IRIO 2022 - distributed linters

Design doc

Michał Chojnowski
Adam Czajkowski
Michał Radwański

Introduction

The goal of this project is to provide an online linter service.
End users should be able to send a source code file to the provided endpoint and receive an analysis of the file in response.
The administrators of the service should be able to perform deployments and rolling updates of linters without disrupting the service.

Detailed project goals

Functional requirements

User-side

A publicly available HTTP service, which accepts source code files from end users and responds with an analysis of that code.

Admin-side

An internally available administration service, which allows the administrators to:

1. Register linter versions. This accepts metadata of a linter container image (URL, name and version) and deploys initial instances of the image to the cluster.
2. Unregister linter versions. This accepts the name and version of the linter to be removed and deletes its instances from the cluster.
   The system prevents removal of versions which still have a proportion of traffic assigned to them.
3. Configure traffic proportions between various available versions of the same linter.

The ability to (un)register versions and change traffic proportions between them is our implementation of the required “gradual update” feature. An update can be performed by registering the new version and progressively raising the traffic proportions directed to it. A rollback can be performed simply by restoring previous traffic proportions. This setup also naturally handles multiple concurrent rollouts.
(Manually setting proportions between versions is cumbersome and error-prone, so ideally there should exist helper tools to aid the administrator in the update process. However, due to time constraints we have settled for only the core functionality as listed above.)

Non-functional requirements

Reliability

The system should be able to handle failures (crashes and unavailability) of individual processes and nodes without user-visible disruptions (except degraded performance), as long as some replicas of every kind remain. A simultaneous failure of all linters of a particular version is allowed to make the fraction of requests owned by that version to fail, but this shouldn't disrupt other versions.
Administrator-side features need not be highly available, but in the event of a container/node failure they should take no more than a few minutes to recover automatically.
The service need not be resistant to large scale (e.g. multiple physical machines failing simultaneously) platform failures.

Scalability

The service should be able to grow smoothly to handle arbitrarily large load, provided that load growth is sufficiently slow. (I.e. the service doesn't have to gracefully handle sudden large spikes in load).
Ideally the system should scale as needed automatically.

Performance

In stable state the service should be able to handle at least a few thousand (small) requests per second per CPU core at <10ms service-time.

Observability

The service's performance should be easily inspected.

Design

Overview

The system consists of four software components:

1. Linters. They handle the business logic: that is, the analysis of source files provided by end users. Linters are fully stateless and oblivious to their environment. They expose their simple service on a port.
2. Load balancers. They have three functionalities:
    - They are the entry point to the system. The client connects directly to them and the load balancer is responsible for forwarding the requests to workers handling their language.
    - They balance the load over available workers to minimize latencies.

- They maintain requested traffic proportions over various versions of the same linter. This is key to the linter update process.

Load balancers are oblivious to the platform they are running on and are informed about current traffic proportions and currently available workers by an outside process.
A load balancer listens for user requests on a port and for configuration changes on another port. It keeps connections to the workers to relay user requests to them.

3. Machine manager. It's main functions are:
   - To monitor the set of available workers and keep the load balancers updated about it.
   - To receive and persist traffic proportions configured by the administrator and to forward them to load balancers.
   - To create and remove deployments of linter versions as requested by the administrator.

   The machine manager is the only component which understands the underlying platform and shares that knowledge with other components as they need it. It is also the only component with persistent state (which is the configuration set by the administrator).
   It listens for configuration requests on a port. It keeps connections to load balancers to update them whenever the configuration or the set of available workers change.
4. Administration frontend. A client tool for the human administrator for communicating with the machine manager. It connects to the admin request port of the machine manager and sends configuration requests to it.

In a typical deployment the system exposes two endpoints to the Internet: the user request ports of load balancers and the admin request port of the machine manager.

An important design idea was to keep everything other than the manager oblivious to the platform. This way all platform-specific knowledge is kept in one place, and most of the system does not need to be concerned with anything other than IP addresses and ports of relevant partners.
This should allow the system to be deployed manually on any setup (without a manager and a cloud platform – which of course incurs the loss of reliability features provided by the platform) or be easily ported to other platforms.

Since in this system the manager only propagates the knowledge stored in the platform, it would be possible to eliminate it and make load balancers track the changes in the platform themselves (with code in the main binary, or – if we want to keep the main binary of load balancers unconcerned with the platform – using a sidecar container). This would be functionally equivalent and would bring the benefit of a lower number of moving parts. However, having a central manager leaves more room for new features in the future (ones more active than just propagating events from the platform).

Reliability considerations

In a typical deployment the system relies heavily on the underlying platform. In our implementation, we lean on Kubernetes and its autohealing, replication and consensus features.

All linters and load balancers are replicated with the use of Kubernetes' "Deployments", which keep the number of replicas of a given container at a target number. Combined with liveness probes (which kill unresponsive containers after several seconds) configured on all containers used in the system, this ensures fault tolerance through replication and autohealing. It allows the system to survive failures of a small number of nodes and processes.

For keeping our information about available machines consistent with reality, we use Kubernetes's "watch" features. They provide a stream of updates about changes to selected resources: in our case the set of load balancers and linters. This happens in manager manager which propagates the information to load balancers. This way we offload most of the necessary bookkeeping.

Any configuration change is persisted before being applied. This way, even if the manager fails while applying a new configuration, it will be eventually restarted and reapply the latest configuration, bringing all machines back to a consistent configuration.
The persistent state of the machine manager (the traffic proportions set by the administrator) is stored as Kubernetes objects in etcd, which guarantees consistency and reliability as long as Kubernetes itself is replicated properly.

We have chosen to have a single instance of the machine manager at a time. This simplifies the system, but it means that:

1. Admin operations (adding and removing linters, reconfiguring load balancers) are unavailable while the manager is unavailable.
2. Information about joining and leaving linters is not propagated to load balancers while the manager is unavailable.
3. Joining load balancers don't receive configuration while the manager is unavailable.
4. The system is susceptible to split brain cases in events of network partition.

However, the first three problems arguably remain even with a replicated manager, since the time needed to restart a single instance of the manager is no different from the time needed to transfer leadership from a failed replica to a live replica. The fourth problem can be guarded against through a number of techniques which we didn't implement due to time constraints. (More about this in the machine manager section).

Component implementation notes

We have implemented all components in Golang, due to its fast development, simple deployment, good Kubernetes client, easy concurrency, built-in HTTP libraries and reasonable performance.
In a production system, we would choose to write the machine manager in Golang due to the above reasons, but the load balancers would be written in C++ or Rust due to superior runtime characteristics and tighter resource control, and linters would be likely written in a variety of languages.

Linters

Since linters are completely stateless (all requests to them are independent) and they don't connect to anything themselves, they are a trivial part of the system.
Linters expose a GRPC service conforming to a common API.
We have chosen GRPC here due to:

- Wide adoption.
- Auto-retrying and auto-reconnecting abilities, which significantly simplifies a reliable implementation of load balancers.

On the other hand, GRPC is relatively expensive compared to pure TCP and we aren't getting much out of GRPC's other features, since the message schemas for this service themselves are so simple and unlikely to change. Since the business logic in the system is relatively cheap, network communication takes the majority of resources and warrants optimization. In a performance-sensitive production system we would opt for pure TCP for this task.

Load balancers

Load balancers are responsible for passing user requests to the correct (with the matching programming language) set of workers, spreading the load between various versions of the same linter, and balancing the load between workers of the same version to minimize latency.

Given traffic proportions (weights) between versions, the algorithm used for selecting a linter version for an incoming request is a variant of the following:

1. Start by assigning a (floating point) counter with the value of 0 to each version.
2. Whenever a request for this programming language arrives, pick the version with the lowest counter and increase that counter by 1/(version weight).

Some additional precautions are taken to prevent problems with floating point inaccuracy during a long run of the algorithm.
In addition to providing the correct proportions over a large number of requests, this scheme also ensures good interleaving.

Load between various versions of the same linter is balanced by always picking the worker with the shortest queue (lowest number of outstanding requests) at the moment of choice (excluding workers who are known to be currently unresponsive). This basic algorithm gives good quality of balancing but is relatively expensive for large clusters. A more advanced solution could improve on this with a O(1) algorithm (e.g. "power of two choices").

The current implementation uses data structures shared between cores (counters for version balancing, queue length for worker balancing). However, this synchronization is avoidable: every core could be a separate load balancer and the results would be similar. Therefore the load balancer could benefit from a shared-nothing thread-per-core design.

On the other hand, we have performed some throughput tests of the system and found out that a large majority of the CPU work is spent on handling HTTP and GPRC, which makes synchronization performance concerns secondary.

In general, load balancers are the most performance-sensitive parts of the system and our implementation could be greatly improved. In hindsight, it would have been a good choice to use existing load balancer software, some of which (like Envoy) has all the features we need (like HTTP routing and weighting).

If a worker doesn't respond to a query within a few seconds, the LB retries the query using a different worker. This adds reliability in the event of workers being moved between nodes or destroyed due to updates.

A load balancer receives end user requests with HTTP and communicates with workers through GRPC.
The advantage of GRPC is that it has built-in heuristics for detecting unresponsive servers and reconnecting. This way a load balancer can easily skip currently unresponsive workers when choosing a worker for a request.
The advantage of HTTP for communication with the end users is wider adoption and platform support. In particular, Google Cloud's external load balancers have better support for HTTP than GRPC.

Machine manager

Due to statefulness, the machine manager is the most complex part of the system. Thankfully it can lean heavily on the capabilities of the underlying platform (in our case: Kubernetes) for most of its functionality.

The persistent state carried by the manager (version weights) is small enough to be stored in etcd via Kubernetes's APIs (ConfigMap).

In the current implementation, the system relies on there existing at most one active instance of the manager at a time. Under normal conditions, this will be ensured by Kubernetes, but in the (rare) event of network partition, the Kubernetes control plane can believe that the existing instance has failed and create a new instance of the manager. If during this time one instance receives an admin configuration request, a subsequent parallel upload of configurations from managers will cause load balancers to have different weights (old and new) configured. This issue can be fixed manually by killing one of managers (or waiting until it's done by Kubernetes, which continuously tries to keep the number of replicas at a target (1) number) and reissuing the most recent configuration.
This problem could be prevented altogether with many techniques, including (in the order of increasing effort):

- Using Kubernetes's "StatefulSet" feature which guarantees the maximum number of replicas.
- Monitoring the used ConfigMap for changes by the manager (this way all managers will stay consistent).
- Replicating the manager and keeping the history of configuration changes consistent using Raft.

Overall, the solution to this problem is rather easy and we haven't implemented it only due to time constraints.

The machine manager tracks changes in the set of available workers and load balancers using Kubernetes's/etcd's "list and watch" feature, wrapped around by the "controller-runtime" library. (In Kubernetes terms, the machine manager is a "controller": a program which monitors changes in the declarative model and modifies some part of the system to match the declarations. "controller-runtime" is the main library used for implementing controllers.)

The manager communicates with the load balancer and the admin frontend using GRPC. The reason for choosing GRPC here is the same as in the load balancer section.

Admin frontend

The admin frontend is a client-side application which translates human-readable commands to GRPC and sends them to the machine manager. Our implementation of it is a simple CLI.

Monitoring

All the components of our solutions can be easily observed as they expose metrics in Prometheus format. Full deployment of the system includes a Prometheus instance, which can be used to display information about the components the user is interested in.

Other potential improvements

We have mentioned that the system was not designed to handle large scale (e.g. zonal) failures. However, there are no obstacles (other than the (easily solvable) problem with potential inconsistencies in the manager mentioned before) to performing a multi-zone deployment for maximal resilience. It would however require more work with configuring the underlying platform and – for optimal performance – would require configuring the load balancers with locality information (to prioritize workers in their zone for load distribution).

The ergonomy of administering the system leaves a lot to be desired.

The system could benefit from a CI/CD pipeline.