Durable Executions in the Face of (some) Failures

Andrew Fitz Gibbon (aka "Fitz")
Staff Developer Advocate
@ Temporal Technologies

fitz@temporal.io
@fitzface

The 8 Fallacies

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

Simple Program

```go
func OrderItem(o OrderRequest) error {
    c, err := api_client.New()
    defer c.Close()

    order, err := c.InitOrder(o)

    status, err := c.FulfillOrder(order)

    db, err := archival.NewClient()
    defer db.Close()

    err = db.Persist(order, status)
}
```

Simple Program

```
func OrderItem(o OrderRequest) error {
    c, err := api_client.New()
    defer c.Close()

    order, err := c.InitOrder(o)

    status, err := c.FulfillOrder(order)

    db, err := archival.NewClient()
    defer db.Close()

    err = db.Persist(order, status)
}
```

Initiate

Simple Program

```
func OrderItem(o OrderRequest) error {
    c, err := api_client.New()
    defer c.Close()

    order, err := c.InitOrder(o)
    status, err := c.FulfillOrder(order)

    db, err := archival.NewClient()
    defer db.Close()

    err = db.Persist(order, status)
}
```

Initiate

Fulfill

Simple Program

```
func OrderItem(o OrderRequest) error {
    c, err := api_client.New()
    defer c.Close()

    order, err := c.InitOrder(o)
    status, err := c.FulfillOrder(order)

    db, err := archival.NewClient()
    defer db.Close()

    err = db.Persist(order, status)
}
```

Initiate

Fulfill

Archive/Log

Unreliable Services

```go
func OrderItem(o OrderRequest) error {
    c, err := api_client.New()
    defer c.Close()

    order, err := c.InitOrder(o)

    status, err := c.FulfillOrder(order)

    db, err := archival.NewClient()
    defer db.Close()

    err = db.Persist(order, status)
}
```

Unreliable Services

```
func OrderItem(o OrderRequest) error {
    c, err := api_client.New()
    defer c.Close()

    order, err := c.InitOrder(o)

    status, err := c.FulfillOrder(order)

    db, err := archival.NewClient()
    defer db.Close()

    err = db.Persist(order, status)
}
```

Unreliable Services

```
func OrderItem(o OrderRequest) error {
    c, err := api_client.New()
    defer c.Close()

    order, err := c.InitOrder(o)

    status, err := c.FulfillOrder(order)

    db, err := archival.NewClient()
    defer db.Close()

    err = db.Persist(order, status)
}
```

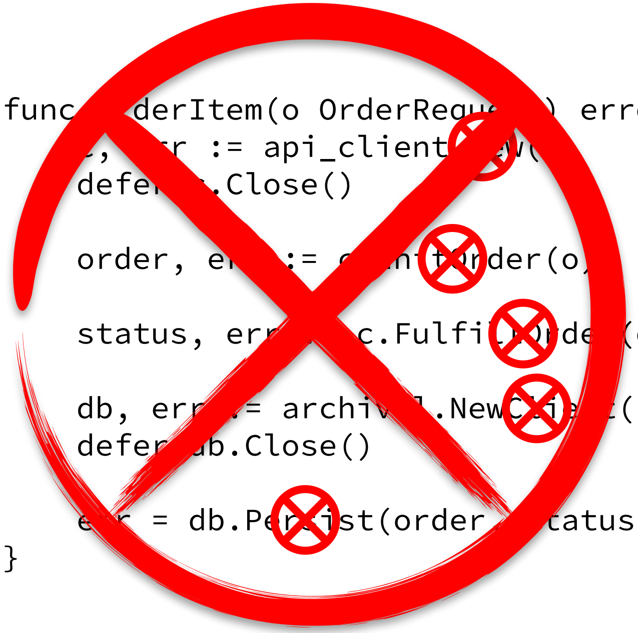
Unreliable Services

```
func OrderItem(o OrderRequest) error {
    c, err := api_client.New()
    defer c.Close()

    order, err := c.InitOrder(o)

    status, err := c.FulfillOrder(order)

    db, err := archival.NewClient()
    defer db.Close()

    err = db.Persist(order, status)
}
```

Unreliable Services

```
func OrderItem(o OrderRequest) error {
	c, err := api_client.New()
	defer c.Close()

	order, err := c.InitOrder(o)

	status, err := c.FulfillOrder(order)

	db, err := archival.NewClient()
	defer db.Close()

	err = db.Persist(order, status)
}
```

Unreliable Services



```
func [illegible]derItem(o OrderReq[illegible]) error {
    [illegible], [illegible]r := api_client[illegible]w(
    defe[illegible].Close()

    order, e[illegible] := c[illegible]nt[illegible]rder(o[illegible]

    status, er[illegible]c.Fulfi[illegible]rde[illegible](order)

    db, err[illegible]= archiv[illegible]l.NewC[illegible]e[illegible]()
    defer db.Close()

    e[illegible]r = db.Pe[illegible]ist(order[illegible]tatus)
}
```

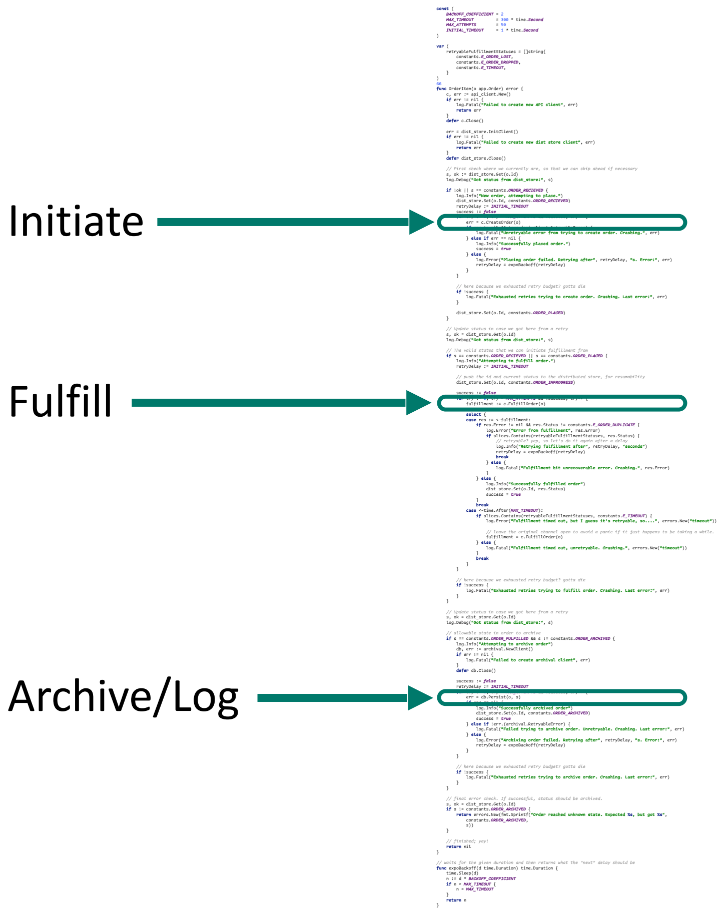
"Proper" Error Handling?

```
func OrderItem(o OrderRequest) error {
    c, err := api_client.New()
    if err != nil { log.Fatal(err) }
    defer c.Close()

    order, err := c.InitOrder(o)
    if err != nil { log.Fatal(err) }

    status, err := c.FulfillOrder(order)
    if err != nil { log.Fatal(err) }

    db, err := archival.NewClient()
    if err != nil { log.Fatal(err) }
    defer db.Close()
    err = db.Persist(order, status)
    if err != nil { log.Fatal(err) }
}
```

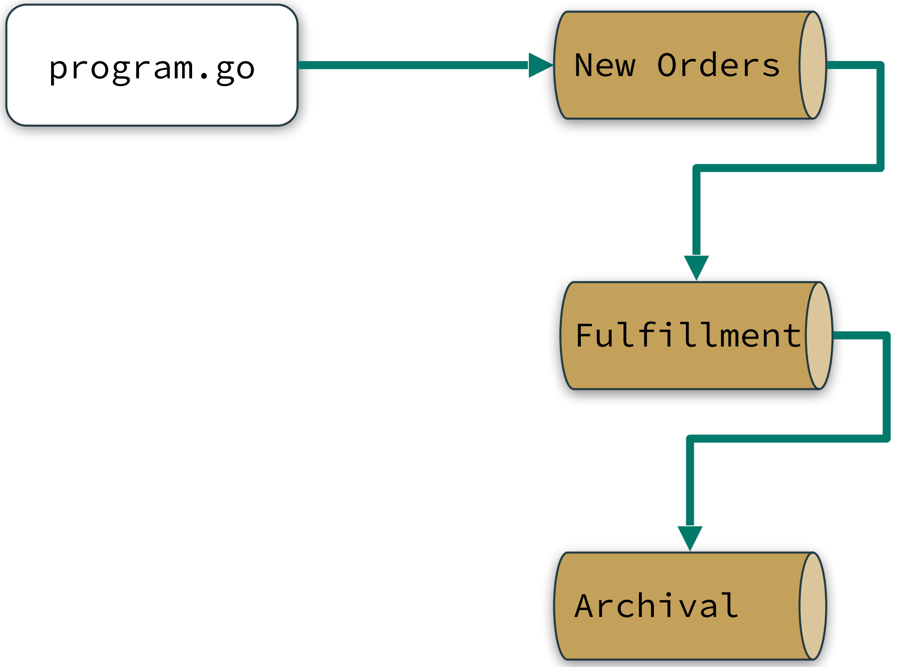
"Better"?

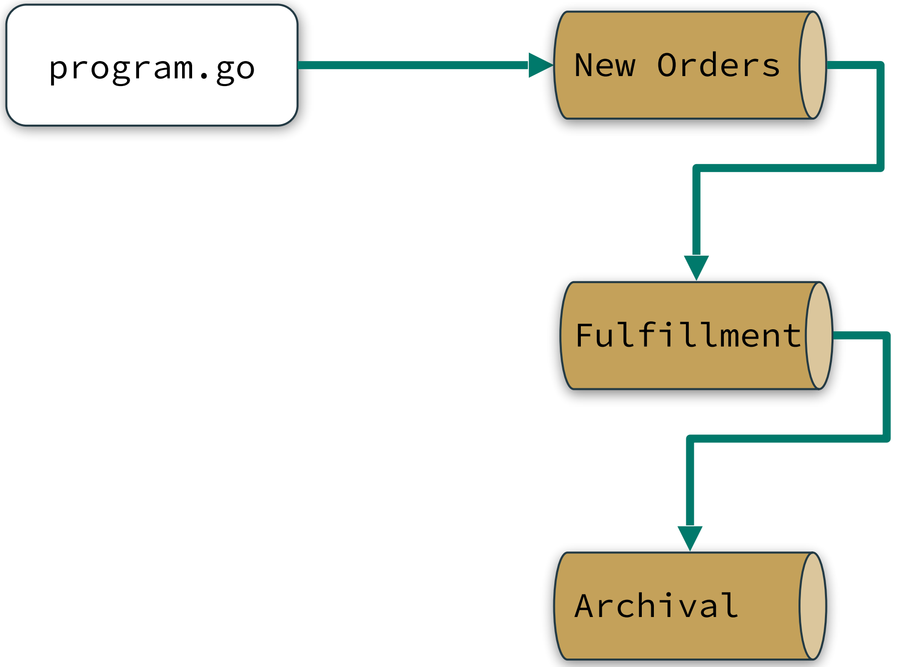
```go
const (
    BACKOFF_COEFFICIENT = 2
    MAX_TIMEOUT         = 300 * time.Second
    MAX_ATTEMPTS        = 50
    INITIAL_TIMEOUT     = 1 * time.Second
)

var (
    retryableFulfillmentStatuses = []string{
        constants.E_ORDER_LOST,
        constants.E_ORDER_DROPPED,
        constants.E_TIMEOUT,
    }
)
66
func OrderItem(o app.Order) error {
    c, err := api_client.New()
    if err != nil {
        log.Fatal("Failed to create new API client", err)
        return err
    }
    defer c.Close()

    err = dist_store.InitClient()
    if err != nil {
        log.Fatal("Failed to create new dist store client", err)
        return err
    }
    defer dist_store.Close()

    // First check where we currently are, so that we can skip ahead if necessary
    s, ok := dist_store.Get(o.Id)
    log.Debug("Got status from dist_store:", s)

    if !ok || s == constants.ORDER_RECIEVED {
        log.Info("New order, attempting to place.")
        dist_store.Set(o.Id, constants.ORDER_RECIEVED)
        retryDelay := INITIAL_TIMEOUT
        success := false
        for try := 0; try < MAX_ATTEMPTS && !success; try++ {
            err = c.CreateOrder(o)
            if err != nil && !err.(api_client.RetryableError) {
                log.Fatal("Unretryable error from trying to create order. Crashing.", err)
            } else if err == nil {
                log.Info("Successfully placed order.")
                success = true
            } else {
                log.Error("Placing order failed. Retrying after", retryDelay, "s. Error:", err)
                retryDelay = expoBackoff(retryDelay)
            }
        }

        // here because we exhausted retry budget? gotta die
        if !success {
            log.Fatal("Exhausted retries trying to create order. Crashing. Last error:", err)
        }

        dist_store.Set(o.Id, constants.ORDER_PLACED)
    }

    // Update status in case we got here from a retry
    s, ok = dist_store.Get(o.Id)
    log.Debug("Got status from dist_store:", s)

    // The valid states that we can initiate fulfillment from
    if s == constants.ORDER_RECIEVED || s == constants.ORDER_PLACED {
        log.Info("Attempting to fulfill order.")
        retryDelay := INITIAL_TIMEOUT

        // push the id and current status to the distributed store, for resumability
        dist_store.Set(o.Id, constants.ORDER_INPROGRESS)

        success := false
        for try := 0; try < MAX_ATTEMPTS && !success; try++ {
            fulfillment := c.FulfillOrder(o)

            select {
            case res := <-fulfillment:
                if res.Error != nil && res.Status != constants.E_ORDER_DUPLICATE {
                    log.Error("Error from fulfillment", res.Error)
                    if slices.Contains(retryableFulfillmentStatuses, res.Status) {
                        // retryable? yep, so let's do it again after a delay
                        log.Info("Retrying fulfillment after", retryDelay, "seconds")
                        retryDelay = expoBackoff(retryDelay)
                        break
                    } else {
                        log.Fatal("Fulfillment hit unrecoverable error. Crashing.", res.Error)
                    }
                } else {
                    log.Info("Successfully fulfilled order")
                    dist_store.Set(o.Id, res.Status)
                    success = true
                }
                break
            case <-time.After(MAX_TIMEOUT):
                if slices.Contains(retryableFulfillmentStatuses, constants.E_TIMEOUT) {
                    log.Error("Fulfillment timed out, but I guess it's retryable, so....", errors.New("timeout"))

                    // leave the original channel open to avoid a panic if it just happens to be taking a while.
                    fulfillment = c.FulfillOrder(o)
                } else {
                    log.Fatal("Fulfillment timed out, unretryable. Crashing.", errors.New("timeout"))
                }
                break
            }
        }

        // here because we exhausted retry budget? gotta die
        if !success {
            log.Fatal("Exhausted retries trying to fulfill order. Crashing. Last error:", err)
        }
    }

    // Update status in case we got here from a retry
    s, ok = dist_store.Get(o.Id)
    log.Debug("Got status from dist_store:", s)

    // allowable state in order to archive
    if s == constants.ORDER_FULFILLED && s != constants.ORDER_ARCHIVED {
        log.Info("Attempting to archive order")
        db, err := archival.NewClient()
        if err != nil {
            log.Fatal("Failed to create archival client", err)
        }
        defer db.Close()

        success := false
        retryDelay := INITIAL_TIMEOUT
        for try := 0; try < MAX_ATTEMPTS && !success; try++ {
            err = db.Persist(o, s)
            if err == nil {
                log.Info("Successfully archived order")
                dist_store.Set(o.Id, constants.ORDER_ARCHIVED)
                success = true
            } else if !err.(archival.RetryableError) {
                log.Fatal("Failed trying to archive order. Unretryable. Crashing. Last error:", err)
            } else {
                log.Error("Archiving order failed. Retrying after", retryDelay, "s. Error:", err)
                retryDelay = expoBackoff(retryDelay)
            }
        }

        // here because we exhausted retry budget? gotta die
        if !success {
            log.Fatal("Exhausted retries trying to archive order. Crashing. Last error:", err)
        }
    }

    // final error check. If successful, status should be archived.
    s, ok = dist_store.Get(o.Id)
    if s != constants.ORDER_ARCHIVED {
        return errors.New(fmt.Sprintf("Order reached unknown state. Expected %s, but got %s",
            constants.ORDER_ARCHIVED,
            s))
    }

    // finished; yay!
    return nil
}

// waits for the given duration and then returns what the "next" delay should be
func expoBackoff(d time.Duration) time.Duration {
    time.Sleep(d)
    n := d * BACKOFF_COEFFICIENT
    if n > MAX_TIMEOUT {
        n = MAX_TIMEOUT
    }
    return n
}
```

"Better"?

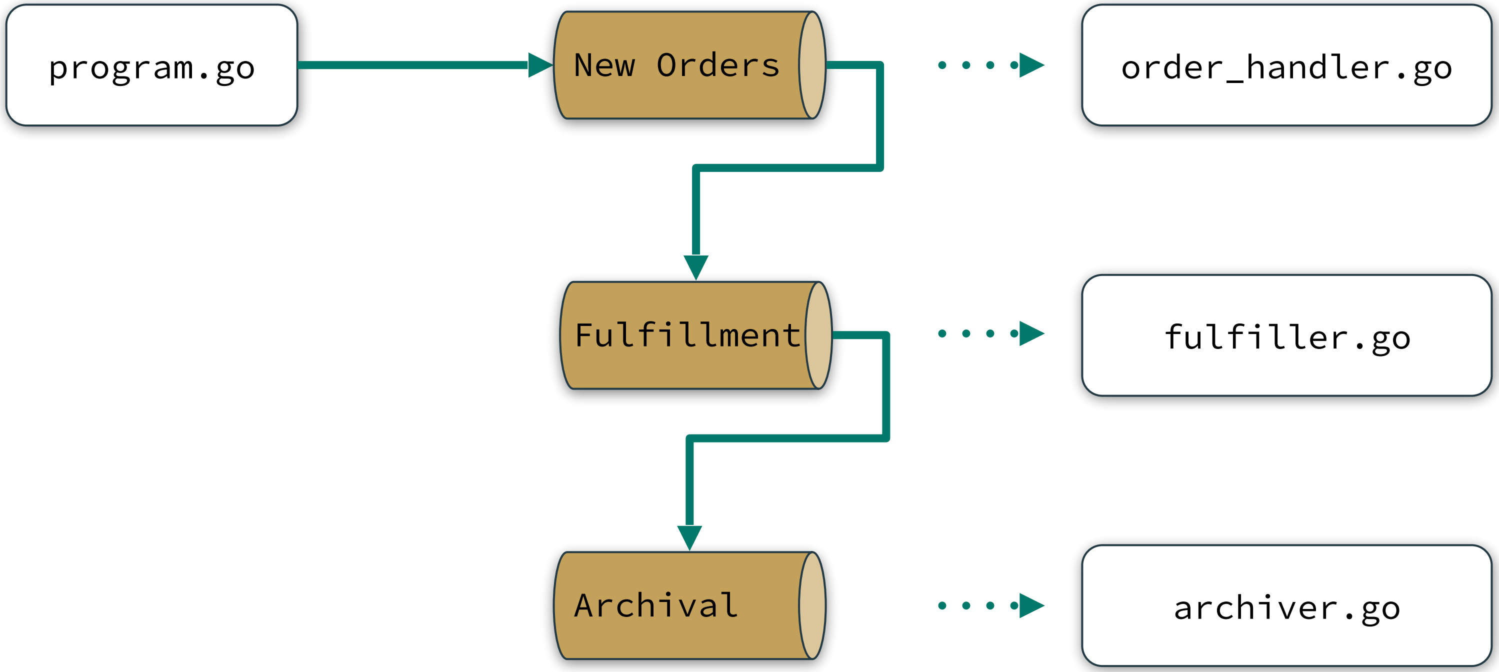


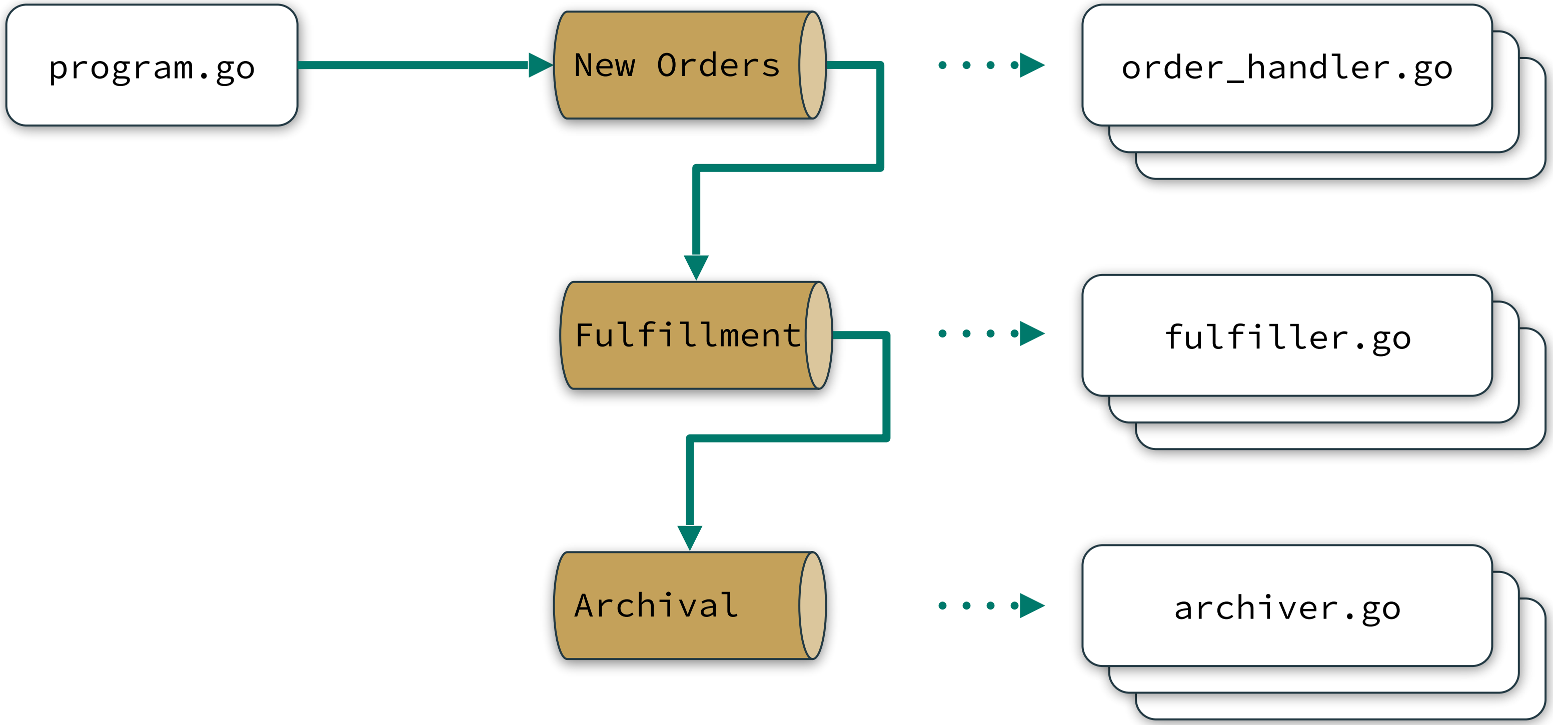
```go
const (
    BACKOFF_COEFFICIENT = 2
    MAX_TIMEOUT         = 300 * time.Second
    MAX_ATTEMPTS        = 50
    INITIAL_TIMEOUT     = 1 * time.Second
)

var (
    retryableFulfillmentStatuses = []string{
        constants.E_ORDER_LOST,
        constants.E_ORDER_DROPPED,
        constants.E_TIMEOUT,
    }
)
66
func OrderItem(o app.Order) error {
    c, err := api_client.New()
    if err != nil {
        log.Fatal("Failed to create new API client", err)
        return err
    }
    defer c.Close()

    err = dist_store.InitClient()
    if err != nil {
        log.Fatal("Failed to create new dist store client", err)
        return err
    }
    defer dist_store.Close()

    // First check where we currently are, so that we can skip ahead if necessary
    s, ok := dist_store.Get(o.Id)
    log.Debug("Got status from dist_store:", s)

    if !ok || s == constants.ORDER_RECIEVED {
        log.Info("New order, attempting to place.")
        dist_store.Set(o.Id, constants.ORDER_RECIEVED)
        retryDelay := INITIAL_TIMEOUT
        success := false
        [illegible]
            err = c.CreateOrder(o)
            [illegible]
                log.Fatal("Unretryable error from trying to create order. Crashing.", err)
            } else if err == nil {
                log.Info("Successfully placed order.")
                success = true
            } else {
                log.Error("Placing order failed. Retrying after", retryDelay, "s. Error:", err)
                retryDelay = expoBackoff(retryDelay)
            }
        }

        // here because we exhausted retry budget? gotta die
        if !success {
            log.Fatal("Exhausted retries trying to create order. Crashing. Last error:", err)
        }

        dist_store.Set(o.Id, constants.ORDER_PLACED)
    }

    // Update status in case we got here from a retry
    s, ok = dist_store.Get(o.Id)
    log.Debug("Got status from dist_store:", s)

    // The valid states that we can initiate fulfillment from
    if s == constants.ORDER_RECIEVED || s == constants.ORDER_PLACED {
        log.Info("Attempting to fulfill order.")
        retryDelay := INITIAL_TIMEOUT

        // push the id and current status to the distributed store, for resumability
        dist_store.Set(o.Id, constants.ORDER_INPROGRESS)

        success := false
        [illegible]
            fulfillment := c.FulfillOrder(o)

            select {
            case res := <-fulfillment:
                if res.Error != nil && res.Status != constants.E_ORDER_DUPLICATE {
                    log.Error("Error from fulfillment", res.Error)
                    if slices.Contains(retryableFulfillmentStatuses, res.Status) {
                        // retryable? yep, so let's do it again after a delay
                        log.Info("Retrying fulfillment after", retryDelay, "seconds")
                        retryDelay = expoBackoff(retryDelay)
                        break
                    } else {
                        log.Fatal("Fulfillment hit unrecoverable error. Crashing.", res.Error)
                    }
                } else {
                    log.Info("Successfully fulfilled order")
                    dist_store.Set(o.Id, res.Status)
                    success = true
                }
                break
            case <-time.After(MAX_TIMEOUT):
                if slices.Contains(retryableFulfillmentStatuses, constants.E_TIMEOUT) {
                    log.Error("Fulfillment timed out, but I guess it's retryable, so....", errors.New("timeout"))

                    // leave the original channel open to avoid a panic if it just happens to be taking a while.
                    fulfillment = c.FulfillOrder(o)
                } else {
                    log.Fatal("Fulfillment timed out, unretryable. Crashing.", errors.New("timeout"))
                }
                break
            }
        }

        // here because we exhausted retry budget? gotta die
        if !success {
            log.Fatal("Exhausted retries trying to fulfill order. Crashing. Last error:", err)
        }
    }

    // Update status in case we got here from a retry
    s, ok = dist_store.Get(o.Id)
    log.Debug("Got status from dist_store:", s)

    // allowable state in order to archive
    if s == constants.ORDER_FULFILLED && s != constants.ORDER_ARCHIVED {
        log.Info("Attempting to archive order")
        db, err := archival.NewClient()
        if err != nil {
            log.Fatal("Failed to create archival client", err)
        }
        defer db.Close()

        success := false
        retryDelay := INITIAL_TIMEOUT
        [illegible]
            err = db.Persist(o, s)
            [illegible]
                log.Info("Successfully archived order")
                dist_store.Set(o.Id, constants.ORDER_ARCHIVED)
                success = true
            } else if !err.(archival.RetryableError) {
                log.Fatal("Failed trying to archive order. Unretryable. Crashing. Last error:", err)
            } else {
                log.Error("Archiving order failed. Retrying after", retryDelay, "s. Error:", err)
                retryDelay = expoBackoff(retryDelay)
            }
        }

        // here because we exhausted retry budget? gotta die
        if !success {
            log.Fatal("Exhausted retries trying to archive order. Crashing. Last error:", err)
        }
    }

    // final error check. If successful, status should be archived.
    s, ok = dist_store.Get(o.Id)
    if s != constants.ORDER_ARCHIVED {
        return errors.New(fmt.Sprintf("Order reached unknown state. Expected %s, but got %s",
            constants.ORDER_ARCHIVED,
            s))
    }

    // finished; yay!
    return nil
}

// waits for the given duration and then returns what the "next" delay should be
func expoBackoff(d time.Duration) time.Duration {
    time.Sleep(d)
    n := d * BACKOFF_COEFFICIENT
    if n > MAX_TIMEOUT {
        n = MAX_TIMEOUT
    }
    return n
}
```

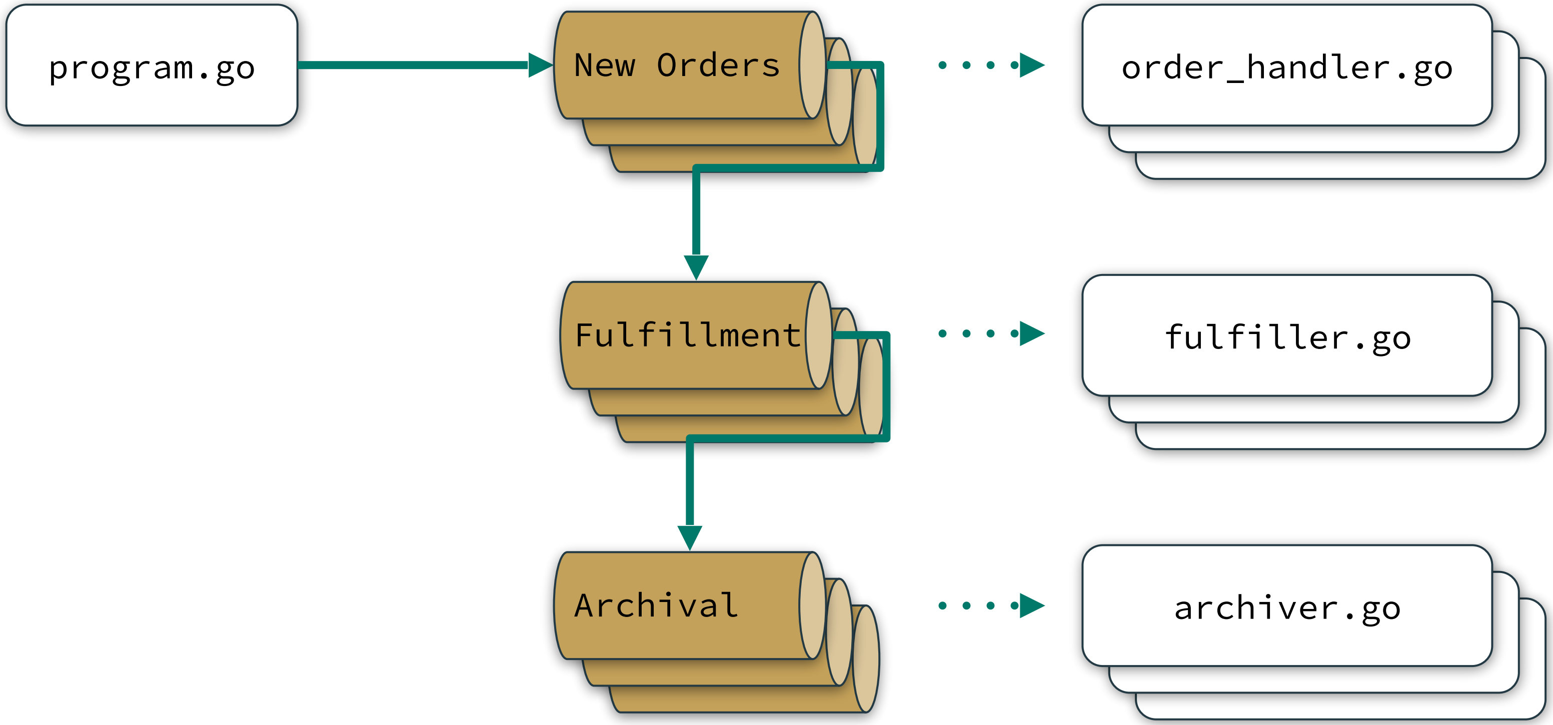

program.go
New Orders
Fulfillment
Archival

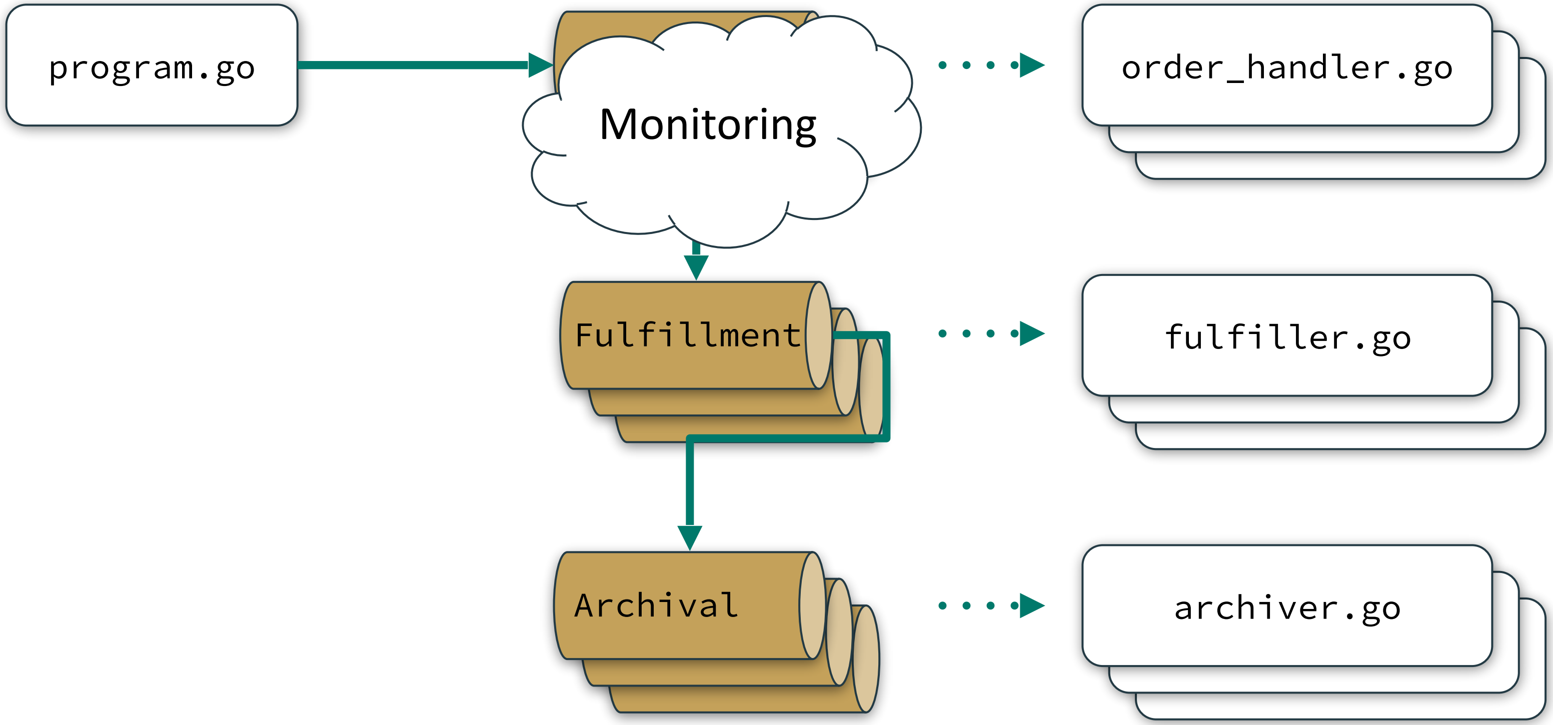


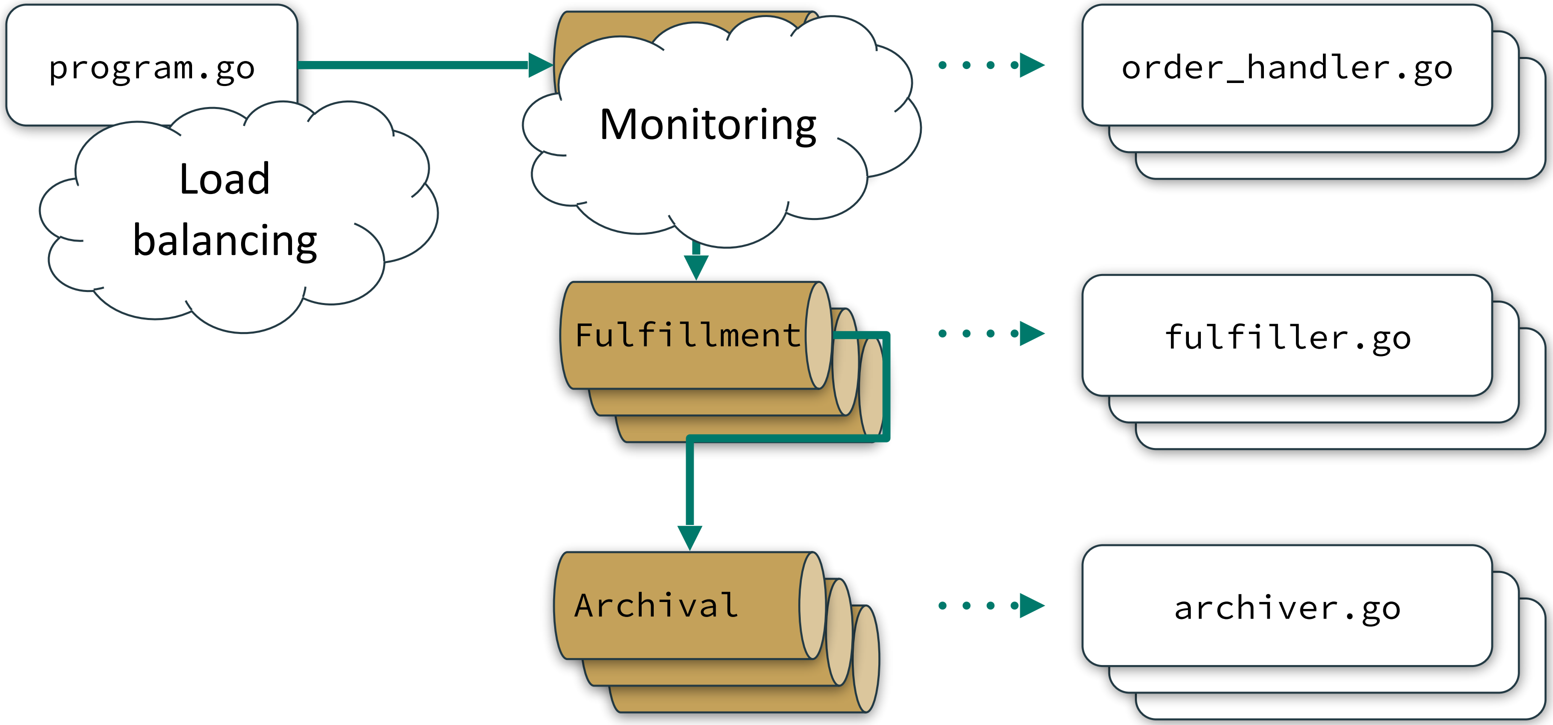
```
// pseudocode
func doThing() {
  orders.sendMessage()
}
```

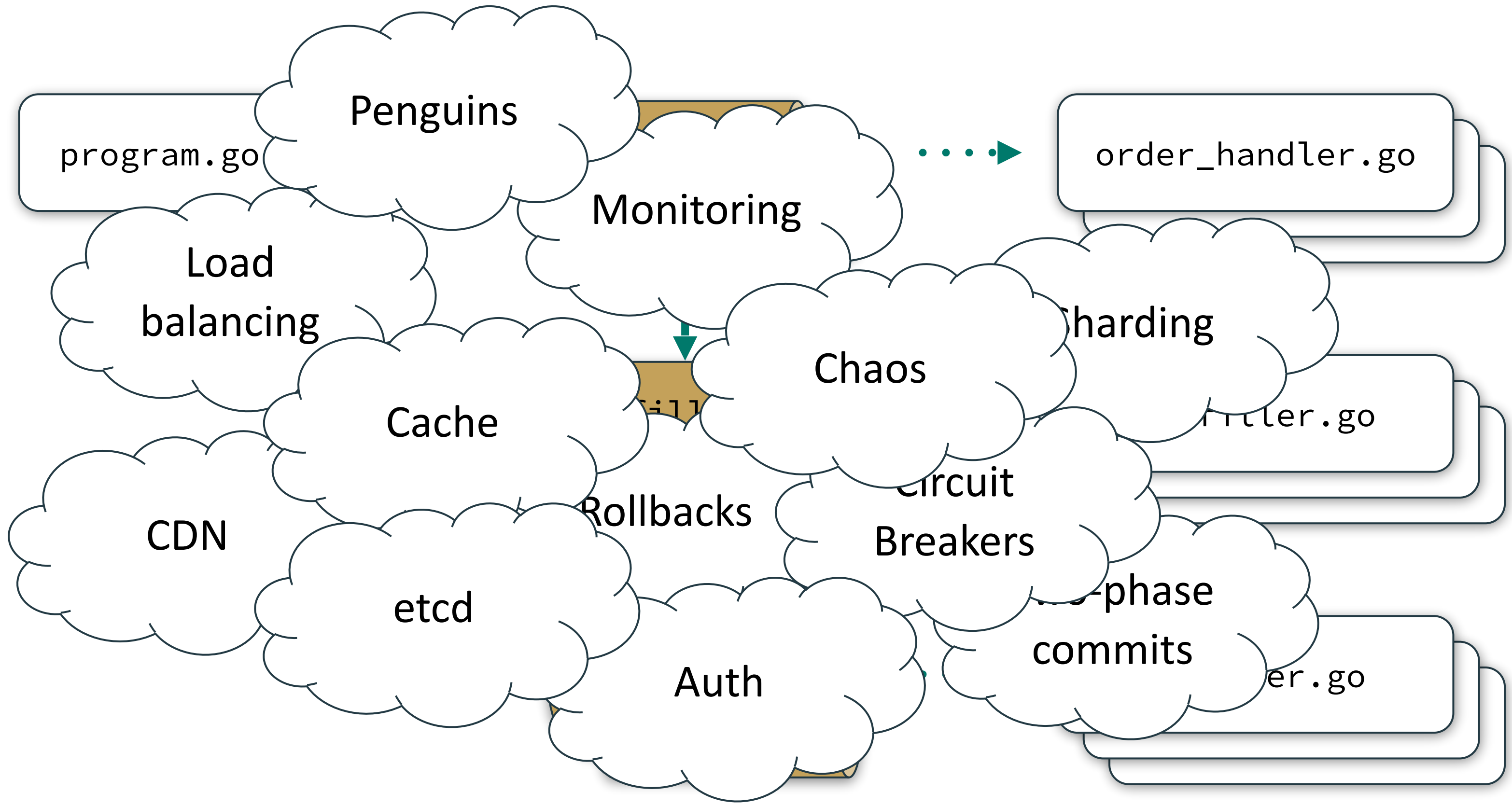

program.go
New Orders
order_handler.go
Fulfillment
fulfiller.go
Archival
archiver.go


program.go
New Orders
Fulfillment
Archival
order_handler.go
fulfiller.go
archiver.go


program.go
New Orders
order_handler.go
Fulfillment
fulfiller.go
Archival
archiver.go


program.go
Monitoring
Fulfillment
Archival
order_handler.go
fulfiller.go
archiver.go


program.go
Load balancing
Monitoring
Fulfillment
Archival
order_handler.go
fulfiller.go
archiver.go


program.go
Penguins
Monitoring
order_handler.go
Load balancing
Sharding
Chaos
Cache
iller.go
Circuit Breakers
Rollbacks
CDN
etcd
two-phase commits
er.go
Auth

❤️‍🔥 Failure ❤️‍🔥

Simple Program (pseudo-go)

```
func OrderItem(o OrderRequest) error {
    c, err := api_client.New()
    defer c.Close()

    order, err := c.InitOrder(o)

    status, err := c.FulfillOrder(order)

    db, err := archival.NewClient()
    defer db.Close()

    err = db.Persist(order, status)
}
```

Caveats:

- Absolutely NO error handling
- Process or node dies?

  → Restart from beginning

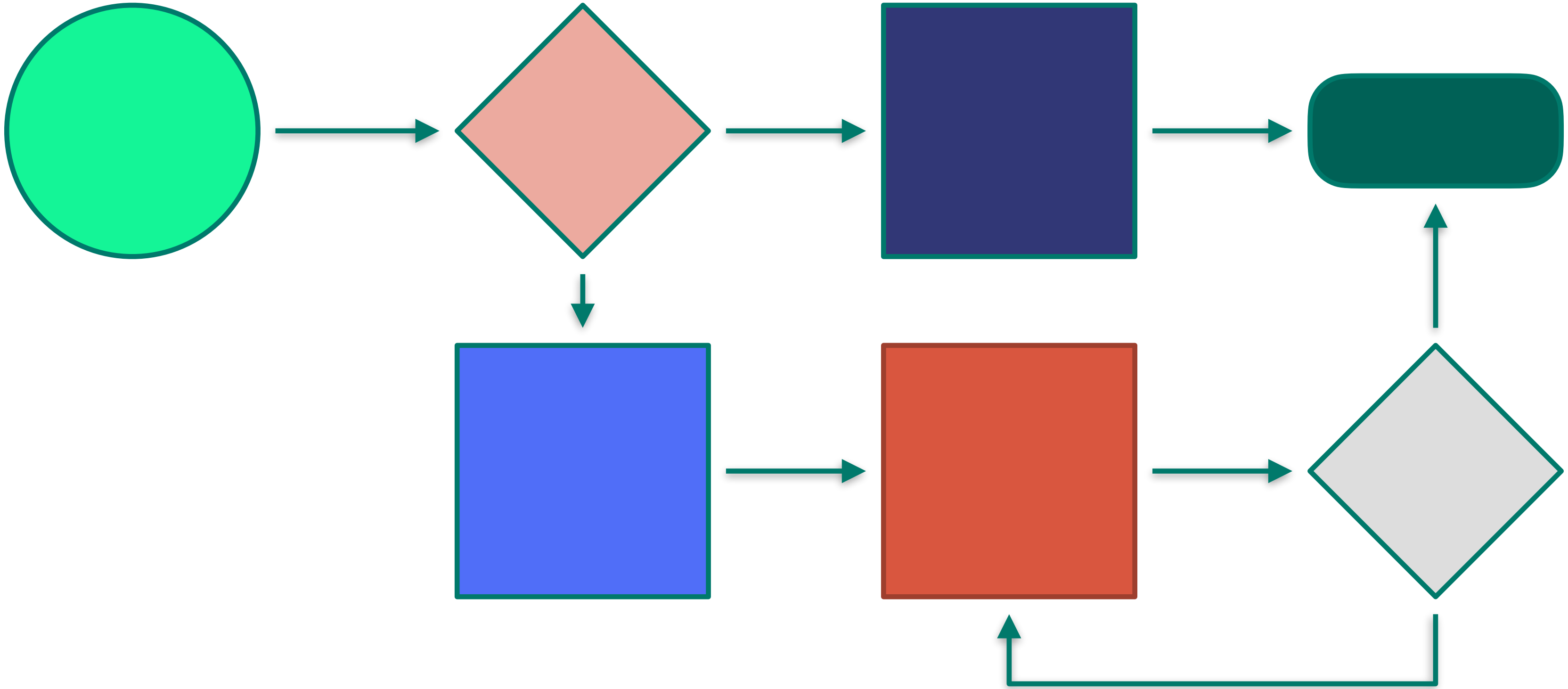
- API call takes too long?

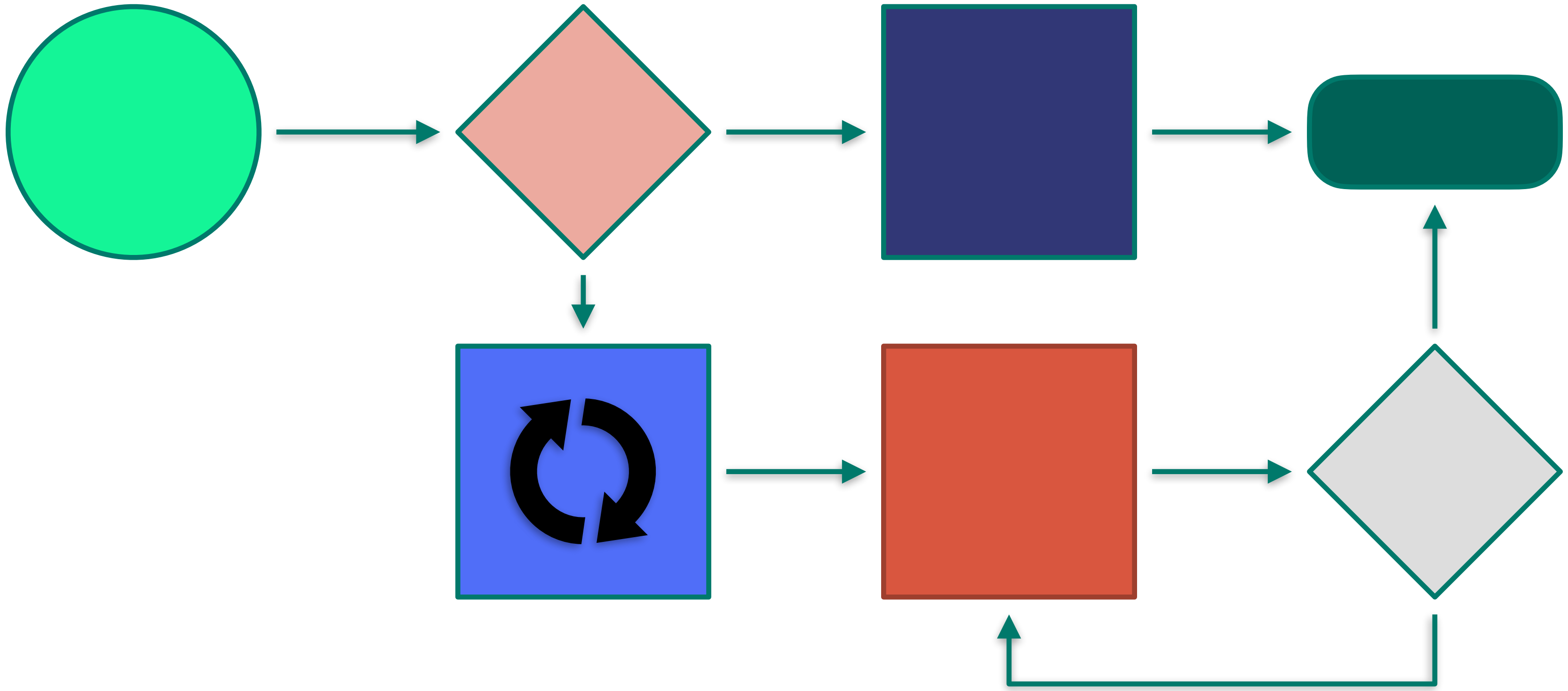
  → Probably also restart

Durable Execution

... with Temporal

https://github.com/temporalio/temporal





Simple Program (Temporal Workflow)

Initiate

Fulfill

Archive

```go
func OrderItem(ctx workflow.Context, o Order) error {
    retryPolicy := &temporal.RetryPolicy{
        InitialInterval:    time.Second,
        BackoffCoefficient: 2,
        MaximumInterval:    10 * time.Second,
        MaximumAttempts:    100,
        NonRetryableErrorTypes: []string{"PaymentFailed"},
    }
    ao := workflow.ActivityOptions{
        StartToCloseTimeout: 10 * time.Second,
        RetryPolicy: retryPolicy,
    }
    ctx = workflow.WithActivityOptions(ctx, ao)
    log := workflow.GetLogger(ctx)

    err := workflow.ExecuteActivity(ctx,
        InitOrder, o).Get(ctx, &o)
    if err != nil {
        log.Error("CreateOrder failed", "Err", err)
        return err
    }

    var status OrderStatus
    err = workflow.ExecuteActivity(ctx,
        FulfillOrder).Get(ctx, &status)
    if err != nil {
        log.Error("FulfillOrder failed", "Err", err)
        return err
    }
    err = workflow.ExecuteActivity(ctx,
        ArchiveOrder).Get(ctx, nil)
    if err != nil {
        log.Error("ArchiveOrder failed", "Err", err)
        return err
    }
}
```

Simple Program (Temporal Workflow)

Initiate

Fulfill

Archive

```
func InitOrder(ctx context.Context, o Order) (Order, error) {
    c, err := api_client.New()
    if err != nil {
        log.Error("Could not create api client", err)
        return err
    }
    defer c.Close()

    order, err := c.InitOrder(o)
    return order, err
}
```

```
func FulfillOrder(ctx context.Context, o Order) (string, error) {
    c, err := api_client.New()
    if err != nil {
        log.Error("Could not create api client", err)
        return err
    }
    defer c.Close()

    status, err := c.FulfillOrder(o)
    return status, err
}
```

```
func ArchiveOrder(ctx context.Context, o Order, s string) error {
    db, err := archiver.NewClient()
    if err != nil {
        log.Error("Could not create api client", err)
        return err
    }
    defer db.Close()

    err = db.Persist(o, s)
    return err
}
```

```go
const (
    BACKOFF_COEFFICIENT = 2
    MAX_TIMEOUT         = 300 * time.Second
    MAX_ATTEMPTS        = 50
    INITIAL_TIMEOUT     = 1 * time.Second
)

var (
    retryableFulfillmentStatuses = []string{
        constants.E_ORDER_LOST,
        constants.E_ORDER_DROPPED,
        constants.E_TIMEOUT,
    }
)
66
func OrderItem(o app.Order) error {
    c, err := api_client.New()
    if err != nil {
        log.Fatal("Failed to create new API client", err)
        return err
    }
    defer c.Close()

    err = dist_store.InitClient()
    if err != nil {
        log.Fatal("Failed to create new dist store client", err)
        return err
    }
    defer dist_store.Close()

    // First check where we currently are, so that we can skip ahead if necessary
    s, ok := dist_store.Get(o.Id)
    log.Debug("Got status from dist_store:", s)

    if !ok || s == constants.ORDER_RECIEVED {
        log.Info("New order, attempting to place.")
        dist_store.Set(o.Id, constants.ORDER_RECIEVED)
        retryDelay := INITIAL_TIMEOUT
        success := false
        for try := 0; try < MAX_ATTEMPTS && !success; try++ {
            err = c.CreateOrder(o)
            if err != nil && !err.(api_client.RetryableError) {
                log.Fatal("Unretryable error from trying to create order. Crashing.", err)
            } else if err == nil {
                log.Info("Successfully placed order.")
                success = true
            } else {
                log.Error("Placing order failed. Retrying after", retryDelay, "s. Error:", err)
                retryDelay = expoBackoff(retryDelay)
            }
        }

        // here because we exhausted retry budget? gotta die
        if !success {
            log.Fatal("Exhausted retries trying to create order. Crashing. Last error:", err)
        }

        dist_store.Set(o.Id, constants.ORDER_PLACED)
    }

    // Update status in case we got here from a retry
    s, ok = dist_store.Get(o.Id)
    log.Debug("Got status from dist_store:", s)

    // The valid states that we can initiate fulfillment from
    if s == constants.ORDER_RECIEVED || s == constants.ORDER_PLACED {
        log.Info("Attempting to fulfill order.")
        retryDelay := INITIAL_TIMEOUT

        // push the id and current status to the distributed store, for resumability
        dist_store.Set(o.Id, constants.ORDER_INPROGRESS)

        success := false
        for try := 0; try < MAX_ATTEMPTS && !success; try++ {
            fulfillment := c.FulfillOrder(o)

            select {
            case res := <-fulfillment:
                if res.Error != nil && res.Status != constants.E_ORDER_DUPLICATE {
                    log.Error("Error from fulfillment", res.Error)
                    if slices.Contains(retryableFulfillmentStatuses, res.Status) {
                        // retryable? yep, so let's do it again after a delay
                        log.Info("Retrying fulfillment after", retryDelay, "seconds")
                        retryDelay = expoBackoff(retryDelay)
                        break
                    } else {
                        log.Fatal("Fulfillment hit unrecoverable error. Crashing.", res.Error)
                    }
                } else {
                    log.Info("Successfully fulfilled order")
                    dist_store.Set(o.Id, res.Status)
                    success = true
                }
                break
            case <-time.After(MAX_TIMEOUT):
                if slices.Contains(retryableFulfillmentStatuses, constants.E_TIMEOUT) {
                    log.Error("Fulfillment timed out, but I guess it's retryable, so....", errors.New("timeout"))

                    // leave the original channel open to avoid a panic if it just happens to be taking a while.
                    fulfillment = c.FulfillOrder(o)
                } else {
                    log.Fatal("Fulfillment timed out, unretryable. Crashing.", errors.New("timeout"))
                }
                break
            }
        }

        // here because we exhausted retry budget? gotta die
        if !success {
            log.Fatal("Exhausted retries trying to fulfill order. Crashing. Last error:", err)
        }
    }

    // Update status in case we got here from a retry
    s, ok = dist_store.Get(o.Id)
    log.Debug("Got status from dist_store:", s)

    // allowable state in order to archive
    if s == constants.ORDER_FULFILLED && s != constants.ORDER_ARCHIVED {
        log.Info("Attempting to archive order")
        db, err := archival.NewClient()
        if err != nil {
            log.Fatal("Failed to create archival client", err)
        }
        defer db.Close()

        success := false
        retryDelay := INITIAL_TIMEOUT
        for try := 0; try < MAX_ATTEMPTS && !success; try++ {
            err = db.Persist(o, s)
            if err == nil {
                log.Info("Successfully archived order")
                dist_store.Set(o.Id, constants.ORDER_ARCHIVED)
                success = true
            } else if !err.(archival.RetryableError) {
                log.Fatal("Failed trying to archive order. Unretryable. Crashing. Last error:", err)
            } else {
                log.Error("Archiving order failed. Retrying after", retryDelay, "s. Error:", err)
                retryDelay = expoBackoff(retryDelay)
            }
        }

        // here because we exhausted retry budget? gotta die
        if !success {
            log.Fatal("Exhausted retries trying to archive order. Crashing. Last error:", err)
        }
    }

    // final error check. If successful, status should be archived.
    s, ok = dist_store.Get(o.Id)
    if s != constants.ORDER_ARCHIVED {
        return errors.New(fmt.Sprintf("Order reached unknown state. Expected %s, but got %s",
            constants.ORDER_ARCHIVED,
            s))
    }

    // finished; yay!
    return nil
}

// waits for the given duration and then returns what the "next" delay should be
func expoBackoff(d time.Duration) time.Duration {
    time.Sleep(d)
    n := d * BACKOFF_COEFFICIENT
    if n > MAX_TIMEOUT {
        n = MAX_TIMEOUT
    }
    return n
}
```

❤️‍🔥 Failure ❤️‍🔥

❤️‍🔥 ~~Failure~~ ❤️‍🔥

Demo

Temporal:
- temporal.io
- github.com/temporalio

This demo (including slide 5's "better?" code):
- github.com/afitz0/gophercon2022-lightning-talk

Andrew Fitz Gibbon (aka "Fitz")
Staff Developer Advocate
@ Temporal Technologies

✉ fitz@temporal.io
@fitzface