ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 09 Cloud Native Patterns

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

September 18, 2024

Outline

Reading Assignment

- This lecture: 4
- Next lecture: 4

Outline

Fallacies of Distributed Computing

- 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
- Services are reliable

Cloud Native Patterns

- Patterns are proven development paradigms.
  - Learn from professionals.
  - Facilitate communication between professionals.
- Cloud native patterns help to address the fallacies of distributed computing and mitigate them.
- Explore unique and novel Go implementations.

Outline

Cloud Native Patterns

The Context Package

Stability Patterns

The Context Package

- Introduced in Go 1.7
- Provides a unified framework for carrying deadlines, cancellation signals, and request-scoped values between function calls and threads.
- Via the `context.Context` interface.

The Context Interface

```
type Context interface {
  Done() <-chan struct{}
  Err() error
  Deadline() (deadline time.Time, ok bool)
  Value(key interface{}) interface{}
}
```

- A context could refer to a request from a user.
- `Done` returns a channel indicating if the context is cancelled.
  - The channel is closed when the context is cancelled.
  - The channel is not used to communicate any data (`struct{}`).
  - `Err` indicates the reason of cancellation.
- `Deadline` returns the deadline of the context if there is any.
  - A function could choose to terminate if it decides there is not enough time left.
- `Value` allows to access data associated with this context organized as key-value pairs.
  - E.g. the identity of the user

What Context Can Do for You

- Consider a RESTful request that need to be handled.
  - Similar to our simple key-value store example, this request will go through multiple functions and threads before a response can be generated.
- What if the request is cancelled during the process?
  - Completing the process but not returning the response will work but be a waste.
  - Need a mechanism to communicate to functions and threads that the request is cancelled.
- Context works as a mechanism to notify so.
  - Functions and threads can check if `Done` returns a closed channel and terminate without consuming additional resources.
  - Context has to be thread safe for such purpose – be careful with the `Value` method to not modify what it returns.

Building Context

```
func Background() Context
func TODO() Context

func WithCancel(Context) (Context, CancelFunc)
func WithTimeout(Context, time.Duration) (Context, CancelFunc)
func WithDeadline(Context, time.Time) (Context, CancelFunc)
func WithValue(parent Context, key, val interface{}) Context
...
```

- Start with the empty context either from `Background` or `TODO`
- Add cancellations, deadlines, and data by calling the other functions.
- This is also known as the Decorator pattern.
  - Widely used to dynamically modify object behaviors.

An Example

```
func ReadObject(ctx context.Context, out chan<- Value) error {
  dctx, cancel := context.WithTimeout(ctx, time.Second * 10)
  defer cancel() // always cancel at the end
  res, err := RequestObject(dctx) // expect response within a deadline
  if err != nil { // if dctx times out
    return err
  }
  for { // it takes time to send response back
    select {
    case out <- res: // receive from res; send to out
    case <-ctx.Done(): // if ctx is cancelled somewhere
      return ctx.Err()
    }
  }
} // adapted from textbook example by giving meaningful names
```

- `ReadObject` need to send the object to `out`.
  - It calls `RequestObject` to obtain a channel to receive it.
- However, `RequestObject` will need to obtain the object from another service, which may be too busy sometimes.
  - `ReadObject` sets a timeout for `RequestObject`.

Outline

Retry

- Errors and failures are always there.
  - Some of them are transient due to temporary conditions.
- Transient faults typically resolve themselves after a bit of time.
- Retry: retrying a failed operation that is most likely transient.
- Participants
  - Effector: the original function interacting with the service.
  - Retry: a closure with the same function signature as Effector that retries Effector if failed.
  - Retry works as a decorator of Effector to change its behavior.

Retry Example

```
type Effector func(context.Context) (string, error)
func Retry(effector Effector, retries int, delay time.Duration) Effector {
  return func(ctx context.Context) (string, error) {
    for r := 0; ; r++ {
      response, err := effector(ctx)
      if err == nil || r >= retries {
        return response, err
      }
      log.Printf("Attempt %d failed; retrying in %v", r + 1, delay)
      select {
        case <-time.After(delay):
        case <-ctx.Done():
          return "", ctx.Err()
      }
    }
  }
}
```

- Decorate `effector` by using an anonymous function.
- Use `select` to wait for delay and cancellation at the same time – whichever comes first will be handled.

Circuit Breaker

- Not all faults are transient.
  - Misconfiguration, databases crash, etc.
  - Simply retry failed requests are not helpful.
- Not addressing it may cause worse issues.
  - Services return nonsense as others fail.
  - Services fall into a crash/restart death spiral.
  - Waste resources and obscure source of original failure.
- Circuit breaker: to degrade service function in response to a possible fault in order to prevent cascading failures.
  - Detect failures and temporarily stop executing requests.
  - Provide meaningful error messages.
- Participants
  - Circuit: the original function interacting with the service.
  - Breaker: a closure as a decorator of Curcuit with desired failure handling logic.

Circuit Breaker Example

```
type Circuit func(context.Context) (string, error)
func Breaker(circuit Circuit) Circuit {
  var consecutiveFailures int = 0
  var lastAttempt = time.Now()
  return func(ctx context.Context) (string, error) {
    if consecutiveFailures >= 5 {
      shouldRetryAt := lastAttempt.Add(time.Second * 10)
      if !time.Now().After(shouldRetryAt) {
        return "", errors.New("service unreachable")
      }
    }
    response, err := circuit(ctx) // Issue request proper
    lastAttempt = time.Now() // Record time of attempt
    if err != nil { // Circuit returned an error,
      consecutiveFailures++ // so we count the failure
      return response, err // and return
    }
    consecutiveFailures = 0 // Reset failures counter
    return response, nil
  } // simplified from textbook example by removing
}   // exponential backoff and multi-thread supports
```

- A circuit breaker to prevent futile retries: retries after 5 consecutive fails need to be 10 seconds apart.

Debounce

- External inputs can be very unpredictable.
  - Users may spam-click buttons because of slow response.
  - Reacting with every such input further slows system down.
  - Not processing some requests is better than not processing anything at all.
- Debounce: limits the frequency of actual function calls so that only the first or the last go through.
- Participants
  - Circuit: the original function to regulate
  - Debounce: a closure as a decorator of Circuit that calls only once but reuses results for others.

Debounce Example

```
type Circuit func(context.Context) (string, error)
func DebounceFirst(circuit Circuit, d time.Duration) Circuit {
  var threshold time.Time
  var result string
  var err error
  var m sync.Mutex
  return func(ctx context.Context) (string, error) {
    m.Lock()
    defer func() {
      threshold = time.Now().Add(d)
      m.Unlock()
    }()
    if time.Now().Before(threshold) {
      return result, err
    }
    result, err = circuit(ctx)
    return result, err
  }
}
```

- Record the result and reuse it for consecutive calls within a short period of time.
- Waht about `DebounceLast`?

Throttle

- Services may limit maximum number of calls per unit of time
  - To address unexpected behavior from external inputs while still allow many requests to go through.
  - To preserve resources for fairness among many clients.
- Throttle: limits the frequency of actual function calls to match system requirements.
- Participants
  - Effector: the original function to regulate
  - Throttle: a closure as a decorator of Effector to limit rate of calls.

Throttle Example

```
type Effector func(context.Context) (string, error)
func Throttle(e Effector, max uint, refill uint, d time.Duration) Effector {
  var tokens = max
  var once sync.Once
  return func(ctx context.Context) (string, error) {
    // handle cancellation
    if ctx.Err() != nil {
      return "", ctx.Err()
    }
    // setup a goroutine to refill tokens
    once.Do(func() {
      ...
    })
    if tokens <= 0 {
      return "", fmt.Errorf("too many calls")
    }
    tokens--
    return e(ctx)
  }
}
```

- Return an error for calls beyond a predefined rate.

Throttle Example (Cont.)

```
// setup a goroutine to refill tokens
once.Do(func() {
  ticker := time.NewTicker(d)
  go func() {
    defer ticker.Stop()
    for {
      select {
      case <-ctx.Done(): // handle cancellation
        return
      case <-ticker.C: // d time has passed
        t := tokens + refill
        if t > max {
          t = max
        }
        tokens = t
      }
    }
  }()
})
```

- Allow to accumulate unused tokens up to a maximum.

Summary

- Distributed computing is very different than simply connecting services via networks.
- Learn proven paradigms for distributed computing via cloud native patterns.