TIGERA

Avoiding fire drills and midnight calls

Achieving Simplicity and Scale for Your Application Connectivity Infrastructure

Many of the common complaints that are voiced about today's application delivery environments are attributable to unnecessary complexity. Difficult, failure-prone infrastructure upgrades and application deployments and time-consuming, ineffective root cause analysis can all be attributed to the complexity that has arisen as a result of an IT organization's organic architectural evolutions. Interdependencies proliferate and progressively grow ever more difficult to understand. Staff struggle with a number of architectural challenges, including conflicting layers of control and orchestration.

Moving forward, the dynamic, large-scale nature of cloud native environments will present another form of complexity. As a result, eliminating the complexity associated with legacy environments is emerging an increasingly urgent imperative.

Luckily, cloud native platforms such as Kubernetes have taken a clean slate, expunging much of the complexity associated with legacy environments. It's now vital to follow those same design patterns for secure cloud native application connectivity—so you can avoid dragging legacy approaches and their associated complexity into these new environments. By taking this approach, your organization will be able to establish a robust, easy-to-operate connectivity environment—and eliminate the complaints that have dogged application delivery teams in the past.

TIGERA

Table of Contents

Introduction

Why have cloud native models been seeing such explosive growth? Quite simply, because they offer so many compelling advantages. One of the primary advantages of cloud native models is that they provide a way to break down complex, monolithic applications into a set of microservices that are simpler, more loosely coupled and easier to maintain. At the same time, cloud native platforms employ those same techniques for their own infrastructure functions. In short, cloud native platforms like Kubernetes are themselves cloud native applications.

To illustrate the advantages of the cloud native approach, consider a comparison between legacy virtual machine environments and Kubernetes. In a legacy virtual machine (VM) environment, a single monolithic controller is responsible not only for implementing, operating and removing VMs, but for health checking, resource allocation and more. That makes a controller very complex. It can also make that controller a resource contention point, and possibly even a single point of failure.

In comparison, these functions are widely distributed in Kubernetes. A scheduler determines which host should run a workload. Another process is responsible for automatically scaling and healing workload instances. An agent on the host handles much of the local assignment of networking and storage resources, either by directly interfacing with the host OS or via a lightweight plugin mechanism. Those plugins then actually perform the resource attachment and maintenance.

Each of these components has a small number of easily understood tasks. They are loosely coupled with one another, through a distributed, resilient key/value store. There is no command and control messaging. Instead, these loosely coupled components publish updates to the key/value store and others act on that information in an asynchronous

manner. It is also easier to understand how things work, as the functions are well bounded and straightforward.

On the surface, it might seem that moving from one component to many would be more complex. However, the complexity has been removed from the system by making each component much simpler.

In the legacy virtual machine world, the most problematic parts of the infrastructure are networking and storage. Almost all complaints about the networking environment stem from the complexity of the system that implements the network. To address that, you can take the same approach for application connectivity that has been used in the broader cloud native environment. The following sections examine how you can take this cloud native approach and apply it to the realm of secure application connectivity.

A Cloud Native Approach to Application Connectivity

As we discussed above, the very model of a cloud native application can lead to a simplification both in design and operation. So much so, in fact, that cloud native platforms use the same model for their internal design and for the applications they host. So, if the model works for an entire platform, why shouldn't it also work for the networking component?

If we look at many of the virtual networking environments in use today, they are very far removed from a cloud native model. They have centralized controllers that determine the necessary configuration for the entire network. Leaving aside other effects of this architecture (such as scaling characteristics), this makes for a very centralized, complex environment. This can have a significant impact on ongoing operations, including upgrades, troubleshooting, and diagnostics. This can also make it difficult to integrate with adjacent technologies and make enhancements. In many ways, these approaches are anathema to cloud

native environments. So, why would you move to the inherent simplicity of the cloud native model and then decide to run one of its most important components, the network, using a complex legacy model? The following section describes what a cloud native oriented networking model looks like, using Kubernetes as an example.

How to Simplify Kubernetes Networking

Leverage Existing Capabilities in the OS and Platform

One of the easiest ways to achieve simplicity is to actually leverage the components that already exist and are well understood. In a Kubernetes environment, there is a very effective and well-understood networking data plane: the Linux kernel. If you can design a networking model that can leverage the commonly used capabilities of the stock Linux kernel, you can reduce the number of components in your system. Further, you can be fairly certain that a Kubernetes operational team will already understand, and have tools to support, your networking platform.

Decompose to Simple Functions

By the nature of the legacy network, functions are necessarily complex. For example, simply connecting a workload to the network may require:

1. Creating a virtual interface.
2. Assigning the workload an IP address.
3. Configuring the underlying system to connect that virtual interface to the rest of the network.
4. Informing the rest of the network about the existence of the workload.
5. Determining the policies that apply to that workload.
6. Calculating the actual configuration to enforce those policies.

7. Installing those configurations.

Another example might be configuring a secure gRPC connection between endpoints. That might require the following steps:

1. Generating a public key/private key pair on each end-point.
2. Generating a Certificate Signing Request (CSR) based on that key-pair on each end-point.
3. Sending the CSR to a Certificate Authority (CA) after verifying its validity.
4. Installing the resulting certificates on each endpoint.
5. Calculating and installing the necessary configuration in the gRPC/HTTP datapath to ensure that the traffic will be encrypted and that the correct certificate will be used.
6. Establishing a TLS session with the remote end-point and validating that the remote end has the correct certificate as well.
7. Encrypting and transmitting the traffic.

These are both fairly complex functions. Now, it is possible to create a single task to perform either or both functions. However, that will be a complex task, one that would make it difficult to perform day ongoing operations such as configuration, diagnostics, repair, modification and so on. It would also continue to get more complex as new capabilities or features are added, for example, if load balancing is integrated to gRPC connectivity. At some point, the tasks become nearly impossible to maintain and manage.

Once again, the solution is to take a page from the cloud native design model, and decompose these functions into a number of simpler tasks. Each task should satisfy one, or at most a few, of the steps, and have well-defined criteria for success or failure. In our examples above, each step could become its own task. The combination of the tasks completes the function. Changing the behavior of a given task is straightforward, as

the tasks are simple and easily understood. Adding a new task to a function is even simpler, just requiring another basic task to be added to the function's run book.

Loosely Couple All Tasks

Once you have established multiple simple tasks, you need to link them together to perform the required functions.

In the legacy model, those functions would be tightly coupled, running in a fixed sequential order, with very specific dependencies. While this works in many environments, it is less likely to be successful in a large-scale environment. As the number of components (such as nodes, network connections, processes and so on) in a system increases, so does the likelihood that there will some fault or performance degradation in the system. Again, this is specifically part of the design consideration in a cloud native environment.

In a cloud native model, the assumption is that, where possible, tasks are independent, and continually run. In this model, the environment is compared to a desired state, and corrections are made until the environment achieves that state. While this seems, at the surface, to be more complex, it is actually easier to comprehend. Instead of evaluating why a complex function has failed, the test now is to look at each task independently and see if it has succeeded, failed or is still in process. The loose coupling of tasks minimizes the risk associated with interdependencies making it difficult to detect failures.

Functional Decomposition to Simplify Horizontal Environments

Inherently, a network is a horizontally distributed system. The simplest way to build and operate this kind of system is to functionally decompose the components that make up that system. Since cloud native environments simplify the operation of decomposed components, they are a natural fit for deploying and operating a modern network environment.

The Longer Metadata is to Exist, the Higher Level It Should Be

One form of operational complexity is introduced when there is a mismatch between the lifetime of an object and that of its metadata. For example, a given workload in a cloud native environment may only have a lifetime of seconds or minutes. Therefore, it makes no sense to use the specific identifiers (such as IP address) of those workloads in a longer-lived context, such as a policy statement. Even in a less dynamic environment, such as a legacy VM implementation. For example, this can lead to firewall "cruft", meaning there are firewall rules that no longer refer to the correct endpoints, or that are no longer relevant.

Distributed Key/Value Store

One of the challenges posed by a large distributed system is ensuring that the data necessary to operate the system available and highly scalable. The same requirements are necessary for a cloud native networking environment. Luckily, most cloud native systems already rely on a distributed, automatically repairing key/value (K/V) store for this

function. Again, it makes sense to leverage this same facility for the networking environment as well.

Do Things One Way and in One Place

Within a given system, if there are multiple methods of performing the same task, troubleshooting an issue can be much more difficult. For example, in many traditional software-defined networks (SDNs), segmentation can occur either by using network VLANs, VXLANs or firewalls. As result, if some communication is segmented when it shouldn't be, or not segmented when it should be, you have to figure out which mechanism was in use, if it is working and if it took the correct action. The more disparate mechanisms that you have in use in a system, the more complex that system is. Quite often, more mechanisms are in place than needed, resulting in more complexity than is needed.

An alternative is to perform a given task exactly one way, or using the fewest mechanisms possible. This means that there is only one mechanism to understand, extend, and troubleshoot.

For example, you could execute all micro-segmentation tasks by applying policy to the network interface closest to the endpoints involved in a given communication. Or you could apply all TLS session control via a sidecar within a Kubernetes pod - implementing the same mechanism for every workload, no matter where the workload is running.

Common Policy Model for all Aspects of Application Connectivity

In a cloud native environment like Kubernetes, the existence of multiple policy models can be another source of complexity. For example, let's say you have a set of application workloads that need to connect to a

specific set of database servers and these workloads should only be allowed to make read requests.

In addition, the read request restriction is defined via one policy, and access to the database servers is defined by another policy. If at some later point in time, the list of application workloads needs to be changed (for example, if you want to enable access for both production and test workloads), both policies must be updated. The problem is that someone may forget that two policies need to be updated—and the longer a system has been running, the more likely this omission may be. As a result, if only one policy is updated, the desired effect won't be achieved. Depending on the diagnostic and logging capabilities of the team, it may be easy to determine that the second policy needs to be changed—or it may lead to a long, frustrating diagnostics session.

Instead, if one policy was written that was rendered in both the network layer (governing database access) and the application connectivity layer (governing read-only access), only one policy change would be required—with no diagnostic effort needed. Further, when systems have one policy model, ongoing system operations are much easier.

Conclusion

The reality is that as any system grows, it invariably grows more complex. However, there is no need to make application connectivity infrastructure more complex than necessary. The cloud native model goes a long way in making large-scale operations as simple as possible. As an organization adopts the complexity-taming benefits of the cloud native model, it's counterproductive to keep using legacy approaches for networking and application connectivity. By maximizing the benefits of the cloud native model, organizations can simplify and scale their applications and their application connectivity infrastructure.

About Tigera

Tigera delivers solutions for secure application connectivity for the cloud native world. Tigera technology is used by the world's largest enterprises and public cloud providers to power connectivity for application development and deployment and to address the connectivity and security challenges that arise in at-scale production. Tigera Securemeets enterprise needs for zero trust network security, multi-cloud and legacy environment support, organizational control and compliance, and operational simplicity. Tigera Secure builds on leading open source projects Kubernetes, Calico, and Istio, which Tigera engineers help maintain and contribute to as active members of the cloud native community.

tigera.io

email: contact@tigera.io

phone: +1.415.612.9546

Tigera, Inc. 58 Maiden Lane, Fifth Floor, San Francisco CA 94018 USA