Design For Chaos

Or how not to automate failure at scale

How apps fail

- Immediately
- Gradually
- Spectacularly

Some notable failures

- Google SRE erasing all disks at their satellite datacenter
- Amazon overwhelming their network with infrastructure tooling RPCs
- Facebook’s network outage that was so bad they couldn’t unlock doors on campus

What we will talk about

- Using overload prevention mechanisms
- Using rate limiters to prevent runaway workflows
- Idempotency and its importance in workflow actions
- Building systems with an emergency stop

Overload Prevention Mechanisms

Overwhelming your network and service

- Chances are that your network is oversubscribed
- And your services cannot handle all their clients hitting the service at once

The most common solutions

- Circuit breakers
- Backoff implementations

The Circuit Breaker

- Comes in 3 states
  - Closed - everything is working
  - Open - some number of failures tripped the breaker
  - Half One - a transition form the open state where we try requests again
- One of the more popular Go circuit breakers: github.com/sony/gobreaker

Example: HTTP Client

```go
// HTTP is a wrapper around http.Client that implements the CircuitBreaker pattern for Get requests.
type HTTP struct {
    client *http.Client
    cb     *gobreaker.CircuitBreaker
}

// New creates an new HTTP instance.
func New(client *http.Client) *HTTP {
    return &HTTP{
        client: client,
        cb: gobreaker.NewCircuitBreaker(
            gobreaker.Settings{
                MaxRequests: 1,                // only one request at a time if in the half-open state
                Interval:    30 * time.Second, // how long before we can leave the Half-Open state
                Timeout:     10 * time.Second, // how long to wait in Open before transiting to Half-Open
                ReadyToTrip: func(c gobreaker.Counts) bool {
                    return c.ConsecutiveFailures > 5 // after 5 failures, trip the circuit
                },
            },
        ),
    }
}
```

Example: HTTP Client

(Continued)

```go
// Get executes an HTTP GET request.
func (h *HTTP) Get(req *http.Request) (*http.Response, error) {
    if _, ok := req.Context().Deadline(); !ok {
        return nil, fmt.Errorf("all requests must have a Context deadline set")
    }

    r, err := h.cb.Execute(
        func() (any, error) {
            resp, err := h.client.Do(req)
            if resp.StatusCode != 200 {
                return nil, fmt.Errorf("non-200 response code")
            }
            return resp, err
        },
    )
    if err != nil {
        return nil, err
    }
    return r.(*http.Response), nil
}
```

Let’s see it action

- Code at: github.com/johnsiilver/gofordevopsclass/design_for_chaos/circuitbreakers/httpclient
- Startup a web application
  - A sample one can be run at:
    - github.com/johnsiilver/gofordevopsclass/automation_the_hard_way/orchestration/lb/sample/web/
    - Can run it with ./web —port [port of your choice]
  - Run the httpclient with:
    - go run . https://127.0.0.1:[port]
  - Turn the web server on and off to see the changes

The Backoff method

- On a failure, introduce some amount of delay before trying again
- Increase the delay until some maximum time
- Introduce some entropy into the delay to prevent retry synchronization across services
- One of the more popular packages: pkg.go.dev/github.com/cenk/backoff
  - Based on Google's Java backoff package

Example: HTTP Client

```go
func (h *HTTP) Do(req *http.Request) (*http.Response, error) {
    if _, ok := req.Context().Deadline(); !ok {
        return nil, fmt.Errorf("all requests must have a Context deadline set")
    }
    var resp *http.Response

    op := func() error {
        var err error
        resp, err = h.client.Do(req)
        if err != nil {
            log.Println("error: unable to fetch URL: ", err)
            return err
        }
        if resp.StatusCode != 200 {
            return fmt.Errorf("non-200 response code")
        }
        return nil
    }

    err := backoff.Retry(
        op,
        &backoff.ExponentialBackOff{
            InitialInterval:     2 * time.Second,
            RandomizationFactor: 0.5,
            Multiplier:          2,
            MaxInterval:         10 * time.Second,
            Clock:               backoff.SystemClock,
        },
    )
    if err != nil {
        return nil, err
    }

    return resp, nil
}
```

Let's see it action

- Code at: github.com/johnsiilver/gofordevopsclass/design_for_chaos/backoff/httpclient
- Startup a web application
  - A sample one can be run at:
    - github.com/johnsiilver/gofordevopsclass/automation_the_hard_way/orchestration/lb/sample/web/
    - Can run it with ./web —port [port of your choice]
  - Run the httpclient with:
    - go run . https://127.0.0.1:[port]
  - Turn the web server on and off to see the changes

Using Rate Limiters To Prevent Runaways

Speed Kills

- More than a few major incidents could have been prevented by being slow and methodical
- When doing operational work, it is important to have rate limiters in place to prevent runaways
- Nothing like rebooting all your routers with a firmware upgrade at once or upgrading an entire cluster of machines at the same time to ruin your day

Creating a localized rate limiter

Let’s define a Work, Block and a Job object

```
type Work struct {
    Blocks []Block
}

type Block struct {
    Job []Job
}

type Job interface {
    Validate(job *pb.Job) error
    Run(ctx context.Context, job *pb.Job) error
}
```

Creating a localized rate limiter (Continued)

- Run some rate limited jobs

```
wg := sync.WaitGroup{}
defer wg.Wait()

limit := make(chan struct{}, limit)
errCount := atomic.Int64{}

for _, block := range work.Blocks {
    if errCount.Load() > errLimit {
        return fmt.Errorf("too many errors")
    }
    for _, job := range block.Jobs {
        if errCount.Load() > errLimit {
            return fmt.Errorf("too many errors")
        }

        job := job
        limit <- struct{}{}

        wg.Add()
        go func() {
            defer wg.Done()
            defer func() {
                <-limit
            }()

            if err := job.Run(ctx); err != nil {
                log.Println(err)
                errCount.Add(1)
            }
        }()
    }
    wg.Wait()
}
```

Token Buckets

- A token bucket can also be used to control execution rates
- For workflow use, this is more often used to control a total number of operations across workflows instead of a single workflow limit, like the total number of servers than can be touched in a cluster at once
- A simple example can be found here:
  - github.com/PacktPublishing/Go-for-DevOps/blob/rev0/chapter/16/workflow/internal/token/token.go

Idempotency

What is Idempotency?

- The simple definition is that if you make a call with the same parameters multiple times, you receive the same result
- In infrastructure, this definition is slightly modified:
  - An idempotent action is one that, if repeated with the same parameters and without changes to the infrastructure outside of this call, will return the same result.

.

Example of non-Idempotent action

- What happens if our process dies before io.WriteFile is called and we restart and repeat the call?
- What happens if io.WriteFile is called and we restart before the call is recorded and need to repeat it?
  - And what if this created the file, but we can't edit the file and this happens?

```
func CopyToFile(content []byte, p string) error {
    return io.WriteFile(p, content)
}
```

Better version

- Checks to see if the file exists
- If it does, checks the content in the file against the content we want
  - If its the content we want, does nothing
  - If not, writes the new content
- Otherwise, we just write the content

```
func CopyToFile(content []byte, p string) error {
    if _, ok := os.Stat(p); ok {
        f, err := os.Open(p)
        if err != nil {
            return err
        }
        h0 := sha256.New()
        io.Copy(h0, f)
        h1 := sha256.New()
        h1.Write(content)
        if h0.Sum(nil) == h1.Sum(nil) {
            return nil
        }
    }
    return io.WriteFile(p, content)
}
```

Emergency Stop

What is an emergency stop

- A mechanism that pauses or stops automation company wide
- Best integrated into a centralized workflow system
  - This allows you to stop or pause a set or subset of workflows in case of a problem
  - If integrated in something all tools must use, has one place to control all workflow
- The simplest versions have two modes:
  - Go
  - Stop
- More advanced versions have a third mode:
  - Pause

A simple ES implementation

- Store ES information in a text file of JSON entries
- Entries have the name of a workflow and the ES status of “go” or “stop”
- Any status not “go” or “stop” is interpreted as “stop”
- Any workflow not found is considered “stop”
- Read the entries from disk every 10 seconds
- Provides a Status() method to check the status

```
{
    "Name": "SatelliteDiskErase",
    "Status": "go"
}
```

Full version at: github.com/johnsiilver/gofordevopsclass/tree/main/design_for_chaos/es

Status and Info types

- Defines our Status type
- Defines an Info type that will be contained in a JSON file
- Defines a Validate() method to validate the file data

```go
// Status indicates the emergency stop status.
type Status string

const (
    // Unknown means the status was not set.
    Unknown Status = ""
    // Go indicates the matching workflow can execute.
    Go Status = "go"
    // Stop indicates that the matching workflow should not execute and
    // existing ones should be stopped.
    Stop Status = "stop"
)

// Info is the emergency stop information for a particular entry in our es.json file.
type Info struct {
    // Name is the WorkReq type.
    Name string
    // Status is the emergency stop status.
    Status Status
}

func (i Info) validate() error {
    i.Name = strings.TrimSpace(i.Name)
    if i.Name == "" {
        return fmt.Errorf("es.json: rule with empty name, ignored")
    }
    switch i.Status {
    case "go", "stop":
    default:
        return fmt.Errorf("es.json: rule(%s) has invalid Status(%s), ignored", i.Name, i.Status)
    }
    return nil
}
```

Reader type

- Stores entries in atomic.Value
  - Better to use atomic.Pointer today
  - Mutex to protect subscribers as they are added/removed
  - newReader() creates a *Reader
    - This is private as access to ES is provided by a singleton

```go
// Reader reads the es.json file at intervals and makes the data
// and changes to the data available.
type Reader struct {
    entries atomic.Value // map[string]Info

    mu          sync.Mutex
    subscribers map[string][]chan Status
}

func newReader() (*Reader, error) {
    r := &Reader{subscribers: map[string][]chan Status{}}

    m, err := r.load()
    if err != nil {
        return nil, err
    }
    r.entries.Store(m)

    go r.loop()
    return r, nil
}
```

Reader type

- Status returns the ES status of a workflow
- If not set to Go, it returns Stop

```
// Status returns the ES status for the named workflow.
func (r *Reader) Status(name string) Status {
    m := r.entries.Load().(map[string]Info)
    switch m[name].Status {
    case Go:
        return Go

    }
    return Stop
}
```

Reader type

- A loop() method that reads in the file every 10 seconds
- If it can’t read the file, then everything is stopped

```
func (r *Reader) loop() {
    for _ = range time.Tick(10 * time.Second) {
        newInfos, err := r.load()
        if err != nil {
            log.Println(err)
            continue
        }
        r.entries.Store(newInfos)
    }
}
```

Reader type

- load() method that reads in the entries in the JSON file
- Handles duplicate entries
- Ignores entries that don’t validate

```go
func (r *Reader) load() (map[string]Info, error) {
    f, err := os.Open("configs/es.json")
    if err != nil {
        return map[string]Info{}, fmt.Errorf("could not open configs/es.json: %w", err)
    }

    dec := json.NewDecoder(f)
    dec.DisallowUnknownFields()

    m := map[string]Info{}

    for dec.More() {
        info := Info{}
        if err := dec.Decode(&info); err != nil {
            r.entries.Store(map[string]Info{})
            return map[string]Info{}, fmt.Errorf("es.json file is badly formatted, all jobs moving into stop state")
        }
        if _, ok := m[info.Name]; ok {
            log.Printf("es.json file has two definitions(%s) with the same name, ignoring the second", info.Name)
            continue
        }
        if err := info.validate(); err != nil {
            log.Printf("es.json file has invalid entry(%s), ignoring: %s", info.Name, err)
            continue
        }
        m[info.Name] = info
    }
    return m, nil
}
```

Let’s try it out

- Run our example code
- Change the status in the file from “go” to “stop”
- See what happens

```
ElephantInTheRoom:example jdoak$ go run .
2023/09/25 23:06:49 happily disk erasing
2023/09/25 23:06:50 happily disk erasing
2023/09/25 23:06:51 happily disk erasing
2023/09/25 23:06:52 happily disk erasing
2023/09/25 23:06:53 happily disk erasing
2023/09/25 23:06:54 happily disk erasing
2023/09/25 23:06:55 happily disk erasing
2023/09/25 23:06:56 happily disk erasing
2023/09/25 23:06:57 happily disk erasing
2023/09/25 23:06:58 happily disk erasing
2023/09/25 23:06:59 happily disk erasing
2023/09/25 23:07:00 happily disk erasing
2023/09/25 23:07:01 happily disk erasing
2023/09/25 23:07:02 happily disk erasing
2023/09/25 23:07:03 happily disk erasing
2023/09/25 23:07:04 happily disk erasing
2023/09/25 23:07:05 happily disk erasing
2023/09/25 23:07:06 happily disk erasing
2023/09/25 23:07:07 happily disk erasing
2023/09/25 23:07:08 happily disk erasing
2023/09/25 23:07:09 emergency stop is in effect
exit status 1
```

Summary

How to use circuit breakers and backoff implementations

Using rate limiters to prevent runaway workflows

What Idempotency is and why they are important to workflow actions

What Emergency Stop is and how to use a simple implementation