the default behavior is to tell Kubernetes this pod is running. When one wants to alter the behavior of the pod, one can apply a pod configuration. This pod configuration can be attached to a pod by adding a label to this pod. When this label matches the `<namespace>/<name>` combination of the pod configuration, this configuration will be applied to this very pod. This means that configuring a pod is independent of configuring a node, and as such, a node needs to keep track of multiple pod configurations and match these to the pods currently running on this node.

Applying a node configuration

Node configurations do not require any labelling and matching. They function as follows: one defines a node configuration and states the amount of replicas required. When this configuration is applied, Apate will take care of spawning and killing nodes accordingly. These nodes are either spawned as goroutines or as Docker containers, based on the preference of the user. An Apatelet can thus be viewed as a separate process, whereas pods are merely stored in an in-memory map.

Controller-gen

To define the CRD structs, we have used the standard Kubernetes format. Normally, one would use

`controller-gen` to generate a REST client for this CRD, but we have not done so, since `controller-gen` does not have gomodules support and we only need to perform two REST calls.

Informers

As mentioned in the previous section, configuration updates are distributed to listeners of a particular CRD. These listeners are called "informers", and we use those to schedule updates on both the control plane and the Apatelets. The control plane listens for updates for emulated nodes, and starts more or stops Apatelets based on the given configuration. The Apatelets watch for updates for both CRDs. Changes to these configurations are reflected in updates to the Apatelet store, as will be discussed in section 5.3. When a new Apatelet is added, it will make sure it is in the same state as the other Apaletets of the same configuration by applying all previous state changes.

5.3 Apatelet store

The Apatelet store and its accessory scheduler are the core of the emulation. The thread-safe Apatelet store contains the current state of the Apatelet and the tasks which have to be executed in the future. The current state is saved on both a node level and on a pod level. Seeing as there can be different types of pods running on a single Apatelet, a single state is not enough. Tasks are more abstract and both pod and node configurations can be found in a single task queue. The reason for this is that this makes our scheduler much simpler, as it only needs to deal with one type.

5.3.1 Internal flags

The store makes use of flags mentioned in 4.1 to save the current state. These flags can be flags such as "how to respond to request x". These flags solely determine how the providers respond to Kubernetes requests and are normalized. Meaning: every flag is distinct and changing flags will always alter the behavior of the provider. This makes implementing the providers much easier, given that these flags do not have complicated relations to each other. We always make a clear distinction between an unset flag and a flag which is actually set. This is needed because flags can be set on both a node and pod level.

5.3.2 Updating the flags

When a - node or pod - configuration update arrives, its direct state is translated into flags, which are instantly updated in the store. When tasks are provided by the configuration, all current tasks in the store which belong to this configuration are removed and the new ones are added. This is done in an efficient manner, as tasks are handled using a priority queue and are internally saved in a heap.

5.3.3 Scheduler

In order to apply the tasks from the store, we use a scheduler. This scheduler will intelligently poll tasks from the store, and will be completely idle while waiting for tasks to be executed. When we want to execute a task, based on the provided relative timestamp, we translate this task into flags just as we did for directly altering state. After this, the task is removed from the store, to mark it as completed.

5.4 Providers

The actual emulation is done in providers, which are interfaces exposed by Virtual Kubelet. These providers receive requests by Kubernetes and need to respond accordingly. We have implemented all the possible providers, as to have as much control over the emulation as possible.

These providers are the core part of each Apatelet, as they provide Kubernetes with all the information necessary for Kubernetes to view them as normal Kubelets. Besides taints that have been applied to these Apatelets, there is no difference between a normal Kubelet and an Apatelet from the outside. This means that pods are also scheduled normally.

5.4.1 Pod provider

The pod provider is arguably the most important provider of all. This provider is called when a pod is created, updated, deleted and retrieved. All these requests are handled similarly: first, we apply some latency to the request, depending on what the user wants. By default this is 0, but the user can set it to any duration. Secondly, we retrieve the pod from our pod manager: a struct which keeps track of the pods we currently have running. Then we check which flags are set in the store for this specific pod and node.

We store flags that determine how to respond to a request for each of the requests, on both a node and pod level, as discussed in section 5.3. These responses can be "normal", "timeout" or "error". Based on these responses we either return an error, do not return at all or just return the correct value from our pod manager. Which response to apply is determined by getting the "worst case" response between the node and pod flag: if either the pod or node configuration defines an error, the pod will error.

5.4.2 Node provider

Another important part is telling Kubernetes what the status of our node is. For this we have implemented a node provider. To start off, this provider needs to respond to ping requests.

This is done similarly as described in the previous section, by retrieving flags and responding correspondingly. This is only done on node level. Next, we need to tell Kubernetes the status of the node. The `ConfigureNode` function is called by Virtual Kubelet when registering with Kubernetes, which will return the resources that the node has available. We can also notify Kubernetes of changes using the `NotifyNodeStatus` callback function, which we call regularly. This update contains statistics about whether we have memory pressure, full disks etc. To provide the conditions for these resources, we use our own wrapper around Kubernetes' conditions in order to provide Kubernetes with enough information.

5.4.3 Stats provider

The stats provider is a bit less important than the other two providers, though still useful. It aggregates all the information we have about the pods and uses this information to return information about the node. These statistics are used in the previously mentioned node provider and by the Kubernetes scheduler to determine where to schedule certain pods and by our logging stack.

5.5 Extending Apate

There are a few ways in which one can extend on the functionality of Apate to add their own custom behavior. This section will demonstrate these options.

5.5.1 Spawning Apatelets

As mentioned in 5.2, the control plane determines when and how to spawn and stop Apatelets. By default, we allow for spawning Apatelets as Docker containers or as goroutines, based on which environment variables are in place. Both of these implementations implement the `ApateletRunner` interface, which only has a single function that needs to be implemented to spawn n Apatelets with certain environment variables.

This interface can be implemented by anyone else too, and new runners can be registered by registering them to the runner registry.

An example of a runner that could be added is a runner utilizing Red Hat's docker alternative Podman[24].

5.5.2 Adding traces

Our client requested adding support for tracing, i.e. emulating previous runs of workflows from actual Kubernetes clusters. After further investigation it became apparent that there is no singular standard to support traces, and we were not very keen on only supporting a single format. We believe the tasks we already supported in CRDs are perfect for this use case.

We suggest the client and other possible users of Apate write conversion scripts between these traces and the CRDs. To aid in supporting traces however, we have implemented features such as optionally defining the timestamp to be relative to the start of the pod instead of the start of the scenario.

5.5.3 Connecting to an external cluster

As mentioned in section 5.1.5, we provide our own testing cluster, for ease of use. It can however be very useful to test with a cloud platform, a different Kubernetes implementation or just your own local cluster. This is why we support this, by passing in an existing kubeconfig to our control plane. The control plane will then take care of joining the Apatelets to this cluster. We have verified this functionality by connecting Apate to k3s and GKE.

5.6 Scenario walkthrough

Just by theory alone, it might be difficult to truly understand how Apate works. For that reason, we will walk through a simple scenario, explaining all steps. Feel free to skip this section, if you grasp the inner

workings of Apate already. In this scenario we will emulate a simple cluster of three workers nodes, running a nginx deployment with three replicas. After ten seconds, relative to the start of the scenario, all worker nodes will fail. The files used for this explanation for the node configuration, pod configuration, and nginx deployments can be found in listing 1, 2, and 3, respectively.

5.6.1 Control plane startup

Before anything else, the control plane will start. This can be done by starting our provided Docker container, or using the CLI.

Once the container started, the control plane will set up its components: the store, the gRPC server, and a Kubernetes cluster (optional). Once all components are running, it is open to requests from the CLI and Apatelets. It also starts informers for our CRDs, so the control plane receives updates when they are modified.

5.6.2 Starting Apatelets

Once the Kubernetes cluster and Apate control plane are running, the user can update the node and pod configurations using the regular Kubernetes tools, kubectl for example.

The moment the user updates the node CRD, the control plane will be notified. When a CRD is updated (which could also mean added or deleted), the control plane will first form a unique selector for said CRD. This is just a combination of the name and namespace. Using the created selector, the control plane will compare the amount of Apatelets registered to the amount of desired replicas. The control plane will act accordingly, spawning new Apatelets (using the selected runner), stopping old ones, or do nothing, depending on the result of the earlier comparison.

Creation

If there is an insufficient amount on Apatelets running, the control plane will spawn new ones. Once an Apatelet starts, it will first initialize its components: the store and the gRPC server. It will then connect to the control plane, using the connection information provided in its environment by the control plane, and attempt to join the Apate cluster. The control plane will add the Apatelet to its store, and return all information needed: a unique identifier, emulated hardware specifications, and the kubeconfig.

Once the Apatelet has joined our cluster, it will start virtual kubelet, and join the Kubernetes cluster. Besides that, it will start informers for our node and pod configurations. This will immediately trigger an update, so the Apatelet can update its internal store with all queued tasks.

Deletion

Deletion is more straight forward than creation. If there are too many Apatelets running, the control plane will select Apatelets to be deleted. They will be removed from both the Kubernetes cluster, and the Apate cluster. Finally, the Apatelets will be notified by a gRPC call, which shuts down the entire Apatelet.

5.6.3 Starting the scenario

A user can start the scenario by interacting with the control plane API. This can be done using the CLI for example.

Once the scenario has been started, the control plane will pick a timestamp in the near future, and notify all Apatelets. All Apatelets will then update the scenario timestamp and start their scheduler.

5.6.4 Node failure

Once started, the scheduler will sleep until the next task should be executed. In this case the next task is ten seconds after the initial timestamp. After ten seconds the scheduler will retrieve the task from the queue, and execute it. In this case the task should result in node failure, so all flags will be set to a timeout state.

Every request from Kubernetes will now result in a timeout. The Apatelet will also no longer update its status in the Kubernetes cluster. After a while (depends on the Kubernetes configuration) Kubernetes will show the worker nodes as unreachable and unschedulable. This completes the scenario.

5.7 Monitoring stack

To monitor what actions Kubernetes is undertaking and what effects emulation has on the cluster, we provide the option to create a monitoring stack. We have opted for Prometheus as Kubernetes supports it out-of-the-box, as Kubernetes exposes a multitude of metrics to Prometheus by default. Since Virtual Kubelet internally uses an old version of Kubernetes, the metrics it

provided did not work with Prometheus out-of-the-box. This unfortunately meant we had to implement our own endpoints to log to Prometheus from Apatelets. Examples of such logs are where and when pods are created, updated, deleted and read by Kubernetes.

Besides Prometheus, it is also still possible to use tools which interact with the Kubernetes API to retrieve the state of the cluster, such as `kubectl` and `k9s`.

5.8 Software quality

This section will discuss the effort we have put in into making Apate a proper program.

5.8.1 Test suite

Our test suite is divided into three parts: unit, end-to-end (e2e) and race. The unit tests assert that the individual components work properly. These unit tests are most useful in parts which have no external dependencies, such as the stores and the scheduler. Unfortunately, many other parts of the application call Kubernetes or other external tools and are therefore hard to unit test, as those functions almost exclusively consist of external side effects. We did try to mock but sometimes even mocking was not possible. To make sure our application works as a whole we use an e2e test suite. These e2e tests run through the entire workflow completely and for example asserts that Kubernetes responds to our providers properly. The last tests are race conditions tests. Race condition tests are built into Go, and it essentially runs our unit tests with some extra checks to test for race conditions. We do not run e2e tests while checking for race conditions because they would take too long, as testing for race conditions increases the time of execution and resource usage of these tests by a fair amount already.

5.8.2 Continuous Integration

We have had an extensive Continuous Integration suite from the very beginning. This suite catches many possible errors before a merge. It is subdivided into the following stages.

Prepare

In the prepare stage we prepare our Docker container which is later used for testing. This Docker container contains all the tools required for building and testing purposes and is based on the Docker Alpine container itself, as running tests may involve starting Docker containers, such as KinD, as explained in section 5.1.5.

Build

This stage builds all tools in the Apate platform and runs the generators we use for generating CRDs, mocks and protobuf files. We run these generators in our CI suite as there have been numerous occasions in which these generators were not run before merging and errors were found when running them later on. Another part is generating our test definitions so we can parallelize our tests in the next stage.

Test

The test stage runs all our tests: unit, e2e and race conditions, as described in the previous section. These tests can be run arbitrarily in parallel due to the previously generated test definitions. We did this because our end-to-end tests were taking far too long if executed in parallel.

Verify

This is the largest stage of all. This runs a few strict static analysis tools, using `golangci` and some static analysis tools integrated into Gitlab. Besides this, it also checks our dependencies for vulnerabilities and checks that our dependencies do not use a copyleft license. Finally this stage also aggregates all coverage reports created in the test stage to provide an overall coverage percentage metric.

Deploy

This final stage is only run on master, and builds the Docker containers and deploys them to the Gitlab and Docker Hub Docker registries. The latter is the registry which is publicly available and can thus be used by end users and by our CLI.

5.8.3 Documentation

Given the fact that we plan on open-sourcing Apate, we have made thorough documentation about how one can use it. This documentation is hosted by Github and can be found at `https://apatekubernetes.nl/`.