MSc Computer Science

Trinity 2021

Perseus: A Latency and Fault Tolerance Library for Microservices Architecture in Golang

Candidate Number: 1051955

Abstract

This project is about the implementation of an open-source latency and fault tolerance library, Perseus, for microservices architecture in Golang. Perseus can make a system more resilient by providing fault tolerance methods, including timeout, fallback, circuit breaker. Several experiments are carried out to show that Perseus is easy to use and efficient.

This report introduces the backgound of microservices, analyses reasons for failures in microservices architecture and presents how Perseus is designed, implemented and tested.

Acknowledgements

I would like to express my gratitude to my supervisor for his guidance, valuable comments, and time invested in consultation throughout the work.

Contents

1 Introduction

In recent years, a shift from large monolithic applications to systems consisting of smaller independent units called microservices could have been observed in the area of enterprise application development[1]. Microservices have emerged from the world of domain-driven design, continuous delivery, on-demand virtualization, infrastructure automation, and the need for small autonomous teams owning the full lifecycle of their products[2]. In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery[3].

There are several issues brought by microservices architecture that need to be carefully addressed, one of which is how to make complex distributed systems more resilient. Microservices are usually spread across multiple machines and communicate over networks, which introduce challenges that developers need to deal with when building applications based on microservices. In a typical scenario, individual services depend on each other and a large number of different microservices are involved in an execution of a particular task. When a single microservice in this chain is unavailable or latent, all the microservices that depend on it may get stuck or fail. In the worst-case scenario, a failure of some unimportant component might eventually turn down the whole system. In order to prevent this, various techniques about how to deal with such failures have been developed. They have been described by stability patterns and implemented by fault tolerance libraries[1].

The biggest difference between microservices and monolithic applications is that service calls have changed from local calls within the same machine to

remote process calls between different machines, which derives two uncertain factors in dealing with failures of service calls.

One is the uncertainty of the service provider. The execution of the service calls is at the service provider side, and even if the service consumer itself is normal, the service provider may fail due to hardware reasons such as CPU, network I/O, disk, memory, network card, or due to program execution problems such as GC pause.

Another is the uncertainty of the network. The call happens between two machines, so information must be transmitted through networks. The complexity of the network is uncontrollable. Network packet loss, network delay, and instant jitter of networks that may occur at any time may cause failures of service calls.

Therefore, when transforming from a monolithic style into microservices architecture, special handling for failures of service calls is required. And thus the particular question I want to study is how to deal with failures of system calls caused by different reasons and provide a latency and fault tolerance library, Perseus, in Golang to handle these failures.

Currently, the most stable and widely used library for this purpose is Hystrix, which is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable[4]. However, it is designed only for Java.

Golang has been more and more popular in the field of high-performance distributed systems due to its efficiency in supporting massive parallelism. And microservices in Golang have been widely used in many technology companies. There are various techniques that companies are using to improve the fault tolerance and stability of their Golang software systems. However, fault tolerance functions are usually integrated into the microservices architecture as a ‘black box’, meaning that users are not allowed to customize the methods of handling

latency and faults according to their own business requirements. What's more, unlike Hystrix for Java, there is no such library for Golang that is considered as the most stable or authoritative. Instead, developers and organizations are continuing looking for suitable and efficient fault tolerance libraries for Golang.

The goal of this project is to provide an open-source latency and fault tolerance library, Perseus, in Golang to handle failures of service calls in microservices architecture. Perseus is easy to use and methods provided can be freely customised by users. The methods provided by the library include fallbacks, timeout, circuit breaking. Testing in Chapter 6 guarantees the correctness of the library. A guide to get started with Perseus is provided in Chapter 7 and experiments in Chapter 8 show that the library is helpful in real-world scenarios and is efficient in terms of the performance overhead produced by it.

1.1 Outline of the Report

This report will begin by providing a brief introduction to microservices architecture and fault tolerance mechanisms in Chapter 2 and Chapter 3. Chapter 4 will show the design of Perseus and Chapter 5 will present how Perseus is implemented according to the design. In Chapter 6, several tests are carried out to ensure the correctness of the library. Chapter 7 is a short guide as to how to get started with Perseus. After this, Perseus is applied on multiple servers simulating real-world applications and performance of Perseus is evaluated in Chapter 8 indicating that Perseus is easy to use and efficient. In the end, a conclusion is provided in Chapter 9, including the summary of the project, limitations of the methods in this project and possible evolution direction of fault tolerance mechanism of microservices architecture in the future.

2 Background

2.1 Microservices

The microservices architecture appeared lately as a new paradigm for programming applications by means of the composition of small services, each running its own processes and communicating via lightweight mechanisms[5]. It is an architectural style that separates large and complex applications into a collection of independently deployed services (processes) according to business capabilities. These services (processes) communicate with each other by lightweight, cross-language synchronous or asynchronous network calls.

The term “microservices” was first introduced at an architectural workshop in 2011 as a way to describe the participants’ common ideas in software architecture patterns[6], where services are built around business capabilities and deployed independently. While it has sprung up over the last few years and has been considered the most successful architectural style, there is no common consensus on microservices architecture. However, there are certain key characteristics of it:

(1) The entire application is split into sub-components that are independent of each other. A component is a unit of software that is independently replaceable and upgradeable[6].

(2) Microservices communicate with each other through lightweight network calls. This is the transformation of local method calls generated by JAR package dependencies in traditional monolithic applications into remote method calls via networks.

(3) Microservices can be deployed independently. If you have an application that consists of multiple libraries in a single process[11], a change to any single component results in having to redeploy the entire application. But if that application is decomposed into multiple services, you can expect many single service changes to only require that service to be redeployed. That's not an

absolute, as some changes will change service interfaces resulting in some coordination, but the aim of a good microservice architecture is to minimize these through cohesive service boundaries and evolution mechanisms in the service contracts[6].

(4) Microservices are organized around business capabilities. Such services take a broad-stack implementation of software for that business area, including user-interface, persistant storage, and any external collaborations[6].

Although these characteristics are believed to be common among them, not all microservice architectures have all the characteristics, but we do expect that most microservice architectures exhibit most characteristics[6]. Nowadays, many development teams have found the microservices style to be a superior approach to a monolithic architecture for that it enables the rapid, frequent and reliable delivery of large, complex applications.

There are many advantages of microservice architecture, including:

(1) Providing higher development speed;

(2) Taking full advantage of the modern software development ecosystem (clouds, containers, DevOps, serverless functions);

(3) Supporting horizontal scaling and fine-grained scaling.

2.2 Challenges

Like any architectural style, microservices architecture brings a lot of benefits as well as many challenges. In distributed systems, it is common for software systems to make remote calls to software running in different processes, probably on different machines across a network. One of the big differences between in-memory calls and remote calls is that remote calls can fail, or hang without a response until some timeout limit is reached. What is worse, if you have many callers on an unresponsive supplier, then you can run out of critical resources leading to cascading failures across multiple systems.

There are several burdens brought by it that development teams need to consider to make a sensible choice, including:

(1) The massive increase in RPC (Remote Procedure Call) and network communications;

(2) The security management of the entire system becoming more challenging;

(3) The design of the entire system becoming more complex and difficult;

(4) Introducing the complexity of distributed systems.

According to the disadvantages mentioned above, there are many open questions to study, such as:

(1) How to build a reliable service monitoring system?

(2) How to identify whether a service node is alive?

(3) How to apply a load balancing algorithm?

(4) What should we do when the service call fails?

Among these open issues, the particular topic this report will discuss is what are useful fault tolerance patterns that a system can use to maintain reliability even when there are some inevitable and uncontrollable errors in it.

With a large number of microservices and a complex infrastructure, it is much more likely that some part of the microservices architecture fails. If that part causes other parts to fail, the system will easily become unavailable[12]. So this project focus on how to make systems tolerate some unavailable microservices with certain technological measures.

3 Fault Tolerance

The biggest difference between microservices and monolithic applications is that service calls have changed from local calls within the same machine to remote process calls between different machines, which derives uncertain factors in dealing with failures of service calls. A consequence of using services as components is that applications need to be designed so that they can tolerate the failure of services. A fault-tolerant system means that the system can still operate normally even if failures of some parts of it occur. Any service call could fail due to unavailability of the supplier, instability of networks and other uncontrollable factors, and the client has to respond to this as gracefully as possible.

3.1 Failures in Microservices Architecture

3.1.1 Unstable Networks

Different components of a distributed system spread among different machines at different locations, and thus it is quite common for a component to make remote calls to components running in different processes, probably on different machines across a network. Therefore, networks are critical building blocks of distributed systems, and it is essential to deal with the fact that networks may be unreliable from time to time.

Before a packet reaches its destination, it must pass through many interconnected devices in the networks, including switches, routers, or bridges, each of which may fail due to some occasional errors. Besides, there are other uncontrollable factors in the networks such as network congestion. As a result, the packet may be lost.

Even if the packet is delivered successfully to the right destination, there are other issues that may affect the performance of the networks negatively, one of which is the network latency. It can heavily reduce the processing power of the software system if remote calls are treated the same way as local ones.

3.1.2 Failures of Service Providers

Apart from the connection issues of networks, failures of the service providers can also introduce stability risks. When clients are making remote calls, there might be errors on the server-side, such as runtime errors. In this case, even if the client and networks are normal, the single remote call will still fail in the end.

Of course, there are other unpredictable factors that may cause failures. For example, instantaneous peak traffic may put too much pressure on the server, making the service unavailable.

Cascading failures are an even more serious problem than a single remote call failure. This is especially true for systems with many interdependent components which communicate with each other using remote calls. If there are no resilience mechanisms applied to the system and one component fails, it is quite likely that it will cause the failure of the components that depend on it and further failure of their consumers. One such failure in quite an unimportant component may eventually turn down the whole system.

3.2 Design Patterns to Ensure Service Resilience

There are various approaches to improving the stability of an application system, including room wiring, network communications, hardware deployment, application architecture, data disaster tolerance. This project focuses on how to achieve a resilient distributed system through architectural designs of an application, including fallback, timeout and circuit breaking, making sure that the entire chain of microservices does not fail with the failure of a single component.

3.2.1 Flow Limiting

Flow limiting and fallback are essential abilities of a service-oriented system to avoid the cascading failure under the circumstance where there is instantaneous increase of clients' requests. If the system does not have the current limiting

capability, and access requests that far exceed the service processing capability are flooded in an instant, the back-end server will directly operate at full capacity. With a large amount of resource preemption and context switching, the platform processing capability will drop to an extremely low level. To this extent, this directly affects the response time of business requests, resulting in more business requests queuing, and ultimately causing the entire system to fail to respond and collapse.

3.2.2 Timeouts

This pattern indicates that clients would not wait for a service response for an indefinite amount of time — throw an exception instead of waiting too long. This will ensure that clients are not stuck in a state and continuing to consume application resources. Once the timeout period is met, the thread is freed up.

Well designed applications should always set a timeout for all remote calls. Without it a network failure or a system on the other side being down could hang the whole application. However, even if timeouts are used, they need to be set carefully in order to make a system work as expected. Otherwise, they may cause additional problems. If a remote call is waiting too long for a reply, it can slow down the whole system. On the other hand, if a connection timeouts too quickly, it may ignore a response that would otherwise be received [2].

3.2.3 Circuit Breakers

Outside of the software world, a circuit breaker is a component in an electric circuit which detects excess usage, fails first and opens the circuit. This is done in order to prevent other appliances from being damaged. Once the danger is over, the circuit breaker can be reset and the whole system is returned to a normal working state.

Circuit breakers in software systems are very similar to the ones in electric circuits. It wraps dangerous calls and makes sure they do not harm the system. Under normal circumstances, a circuit is closed and remote calls are executed as

usual. When a failure occurs, the circuit breaker takes note of it. If the failure rate exceeds a certain threshold, the circuit breaker trips and all further calls will fail immediately without reaching the remote destination. After some time a few requests are let through to the remote server to test if it is up again. If they succeed, the circuit is closed and all remote calls are executed as before. Otherwise, the circuit remains open and the same check is done again after some time[7].

Circuit breakers are an efficient way of preventing cascading failures. They work closely with timeouts and together with them provide good protection against unexpected failures in external components. They can even be used when turning down some part of the system for maintenance. In that case, all the circuit breakers in the dependent system can be tripped manually in order to fail fast while the component is being maintained [3].

Since circuit breakers need to collect some data in order to work properly, they are a valuable place for monitoring. They can provide different metrics, reveal details about the health of the system and be a good indicator of some hidden problems in the environment [10].

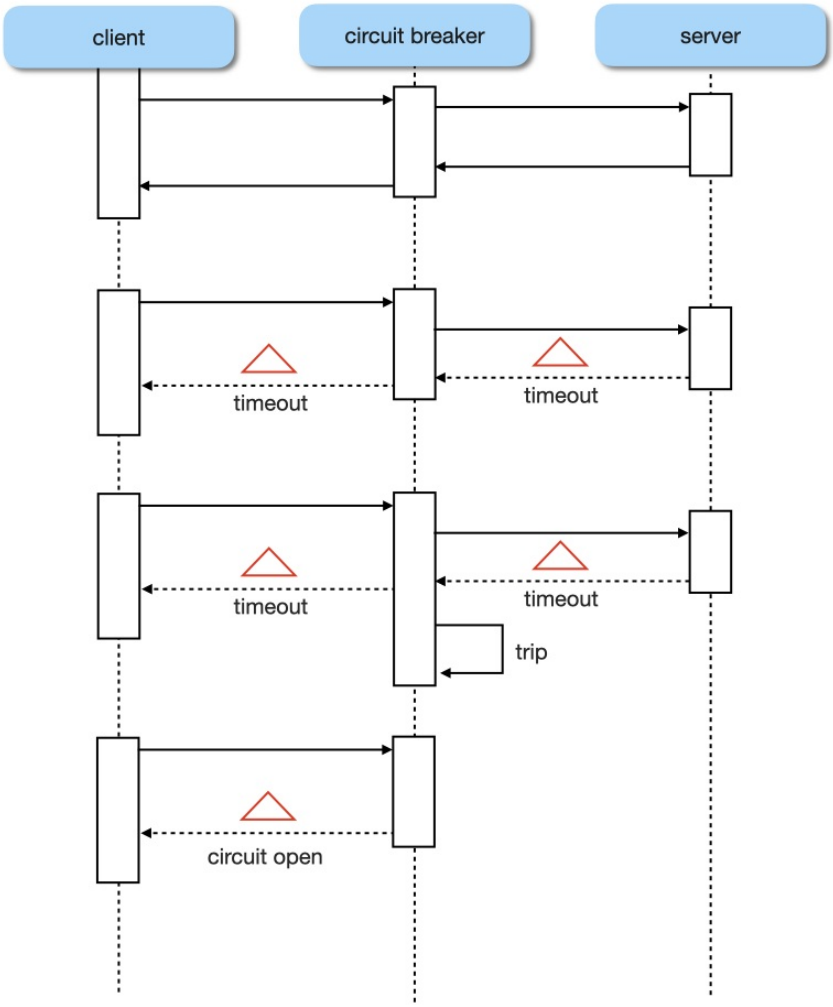

Figure 1. Sequence Diagram of Opening a Circuit Due to Multiple Timeouts

3.2.4 Retry Design Pattern

This pattern states that you can retry a connection automatically which has failed earlier due to an exception. This is very handy in case of temporary issues with one of your services. A lot of times a simple retry might fix the issue. The load balancer might point you to a different healthy server on the retry, and your call might be a success.

3.2.5 Bulkheads

In a ship, bulkheads are partitions that divide the inside of a ship into separate areas. They ensure that a single penetration of the hull does not necessarily sink

the ship. If the ship is in distress, hatches are closed and bulkheads prevent water from flooding other sections.

Similar techniques can be used in software systems. By partitioning a system, one can keep a failure in one component from affecting other components and eventually bringing down the whole system. Having a system split into several independent components ensures that the critical ones will keep running when a failure occurs in one of the less important components. For example, a system overwhelmed by flight status queries will still be able to provide booking or passenger check-in with reasonable latency [7].

One implementation of bulkheads is using separate connection pools for each connection. In this case, if one connection pool gets exhausted, the other connections will not be affected. This ensures that problems with one remote service do not affect components which do not communicate with it [3].

3.2.6 Fallback

A fallback function is a backup function executed whenever a command execution fails: when the command is short-circuited because the circuit is open, when the thread pool is at capacity or when the command has exceeded its timeout length.

Fallback is often written by users to provide a generic response, without any network dependency, from an in-memory cache or by means of other static logic. However, you can also use a network call in the fallback, in which case you should consider the fault tolerance of commands in the fallback similar to those outside.

4 Design of Perseus

4.1 Brief Introduction to Perseus

Perseus is a fault tolerance library designed for microservices architecture applications in Golang. It provides isolation between services, avoids cascading failures of the system and offers fallback options to maintain the resilience of the distributed system when failures are inevitable. It is a middleware that can be plugged into a microservices architecture or an application dependent on external systems to improve the stability of the system.

4.2 Goals of Perseus

Perseus is aimed to improve the resilience and stability of a system by providing:

(1) Isolated thread pools for different groups of command. This will provide flow limiting (see 3.2.1) and bulkheads (see 3.2.5) mechanism.

(2) Fallback (see 3.2.6) functions to be executed whenever a command execution fails.

(3) Timeout (see 3.2.2) mechanism which avoids waiting for a response for too long.

(4) Circuit breakers (see 3.2.3) which prevents cascading failures by rejecting further requests when the circuit is unhealthy.

4.3 Design Patterns of Perseus

4.3.1 Flow Chart

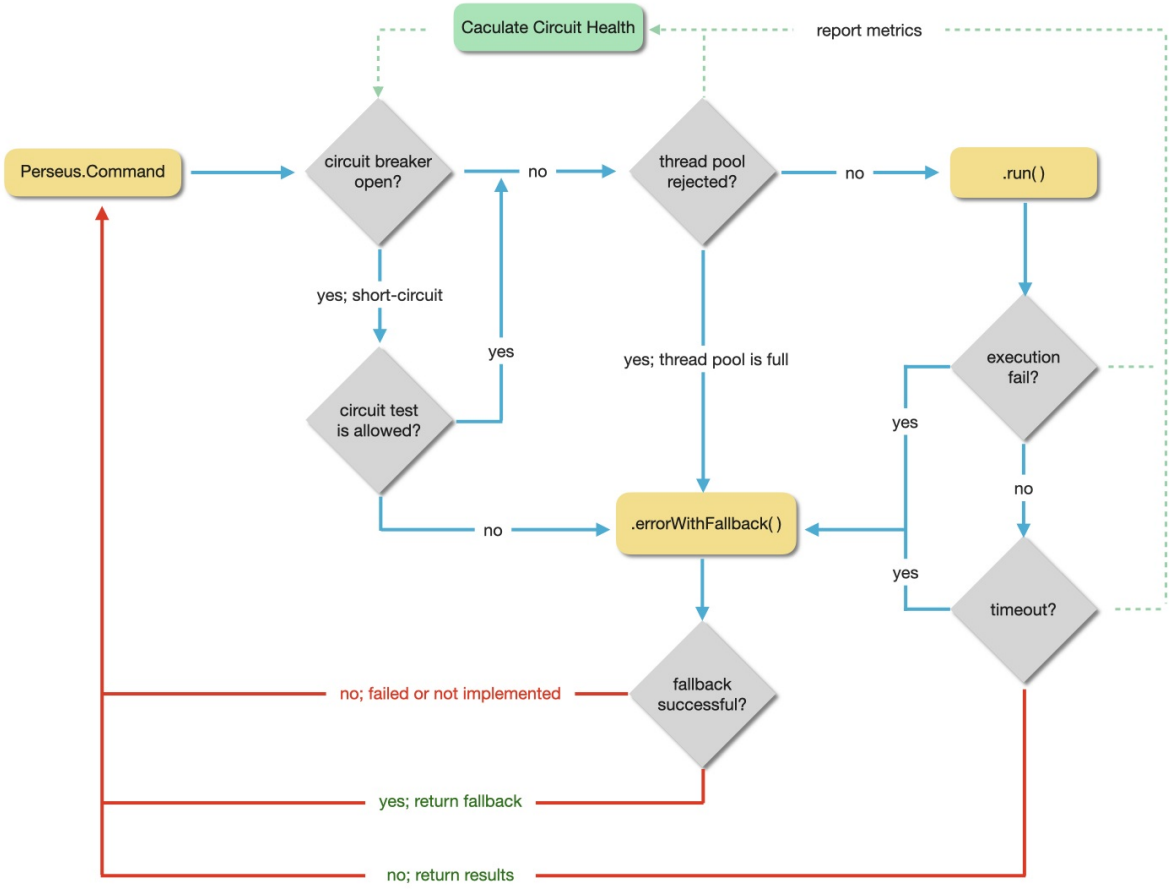

Figure 2. Flowchart of Perseus

The first step is to construct a Command object to represent the request sent to the external dependency. Then Perseus checks if the circuit breaker is open or not. If the circuit breaker is open and circuit test is not allowed, then Perseus will not execute the command but will execute .errorWithFallback() to execute the fallback function. If the circuit breaker is closed or the circuit test is allowed, then the command will proceed to check if the thread pool that is associated with the command is full. If the pool is full, then .errorWithFallback() will be triggered like before with the command not being executed. But if the thread pool is not full, the command will be executed by triggering .run() to invoke the request to the dependency.

If the .run() method fails or exceeds the command's timeout value, the thread will trigger .errorWithFallback() instead, and it discards the eventual return value

of .run() (note that the method does not cancel/interrupt). If the command did not throw any exceptions and it returned a response, Perseus returns this response after it performs some logging and metrics reporting.

Perseus reports successes, failures, rejections, and timeouts to the circuit breaker, which maintains a rolling set of counters that calculate statistics. It uses these stats to determine when the circuit should “trip,” at which point it short-circuits any subsequent requests until a recovery period elapses, upon which it closes the circuit again after first checking certain health checks.

4.3.2 Circuit Breaker

If there are failures in your microservices ecosystem, then you need to fail fast by opening the circuit. This ensures that no additional calls are made to the failing service, once the circuit breaker is open. So we return an exception immediately. This pattern also monitors the system for failures and once things are back to normal, the circuit is closed to allow normal functionality.

The following diagram shows how a Command interacts with a circuit breaker.

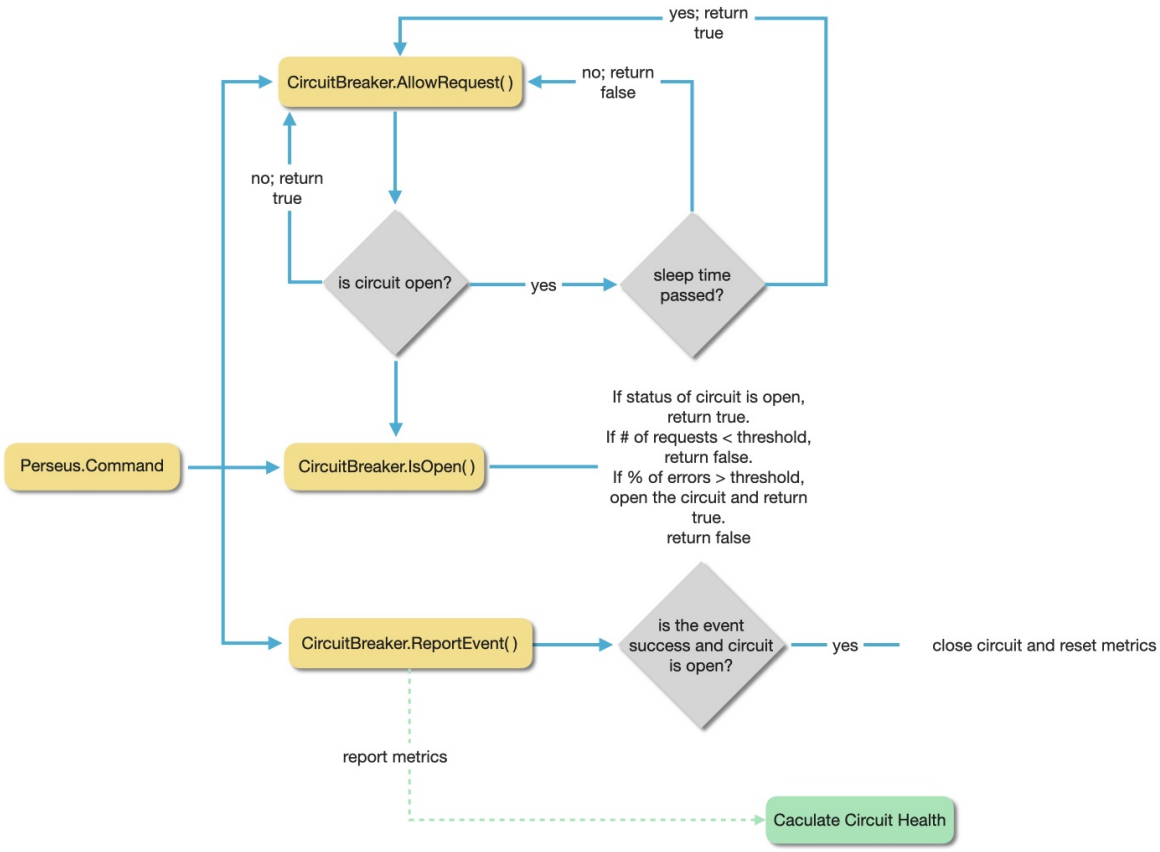

Figure 3. How a Command Interacts With a Circuit Breaker

The precise way that the circuit opening and closing occur is as follows:

(1) Assuming the volume across a circuit meets a certain threshold,

(2) And assuming that the error percentage exceeds the threshold error percentage,

(3) Then the circuit breaker transitions from ‘closed’ to ‘open’.

(4) While it is open, it short-circuits all requests made against that circuit-breaker.

(5) After some amount of time, the next single request is let through (this is the ‘half-open’ state). If the request fails, the circuit breaker returns to the ‘open’ state for the duration of the sleep window. If the request succeeds, the circuit breaker transits to ‘closed’ and the logic in (1) takes over again.

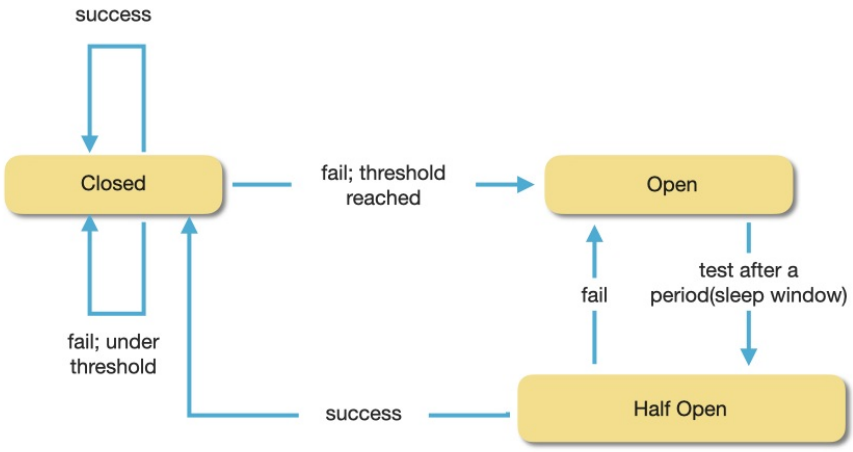

Figure 4. States of a Circuit

4.3.3 Thread Pools

Perseus uses separate, per-dependency thread pools as a way of constraining any given dependency. Apart from controlling the number of threads in a certain thread pool according to business capabilities, another benefit of using different thread pools for different business types is that errors only have impacts within a thread pool instead of making the whole system fail. Plus, latency and faults on the underlying executions will saturate the available threads only in that pool.

A thread pool and a circuit breaker are in a one-to-one relationship. In other words, a single thread pool is connected to a single circuit breaker and vice versa. More precisely, a circuit breaker contains a thread pool. They together support concurrency controls and health checks.

5 Implementation

This chapter shows how Perseus is implemented according to the designs mentioned in Chapter 4 with implementation details explained by texts, diagrams and pseudocode. The programming language is Golang, which is also the type of the language that Perseus supports, as mentioned before. The source code of this project is open-source with Apache License and is maintained on Github (https://github.com/xiaoyisha/Perseus).

The tree structure of my project is as follows:

```
├── LICENSE
├── README.md
├── circuit
│   ├── circuit.go
│   ├── circuit_test.go
│   ├── pool.go
│   └── pool_test.go
├── config
│   ├── config.go
│   └── config_test.go
├── experiments
│   ├── server0.go
│   ├── server1.go
│   ├── server2.go
│   ├── server3.go
│   ├── server4.go
│   ├── server5.go
│   └── server6.go
├── metrics
│   ├── default_metric_collector.go
│   ├── metric_collector.go
│   ├── metric_test.go
│   └── metrics.go
├── rolling
│   ├── number.go
│   └── timing.go
├── service.go
└── service_test.go
```

There are four modules in this project that implement fault tolerance

mechanism: circuit module, configure module, metric module and rolling module. The interface provided to users to call to use the library is written in service.go.

service.go implements Command type and provides interfaces that users can call to use the library by wrapping users' commands into Perseus.Go() or Perseus.Do(), which internally call the methods in the four modules above to provide fault tolerance mechanism of the user's command.

5.1 Circuit Module

The circuit module implements CircuitBreaker and ExecutorPool, which are key components of Perseus. CircuitBreaker tracks whether requests should be attempted or rejected for each ExecutorPool according to the health of the circuit.

CircuitBreaker is implemented as a struct in Golang:

```
type CircuitBreaker struct {
    Name                   string
    open                   bool
    forceOpen              bool
    mutex                  *sync.RWMutex
    openedOrLastTestedTime int64
    ExecutorPool           *ExecutorPool
    Metrics                *metrics.MetricExchange
}
```

(1) **Name**: unique identification of a circuit breaker.

(2) **open**: true if the circuit breaker is opened by health checks, false otherwise.

(3) **forceOpen**: true if the circuit breaker is manually forced to open, false otherwise.

(4) **mutex**: a mutual exclusion lock.

(5) **openedOrLastTestedTime**: most recent opened or tested Unix time.

(6) **ExecutorPool**: corresponding executor pool.

(7) **Metrics**: containing metric collectors and updates channel for collecting

event types. More details about metrics will be discussed in 5.3.

Note that in Golang, if names of variables or functions are capitalized, the variables or functions can be visited by other packages, otherwise, they are only accessible in the current package.

The init() function of the package is as follows, which is a function that takes no argument and returns nothing. This function executes after the package is imported and initialises a map maintaining the relationship between circuit breakers and their names, and a lock is also initialised to provide mutual exclusion between threads running in the program.

```
func init() {
    circuitBreakersMutex = &sync.RWMutex{}
    circuitBreakers = make(map[string]*CircuitBreaker)
}
```

The member functions of CircuitBreaker struct are:

(1) **SwitchForceOpen(forceOpen bool) error** sets the member variable .forceOpen to be equal to the parameter forceOpen.

(2) **IsOpen() bool** returns true if circuit is open and false otherwise. An 'open' circuit means the command should be rejected. If the circuit is not healthy, open it.

(3) **AllowRequest() bool** is called before any Command execution. When the circuit is open, this call will occasionally return true to measure whether the external service has recovered.

(4) **allowSingleTest() bool** returns true if sleep window has passed since the circuit was last opened or tested and false otherwise.

(5) **SetOpen()** sets the member variable .open to be true.

(6) **setClose()** sets the member variable .open to be false and reset the metrics of the circuit.

(7) **ReportEvent(eventTypes []string, start time.Time, runDuration**

time.Duration) error records command metrics for tracking recent error rates. If the circuit is currently open and the event type is 'success', close the circuit.

Since a program usually contains many threads, it is possible that a circuit breaker can be accessed by multiple threads at the same time. Therefore, it is crucial to lock a circuit breaker to provide mutual exclusion between threads when reading and writing values of member variables of it. For example, setClose() is written as follows:

```
func (circuitBreaker *CircuitBreaker) setClose() {
        circuitBreaker.mutex.Lock()
        defer circuitBreaker.mutex.Unlock()
        ......
        circuitBreaker.open = false
        ......
}
```

In Go language, defer statements delay the execution of the function or method or an anonymous method until the surrounding functions returns. In other words, the statement circuitBreaker.mutex.Unlock() will not be executed until setClose() returns, which means that the statement

```
circuitBreaker.open = false
```

is protected by the lock and will not be executed by more than one thread.

The ExecutorPool is another struct as follows:

```
type ExecutorPool struct {
        Name      string
        MaxReq    int
        Tickets   chan *struct{}
        Metrics   *poolMetrics
}
```

(1) **Name**: unique identification of an executor pool, and is the same as the name of the corresponding circuit breaker.

(2) **MaxReq**: max number of request that is allowed in the executor pool.

(3) **Tickets**: a channel of tickets, from which any thread that wants to be executed in the executor pool needs to acquire.

(4) **Metrics**: metrics of the executor pool, including the max number of active requests and number of executed requests.

Channels in Golang are a medium that the goroutines use in order to communicate effectively. Here, I use Tickets channel to initially hold MaxReq number of tickets, and control the total number of threads running in the executor pool by letting threads acquire tickets and return tickets through the channel.

The member functions of ExecutorPool are:

(1) **ReturnTicket()** returns a ticket to the executor pool.

(2) **ActiveCount()** returns the number of threads that are active in the executor pool.

As is shown in its struct, CircuitBreaker contains a reference to its executor pool, meaning that the two components are inseparable. Actually, they are in a one-to-one relationship, meaning that a circuit breaker only has one executor pool, and an executor pool only corresponds to one circuit breaker. They together provide infrastructures for implementing fault tolerance patterns.

5.2 Configure Module

The configure module provides methods of changing values of timeout limit, volume threshold, error percentage and other essential parameters of fault tolerance to customise the circuit breaker according to users' requirements. The Config struct is as follows:

```
type Config struct {
    Timeout                time.Duration
    MaxConcurrentRequests  int
    SleepWindow            time.Duration
    RequestVolumeThreshold uint64
    ErrorPercentThreshold  int
}
```

(1) **Timeout** is how long to wait for a command to complete, in milliseconds.

(2) **MaxConcurrentRequests** is how many commands can run at the same time in an executor pool.

(3) **SleepWindow** is the minimum number of requests needed before a circuit can be tripped due to health.

(4) **RequestVolumeThreshold** is how long, in milliseconds, to wait after a circuit opens before testing for recovery.

(5) **ErrorPercentThreshold** causes circuits to open once the rolling measure of errors exceeds this percentage of requests.

The parameters in Config struct control a circuit breaker's behaviours. These parameters are configurable by users and control when the corresponding circuit should be opened, closed, or tested, and how big the corresponding executor pool is. All of these parameters have default values providing fault tolerance suitable for common use cases:

```
DefaultTimeout = 1000
DefaultMaxConcurrent = 10
DefaultVolumeThreshold = 20
DefaultSleepWindow = 5000
DefaultErrorPercentThreshold = 50
```

The init() function in config package is very similar to that in circuit package, which initialises a map to maintain relationships between a circuit's name and its configuration. Also, a lock providing mutual exclusion to avoid multiple threads accessing the map at the same time is initialised.

```
func init() {
        circuitConfig = make(map[string]*Config)
        configMutex = &sync.RWMutex{}
}
```

Whenever a thread wants to get or change a certain configuration of a circuit breaker, the configMutex lock should be acquired before the thread accessing the circuitConfig map and released afterwards. For example, when configuring a circuit breaker, the configMutex lock should be acquired:

```
func ConfigureCommand(name string, config CommandConfig) {
        configMutex.Lock()
        defer configMutex.Unlock()

        timeout := DefaultTimeout
        if config.Timeout != 0 {
                timeout = config.Timeout
        }
        ......
        circuitConfig[name] = &Config{
                Timeout: time.Duration(timeout) * time.Millisecond,
                ......
        }
}
```

5.3 Metric Module

The metric module implements MetricCollector to collect the events of circuit breakers, including success, failure, timeout, rejected, short-circuit and so on.

MetricCollector represents the contract that all collectors must fulfil to gather circuit statistics.

```
type MetricCollector interface {
        // Update accepts a set of metrics from a command execution
for remote instrumentation
        Update(MetricResult)
        // Reset resets the internal counters and timers.
        Reset()
}
```

The MetricResult is a struct containing metrics that need to be considered when checking a circuit's health:

```
type MetricResult struct {
        Attempts                    float64
        Errors                      float64
        Successes                   float64
        Failures                    float64
        Rejects                     float64
        ShortCircuits               float64
        Timeouts                    float64
        FallbackSuccesses           float64
        FallbackFailures            float64
        ContextCanceled             float64
        ContextDeadlineExceeded float64
        TotalDuration               time.Duration
        RunDuration                 time.Duration
        ConcurrencyInUse            float64
}
```

The metric ‘Errors’ is the numerator and the metric ‘Attempts’ is the denominator when computing the error percentage of a circuit. The value of ‘Errors’ is equal to the sum of values of ‘Failures’, ‘Rejects’, ‘ShortCircuits’ and ‘Timeouts’. The metric ‘Failures’ is the rest of all the possible event types except Success, Rejects, ShortCircuits, Timeouts, FallbackSuccesses, FallbackFailures, ContextCanceled and ContextDeadlineExceeded.

DefaultMetricCollector is an implementation of the interface MetricCollector, which is the canonical source of information about the circuit used for circuit health checks. It contains all the information in the MetricResult. However, the type of each metric is an array of numbers, which will be discussed in 5.4.

The type that monitors when there are events that need to be reported to the metric collector is MetricExchange, which is contained in the CircuitBreaker struct.

```
type MetricExchange struct {
        Name       string
        Updates chan *CommandExecution
        Mutex      *sync.RWMutex
        metricCollectors []MetricCollector
}
```

‘Updates’ is a channel through which event types, such as success, timeouts, and short circuits, can be reported to metric collectors.

```
func NewMetricExchange(name string) *MetricExchange {
        m := &MetricExchange{}
        ......
        go m.Monitor()

        return m
}
```

When a MetricExchange instance is constructed, a goroutine executing the member function .Monitor() is launched, which monitors the updates channel, and change information stored in metric collectors. The pseudocode of .Monitor() is as follows:

Keep monitoring the Updates channel:
if there is a message in the Updates channel:
Lock the MetricExchange instance;
Update metrics;
Unlock the MetricExchange instance.

Note that multiple metric collectors are allowed in case users want to define various metric collectors that perform differently or send to different visualising dashboards.

5.4 Rolling Module

The rolling module implements types of Number and Timing, which helps calculate a variety of statistics obtained from the metric collector. The statistics will be taken into account when changing the status (open/closed/half-open) of circuit breakers.

The type Number tracks a bounded number of time buckets. In Perseus, the buckets are one second long and only the last ten seconds are kept. Some information stored in DefaultMetricCollector is of the type Number, such as successes, failures and rejects. Number contains a map maintaining the

relationship between a Unix time and a value. When Number is updated, the old time bucket (ten seconds earlier than current time) will be removed, and the new bucket (current Unix time) will be added to the map.

Likewise, the type Timing maintains time Durations for each time bucket. The Durations are kept in an array to allow for a variety of statistics to be calculated from the source data. The values totalDuration and runDuration stored in DefaultMetricCollector are of the type Timing.

5.5 Service Interface

This package provides service interfaces for users to wrap their commands to use Perseus, including Perseus.Go() and Perseus.Do(), which are asynchronous and synchronous requests, respectively.

There are two goroutines launched in .Go(). The first goroutine executes the main logic of fault tolerance while the second goroutine can be seen as a timer that provides timeout control. The pseudocode of the first goroutine is:

```
If circuit is open:
    Fallback with short-circuit error;
    Return;
Try to acquire a ticket;
If failed to acquire a ticket:
    Fallback with max concurrency error;
    Return;
Run the command;
If an error is returned:
    Fallback with run error;
Report event types.
```

As for .Do(), the logic is very similar to .Go() except it runs the command in a synchronous manner, blocking until either the command succeeds or an error is returned, including Perseus circuit errors.

6 Unit Testing

This module aims to check the correctness of the library by unit testing. Unit testing is essential when creating software. It tests each component one after another but does not test it entirely[8].

6.1 Circuit Module

The unit testing methods in circuit module are:

(1) TestGetCircuit

This method checks that if a circuit breaker doesn't exist, a new circuit will be created and returned; otherwise, the existing circuit breaker will be returned.

(2) TestMultithreadedGetCircuit

This method launches 100 identical threads executing GetCircuitBreaker() at the same time, and ensures that GetCircuitBreaker() is thread-safe by assuring only one thread create the circuit breaker.

(3) TestReportEventMultiThreaded

This method checks that multiple threads can successfully report events at the same type. In other words, this method ensures that ReportEvent() is thread-safe.

(4) TestReturnTicket

This method tests if a thread can return a ticket successfully into the thread pool.

(5) TestActiveCount

This method checks if the number of threads that have gained tickets is the same as the value of active count of the thread pool.

6.2 Configure Module

(1) TestConfigureConcurrency

(2) TestConfigureTimeout

(3) TestConfigureRVT

(4) TestSleepWindowDefault

(5) TestGetCircuitSettings

The logic of unit testing methods in configure module is really simple: change settings for a circuit and then check the member variable of the circuit is successfully updated.

6.3 Metric Module

(1) TestErrorPercent

The testing method in this module checks if metrics of a circuit are unhealthy if the error percentage exceeds the ErrorPercentThreshold.

6.4 Service Interface

The methods in this module tests whether .Go() and .Do() have expected behaviours.

(1) TestSuccess

What this method do is to first pass a successful command to Perseus, which sends a value to a channel, and then read from the channel to check if there is the expected value in it. Plus, no errors should be returned and metrics should be recorded(‘success’ equals 1).

(2) TestFallback

This method pass a command which fails and whose fallback sends to a channel to .Go(), and check there is the expected value in the channel, no errors is returned and metrics are recorded(‘success’ equals 0, ‘failures’ equals 1, ‘fallback_success’ equals 1).

(3) TestTimeout

This method pass a command which times out and whose fallback sends to a channel to .Go(), and check there is the expected value in the channel, no errors is returned.

(4) TestTimeoutEmptyFallback

This method pass a command which times out and has no fallback to .Go(), and check a timeout error is put into the error channel and metrics are recorded.

(5) TestMaxConcurrent

This method tests that if a circuit has max concurrency set to 2, and 3 threads try to execute at the same time, then one will return a ‘max concurrency’ error and two will be successfully executed.

(6) TestForceOpenCircuit

This method tests that when a command with a forced open circuit is run, a ‘circuit open’ error is put into the error channel and metrics are recorded.

7 How to Use Perseus

7.1 Import Perseus

Firstly, you need to import Perseus into your program:

```
import “github.com/xiaoyisha/Perseus”
```

7.2 Pass Your Function to Perseus’s Command

Then you should wrap your own function in Perseus command. In other words, you can write your business logic in a function which may depend on other microservices, and then pass the function to Perseus.Go(), or Perseus.Do() if you want synchronous behaviours.

Note that if the dependencies are healthy, your function will be the only part that executes.

```
Perseus.Go("my_command", func() error {
        // your business logic
        // which may depend on remote services
        return nil
}, nil)
```

7.3 Defining a Fallback Function

You can choose to define a fallback function to execute during a service outage. The fallback function can only use local resources to return a reasonable response to users or depend on other remote servers as well, in which case it is recommended that you use another Perseus command in the fallback function. You can define different fallback logic based on different errors.

```
Perseus.Go("my_command", func() error {
        // your business logic
        // which may depend on remote services
        return nil
}, func(err error) error {
if err == Perseus.ErrMaxConcurrency
        // handle the error based on its type
        return nil
})
```

Note that the fallback function is supposed to be triggered when the first function containing your business logic returns an error, either runtime error or error caused by health checks of the corresponding circuit breaker.

7.4 Monitor or Wait For a Response

If you use the asynchronous interface Perseus.Go(), you will receive a channel of errors that can be monitored. To monitor errors and successful response in a graceful manner, you can use the following structure:

```
output := make(chan bool, 1)
errors := Perseus.Go("my_command", func() error {
        // your business logic
        // which may depend on remote services
        output <- true
        return nil
}, nil)

select {
case out := <-output:
        // success
case err := <-errors:
        // failure
}
```

If you use the synchronous interface Perseus.Do(), you call a command and wait for it to finish and receive a single error:

```
err := Perseus.Do("my_command", func() error {
        // talk to other services
        return nil
}, nil)
```

7.5 Configure Settings If You Need

If you want to change parameters relating to the fault tolerance behaviour to meet your own requirements, you can call .ConfigureCommand() to customise your command whenever you want. You need to import the config package of Perseus:

```
import pconfig "Perseus/config"
pconfig.ConfigureCommand("user", pconfig.CommandConfig{
        Timeout: 1000,
        MaxConcurrentRequests: 20,
        RequestVolumeThreshold: 100,
        SleepWindow: 5000,
        ErrorPercentThreshold: 20,
})
```

You can also use .Configure() which accepts a map[string]CommandConfig.

8 Experiments

In real-world applications, different components of a distributed system run on different processes, or even on different machines. In the experiments, however, different processes are running on a local computer to simulate real-world situations due to computing resource limitations. Each server in the following diagrams is running a service process on different ports of my computer and they communicate with each other via Socket. Apart from Perseus, the main Golang library that is used in the experiments is 'net'.

8.1 Experiment 1: A Chain of Microservices Where Timeout Occurs

Chains of microservices are common patterns of remote calls between servers in a distributed system where microservices architecture is applied. When a client sends a request to a server, it is possible that the client will wait for the request for an unbearably long time consuming the computing resources or even wait forever. In this case, Perseus provides timeout pattern to avoid unnecessary resource consumption by executing a fallback function and make the system maintain the ability to handle further requests.

In this experiment, three servers form a chain of remote calls. Server0 calls server1, after which server1 calls server2, as shown in the following sequence diagram. To simulate the timeout situation, I let server1 sleep for 10 seconds when handling requests from server0. The timeout limit configured in Perseus in this experiment is 1 second (timeout limit is configurable by users).

I ran the program in different situations, including (1) normal situation, (2) a situation where timeout occurs without Perseus, (3) Perseus handling timeout without fallback function and (4) Perseus handling timeout with fallback function. I measured run duration and logged the program results (success/failure)

respectively(see table 1).

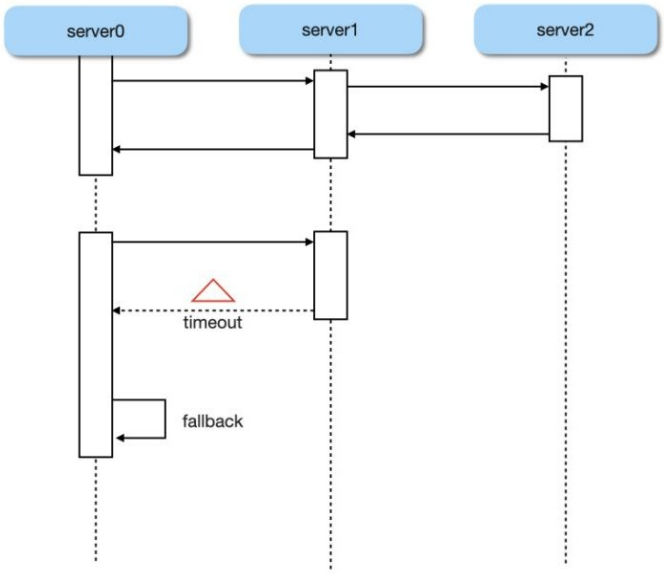

Figure 5. Sequence Diagram of Situations (1) and (4)

	Run Duration	Results
Normal	4.04ms	Success
Timeout	10015.36ms	Success
Normal with Perseus	4.09ms	Success
Timeout with Perseus (Fallback function implemented, timeout limit is 1000ms)	1004.10ms	Success
Timeout with Perseus (Fallback function not implemented, timeout limit is 1000ms)	1002.61ms	Failure

Table 1. Statistics of Experiment 1

The table indicates that if timeout occurs and Perseus is absent, server0 will keep hanging on until it receives a response from server1, in which case run duration is over 10 seconds. However, if Perseus is applied to the system, server0 will stop waiting after waiting for 1 second, and it will then execute the fallback function or directly return, according to whether the fallback function is implemented or not by the user. In both cases (fallback function implemented/not implemented), run duration is over 1 second and program results are ‘success’ and

‘failure’ respectively. The error returned by the ‘failure’ situation is ‘Perseus: timeout’ and this error will be sent to the upstream to deal with.

Comparing run duration of ‘Normal’ and ‘Normal with Perseus’, the performance overhead of Perseus, in this case, is tiny, even can be ignored. When there are 100 thread pools, the run duration are 548.29ms and 558.65ms, respectively. The computational overhead mainly involves threading (queuing, scheduling, and context switching), metrics, logging, circuit breaker and so on. When there are large amount requests sent to each thread pool, the cost of Perseus will be relatively more obvious, but is minor enough and do not have a major performance impact.

8.2 Experiment 2: Different Flow Capabilities of Different Thread Pools

In complex distributed systems, concurrency is used to improve the throughput of the system. In real-world circumstances, different business types have different throughput requirements. For example, there are business types like ‘user’, ‘order’ and ‘delivery’ on an E-commerce platform, each of these provides different functions to make the whole system working. Requests of business type ‘user’ may be much more frequent than those of business type ’delivery’. Thus, it is sensible for threads of different business types to run in isolated thread pools to improve the entire efficiency of the platform.

In this experiment, I use goroutines to achieve concurrency. Goroutines can be simply seen as threads in a process, but more lightweight. Each server is running a process on a certain port of my computer, and several goroutines are launched in a certain process. I only use two servers in this experiment, for that it is easily generalised to more servers in terms of using isolated thread pools.

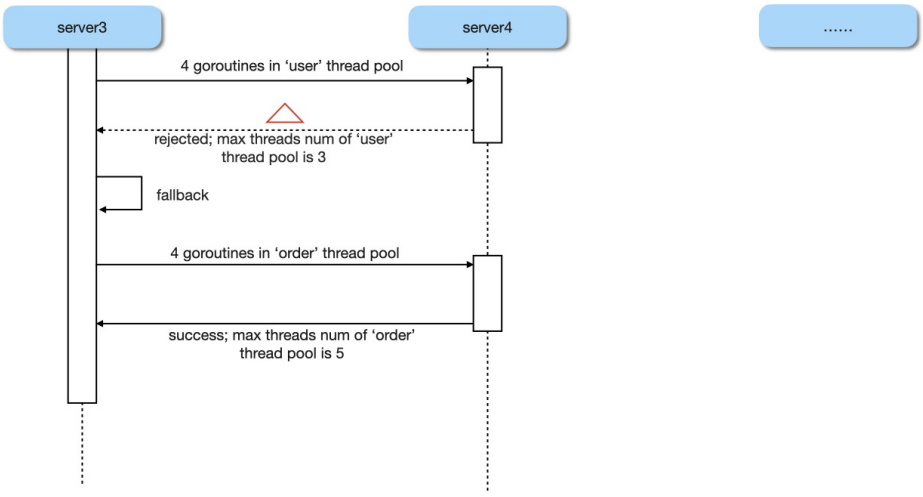

Figure 6. Sequence Diagram of Experiment 2

Figure 6 presents that Perseus controls the max threads number in different threads pools identified by the business types. Server3 concurrently sends 4 requests of business type 'user' to server4, and only three reached the destination while the 4th goroutine is rejected by the thread pool because the limit of thread number of 'user' thread pool is 3(this number is configurable in Perseus), which is smaller than the number of current threads. As a comparison, server3 can launch 4 goroutines to send a request of type 'order' to server4 because the max number of 'order' thread pool is 5, which is larger than the number of current threads.

8.3 Experiment 3: Circuit Open and Recover in Different Thread Pools

In a business system of multiple servers, it is essential to isolate failures to limit the impact of any one dependency. Otherwise, a single small failure can stop the whole system from working. A good way of achieving this is using circuit breaker pattern, where each thread pool corresponds to one circuit breaker and vice versa. When a client sends requests over a certain amount and these requests fail up to a certain error percentage, the corresponding circuit will be tripped so that further requests sent to the circuit will fail fast instead of queuing while those

sent to other circuits will not be affected. After a period of time, requests sent to the circuit will be executed again to test if a successful response can be returned. If so, the circuit will be recovered (closed); otherwise, the circuit will keep open until the next testing time.

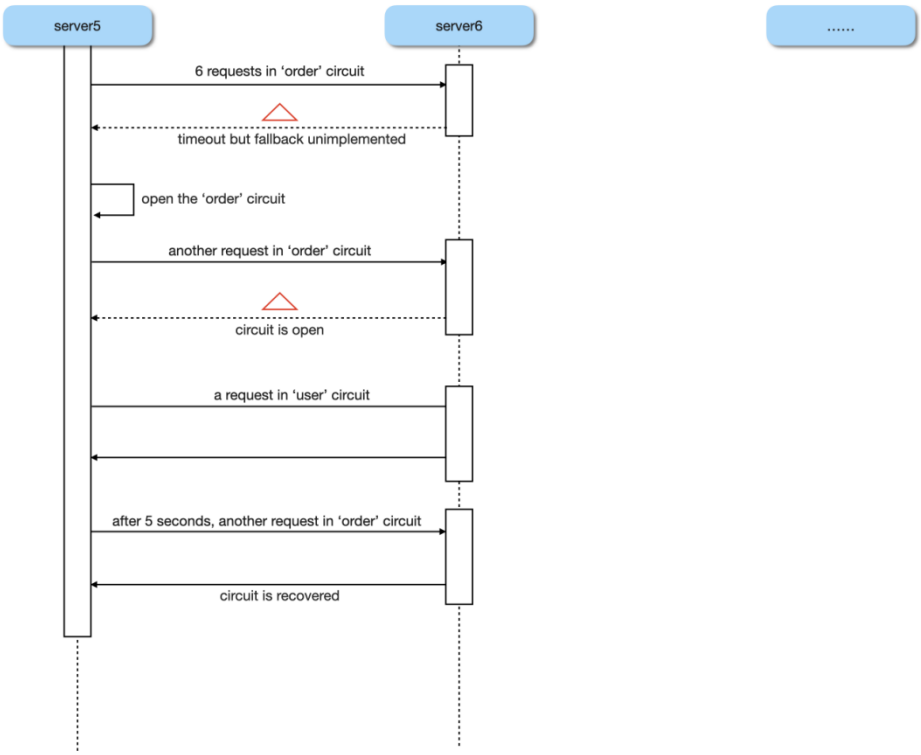

Figure 7. Sequence Diagram of Experiment 3

In this experiment, server5 and server6 simulate how a circuit can be opened and recovered while isolated from other circuits (see Figure 7). The value of volume threshold (the minimum number of requests needed in the past 10 seconds before a circuit can be tripped due to health) of 'order' circuit is 6 and the error percentage threshold is 0.5 (the minimum error percentage needed in the past 10 seconds before a circuit can be tripped due to health) here, both of which are configurable in Perseus. Firstly, server5 sends 6 requests in 'order' circuit, all of which timeout with fallback unimplemented within 10 seconds, which tripped the 'order' circuit. While the 'order' circuit is open, any further attempt to send a request in the 'order' circuit will be blocked and won't reach the destination. After

passing the sleep window (which is how long to wait after a circuit opens before testing for recovery and configurable in Perseus), a request is executed successfully in ‘order’ circuit and recover the ‘order’ circuit. While ‘order’ circuit can be opened or closed due to health checks, ‘user’ circuit is totally isolated and can be opened or closed according to its own health conditions.

To conclude, these three experiments simulate different real-world scenarios that Perseus can be applied to make the whole system more resilient. Perseus is very easy to use, for that you only need to wrap your command into `.Go()`. Also, Perseus is efficient in terms of the extra overhead produced by it. However, it tends to be much more servers running in a distributed system and the business logic is usually much more complex in the real scenarios, the overhead produced by Perseus and the fault tolerance requirements of the system should be trade-offs that the development team should be carefully consider.

9 Conclusions

9.1 Summary

The main outcome of this project is Perseus, an open-source latency and fault tolerance library developed in Golang providing methods that can make a system applying microservices architecture more reliable. The functions of Perseus include timeout, fallback, command thread pools and circuit breaker, which together help the system behave gracefully when a dependency fails.

Apart from the library itself, the project also carries out several experiments to show that Perseus is of great use when multiple servers communicate with each other via remote network calls. Experiments also indicate that the overhead of Perseus is negligible. Although real-world applications may have much more servers and have much more complex business logic compared to those on my personal device for the experiments, it is expected that the overhead of Perseus is acceptable when the system needs resilience assurance.

This report shows the necessity of fault tolerance mechanism in the world of microservices and introduces several design patterns relative to service resilience, all of which have been implemented and integrated into Perseus. Plus, details about the design, implementation, unit testing are also contained in this report, which may help readers understand how Perseus works.

9.2 Limitations

This library can enhance the stability of a system that applies the traditional microservice architectures while there are still some limitations.

(1) High development cost when calling across languages. In most companies, there are usually multiple business teams, and the development languages used by each team are usually different. Take the business of a mobile app as an example, backends use PHP as the development language, API platform is developed in

Java, and front ends call the API platform using HTTP requests. If it is to be service-oriented and changed to RPC, a service-oriented framework that supports both PHP and Java language is needed. There are several open-source service-oriented frameworks either bound to a specific language, such as Dubbo and Spring Cloud, which only support the Java language, or cross-language, such as gRPC and Thrift. For a cross-language RPC framework, an IDL file is necessary, based on which backends and frontends generate SDKs in their own languages. In this case, fault tolerance functions need to be implemented in the SDKs of each language, resulting in high development costs for development teams.

(2) Inconsistent with the concept of cloud-native applications. In traditional microservice architectures, in addition to the implementation of its own business logic, the service consumer also needs to integrate some logic of the service framework, such as service discovery, load balancing, circuit breaker, interface encapsulation, etc. Similarly, on the service provider side, some logic of service framework should also be integrated, such as thread pools and flow limiting. Traditional service governance requires the integration of the SDK of the service framework in the business code, which is obviously contrary to the concept of cloud-native applications. Microservices have begun to develop in the direction of containerization, and use various container platforms to manage services, and gradually evolve in the direction of cloud-native applications.

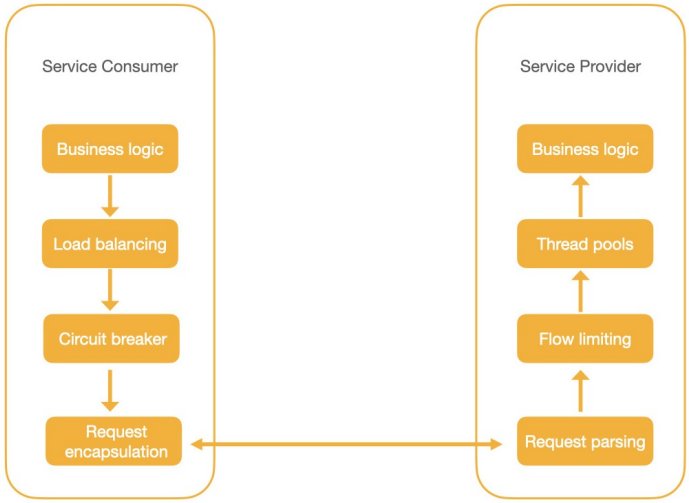

Figure 8. traditional microservice architectures

9.3 Next Generation of Microservices

With the popularity of microservices among technology companies, there is an increasing demand for cross-language service calls; on the other hand, thanks to the containerization of microservices, more and more cloud-native applications are deployed on cloud platforms such as Kubernetes. The demand for service governance is also becoming stronger. Based on this, the idea of Service Mesh came into being.

It is believed that Service Mesh is the next generation of microservices architecture. The concept of Service Mesh was first proposed in an article by William Morgan, CEO of Buoyant[13]. The definition of service mesh he gave is:

> *A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It's responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud-native application. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware.*

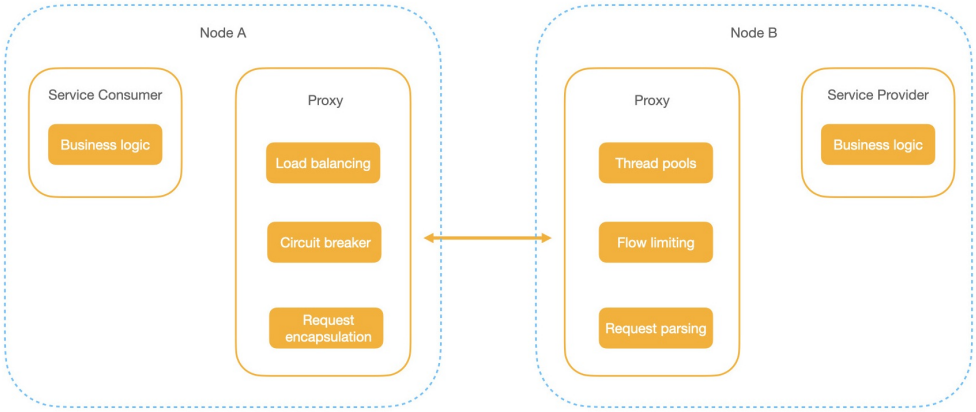

Figure 9. Service Mesh

Service Mesh has achieved remarkable development in less than two years since its birth, and a number of representative products have emerged. The most famous one is Istio led by Google and IBM. In the future, I plan to study the overall architecture and implementation principles of the service mesh, and how the fault tolerance mechanism is integrated into it.

10 Reference

[1] Livora, Tomáš. Fault Tolerance in Microservices. 2017.
[2] Newman, Sam. Building Microservices. 2015.
[3] James Lewis and Martin Fowler. Microservices Guide, 2014.
https://www.martinfowler.com/microservices/
[4] Netflix. Hystrix. https://github.com/Netflix/Hystrix
[5] Dragoni, Nicola, et al. “Microservices: Yesterday, Today, and Tomorrow.” Present and Ulterior Software Engineering, 2017, pp. 195–216.
[6] Martin Fowler and James Lewis. Microservices, 2014.
http://martinfowler.com/articles/microservices.html.
[7] Nygard, Michael. Release It!: Design and Deploy Production-Ready Software. 2007.
[8]Unit Testing in Golang.
https://golangdocs.com/unit-testing-in-golang#:~:text=Unit%20testing%20in%20GoLang%20-%20GoLang%20Docs%20Unit,after%20another%20but%20does%20not%20test%20it%20entirely.
[9] Samir Behara. Making your microservices resilient and fault tolerant.
https://dzone.com/articles/making-your-microservices-resilient-and-fault-tole-1
[10] Martin Fowler. CircuitBreaker. 2014. http://martinfowler.com/bliki/CircuitBreaker.html
[11] Martin Fowler. Application Boundary.
https://www.martinfowler.com/bliki/ApplicationBoundary.html
[12] Eberhard Wolff. Why Microservices Fail: An Experience Report. 2019
[13] William Morgan. What's a Service Mesh? And Why Do I Need One?
https://dzone.com/articles/whats-a-service-mesh-and-why-do-i-need-one#:~:text=What%20Is%20a%20Service%20Mesh%3F%20A%20service%20mesh,services%20that%20comprise%20a%20modern%2C%20cloud%20native%20application.