Concurrency in Go

Nurali Virani
Gojek

Agenda

- Concurrency Constructs
  - goroutine
  - channel
  - select
- Concurrency Pattern and Use-case
  - Generator
  - Fan-in
  - Fan-out
- Go Concurrency Simple and Powerful
  - Idiom 1: Don't communicate by sharing memory, share memory by communicating.
  - Idiom 2: Concurrency is NOT parallelism.
  - Daisy chain
- Q & A

goroutine

- It's an independently executing function, launched by a go statement.
- They're a bit like threads, but they're much cheaper.
- It has its own call stack, which grows and shrinks as required.
- Goroutines are multiplexed onto OS threads as required.

```
func main()
    f("hello", "world") // f runs; main wait
```

```
func main()
    go f("hello", "world") // f starts running
    // main continues, does not wait for f to return
```

```
go func() { // anonymous function
   fmt.Print("hello world")
}()
```

channel

- A channel in Go provides a connection between two goroutines, allowing them to communicate and synchronize.
- Go channels can also be created with a buffer .. Buffering removes synchronization.

```
// Declaring and initializing.
var c chan int
c = make(chan int)
```

```
// or
c := make(chan int)
```

```
// Sending on a channel.
c <- 1
```

```
// Receiving from a channel.
// The "arrow" indicates the direction of data flow.
value = <-c
```

channel - first class values

- channels are first-class values.

```
// channel as input and out.
func work(data chan int) (result chan int)
```

```
type Worker struct {
    data chan int  // channel as struct element
}
```

```
// channel data can be struct
ch := make(chan Worker)
```

```
// channel data can be another channel
ch := make(chan chan int)
```

```
// channel data can be another channel
ch := make(chan func(int))
```

select

It's like a switch, but each case is a communication.

- All channels are evaluated.
- Selection blocks until one communication can proceed.
- If multiple can proceed, select chooses pseudo-randomly.
- default clause, if present, executes immediately if no channel is ready.

```
select {
case msg := <-ch1:
    fmt.Println("from ch1,", msg)
case msg := <-ch2:
    fmt.Println("from ch2,", msg)
default:
    fmt.Println("from default")
}
```

Concurrency Construct Summary

goroutines - concurrent execution

channels - synchronization and communication

select - multi-way concurrent control

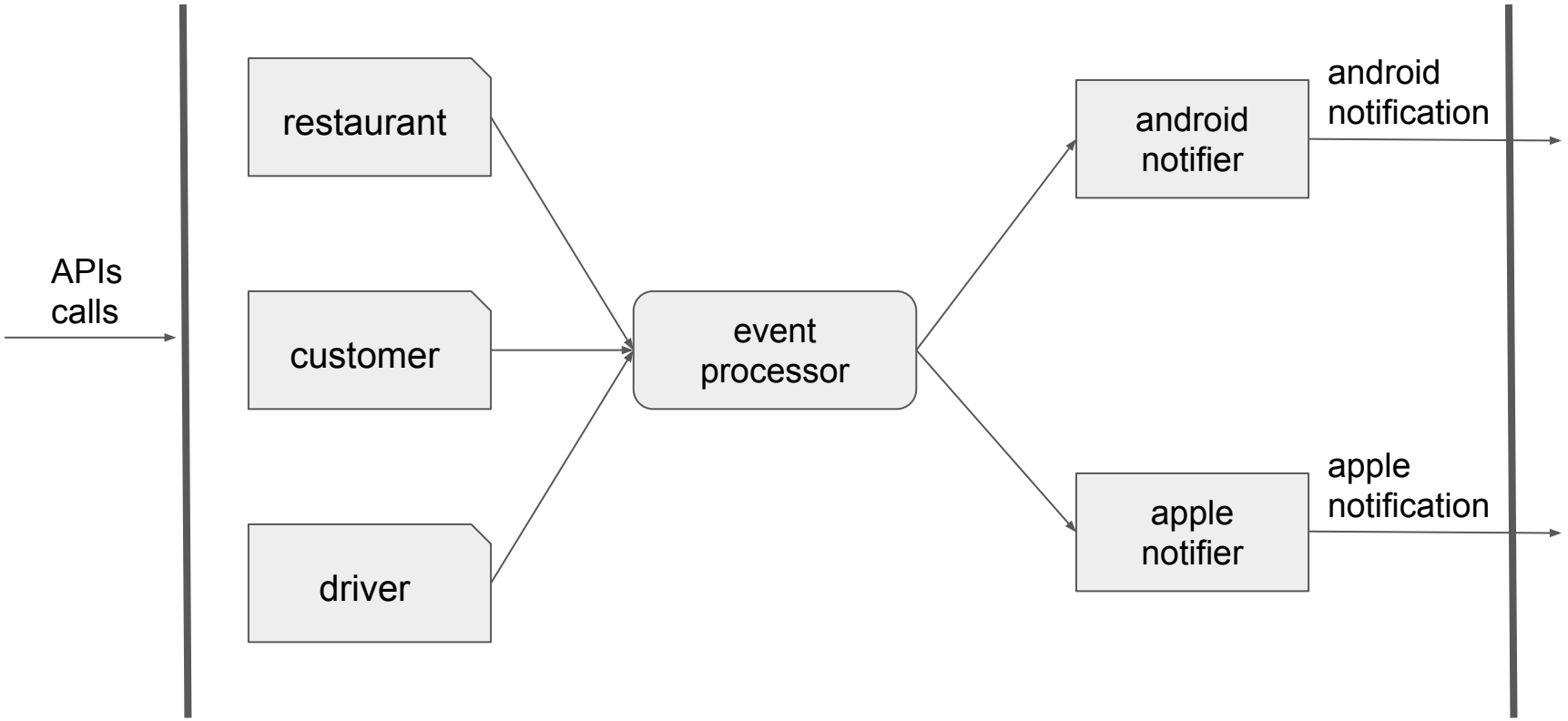
Concurrency Pattern

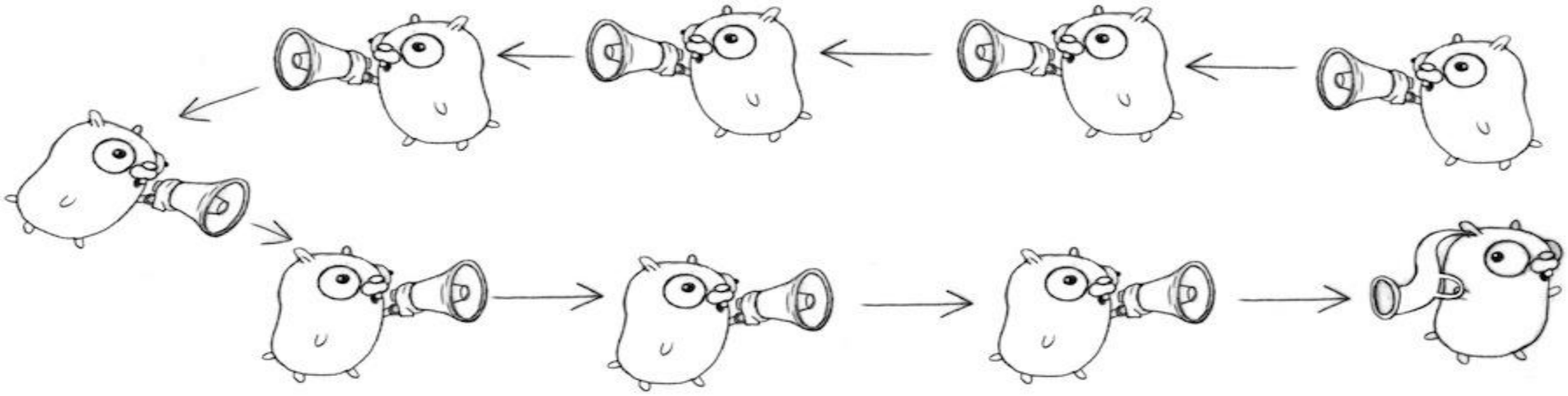
generator / producer - function that returns a channel

fanIn / aggregator - receive data from multiple channel and send all data to one channel

fanOut - receive data from one channel and send to multiple channel

Concurrency Use-case





```
func f(left, right chan int) {
    left <- 1 + <-right
}

func main() {
    const n = 100000
    leftmost := make(chan int)
    right := leftmost
    left := leftmost
    for i := 0; i < n; i++ {
        right = make(chan int)
        go f(left, right)
        left = right
    }
    go func(c chan int) { c <- 1 }(right)
    fmt.Println(<-leftmost)
}
```

Concurrency topics not covered ..

- buffer channel
- deadlock
- channel close

Links & Resources

Concurrency is not Parallelism - by Rob Pike (link)

Go Concurrency Patterns - by Rob Pike (link)

Visualizing Concurrency in Go (link)

A tour of Go (link)

https://go.dev/ (start here for go developer)

http://gophervids.appspot.com/ (go talks, slides library)

https://github.com/nurali-techie/meetup-golang (my sessions)

https://www.meetup.com/Golang-Bangalore/ (please join meetup group)

Thank you

Nurali Virani
https://www.linkedin.com/in/nurali-techie/