

# Concurrent Programming Languages

Channel-based Concurrency Module

## **Lecture 3:** High-level Patterns

26 October 2021

**MIEI - Integrated Masters in Comp. Science and Informatics  
Specialization Block**

**Bernardo Toninho**

(with António Ravara and Carla Ferreira)



**NOVALINCS**

# Last Lecture

- Coordinating and synchronising go routines using channels.
- Selective communication.
- First-class channels.
- Various useful “micro” patterns.
- Assembling and composing patterns.

# Today

- High-level coordination patterns:
  - Contexts
  - Request Replication
  - Worker Pools
  - Pipelines
  - Rate Limits

# How to make concurrency safe?

- Immutable data
  - Implicitly concurrent-safe!
  - May pay too high a runtime penalty if not careful.
- Protect data by “confinement”
  - Ensure data is only ever available to/from one concurrent process.
- Sometimes synchronisation is just unavoidable...

# Structuring Go Concurrency

- Go concurrent/async apps naturally structured as client(s)-server(s).
- In a server “component”, each request is handled in its own goroutine.
- Handlers may start additional goroutines for various purposes, using request specific values (e.g. auth tokens).
- How to handle cancellation / time-outs that need to cascade?

# Go Contexts (`context` package)

- To assist with passing request-scoped data, cancellation signals and deadlines *across* API boundaries.

# Go Contexts (`context` package)

- To assist with passing request-scoped data, cancellation signals and deadlines *across* API boundaries.

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

- Returns a channel that is closed when the Context is cancelled or times out.

# Go Contexts (`context` package)

- To assist with passing request-scoped data, cancellation signals and deadlines *across* API boundaries.

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

- Returns the cause of the cancellation, after the Done channel is closed.



# Go Contexts (`context` package)

- To assist with passing request-scoped data, cancellation signals and deadlines *across* API boundaries.

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

- Returns the time when the Context will be cancelled, if any.

# Go Contexts (`context` package)

- To assist with passing request-scoped data, cancellation signals and deadlines *across* API boundaries.

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

- Returns the value associated with key, or nil if no association exists (data must be safe for concurrency).

# Go Contexts (`context` package)

- Contexts are *derived* from other Contexts, forming a tree.
- Cancelling a Context cancels all *derived* Contexts.

```
func Background() Context
```

```
type CancelFunc func()
```

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
```

```
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

```
func WithValue(parent Context, key interface{}, val interface{}) Context
```

- Functional-style API, returning the new Context.

# Go Contexts (`context` package)

- Contexts are *derived* from other Contexts, forming a tree.
- Cancelling a Context cancels all *derived* Contexts.

```
func Background() Context
```

```
type CancelFunc func()
```

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
```

```
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

```
func WithValue(parent Context, key interface{}, val interface{}) Context
```

- An “empty” context. Never cancelled, no deadline, no values.

# Go Contexts (`context` package)

- Contexts are *derived* from other Contexts, forming a tree.
- Cancelling a Context cancels all *derived* Contexts.

```
func Background() Context
```

```
type CancelFunc func()
```

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
```

```
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

```
func WithValue(parent Context, key interface{}, val interface{}) Context
```

- Returns a copy of parent, whose `Done` channel is closed as soon as `parent.Done` or `cancel` is called.

# Go Contexts (`context` package)

- Contexts are *derived* from other Contexts, forming a tree.
- Cancelling a Context cancels all *derived* Contexts.

```
func Background() Context
```

```
type CancelFunc func()
```

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
```

```
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

```
func WithValue(parent Context, key interface{}, val interface{}) Context
```

- Same as `WithCancel`, but with a timeout.

# Go Contexts (`context` package)

- Contexts are *derived* from other Contexts, forming a tree.
- Cancelling a Context cancels all *derived* Contexts.

```
func Background() Context
```

```
type CancelFunc func()
```

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
```

```
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

```
func WithValue(parent Context, key interface{}, val interface{}) Context
```

- Associates (`key, val`) to the Context.

# Using Contexts

- Functions receive values of `Context` type. Functions cannot cancel callers, only callees.
- If a function needs to cancel a goroutine below it in the call graph, it derives a new `Context` with the appropriate cancellation spec. and passes it to its children.
- Its crucial to not “store” instances of `Context` but rather to treat them “functionally”.



# Example — Workers & Contexts

```
type OpType = int
```

```
const (  
  Add op = iota  
  Sub  
  Mul  
  Div  
)
```

```
type WorkReq struct {  
  Op OpType  
  Val1 int  
  Val2 int  
}
```

```
type WorkResp struct {
```

```
  Wr *WorkReq
```

```
  Result int
```

```
  Err error  
}
```

```
func Worker(ctxt context.Context, in chan *WorkReq,  
           out chan *WorkResp) {
```

```
  for {
```

```
    select {
```

```
      case <- ctxt.Done():
```

```
        return
```

```
      case job := <- in:
```

```
        out <- Calc(wr) //Obvious calculation
```

```
    }
```

```
  }
```

```
}
```

# Example — Workers & Contexts

```
type OpType = int
```

```
const (  
  Add op = iota  
  Sub  
  Mul  
  Div  
)
```

```
type WorkReq struct {  
  Op OpType  
  Val1 int  
  Val2 int  
}
```

```
type WorkResp struct {  
  Wr *WorkReq  
  Result int  
  Err error  
}
```

```
func Worker(ctxt context.Context, in chan *WorkReq,  
           out chan *WorkResp) {  
  for {  
    select {  
      case <- ctxt.Done():  
        return  
      case job := <- in:  
        out <- Calc(wr) //Obvious calculation  
    }  
  }  
}
```

# Example — Workers & Contexts

```
type OpType = int
```

```
const (  
  Add op = iota  
  Sub  
  Mul  
  Div  
)
```

```
type WorkReq struct {  
  Op OpType  
  Val1 int  
  Val2 int  
}
```

```
type WorkResp struct {  
  Wr *WorkReq  
  Result int  
  Err error  
}
```

```
func Worker(ctxt context.Context, in chan *WorkReq,  
           out chan *WorkResp) {  
  for {  
    select {  
      case <- ctxt.Done():  
        return  
      case job := <- in:  
        out <- Calc(wr) //Obvious calculation  
    }  
  }  
}
```

# Example — Workers & Contexts

```
func Producer(ctxt context.Context,  
             in chan *WorkReq) {  
    for {  
        select {  
            case <- ctxt.Done():  
                return  
            case in <- ...:  
        }  
    }  
}
```

```
func main() {  
    in := make(chan *WorkReq, 10)  
    out := make(chan *WorkResp, 10)  
    ctxt := context.Background()  
    ctxt, cancel := context.WithCancel(ctxt)  
    go Processor(ctxt, in, out)  
    go Producer(ctxt, in)  
    timeout := time.After(5*time.Second)  
    for {  
        select {  
            case ans := <- out:  
                fmt.Printf(...)  
            case <- timeout:  
                cancel()  
                break  
        }  
    }  
}
```

# Example — Workers & Contexts

```
func Producer(ctxt context.Context,  
             in chan *WorkReq) {  
    for {  
        select {  
            case <- ctxt.Done():  
                return  
            case in <- ...:  
        }  
    }  
}
```

```
func main() {  
    in := make(chan *WorkReq, 10)  
    out := make(chan *WorkResp, 10)  
    ctxt := context.Background()  
    ctxt, cancel := context.WithCancel(ctxt)  
    go Processor(ctxt, in, out)  
    go Producer(ctxt, in)  
    timeout := time.After(5*time.Second)  
    for {  
        select {  
            case ans := <- out:  
                fmt.Printf(...)  
            case <- timeout:  
                cancel()  
                break  
        }  
    }  
}
```

# Example — Workers & Contexts

```
func Producer(ctxt context.Context,  
             in chan *WorkReq) {  
    for {  
        select {  
            case <- ctxt.Done():  
                return  
            case in <- ...:  
        }  
    }  
}
```

```
func main() {  
    in := make(chan *WorkReq,10)  
    out := make(chan *WorkResp,10)  
    ctxt := context.Background()  
    ctxt,cancel := context.WithCancel(ctxt)  
    go Processor(ctxt,in,out)  
    go Producer(ctxt,in)  
    timeout := time.After(5*time.Second)  
    for {  
        select {  
            case ans := <- out:  
                fmt.Printf(...)  
            case <- timeout:  
                cancel()  
                break  
        }  
    }  
}
```

# Using Contexts

- All goroutines must `select` on the `Done()` channel.
- Note how `Producer` relies on `select` to peek if a worker is listening while also being prepared to cancel (this is known as a **mixed choice** in concurrency theory).

# Example — Search

```
func handleSearch(w http.ResponseWriter, req *http.Request) {  
    var (  
        ctx      context.Context  
        cancel   context.CancelFunc  
    )  
    timeout, err := time.ParseDuration(req.FormValue("timeout"))  
    if err == nil {  
        ctx, cancel = context.WithTimeout(context.Background(), timeout)  
    } else {  
        ctx, cancel = context.WithCancel(context.Background())  
    }  
    defer cancel()  
    query := req.FormValue("q")  
    start := time.Now()  
    results, err := google.Search(ctx, query)  
    elapsed := time.Since(start)  
    //Do something with results if search succeeds.  
}
```



# Example — Search

```
func handleSearch(w http.ResponseWriter, req *http.Request) {  
    var (  
        ctx      context.Context  
        cancel  context.CancelFunc  
    )  
    timeout, err := time.ParseDuration(req.FormValue("timeout"))  
    if err == nil {  
        ctx, cancel = context.WithTimeout(context.Background(), timeout)  
    } else {  
        ctx, cancel = context.WithCancel(context.Background())  
    }  
    defer cancel()  
    query := req.FormValue("q")  
    start := time.Now()  
    results, err := google.Search(ctx, query)  
    elapsed := time.Since(start)  
    //Do something with results if search succeeds.  
}
```

# Example — Search

```
func Search(ctx context.Context, query string) (Results, error) {  
    req, err := http.NewRequest("GET", "https://ajax.googleapis.com/ajax/  
services/search/web?v=1.0", nil)  
    if err != nil {  
        return nil, err  
    }  
    q := req.URL.Query()  
    q.Set("q", query)  
  
    req.URL.RawQuery = q.Encode()  
    var results Results  
    err = httpDo(ctx, req, func(resp *http.Response, err error) error {  
        if err != nil {  
            return err  
        }  
        defer resp.Body.Close()  
        // Parse the JSON search result and add to results  
    })  
    return results, err
```

# Example — Search

```
func Search(ctx context.Context, query string) (Results, error) {
    req, err := http.NewRequest("GET", "https://ajax.googleapis.com/ajax/
services/search/web?v=1.0", nil)
    if err != nil {
        return nil, err
    }
    q := req.URL.Query()
    q.Set("q", query)

    req.URL.RawQuery = q.Encode()
    var results Results
    err = httpDo(ctx, req, func(resp *http.Response, err error) error {
        if err != nil {
            return err
        }
        defer resp.Body.Close()
        // Parse the JSON search result and add to results
    })
    return results, err
}
```

# Example — Search

```
func httpDo(ctx context.Context, req *http.Request, f func(*http.Response,
error) error) error {
    // Run the HTTP request in a goroutine and pass the response to f.
    c := make(chan error, 1)
    req = req.WithContext(ctx)
    go func() { c <- f(http.DefaultClient.Do(req)) }()
    select {
    case <-ctx.Done():
        <-c // Wait for f to return.
        return ctx.Err()
    case err := <-c:
        return err
    }
}
```

- Discards request if ctx.Done() is closed before request is done.

# Example — Search

```
func httpDo(ctx context.Context, req *http.Request, f func(*http.Response,
error) error) error {
    // Run the HTTP request in a goroutine and pass the response to f.
    c := make(chan error, 1)
    req = req.WithContext(ctx)
    go func() { c <- f(http.DefaultClient.Do(req)) }()
    select {
    case <-ctx.Done():
        <-c // Wait for f to return.
        return ctx.Err()
    case err := <-c:
        return err
    }
}
```

- Discards request if ctx.Done() is closed before request is done.

# Contexts

- Useful when we need the ability to cancel or timeout subsets of goroutines and/or share multiple pieces of data across goroutines.
- Different subsets of goroutines can have different timeouts or cancellation conditions.
- Must ensure shared data is concurrency safe (e.g. copied).

# Replicating Requests

- In some scenarios, receiving a response ASAP is key.
- We can “trade” computational resources for responsiveness.
- Replicate the same request across  $N$  workers, take the first answer.

# Replicating Requests

```
func Worker(ctxt Context.context, id int, result chan<- int) {  
    c := make(chan int)  
    go func() { c <- query(id) }()  
    select {  
        case <- ctxt.Done():  
            return  
        case ans := <-c:  
            result <- ans  
    }  
}  
func main() {  
    ctxt := context.Background()  
    ctxt, cancel := context.WithCancel(ctxt)  
    result := make(chan int)  
    for i := 0; i<10; i++ {  
        go Worker(ctxt, i, result)  
    }  
    firstAnswer := <-result ; cancel(); fmt.Println(firstAnswer)  
}
```



# Replicating Requests

```
func Worker(ctxt Context.context, id int, result chan<- int) {  
    c := make(chan int)  
    go func() { c <- query(id) }()  
    select {  
        case <- ctxt.Done():  
            return  
        case ans := <-c:  
            result <- ans  
    }  
}  
func main() {  
    ctxt := context.Background()  
    ctxt, cancel := context.WithCancel(ctxt)  
    result := make(chan int)  
    for i := 0; i<10; i++ {  
        go Worker(ctxt, i, result)  
    }  
    firstAnswer := <-result ; cancel(); fmt.Println(firstAnswer)  
}
```

# Replicating Requests

```
func Worker(ctxt Context.context, id int, result chan<- int) {  
    c := make(chan int)  
    go func() { c <- query(id) }()  
    select {  
        case <- ctxt.Done():  
            return  
        case ans := <-c:  
            result <- ans  
    }  
}  
func main() {  
    ctxt := context.Background()  
    ctxt, cancel := context.WithCancel(ctxt)  
    result := make(chan int)  
    for i := 0; i<10; i++ {  
        go Worker(ctxt, i, result)  
    }  
    firstAnswer := <-result ; cancel(); fmt.Println(firstAnswer)  
}
```

# Replicating Requests

```
func main() {  
    ctxt := context.Background()  
    ctxt, cancel := context.WithCancel(ctxt)  
    ctxt, cancel := context.WithTimeout(5*time.Second)  
    result := make(chan int)  
    for i := 0; i<10; i++ {  
        go Worker(ctxt, i, result)  
    }  
    select {  
        case firstAnswer := <-result:  
            cancel()  
            fmt.Println(firstAnswer)  
        case <- time.After(ctxt.Deadline()):  
            fmt.Println("Everyone timed out.")  
    }  
}
```

# Replicating Requests

```
func main() {
    ctx := context.Background()
    ctx, cancel := context.WithCancel(ctx)
    ctx, cancel := context.WithTimeout(5*time.Second)
    result := make(chan int)
    for i := 0; i < 10; i++ {
        go Worker(ctx, i, result)
    }
    select {
        case firstAnswer := <-result:
            cancel()
            fmt.Println(firstAnswer)
        case <- time.After(ctx.Deadline()):
            fmt.Println("Everyone timed out.")
    }
}
```

# Replicating Requests

- Can set-up cancellation via a context.
- Can also use the context mechanism to set a timeout.
- Useful for e.g. external queries on multiple replicas.

# Worker Pools

- Dispatch long-running goroutines as workers.
- Workers can process work using a single channel, multiple channels or by using a stateful request structure.
- The matrix ops example was an instance of this pattern.

# Worker Pools

```
func Dispatch(numWorkers int) (context.CancelFunc, chan WorkReq, chan WorkResp) {  
    ctxt := context.Background()  
    ctxt, cancel := context.WithCancel(ctxt)  
    in := make(chan WorkReq, 10)  
    out := make(chan WorkResp, 10)  
  
    for i := 0; i < numWorkers; i++ {  
        go Worker(ctxt, i, in, out)  
    }  
    return cancel, in, out  
}
```

# Worker Pools

```
func Dispatch(numWorkers int) (context.CancelFunc, chan WorkReq, chan WorkResp) {  
    ctxt := context.Background()  
    ctxt, cancel := context.WithCancel(ctxt)  
    in := make(chan WorkReq, 10)  
    out := make(chan WorkResp, 10)  
  
    for i := 0; i < numWorkers; i++ {  
        go Worker(ctxt, i, in, out)  
    }  
    return cancel, in, out  
}
```



# Worker Pools

```
func Worker(ctxt context.Context, id int, in chan WorkReq, out chan WorkResp) {  
    for {  
        select {  
            case <- ctxt.Done():  
                return  
            case req := <-in:  
                fmt.Printf("worker %d doing work of type %d\n", id, req.Op)  
                out <- Process(req)  
        }  
    }  
}
```

- Using tagged messages, workers can handle many types of work.

# Worker Pools

```
type WorkReq struct {  
    Op op  
    Text []byte  
    Compare []byte  
}  
  
type WorkResp struct {  
    Wr WorkReq  
    Result []byte  
    Matched bool  
    Err error  
}
```

```
func Process(wr WorkReq) WorkResp {  
    switch wr.Op {  
    case Hash:  
        return hashWork(wr)  
    case Compare:  
        return compareWork(wr)  
    default:  
        return WorkResp{Err:errors.New("Bad opcode")}  
    }  
}  
  
func hashWork(wr WorkReq) WorkResp {  
    val, err := bcrypt.GenerateFromPassword(wr.Text,  
        bcrypt.DefaultCost)  
    return WorkResp{Result:val,Err:err,Wr:wr}  
}  
  
func compareWork(wr WorkReq) WorkResp {  
    err := bcrypt.CompareHashAndPassword(wr.Compare,wr.Text)  
    return WorkResp{Matched:err==nil,Err:err,Wr:wr}  
}
```

# Worker Pools

```
type WorkReq struct {  
    Op op  
    Text []byte  
    Compare []byte  
}  
  
type WorkResp struct {  
    Wr WorkReq  
    Result []byte  
    Matched bool  
    Err error  
}
```

```
func Process(wr WorkReq) WorkResp {  
    switch wr.Op {  
    case Hash:  
        return hashWork(wr)  
    case Compare:  
        return compareWork(wr)  
    default:  
        return WorkResp{Err:errors.New("Bad opcode")}  
    }  
}  
  
func hashWork(wr WorkReq) WorkResp {  
    val, err := bcrypt.GenerateFromPassword(wr.Text,  
        bcrypt.DefaultCost)  
    return WorkResp{Result:val,Err:err,Wr:wr}  
}  
  
func compareWork(wr WorkReq) WorkResp {  
    err := bcrypt.CompareHashAndPassword(wr.Compare,wr.Text)  
    return WorkResp{Matched:err==nil,Err:err,Wr:wr}  
}
```

# Worker Pools

- Can control maximum concurrency via worker pool size.
- One pool per task category (direct tasks accordingly).
- Can also specialize workers (as in the Matrix examples).
- Useful when spawning goroutines per request is CPU/  
Memory intensive.
- Remember that workers can be “richer” (multiple channels,  
have more state — have an internal queue, etc).

# Pipelines

- We have already seen a simplified pipeline in the prime sieve example.
- All “workers” (sieves) were identical.
- Could only spawn more workers “sequentially”.

# Pipelines

- A more general pattern:
  - Two stage pipeline (could further generalize to N stage).
  - Each stage is a pool of specialized workers.
  - Workers in one stage feed those in the next (via shared channels).

# Pipelines

```
type Worker struct {  
    in chan string  
    out chan string  
}
```

```
type Job = int  
const (  
    Print Job = iota  
    Encode  
)
```

```
func (w *Worker) Work(ctxt context.Context, j Job) {  
    switch j {  
        case Print:  
            w.Print(ctxt)  
        case Encode:  
            w.Encode(ctxt)  
        default:  
            return  
    }  
}  
  
func (w *Worker) Print(ctxt context.Context) {  
    for {  
        select {  
            case <- ctxt.Done():  
                return  
            case val := <- w.in: fmt.Println(val); w.out <- val  
        }  
    }  
}
```

# Pipelines

```
type Worker struct {  
    in chan string  
    out chan string  
}
```

```
type Job = int  
const (  
    Print Job = iota  
    Encode  
)
```

```
func (w *Worker) Work(ctxt context.Context, j Job) {  
    switch j {  
        case Print:  
            w.Print(ctxt)  
        case Encode:  
            w.Encode(ctxt)  
        default:  
            return  
    }  
}
```

```
func (w *Worker) Print(ctxt context.Context) {  
    for {  
        select {  
            case <- ctxt.Done():  
                return  
            case val := <- w.in: fmt.Println(val); w.out <- val  
        }  
    }  
}
```



# Pipelines

```
type Worker struct {  
    in chan string  
    out chan string  
}
```

```
type Job = int  
const (  
    Print Job = iota  
    Encode  
)
```

```
func (w *Worker) Work(ctxt context.Context, j Job) {  
    switch j {  
        case Print:  
            w.Print(ctxt)  
        case Encode:  
            w.Encode(ctxt)  
        default:  
            return  
    }  
}  
  
func (w *Worker) Print(ctxt context.Context) {  
    for {  
        select {  
            case <- ctxt.Done():  
                return  
            case val := <- w.in: fmt.Println(val); w.out <- val  
        }  
    }  
}
```

# Pipelines

```
//Base64 string encoding
func (w *Worker) Encode(ctx context.Context) {
    for {
        select {
            case <- ctx.Done():
                return
            case val := <- w.in:
                w.out <- fmt.Sprintf("%s => %s", val,
                    base64.StdEncoding.EncodeToString([]byte(val)))
        }
    }
}
```

# Pipelines

```
func NewPipeline(ctxt context.Context, numEncoders, numPrinters int)
    (chan string, chan string) {
    inEncode := make(chan string, numEncoders)
    inPrint := make(chan string, numPrinters)
    outPrint := make(chan string, numPrinters)
    for i := 0; i < numEncoders; i++ {
        w := Worker{in: inEncode, out: inPrint}
        go w.Work(ctxt, Encode)
    }
    for i := 0; i < numPrinters; i++ {
        w := Worker{in: inPrint, out: outPrint}
        go w.Work(ctxt, Print)
    }
    return inEncode, outPrint
}
```

# Pipelines

```
func main() {  
    ctxt := context.Background()  
    ctxt, cancel := context.WithCancel(ctxt)  
    defer cancel()  
    in, out := NewPipeline(ctxt, 20,5)  
  
    go func() {  
        for i := 0 ; i < 20; i++ {  
            in <- fmt.Sprintf("Message", i)  
        }  
    }()  
  
    for i := 0; i < 20; i++ {  
        <-out  
    }  
}
```

# Pipelines

- The two pools are “wired” by the NewPipeline function.
- One pool feeds the next, leveraging pipeline width.
- Main fans-out work to the Encoder pool (as much as the **width** of the pipeline allows).
- Main fans-in answers from the Printer pool.
- **Variant:** Workers have an array of in and out channels, or a map of channels (e.g. multiple outputs at each step).

# Rate Limiting

- Common in distributed settings (limit attack potential, control performance degradation due to load, etc.)
- Most rate limiting is done by some variant of the *token bucket* algorithm.
- Basic idea: To use a resource, an access token is needed.
- Tokens are stored in (and taken from) a “bucket” of depth  $d$ .
- If no tokens left, we have to wait.

# Rate Limiting

- How are tokens replenished? A replenishing rate defines the rate at which tokens are added back into the bucket.
- Lets mock up these ideas...

# Rate Limiting

```
type APIConnection struct {}

func Open() *APIConnection {
    return &APIConnection{}
}

func (a *APIConnection) ReadData(ctxt context.Context) error {
    return nil
}

func (a *APIConnection) PerformComp(ctxt context.Context) error {
    return nil
}
```

- Some abstract service offering two operations.



# Rate Limiting

- Package `golang.org/x/time/rate` implements a token bucket rate limiter.
- **type** `Limit` defines the max. frequency of events.
- **func** `newLimiter(r Limit, b int) *Limiter` returns a limiter allowing events up to rate `r` and bursts of at most `b` tokens.
- **func** `Every(interval time.Duration) Limit` converts durations into limits (min. interval between events).

# Rate Limiting

- With a Limiter, we can block requests until a token is available.
- **func** (lim \*Limiter) WaitN(ctxt context.Context, n **int**) error  
blocks until n events can happen. Errors if  $n > b$  or if ctxt demands.
- **func** (lim \*Limiter) Wait(ctxt context.Context) error

# Rate Limiting

```
type APIConnection struct { rateLimiter *rate.Limiter }
func Open() *APIConnection {
    return &APIConnection{rate.NewLimiter(rate.Limit(1),1)}
}
func (a *APIConnection) ReadData(ctxt context.Context) error {
    if err := a.rateLimiter.Wait(ctxt); err != nil {
        return err
    }
    return nil
}
func (a *APIConnection) PerformComp(ctxt context.Context) error {
    if err := a.rateLimiter.Wait(ctxt); err != nil {
        return err
    }
    return nil
}
```

- One event per second allowed.

# Rate Limiting

- Often we want more sophisticated limits:
  - 2 times per second, no burst + 5 per minute, 5 burst.
  - Limit per class of operation.
  - Etc.

# Rate Limiting

```
type RateLimiter interface {  
    Wait(context.Context) error  
    Limit() rate.Limit  
}
```

```
type multiLimiter struct {  
    limiters []RateLimiter  
}
```

```
func MultiLimiter(limiters ...RateLimiter) *multiLimiter {  
    byLimit := func(i,j int) bool {  
        return limiters[i].Limit() < limiters[j].Limit()  
    }  
    sort.Slice(limiters, byLimit)  
    return &multiLimiter{limiters}  
}
```

```
func Per(eventCount int, duration time.Duration) rate.Limit { //Frequency  
    return rate.Every(duration/time.Duration(eventCount)) }
```

- Sort by Limit, most restrictive first.

# Rate Limiting

```
type RateLimiter interface {  
    Wait(context.Context) error  
    Limit() rate.Limit  
}
```

```
type multiLimiter struct {  
    limiters []RateLimiter  
}
```

```
func MultiLimiter(limiters ...RateLimiter) *multiLimiter {  
    byLimit := func(i,j int) bool {  
        return limiters[i].Limit() < limiters[j].Limit()  
    }  
    sort.Slice(limiters, byLimit)  
    return &multiLimiter{limiters}  
}
```

```
func Per(eventCount int, duration time.Duration) rate.Limit { //Frequency  
    return rate.Every(duration/time.Duration(eventCount)) }
```

- Sort by Limit, most restrictive first.

# Rate Limiting

```
type RateLimiter interface {  
    Wait(context.Context) error  
    Limit() rate.Limit  
}
```

```
type multiLimiter struct {  
    limiters []RateLimiter  
}
```

```
func MultiLimiter(limiters ...RateLimiter) *multiLimiter {  
    byLimit := func(i,j int) bool {  
        return limiters[i].Limit() < limiters[j].Limit()  
    }  
    sort.Slice(limiters, byLimit)  
    return &multiLimiter{limiters}  
}
```

```
func Per(eventCount int, duration time.Duration) rate.Limit { //Frequency  
    return rate.Every(duration/time.Duration(eventCount)) }
```

- Sort by Limit, most restrictive first.

# Rate Limiting

```
type RateLimiter interface {  
    Wait(context.Context) error  
    Limit() rate.Limit  
}
```

```
type multiLimiter struct {  
    limiters []RateLimiter  
}
```

```
func MultiLimiter(limiters ...RateLimiter) *multiLimiter {  
    byLimit := func(i,j int) bool {  
        return limiters[i].Limit() < limiters[j].Limit()  
    }  
    sort.Slice(limiters, byLimit)  
    return &multiLimiter{limiters}  
}
```

```
func Per(eventCount int, duration time.Duration) rate.Limit { //Frequency  
    return rate.Every(duration/time.Duration(eventCount)) }
```

- Sort by Limit, most restrictive first.



# Rate Limiting

```
type RateLimiter interface {  
    Wait(context.Context) error  
    Limit() rate.Limit  
}
```

```
type multiLimiter struct {  
    limiters []RateLimiter  
}
```

```
func (l *MultiLimiter) Wait(ctxt context.Context) error {  
    for _, l := range l.limiters {  
        if err := l.Wait(ctxt); err != nil {  
            return err  
        }  
    }  
    return nil  
}
```

```
func (l *MultiLimiter) Limit() rate.Limit { return l.limiters[0].Limit() }
```

- Wait for exactly the time of the longest wait. Why?

# Rate Limiting

```
type RateLimiter interface {  
    Wait(context.Context) error  
    Limit() rate.Limit  
}
```

```
type multiLimiter struct {  
    limiters []RateLimiter  
}
```

```
func (l *MultiLimiter) Wait(ctxt context.Context) error {  
    for _, l := range l.limiters {  
        if err := l.Wait(ctxt); err != nil {  
            return err  
        }  
    }  
    return nil  
}
```

```
func (l *MultiLimiter) Limit() rate.Limit { return l.limiters[0].Limit() }
```

- Wait for exactly the time of the longest wait. Why?

# Rate Limiting

```
type APIConnection struct { rateLimiter RateLimiter }  
func Open() *APIConnection {  
    second := rate.NewLimiter(Per(2, time.Second), 1)  
    minute := rate.NewLimiter(Per(5, time.Minute), 5)  
    return &APIConnection{MultiLimiter(second,minute)}  
}  
func (a *APIConnection) ReadData(ctxt context.Context) error {  
    //As before...  
}  
func (a *APIConnection) PerformComp(ctxt context.Context) error {  
    //As before...  
}
```

# Rate Limiting

```
type APIConnection struct { rateLimiter RateLimiter }  
func Open() *APIConnection {  
    second := rate.NewLimiter(Per(2, time.Second), 1)  
    minute := rate.NewLimiter(Per(5, time.Minute), 5)  
    return &APIConnection{MultiLimiter(second,minute)}  
}  
func (a *APIConnection) ReadData(ctxt context.Context) error {  
    //As before...  
}  
func (a *APIConnection) PerformComp(ctxt context.Context) error {  
    //As before...  
}
```

- Can set up operation-based limits on top of this. How?

# Rate Limiting

```
type APIConnection struct { apiLimiter,readLimiter RateLimiter }  
func Open() *APIConnection {  
    second := rate.NewLimiter(Per(2, time.Second), 1)  
    minute := rate.NewLimiter(Per(5, time.Minute), 5)  
    read := rate.NewLimiter(Per(3, time.Second), 3)  
    return &APIConnection{apiLimiter:MultiLimiter(second,minute)  
                        readLimiter:MultiLimiter(read)}  
}  
func (a *APIConnection) ReadData(ctxt context.Context) error {  
    err := MultiLimiter(a.apiLimiter,a.readLimiter).Wait(ctxt)  
    if err != nil {  
        return err  
    }  
    return nil  
}
```

# Rate Limiting

```
type APIConnection struct { apiLimiter,readLimiter RateLimiter }  
func Open() *APIConnection {  
    second := rate.NewLimiter(Per(2, time.Second), 1)  
    minute := rate.NewLimiter(Per(5, time.Minute), 5)  
    read := rate.NewLimiter(Per(3, time.Second), 3)  
    return &APIConnection{apiLimiter:MultiLimiter(second,minute)  
                        readLimiter:MultiLimiter(read)}  
}  
func (a *APIConnection) ReadData(ctxt context.Context) error {  
    err := MultiLimiter(a.apiLimiter,a.readLimiter).Wait(ctxt)  
    if err != nil {  
        return err  
    }  
    return nil  
}
```

# Rate Limiting

- We can compose logical rate limiters into groups that make sense for each call / group of calls.
- Composite limiter idea is a nifty use of Go's structural typing.
- `rate.Limiters` are also `RateLimiters`.

# Summary

- We have seen how to solve quite a few high-level problems:
  - Disciplined cancellation and timeouts of (subtrees) processes.
  - Request replication for performance/responsiveness.
  - Process (various kinds of) work concurrently with worker pools.
  - Process work in stages, leveraging concurrency, via pipelines.
  - Explore a uniform way of rate limiting processes for fairness/avoid DoS.



# Summary

- A lot of room for useful variations / combinations:
  - Pipelines where stages read/write from multiple channels (e.g. array of channels).
  - Workers that maintain internal state (e.g. a work queue).
  - Pipeline width as explicit “pool of pipelines”.
  - Rate limiter as a “gate” to a pipeline/worker pool.