GO in the Distributed Systems world

Alberto Giunta

Department of Computer Science
University of Bologna

October 30, 2017

Outline

Outline

Let's GO!

The What, Where and Why of GO

What

GO is a compiled, concurrent, garbage-collected, statically typed, open source programming language, which aims to be the *next* C.
GO is about language design in the service of software engineering.

Where and Why

Created at Google in 2007 in order to solve the problems they were facing with their *really* large distributed and computation-based architecture

Put in other words

"All three of the language designers cited their shared dislike of C++'s complexity as a primary motivation for designing a new language"[4]

Let's GO!

The What, Where and Why of GO

- It was built and designed with a few requirements in mind:
  - It must work at scale (as Java and C++)
  - It must be familiar, productive and readable
  - It must support networking and multiprocessing
- When implemented, it was decided to follow this path:
  - Start from C
  - Remove its complex parts
  - add first class support to solutions for modern problems (i.e. concurrency)

Outline

Let's GO!

PROs and CONs of GO

PROs:

- C family (imperative, braces)
- Statically typed
- Quick compilation times
- First-class support to concurrency
- Well suited language for networking projects, distributed functions or services
- Everything is a function

Let's GO!

PROs and CONs of GO

CONs:

- Lacks of generics
- No constructors
- No exceptions
- No annotations
- No classes (they can be achieved through the use of Structs though)
- No inheritance (GO strictly follows the "Composition over Inheritance" law)

Outline

It's time to GO!

Let's install GO on your machine

- Open IntelliJ and install the GO plugin
- Click here to download the GO binaries and install them on your local machine
- Set the environment variables in your *nix console:
  - `$ export GOPATH=$HOMEgo`
  - `$ export PATH=$PATH:$GOPATHbin`
- Let's run our first GO program `hello.go`!

```go
package main

import "fmt" // the standard library

func main() {
  fmt.Printf("Hello, world.\n")
}
```

Outline

Structs

Characteristics of Structs

- A Struct is a typed collection of named fields
- They're useful for grouping data together to form records
- Structs and Interfaces, together, make it possible to have OOP in GO (i.e. you can see them as Java classes, with field and methods)

Structs

```
package main
import "fmt"

type person struct {
    name string
    age  int
}

func main() {
    fmt.Println(person{"Bob", 20}) // create a new struct.
    // the & creates a pointer to the struct
    fmt.Println(&person{name: "Ann", age: 40})
    s := person{name: "Sean", age: 50}
    fmt.Println(s.name) // access struct fields with a dot.
    sp := &s
    fmt.Println(sp.age) // struct pointers are deferentiated
    sp.age = 51 // Structs are mutable.
    fmt.Println(sp.age)
}
```

Pointers

Characteristics of Pointers

- They basically behave just like C pointers do: * and & respectively represent a pointer to a value in memory, and its memory address
- Pointers are available for all types, and the pointer-to-T type is denoted `*T`
- There's no pointer arithmetic though

```
p := Vertex{1, 2}  // p is a Vertex
q := &p           // q is a pointer to a Vertex
r := &Vertex{1, 2} // r is also a pointer to a Vertex

// The type of a pointer to a Vertex is *Vertex

var s *Vertex = new(Vertex) // new creates a pointer to a new struct instance
```

Interfaces

Characteristics of Interfaces

Similar to Structs, but instead of declaring fields, they contain a "method set", which is a list of methods that a type must have in order to "implement" the interface (hence they behave just like interfaces behave in languages like java)

```go
// interface declaration
type Awesomizer interface {
    Awesomize() string
}

// types do *not* declare to implement interfaces
type Foo struct {}

// instead, types implicitly satisfy an interface if they implement all required methods
func (foo Foo) Awesomize() string {
    return "Awesome!"
}
```

Outline

CSP

Communicating Sequential Processes

What is CSP

- GO is built upon CPS (Communicating Sequential Processes)
- First described in a 1977 paper by Tony Hoare
- Formal language for describing patterns of interaction in concurrent systems
- It is a member of the family of mathematical theories of concurrency known as process algebras, based on message passing via channels
- Independent executing processes that communicate by passing messages

Differences between CSP and the Actor Model

They're both about concurrent processes that exchange messages, but with some key differences in their primitives:

CSP

- CSP processes are anonymous
- CSP message-passing fundamentally involves a rendezvous between the processes involved in sending and receiving the message, i.e. the sender awaits for a response message sent by the receiver after it received the first message
- CSP uses explicit channels for message passing

Differences between CSP and the Actor Model

Actor Model

- Actors have identities
- Message-passing in actor systems is fundamentally asynchronous, i.e. message transmission and reception do not have to happen at same time, and senders may transmit messages before receivers are ready to accept them.
- Actor systems transmit messages to named destination actors

Goroutine

What is a Goroutine

- A Goroutine is a function that is capable of running concurrently with other functions
- In order to create a Goroutine we use the keyword go followed by a function invocation
- Goroutines are lightweight and we can easily create thousands of them

Use case of a Goroutine

- Normally when we invoke a function our program will execute all the statements in a function and then return to the next line following the invocation.
- With a goroutine we return immediately to the next line and don't wait for the function to complete

Goroutine

Using Goroutines

```go
package main
import "fmt"

func printAsync(from string) {
    for i := 0; i < 3; i++ {
        fmt.Println(from, ":", i)
    }
}

func main() {
    printAsync("direct")        // function executed synchronously

    go printAsync("goroutine")      // function executed asynchronously with goroutine

    go printAsync(msg string) {
        fmt.Println(msg)            // anonymous function inside goroutine
    }("going")

    var input string
    fmt.Scanln(&input)
    fmt.Println("done")
}
```

Channel

The good ol' problem of sharing memory in a concurrent program

- GO enforces the concept of "Do not communicate by sharing memory; instead, share memory by communicating"
- Go encourages a different approach in which shared values are passed around on channels and, in fact, never actively shared by separate threads of execution
- Only one Goroutine has access to the value at any given time. Data races cannot occur, by design.
- Channels are the pipes that connect concurrent Goroutines. You can send values into channels from one Goroutine and receive those values into another Goroutine.

Channel

Channels

- A channel provides a mechanism for concurrently executing functions to communicate by sending and receiving values of a specified element type
- Channels provide a way for two Goroutines to communicate with one another and synchronize their execution
- A channel type is represented with the keyword `chan` followed by the `type` of the things that are passed on the channel
- By default sends and receives block until both the sender and receiver are ready.

Channels

A quick sample of how channels work

Use case of a Channel

The <- (left arrow) operator is used to receive messages on a certain channel or to receive a value from a channel and put it in a variable

```
ch := make(chan int) // create a channel of type int
ch <- 42             // Send a value to the channel ch.
v := <-ch            // Receive a value from ch
close(ch)            // closes the channel (only sender should close)
```

Channel

Buffered Channel

It's also possible to pass a second parameter to the make function when creating a channel (i.e. create a channel of capacity of 1):

```
c := make(chan int, 1)
```

Difference between a Buffered and Non-buffered Channel

- Normally channels are synchronous; both sides of the channel will wait until the other side is ready.
- A buffered channel is asynchronous; sending or receiving a message will not wait unless the channel is already full.

Select

Select

- The Select statement lets you wait on multiple channel operations.
- Combining Goroutines and Channels with Select is a powerful feature of Go
- Each channel will receive a value after some amount of time, to simulate e.g. blocking RPC operations executing in concurrent Goroutines

```
// select blocks on multiple channel operations
// if one unblocks the corresponding case is executed
func doStuff(channelOut, channelIn chan int) {
    select {
    case channelOut <- 42:
        fmt.Println("Write 42 into channelOut")
    case x := <- channelIn:
        fmt.Println("Save from channelIn into x")
    case <-time.After(time.Second * 1):
        fmt.Println("Timeout received. Not going to accept any other message here.")
    }
}
```

Outline

Es 1: Ping Pong

We will now try to build a simple program that simulates the Ping Pong game, by using Goroutines and Channels.

```go
package main
import "fmt"

func player(playerName string, ch chan string, rounds int) {
  // TODO create the player behaviour, by implementing a loop where
  // the player initially waits for a message,
  // then prints its own name
  // and then sends another message on the channel to unlock who's waiting for a message
}

func main() {
  fmt.Println("Welcome to the Ping Pong Game!")
  rounds := 10

  // TODO create a new channel that will used to transmit strings

  // TODO create two goroutines where you create two players, pinger and ponger

  // TODO send an initial message "init" on the channel to start the players

  var input string
  fmt.Scanln(&input)
}
```

Ex 1: Ping Pong

Solution

```go
package main
import "fmt"

func player(playerName string, ch chan string, rounds int) {
  for i := 0; i < rounds; i++ {
    <-ch
    fmt.Println(playerName)
    ch <- "unlock"
  }
}

func main() {
  fmt.Println("Welcome to the Ping Pong Game!")

  rounds := 10

  var sharedChannel chan string = make(chan string)

  go player("ping", sharedChannel, rounds)
  go player("pong", sharedChannel, rounds)

  sharedChannel <- "init"

  var input string
  fmt.Scanln(&input)
}
```

Outline

searchUtils.go

What are we going to build

- We're going to build a distributed parallel system that will retrieve some information from different sources in different locations, just like Google does
- For example, we query "foo" and the system will give us results regarding Websites, Images and Videos about our query "foo"
- Clearly, we're not Google ourselves, so we're going to mock the data, and just implement the basic logic behind it all

How is the code organized

- The code is organized in two different packages:
  - `google_launchers` contains all the `main` programs, and each launches the system using a different function and logic
  - `google` contains all the files regarding the system's logic. This is where the magic happens and where we're going to write our algorithms

Parallel Computation and Systems

Parallel computation

A computation is parallel whenever the temporal context is the same for all computational process

Parallel system

A parallel system is a computational system performing parallel computations

Distributed Computation and Systems

Distributed computation

A computation is distributed whenever at least two computational processes have a different spatial context

Distributed system

A distributed system is a computational system performing distributed computations

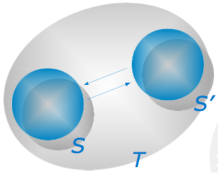


Figure: Distributed parallel computing: $S \neq S', sameT$

Quality of Service

Since we're going to iterate quite a lot of times over the next exercise, it's important to understand why.
A big part of the problems we face while building a distributed system come from the importance of QoS

QoS

Quality of service (QoS) is the description or measurement of the overall performance of a service, such as a telephony or computer network or a cloud computing service, particularly the performance seen by the users of the network.[1]

It the next few slides we're going to implement various versions of our systems, and we'll try to raise the QoS level at each iteration[2]

searchUtils.go

Already given and functioning code

`SearchUtils.go` is an helper class that contains the code regarding the data in the different repositories and some other simple data types

```go
// function that will be passed around to be executed in all the different ways (serial, parallel etc.)
type SearchFunc func(query string) Result

// the resulting data type of a Search
type Result struct {
  Title, URL string
}

// the data type we'll use to display the results on screen
type Response struct {
  Results []Result
  Elapsed time.Duration
}

// function that will be passed around to be executed in all the different ways (serial, parallel, with timeout, distributed)
func FakeSearch(kind, title, url string) SearchFunc {
  return func(query string) Result {
    time.Sleep(time.Duration(rand.Intn(100)) * time.Millisecond) // artificial delay for instructional reasons
    return Result{
      Title: fmt.Sprintf("%s(%q): %s", kind, query, title),
      URL:   url,
    }
  }
```

searchUtils.go

Already given and functioning code

The functions that will be used in the various types of data fetching processes

```
// serial, parallel, with timeout
var (
  Web = FakeSearch("web", "Distributed Systems are the best!", "http://apice.unibo.it/xwiki...")
  Image = FakeSearch("image", "But what's really a distributed system?", "https://www.unibo.it/uniboweb...")
  Video = FakeSearch("video", "Cool video about distributed systems", "https://www.youtube.com/watch?v=...")
)

// distributed
var (
  replicatedWebFakeSearch = FirstFetchAmong(Web1, Web2)
  replicatedImageFakeSearch = FirstFetchAmong(Image1, Image2)
  replicatedVideoFakeSearch = FirstFetchAmong(Video1, Video2)

  Web1 = FakeSearch("web1", "Distributed Systems are the best!", "http://apice.unibo.it/xwiki...")
  Web2 = FakeSearch("web2", "Distributed Systems are the best!", "http://apice.unibo.it/xwiki...")
  Image1 = FakeSearch("image1", "But what's really a distributed system?", "https://www.unibo.it/uniboweb...")
  Image2 = FakeSearch("image2", "But what's really a distributed system?", "https://www.unibo.it/uniboweb...")
  Video1 = FakeSearch("video1", "Cool video about distributed systems", "https://www.youtube.com/watch?v=...")
  Video2 = FakeSearch("video2", "Cool video about distributed systems", "https://www.youtube.com/watch?v=...")
)
```

Outline

googleSerial.go
DIY

- Let's implement a function that will retrieve results from the different repositories (Web, Images, Videos).
- We should fill an array on type `Result` with the results of each of these searches.

Serial search function's signature

```go
func SearchSerial(query string) ([]Result, error) {
  // TODO should return an array of results made of every possible result from Web, Image, Video for a specific query
  return nil, nil
}
```

googleSerial.go
SOLUTION

- By being serial, this function has serious scalability problems: each search on each repository will only begin when the previous one has finished
- What if a search will take a long time to complete?
- The system will be stuck and the user unhappy
- NB: keep an eye on the amount of elapsed time. We're going to reduce that a lot!

```go
func SearchSerial(query string) ([]Result, error) {
  results := []Result{
    Web(query),
    Image(query),
    Video(query),
  }
  return results, nil
}
```

Outline

googleParallel.go

DIY

- Let's iterate over our past serial solution and let's make it parallel by using Channels and Goroutines!
- The search will be still run on a single machine, but by using Goroutines we'll be able to search in every repository all at once
- We're still bound to each search function to return to our main control flow. Only after they all returned we can finally move on

```go
func SearchParallel(query string) ([]Result, error) {
  // should return the same result as the SearchSerial function, but the search should be parallelized
  // TODO create a channel of type Result and use Goroutines to parallelize the work you did in SearchSerial
  // HINT send each search result on the channel and receive those results inside the array of results
  return []Result{}, nil
}
```

googleParallel.go

SOLUTION

- Look at the elapsed time! It has been reduced by 1/3
- Also note how little more code we had to add to the previous solution in order to get this much benefit
- This solution still has problems though
- The elapsed time will now be capped to the slowest search that has been run amongst all
- Let's try to iterate once again and make it better

```go
func SearchParallel(query string) ([]Result, error) {
  c := make(chan Result)
  go func() { c <- Web(query) }()
  go func() { c <- Image(query) }()
  go func() { c <- Video(query) }()

  return []Result{<-c, <-c, <-c}, nil
}
```

Outline

googleTimeout.go

DIY

- We want to be able to get as many results as possible inside a predefined window of time, so that the elapsed time will always be shorter than X ms (timeout), no matter the query
- Consider the timer as an Agent that sends a message on the `timer` channel whenever it expires. Therefore, you can use it just like you use channels you defined yourself

```go
func SearchTimeout(query string, timeout time.Duration) ([]Result, error) {
  // should return the same result in the same way as the SearchParallel function, but only if a timeout doesn't occur
  // TODO use the timer and the select statement inside the result array to either receive the search results from
  // the channel you created before, or return with what you have when the timer expires
  // HINT you can think of the timeout as a message on a whatever channel, also return a proper error string
  timer := time.After(timeout)

  return nil, nil
}
```

googleTimeout.go

SOLUTION

- Look at the next slide for further explanation

```
func SearchTimeout(query string, timeout time.Duration) ([]Result, error) {
  timer := time.After(timeout)
  c := make(chan Result, 3)

  go func() { c <- Web(query) }()
  go func() { c <- Image(query) }()
  go func() { c <- Video(query) }()

  var results []Result
  for i := 0; i < 3; i++ {
    select {
    case result := <-c:
      results = append(results, result)
    case <-timer:
      return results, errors.New("timed out")
    }
  }
  return results, nil
}
```

googleTimeout.go

SOLUTION

- Now our search function will either return all the results if they took less than the timeout amount of time, or just the results it got before it expired
- Try and play with the timeout value, to see what happens if it becomes too little or very big
- The main problem left is that the search function is run on a single machine, on a single location for all the data.
- What if we could have multiple machines, and i.e. the closest ones to the source of the request could answer faster than the others?

Outline

googleReplicated.go
DIY

- Let's solve the replication problem!
- We will first need to modify `SearchTimeout` in order to use the `replicated*FakeSearchFunction` that we have in our `searchUtils.go` file

```go
func SearchReplicated(query string, timeout time.Duration) ([]Result, error) {
  // TODO same as you did in timeout, but this time use the replicated*FakeSearchFunctions
  // TODO also go complete the FirstFetchAmong function in searchUtils
  return nil, nil
}
```

googleReplicated.go
SOLUTION

- Here's the solution to the previous task
- As you can clearly see, not much has changed from SearchTimeout, apart from the functions we call to retrieve the results from the various repositories

```
func SearchReplicated(query string, timeout time.Duration) ([]Result, error) {
  timer := time.After(timeout)
  c := make(chan Result, 3)
  go func() { c <- replicatedWebFakeSearch(query) }()
  go func() { c <- replicatedImageFakeSearch(query) }()
  go func() { c <- replicatedVideoFakeSearch(query) }()

  var results []Result
  for i := 0; i < 3; i++ {
    select {
    case result := <-c:
      results = append(results, result)
    case <-timer:
      return results, errors.New("timed out")
    }
  }
  return results, nil
}
```

fetchFirstAmong.go
DIY

- Now let's implement a new function that will be used to distribute a search among all the machines we have
- In this example the different machines will just be simulated, but it should give you an idea on how it should work in practice

```go
func FirstFetchAmong(replicas ...SearchFunc) SearchFunc {
  return func(query string) Result {
      // TODO create a buffered channel of type Result and as long as the number of replicas that were passed in input
      // TODO create an inner function that given a index runs the replicated function at that index of replicas
      // TODO launch this function in a new goroutines for every index of replicas
      // TODO return the very first value received on the channel
    return nil
  }
}
```

fetchFirstAmong.go
SOLUTION

- Now whenever we're triggering a new query the system will handle it to different instances, and return the various results from whichever machine solved it faster
- For example, for a certain search we could get the `Web(query)` result from Machine #1, and the `Images(query)` result from Machine #2.

```
func FirstFetchAmong(replicas ...SearchFunc) SearchFunc {
  return func(query string) Result {
    c := make(chan Result, len(replicas))
    searchReplica := func(i int) {
      c <- replicas[i](query)
    }
    for i := range replicas {
      go searchReplica(i)
    }
    return <-c
  }
}
```

Outline

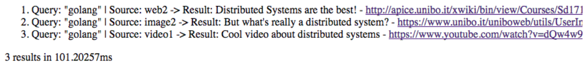
ws.go

```go
func main() {
  http.HandleFunc("/fakeGoogleSearch", handleSearch) // init a new API
  fmt.Println("serving on http://localhost:8080/fakeGoogleSearch")
  log.Fatal(http.ListenAndServe("localhost:8080", nil))
}

func handleSearch(w http.ResponseWriter, req *http.Request) {
  log.Println("serving", req.URL)
  start := time.Now()
  results, err := google.SearchReplicated("golang", 104*time.Millisecond) // run a new search
  elapsed := time.Since(start)
  if err != nil {
    http.Error(w, err.Error(), http.StatusInternalServerError)
    return
  }
  resp := google.Response{Results: results, Elapsed: elapsed}
  responseTemplate.Execute(w, resp) // build and print to screen the results
  fmt.Println(resp, err)
}

var responseTemplate = template.Must(template.New("results").Parse(
`<html><head/><body>
  <ol>{{range .Results}}
    <li>{{.Title}} - <a href="{{.URL}}">{{.URL}}</a></li>
  {{end}}</ol>
  <p>{{len .Results}} results in {{.Elapsed}}</p>
</body></html>`))
```

Final step

The webserver

What we saw in the last slide is a simple implementation of a WebServer in GO that uses what we built so far, and prints it on screen, on a webpage that (with a little fantasy on our part) kind of reminds us of Google!



```
   1. Query: "golang" | Source: web2 -> Result: Distributed Systems are the best! - http://apice.unibo.it/xwiki/bin/view/Courses/Sd171
   2. Query: "golang" | Source: image2 -> Result: But what's really a distributed system? - https://www.unibo.it/uniboweb/utils/UserIn
   3. Query: "golang" | Source: video1 -> Result: Cool video about distributed systems - https://www.youtube.com/watch?v=dQw4w9

3 results in 101.20257ms
```

We can now say that what we built is an Hybrid Architecture[3] (Horizonal and Vertical):

- Horizontal: there are many replicated machines computing a certain task (a specific search in our case)
- Vertical: there's a machine that acts like a middleware between the client and the `n` machines that should compute a certain task.

Bibliography

E. 800. "Definitions of terms related to quality of service". In: *ITU-T Recommendation* (1994).

Sameer Ajmani. *Program your next server in GO*. 2016. URL: https://www.youtube.com/watch?v=5bYO60-qYOI.

Jmaarten Van Steen Andrew S. Tanenbaum. *Distributed Systems, Principle and Paradigms*. PHI, 2009.

Rob Pike. *Less is exponentially more*. 2012. URL: https://commandcenter.blogspot.it/2012/06/less-is-exponentially-more.html.