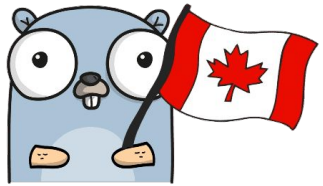
Building a Self-Regulating Pressure Relief Mechanism in Go

GopherCon 2024
Speaker: Ellen Gao

Gopher Introduction

- 4+ years of Go experience
- Backend Engineer at Flybits, Canadian-based tech company

Flybits



Talk Background

- Full solution implemented at Flybits
- Backend made up of 50+ microservices
  - Almost entirely written in Go
- Multiple modes of communication
  - gRPC used for synchronous communication
  - rabbitMQ/kafka used for asynchronous communication
- Today's focus: asynchronous communication

Talk Background

- Full solution implemented at Flybits
- Backend made up of 50+ microservices
  - Almost entirely written in Go
- Multiple modes of communication
  - gRPC used for synchronous communication
  - rabbitMQ/kafka used for asynchronous communication
- **Today's focus: asynchronous communication**

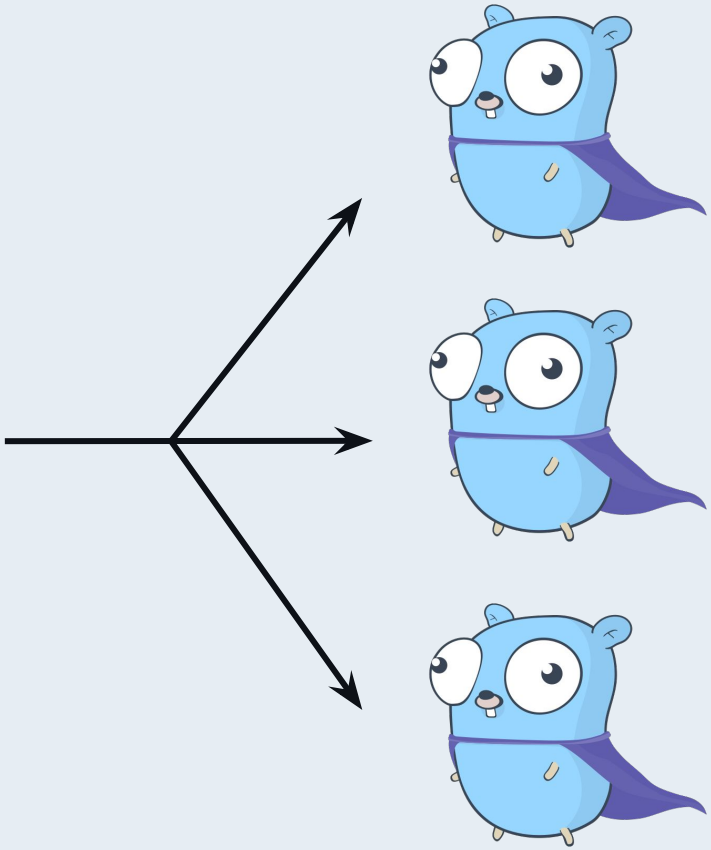
Coming up...

- Problem discovery and end goal definition
- Code setup and terminology
- Pressure Relief via Throttling
  - Iterate! Iterate! Iterate!
- Adaptations for real-world systems

So... what is the **problem**?

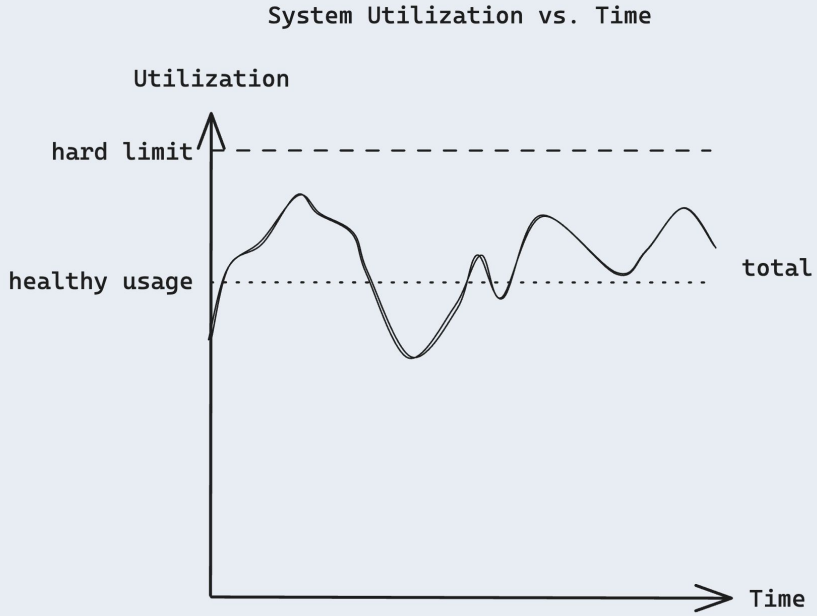
a lot of requests

even more requests



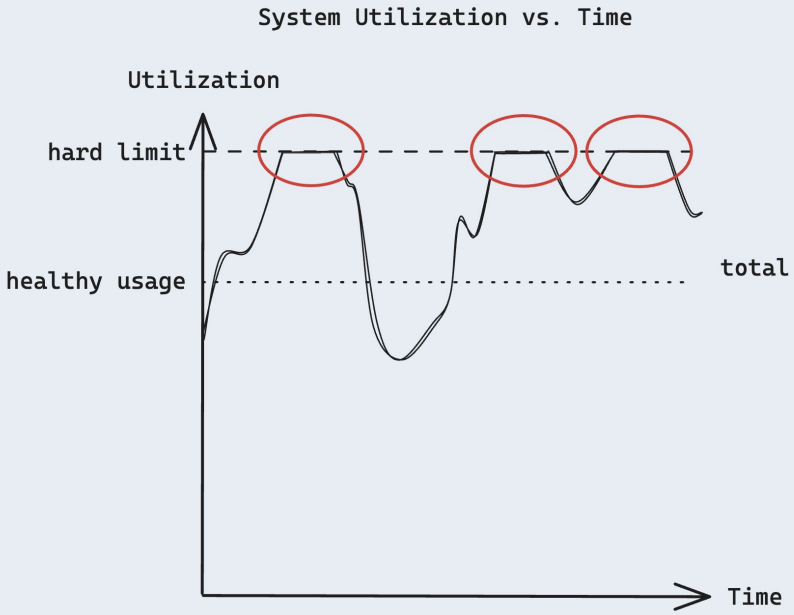

COST OPTIMIZED
PERFORMANCE OPTIMIZED

Healthy Utilization



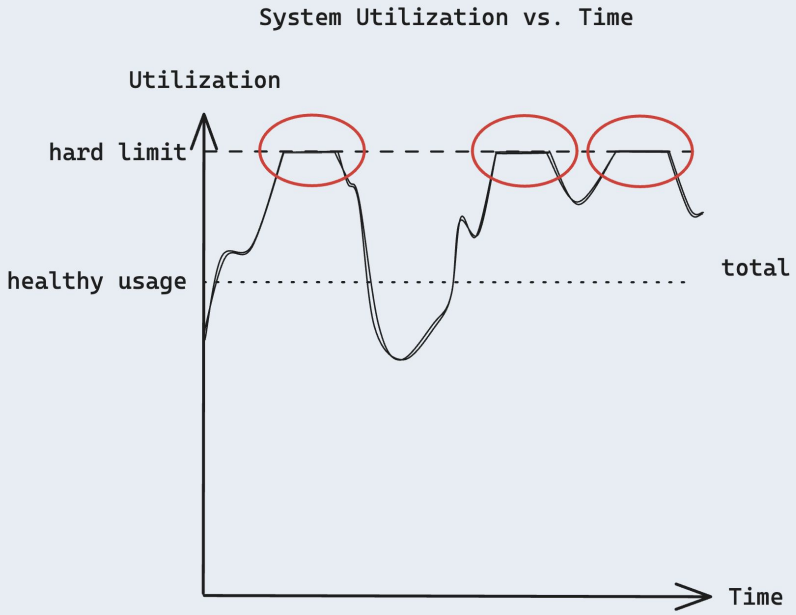
- Utilization constantly fluctuates
- Patterns emerge as fluctuations stabilize over time
- Analyzing patterns helps define normal usage thresholds
- Healthy usage makes good usage of the resource without approaching hard limit

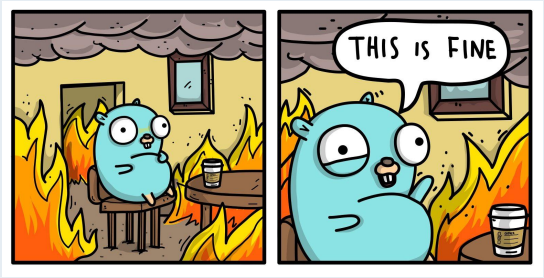
Problematic Utilization



- Resource limit reached
- Requests could be dropped
- Possible degraded performance or system outages
- Potential cascading effects

Problematic Utilization





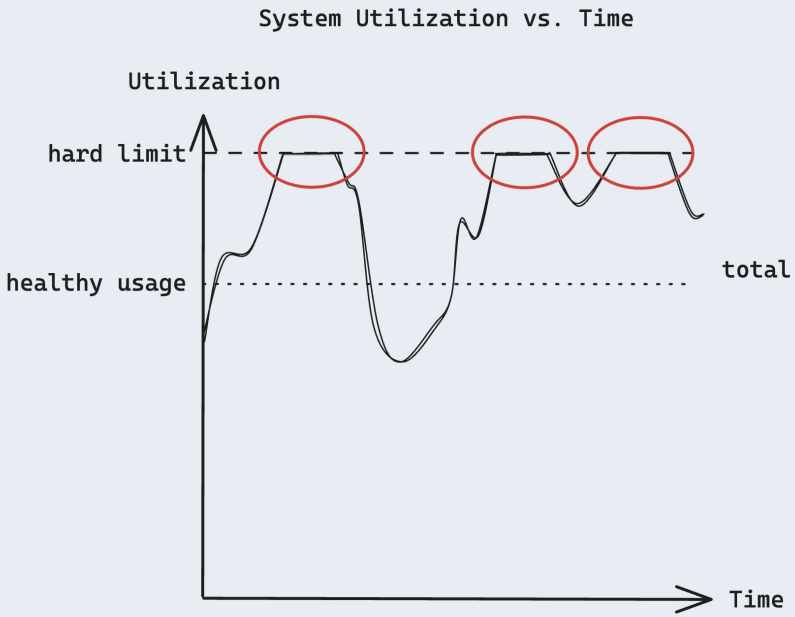
- Resource limit reached
- Requests could be dropped
- Possible degraded performance or system outages
- Potential cascading effects

So... what is the **solution**?

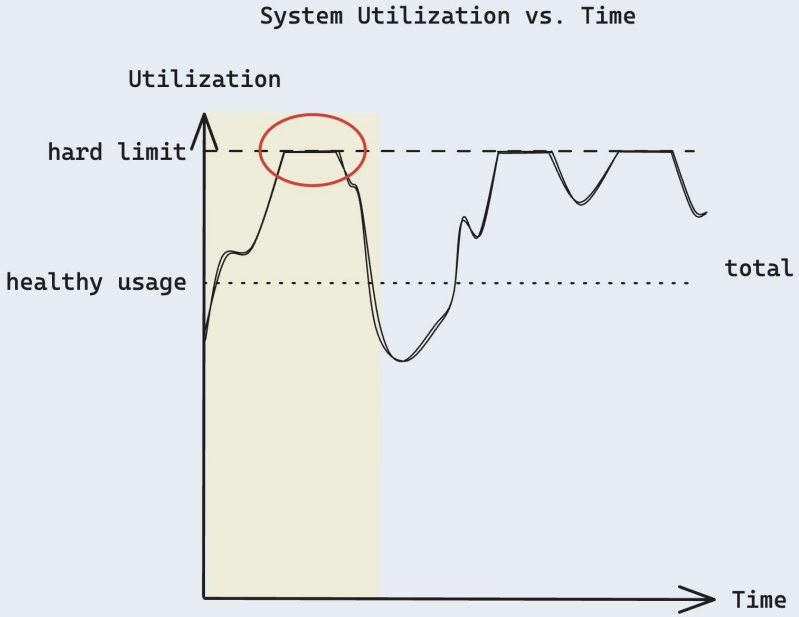
Increase Resources?

- May be best option if problem appears frequently
- Cost may not be justified
- Not always simple (ex. database scaling)

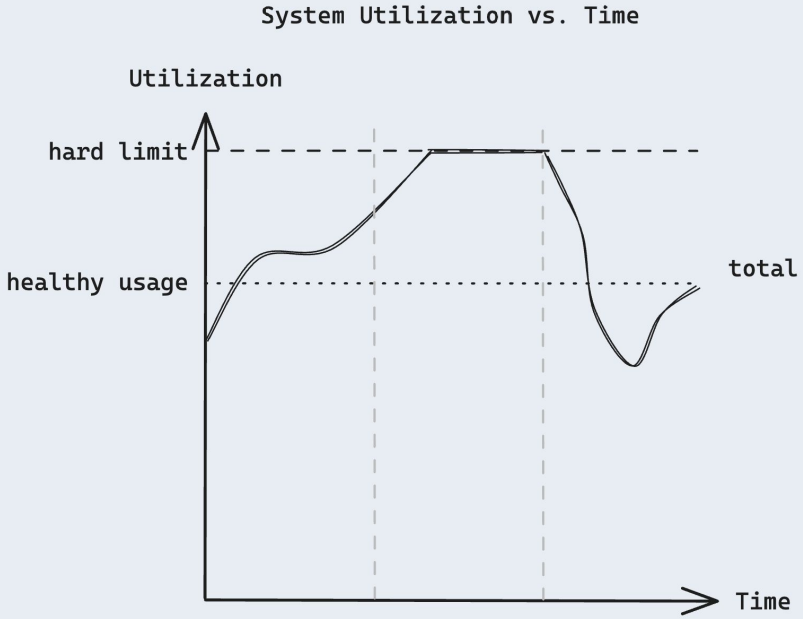
Exploring a Pressure Relief Valve



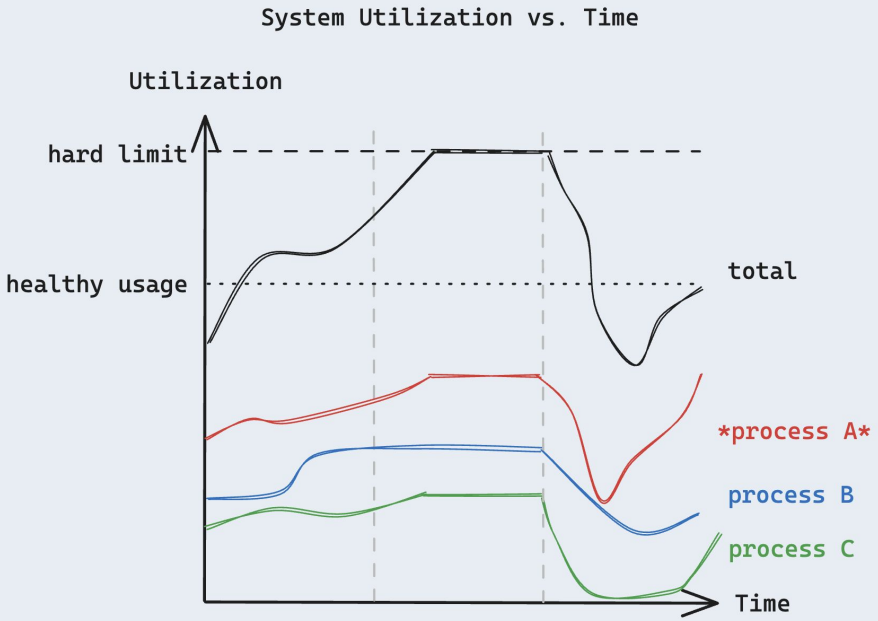
Exploring a Pressure Relief Valve



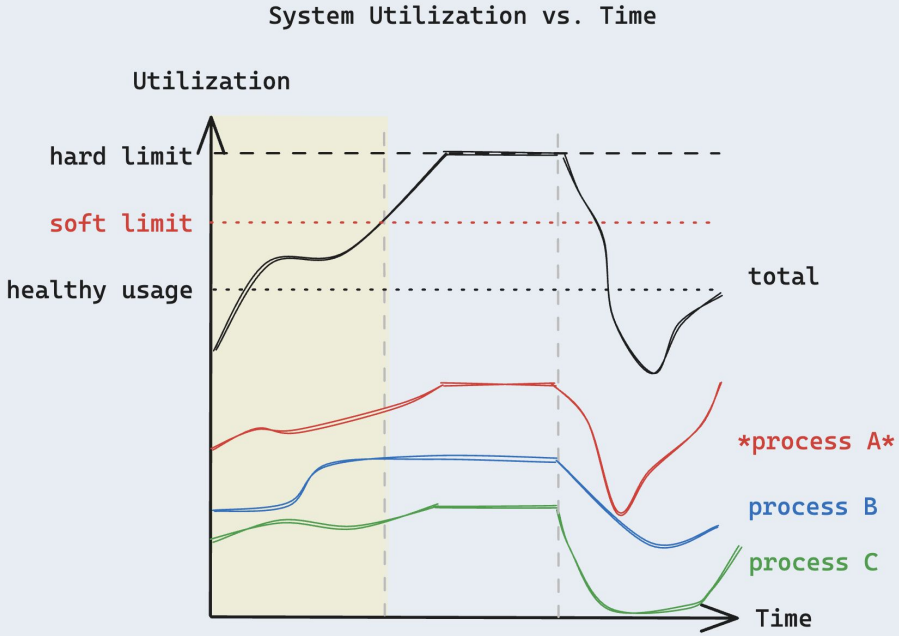
Exploring a Pressure Relief Valve



Exploring a Pressure Relief Valve



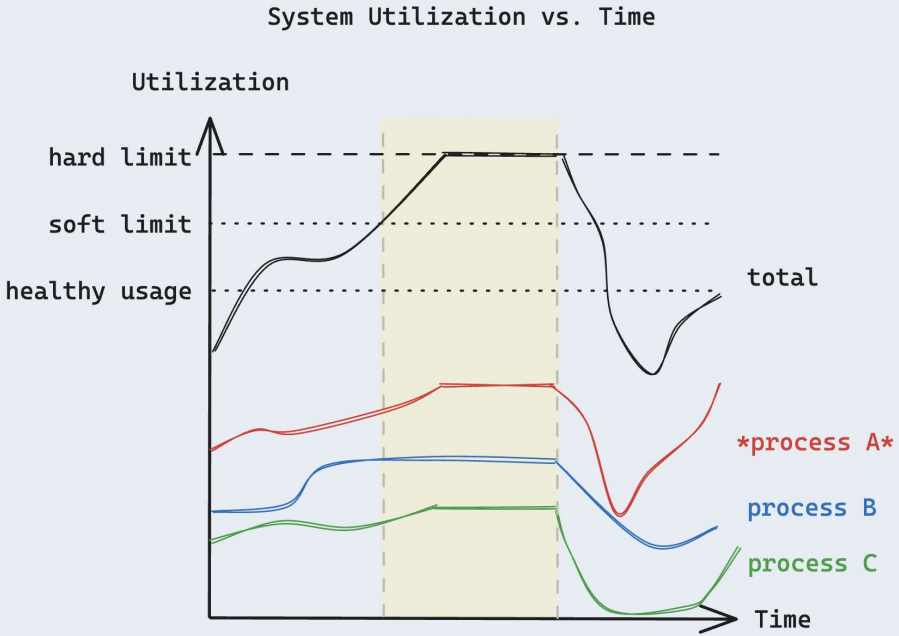
Building a Pressure Relief Valve



Detect

- Define a soft limit at which action is taken

Building a Pressure Relief Valve



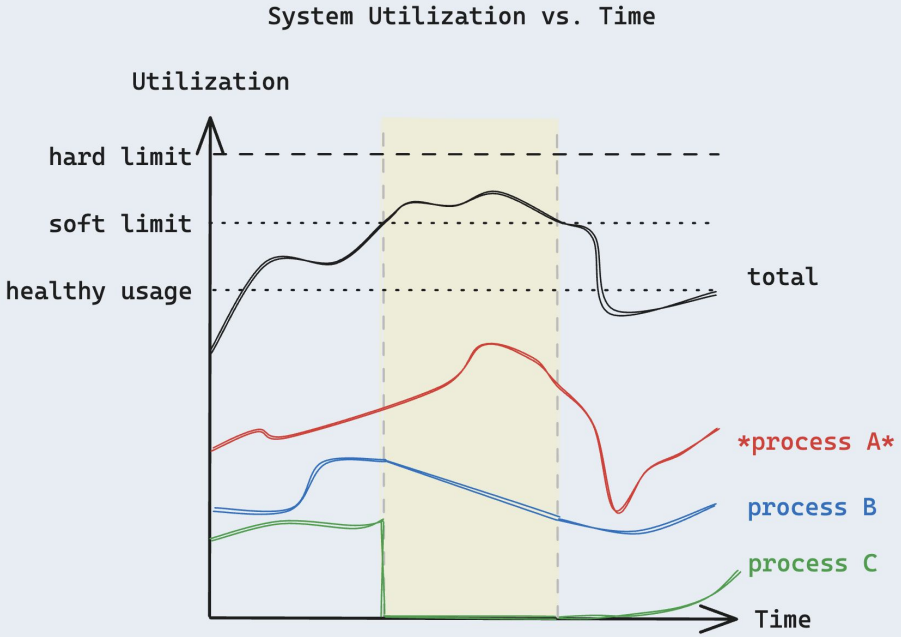
Detect

- Define a soft limit at which action is taken

Respond

- Slow down processing of non-critical processes
- Messages are **not** dropped! They are just processed **later**

Building a Pressure Relief Valve



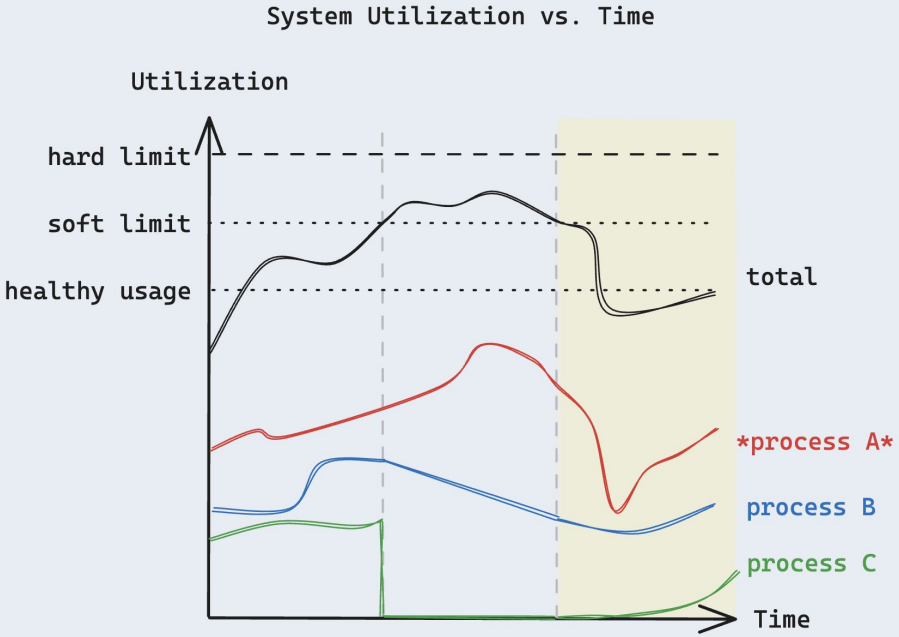
Detect

- Define a soft limit at which action is taken

Respond

- Slow down processing of non-critical processes
- Messages are **not** dropped! They are just processed **later**

Building a Pressure Relief Valve



Detect

- Define a soft limit at which action is taken

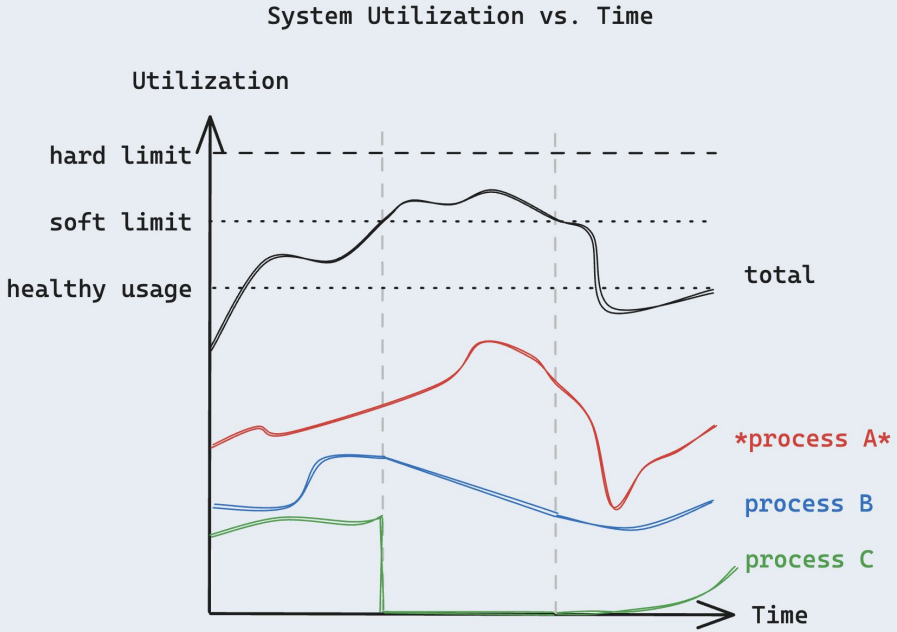
Respond

- Slow down processing of non-critical processes
- Messages are **not** dropped! They are just processed **later**

Recover

- Gradually resume message processing

Summary of Strategy



Alert **activated**

- process A → no action
- process B → slow down
- process C → stop

Alert **resolved**

- process A → no action
- process B → speed up
- process C → begin

Demo Code

Follow along here:

@exgao/GopherCon2024

High-level architecture

Producer

- Publishes messages to a topic

Consumer

- Processes messages from a topic

Messenger

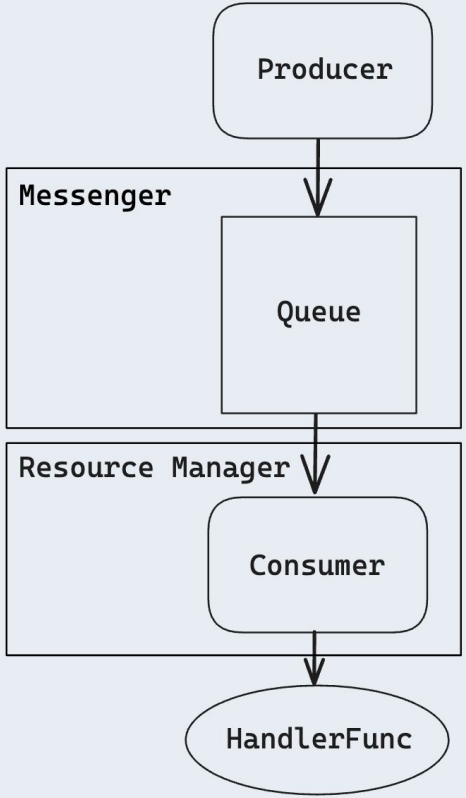
- Routes messages to appropriate queue

Resource Manager

- Keeps track of resource utilization

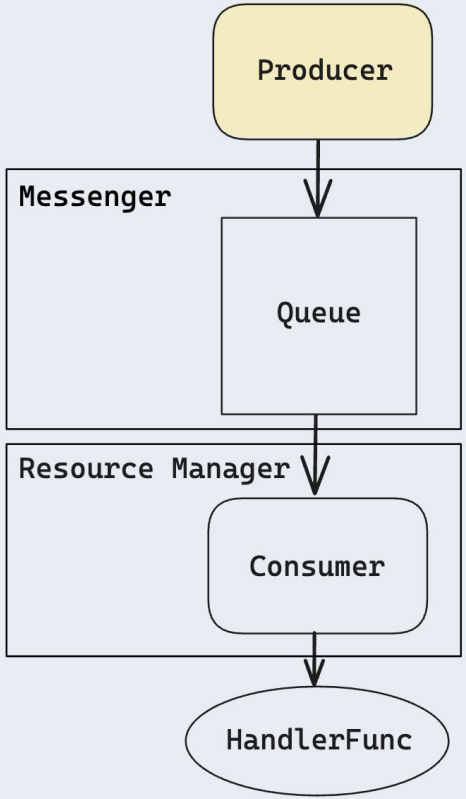
HandlerFunc

- Function for message processing logic



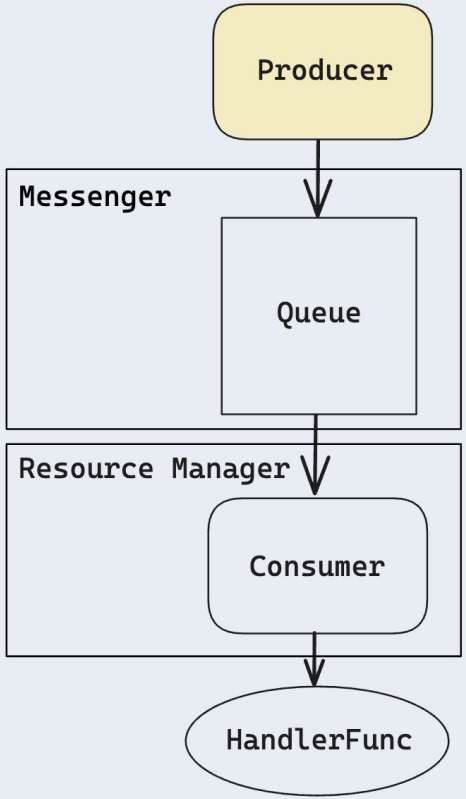
Producer

- Publishes messages to a topic
- Not responsible for message routing



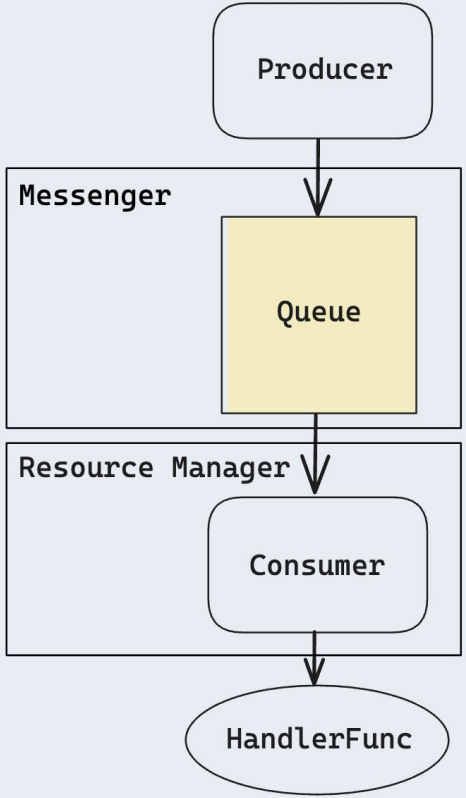
Producer

```
func (p *Producer) Run() {
    go func() {
        // ...
        for {
            p.Publisher.Publish(p.topic, gc.Message{
                Body:         p.topic,
                ResourceCost: rand.Intn(p.resourceMax),
            })
        }
    }()
}
```



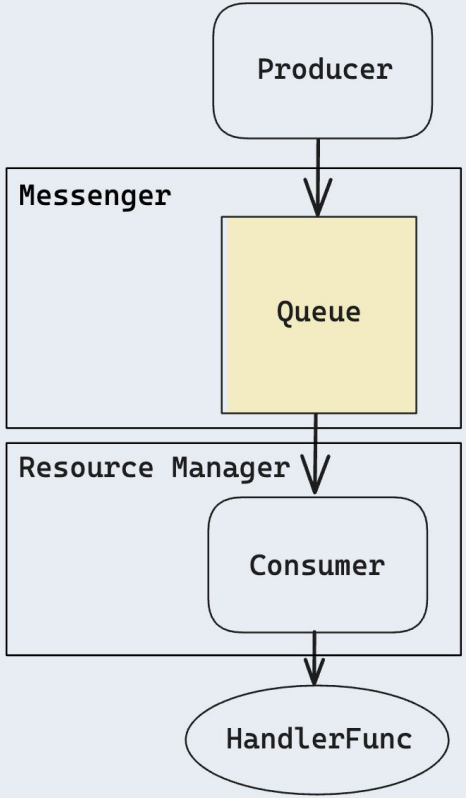
Queue

- FIFO storage mechanism for messages of a topic
- Each queue is associated with a topic and a handler
- Topic is used by the messenger to route and deliver messages
- Handler is invoked by the consumer to hand off a message for processing



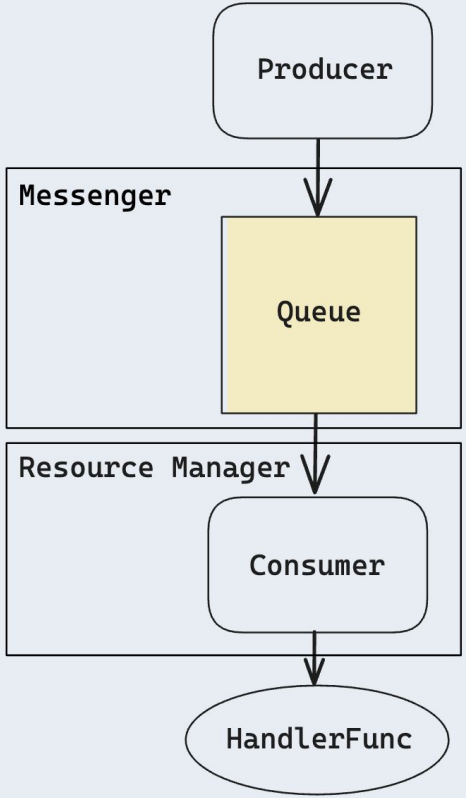
Queue

```
queues := []gc.Queue{
    {
        Topic:   gc.Critical,
        Handler: rm.ProcessMessage,
    },
    // ...
    {
        Topic:   gc.Low,
        Handler: rm.ProcessMessage,
    },
}
```



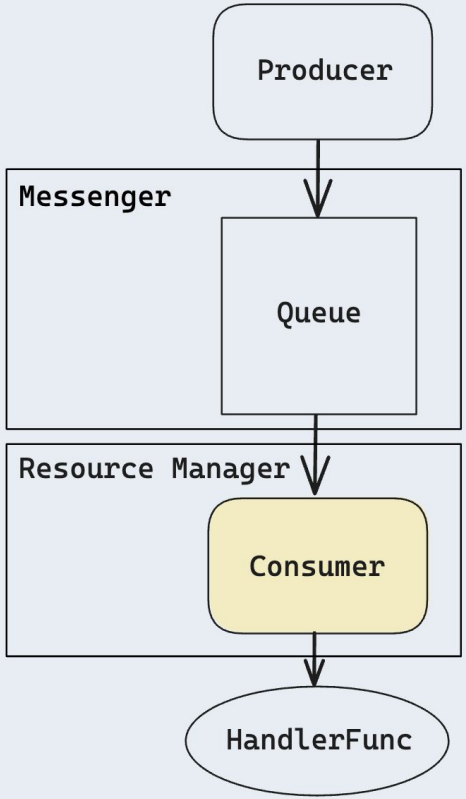
Queue

```
m := &Messenger{
    //...
    CriticalChan:    make(chan Message, 100),
    PriorityChan:    make(chan Message, 100),
    NormalChan:      make(chan Message, 100),
    LowPriorityChan: make(chan Message, 100),
}
```



Consumer

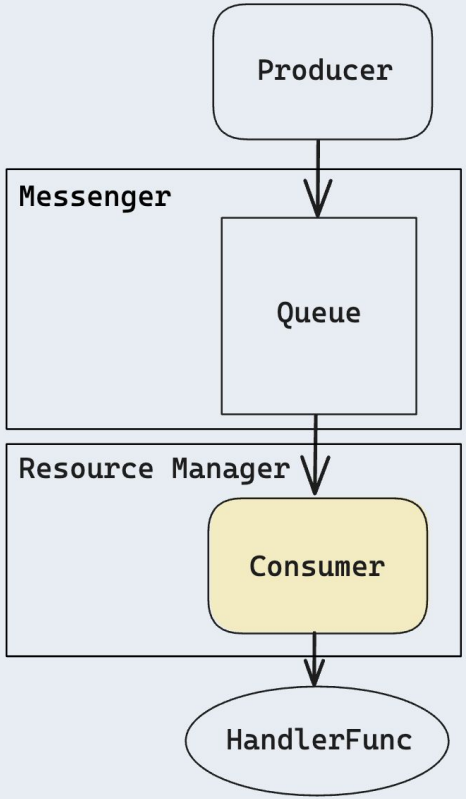
- Accepts messages from a queue
- Forwards message to appropriate handler by invoking HandlerFunc



Consumer

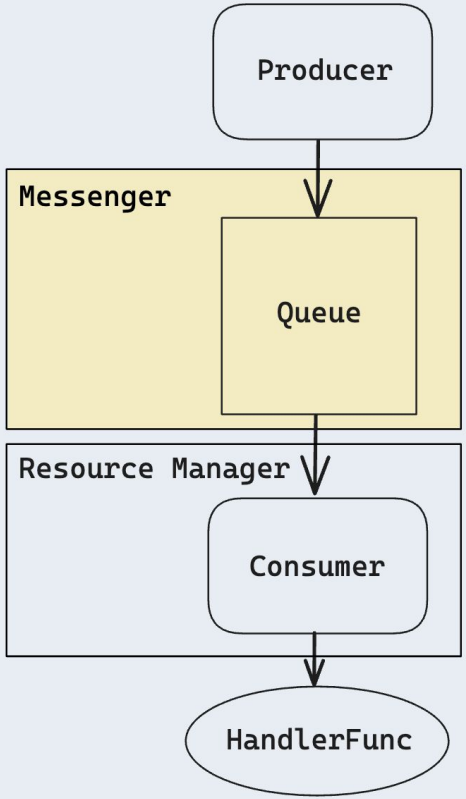
```
func (c *Consumer) Consume(
    deliveryChan <-chan gc.Message,
    handler gc.HandlerFunc,
) {
    // ...
    for {
        d, _ := <-deliveryChan
        // ...

        // invoke handlerFunc
        handler(d)
    }
}
```



Messenger

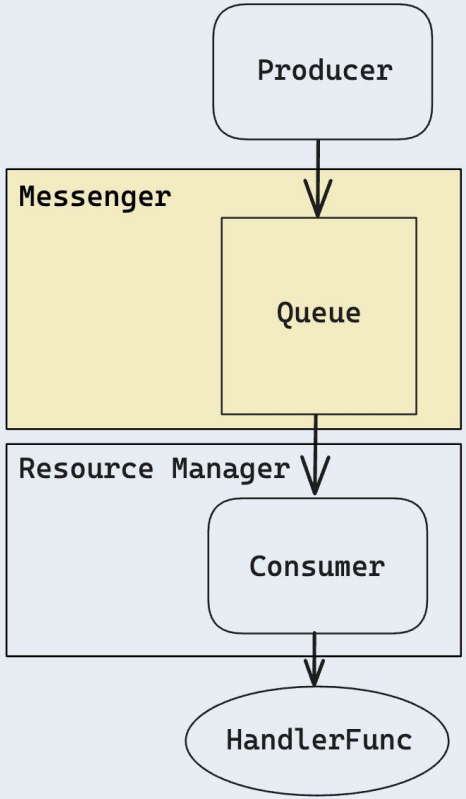
- Routes messages to corresponding queue
- Exposes channel for consumer to accept messages
- rabbitMQ, kafka



Messenger

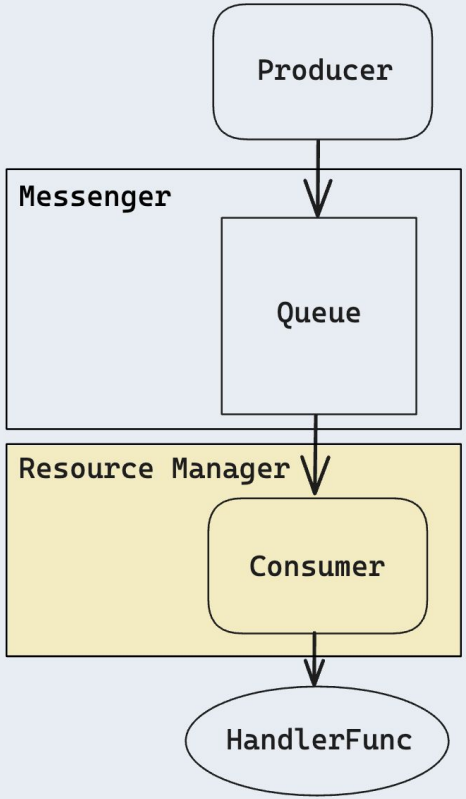
```
func (m *Messenger) Publish(topic string, msg Message) {
    // ...

    switch topic {
    case Critical:
        m.CriticalChan <- msg
    case Priority:
        m.PriorityChan <- msg
    case Normal:
        m.NormalChan <- msg
    case Low:
        m.LowPriorityChan <- msg
    }
}
```



Resource Manager

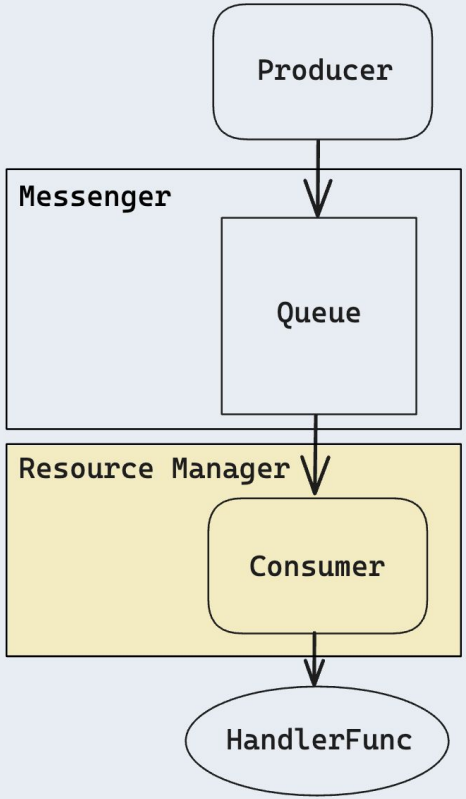
- Updates total utilization as messages are processed
- Keeps track of total processed messages
- Terminates the program if utilization reaches 100
- Publishes alerts if utilization crosses upper or lower threshold



Resource Manager

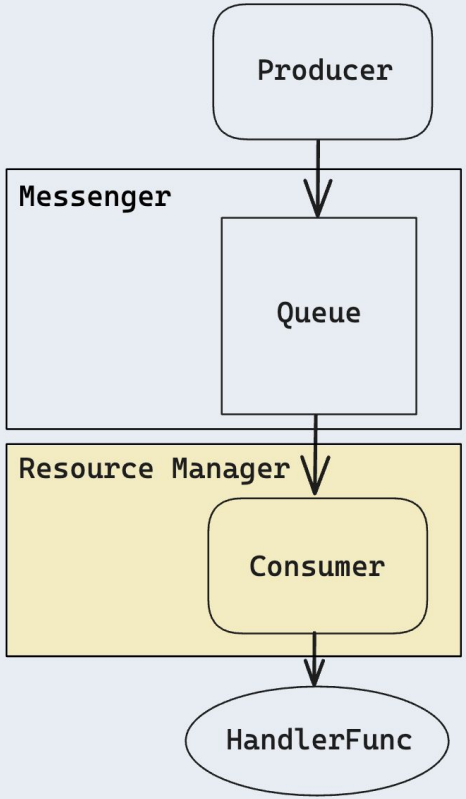
```
func (r *ResourceManager) Add(cost int) {
    r.Utilization += cost
    r.MessagesProcessed++

    if r.Utilization >= 80 && !r.LastAlertVal {
        r.LastAlertVal = true
        for range cap(r.AlertChan) {
            r.AlertChan <- true
        }
    }
    if r.Utilization >= 100 {
        os.Exit( code: 1)
    }
    return
}
```



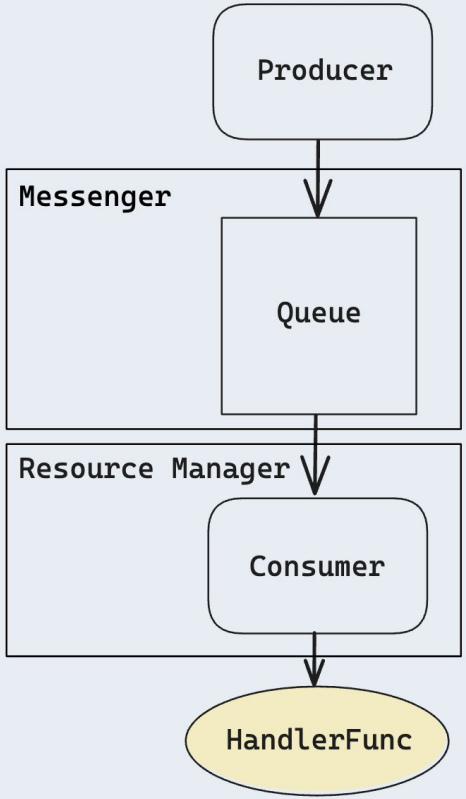
Resource Manager

```
func (r *ResourceManager) Remove(cost int) {
    r.Utilization -= cost
    if r.Utilization < 0 {
        r.Utilization = 0
    }
    if r.Utilization < 40 && r.LastAlertVal {
        r.LastAlertVal = false
        for range cap(r.AlertChan) {
            r.AlertChan <- false
        }
    }
    return
}
```



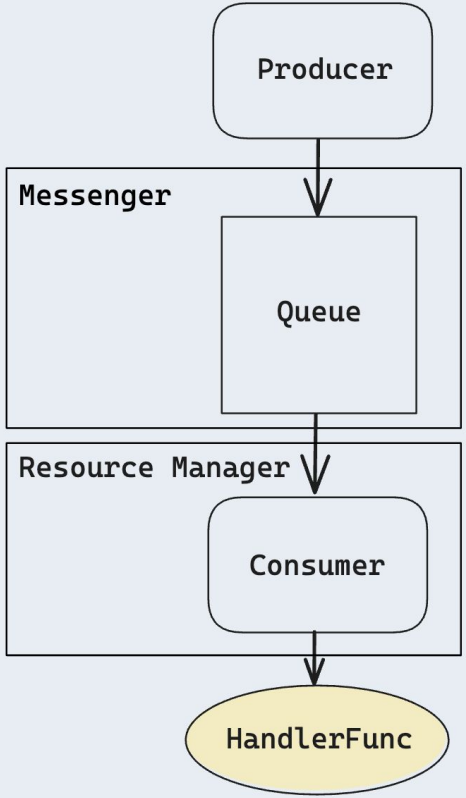
HandlerFunc

- Function type that defines structure for recipient's handler
- Invoked by the consumer to deliver the message to the recipient



HandlerFunc

```
queues := []gc.Queue{
    {
        Topic:   gc.Critical,
        Handler: rm.ProcessMessage,
    },
    // ...
    {
        Topic:   gc.Low,
        Handler: rm.ProcessMessage,
    },
}
```

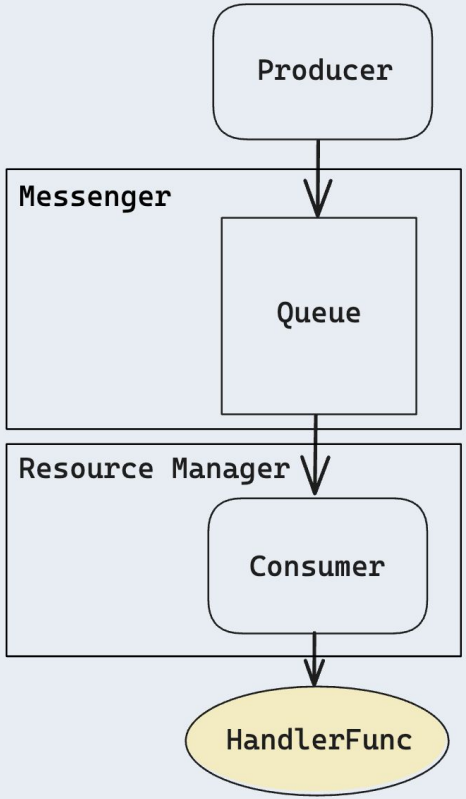


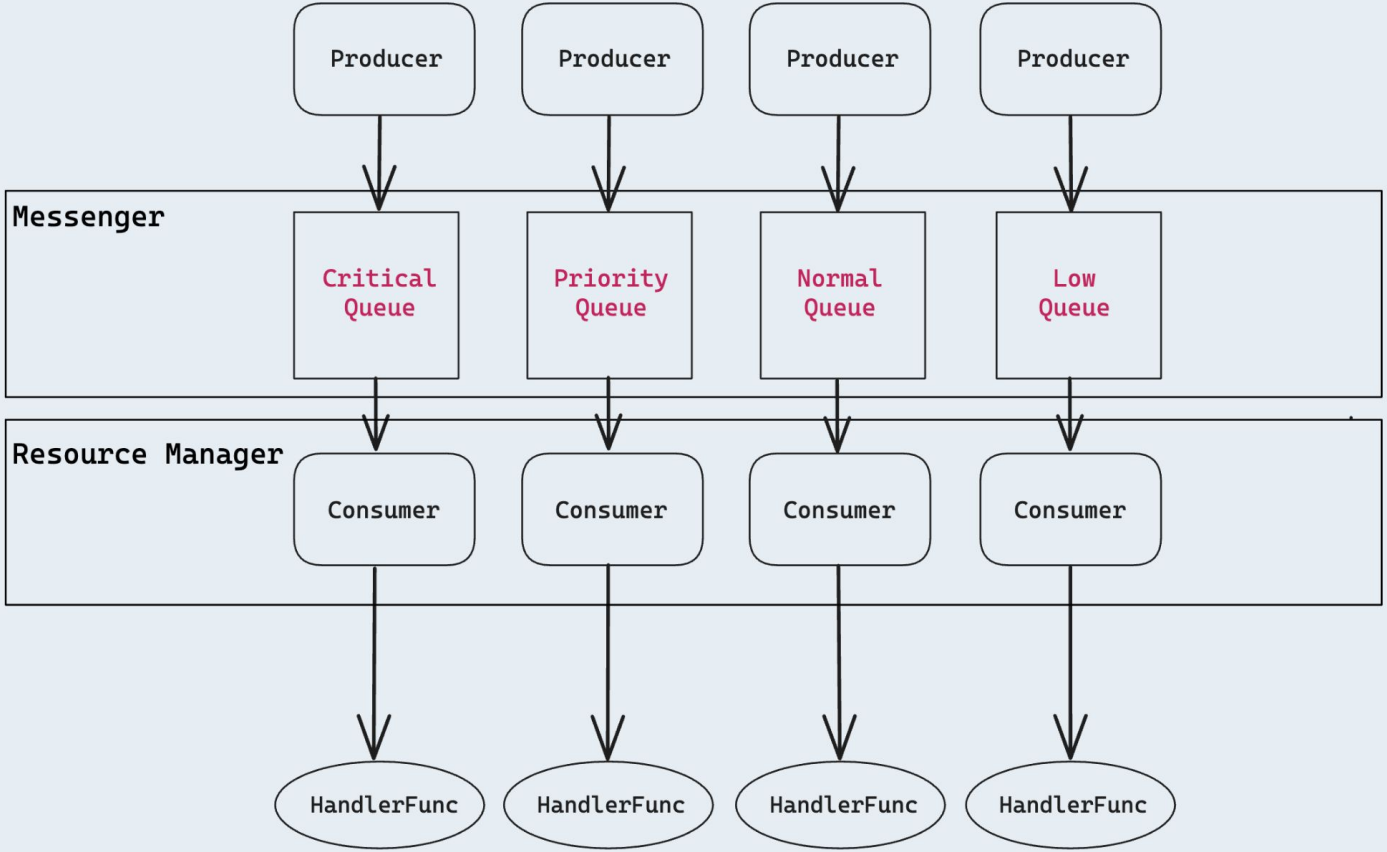
HandlerFunc

```go
func (r *ResourceManager) ProcessMessage(msg Message) {
    r.Add(msg.ResourceCost,  incrementCount: true)

    time.Sleep(300 * time.Millisecond)

    go func() {
        time.Sleep(800 * time.Millisecond)
        r.Remove(msg.ResourceCost)
    }()
}
```




Producer
Producer
Producer
Producer
Messenger
Critical Queue
Priority Queue
Normal Queue
Low Queue
Resource Manager
Consumer
Consumer
Consumer
Consumer
HandlerFunc
HandlerFunc
HandlerFunc
HandlerFunc

Let’s see it in action!

Base demo

- Show fluctuations of utilization
- Understand impact of load spikes without pressure valve
- Messages on all topics are negatively impacted by load spike

Output Format

Message Priority	Message Cost	Total Utilization
[Low]	cost 12	total: 55
[Critical]	cost 9	total: 55
[Normal]	cost 2	total: 55
[Priority]	cost 5	total: 55

```
[Critical]      cost 8          total: 59
[Critical]      cost 11         total: 52
[Low]           cost 11         total: 63
[Priority]      cost 4          total: 74
[Normal]        cost 5          total: 78
[Normal]        cost 4          total: 77
[Low]           cost 10         total: 77
[Critical]      cost 1          total: 77
[Priority]      cost 10         total: 77
68 messages processed... maximum utilization reached
```

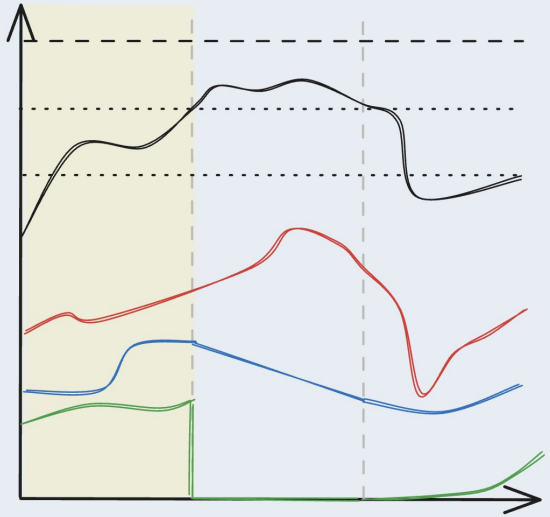
Recap

Base demo

- Program exited prematurely
- Maximum utilization reached
- All processes were negatively affected

Iteration 0.5: Alerting

- Emit alert on resource manager when utilization crosses upper threshold
- Receive alerts on consumer



Resource Manager

```
func (r *ResourceManager) Add(cost int) {
    r.Utilization += cost

    if r.Utilization >= 80 && !r.LastAlertVal {
        r.LastAlertVal = true
        for range cap(r.AlertChan) {
            r.AlertChan <- true
        }
    }
    if r.Utilization >= 100 {
        os.Exit( code: 1)
    }
    return
}
```

Resource Manager

```
func (r *ResourceManager) Remove(cost int) {
    r.Utilization -= cost
    if r.Utilization < 0 {
        r.Utilization = 0
    }
    if r.Utilization < 40 && r.LastAlertVal {
        r.LastAlertVal = false
        for range cap(r.AlertChan) {
            r.AlertChan <- false
        }
    }
    return
}
```

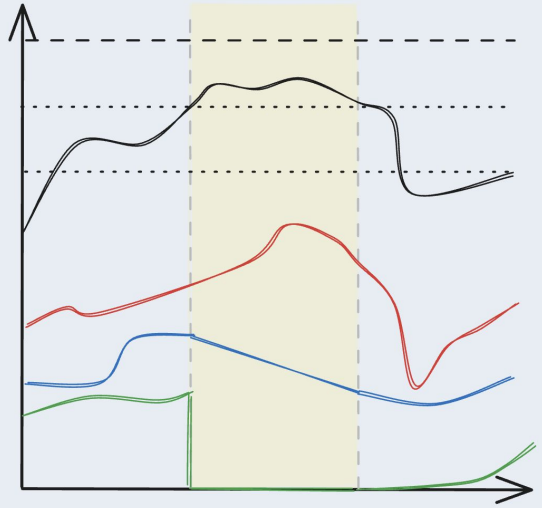
Code Changes - Consumer

Add consumer method to keep track of current alert value

```
func (c *Consumer) Watch(alertChan <-chan bool) {
    for {
        alert := <-alertChan
        // ...
        c.alertActive = alert
    }
}
```

Iteration 1: Basic Throttling

- Expose configuration options for per-queue throttling
- Update consumer logic to apply a fixed delay between messages



Code Changes - Queue

Expose queue configuration to enable throttling

```
queues := []gc.Queue{
    {
        Topic:   gc.Critical,
        Handler: rm.ProcessMessage,
    },
    {
        Topic:   gc.Priority,
        Handler: rm.ProcessMessage,
    },
    {
        Topic:             gc.Normal,
        Handler:           rm.ProcessMessage,
        ThrottlingEnabled: true,
    },
    {
        Topic:             gc.Low,
        Handler:           rm.ProcessMessage,
        ThrottlingEnabled: true,
    },
}
```

Code Changes - Consumer

On consume, apply delay if alert is active

```
func (c *Consumer) Consume(deliveryChan <-chan gc.Message, handler gc.HandlerFunc) {
    for {
        d, _ := <-deliveryChan

        if c.alertActive {
            time.Sleep(500 * time.Millisecond)
        }

        // invoke handlerFunc
        handler(d)
    }
}
```

Let’s see it in action!

Basic demo

- Automatic detection of resource usage beyond soft limit
- Critical/Priority messages should be processed as normal
- Normal/Low messages processed at a slower rate when total usage is high

Output Format

Message Priority	Message Cost	Total Utilization
[Low]	cost 12	total: 55
[Critical]	cost 9	total: 55
[Normal]	cost 2	total: 55
[Priority]	cost 5	total: 55

```
[Normal]        cost 9          total: 76
-------------------- slowing down --------------------
[Critical]      cost 4          total: 76
applying delay on Low
[Priority]      cost 0          total: 81
[Critical]      cost 4          total: 75
applying delay on Normal
[Priority]      cost 9          total: 55
[Critical]      cost 1          total: 55
[Low]           cost 0          total: 40
-------------------- speeding up --------------------
[Priority]      cost 0          total: 35
[Normal]        cost 4          total: 35
[Critical]      cost 0          total: 39
```

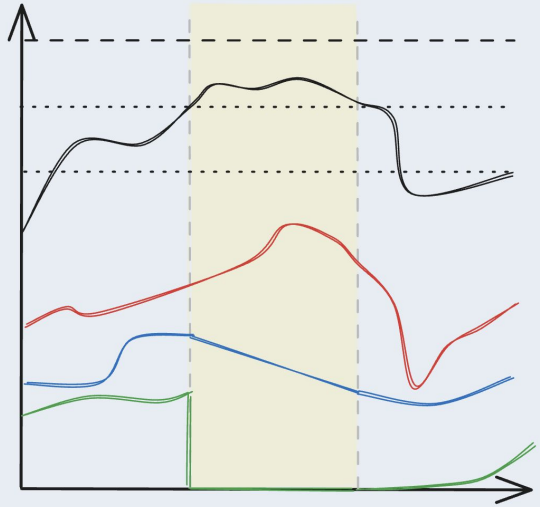
Recap

Basic throttling demo

- All messages were successfully processed
- Critical/Priority messages were unaffected by throttling
- Normal/Low messages were consumed at a slower rate during periods of high utilization

Iteration 2: Custom Throttling

- Abstract throttling implementations using Go interfaces
- Implement 3 different throttling strategies



Consumer shouldn’t care *how* throttling is done, just that it *is* getting done.

Throttler Interface

```
type Throttler interface {
    Apply()
}
```

Code Changes - Consumer

Instead of static delay...

```
func (c *Consumer) Consume(deliveryChan <-chan gc.Message, handler gc.HandlerFunc) {
    for {
        d, _ := <-deliveryChan

        if c.alertActive {
            time.Sleep(500 * time.Millisecond)
        }

        // invoke handlerFunc
        handler(d)
    }
}
```

Code Changes - Consumer

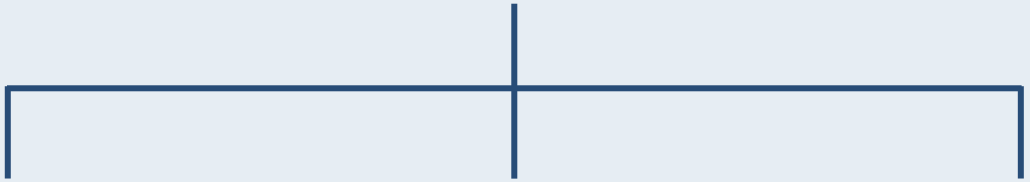
Use the interface method

```
func (c *Consumer) Consume(deliveryChan <-chan gc.Message, handler gc.HandlerFunc) {
    for {
        d, _ := <-deliveryChan

        if c.alertActive && c.throttle != nil {
            c.throttle.Apply()
        }

        // invoke handlerFunc
        handler(d)
    }
}
```

Throttling Strategies

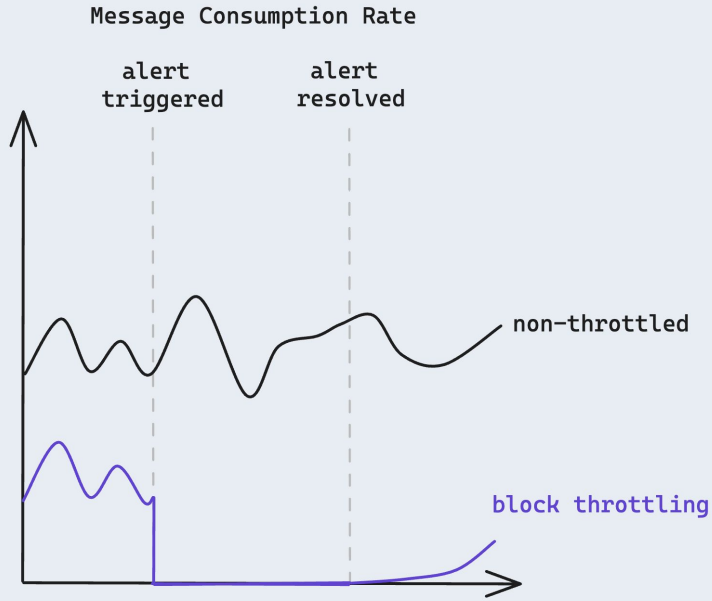


Block

Interval

Backoff

Block Throttling



Most **aggressive** strategy

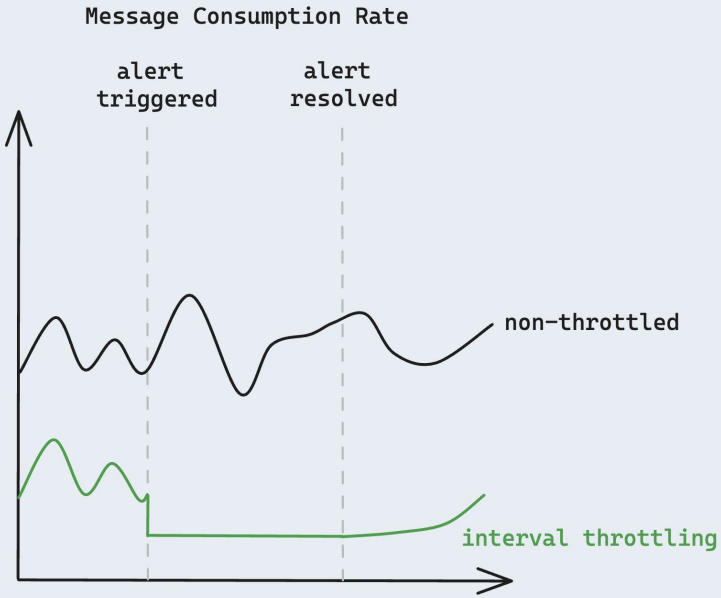
Alert Triggered

- **Stop** message consumption entirely

Alert Resolved

- Slowly **begin** consumption again

Interval Throttling



Most **straightforward** strategy

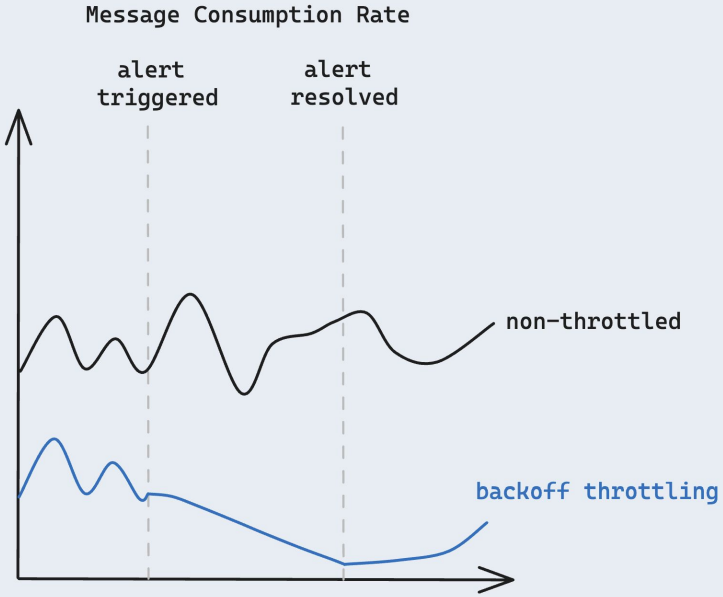
Alert Triggered

- Apply **static delay** between messages

Alert Resolved

- Slowly **speed up** consumption again

Backoff Throttling



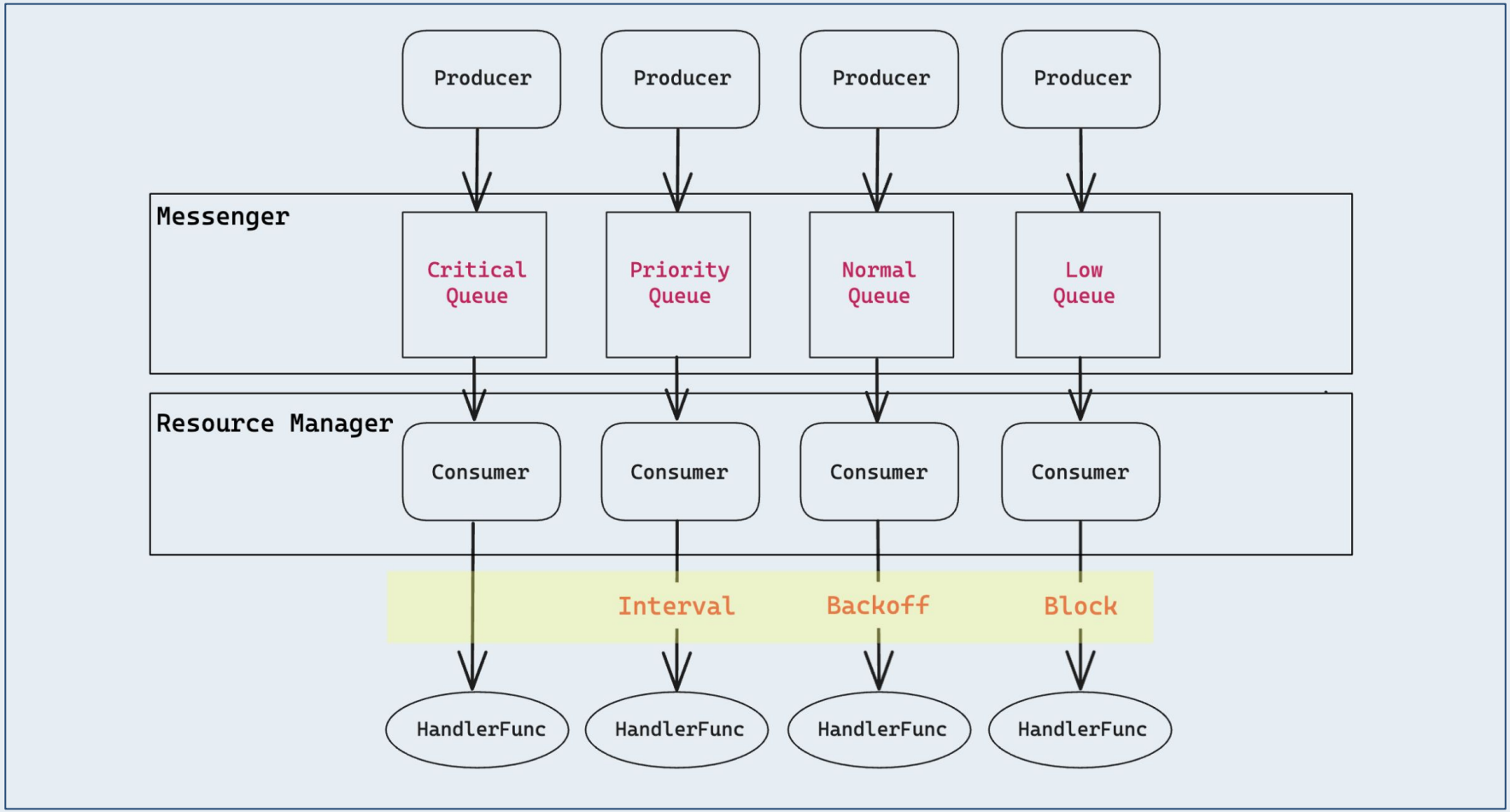
Most **responsive** strategy

Alert Triggered

- Start applying **initial delay**
- **Increase** delay at a fixed **interval**

Alert Resolved

- Slowly **speed up** consumption again


Producer
Producer
Producer
Producer
Messenger
Critical Queue
Priority Queue
Normal Queue
Low Queue
Resource Manager
Consumer
Consumer
Consumer
Consumer
Interval
Backoff
Block
HandlerFunc
HandlerFunc
HandlerFunc
HandlerFunc

Code Changes - Block Throttler

Stop message consumption completely

```
type block struct{}  3 usages

func (b *block) Apply() {
    //https://go.dev/ref/spec#Select_statements
    //
    //From go language spec: Since communication on nil channels can never proceed,
    //a select with only nil channels and no default case blocks forever.
    select {}
}
```

Code Changes - Interval Throttler

Apply static delay before consuming message

...sound familiar?

```
type interval struct {  4 usages
    throttleDelay time.Duration
}

func (i *interval) Apply() {
    time.Sleep(i.throttleDelay)
}
```

Implementing a Backoff throttler

Start by applying an **initial delay**.

After some **interval**, if alert has not been resolved, **increase** the delay (up to a **maximum**).

Adjustable Delay

- Delay updated while consumers are receiving messages
- Updated in separate goroutine
- Synchronization needed to prevent data races

Code Changes - Adjustable Delay

```
type adjustableDelay struct {  4 usages
    val time.Duration
    mu  sync.RWMutex
}

func (a *adjustableDelay) adjust(interval time.Duration, multiplier float64, maxDelay time.Duration) {
    t := time.NewTicker(interval)
    for {
        <-t.C
        a.val = time.Duration(float64(a.val.Milliseconds())*multiplier) * time.Millisecond
        if (multiplier > 1 && a.val.Milliseconds() >= maxDelay.Milliseconds()) ||
            (multiplier < 1 && a.val.Milliseconds() <= 50) {
            t.Stop()
            return
        }
    }
}
```

Code Changes - Backoff Throttler

Delay is applied just like the interval throttler

```
type backoff struct {  3 usages
    // user input
    initialDelay    time.Duration
    backoffInterval time.Duration
    multiplier      float64
    maxDelay        time.Duration

    // internal
    throttleStart int64
    delay         adjustableDelay
}

func (bo *backoff) Apply() {
    time.Sleep(bo.delay.getVal())
}
```

Invoking the Delay Adjustment

Consumer needs a way to kick off throttler delay adjustment when alert is received

Time to revisit our throttler interface!

Interface Re-definition

Introduce method to initialize throttler values

```
type Throttler interface {
    Apply()
    InitThrottle()
}
```

Code Changes - Throttlers

Implement new method for each throttler

```
func (b *block) InitThrottle() {}

func (i *interval) InitThrottle() {
    i.delay.val = i.throttleDelay
}

func (bo *backoff) InitThrottle() {
    bo.delay.val = bo.initialDelay
    go bo.delay.adjust(bo.backoffInterval, bo.multiplier, bo.maxDelay)
}
```

Code Changes - Consumer

In addition to updating the current alert value, initialize throttler if one is configured

```
func (c *Consumer) Watch(alertChan <-chan bool) {
    for {
        alert := <-alertChan
        c.mu.Lock()
        c.alertActive = alert
        if c.throttle != nil {
            c.throttle.InitThrottle()
        }
        c.mu.Unlock()
    }
}
```

Code Changes - Queues

```
{
    Topic:   gc.Priority,
    Handler: rm.ProcessMessage,
    BasicThrottler: &interval{
        throttleDelay: 100 * time.Millisecond,
    },
},
```

Let’s see it in action!

Custom throttling demo

- Observe impact of 3 separate throttling strategies

Output Format

Message Priority	Message Cost	Total Utilization
[Low]	cost 12	total: 55
[Critical]	cost 9	total: 55
[Normal]	cost 2	total: 55
[Priority]	cost 5	total: 55

```
-------------------- slowing down -----------------
apply interval 100ms
apply backoff 100ms
[Critical]      cost 4          total: 74
[Normal]        cost 0          total: 59
[Priority]      cost 6          total: 59
[Critical]      cost 4          total: 50
apply interval 100ms
apply backoff 400ms
[Priority]      cost 5          total: 61
[Critical]      cost 5          total: 66
apply interval 100ms
[Normal]        cost 4          total: 52
```

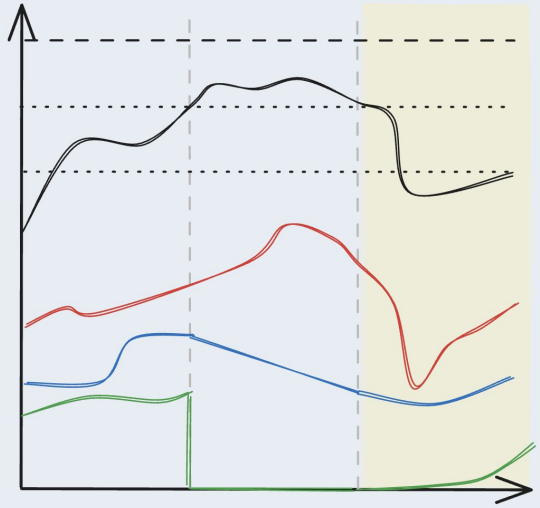
Recap

Custom throttling demo

- Critical messages unaffected
- Priority messages processed with fixed delay
- Normal messages processed with increasing delay
- Low messages not processed

Iteration 3:
Throttling with Recovery

- Implement recovery mechanism by leveraging context cancellation
- Putting it all together!



From a previous slide...

Implementing Backoff throttling

Start by applying an initial delay. After some interval, if alert has not been resolved, increase the delay (up to a maximum).

From a previous slide...

Implementing Backoff throttling

Start by applying an initial delay. After some interval, if alert has not been ***resolved***, ***increase*** the delay (up to a ***maximum***).

From a previous slide...

Implementing Backoff recovery

Start by applying an initial delay. After some interval, if alert has not been ***received***, ***decrease*** the delay (down to a ***minimum***).

Implementing Recovery

Recovery uses the same mechanism as the backoff throttler from the previous iteration.

Implementing Recovery

Similar to throttler initialization, we need to expose recovery initialization to the consumer

```
type Throttler interface {
    Apply(ctx context.Context)
    InitThrottle()
    InitRecovery()
}
```

Code Changes - Recovery (Backoff)

```
func (bo *Backoff) InitRecovery() {  4 usages
    bo.throttleStart = 0
    bo.recoveryStart = time.Now().Unix()
    bo.delay.val = min(bo.delay.getVal(), bo.maxDelay)

    bo.delay.adjust(bo.backoffInterval, 1/bo.multiplier,  maxDelay: 0)
}
```

Code Changes - Recovery (Interval)

```
func (i *Interval) InitRecovery() {  4 usages
    i.throttleStart = 0
    i.delay.val = i.throttleDelay
    i.recoveryStart = time.Now().Unix()

    i.delay.adjust(i.recoveryIncrement, i.recoveryMultiplier,  maxDelay: 0)
}
```

Code Changes - Recovery (Block)

```
func (b *Block) InitRecovery() {  4 usages
    b.throttleStart = 0
    b.recoveryStart = time.Now().Unix()
    b.delay.val = b.recoveryDelay

    b.delay.adjust(b.recoveryIncrement, b.recoveryMultiplier,  maxDelay: 0)
}
```

Code Changes - Recovery (Block)

What about the block throttler?

```
type block struct{}  3 usages

func (b *block) Apply() {
    //https://go.dev/ref/spec#Select_statements
    //
    //From go language spec: Since communication on nil channels can never proceed,
    //a select with only nil channels and no default case blocks forever.
    select {}
}
```

Code Changes - Block Throttler

```go
func (b *Block) Apply(ctx context.Context) {
    if b.IsThrottling() {
        // throttling mode, block until context canceled
        <-ctx.Done()
    } else if b.IsRecovering() {
        t := time.NewTimer(b.delay.getVal())
        defer t.Stop()
        select {
        case <-ctx.Done():
            return
        case <-t.C:
            return
        }
    }
}
```

Code Changes - Consumer

```
func (c *Consumer) Consume(deliveryChan <-chan gc.Message, alertChan <-chan bool, handler gc.HandlerFunc) {
    processed := make(chan bool)
    for {
        d, _ := <-deliveryChan
        ctx, cancel := context.WithCancel(context.Background())

        go func() {
            select {
            case <-processed:
                cancel()
                return
            case alert := <-alertChan:
                cancel()
                c.SetAlertVal(alert)
                <-processed
                return
            }
        }()
        ...
    }
}
```

Code Changes - Consumer

```
func (c *Consumer) Consume(deliveryChan <-chan gc.Message, alertChan <-chan bool, handler gc.HandlerFunc) {
    processed := make(chan bool)
    for {
        d, _ := <-deliveryChan
        ctx, cancel := context.WithCancel(context.Background())

        go func() {
            ...
        }()
        c.Throttle(ctx)
        // invoke handlerFunc
        handler(d)
        processed <- true
    }
}
```

Let’s see it in action!

Throttling with Recovery demo

- Throttling response unchanged from last iteration
- Throttlers switch to **recovery** strategy when alert is resolved
- Eventually normal operation is **restored**

Output Format

Message Priority	Message Cost	Total Utilization
[Low]	cost 12	total: 55
[Critical]	cost 9	total: 55
[Normal]	cost 2	total: 55
[Priority]	cost 5	total: 55

```
-------------------- slowing down ------------------
[Critical]      cost 3          total: 80
backoff waited 100ms
[Normal]        cost 7          total: 54
interval waited 150ms
[Priority]      cost 1          total: 47
[Critical]      cost 5          total: 48
[Critical]      cost 3          total: 61
interval waited 150ms
[Priority]      cost 10         total: 64
backoff waited 200ms
[Normal]        cost 1          total: 68
[Critical]      cost 5          total: 53
```

```
interval waited 150ms
[Priority]      cost 2          total: 57
block waited 250ms
[Low]           cost 0          total: 53
[Critical]      cost 4          total: 53
backoff waited 350ms
[Normal]        cost 1          total: 51
interval waited 75ms
[Priority]      cost 2          total: 48
[Critical]      cost 6          total: 50
block waited 125ms
[Low]           cost 10         total: 49
backoff waited 87ms
```

Recap

Throttling with Recovery demo

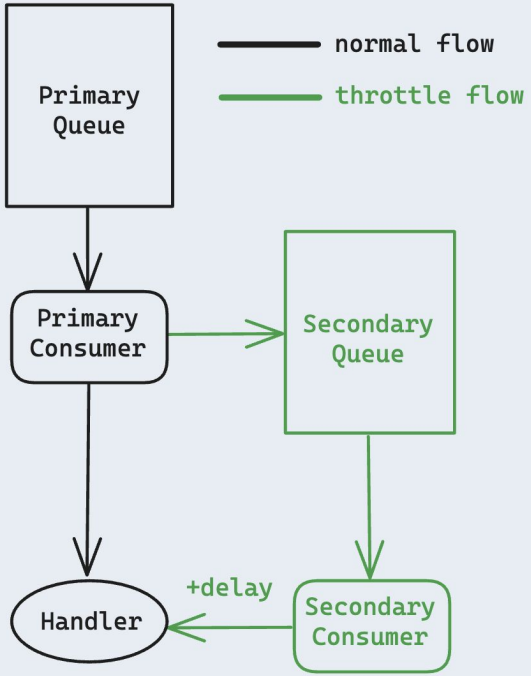
- Same throttling we saw from previous demo
- On alert resolution, delay intervals were shortened until normal operation was restored

Adaptations for Real-World Systems

Challenge: **Throttling Granularity**

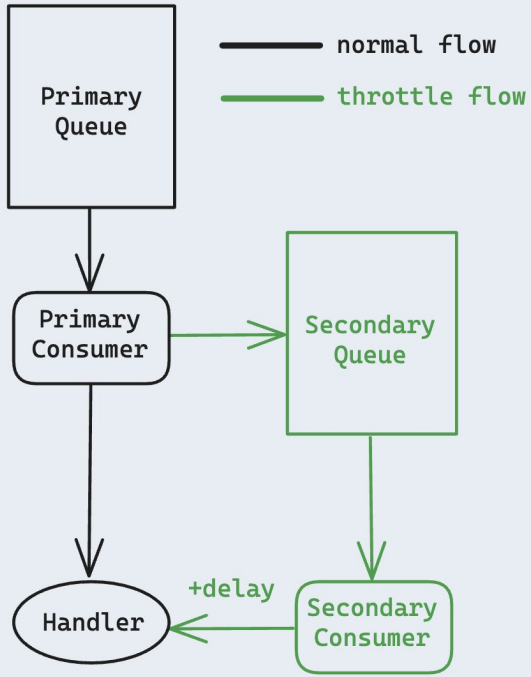
- Message routing determined by combination of **topic** and **routing key**
- Queues stored messages from **multiple** routing keys
  - Not all messages on a queue should be throttled
  - Can have multiple throttlers involved per topic

Adaptation: **Throttling Granularity**



Introduction of **secondary** queue and consumer

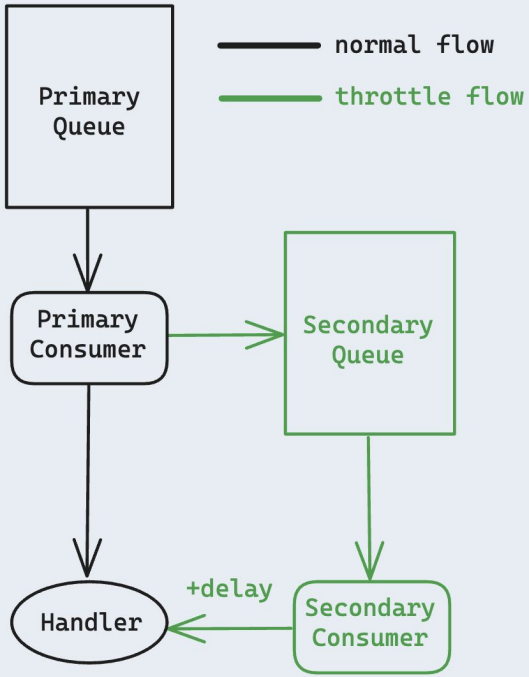
Adaptation: **Throttling Granularity**



When alert is **triggered**

- **Primary** consumer publishes throttled messages to secondary queue while delivering non-throttled messages
- **Secondary** consumer applies throttler delay before delivering message to the same handler

Adaptation: **Throttling Granularity**



When alert is **resolved**

- **Primary** consumer delivers all messages directly to handler
- **Secondary** consumer applies recovery delay before delivering message

Bonus Advantage

Before throttling:

- HPA configured to scale up if queue length exceeded a certain length
- Additional pods added to help process messages faster

Bonus Advantage

Before throttling:

- HPA configured to scale up if queue length exceeded a certain length
- Additional pods added to help process messages faster
- Increased load occasionally exhausted database resources
- Led to even more backed up requests, which made HPA scale up even more

Bonus Advantage

After throttling:

- Separated HPA configurations for primary vs. secondary queues
- During load spikes, minimal impact on critical processes
- No manual intervention required

Challenge: **Usability**

- Ideally minimize developer overhead
- Package pressure relief valve functionality and provide easy developer access across all 50+ backend services
- Backwards compatibility with current message queue operation

Adaptation: **Usability**

- Packaged all features together in an internal library available within the cluster
- No changes required for services that do not need it
- Able to fine-tune pressure relief valve based on developer discretion, as opposed to using a managed solution

Challenge: **Alerting Framework**

- Today's demos
  - Logic for utilization calculation was **one-dimensional**
  - Everything was **co-located** in one repository
- Reality
  - Metrics are often exposed through **numerous** sources in **different** formats
  - Having every service monitor these metrics creates unnecessary **overhead** and **complexity**

Adaptation: **Alerting Framework**

- Built a custom monitoring service that ran in the cluster
- Service periodically performs health checks and compares usage metrics of critical resources (ex. database CPU)
- Metric thresholds for each critical resource are set through configmaps
- Monitoring service publishes messages on a specialized “alert” topic anytime an alert is triggered or resolved
- Any backend service can subscribe to the topic to receive alerts

Summary

- Protect critical components of the platform from reaching resource limits in situations where scaling is not feasible
- Build a pressure relief valve that automatically detects when critical components are at risk

Summary

- Built 3 iterations of pressure relief valve
  - Basic Throttling
  - Custom Throttling
  - Throttling with Recovery
- Discussed adaptations of bringing this pressure relief valve to a production codebase
- Worked with a handful of tools from Go's standard library
  - Channels, tickers, timers, context

Acknowledgements

- Arman Masoumi
- Lucas Rosa
- Flybits backend team
- Amy Wu
- GopherCon organizers

THANK YOU!

@exgao/GopherCon2024