

Designing Concurrent Components

Design Philosophy

Goal: Build components that are concurrent by nature, not retrofitted.

Key principles:

1. **Single responsibility:** Each component does one thing
2. **Clear boundaries:** Explicit inputs/outputs
3. **Independent state:** Minimal sharing
4. **Explicit coordination:** Visible synchronization

Decomposition Strategies

Strategy 1: Pipeline Decomposition

Break processing into sequential stages.

```
// Stage-based decomposition
type Pipeline struct {
    // Stage 1: Input
    input chan Request

    // Stage 2: Validation
    validated chan Request

    // Stage 3: Processing
    processed chan Result

    // Stage 4: Storage
    stored chan Result
}

func NewPipeline() *Pipeline {
    p := &Pipeline{
        input:    make(chan Request, 100),
        validated: make(chan Request, 100),
        processed: make(chan Result, 100),
        stored:    make(chan Result, 100),
    }

    // Each stage runs independently
    go p.validateStage()
    go p.processStage()
    go p.storageStage()

    return p
}

func (p *Pipeline) validateStage() {
    for req := range p.input {
```

```

        if req.IsValid() {
            p.validated <- req
        }
    }
    close(p.validated)
}

func (p *Pipeline) processStage() {
    for req := range p.validated {
        result := process(req)
        p.processed <- result
    }
    close(p.processed)
}

func (p *Pipeline) storageStage() {
    for result := range p.processed {
        store(result)
        p.stored <- result
    }
    close(p.stored)
}

```

When to use:

- Sequential transformations
- Each stage has different performance characteristics
- Want to scale stages independently

Strategy 2: Worker Pool Decomposition

Distribute work across parallel workers.

```

type WorkerPool struct {
    tasks    chan Task
    results  chan Result
    workers  int
    wg       sync.WaitGroup
}

func NewWorkerPool(workers int) *WorkerPool {
    wp := &WorkerPool{
        tasks:    make(chan Task, workers*2),
        results:  make(chan Result, workers*2),
        workers:  workers,
    }

    // Start workers
    for i := 0; i < workers; i++ {
        wp.wg.Add(1)
        go wp.worker(i)
    }
}

```

```

    return wp
}

func (wp *WorkerPool) worker(id int) {
    defer wp.wg.Done()

    for task := range wp.tasks {
        result := task.Execute()
        wp.results <- result
    }
}

func (wp *WorkerPool) Submit(task Task) {
    wp.tasks <- task
}

func (wp *WorkerPool) Shutdown() {
    close(wp.tasks)
    wp.wg.Wait()
    close(wp.results)
}

```

When to use:

- CPU-bound work
- Independent tasks
- Want to control parallelism

Strategy 3: Actor Model Decomposition

Each component is an independent actor with mailbox.

```

type Actor struct {
    mailbox chan Message
    state   interface{}
    quit    chan struct{}
}

func NewActor() *Actor {
    a := &Actor{
        mailbox: make(chan Message, 100),
        quit:    make(chan struct{}),
    }

    go a.run()
    return a
}

func (a *Actor) run() {
    for {
        select {

```

```

        case msg := <-a.mailbox:
            a.handle(msg)
        case <-a.quit:
            return
    }
}

func (a *Actor) Send(msg Message) {
    a.mailbox <- msg
}

func (a *Actor) handle(msg Message) {
    // Update internal state
    // Send messages to other actors
}

```

When to use:

- Stateful components
- Complex interactions
- Want location transparency (can move to another process/machine)

Strategy 4: Event-Driven Decomposition

Components react to events.

```

type EventBus struct {
    subscribers map[EventType][]chan Event
    mu          sync.RWMutex
}

func NewEventBus() *EventBus {
    return &EventBus{
        subscribers: make(map[EventType][]chan Event),
    }
}

func (eb *EventBus) Subscribe(et EventType) <-chan Event {
    eb.mu.Lock()
    defer eb.mu.Unlock()

    ch := make(chan Event, 10)
    eb.subscribers[et] = append(eb.subscribers[et], ch)
    return ch
}

func (eb *EventBus) Publish(event Event) {
    eb.mu.RLock()
    defer eb.mu.RUnlock()

    for _, ch := range eb.subscribers[event.Type] {

```

```

        select {
        case ch <- event:
        default:
            // Subscriber slow, drop event
        }
    }
}

// Component using event bus
type OrderProcessor struct {
    events <-chan Event
}

func NewOrderProcessor(bus *EventBus) *OrderProcessor {
    return &OrderProcessor{
        events: bus.Subscribe(OrderCreated),
    }
}

func (op *OrderProcessor) Run(ctx context.Context) {
    for {
        select {
        case event := <-op.events:
            op.process(event)
        case <-ctx.Done():
            return
        }
    }
}

```

When to use:

- Loosely coupled components
- Publish-subscribe patterns
- Event sourcing

State Management

Pattern 1: No Shared State

Best: Each goroutine owns its state.

```

type Worker struct {
    // Own state, no sharing
    id      int
    cache   map[string]string
    requests chan Request
}

func (w *Worker) Run() {
    for req := range w.requests {
        // Access cache without locks
    }
}

```

```

        result := w.cache[req.Key]
        req.Response <- result
    }
}

```

Pattern 2: Immutable State

Good: Share read-only data.

```

type Config struct {
    // Immutable after creation
    Timeout time.Duration
    MaxRetry int
    Endpoints []string
}

// Safe to share across goroutines
var globalConfig *Config

func UpdateConfig(newConfig *Config) {
    // Atomic pointer swap
    atomic.StorePointer((*unsafe.Pointer)(unsafe.Pointer(&globalConfig)),
        unsafe.Pointer(newConfig))
}

```

Pattern 3: Message Passing

Good: Communicate by sending copies.

```

type Cache struct {
    data map[string]string
    ops  chan CacheOp
}

type CacheOp struct {
    Type    OpType
    Key     string
    Value   string
    Response chan string
}

func (c *Cache) Run() {
    for op := range c.ops {
        switch op.Type {
        case Get:
            op.Response <- c.data[op.Key]
        case Set:
            c.data[op.Key] = op.Value
        }
    }
}

```

```
func (c *Cache) Get(key string) string {
    resp := make(chan string)
    c.ops <- CacheOp{Type: Get, Key: key, Response: resp}
    return <-resp
}
```

Pattern 4: Atomic Operations

OK: For simple counters/flags.

```
type Counter struct {
    value int64
}

func (c *Counter) Inc() {
    atomic.AddInt64(&c.value, 1)
}

func (c *Counter) Get() int64 {
    return atomic.LoadInt64(&c.value)
}
```

Pattern 5: Protected State (Last Resort)

Use sparingly: Mutexes for complex shared state.

```
type SharedCache struct {
    mu    sync.RWMutex
    data  map[string]string
}

func (sc *SharedCache) Get(key string) string {
    sc.mu.RLock()
    defer sc.mu.RUnlock()
    return sc.data[key]
}

func (sc *SharedCache) Set(key, value string) {
    sc.mu.Lock()
    defer sc.mu.Unlock()
    sc.data[key] = value
}
```

Component Interface Design

Principle 1: Accept Channels

```
// BAD: Exposes internals
type Processor struct {
    Input []Request
    mu     sync.Mutex
}

func (p *Processor) AddRequest(req Request) {
    p.mu.Lock()
    p.Input = append(p.Input, req)
    p.mu.Unlock()
}

// GOOD: Clean interface
type Processor struct {
    input chan Request
}

func (p *Processor) Submit(req Request) {
    p.input <- req
}
```

Principle 2: Pass Context

```
// GOOD: Cancellable operations
func (p *Processor) Process(ctx context.Context, req Request) error {
    select {
    case result := <-p.process(req):
        return result
    case <-ctx.Done():
        return ctx.Err()
    }
}
```

Principle 3: Return Closing Channels

```
// GOOD: Indicate completion
func (p *Processor) Results() <-chan Result {
    return p.results // Read-only channel
}

// Consumer knows when done
for result := range processor.Results() {
    handle(result)
}
// Channel closed = processing complete
```

Principle 4: Provide Shutdown

```

type Component interface {
    Start() error
    Shutdown(ctx context.Context) error
}

func (p *Processor) Shutdown(ctx context.Context) error {
    close(p.quit) // Signal stop

    // Wait for graceful shutdown
    select {
    case <-p.done:
        return nil
    case <-ctx.Done():
        return ctx.Err() // Timeout
    }
}

```

Real Example: HTTP Request Processor

```

type HTTPRequestProcessor struct {
    // Inputs
    requests chan Request

    // Internal stages
    validated chan Request
    fetched   chan *http.Response
    parsed    chan Result

    // Output
    results chan Result

    // Control
    ctx      context.Context
    cancel    context.CancelFunc
    wg       sync.WaitGroup
}

func NewHTTPRequestProcessor(ctx context.Context) *HTTPRequestProcessor {
    ctx, cancel := context.WithCancel(ctx)

    p := &HTTPRequestProcessor{
        requests: make(chan Request, 100),
        validated: make(chan Request, 100),
        fetched:   make(chan *http.Response, 100),
        parsed:    make(chan Result, 100),
        results:   make(chan Result, 100),
        ctx:       ctx,
        cancel:    cancel,
    }
}

```

```

    // Start pipeline stages
    p.wg.Add(4)
    go p.validateStage()
    go p.fetchStage()
    go p.parseStage()
    go p.outputStage()

    return p
}

func (p *HTTPRequestProcessor) validateStage() {
    defer p.wg.Done()
    defer close(p.validated)

    for {
        select {
        case req, ok := <-p.requests:
            if !ok {
                return
            }
            if req.IsValid() {
                p.validated <- req
            }
        case <-p.ctx.Done():
            return
        }
    }
}

func (p *HTTPRequestProcessor) fetchStage() {
    defer p.wg.Done()
    defer close(p.fetched)

    // Parallel fetching with worker pool
    sem := make(chan struct{}, 10) // Max 10 concurrent

    for req := range p.validated {
        select {
        case sem <- struct{}{}:
            go func(req Request) {
                defer func() { <-sem }()

                resp, err := http.Get(req.URL)
                if err == nil {
                    p.fetched <- resp
                }
            }(req)
        case <-p.ctx.Done():
            return
        }
    }
}

```

```

}

func (p *HTTPRequestProcessor) parseStage() {
    defer p.wg.Done()
    defer close(p.parsed)

    for resp := range p.fetched {
        result := parse(resp)
        p.parsed <- result
    }
}

func (p *HTTPRequestProcessor) outputStage() {
    defer p.wg.Done()
    defer close(p.results)

    for result := range p.parsed {
        select {
        case p.results <- result:
        case <-p.ctx.Done():
            return
        }
    }
}

// Public API
func (p *HTTPRequestProcessor) Submit(req Request) error {
    select {
    case p.requests <- req:
        return nil
    case <-p.ctx.Done():
        return p.ctx.Err()
    }
}

func (p *HTTPRequestProcessor) Results() <-chan Result {
    return p.results
}

func (p *HTTPRequestProcessor) Shutdown(timeout time.Duration) error {
    close(p.requests) // No more inputs

    done := make(chan struct{})
    go func() {
        p.wg.Wait()
        close(done)
    }()

    select {
    case <-done:
        return nil
    case <-time.After(timeout):

```

```

        p.cancel() // Force shutdown
        return fmt.Errorf("shutdown timeout")
    }
}

```

Component Testing

```

func TestHTTPRequestProcessor(t *testing.T) {
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    processor := NewHTTPRequestProcessor(ctx)

    // Submit requests
    for i := 0; i < 10; i++ {
        processor.Submit(Request{URL: fmt.Sprintf("http://example.com/%d", i)})
    }

    // Collect results
    var results []Result
    go func() {
        for result := range processor.Results() {
            results = append(results, result)
        }
    }()

    // Shutdown
    if err := processor.Shutdown(time.Second); err != nil {
        t.Fatal(err)
    }

    // Verify
    if len(results) != 10 {
        t.Errorf("Expected 10 results, got %d", len(results))
    }
}

```

Design Checklist

Component should:

- ☐ Have single clear responsibility
- ☐ Own its state (no shared mutable state)
- ☐ Accept context for cancellation
- ☐ Use channels for input/output
- ☐ Provide graceful shutdown
- ☐ Be independently testable
- ☐ Have clear error handling

- ☐ Avoid blocking operations without timeout
- ☐ Scale horizontally (add more instances)
- ☐ Monitor health (expose metrics)

Interview Questions

Q: "How do you decompose a system into concurrent components?"

"1) Identify stages: Sequential steps → pipeline. 2) Find parallelizable work → worker pool. 3) Look for independent state → actors. 4) Check for event reactions → event-driven. Choose based on: Pipeline for transformations, workers for CPU-bound tasks, actors for stateful components, events for loose coupling. Each component should own its state, communicate via channels, and provide graceful shutdown."

Q: "Channel vs mutex for component communication?"

"Prefer channels for: 1) Cross-goroutine communication (cleaner than mutex+cond). 2) Natural flow control (backpressure with buffer). 3) Clear ownership (sender produces, receiver consumes). Use mutex for: 1) Protecting in-memory data structure. 2) High-frequency updates (atomic counters). 3) Read-heavy workloads (RWMutex). Guideline: Channels for communication, mutexes for protecting data."

Q: "How to handle component failures?"

"1) Error propagation: Return errors through channels or callbacks. 2) Supervision: Parent monitors child, restarts on failure. 3) Circuit breaker: Stop calling failing component. 4) Timeout: Don't wait forever on broken component. 5) Health checks: Periodic pings. 6) Graceful degradation: Operate with reduced functionality. Example: If cache fails, bypass and hit database directly."

Q: "What makes a good concurrent API?"

"1) Accept context for cancellation. 2) Use channels for async results. 3) Provide blocking and non-blocking variants (Send vs TrySend). 4) Return read-only channels (<-chan). 5) Offer graceful shutdown with timeout. 6) Hide internal mutexes. 7) Document goroutine safety. Example: processor.Submit(ctx, req) <-chan Result with processor.Shutdown(timeout)."

Key Takeaways

1. Design for concurrency from start, not retrofit
2. Each component owns its state
3. Use channels for cross-component communication
4. Decompose by stages, workers, actors, or events
5. Always provide graceful shutdown
6. Pass context for cancellation
7. Return read-only channels
8. Test components independently
9. No shared mutable state
10. Clear boundaries, explicit coordination

Exercises

1. Design rate limiter with pipeline decomposition.
2. Build actor-based chat system with user actors.
3. Create event-driven order processing system.

4. Implement worker pool with dynamic scaling.

5. Design component with graceful shutdown under load.

Next: [concurrency-boundaries.md](#) - Where to introduce concurrency.