DEPARTMENT OF COMPUTER SCIENCE

Automation of RBAC Permissions in Kubernetes Clusters

Joshua Van Leeuwen

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering.

Monday 7th May, 2018

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of BSc in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Joshua Van Leeuwen, Monday 7th May, 2018

Abstract

With the mass adoption of cloud application containerization in industry, an emergence of container management systems and cluster technologies have followed. One such container and cluster management system is Kubernetes, an open source, community driven project that strives to provide its users with a portable platform that presents an abstraction of the cluster infrastructure and delivers tools to manage these clusters effectively. As comes with the added complexity of a distributed network of running components, so too does the implementation of security and access permissions of their interactions.

Kubernetes implements a system called Role-Based Access Control to enforce access permissions of components and users to various resources within its infrastructure. Within this system, I have identified three main problems that greatly affect the usability of the system leading to a significant increase in risk of jeopardising the integral security of the cluster. These three problems being the administration cost in ensuring proper access control of a scaling cluster, the static nature of permission declaration of subjects, as well as the lack of features to grant cluster operators with the ability to replicate permissions of subjects actively.

This dissertation investigates and develops a software solution to solve these problems whereby users are able to write simple rules that detail some permission declaration on subjects that dynamically react to changes within the cluster as well as time. This provides cluster operators with features that drastically reduce manual administration of access permissions and consolidates configuration into a manageable set of clear, extensible rules. Several techniques have been used to ensure functional correctness of the software as well as evaluation of the system.

Contents

Chapter 1

Motivation

1.1 Background

Containerisation within Linux systems provide an operating system level of virtualisation. Much like virtual machines, they offer isolation of applications running within the container from the host machine, although this is achieved through a different approach. Virtual machines rely on a hypervisor which is some software providing a virtual operating system platform that manages the execution of the virtual machines' operating system. Containers conversely, are run within the user space of the host machine, accessing basic operating system services through use of isolation of virtual memory. Since containers do not require their own operating system, they are able to be booted much faster than virtual machines. Moreover, containers are more resource efficient since their resources do not need to be permanently fixed.

The isolation of containers is achieved through use of two Linux kernel features; Namespaces and Control Groups. Since containers are simply Linux processes by definition, they can be subject under such Control Groups. A Control Group gives the ability to limit the resources (CPU, memory, network, etc.) that a set of processes has access to. This ensures that there is fine control of containers within the host machine, managing the scaling of multiple containers on one host appropriately. Namespaces are used to isolate and virtualise system resources of processes running within the Linux kernel. Again, since containers are processes running within the kernel, the memory, user space and network can be effectively isolated from its host as well as other containers.

Containerisation has been an ongoing development for many years, dating back to 1979 with the introduction of the *chroot* operation included in Version 7 Unix. This operation changes the apparent root directory of the current running process; an essential concept in achieving isolation through Namespaces. Today, many technologies provide deployment and management of containerised systems, although Docker is considered the leading solution [1]. Docker provides its users with a large amount of control over container management and deployment along with a massive user generated registry, boasting over 100,000 images and a large active community of contributors [2].

Although these container deployment systems offer a way of deployment and life cycle management of containers, more software is required in order to deploy and orchestrate any large scale containerised distributed system that can scale effectively. Google discussed its very own large scale container management solution *Borg* that has been in use for more than a decade [3], enabling many high availability online applications such as *Gmail* and *Google Docs* [4][5]. With this knowledge, some Google engineers who had previously worked on Borg began work on a new, open source management system, Kubernetes [6]. Through their knowledge and lessons learnt through internal development of containerised management systems, they were able to steer Kubernetes away from shortfalls previously encountered and help create the most popular solution today.

Kubernetes' ubiquity within this domain over its other contemporaries such as Docker Swarm [7] and Apaches' Mesos [8], can be partially contributed to Kubernetes massive community of open source contributors. As of the start of 2018, the core Kubernetes GitHub repository now totals over 1,500 contributors with hundreds more supporting projects within its ecosystem [9]. Being in constant development and receiving regular updates, Kubernetes can be considered a state of the art solution, making it an attractive option for industry. This is indeed the case, according to Portworx Annual Container Adoption Survey of 2017, Kubernetes is the most popular with 32% of participants stating that it is their primary orchestration tool [10]. OpenStack also announced that 47% of their users, who are deploying container orchestration or platform services, are using Kubernetes [11]. With 451 Research estimating application containers currently at a $1.5 billion industry as of 2018, increasing to $2.7 billion by 2020, Kubernetes represents a massive source of growth and business within the industry [12].

1.2 Technical Background

1.2.1 Kubernetes

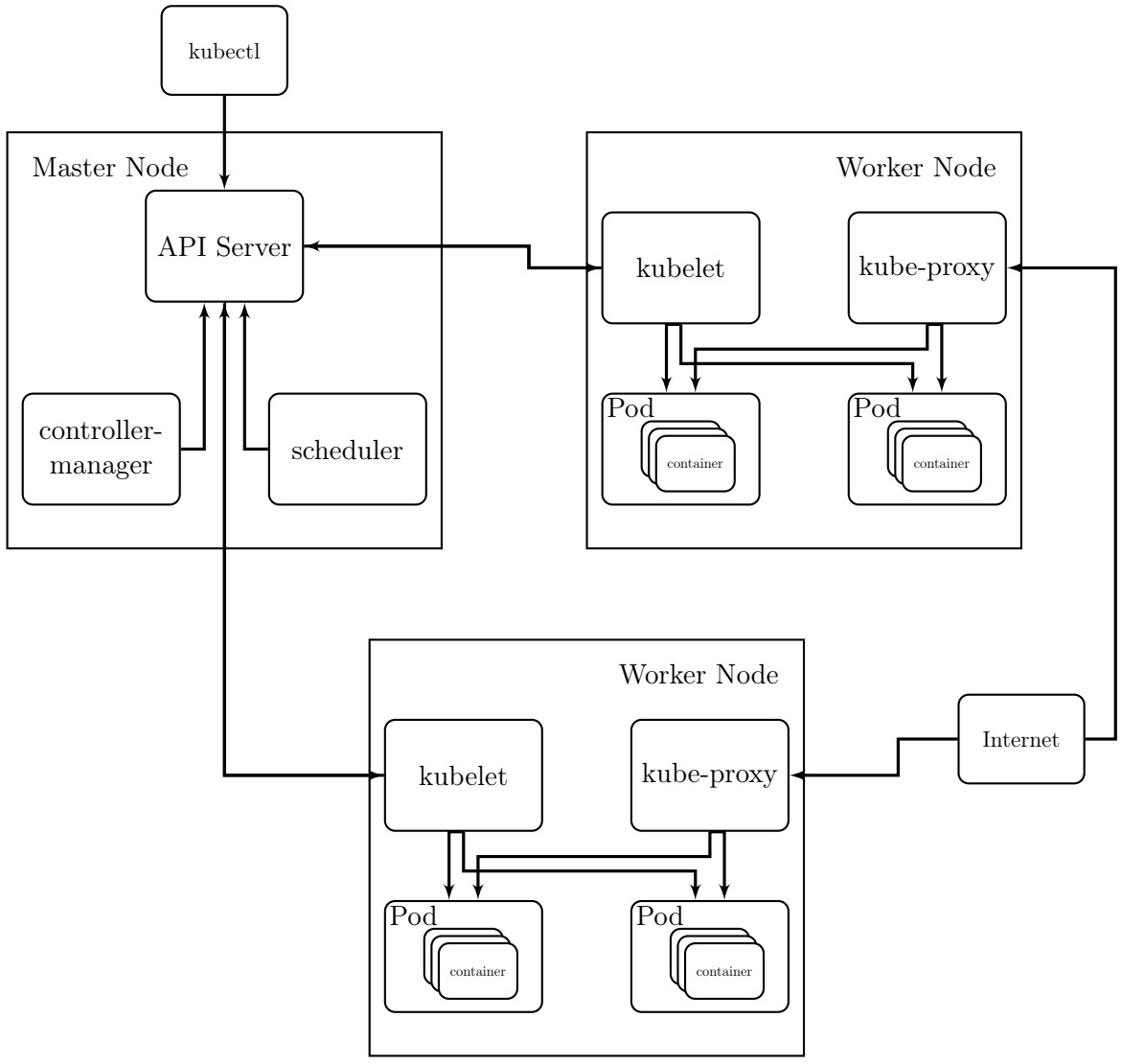

Figure 1.1: Kubernetes Topology [13]

In short, Kubernetes is a collection of components working together to create a portable,

extensible and self-healing distributed system [14]. Figure 1.1 shows a simplified representation of the Kubernetes architecture. The single *master node* controls the operations of the *worker nodes*. Worker nodes house a number of *pods* in which accommodate one or more containers, usually facilitated via Docker, in which the actual applications are executed. Each pod within the worker node is then able to communicate with the Internet through the use of a local process running on each worker node, *kube-proxy*.

The state of the entire system, or cluster, is constantly being checked for changes in its *current* state, and *desired* state. This is monitored through the *master node*, a collection of three main processes; *kube-apiserver*, *kube-controller-manager* and *kube-scheduler* [15]. The Application Programming Interface (API) server receives messages from a user to set the desired state of the cluster that create, manipulate or destroy internal resource objects. Communication of the API server is achieved through Representational State Transfer (REST) operations over some network, local or across the Internet . The *kube-controller-manager* is a daemon that executes controller loops, implementing the constant checking of the current state, attempting to make changes to converge toward the desired state . The *kube-scheduler* is responsible for where workloads should be assigned on the cluster, ensuring the best performance for all running resources. The master node is then able to communicate to all non-master nodes through the *kubelet* processes running on each [16] [17].

Although there are many more aspects of the workings of Kubernetes, these mechanisms will be the main basis of this thesis.

1.2.2 Role-Based Access Control

Kubernetes uses a technique called Role-Based Access Control (RBAC) to ensure authorization of communicating components across its distributed system, restricting their ability to perform different levels of actions according to admin specifications. RBAC was first devised in 1992 by David Ferraiolo and Richard Kuhn as a replacement for Discretionary Access Controls (DAC) that was used at the time for most commercial systems [18]. Whilst DAC usually relies on each resource having some owner who grants access to other subjects in the system, RBAC relies on roles that are assigned to subjects in order to gain access to those resources. In Kubernetes, this system is achieved through a series of default and custom *Roles* and *Cluster Roles*, detailing some set of actions which can be performed to some set of resources. While Roles are limited to granting permissions within some namespace, Cluster Roles conversely grant permissions across the whole cluster (all namespaces). The Role shown in Figure 1.2 defines a role named *pod-reader*, giving the ability to perform the actions *get*, *watch* and *list* on all *pod* resources within the namespace *development*.

```
1 kind: Role
2 apiVersion: rbac.authorization.k8s.io/v1
3 metadata:
4   namespace: development
5   name: pod-reader
6 rules:
7 - apiGroups: [""]         # "" indicates the core API group
8   resources: ["pods"]
9   verbs: ["get", "watch", "list"]
```

Figure 1.2: pod-reader Role [19]

This specification, along with all configurations that are accepted by Kubernetes, is writ-

ten in a file format called *YAML Ain't Markup Language* or *Yet Another Markup Language* (YAML). This format is a superset of JavaScript Object Notation (JSON), aimed at increasing the readability and human-friendly creation via its clear and minimalist design [20].

Role resource objects stored within Kubernetes are then used as a descriptor to grant permissions to a set of subjects; Users, Groups and Service Accounts. This delegation of linking roles to subjects is achieved via *Role Bindings* (namespaced) and *Cluster Role Bindings* (cluster wide, all namespaces). Figure 1.3 defines a Role Binding in which grants the user "jane" the permissions previously defined by the *pod-reader* role within the namespace *development*.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: development
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Figure 1.3: pod-reader Role Binding [21]

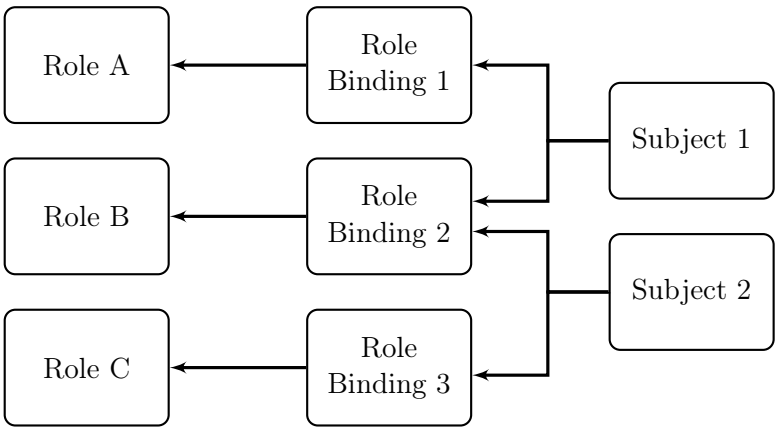

Figure 1.4: Example Kubernetes RBAC Topology

This system of templating permissions that are then linked to subjects is a powerful method of granting an administrator a great amount of granularity and control over all actions that can be performed within a cluster.

Figure 1.4 represents a topological representation of some RBAC configuration within a Kubernetes cluster. Here *Subject 1* has the permissions granted by role *a* and *b*, connected via *Role Binding 1* and *Role Binding 2*. *Subject 2* has the permissions *b* and *c*, connected via *Role Binding 2* and *Role Binding 3*.

1.3 Problem Definition

Although RBAC in Kubernetes works well when executed properly, it contains three main identifiable problems that need to be addressed in order to uphold a correct and usable system.

1.3.1 Administration

RBAC enables an administrator to only need to write roles once, which are then consequently stored as resource objects within the Kubernetes API. It is still required however, that every binding be created as and when they are required by the administrator; also to be stored as resource objects within Kubernetes. This introduces the first problem, forcing cluster operators to manually keep track of administrating bindings correctly and consistently throughout the cluster life cycle where new subjects become active and removed. This causes further overhead to the workload of an operator. This administrative work will always need to be done to a cluster however, with an increased complexity of a clusters topology, human error becomes a greater possibility. Bespoke solutions by numerous organisations will have been developed to solve this, however, there is no clear open source canonical way in solving this problem. There should exist some solution to consolidate Role Binding creations, automating permission delegation when scaling a cluster.

1.3.2 Dynamic Permissions

Permissions of subjects may be needed at different levels during its life cycle according to the needs of the application. Helm, for example, is a package manager for Kubernetes; giving the ability to deploy applications within a cluster in accordance to how the developer specifics in a simple and consistent way, removing complexity to its users [22]. To achieve this, the client (Helm) communicates with the corresponding server (Tiller) which then executes these requests. This server is most commonly contained within a pod (a logical collection of one or more containers) with a Service Account exposing a subject to be used to bind permissions for access to the API server. In order to have the ability to execute its requests, involving creation and management of many resource types within the cluster, it must be granted a high level of admin permissions. Logically, this is achieved through the use of applying a Role Binding to the Tiller server's Service Account, usually with the default role *cluster-admin*. After its operation has been completed, the server will still reside in an elevated permissions state, one which would have disastrous effects if it was able to be exploited maliciously. This server represents a security risk to the entire cluster. It is therefore paramount that the Role Binding to the server's Service Account is controlled appropriately via managing its permissions levels accordingly, this can however, be easily mishandled due to human error; leading to the second problem. There should exist some method to automatically prevent subjects residing in a highly elevated permissions state when not needed, to be defined in some user-friendly manner.

1.3.3 Permission Replication

When adding a new user to a cluster, it is often required that this new user is given permissions based on another. For example, if a new developer is added to an organisation, they will need a certain level of permissions that will match their colleagues. This requires manually sorting through all Role Binding of these users and matching the Role Binding to this new user. This is not only time consuming and prone to human error, but also susceptible to subjects that are desired to carry the same permissions, becoming unsynchronised from one another. The third problem to Kubernetes RBAC is therefore, no clear and user-friendly method to give a subject permissions that always match another. Whilst Kubernetes does implement an impersonation header for requests, the permissions for the subject are not changed and the request is instead acted upon as the impersonated subject, not the subject making the request. There should exist

some method where a cluster operator is able to enable subjects to synchronously use permissions of another.

1.4 Proposed Solution

1.4.1 Approaches

There are a number of options to develop a solution in automating RBAC permissions within a Kubernetes cluster, however firstly, I will not be changing the core of Kubernetes code base. By modifying the core code base of Kubernetes, end users will be forced to use modified binaries to run Kubernetes, presenting problems to versioning; every version release of Kubernetes will need another release of a modified version with the solution built in. This adds extra complexity to the end users experience as they must ensure that they, themselves, manage versioning of Kubernetes in line with releases. This method also means that solutions must be constantly maintained to match versions. This goes against the canonical ethos of how Kubernetes should be used and developed for. Older versions of resource objects for example, are still supported in newer versions within the system. A Kubernetes client can be of a different version number to the API server's whilst still working correctly.

It is therefore most ideal for the solution to be written using software which, is itself, a different component to the core Kubetnetes code base and will instead communicate with the distributed system separately. This communication should be conducted through the API server since this is the main hub of all communications between all Kubernetes components and will give the software the ability to execute the changes to RBAC permissions in the canonical way.

With this, the software will need now be able to accept rules by the user which then needs to be acted upon. To achieve this there must exist some program that is constantly running that will be able to monitor for when some rule has been met. In this case, there has to be another separate method of sending this program control messages for manipulating the rules. This can be achieved in several different approaches.

1.4.1.1 Local Filling System

One method is to create a user maintained file that is constantly checked by the software. This may be the easiest method for developing the software since little networking is needed in order to achieve and has little complexity. Nevertheless, it is by far the worst method. Implementing this option would involve the software to be constantly reading from file and assessing whether anything has changed by means of the user. This method creates the possibility of the software and user reading or writing a file at the same time which may cause the software to become out of sync with the users wants or worse, corruption of data.

1.4.1.2 Inter-Process Communication

The software listens to some inter-process communication, such as shared memory, which is used to send messages from some second user program. This option is better than the last as it ensures that the flow of communication between the user and the controlling software does not suffer from the problems of concurrent read/write communication as communication can be forced one way. This can be controlled through simple buffers and semaphore channel locks. As mentioned, this method is better than the previous although suffers from the problem that if the controlling software exits or crashes for what ever reason, the state it was in will be lost. This means that any stateful data such as current rules in action or state of permissions will be lost (without any supporting persistent storage). This again leads to the need of some file storage which is not ideal, leading to more user administrating overhead to ensure the program is maintained as intended. This always means that the user and controller will have to be run on the same

machine since it is using inter-process communication which is itself, not desirable for managing a remote distributed system.

1.4.1.3 Networked Communication

Some software listens to some Transmission Control Protocol (TCP) socket that, not only locally, can be used to receive messages remotely over the internet. This option is the most ideal for the user experience of managing a distributed system that most likely manages such system remotely. Designing such a system in this way also means the software can also take utility that the user could communicate with the software through the API server instead, something which is already familiar with current users. With this, the software is able to utilise many extra components that Kubernetes has to offer.

Kubernetes contains a feature known as *Custom Resource Definitions* in which the user is able to extend the API server to accommodate a new user defined resource type. The API will then watch this resource, and like any other resource, other components can interact with these objects. This will be the method to which the solution software will communicate with the user.

1.4.2 Software Design

The solution to the aforementioned problems will be solved using some software that communicates directly with the Kubernetes API server, constantly watching for any changes that the user has made to the desired state. The user will then use the standard tools available to communicate these changes of the desired state. The automation of RBAC permissions will be delegated through user written rules containing event and time based triggers, that when all are met, will execute some RBAC based event within the cluster. These rules will be, by design, highly configurable, giving the user the ability to create completely bespoke rules that meet their specific needs.

1.4.2.1 Custom Resource Definition

The user written rules will be represented by a custom resource definition consisting of 5 segments:

- Header: metadata of rule consisting of the name of the rule and the namespace the rule is active in.
- Options: detailing the number of times the rule should be repeated.
- Origin Subject: the origin subject as to where the permissions originates from. This could be a Group, User, Service Account or simply a Role.
- Destination Subjects: a list of one or more destination subjects that can be any one of a Group, User or Service Account.
- Triggers: a list of one or more triggers which will need to be satisfied in order for the RBAC delegation to be executed and removed. This can be in the form of some time based requirement or any event relating to resources within the cluster.

These rules will then be written by the user as a YAML file that is then sent to the API server.

1.4.2.2 Controller

Kubernetes implements several internal controllers that are used to constantly check the current and desired state of Kubernetes. Controllers follow a general implementation of an infinite loop shown in Figure 1.5.

```
for {
  currentState := GetCurrentState()
  desiredState := GetDesiredState()
  MakeChanges(CurrentState, DesiredState)
}
```

Figure 1.5: Kubernetes Controller in Golang Pseudo Code [23]

Although these controllers are written for other operations within Kubernetes, they are all based from the same Kubernetes client library, *client-go*, containing frameworks for building multiple client components within Kubernetes [24]. By utilising Kubernetes libraries, not only will less work need to be done in order to achieve proper functionality, there will be confidence that the controller will have perfect compatibility with Kubernetes itself.

This solution will implement one of these controllers, configured to listen for changes to the desired and current state of the custom resource objects. When a user summits a new rule to the API server the controller will catch a change of desired state to the custom resource definition. When the controller receives one of these new rules, it will parse its information and begin to compute its contents.

As with many of the triggers within rules, they require that the software is also aware of other resource changes within the cluster such as a pod being created or terminated. In order to also keep track of arbitrary resource changes, controllers need to be dynamically created and destroyed to accommodate such requirements.

1.5 Requirements and Design Decisions

In order to ensure a full and complete solution, a clear scope and list of limitations must be outlined.

A list of requirements that have to be met in order to meet a successful solution;

- The solution should be functionally correct, in accordance to executing user requests as expected. This will be achieved through a series of different kinds of testing.
- The software will be written in Go. Since Kubernetes is written in Go, I will also be writing the software in Go to utilise its internal libraries.
- Software source code must become open sourced. Not only does this keep in accordance to Kubernetes itself being open source, but for any user to have confidence in software that directly manages their security, they may wish to personally check it is functionally sound.
- Produce detailed documentation in order for users to use the software.

A list of design decisions which will be adhered to when implementing the software:

- Use of best practices when writing code according to Golangs official documentation [25].
- Structure features of the software in self contained modules. Contributions in the future will be able to be added much easier by the open source community, a characteristic which helps the software increase its usability for a larger range of use cases.
- Use of object oriented programming paradigms.
- Software will be targeted towards the Linux and OSX operating systems.

1.6 Aims and Objectives

This thesis aims to research, develop and deploy an application that will solve the aforementioned problems. This application is to be targeted toward the Kubernetes open source community whereby its users are cluster operators who are familiar with Kubernetes and its available tools as well as ensuing future development is community feedback driven. A focus will be made with regards to the applications ability for ease of open source contributions as to maximise its feature set and broaden the scope of real community use cases. Evaluation of the success of this application will rely on its functional correctness of the software and the feedback given by the community.

This application is designed to be used in conjunction with Kubernetes as an external tool, not as a replacement of any existing functionality. For example, the application will not be replacing the current Kubernetes RBAC system in anyway in favour of providing tools to improve the management of this system. Due to the nature of the goals and context of the thesis, quantitative data of user surveys or performance benchmarks will not be used as a bases for the main evaluation and is outside of scope.

Chapter 2

Design

2.1 User Written Rules

In order to create a user experience that aims to reduce user input and provide clear but extensible control over rules and their outcomes, a suitable standard must be adhered. Not only will a clear standard make reading existing rules easier, but users creating and extending new rules will be more efficient and reduce the risk of erroneous or undesired outcomes when learning the syntax of the standard.

Firstly, field names of the resource type will be using the convention of camel case where the first letter of the field name is lower case, with all subsequent beginnings of following words will be abbreviated to no space and a capital letter, for example, *Origin Subject* receives the abbreviated field name *originSubject*. This convention follows that of Kubernetes' built in resource field names that a user familiar with Kubernetes will be expecting.

Declaring the 'kind' or type of a particular segment of the object will follow standard camel casing with the first letter also being capital. This again follows Kubernetes' own standards for a kind, for example *kind: ServiceAccount*. A trigger with the type of a new pod being added to the cluster will therefore be shown as *kind: AddPod*. Moreover, designating naming in this declarative way provides a clear and concise meaning to what is being written within the configuration whereby, without any prior reading of the documentation of writing correct configuration, the correct meaning can be inferred; 'trigger with kind addPod, name nginx' can be easily inferred as 'a trigger is triggered by a pod being added named nginx'.

For the rules to satisfy the goal of reducing the amount of user written configuration to help in administration, it is important for the amount input required by the user to be as minimal as possible. As such, each field requires only the minimal information needed to execute the rule so, for most segments of the rule, only the name and kind are required. Details of the namespace for example, can be omitted as this is inferred from the namespace declared within the header of the rule.

Finally, the rules are designed to be extensible and highly configurable. Lists of each segment, such as activation triggers, are collections of self contained modules so that any composition of selection of options is valid and will execute as expected. This gives a vast amount of user control and granularity over the execution and intended outcome of each rule, accommodating broad use cases.

2.2 Interfaces

One the main goals for the software is engagement with the open source community with the inclusion of contributions. As such, the software has been designed in such a way that a community member is able to easily contribute code to the main code base without major understanding of the entire software stack and architectural topology. To achieve this, the code structure is

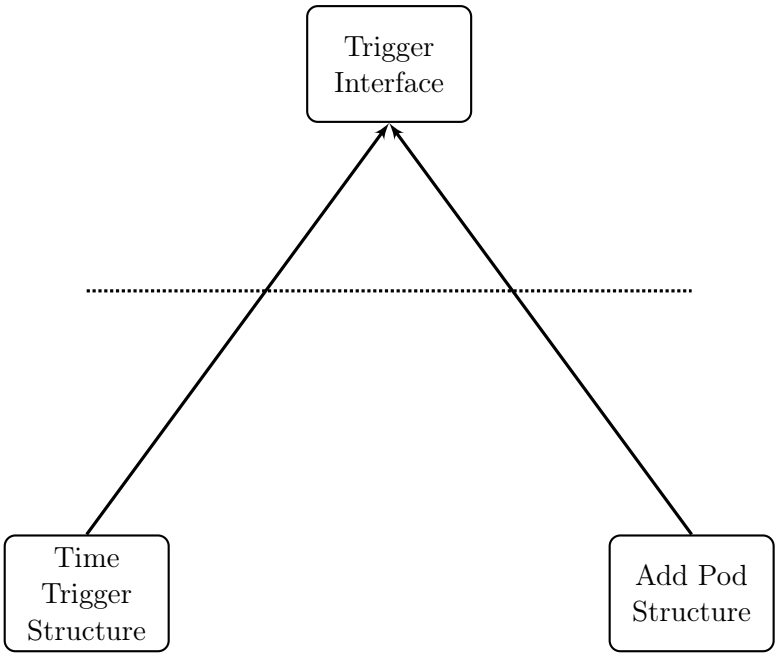

Figure 2.1: Trigger Interface implementing Time and Add Pod Trigger

extremely modular meaning that each logically similar component is packaged into appropriate groups. In order to realise this, a major feature of the Go language has been utilised; Interfaces.

Go is a strongly typed language meaning that each variable or structure data type is predefined and set once declared that is checked statically at compile time. Go however provides a way to obfuscate away these strongly typed variables and provide runtime polymorphism through use of the special data type *Interface*. Through requiring a set of functions that a receiver must satisfy, the interface is able to obfuscate the strongly typed structure or type that the interface is representing underneath. For example, a *Shape* interface can implement several shape like, strongly typed, structures such as *Square* and *Triangle* so long as they satisfy some defined set of functions. Code is therefore able to deal with a single abstract shape interface as opposed to writing specific code for all structure types.

This is invaluable for the project software since there are a large number of structure types that can be simplified down to a single interface type for a single entry point to the structure. One such example is the *Trigger* interface that implements the many number of trigger types within rules. Shown in Figure 2.1 is an example of two such examples. Here both structure types need only implement the set of functions within the interface definition in Figure 2.2 for the structure to be granted as a valid Trigger interface.

```
62 type Trigger interface {
63     Activate()
64     Completed() bool
65     WaitOn() (forcedClosed bool)
66     Delete() error
67     Replicas() int
68     Kind() string
69   }
```

Figure 2.2: Trigger Interface [*pkg/interfaces/interfaces.go*]

For any contributor wishing to implement their own Trigger type they simply need to only satisfy these set of functions and they will be able to include it into the set of available trigger types. This greatly increases the scope of functionality that a contributor is able to implement within the system, catering to larger use cases that are not already present in the base release of the software.

2.3 Code Generation

In order to interact with the custom resources created by the user the controller requires a client in order to interface against the API server. Creating such clients itself can be a large undertaking and can be a source of large amounts of bugs within the code base. In order to alleviate this problem, a set of code generators has been developed by the Kubernetes developers to automate the process of creating such clients [26]. Although code generation has been used within this software to provide key functionality, developing a code generator is itself a large topic which is outside the scope of this thesis.

Multiple generators are to be used in conjunction to generate a fully featured client, all of which can be found within the binary directory *bin/*;

- *deepcopy-gen*: generates all deep copy functions for copying custom resource API objects.
- *client-gen*: generates typed client side API calls for the custom resource types.
- *informer-gen*: generates informers in which provide event based hooks on the custom resource type.
- *lister-gen*: generates API calls for GET and LIST to provide a read-only caching layer.

Together they produce a native like interface to the custom resource just like any other resource within the Kubernetes cluster.

In order to generate these packages they must first take in as input the targeted custom resource type which is wished to be created and used within the API server. As such, a *types.go* file is created, written within the *pkg/apis/authz/v1alpha1* package directory. Here is the source of the typed definition for the Subject Access Delegation custom resource type. Figure 2.3 contains a snippet of this file containing the main definition of the object.

Two features of the file that are of note are the use of both the compiler directive tags as well as the JSON tags next to a select number of definition elements. The directives declared within comments at the head of the file detail options which are to be read by the code generators. Here they are stating to generate a client with no Status sub resource, a feature not yet available to custom resources, generate deep copy functions for all object structures and declaring the resource path for the custom resource to reside within the local software packaging.

The JSON tagging of elements within the resource is used to declare how that element is to be displayed or titled within YAML files, that is, what the user will use to input values to the rule or how a rule is to be displayed when describing resource objects of this type. This gives the ability to enforce naming that is more intuitive when writing user rules instead of using Go's structure naming conventions.

With the generated code now created and included into the applications packaging, the controller now has access to a native like API interface that can be easily used in conjunction with the existing Kubernetes object APIs, providing coherence for development on both native and custom resource objects.

```
// +genclient
// +genclient:noStatus
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime .Object
// +resource:path=subjectaccessdelegation

type SubjectAccessDelegation struct {
        metav1.TypeMeta   `json:",inline"`
        metav1.ObjectMeta `json:"metadata,omitempty"`
        Spec   SubjectAccessDelegationSpec   `json:"spec"`
        Status SubjectAccessDelegationStatus `json:"status"`
}
type OriginSubject struct {
        Kind string `json:"kind"`
        Name string `json:"name"`
}
type DestinationSubject struct {
        Kind string `json:"kind"`
        Name string `json:"name"`
}
type EventTrigger struct {
        Kind      string `json:"kind"`
        Value     string `json:"value"`
        Replicas  int    `json:"replicas"`
        UID       int
        Triggered bool
}
type SubjectAccessDelegationSpec struct {
        Repeat int `json:"repeat"`

        OriginSubject       OriginSubject        `json:"originSubject"`
        DestinationSubjects []DestinationSubject `json:"destinationSubjects"`
        EventTriggers       []EventTrigger       `json:"triggers"`
        DeletionTriggers    []EventTrigger       `json:"deletionTriggers"`
}
type SubjectAccessDelegationStatus struct {
        Processed     bool `json:"processed"`
        Iteration     int
        TimeActivated int64
        TimeFired     int64
        RoleBindings        []string
        ClusterRoleBindings []string
}
```

Figure 2.3: Subject Access Delegation Custom Resource Definition [*pkg/apis/authz/v1alpha1/types.go*]

2.4 Network Time Protocol

With the potential for time based triggers to be used extensively within the application by users, it is paramount that the internal clock being used is correct. As such, a mechanism should be included to provide users with the ability to control the internal clock of the system. The reason as to why the user may want to ensure that a time clock is consistent to their expectations is due to unreliability of automated remote instances. The deployed cluster in which the controller is held in or any plain instance that the controller is deployed in may have an unreliable clock. The user may be distrustful of the target instance where automated delivery means little or no checking of instance time checking is conducted.

One solution to this problem is for the application to change the host's machines internal clock. Although this method would work, it would mean that the software would need root access within the host machine. This is something that is undesirable for end users. Instead, the application itself should hold an internal offset from the host machines internal clock in which is used to calculate the desired time from the machines.

In order to obtain this offset an external time source is needed to provide a time synchronisation. One such example is the use of the Network Time Protocol (NTP) [27]. NTP, first developed around 1980 by David L. Mills at the University of Delaware, is an internet protocol used to ensure synchronisation of time of many computers within a network [28]. It is based on a concept of a hierarchy of several layers known as Stratums that house time provider computers. The first Stratum, Stratum 1, is the highest in the hierarchy consisting of a single time provider that is directly connected to a trusted time source. Unlike all other time providers in the hierarchy, this connection is not over a network and is instead directly connected. Stratum 1 time providers are typically not accessible to the public and are instead only reachable by the Stratum 2 time providers. The time kept by the Stratum 1 time provider will then, on request, return its current time that is consequently propagated through the Stratum hierarchy. A NTP client will typically request time from a Stratum 2 provider whereas larger Stratum layers are used in internal networks where providers will synchronise with each other to reduce load on the public Stratum 2 time provider.

Using NTP server URLs parsed as command line flags to the application, the software is able to gather the time stamps of the current time from those servers. With one or more results from these servers, the application is then able to average the results and is able to capture an accurate time to use internally.

In order to increase the security of the user, the NTP client will send a random transmit time when requesting times from NTP servers. This is in an effort to increase privacy and reduce the ability of spoofing of the client.

2.5 Residence of Application

The application operates with no requirements for persistent storage through use of the API server being the single source of truth of all resource objects, including delegation objects. This designates all resource objects within the cluster are, in effect, stored once within the API server; any and all objects outside of this component are considered only reference to the 'real' object. As such, the application is a good candidate to be deployed within the Kubernetes cluster as a Deployment. By running within the Kubernetes cluster the application will have the option to either consume a user configuration file to establish a connection with the API server, or instead, use the *kubelet* component present with the worker node.

The application running within the Kubernetes cluster itself grants a number of benefits that are not available or difficult to implement were the controller to be run outside of the cluster entirely. Firstly, since the application is run from some deployment it is subject to auto healing features that Kubernetes implements on these resources. This grantees any user of the software

high availability of the application where Kubernetes will attempt to orchestrate a revival of the software if the software falls into an erroneous state. High availability is a crucial requirement for a security critical system.

Secondly, a number of tools are available for monitoring various resources within a Kubernetes cluster. One such example is Prometheus, an open source project that aims to provide its users with a vast set of monitoring solutions for numerous metrics [29]. Prometheus enables its users to detail a number of queries to gather up-to-date metrics on some system and make decisions on whether to make various sets of alarms to it users on the state of that system. This means Prometheus is able to query the Kubernetes API server and gather metrics of the health status of all resources within the cluster. If the Subject Access Delegation controller became in an unhealthy state consistently then this will be notified to the cluster operator who would be able to perform some manual action to solve this. Again, a crucial feature that is accomplished through the software residing within the cluster itself.

Chapter 3

Implementation

3.1 Execution Flow

To facilitate a stable system that provides scaling and a dynamic topology of executing components, a clear execution flow must be established. The structure of the application falls into a hierarchy where each component within it communicates directly with the component that is directly higher. The components within this hierarchy are as follows;

- Controller: the main software component which is run by the user at runtime.
- Subject Access Delegation Handler: component created by the controller for every rule created by the user.
- Origin Subject Handler: the single component created by the Subject Access Delegation handler for the single Origin Subject of the rule.
- Trigger Handler: created by the Subject Access Delegation handler for each activation and deletion trigger of the rule.

Figure 3.1 shows a diagrammatic view of this hierarchy. Each component is run concurrently with one another and as such, a number of synchronisation points are used.

When the application is first run the main controller component is created which completes a number of tasks during initialisation. Firstly the controller will establish a connection with the API server which when is achieved, ensures the custom resource definition of the Subject Access Delegation resource type is present in the API server, creating it if it is not. Secondly, the controller will then begin to gather the timestamps from NTP server URLs from any that were given to the application. Upon receiving the result, the controller workers which are used to process each delegation received from the API server are started, finally gathering any Subject Access Delegation objects which are already present in the API server.

With the main controller now ready, it will begin to listen for any new Subject Access Delegation objects from the API server. Once a user has written a rule and submitted it to the API server, the controller listening to this resource type will receive this object from the API server. Upon receiving, the controller will create a new Subject Access Delegation handler and pass the resource object to it. This handler is then executed concurrently for the controller to be free to continue to listen for more delegation objects. It is now the delegation handler's responsibility for this users delegation.

With the delegation handler now created, the contents of the user written rule are now parsed and its contents inspected to determine which and how many of each type of component should be created. The Origin Subject component of the appropriate type is created. This Origin Subject also creates its own informers to run concurrently with this delegation handler, this is for listening to updates to the Origin Subject permissions of that delegation. Destination subjects

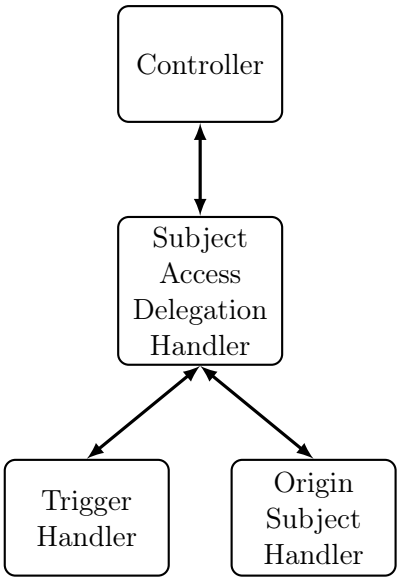

Figure 3.1: Application Hierarchy

are determined and finally the Trigger components are created. Each trigger component is to be run concurrently and requires its own set of informers for their relevant trigger requirement.

Once a successful delegation has been parsed by the delegation handler, with all required components created, the delegation can be activated. This involves starting all activation triggers to begin to wait for their individual requirements to be met. Since these triggers are executing concurrently to the delegation handler, a blocking mechanism is used in the form of channels. A channel is created within each trigger whereby the delegation handler can loop through until all channels have been closed due to the requirement met by the trigger.

Once all activation triggers have been satisfied and are no longer blocking, the delegation handler will execute the relevant RBAC permissions in the cluster. The handler will begin by retrieving all permissions that are assigned to the Origin Subject from that component. New Role Bindings and Cluster Role Bindings will then be created which replicate these permissions, setting the subjects of these bindings to the Destination Subjects and finally transmitting these bindings to the API to be created. The delegation has now been fully applied within the cluster and so now the delegation handler will activate the deletion triggers, which once have also been satisfied, will then remove all bindings that were created from the API server, retracting all permissions that were granted to the Destination Subjects.

This algorithm will repeat until all specified iterations in the rule have been completed, when finally, the delegation will finish and the Subject Access Delegation resource object will be deleted from the API server by the controller. An example of the components in action is shown in Figure 3.2 that demonstrate the components executing independently within the hierarchical structure of the application.

3.2 Concurrency

As is shown in Figure 3.2, the application must facilitate execution of the various dynamically created components, as and when they are required, at the same time as one another. To achieve this, the use of Go's *goroutines* are used through out. A goroutine is a lightweight thread that is executed within the same address space as its host meaning all memory within scope is also addressable to this thread.

3.3 Failure Recovery

If, for whatever reason, the application fails and the API server becomes unreachable due to a crash or network error, the permissions state within the cluster will drift into an undesired state. Once the application has reconnected to the API sever the application will need to converge the permissions state into the desired. As such, not only must the application itself need to keep state of the current set of rules in waiting to be fired as well as fired, so must also the API server. This is so, if either fail, they are able to resynchronise at reconnection.

To achieve this, the custom Subject Access Delegation resource object must hold state of the rule it contains, specifically;

- Processed: whether the rule has been consumed by the controller and has been handed over to a new Subject Access Delegation handler.
- Iteration: the current iteration of rule of the total replicas.
- Triggered: the fired sate of each trigger.
- Time Activated: the time in which the trigger or rule has been activated in that iteration.
- Time Fired: the time in which all triggers of the rule were met.
- Role Bindings and Cluster Role Bindings: to keep track of all bindings that have been created by the application.

Each of these fields can be seen in the code snippet shown in Figure 2.3 where the custom resource object has been defined. These fields are not designed to be manipulated by the user and as such have not been given display naming JSON tags, instead they should only be controlled through the Subject Access Delegation handler.

During the life cycle of a delegation, each executing component of the delegation will be continuously updating the status of the rule by passing to the handler the relevant data that needs to be manipulated to be stored with in the API server. The resource object stored within the API server will therefore hold a continuous record of the current state of the delegation. After failure, the delegation handler is able to restore the state by propagating the resource objects data to the relevant components, continuing with the delegation.

Failure recovery is a key feature that is required in order to work in conjunction with Kubernetes auto-healing properties.

3.4 Dynamic Permissions

Dynamic permissions of subjects under delegation require monitoring of the current state of permissions of all relevant subjects at all times during execution. As such, not only must the application contain some state of the permission at activation of the delegation, but also be able to react to any changes in permissions over its life cycle. In order to ensure a reactive delegation of permissions each Origin Subject component of its delegation adopts the responsibility of monitoring their subject for changes in its Bindings.

To monitor these changes each Origin Subject Handler, when created, will create a new informer which attaches informer function hooks to listen to updates of Role Bindings in the delegated namespace and Cluster Role Bindings cluster wide. These functions come in the form of, *Add*, *Delete* and *Update*, corresponding to the three equally named verbs that can be used to manipulate objects within the API server. If any changes to Role Bindings within the cluster occurs then the handler will capture this new object and determine whether one of the effected subjects is the Origin Subject of the delegation. If this is the case, the appropriate update needs to be passed up to the delegation handler to be applied accordingly.

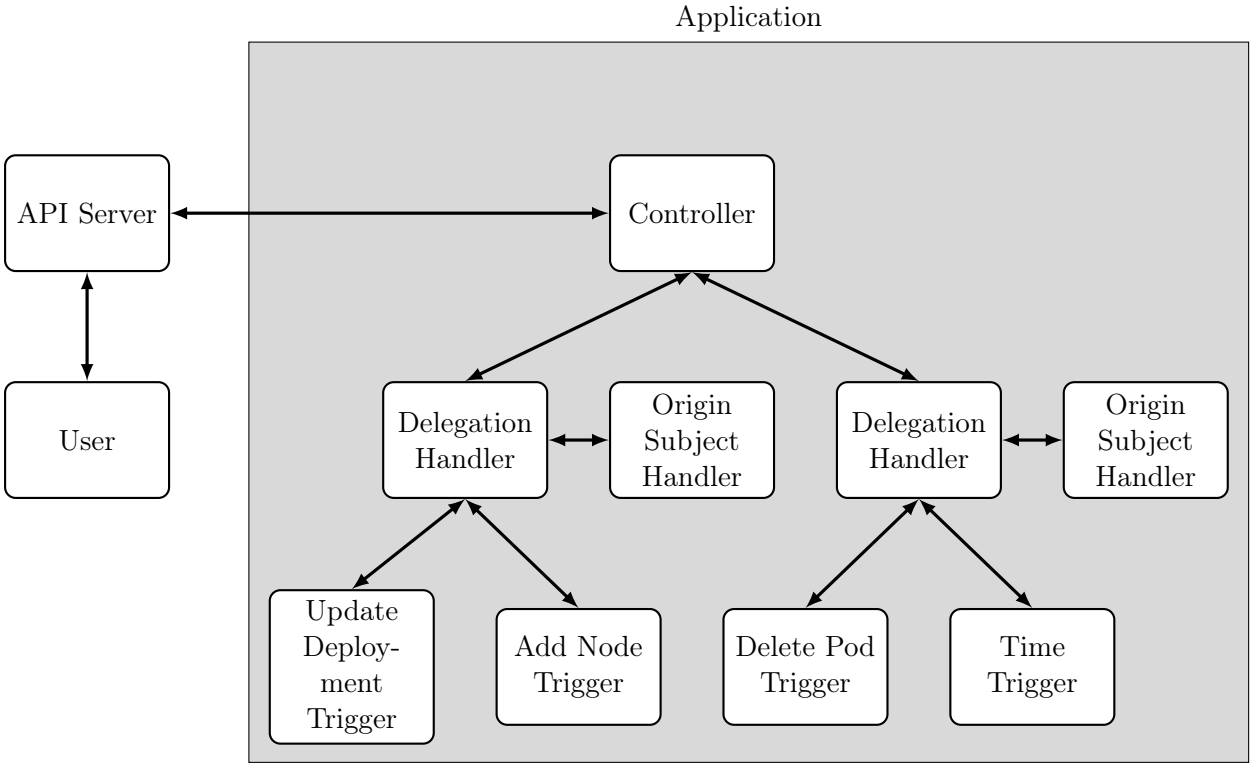

Figure 3.2: Diagrammatic Topology of the Software

In order to ensure no binding is erroneously or over reported to the delegation handler the Origin Subject handler conducts a number of checks once a new object is received, namely whether the Binding contains the Origin Subject and whether the binding has been seen before. Checking the binding subject is to obviously ignore all bindings that do not concern the Subject Access Delegation, whereas, the reason to check whether the binding has been seen before is due to the behaviour of the informer in various scenarios.

The API will trigger the Informer to call the handling functions multiple times during any objects life cycle. Moreover, the informer will call the *Add* function once a connection has first been established with the API sever for each object of its type that are already present in the API server. This means that, for example, if a newly created Origin Subject handler within an already activated rule establishes an informer with the API server, the handler will receive all bindings that may contain bindings pointing to the Origin Subject of the delegation. The handler will then attempt to relay these bindings to the main delegation handler in an attempt the replicate them to the Destination Subject. Since this is an already activate delegation, these replicated bindings already exist and will cause a series of errors where duplicated bindings are attempted to be created multiple times. This problem is not limited to this specific example and in fact will cause a mass of errors in many delegation states resulting in a myriad of problems that produce widely unpredictable outcomes.

To resolve this issue, a record of all bindings seen by the Origin Subject Handler is stored. Using this set of seen binding objects the handler is able to determine what the handler should do with the binding once it has been collected from an informer call. In the case of a newly created handler in an already active delegation, the handler will have already gathered the set of bindings which are present in the cluster so the bindings collected from the *Add* informer hook will be ignored.

The values to which denote the references to the bindings seen by the Handler are stored as the Unique Identifier (UID) type of the resource object. By using this value, this ensures that

even an update resource object will still be identifiable by the handler, regardless of the objects contents.

A demonstration of an Origin Subject Handler in action, performing a reactive permission update is illustrated in Figure 3.3. Here stage *1* of the diagram represents a topology of some active rule within a cluster where the left of the container represents a configuration of RBAC permissions. Here the Origin Subject has been bound to Role A through Role Binding 1 which has been replicated to the Destination Subjects with Role Binding 2. To the right represents the Origin Subject Handler listening to Binding updates within the cluster through use of an informer connected to the API server.

In the following stage *2*, *Role Binding 1* has been deleted from the cluster. This deleted Role Binding object is then sent to the Origin Subject Handler's informer from the API server. Since this Role Binding is pointing to the Origin Subject and the resource object has been seen before, the Origin Subject Handler passes on to the Subject Access Delegation Handler the corresponding replicated Role Binding pointing to the Destination Subjects for deletion. Upon receiving this, the Subject Access Delegation Handler then executes this deletion to the API server and thus is deleted from the cluster. The resulting state of the permissions is therefore represented to the left where no Role Bindings are attached to the Origin Subject or Destination Subjects. The Origin Subject Handler will have received the deletion of *Role Binding 2* to its informer as well, but, since both of the disjunctive conditions of referencing the Origin Subject as well as not being marked as deleted are false, the Origin Subject Handler does not take any further action.

Within the final stage *3*, a new Role Binding, *Role Binding 3*, has been created with its subject pointing to the Origin Subject. This new binding object is then received from the Origin Subject Handler from the API server to its informer. Both conjunctive conditions of this new binding object has a subject pointing to the Origin Subject as well as the resource object not been seen before by the Origin Subject Handler, the handler concludes that a new Role Binding replica must be created for the Destination Subjects. A new Role Binding is created whose permissions match that of the received binding, pointing to the Destination Subjects, and the appropriate name is applied in accordance to the Subject Access Delegation naming convention. This new binding is passed onto the Subject Access Delegation Handler for creation who finally executes this action against the API server. This new Role Binding, *Role Binding 4*, is therefore created in the cluster which can be seen in the left of this container, and as shown, is a replication of *Role Binding 3*, except, bound to both Destination Subjects.

3.5 Updating Subject Access Delegation Resource Objects

Whilst creation and deletion of Subject Access Delegation resources can be handled in a sequential manner within the delegation handler, the last verb to be dealt with by the controller, updating the resource, requires an involved set of logic in order to deal with updates by the user. With the main controller listening to changes of each Subject Access Delegation, it will pass the updated resource object to the relevant Subject Access Delegation handler. This handler will then make several decisions on what action should be taken within the cluster.

Updates of the resource object are broken into 4 main groups of what has changed in the update of the resource object;

- One or more Origin or Destination Subjects have been changed.
- One or more Triggers have been changed.
- Some Subjects and Triggers have been changed.
- No Subjects or Triggers have been changed.

Determining what group this update falls into is then handled appropriately depending on what stage in the deletion the rule is currently residing in.

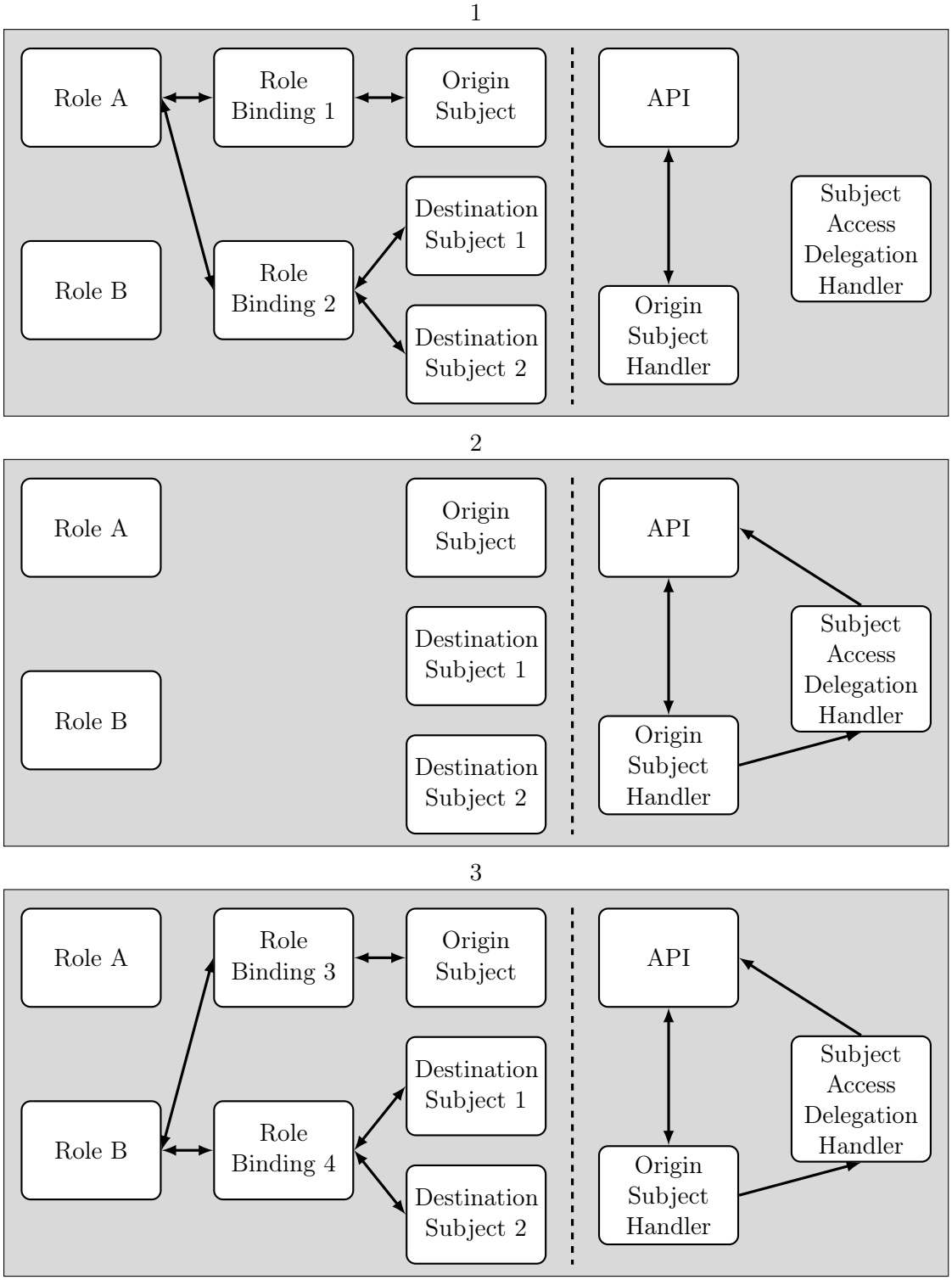

Figure 3.3: Dynamic Role Binding Permission Flow

For a rule which is not currently active, as in, the activation triggers have not been fired, nothing is needed to be changed in terms of permissions for any subjects. If any subject have been changed with the updated rule, only subjects are updated locally. If any activation triggers have been updated then all activation triggers of the rule are stopped, and all triggers are activated again, using the new set of triggers.

For a rule which has been activated, in that all activation triggers of the rule have been fired, if any subject has been changed in the updated rule and has been changed then a permissions change must take place within the cluster. This process follows in the form of removing all bindings currently applied in the cluster by the Subject Access Delegation handler are removed and then reapplied. If any deletion triggers have been changed, the rule is kept within the delegated state and these triggers are reactivated, in wait to be fired.

Chapter 4

Testing

For system security critical systems it is paramount that it is completely functionally correct such that all operation has an expected outcome to the system operator. As such, several kinds of testing must be implemented and performed on the system at different abstracted levels of the application's execution. The software implements two main methodologies of testing, unit and end to end testing. Whilst unit testing aims to verify functional correctness of a particular component or package of the system, end to end testing strives to verify functional correctness of several components interacting with one another, in particular, the entire software stack over its execution life cycle.

4.1 Unit Testing

In order to verify key components of the software, unit tests have been devised to test each logical package. In order to facilitate the framework to create an appropriate testing suite, the built in testing package has been used, as is best practice [30]. This package provides a concise API to Go's built in testing mechanisms, namely the *go test* command line executable. This actively seeks unit tests within the passed files and compiles and runs the appropriate target code at runtime whilst the compiler will ignore these tests files. This gives writing and continuous testing a native action within development cycles.

4.1.1 Mocking

Whilst almost all code that has no networking component can be tested within standard self contained tests, networked code becomes challenging to properly test. This is due to the necessity for not only some receiver to call API requests to, but also, forcing some kind of edge and corner case states of the application. As such, another framework is needed to mock a live system to interact with.

To solve this problem the Golang maintainers themselves have developed a framework to effectively mock such networked interfaces [31]. In fact, this framework is able to mock any interface type within a code base. Its strategy to mocking interfaces is by first through use of a code generator, *mockgen*. This generator takes as input some Go package directory along with one or more interfaces that are wished to be mocked. Shown in Figure 4.1 is an example of one such generation used in the application. Here mocked interfaces are created from the core resource Kubernetes package, namely the *CoreV1Interface*, *ServiceAccountInterface* and *PodInterface* interfaces. This generated mocking code is then written to the applications own mocking package directory, storing all mocking interfaces needed for testing.

The code generation will create its own internal structure that implements all required interface functions in order to have the target interface implemented. With this, the new structure

```
114   bin/mockgen $(CLIENTGoCore)
      ↪ CoreV1Interface,ServiceAccountInterface,PodInterface >
      ↪ $(MOCKDIR)/core_v1.go
```

Figure 4.1: Makefile mockgen [*Makefile*]

is now ready to be used to strongly type the obfuscated structure behind the interface to be mocked.

Along with creating this dummy structure that will implement the target interface, this structure also provides a number of calls that can be used to control computation of said structure through use of an additional 'Expect()' member function. With this, the testing code is able to specify the expected function call from the target code under verification with some additional controlling parameters such as number of times the function is expected to be called, order of computation or even execution of some other arbitrary function call. Finally the mocking structure will return some return values that can be controlled by the testing suite. Through a combination of control over expected function calls and return values the testing suite is able to verify execution flow whilst ensuring proper testing of edge cases that could occur when calling API functions.

A simple example of use of the mocking system within one of the application's many test files is the following example shown in Figure 4.2 in which is a code snippet of one of the tests for a Group Origin Subject. Here, by returning an error to the API call we are able to test that the unit deals with it as such as well as not continuing to perform any execution that is unintentional.

```
169 func TestGroup_RoleBindings_ErrorBinding(t *testing.T) {
170   u := newFakeGroup(t)
171     defer u.ctrl.Finish()
172
173     options := metav1.ListOptions{}
174
175     u.fakeClient.EXPECT().Rbac().Times(1).Return(u.fakeRbac)
176
177     u.fakeRbac.EXPECT().RoleBindings(gomock.Any())
      ↪  .Times(1).Return(u.fakeRoleBindingsInterface)
178
179     u.fakeRoleBindingsInterface.EXPECT() .List(options).Times(1).Return(nil,
      ↪  errors.New("this is an error"))
180
181     if err := u.roleBindings(); err == nil {
182       t.Error("expected error, got=none")
183     }
184   }
```

Figure 4.2: Error Response from Role Binding API [*pkg/subject_access_delegation/origin_subject/group/group_test.go*]

4.2 End to End Testing

In order to ensure verification of components interacting with one another, end to end testing must be conducted. In the case of this application this must involve spinning up a full Kubernetes cluster, along with the application, and executing a variety of rule scenarios and monitoring the resulting state within the cluster.

The previous approach of using the built in testing framework is not best suited to run such tests since, even though possible, is not suited to some hard coded solution that would fork out several programs with various commands. Instead a new, bespoke solution is to be developed.

4.2.1 Framework

In order to facilitate a useable and extensible framework to facilitate testing of the application a bespoke solution is developed. Several requirements for such a system must be met in order to achieve the needs of this system, namely;

- Extensibility: the testing suite in which tests are written must be easily modifiable and new tests be written with little complex coding involved.
- Robustness: all tests run against the target application must be robust with regards to ensuring functional correctness and non-deterministic with regards to testing results.
- Readability: tests written within the framework must be readable to any developer in order to evaluate the cause or origin of any unexpected behaviour.

To achieve these requirements a framework has been written in which a developer is able to template a list of commands that are to be executed, whose Stdout and Stderr is captured, and tested against the defined test conditions. These conditions come in the form of some string which is be compared to some whole line of output or a particular word of such line. This approach to testing means that the states which are being tested against are indeed the end user exposed state in which meets the goals of end to end testing.

The testing blocks or template of commands that a developer would write such tests are written in a YAML file. The testing file it split into testing blocks in which one or more commands are listed, each with their various options and conditions attached. Upon execution, the end to end testing binary parses this file and begins to execute its contents. One such example of a test in which a Role Binding is expected to have been created is shown in Figure 4.3. Here a rule is to be created within the cluster, shown in Figure 4.4, of which a Group is to take on the bindings of the Service Account Origin Subject once a Pod of any name is created. As seen in the test block, this pod is then created and the corresponding expected Role Binding is checked for its existence. This is only a simple example, further conditions can be enforced such as checking each component of the Role Binding for more extensive state verification.

Using this framework to conduct end to end testing meets all three requirements set out that must be satisfied. This framework is extensible since writing tests does not require any compilation of code, only writing simple YAML fields that are consumed by the executable. The execution of the tests are robust in that running the commands themselves individually as a user are robust. If the resulting tests are non-deterministic then this would be a bug itself with the application, not with the testing suite. The testing blocks are readable in that they are written in YAML, as detailed earlier, is designed to be a highly readable file format. It is also clear with the order and collection of conditions that are attached to each command run against the application.

```yaml
- name: "Single Binding"
  commands:
  - program: "kubectl"
    arguments: "create -f docs/e2e_4.yaml"
    delay: 1
  - program: "kubectl"
    arguments: "create -f docs/nginx_pod.yaml"
    delay: 2
  - program: "kubectl"
    arguments: "get rolebindings"
    delay: 3
    split_string_conditions:
    - line: 3
      split: 0
      match: "test-group-test-sa-default-pod-logs-reader"
  - program: "kubectl"
    arguments: "delete -f docs/e2e_4.yaml"
    delay: 3
```

Figure 4.3: Single Binding Test [*docs/tests.yaml*]

4.3 Regression Testing

Software regression testing is the act of continuous testing of the target application during development cycles in order to ensure that no new bugs or execution becomes broken due to new additions to the code base. In order to facilitate regression testing, all tests are included into the build process of the application meaning that during every compilation, all tests are re-run against the application.

In order to ensure that these tests are up to date with regards to the application's total packages, tests are written after every completed package. This approach to continuous development and then testing of packages ensures application correctness throughout the development life cycle.

4.4 Notable Bugs

Continuous development and testing captured many bugs throughout the application development cycles. Although mostly minor, there were a few notable bugs that were less obvious to solve.

4.4.1 Use of Deep Copy

During execution of the controller, API objects are often read and written to, to be updated within the API server. This process is completed by downloading the objects from the API server to a local copy which can then be manipulated and then sent back to the API server to be updated. As such, it is very important for these objects to not only be regularly updated against the correct sate but their pointers are in fact pointing to the expected object. During development a bug arose in which the pointers to these objects appeared to not be attached to the correct object at all.

```yaml
apiVersion: authz.sad/v1alpha1
  kind: SubjectAccessDelegation
  metadata:
    name: test-group
    namespace: default
  spec:
    repeat: 1
    originSubject:
      kind: ServiceAccount
      name: test-sa
    destinationSubjects:
    - kind: Group
      name: test-group1
    triggers:
    - kind: AddPod
      value: "*"
    deletionTriggers:
    - kind: Time
      value: 5s
```

Figure 4.4: Subject Access Delegation - test 4 [*docs/e2e_4.yaml*]

Internal workings of the Kubernetes packages will often reuse pointers when listing several API objects of the same types meaning that use of these pointers used elsewhere will produce widely unpredictable and undesired results. This was indeed the case, with many components erroneously acting on the same object producing crashes in the application or widely incorrect results.

To eliminate this problem, the use of deep copies of these objects must be used. Instead of the shallow copying of the highest elements or pointers within its highest hierarchy structure elements, the entire underlying elements are copied. This removed the bugs of unexpected behaviour.

4.4.2 Trigger Channel Closing

With the addition of wild cards being accepted within trigger names, this enabled users with a great feature to expand what names a trigger could be satisfied by. This had some side effects however. As previously before this addition, triggers were only ever effected by one possible object of that defined name, even if shared with other triggers. This meant the trigger would be satisfied by one and only one object as per the unique name convention enforced by Kubernetes itself. The addition of the wild cards introduced a problem whereby triggers would be satisfied by objects with the same type, but with different names but still satisfy the wild card conditioned name. This meant that triggers could be made to trigger more than once, causing erroneous execution and finally a crash due to the closing of an already closed stop channel.

The name of objects can no longer be used to uniquely identify objects and as such a new method for ensuring triggers are not executed more than once; use of object Unique Identification (UID). Object UID's are strings attached to all unique API objects which the API server ensures that are unique from all others. The main controller can keep a local record of all API objects have been seen or deleted which can be checked within the trigger so that is it not erroneously triggered.

4.4.3 Resource Object Versioning

During the execution of a rule within the application the API server was responding with error messages detailing that it was unable to update the Subject Access Delegation custom resource object due to the current object version not matching the latest. The reason for these errors was that the API is the single source of truth for object versioning within the cluster meaning the it is the API's responsibility to hold the latest version of any resource object, denoted by a resource version string in all objects. If a component attempts to update an object with an outdated resource version number then this will be rejected.

In this case, several components of the application running concurrently, namely the controller, delegation handler and triggers, will all make atempts to update the same custom resource object in an effort to synchronise state with the API server on the sate of the current delegation. When two of more attempts are made at the same time then this will cause these errors of objects submitted with an outdated version. To resolve this issue, a mechanism was chosen to retry a failed update by firstly updating the local API object again if there was any update problem. This will cause multiple components to use a group back off and gradually complete the desired number of updates on the same object.

Chapter 5

Evaluation

In order to reach an acceptable level of success with the application against its main goals beyond functional correctness, a number of evaluation methodologies have been considered.

5.1 Stress Testing

The application is responsible for making critical security changes to the cluster, often in a time sensitive manor. With time being a sensitive component of the reliability of the application it must demonstrate that the changes it applies are done so in timely fashion as to not leave the cluster's permissions in an erroneous state, longer than is reasonable. The problem with testing this such property is that the number of variables that are responsible for determining how long a change to permissions in the cluster is very large. Just some examples of such variables include network latency, network traffic, API server load, scale of the cluster as well as the load and available resources that the application itself has access to. A combination of these variables, among others, concludes that determining the success of the application becomes difficult to obtain a quantitative result.

Nevertheless, a preliminary investigation has been conducted to gain some suggestion of how the application performs when scaling the cluster that has access to limited resources. In an effort to reduce noise in the results, the number of controlled variables have been maximised as well as all results shown will be the means taken from a set of 10 conducted tests.

The stress test has been conducted by creating a simple Go program that will query the API server with two GET requests on two resource objects every second. This program is built into a Docker container and deployed to be consequently scaled, increasing the stress on the API server. To determine the effect of this stress, a rule will be activated into the cluster that will take a Service Account Origin Subject that has two associated Role Bindings attached as well as a single User Destination Subject. Both activation and deletion triggers are a single Time Trigger that has a value of *now* meaning an immediate trigger. This rule gives a good baseline for the time it takes to execute a delegation without using any triggers.

Testing has been done using Minikube, a tool used to run a local Kubernetes cluster within a virtual machine [32]. The virtual machine has a number of restrictions in order achieve a level of stress that can produce results. The virtual machine has access to 2048MB of memory, 10GB of storage and a single CPU core of Intel's i7-7500U running at 2.70GHz.

The results of the tests are shown in Figure 5.1. As shown by the graph, the total cycle time of the delegation was consistent, with almost no change in execution time from 0 API requests to 80 per second. After this threshold, the API server begins an exponential time increase when the API server is no longer able to service the number of requests. At the highest number of requests per second the API server began to drop requests as the Time To Live (TTL) of the Transport Layer Security (TLS) packets began to expire causing the entire delegation to fail. The results here are showing only the delegations that eventually did succeed, sometimes requiring more

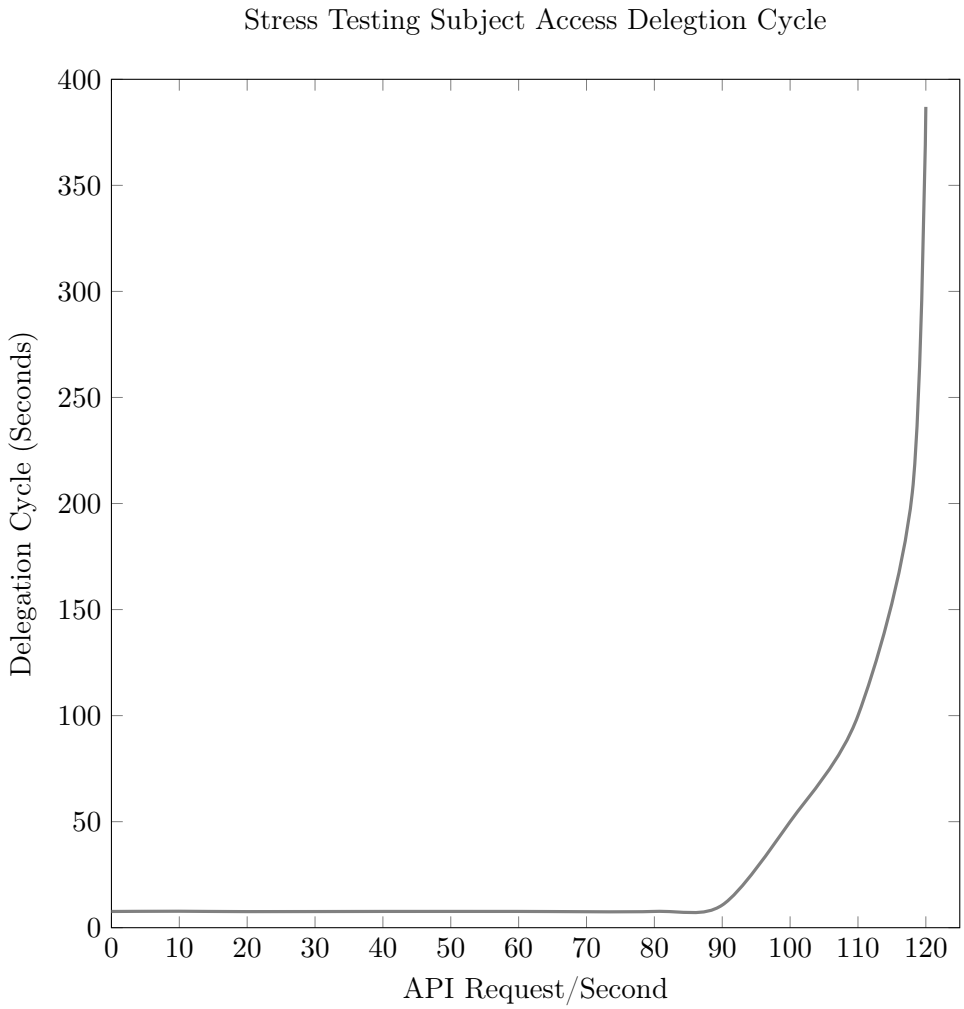

Figure 5.1: Subject Access Delegation Stress Test Results

than 8 minutes to complete.

What these results show is that the performance of the application are relatively consistent when the API server is put under reasonable stress by the network. It requires some threshold for the API server to begin to fail and fall behind on requests resulting in massive effects on the application's delegation time. This concludes that application response times are reliable and will begin to only start to be effected when the API server begins to fail completely, which would also begin to effect other functionality in the cluster also.

Although some preliminary testing has been done on the ability for the system to perform adequately, the application suffers from some potential attacks that could render the cluster vulnerable. One such attack, as demonstrated through the stress testing, is a distributed denial of service attack on the API server or entire cluster. This attack involves sending large amounts of traffic to the API sever or exposed end points of the cluster rendering internal traffic to become slow or halt completely. In such a scenario, the application may not be able to reach the API server, and as such, will be unable to change the permissions state within the cluster. This provides an attacker with a potential attack vector to sustain or prevent permissions change that can be used in conjunction with another attack. The application can not prevent this attack from taking place or, mitigate its effectiveness. Here, it is the cluster operators responsibility and obligation to ensure this attack is not feasible.

Although this attack possible, it requires the attacker have some prior knowledge of the rules which are used as well the cluster operator not protecting the cluster from such an attack. It also

suffers from resulting in non-deterministic behaviour that makes achieving a consistent attack difficult.

5.2 User Study

One possible methodology for determining the usability and effectiveness of the delegation rules and their successfulness in reducing administration requirements for users is to conduct a user study. This would involve surveying a large number of Kubernetes users from several organisations and their various use cases of the software. This has not been done due to being unfeasible in practice. In order to generate meaningful results of the study a large population size of users will need to be surveyed that, in order to produce quality data, must all use the application for some meaningful amount of time. Instead, a more passive approach has been used by requesting members of the community to give feedback on the application instead. This continuous feedback also gives the ability for updates to software in the future based on user experiences to inform how the application changes based on real world usage.

5.3 Community Correspondence

One other method in evaluation of the application beyond it's functional correctness and ability to action tasks based on the requirements, is the feedback given by correspondence with members of the community that actively work and use Kubernetes. Use of feedback at evaluation and during future work is a key tool to be used to assess the success of the application and drive focus of the software toward real use cases.

Feedback given by the community is shown in A.1 whereby Mo Khan, a software engineer at OpenShift Security, gave some thoughts and suggestions on the application. Although including some minor suggestions on some small changes to be made such as changing the API group of the Subject Access Delegation resource object, other larger suggestions have been made on main aspects of the technical design. These suggestions meant that not only could the initial release of the application improve in quality, with regards to standards of Kubernetes tools, but also provide objectives to strive toward in future development.

Through constant feedback, or even code contributions, from users and members of the industry, a continuous development strategy can be implemented whereby goals and objectives of the application can be dynamically shaped by the users themselves. This gives the application longevity and maximises the feature set, consolidating Kubernetes permission management into a single convenient tool.

Chapter 6

Conclusion

The goal of building this application was to tackle three key problems I had observed within the RBAC system of Kubernetes; administration, dynamic permissions and permission replication. By using this application, a cluster operator will have eliminated these problems.

6.1 Administration

A key issue that cluster operators suffer from is the administrative cost that derives from correctly managing access permissions of resources and users as their cluster increases in scale. With large amounts of user input required for correct configuration of access permissions the process can become labour intensive and by extension, prone to human error; unacceptable for security critical systems. With use of this application, configuration can become consolidated, greatly reducing user input and again, reducing human error.

One such example of how configuration can become consolidated and scaled automatically is shown in Figure 6.1. In this scenario, permissions have been correctly delegated for the so called *origin-service-account* Service Account whereby future Service Accounts will need these base permissions. A rule has been written that will activate immediately, triggering replicated permissions onto all Service Accounts with a name starting with *sa-*. Any Service Account added to the namespace will cause the rule to be re-delegated, updating permissions to include this new Service Account. With use of the wild card *, this rule will ensure future Service Accounts added will acquire these base permissions.

This example illustrates that with a single simple rule, permission configuration can be consolidated into a single declaration of permissions on the origin Service Account which will then be replicated onto future Service Accounts as the cluster scales, automatically, without the need for manual intervention of any RBAC configuration. Not only is replication automatically delegated for any new Service Account but if the permissions for the origin subject were to be updated in any way, this would also be reflected onto the destination subjects meaning adjustments to all Service Accounts need only be done for a single subject, again, greatly reducing need for manual configuration for multiple subjects in potentially many resource objects.

This example has shown that with use of a collection of small manageable rules, the application is able to reduce the amount of configuration written by a cluster operator making permission administration of a scaling cluster reliable and automated.

6.2 Dynamic Permissions

With the static permission declaration of Kubernetes' RBAC system, there is no method for cluster operators to automate the process of dynamically adjusting resources access permissions. A rule shown in Figure 6.2 is an example of one such situation as to where the application has alleviated this issue. In this example, a Tiller is wished to be deployed which will perform

```
apiVersion: authz.sad/v1alpha1
  kind: SubjectAccessDelegation
  metadata:
    name: ServiceAccount-auto-permissions
    namespace: default
  spec:
    repeat: 100
    originSubject:
      kind: ServiceAccount
      name: origin-service-account
    destinationSubjects:
    - kind: ServiceAccount
      name: sa-*
    triggers:
    - kind: Time
      value: now
    deletionTriggers:
    - kind: AddServiceAccount
      value: sa-*
```

Figure 6.1: Subject Access Delegation - Service Account Scaled Permissions

several deployments within the cluster. In order to execute these deployments it is required that the Tiller server acquire cluster admin permissions, however, once it has completed these deployments it will reside in an escalated permissions state.

The rule shown in Figure 6.2 is a solution to this scenario whereby as the Tiller is created, the application will grant the Tiller the needed permissions for its execution however, once it has completed its jobs as per the requested deployments have been added to the cluster, its admin cluster permissions are revoked automatically.

This example demonstrates how cluster operators are able to use this application as a tool to automate the process of permissions on resources according to states of the application.

6.3 Permission Replication

The Kubernetes RBAC system is such that bindings onto subjects can only be done with Roles or Cluster Roles meaning that there is no ability for subjects to replicate permissions of another. By virtue of the activation of any rule with the origin subject being one of a Service Account, User or Group, a delegation will execute permissions such that the destination subjects will be replicating permissions from the origin. This permission replication is also kept up to date with any permission changes of the origin subject. An example of such a scenario solved through this application is the rule shown in Figure 6.3. Here, both *Remote-Employee1* and *Remote-Employee2* will be using the same permissions as the *Employee* user at 6:00pm for 14 hours for 365 days.

The application therefore solves this problem and grants cluster operators with this feature.

```
apiVersion: authz.sad/v1alpha1
  kind: SubjectAccessDelegation
  metadata:
    name: tiller-dynamic-permission
    namespace: default
  spec:
    repeat: 1
    originSubject:
      kind: ClusterRole
      name: cluster-admin
    destinationSubjects:
    - kind: ServiceAccount
      name: tiller-service-account
    triggers:
    - kind: AddServiceAccount
      value: tiller-service-account
    deletionTriggers:
    - kind: AddDeployment
      value: nginx
    - kind: AddDeployment
      value: prometheus
      ...
```

Figure 6.2: Subject Access Delegation - Tiller Dynamic Permissions

6.4 Future Work

Although the application can be considered in a fully featured state, achieving the goals laid out, there are a number of future goals which should be considered for future work.

6.4.1 Role Binding Protection

All created Bindings by the delegation hander follow a naming convention to ensure there is a clear distinction of which bindings have been made by the handler. In this effort, it is clear from a user observing the bindings of the cluster which have been made by the handler and, as such, will not intentionally attempt to manipulate these bindings in any way. If for what ever reason these bindings do get manipulated or deleted, there is no mechanism in place for the application to attempt to correct this mistake. Future developments would need to ensure such a mechanism is implemented such that, for all bindings created by the application are to be watched in case of change. If a change does take place, the application will attempt to correct this mistake.

6.4.2 Sub Resources

As of the current implementation configuration of the Subject Access Delegation custom resource definition, the object has a single end point in which a user or software component has to access and manipulate the resource in the API server. This means that there is no separation of status data such as current Role Bindings created by the controller or, its current iteration count, and user generated data written in the form of a submitted rule. This lack of clear separation of user and status data needed by the application means that any user is able to manipulate status data

```
apiVersion: authz.sad/v1alpha1
kind: SubjectAccessDelegation
metadata:
  name: my-subject-access-delegation
  namespace: dev-namespace
spec:
  repeat: 365
  deletionTime: 14h
  originSubject:
    kind: User
    name: Employee
  destinationSubjects:
  - kind: User
    name: Remote-Employee1
  - kind: User
    name: Remote-Employee2
  triggers:
  - kind: Time
    value: 6:00pm
```

Figure 6.3: Subject Access Delegation - Remote Employee Shift

easily through intention or not. If this was to happen, an update to the locally stored resource delegation object within the application would cause a multitude of errors as any number of internal logic components would produce any undefined behaviour. With the API server being the single source of truth, this would never be self corrected by the application. Likewise, if the application were to update the resource object at the same as a user requests an update, a data race will occur and equally result in undefined behaviour.

To resolve this issue many primitive resource types within the Kubernetes API facilitate so called *subresources* of many resource types. These subresources contain data which is not meant for user manipulation but instead to be used by Kubernetes internal components for maintaining some sort of state in the API. These subresources not only provide nouns for finer access control but also provide separate end points for components to perform actions against the resource. This means updates will only effect the data within this subresource namespace, contained in the larger resource object. This provides a clear separation of user generated and controller component data within a single resource object within the API server.

Subresources would be ideal to be utilised within this application however this feature has not yet been included for use in custom resource definitions. In the release of Kubernetes version 1.10, subresources have been added as an alpha feature meaning in later releases they will become fully integrated and fully supported [33] [34]. In the future, this is a clear extension to be implemented in the application.

6.4.3 Cluster and Namespaced Subject Access Delegations

One criticism presented by Mo Khan was on the lack separation of delegation of cluster and namespaced Role Bindings shown in Figure A.1. As stated, a namespaced rule is able to implement cluster wide Role Bindings which can be confusing to a user using the application and so an improvement of this system would be to instead have two separate kinds of Subject Access Delegation resources, namespaced and cluster wide. This separation would make the actions

the rule operates in clearer and in line with the cluster and namespaced separation with the Kubernetes' RBAC.

6.4.4 Extending Trigger Requirement Logic Options

Currently the implementation of triggers in a rule means that satisfying the requirements calls for a logical conjunction of all triggers to be fired. Although this provides a wide range of options for a user to template various scenarios, the lack of logical operators may be a restriction of its functionality. A further set of operates could be implemented such as disjunction, negation or even order of satisfaction. This clearly provides a larger set of features available to the user however would involve a large amount of implementation to the complexity of the trigger logic. Determining whether this should be implemented will be decided by whether users in the future request this feature on the grounds of community feedback.

Bibliography

[1] Docker Inc. Docker. https://www.docker.com, 2018.

[2] Docker Inc. Docker Hub. https://hub.docker.com, 2018.

[3] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Google Inc., 2015. URL https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43438.pdf.

[4] Google LLC. Gmail. https://www.google.com/gmail, 2018.

[5] Google LLC. Google Docs. https://www.google.co.uk/docs/about/, 2018.

[6] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, John Wilkes, and Google Inc. Borg, Omega, and Kubernetes - Lessons learned from three container-management systems over a decade. *ACM Queue, Volume 14 Issue 1*, 2016. URL https://queue.acm.org/detail.cfm?id=2898444.

[7] Docker Inc. Docker Swarm. https://docs.docker.com/swarm/, 2018.

[8] The Apache Software Foundation. What is Mesos? A distributed systems kernel. http://mesos.apache.org/, 2018.

[9] GitHub Inc. Production-Grade Container Scheduling and Management http://kubernetes.io. https://github.com/kubernetes/kubernetes, 2018.

[10] Portworx. Portworx Annual Container Adoption Survey 2017. 2017. URL https://portworx.com/wp-content/uploads/2017/04/Portworx_Annual_Container_Adoption_Survey_2017_Report.pdf.

[11] OpenStack. OpenStack User Survey: A snapshot of OpenStack users' perspectives and deployments. 2017. URL https://www.openstack.org/assets/survey/April2017SurveyReport.pdf.

[12] 451 Research. 451 Research: Application containers will be a $2.7bn market by 2020. 2017. URL https://451research.com/images/Marketing/press_releases/Application-container-market-will-reach-2-7bn-in-2020_final_graphic.pdf.

[13] Nune Isabekyan. Introduction to Kubernetes Architecture. https://x-team.com/blog/introduction-kubernetes-architecture/, 2016.

[14] The Kubernetes Authors. What is Kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, 2018.

[15] The Kubernetes Authors. Concepts: Overview. https://kubernetes.io/docs/concepts/, 2018.

[16] The Kubernetes Authors. Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components, 2018.

[17] The Kubernetes Authors. Master-Node communication. https://kubernetes.io/docs/concepts/architecture/master-node-communication/, 2018.

[18] David F. Ferraiolo and D. Richard Kuhn. Role-Based Access Controls. *Proceedings of the 15th National Computer Security Conference (NCSC), pp. 554-563*, 1992. NIST - Computer Security Resource Center, 15th National Computer Security Conference (NCSC).

[19] The Kubernetes Authors. Using RBAC Authorization - API Overview: Role and ClusterRole. https://kubernetes.io/docs/admin/authorization/rbac/#rolebinding-and-clusterrolebinding, 2018.

[20] Clark C. Evans. YAML Ain't Markup Language (YAML) Version 1.2. `http://www.yaml.org/spec/1.2/spec.html`, 2009.

[21] The Kubernetes Authors. Using RBAC Authorization - API Overview: RoleBinding and ClusterRoleBinding. `https://kubernetes.io/docs/admin/authorization/rbac/#rolebinding-and-clusterrolebinding`, 2018.

[22] Cloud Native Computing Foundation. The package manager for Kubernetes: What is helm? `https://helm.sh/`, 2018.

[23] Tu Nguyen. A Deep Dive Into Kubernetes Controllers. `https://engineering.bitnami.com/articles/a-deep-dive-into-kubernetes-controllers.html`, 2017.

[24] Kubernetes Contributors. client-go. `https://github.com/kubernetes/client-go`, 2018.

[25] Google LLC. Effective Go. `https://golang.org/doc/effective_go.html`, 2018.

[26] Kubernetes Contributors. Generators for kube-like API types. `https://github.com/kubernetes/code-generator`, 2018.

[27] Network Time Foundation. NTP: The Network Time Protocol. `http://www.ntp.org/`, 2018.

[28] David L. Mills. A Brief History of NTP Time: Memorirs of an Internet Timekeeper. *ACM SIGCOMM Computer Communication Review, Volume 33, Issue 2, 9–21, 2003.*, 2003. ACM SIGCOMM Computer Communication Review. Computer Communication Review - CCR.

[29] Prometheus Authors. Prometheus - Monitoring system & time series database. `https://prometheus.io/`, 2018.

[30] Google LLC. Package testing. `https://golang.org/pkg/testing/`, 2018.

[31] Google LLC. pacage gomock. `https://godoc.org/github.com/golang/mock/gomock`, 2018.

[32] The Kubernetes Authors. Minikube. `https://github.com/kubernetes/minikube`, 2018.

[33] Nikhita Raghunath. apiextensions: add subresources for custom resources #55168. `https://github.com/kubernetes/kubernetes/pull/55168`, 2018.

[34] The Kubernetes Authors. CHANGELOG-1.10.md. `https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG-1.10.md`, 2018.

[35] Mo Khan. Feedback. `https://github.com/JoshVanL/k8s-subject-access-delegation/issues/10`, 2018.

Appendix A

Community Correspondence

Hi @JoshVanL, you requested some feedback on the sig-auth mailing list.
Here are some of my thoughts based purely on the README - I have not read any of the implementation.

This is a great topic for an undergraduate project! smile

subjectaccessdelegation appears to be a namespaced object but can cause cluster wide changes via cluster role bindings? This seems a bit strange to me. Perhaps this object would benefit from having both cluster and namespace scoped variants in the same way RBAC does?

For Origin Subject Cluster Role: Destination Subjects are simply bound to this role cluster wide: This seems like the wrong approach. The most common use case for RBAC is likely a namespaced role binding to the cluster role admin (or edit or view). This allows the reuse of admin without having to redefine it in every namespace via a role.

For Origin Subject SA, User, Group you state that role bindings are replicated. Does this replication read or write data cluster wide (would be a serious performance issue in larger clusters)? Also, it is unclear to me how you would reasonably protect the "source" role bindings (from deletion for example). Furthermore, if a binding gave the user admin level access to a namespace, they could simply create a new role binding to the admin cluster role which would presumably not be managed by your controller, and thus not be time bounded by it? In general it is very hard to control how a user shares their access rights.

apiVersion: authz.k8s.io/v1alpha1: please use your own API group and not k8s.io.

A diagram showing a more complicated example would likely be helpful (which indications of "origin" objects vs controller created objects). Changing an origin subject and showing the delta would also be helpful. I would particularly pay attention to showing how namespaced and cluster objects change (this is the part that is most unclear to me).

HTH.

Figure A.1: Mo Khan, Software Engineer, OpenShift Security [35]