TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Introduction to Go and RPC in Go

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2021/22

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

What is Go?

- "Go is an open source programming language that makes it easy to build simple, reliable, and efficient software." (From golang.org)
- Conceived in September 2007 at Google by R. Griesemer, R. Pike and K. Thompson, and announced in November 2009
- Goals of the language and its tools:
  - To be expressive, efficient in both compilation and execution, and effective in writing reliable and robust programs
  - Fast, statically typed, compiled language that feels like a dynamically typed, interpreted language
- Go's ancestors: mainly C and CSP (communicating sequential processes) formal language by T. Hoare

Go and C

- Go: “C-like language” or “C for the 21st century”
- From C, Go inherited
  - Expression syntax
  - Control-flow statements
  - Basic data types
  - Call-by-value parameter passing
  - Pointers
  - Emphasis on programs that compile to efficient machine code and cooperate naturally with OS abstractions

Go and other languages

- New and efficient facilities for concurrency
- Flexible approach to data abstraction and object-oriented programming
- Automatic memory management (*garbage collection*)

Go and distributed systems

- Go allows you to concentrate on distributed systems problems
  - good support for concurrency
  - good support for RPC
  - garbage-collected (no use after freeing problems)
  - type safe
- Simple language to learn

Go and cloud

- A language for cloud native applications
- Go Cloud: a library and tools for open cloud development in Go
  https://github.com/google/go-cloud
  - Goal: allow application developers to seamlessly deploy cloud applications on any combination of cloud providers
  - E.g., read from blob storage (AWS S3 or Google Cloud Storage)

```
ctx := context.Background()
bucket, err := blob.OpenBucket(ctx, "s3://my-bucket")
if err != nil {
    return err
}
defer bucket.Close()
blobReader, err := bucket.NewReader(ctx, "my-blob", nil)
if err != nil {
    return err
}
```

References

- golang.org
- Online Go tutorial tour.golang.org
- Go Playground play.golang.org
- Go by Examples gobyexample.com

- A. Donovan, B. Kernighan, “The Go Programming Language”, Addison-Wesley, 2016.

- Learn Go Programming: 7 hours video on Youtube www.youtube.com/watch?v=YS4e4q9oBaU

Editor plugins and IDEs

- vim-go plugin for vim

- GoLand by JetBrains

- Atom package Go-Plus

- Go extension for Visual Studio Code

Hello world example

```go
package main

import "fmt"

func main() {
	fmt.Println("Hello, 世界")
}
```


Some notes on the first example

- No semicolon at the end of statements or declarations
- Go natively handles Unicode
- Every Go program is made up of packages (similar to C libraries or Python packages)
  - Package: one or more .go source files in a single directory
- Source file begins with package declaration (which package the file belongs to), followed by list of other imported packages
  - Programs start running in `main`
  - `fmt` package contains functions for printing formatted output and scanning input

Go tool

- Go is a compiled language
- Go tool: the standard way to fetch, build, and install Go packages and commands
  - A zero configuration tool
- To run the program, use **`go run`**

```
$ go run helloworld.go
hello, 世界
```

- To build the program into binary, use **`go build`**

```
$ go build helloworld.go
$ ls helloworld*
helloworld      helloworld.go
$ ./helloworld
hello, 世界
```


Packages

- Go codes live in packages
- Programs start running in package `main`
- Packages contain type, function, variable, and constant declarations
- Packages can even be very small or very large
- Case determines visibility: a name is exported if it begins with a capital letter
  - `Foo` is exported, `foo` is not

Imports

- **`Import`** statement: groups the imports into a parenthesized, “factored” statement

```
package main
import (
    "fmt"
    "math")

func main() {
    fmt.Printf("Now you have %g problems.\n", math.Sqrt(7))
}
```


Functions

- Function can take zero or more arguments

```
func add(x int, y int) int {
    return x + y
}
```

  - add takes as input two arguments of type `int`
- Type comes *after* variable name
- Shorter version for input arguments:

```
func add(x, y int) int {
```

- Function can return multiple values

```
func swap(x, y string) (string, string) {
    return y, x
}
```

  - Also useful to return both result and error values

Functions

```go
package main

import "fmt"

func swap(x, y string) (string, string) {
	return y, x
}

func main() {
	a, b := swap("hello", "world")
	fmt.Println(a, b)
}
```


Functions

- Return values may be named

```go
package main

import "fmt"

func split(sum int) (x, y int) {
	x = sum * 4 / 9
	y = sum - x
	return // same as return x, y
}

func main() {
	fmt.Println(split(17))
}
```

Variables

- **var** statement: declares a list of variables
  - Type is last
- Can be at package or function level

```
package main
import "fmt"

var c, python, java bool

func main() {
    var i int
    fmt.Println(i, c, python, java)
}
```

- Can include initializers, one per variable
  - If initializer is present, type can be omitted
- Variables declared without an explicit initial value are given their *zero value*
- Short variable declaration using `:=`

Types

- Usual basic types
  - `bool, string, int, uint, float32, float64, …`
- Type conversion

```
var i int = 42
var f float64 = float64(i)
```

  - Unlike in C, in Go assignment between items of different type requires an explicit conversion
- Type inference
  - Variable's type inferred from value on right hand side

```
var i int
j := i // j is an int
```

Flow control statements

- `for`, `if` (and `else`), `switch`
- `defer`

Looping construct

- Go has only one looping construct: **`for`** loop
- Three components
  - Init statement
  - Condition expression
  - Post statement

```
sum := 0
    for i := 0; i < 10; i++ {
            sum += i
    }
```

- No parentheses surrounding the three components of the for statement
- Braces { } are always required

Looping construct

- Init and post statements are optional: for is Go's "while"

```
sum := 1
    for sum < 1000 {
            sum += sum
    }
```

- If we omit condition, infinite loop

```
for {
    }
```


Example: echo

```
// Echo prints its command-line arguments.
package main
import (
        "fmt"
        "os"
)
func main() {
        var s, sep string
        for i := 1; i < len(os.Args); i++ {
                s += sep + os.Args[i]
                sep = " "
        }
        fmt.Println(s)
}
```

s and sep initialized to empty strings

os.Args is a slice of strings (see next slides)

Conditional statements: if

- Go's **if** (and **else**) statements are like for loops:
  - Expression is not surrounded by parentheses ( )
  - Braces { } are required

```
if v := math.Pow(x, n); v < limit {
            return v
    } else {
            fmt.Printf("%g >= %g\n", v, limit)
    }
```

  - Remember that `} else` must be on the same line
  - Variable v is in scope only within the `if` statement
- `if...else if...else` statement to combine multiple `if...else` statements

Conditional statements: switch

- **switch** statement selects one of many cases to be executed
  - Cases evaluated from top to bottom, stopping when a case succeeds
- Differences from C
  - Go only runs the selected case, not all the cases that follow (i.e., C's break is provided automatically in Go)
  - Switch cases need not be constants, and the values involved need not be integers

Defer statement

- New mechanism to defer the execution of a function *until* the surrounding function returns
  – The deferred call's arguments are evaluated immediately, but the function call is not executed until the surrounding function that contains `defer` has terminated

```
package main
import "fmt"

func main() {
  defer fmt.Println("world")
  fmt.Println("hello")
}
```

```
hello
world
```

- Deferred function calls pushed onto a stack
  – Deferred calls executed in LIFO order
- Great for cleanup things, like closing files or connections!

Pointers

- Pointer: value that contains the address of a variable
  – Usual operators `*` and `&`: & operator yields the address of a variable, and * operator retrieves the variable that the pointer refers to

```
var p *int
i := 1
p = &i    // p, of type *int, points to i
fmt.Println(*p) // "1"
*p = 2 // equivalent to i = 2
fmt.Println(i) // "2"
```

- Unlike C, Go has no pointer arithmetic
- Zero value for a pointer is `nil`
- Perfectly safe for a function to return the address of a local variable, because the local variable will survive the scope of the function

Composite data types: structs and array

- Aggregate data types: structs and arrays
- **Struct**: typed collection of fields
  – Syntax similar to C, fixed size

```
type Vertex struct {
        X int
        Y int
}
```

  – Struct fields are accessed using a dot; can also be accessed through a struct pointer

- **Array**: `[n]T` is an array of n values of type T
  – Fixed size (cannot be resized)

```
var a [2]string
a[0] = "Hello"
```


Composite data types: slices

- Slice: key data type in Go, more powerful than array

- `[ ]T` is a **slice** with elements of type T: dynamically-sized, flexible view into the elements of an array
  – Specifies two indices, a low and high bound, separated by a colon: `s[i : j]`
  – Slice include first element, but excludes last one

```
primes := [6]int{2, 3, 5, 7, 11, 13}
var s []int = primes[1:4]
```

`[3 5 7]`

- Slice: section of the *underlying array*
  – When you modify the slice elements, you also modify the elements of the underlying array

Slices: operations

- Length of slice `s`: number of elements it contains, use `len(s)`
- Capacity of slice `s`: number of elements in the underlying array, counting from the first element in the slice, use `cap(s)`
- Compile or run-time error if array length is exceeded: Go performs bounds check because it's a memory safe language

- Slices can be created using `make`
  - Length and capacity can be specified

Slices: operations

- Let's create an empty slice

```
package main
import "fmt"
func main() {
  a := make([]int, 0, 5)      // len(s)=0, cap(s)=5
  printSlice("a", a)
}

func printSlice(s string, x []int) {
  fmt.Printf("%s len=%d cap=%d %v\n", s, len(x), cap(x), x)
}
```

```
a len=0 cap=5 []
```

Slices: operations

- New items can be appended to a slice using **append**

```
func append(slice []T, elems ...T) []T
```

  – When append a slice, slice may be enlarged if necessary

```
func main() {
    var s []int
    printSlice(s)

    s = append(s, 0) // works on nil slices
    printSlice(s)

    s = append(s, 1) // slice grows as needed
    printSlice(s)

    s = append(s, 2, 3, 4) // more than one element
    printSlice(s)
}
```


Composite data types: maps

- **map**: maps keys to values
  – Map type **map[K]V** is a reference to a hash table where K and V are the types of its keys and values
  – Use make to create a map

```
m = make(map[string]Vertex)
m["Bell Labs"] = Vertex{
    40.68433, -74.39967,
}
```

- You can insert or update an element in a map, retrieve an element, delete an element, test if a key is present

```
m[key] = element        // insert or update
elem = m[key]           // retrieve
delete(m, key)          // delete
elem, ok = m[key]       // test
```

Range

- `range` iterates over elements in a variety of data structures
  - range on arrays and slices provides both index and value for each entry
  - range on map iterates over key/value pairs

```
package main
import "fmt"

var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}

func main() {
    for i, v := range pow {
            fmt.Printf("2**%d = %d\n", i, v)
    }
}
```


Range: example

```
func main() {
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }
    for k := range kvs {
        fmt.Println("key:", k)
    }
}
```

```
$ go run range2.go
sum: 9
index: 1
a -> apple
b -> banana
key: a
key: b
```

Methods

- Go does not have classes, but supports **methods** defined on struct types
- A method is a function with a special *receiver* argument (extra parameter before the function name)
  - The receiver appears in its own argument list between the `func` keyword and the method name

```
type Vertex struct {
    X, Y float64
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```


Interfaces

- An *interface type* is defined as a named collection of method signatures
- Any type (struct) that implements the required methods, implements that interface
  - Instead of designing the abstraction in terms of what kind of data our type can hold, we design the abstraction in terms of *what actions* our type can execute
- A type is not explicitly declared to be of a certain interface, it is implicit
  - Just implement the required methods
- Let's code a basic interface for geometric shapes

Interface: example

```go
package main

import "fmt"
import "math"

// Here's a basic interface for geometric shapes.
type geometry interface {
    area() float64
    perim() float64
}

// For our example we'll implement this interface on
// rect and circle types.
type rect struct {
    width, height float64
}
type circle struct {
    radius float64
}
```


Interface: example

```go
// To implement an interface in Go, we just need to
// implement all the methods in the interface. Here we
// implement `geometry` on `rect`s.
func (r rect) area() float64 {
    return r.width * r.height
}
func (r rect) perim() float64 {
    return 2*r.width + 2*r.height
}

// The implementation for `circle`s.
func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}
func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}
```

Interface: example

```
// If a variable has an interface type, then we can call
// methods that are in the named interface. Here's a
// generic `measure` function taking advantage of this
// to work on any `geometry`.
func measure(g geometry) {
    fmt.Println(g)
    fmt.Println(g.area())
    fmt.Println(g.perim())
}
func main() {
    r := rect{width: 3, height: 4}
    c := circle{radius: 5}

    // The `circle` and `rect` struct types both
    // implement the `geometry` interface so we can use
    // instances of these structs as arguments to `measure`.
    measure(r)
    measure(c)
}
```

```
$ go run interfaces.go
{3 4}
12
14
{5}
78.53981633974483
31.41592653589793
```


Concurrency in Go

- Go provides concurrency features as part of the core language
- Goroutines and channels
  - Support CSP concurrency model
- Can be used to implement different concurrency patterns

Goroutines

- A **goroutine** is a lightweight thread managed by the Go runtime

```
go f(x, y, z)  // starts a new goroutine running
               // f(x, y, z)
```

- Goroutines run in the same address space, so access to shared memory must be synchronized

Channels

- Communication mechanism that lets one goroutine sends values to another goroutine
  – A channel is a thread-safe queue managed by Go and its runtime
  – It blocks threads that read on it, etc.
- Hides a lot of pain of inter-thread communication
  – Internally, it uses mutexes and semaphores just as one might expect
- Multiple senders can write to the same channel
  – Really useful for notifications, multiplexing, etc.
- And it's totally thread-safe!
- But be careful: only one can `close` the channel, and can't send after close!

Channels

- A typed conduit through which you can send and receive values using the channel operator `<-`

```
ch <- v     // Send v to channel ch
v := <- ch // Receive from ch, and
           // assign value to v
```

Data flows in the direction of the arrow

- Channels must be created before use

```
ch := make(chan int)
```

- Sends and receives block until the other side is ready
  - Goroutines can synchronize without explicit locks or condition variables

Channels: example

```
import "fmt"
func sum(s []int, c chan int) {
        sum := 0
        for _, v := range s {
                sum += v
        }
        c <- sum // send sum to c
}

func main() {
        s := []int{7, 2, 8, -9, 4, 0}
        c := make(chan int)
        go sum(s[:len(s)/2], c)
        go sum(s[len(s)/2:], c)
        x, y := <-c, <-c // receive from c
        fmt.Println(x, y, x+y)
}
```

- Distributed sum: sum is distributed between two Goroutines
- An example of applying the common SPMD pattern for parallelism

Channels: example

```
package main
import "fmt"
func fib(c chan int) {
        x, y := 0, 1
        for {
                c <- x
                x, y = y, x+y
        }
}
func main() {
        c := make(chan int)
        go fib(c)
        for i := 0; i < 10; i++ {
                fmt.Println(<-c)
        }
}
```

Elegant and efficient!

- Fibonacci sequence: iterative version using channel

Buffered channels

- By default (i.e., *unbuffered channel*), channel operations block
  - In spec.: *If the capacity is zero or absent, the channel is unbuffered and communication succeeds only when both a sender and receiver are ready.*
  - *If the channel is unbuffered, the sender blocks until the receiver has received the value.*
- Buffered channels do not block if they are not full
  - Buffer length as `make` second argument to initialize a buffered channel

    ```
    ch := make(chan int, 100)
    ```
  - Send to a buffered channel blocks only when buffer is full
  - Receive blocks when buffer is empty (no data to receive)

More on channels: close and range

- Close on buffers
  - Sender can `close` a channel
  - Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression

    ```
    v, ok := <-ch
    ```

    - ok is false if there are no more values to receive and the channel is closed

- Range on buffers

  ```
  for i := range ch
  ```

  - Use it to receive values from the channel repeatedly until it is closed

More on channels: select

- `select` lets a goroutine wait on multiple communication operations
  - Blocks until one of its cases can run, then it executes that case (one at random if multiple are ready)
- We can use select with a `default` clause to implement *non-blocking* sends, receives, and even non-blocking multi-way selects

```
select {
    case mgs1 := <- ch1:      // receive
        // ...
    case msg2 := <- ch2:      // receive
        // ...use x...
    case ch3 <- msg3:         // send
        // ...
    default:
        // ...
}
```

Using select: example

- Fibonacci sequence: iterative version using two channels, the latter being used to quit

```
package main
import "fmt"

func fibonacci(c, quit chan int) {
        x, y := 0, 1
        for {
                select {
                case c <- x:
                        x, y = y, x+y
                case <-quit:
                        fmt.Println("quit")
                        return
                }
        }
}
```


Using select: example

```
func main() {
        c := make(chan int)
        quit := make(chan int)
        go func() {
                for i := 0; i < 10; i++ {
                        fmt.Println(<-c)
                }
                quit <- 0
        }()
        fibonacci(c, quit)
}
```

- Go supports anonymous functions

Timers

- You can implement timeouts by using a timer channel

```
//to wait 2 seconds
timer := time.NewTimer(time.Second * 2)
    <- timer.C
```

  - You tell the timer how long you want to wait, and it provides a channel that will be notified at that time
  - `<-timer.C` blocks on the timer's channel `C` until it sends a value indicating that the timer fired
  - Timer can be canceled before it fires

A few more things

- Error handling
- Variadic functions
- Modules
- Go tools

- Many others, but this is an introduction to Go!

Error handling

- Go code uses error values to indicate abnormal state
- Errors are communicated via an explicit, separate return value
  - By convention, the last return value of a function
  - `nil` value in the error position: no error
  - *"Error handling [in Go] does not obscure the flow of control." (R. Pike)*

```
result, err := SomeFunction()
if err != nil {
    // handle the error
}
```

- Built-in error interface

```
type error interface {
    Error() string
}
```

  - `errors.New` constructs a basic error value with the given error message

See https://blog.golang.org/error-handling-and-go

Variadic functions

- Go functions can accept a variable number of arguments: variadic functions
  - E.g., `fmt.Println` is a variadic function

```
package main

import "fmt"

func sum(nums ...int) {
    fmt.Print(nums, " ")
    total := 0
    for _, num := range nums {
        total += num
    }
    fmt.Println(total)
}
```

```
func main() {

    sum(1, 2)
    sum(1, 2, 3)

    nums := []int{1, 2, 3, 4}
    sum(nums...)
}
```

```
$ go run variadic-functions.go
[1 2] 3
[1 2 3] 6
[1 2 3 4] 10
```

Go modules

- **Module**: collection of related Go packages stored in a file tree with a `go.mod` file at its root
- go.mod file defines:
  - module path, which is also the import path used for the root directory
  - minimum version of Go required by the current module.
  - its dependency requirements, which are the other modules needed for a successful build with their minimum version we can use
- To generate `go.mod` file:

```
$ go mod init <module_name>
```

- To add missing and remove unused module requirements:

```
$ go mod tidy
```

```
module example.com/mymodule

go 1.14

require (
    example.com/othermodule v1.2.3
    example.com/thismodule v1.2.3
    example.com/thatmodule v1.2.3
)
```


Common errors and Go tools

- Go can be somewhat picky
- Unused variables raise errors, not warnings
  - Use "_" for variables you don't care about
- Unused imports raise errors
  - Use goimports command to automatically add/remove imports
- In if-else statements `{` must be placed at the end of the same line
  - E.g., `} else {`
  - E.g., `} else if … {`
  - Use gofmt command to format Go code

RPC in Go

- Go standard library has support for RPC right out-of-the-box
  - Package `net/rpc` in standard Go library
  https://golang.org/pkg/net/rpc/
- TCP or HTTP as "transport" protocols
- Some constraints for RPC methods
  - Only two arguments are allowed
  - Second argument is a pointer to a reply struct that stores the corresponding data
  - An error is always returned

```
func (t *T) MethodName(argType T1, replyType *T2) error
```


RPC in Go: marshaling and unmarshaling

- Use `encoding/gob` package for parameters marshaling (encode) and unmarshaling (decode)
  https://pkg.go.dev/encoding/gob
  - Package gob manages streams of gobs (binary values) exchanged between an Encoder (transmitter) and a Decoder (receiver)
  - A stream of gobs is *self-describing*: each data item in the stream is preceded by a specification of its type, expressed in terms of a small set of predefined types; pointers are not transmitted, but the values they point to are transmitted
  - Basic usage: create an encoder, transmit some values, receive them with a decoder
  - Requires that both RPC client and server are written in Go

RPC in Go: marshaling and unmarshaling

- Two alternatives to using gob
- `net/rpc/jsonrpc` package
  - Implements a JSON-RPC 1.0 ClientCodec and ServerCodec for `rpc` package https://pkg.go.dev/net/rpc/jsonrpc
- gRPC
  - See next lesson, can also be used to write polyglot RPC client and server

RPC in Go: server

- On server side
  - Use **`Register`** (or **`RegisterName`**)

```
func (server *Server) Register(rcvr interface{}) error
func RegisterName(name string, rcvr interface{}) error
```

- To publish the methods that are part of the given interface on the default RPC server and allows them to be called by clients connecting to the service
- Takes a single parameter, which is the interface

  - Use **`Listen`** to announce on the local network address

```
func Listen(network, address string) (Listener, error)
```

RPC in Go: server

– Use **`Accept`** to receive connections on the listener and serve requests for each incoming connection

```
func (server *Server) Accept(lis net.Listener)
```

- `Accept` is blocking; if the server wishes to do other work as well, it should call this in a goroutine

– Can also use HTTP handler for RPC messages (see example on the course site)

RPC in Go: client

- On client side
  – Use **`Dial`** to connect to RPC server at the specified network address (and port)

```
func Dial(network, address string) (*Client, error)
```

  - Use `DialHTTP` for HTTP connection

  – Use **`Call`** to call synchronous RPC

  – Use **`Go`** to call asynchronous RPC
    - Associated channel will signal when the call is complete

RPC in Go: example

- Let's consider two simple remote functions, multiply and divide two integers
- Code available on course site

RPC in Go: synchronous call

- Need some setup in advance of this…
- `Call` makes blocking RPC call
- `Call` invokes the named function, waits for it to complete, and returns its error status

```
// Synchronous call
args := &server.Args{7,8}
var reply int
err = client.Call("Arith.Multiply", args, &reply)
if err != nil {
   log.Fatal("arith error:", err)
}
fmt.Printf("Arith: %d*%d=%d", args.A, args.B, reply)
```

```
func (client *Client) Call(serviceMethod string,
       args interface{}, reply interface{}) error
```

RPC in Go: asynchronous call

- How to make an asynchronous RPC? `net/rpc/Go` uses a channel as parameter to retrieve the RPC reply when the call is complete
- `Done` channel will signal when the call is complete by returning the same object of `Call`
  - If `Done` is nil, Go will allocate a new channel

```
// Asynchronous call
quotient := new(Quotient)
divCall := client.Go("Arith.Divide", args, quotient, nil)
divCall = <-divCall.Done
// check errors, print, etc.
```

```
func (client *Client) Go(serviceMethod string, args
interface{}, reply interface{}, done chan *Call) *Call
```

- For Go internal implementation, see https://golang.org/src/net/rpc/client.go?s=8029:8135#L284