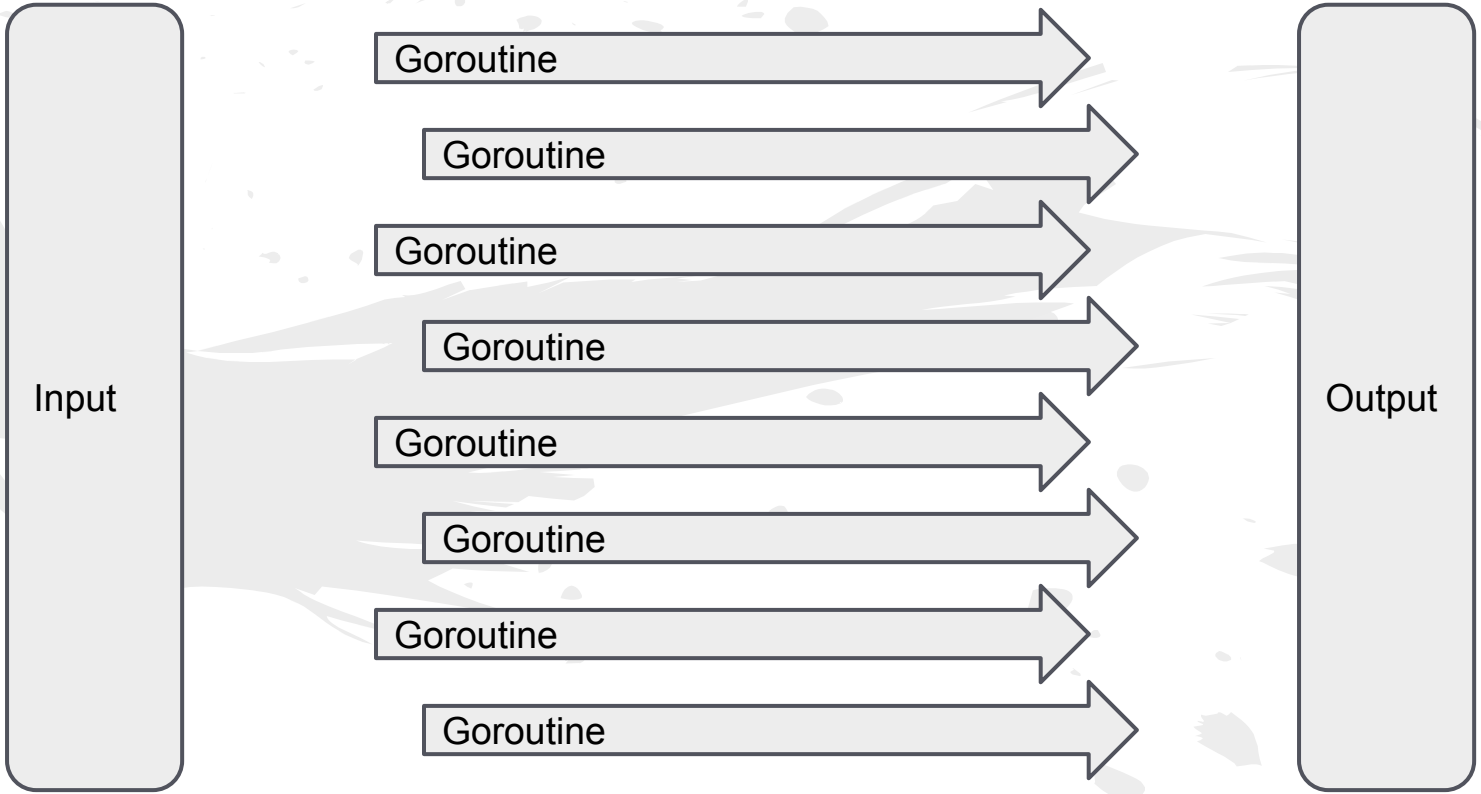
Complex Concurrency Patterns in Go

Evan Huus - Shopify, Inc.
@eapache


Input
Goroutine
Goroutine
Goroutine
Goroutine
Goroutine
Goroutine
Goroutine
Goroutine
Output

In Theory

In Practice

In Practice

Yes, Go makes concurrency easier.

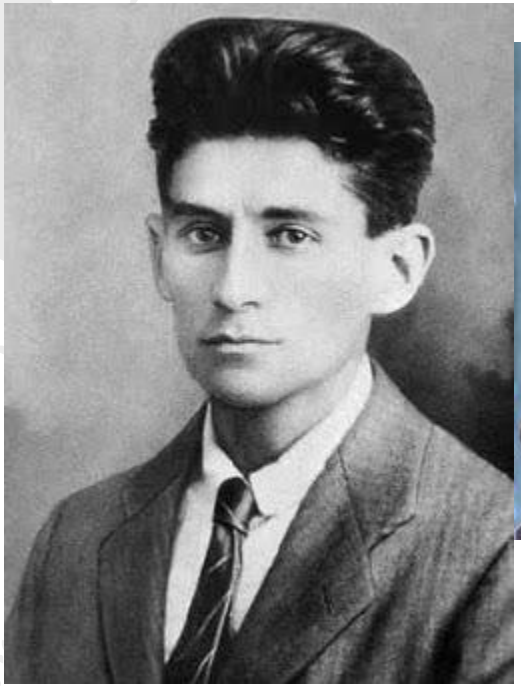
Yes, Go makes concurrency easier.

It’s still really hard.

Overview

- A little bit of context
- A lot of case study

Literary Giants



Kafka (https://kafka.apache.org/)

- Java-based Apache project for distributed publish-subscribe messaging.
- Messages are grouped into topics, topics are subdivided into partitions, and partitions are led or replicated by brokers.
- Clients are **thick**.

Sarama.go (https://github.com/Shopify/sarama)

- Native Golang client for producing and consuming messages via Kafka.
- Implements wire protocol, producer and consumer.
- First version was a proof-of-concept, kept it simple, but...

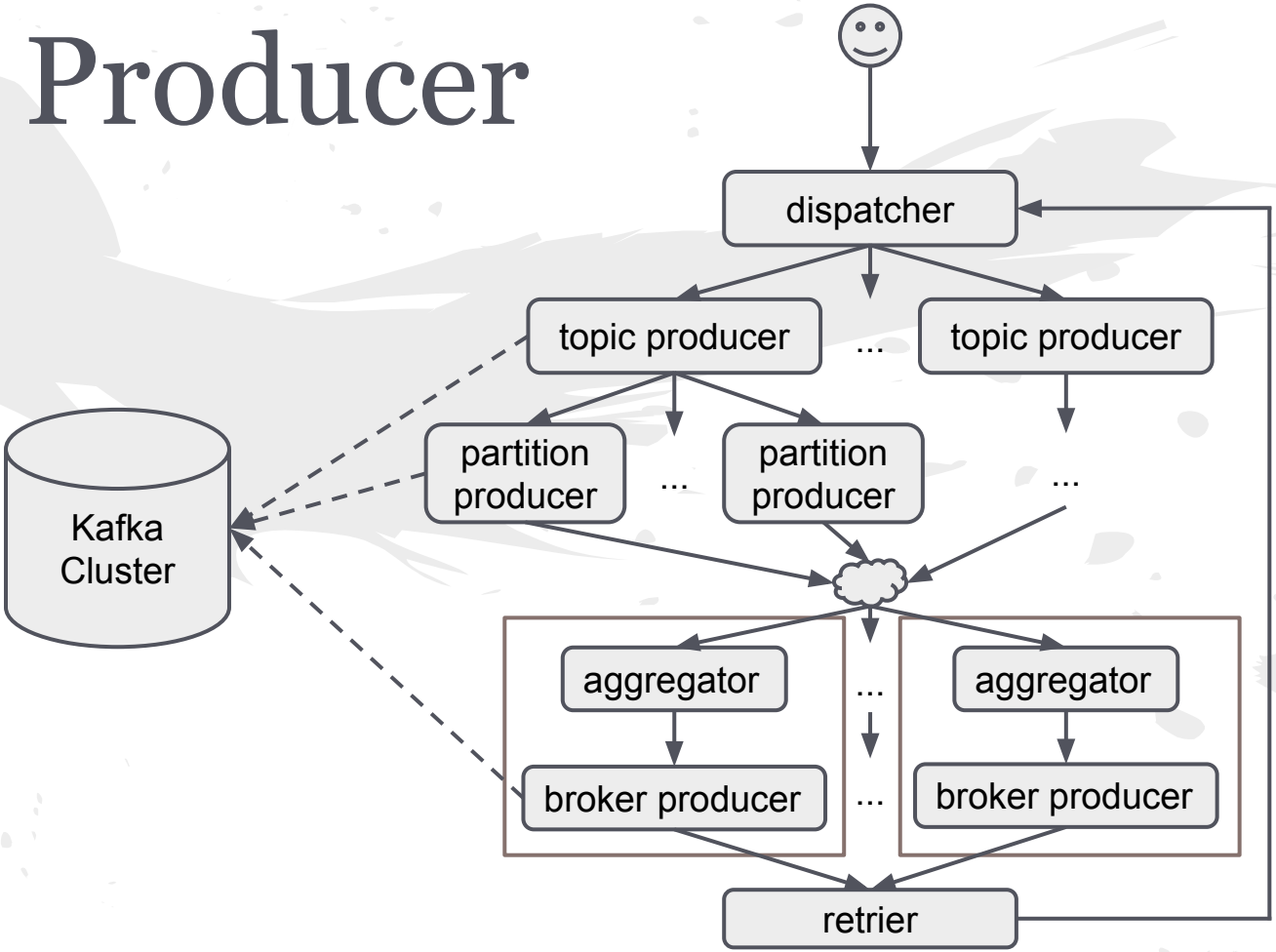
Knuth

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. ***Yet we should not pass up our opportunities in that critical 3%.***"

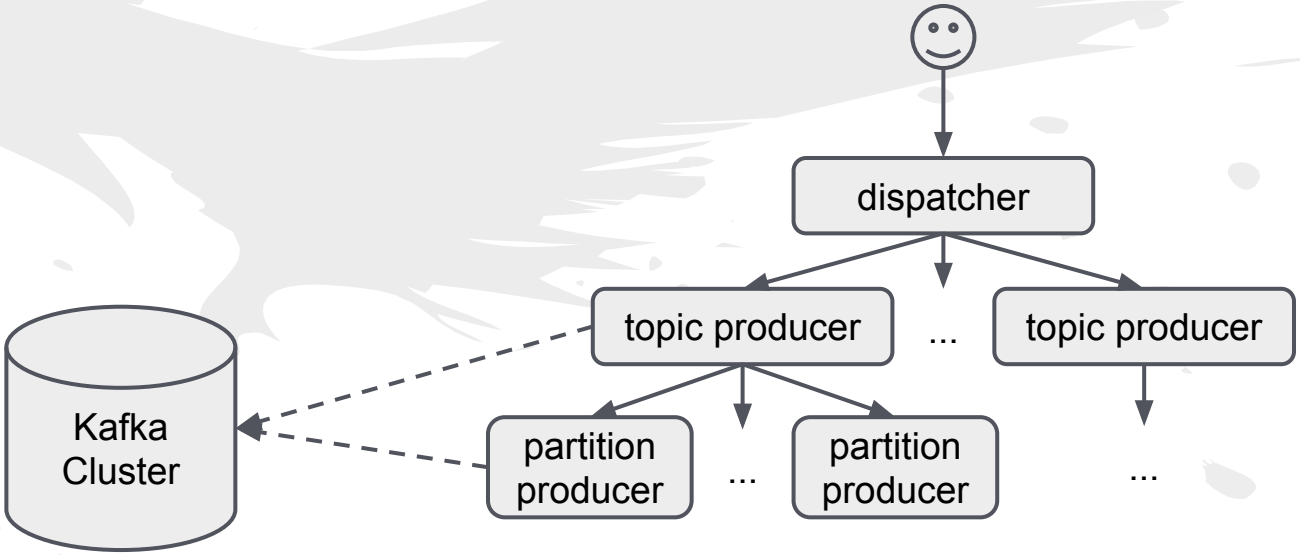
Second Draft Requirements

- Fast
- Configurable
- Resilient
- Correct

Producer


dispatcher
topic producer
...
topic producer
partition producer
...
partition producer
...
Kafka Cluster
aggregator
...
aggregator
broker producer
...
broker producer
retrier

Resiliency and Isolation



Resiliency and Isolation

- fan-out (dispatcher)

```
handlers := make(map[string]chan<- *Message)

for msg := range input {
    handler := handlers[msg.Topic]

    if handler == nil {
        handler = p.newTopicProducer(msg.Topic)
        handlers[msg.Topic] = handler
    }

    handler <- msg
}
```

Resiliency and Isolation

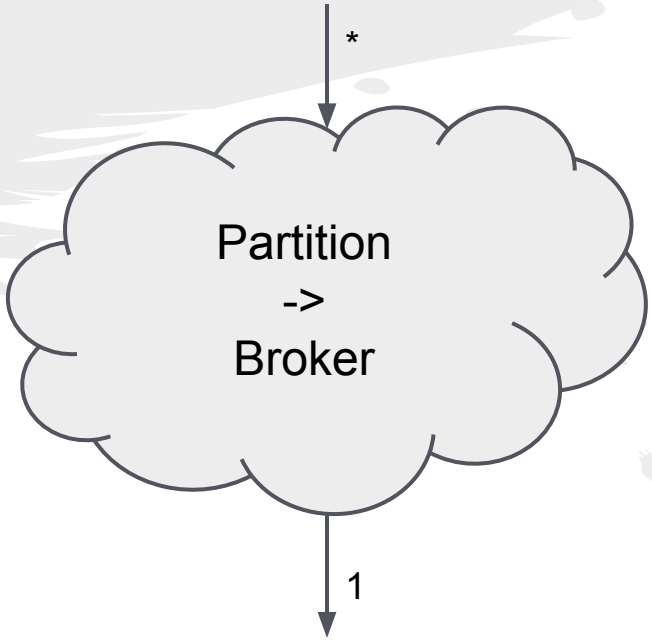
- circuit-breakers (https://github.com/eapache/go-resiliency)

```
partitions, err = client.Partitions(msg.Topic)
```

versus

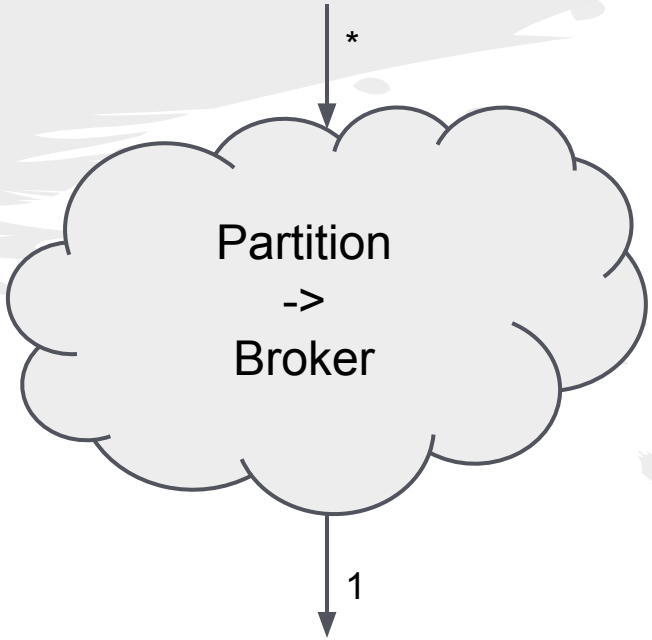
```
breaker := breaker.New(3, 1, 10*time.Second)

var partitions []int32
err := breaker.Run(func() (err error) {
    partitions, err = client.Partitions(msg.Topic)
    return
})
```

Dynamic Multiplexing



Dynamic Multiplexing

- global, locked, reference-counted map



Dynamic Multiplexing

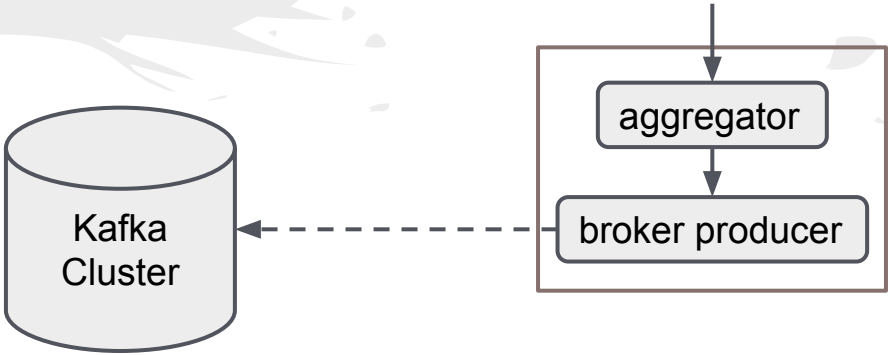
- acquire-broker

```
p.brokerLock.Lock()
defer p.brokerLock.Unlock()

bp := p.brokers[broker]
if bp == nil {
    bp = p.newBP(broker)
    p.brokers[broker] = bp
}
p.brokers[broker].refs++

return bp
```

- release-broker

```
p.brokerLock.Lock()
defer p.brokerLock.Unlock()

p.refs[bp]--
if p.refs[bp] == 0 {
    close(bp.input)
    delete(p.brokers, bp.broker)
}
```

Batching and I/O



Batching and I/O

- aggregator

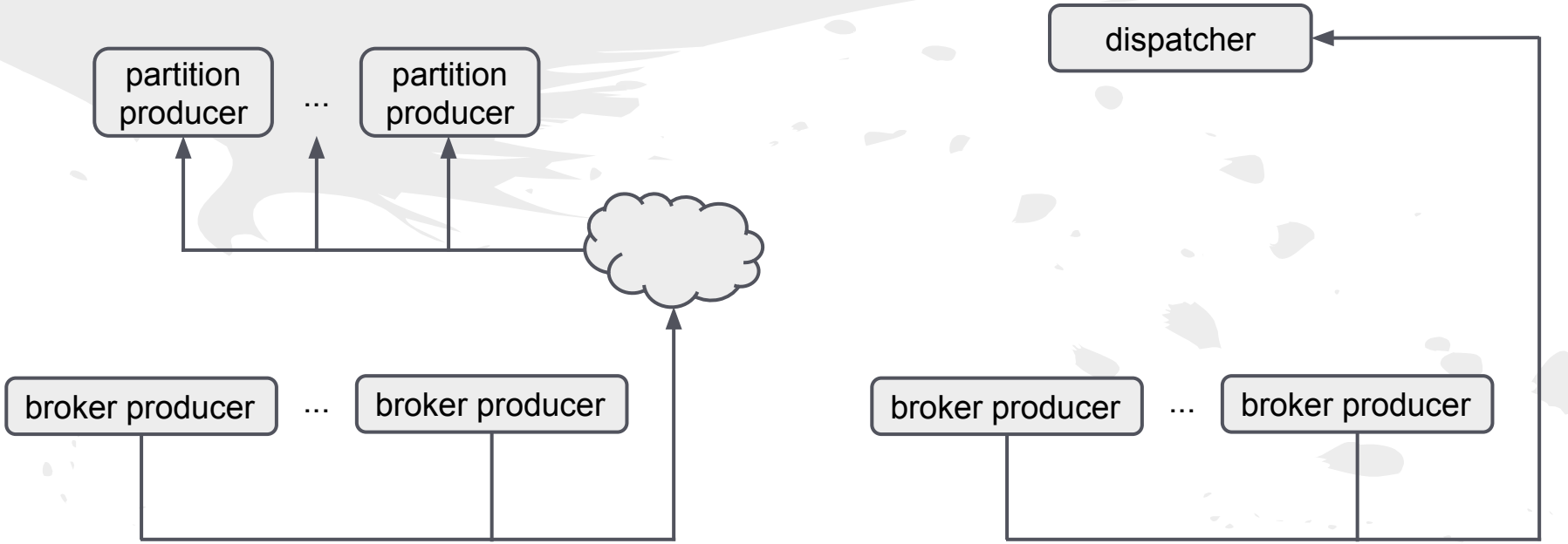
```
for {
    select {
    case msg := <-input:
        req.addMessage(msg)
        if req.full() { output = realOutput }
    case <-timer:
        output = realOutput
    case output <- req:
        output = nil
        req = new(Request)
    }
}
```

Batching and I/O

- buffer-producer

```
for request := range input {
    response, err := broker.Produce(request)

    switch err.(type) {
        // ...
    }

    p.handleResponse(response)
}
```

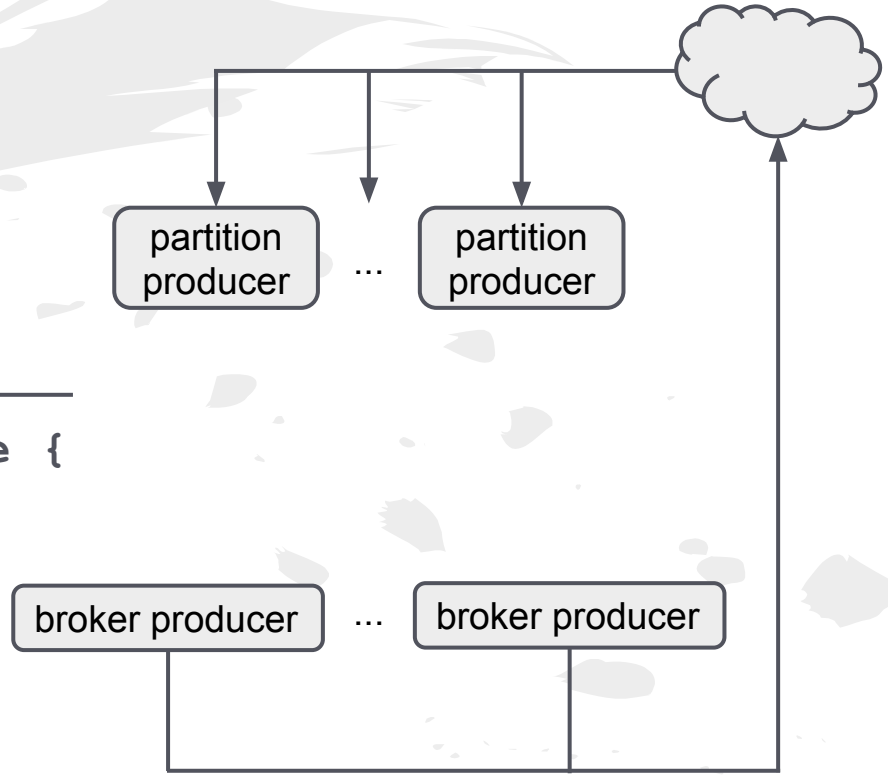
Try, Try Again



Try, Try Again (option #1)

```
for {
    select {
    case msg := <-input:
        buf = append(buf, msg)
    case ack := <-acks:
        // ...
}
```

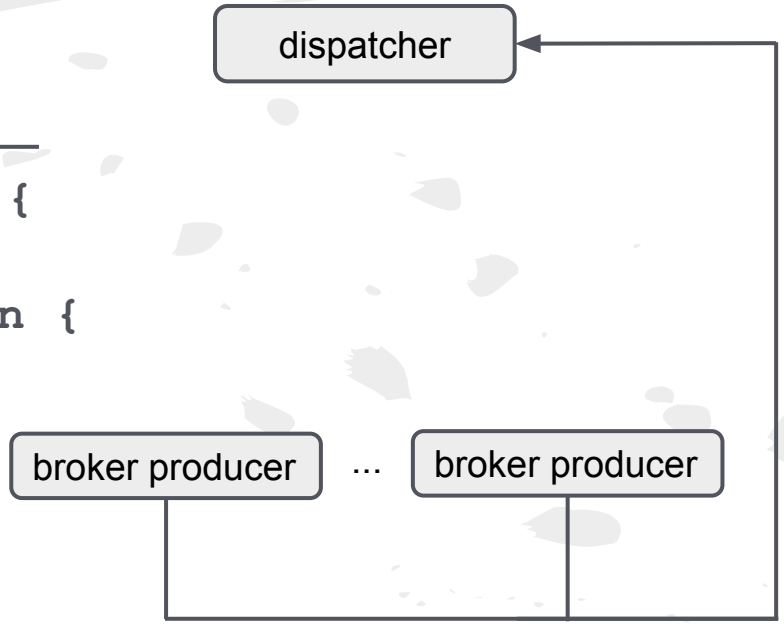
```
for partition := range response {
    if partition.success {
        partition.sendAck()
    } else {
        partition.sendNack()
    }
}
```



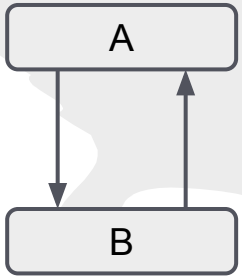
Try, Try Again (option #2)

```
if msg.retries > 0 {
    // ...
}
```

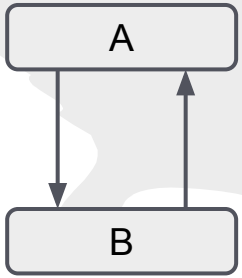
```
for partition := range response {
    if !partition.success {
        for msg := range partition {
            msg.retries++
            dispatcher <- msg
        }
    }
}
```



Try, Try Again (continued)

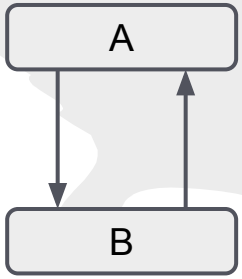


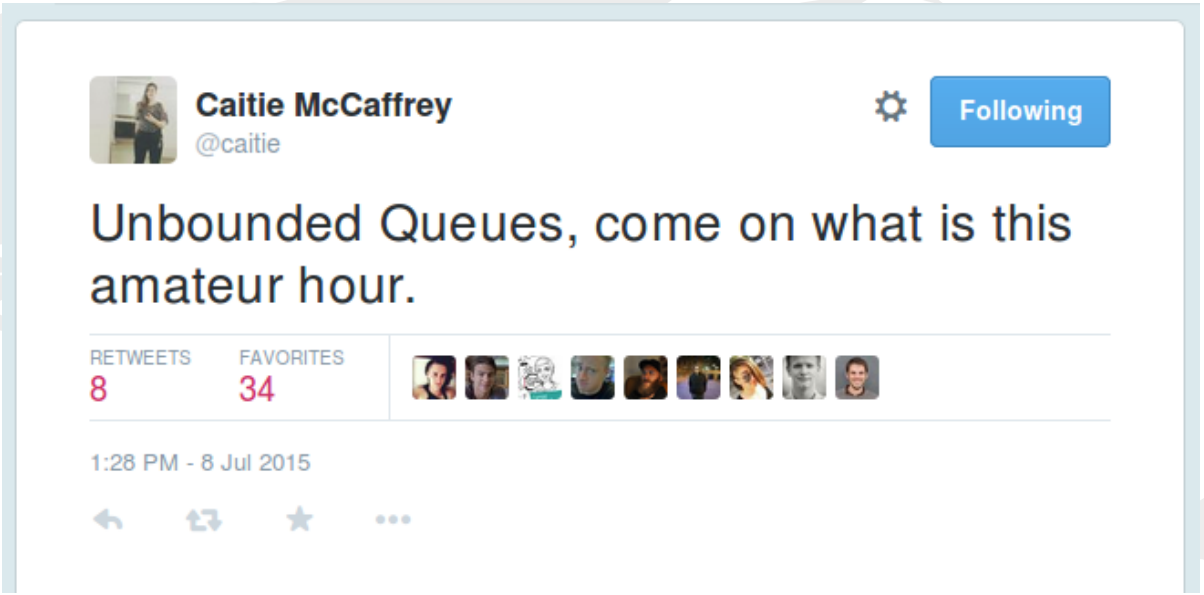
Try, Try Again (continued)




```
select {
    case msg := <-input:
        // ...
    case output <- msg:
        // ...
}
```

Try, Try Again (continued)

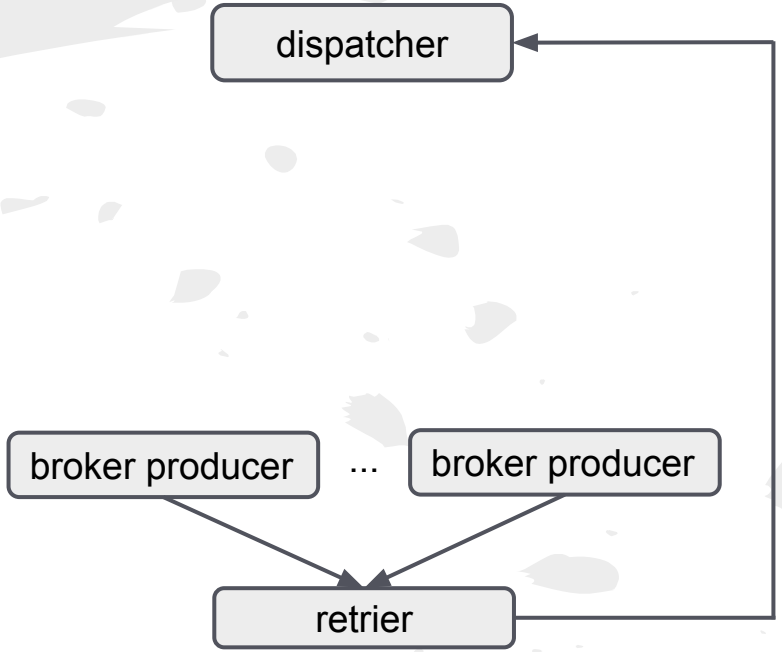




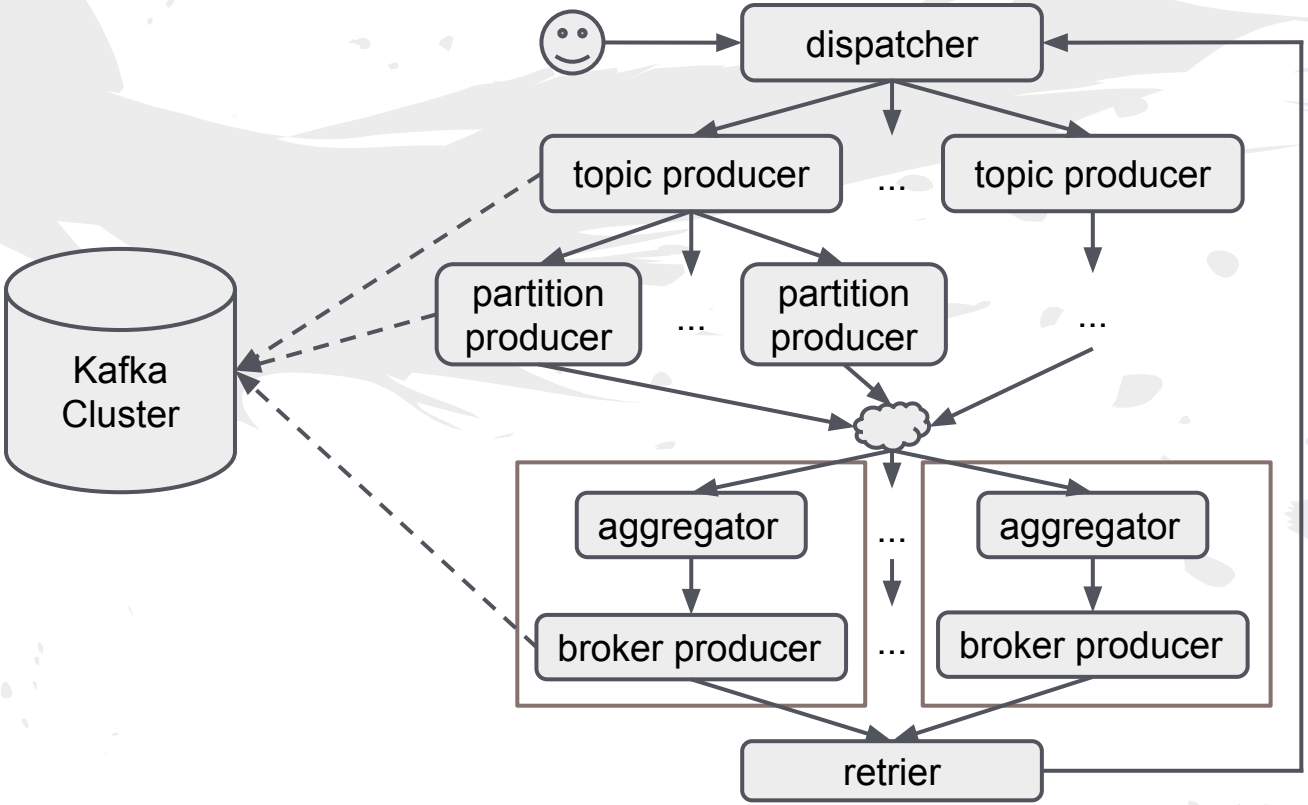
Try, Try Again (continued)

- if it's stupid, but it work (https://github.com/eapache/channels/)

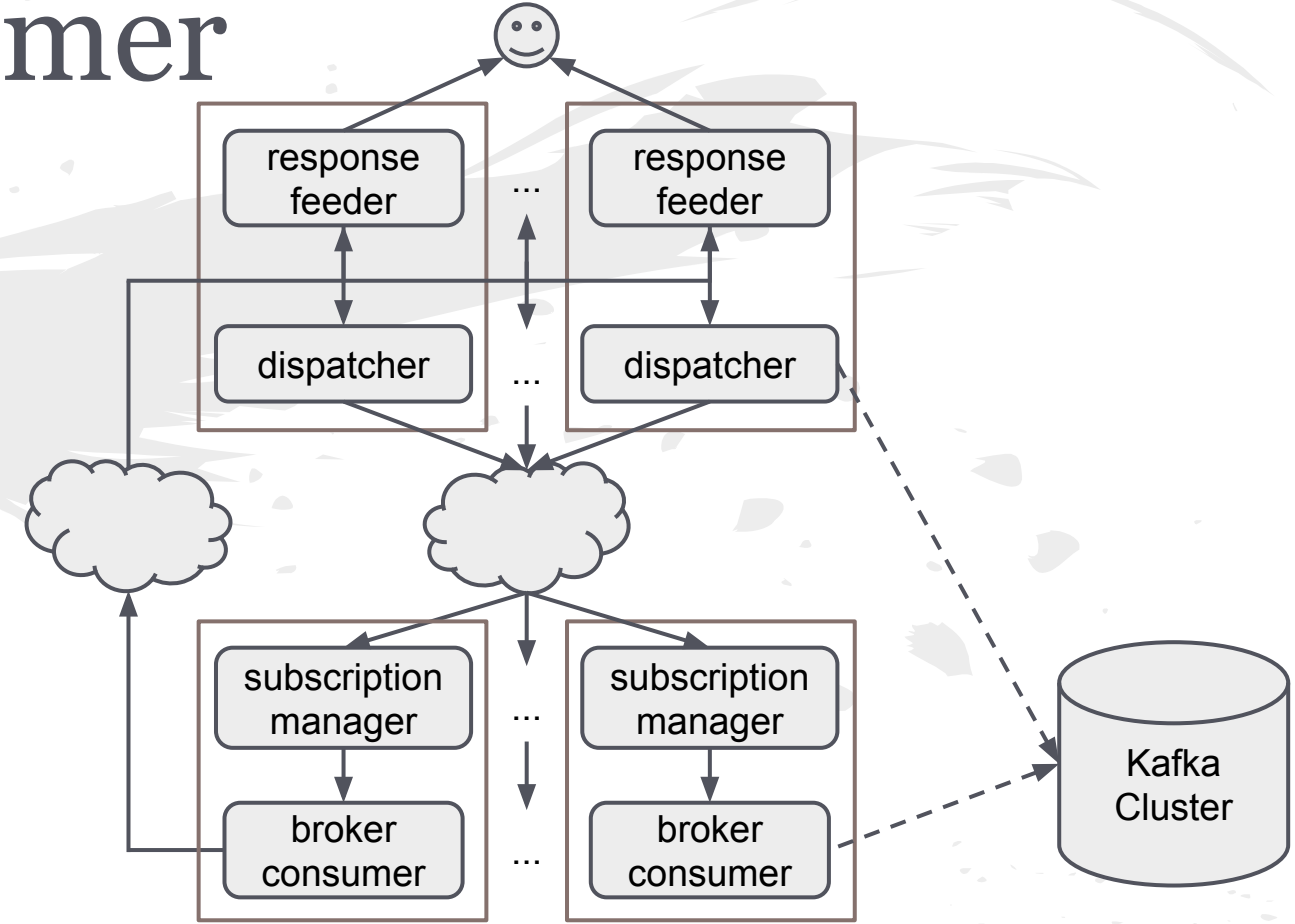
```
for {
    if len(buf) == 0 {
        msg = <-p.retries
    } else {
        select {
        case msg = <-p.retries:
        case p.input <- buf[0]:
            buf = buf[1:]
            continue
        }
    }
    buf = append(buf, msg)
}
```



Putting it all together



Consumer


response feeder
response feeder
dispatcher
dispatcher
subscription manager
subscription manager
broker consumer
broker consumer
Kafka Cluster

Structure Your Goroutines

Anonymous -> Named -> Structured

Structure Your Goroutines

Anonymous

```
go func() {
    // ...
}()
```

Named

```
func foo() {
    // ...
}

go foo()
```

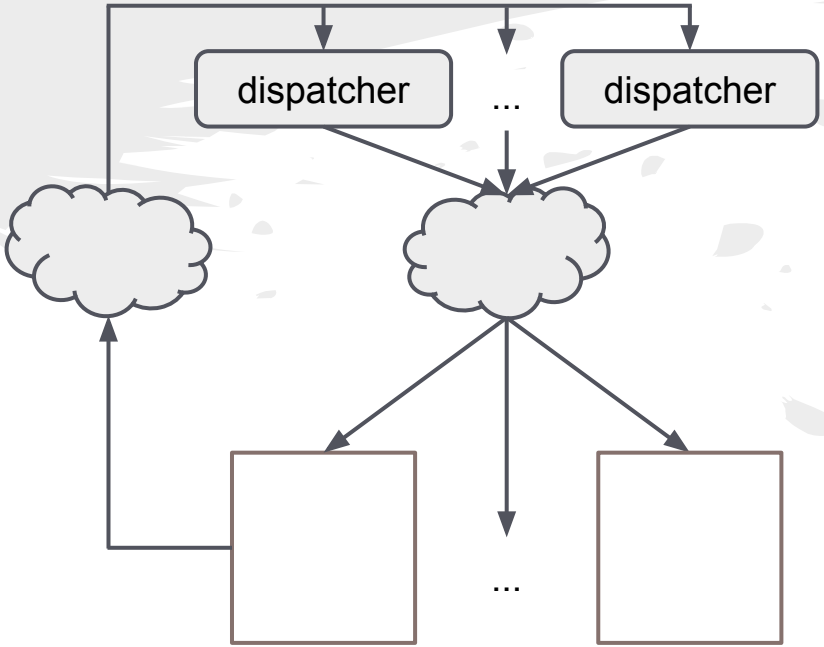
Structure Your Goroutines

Structured

```
type foo struct {
    // ...
}

func (f *foo) run() {
    // ...
}
```

```
func newFoo(...) {
    foo := &foo{
        // ...
    }
    go foo.run()
}
```

Ownership Semantics



Ownership Semantics

- dispatcher

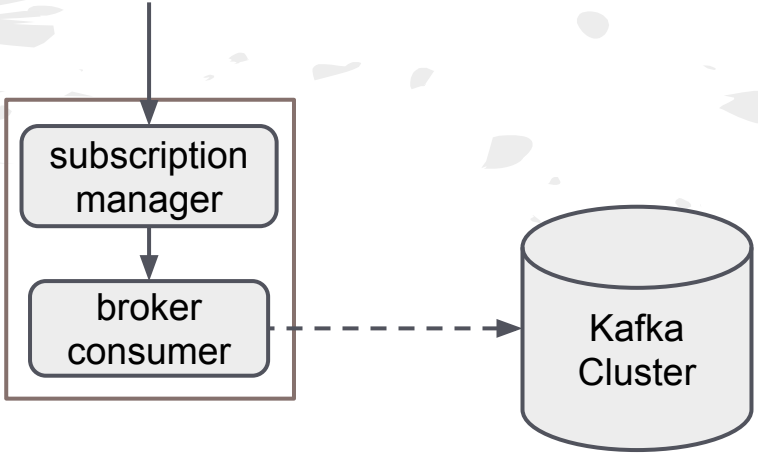
```
trigger := make(chan struct{}, 1)

for _ = range trigger {
    broker, err := findNewLeader()
    if err != nil {
        time.Sleep(...)
        trigger <- struct{}{}
    } else {
        broker.subscribe <- partition
    }
}
```

Ownership Semantics

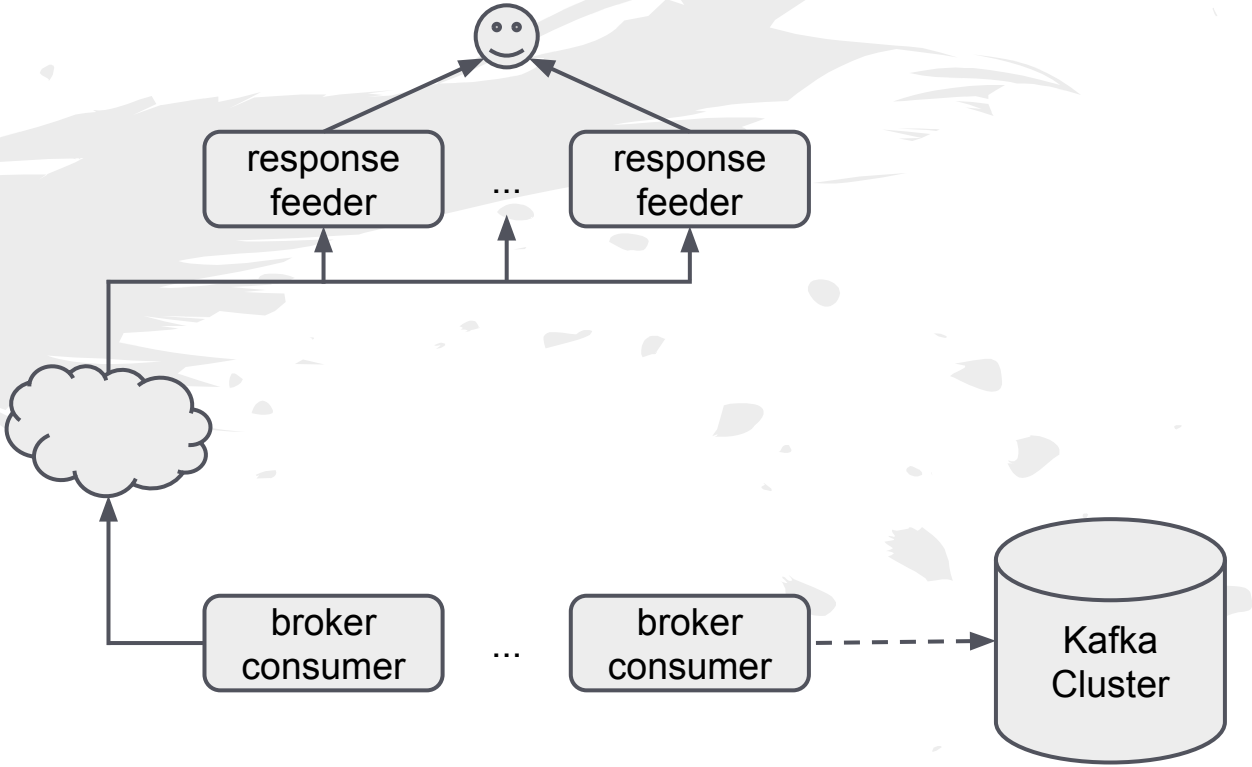
- broker

```
for partition, messages, err := range response {
    if err != nil {
        delete(subscriptions, partition)
        partition.trigger <- struct{}{}
        continue
    }

    sendToUser(messages)
}
```

Isolating I/O (redux)



Feeding the User



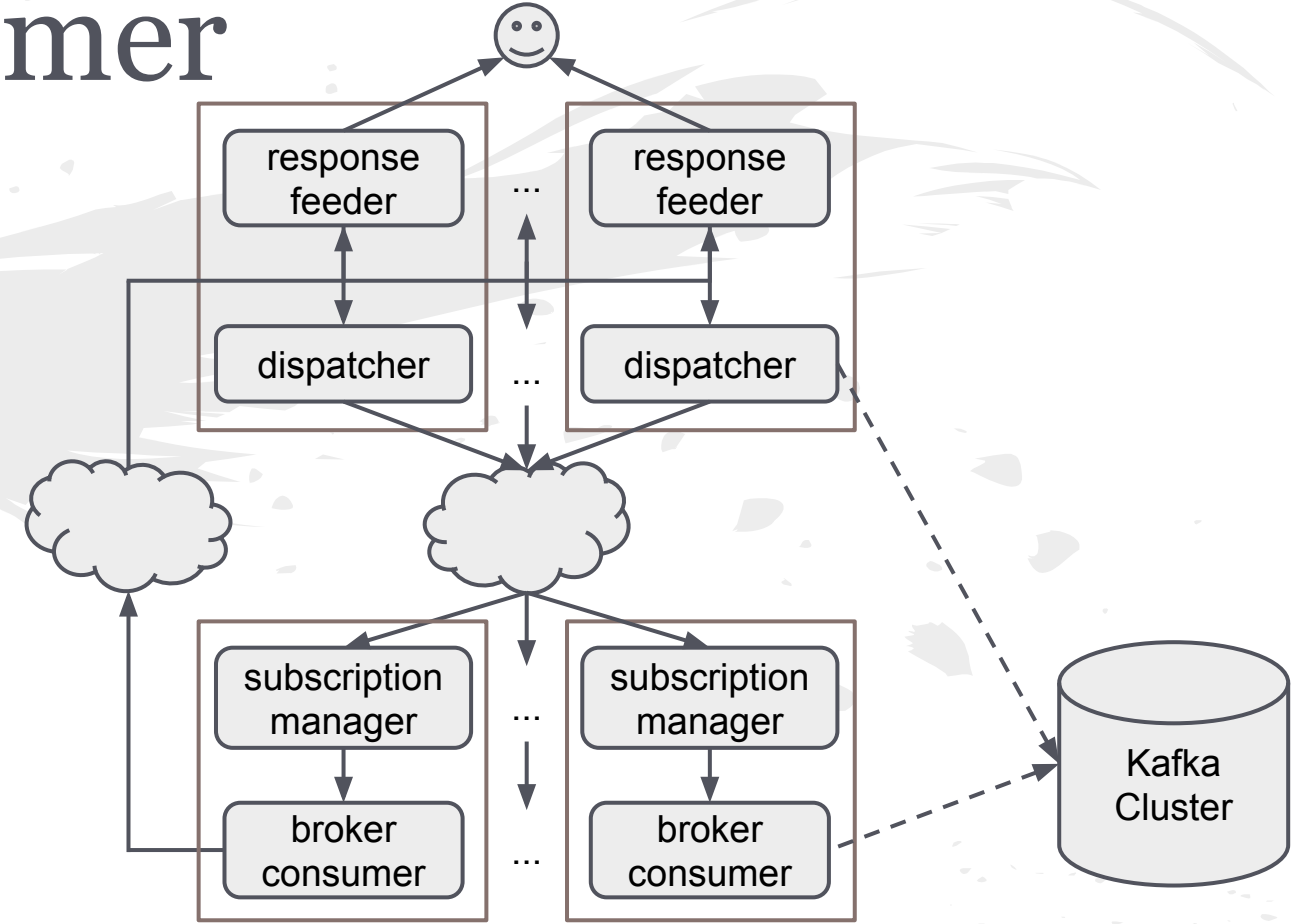
Feeding the User (response-feeder)

```
for messages := range input {
    for msg := range messages {
        select {
        case output <- msg:
        case <-time.After(timeout):
            delete(broker.subscriptions, partition)
            broker.acks.Done()
            // feed remaining messages
            broker.subscribe <- partition
            continue outerLoop
        }
    }
    broker.acks.Done()
}
```

Feeding the User (broker-consumer)

```
broker.acks.Add(len(subscriptions))

for sub := range subscriptions {
    sub.feeder <- response.messages[sub]
}

broker.acks.Wait()
```

Consumer


response feeder
response feeder
dispatcher
dispatcher
subscription manager
subscription manager
broker consumer
broker consumer
Kafka Cluster

Lessons Learned

1. Channels are primitives.
2. Structure your goroutines.
3. Don’t trust the network **or** the user.
4. Infinite buffers smell.
5. Don’t be afraid of locks and “anti-go” tricks.

Credits

- Photo of Franz Kafka: public domain (via Wikimedia Commons).
- Photo of José Saramago: CC-BY 2.0 (from the website of the Presidencia de la Nación Argentina, via Wikimedia Commons)
- Photo of Donald Knuth: CC-BY-SA 2.5 (by Jacob Appelbaum, via Wikimedia Commons)
- Tweet from @caitie: used with permission.

Questions?

@eapache
eapache@gmail.com
https://eapache.github.io
(feedback: https://joind.in/talk/view/14954)

Thanks!

@eapache
eapache@gmail.com
https://eapache.github.io
(feedback: https://joind.in/talk/view/14954)