IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Ioana Leustean

PROGRAMARE CONCURENTA IN

GO

The Go Programming Language

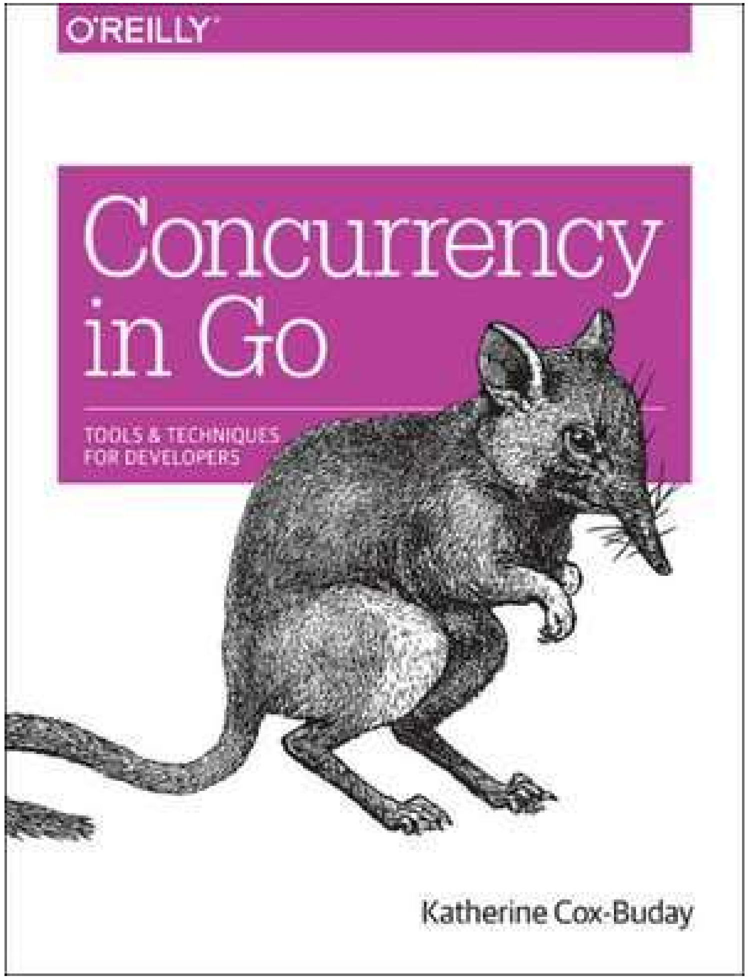
Goroutines and channels

"Before Go was first revealed to the public, this was where the chain of abstraction ended for most of the popular programming languages. If you wanted to write concurrent code, you would model your program in terms of threads and synchronize the access to the memory between them. If you had a lot of things you had to model concurrently and your machine couldn't handle that many threads, you created a *thread pool* and multiplexed your operations onto the thread pool.

Go has added another link in that chain: the *goroutine*. In addition, Go has borrowed several concepts from the work of famed computer scientist Tony Hoare, and introduced new primitives for us to use, namely *channels*.

…

Threads are still there, of course, but we find that we rarely have to think about our problem space in terms of OS threads. Instead, we model things in goroutines and channels, and occasionally shared memory."



➢Bibliografie

Katherine Cox-Buday, Concurrency in Go, O'Reilly, 2017

Concurrency — An Introduction to Programming in Go | Go Resources

Get Started - The Go Programming Language

Go Packages - Go Packages

The Go Programming Language Specification - The Go Programming Language

Go Wiki: Home - The Go Programming Language

```
package main

import "fmt"

var x string = "hello"

func main() {

    fmt.Println(x)
}
```

Programele sunt grupate in pachete (module).

Un pachet poate continue mai multe fisiere.

Fisierele executabile trebuie sa contina functia **main**.

```
PS C:\Users\igleu\Documents\DIR\ICLP\GO\hello> go mod tidy
PS C:\Users\igleu\Documents\DIR\ICLP\GO\hello> go run hello.go
hello
```

- Sistemul tipurilor este static.
- Declaratia tipului nu e obligatorie, variabilele se pot declara cu valoare initiala si tipul e dedus:

```
 var x int
  x= 32
```

```
var x int = 32
```

x:= 32 (in acest caz nu declaram tipul)

- Tipuri:

  simple: **int, int16, int32, float32,**
  float64, string (immutable), **bool**

  compuse: **map, array, struct, slice**

- Pointeri:

```
  x:=1
  p:=&x
 *p:=2
```

- Tipuri definite de utilizator
- Metode associate tipurilor
- Interfete

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

```
func (p Point) Distance(q Point) float64 {
     return math.Hypot(q.X-p.X, q.Y-p.Y)}
```

Distance este o metoda asociata **Point**

```
for i <= 3 {
        fmt.Println(i)
        i = i + 1
    }

    for j := 0; j < 3; j++
{
        fmt.Println(j)
    }

    for {

fmt.Println("loop")
        break
    }
```

```
if 7%2 == 0 {
        fmt.Println("7 is even")
    } else {
        fmt.Println("7 is odd")
    }


    if 8%4 == 0 {
        fmt.Println("8 is divisible by 4")
    }
```

Functiile sunt valori

```
import (
      "fmt")

var salutation string
var nume string = "Ioana"

func g() func(s string) {
      var salutation string
      salutation = "welcome"
      nume = "Ana"
      return func(s string) { fmt.Println(salutation, s, nume) }
}
```

```
func main() {
      salutation = "hello"
      fmt.Println(salutation, nume)
      g()(" you")
      fmt.Println(salutation, nume)
   }
```

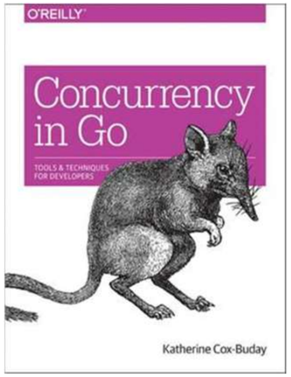
```
> go run fctex.go
hello Ioana
welcome  you Ana
hello Ana
```

Concurenta in Go

Gorutine si canale

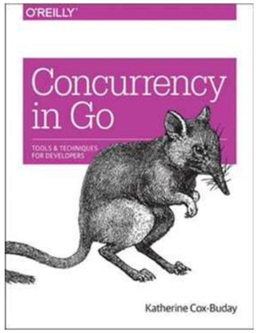
Goroutines are unique to Go (though some other languages have a concurrency primitive that is similar). They're not OS threads, and they're not exactly green threads—threads that are managed by a language's runtime—they're a higher level of abstraction known as *coroutines*. Coroutines are simply concurrent subroutines (functions, closures, or methods in Go) that are *nonpreemptive*—that is, they cannot be interrupted. Instead, coroutines have multiple points throughout which allow for suspension or reentry.

What makes goroutines unique to Go are their deep integration with Go's runtime. Goroutines don't define their own suspension or reentry points; Go's runtime observes the runtime behavior of goroutines and automatically suspends them when they block and then resumes them when they become unblocked. In a way this makes them preemptable, but only at points where the goroutine has become blocked. It is an elegant partnership between the runtime and a goroutine's logic. Thus, goroutines can be considered a special class of coroutine.



Go’s mechanism for hosting goroutines is an implementation of what’s called an *M:N scheduler*, which means it maps M green threads to N OS threads. Goroutines are then scheduled onto the green threads. When we have more goroutines than green threads available, the scheduler handles the distribution of the goroutines across the available threads and ensures that when these goroutines become blocked, other goroutines can be run.

Go follows a model of concurrency called the *fork-join* model.1 The word *fork* refers to the fact that at any point in the program, it can split off a *child* branch of execution to be run concurrently with its *parent*. The word *join* refers to the fact that at some point in the future, these concurrent branches of execution will join back together. Where the child rejoins the parent is called a *join point*.



O'REILLY

Concurrency in Go

TOOLS & TECHNIQUES FOR DEVELOPERS

Katherine Cox-Buday

```go
package main

import (
	"fmt"
)

func f(s string) {
	for i := 0; i < 30; i++ {
		fmt.Print(s)
			}
}
```

```go
func main() {

	go f("A")
	go f("B")

	fmt.Println("gata")
}
```

main este o gorutina

```
PS C:\Users\igleu\Documents\DIR\ICLP\GO\conc> go mod tidy
PS C:\Users\igleu\Documents\DIR\ICLP\GO\conc> go run concAB.go
gata
```

```
package main

import (
	"fmt")

func f(s string) {
	for i := 0; i < 30; i++ {
		fmt.Print(s)
			}
}
```

```
func main() {
	go f("A")
	go f("B")

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

```
PS C:\Users\igleu\Documents\DIR\ICLP\GO\conc> go run concAB.go
AAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAe
gata
PS C:\Users\igleu\Documents\DIR\ICLP\GO\conc> go run concAB.go
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBe
gata
```

```
package main

import (
	"fmt")

func f(s string) {
	for i := 0; i < 30; i++ {
		fmt.Print(s)
			}
}
```

```
func main() {
	go f("A")
	  go func(s string) {
		for i := 0; i < 30; i++ {
			fmt.Print(s)
		}
	}("B")

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

apelul unei functii anonime

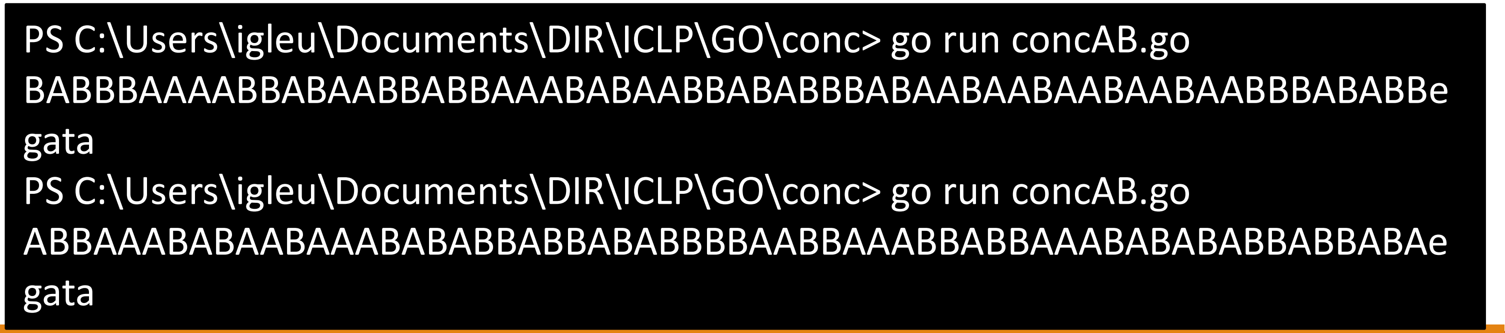
```
PS C:\Users\igleu\Documents\DIR\ICLP\GO\conc> go run concAB.go
AAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAe
gata
```

```
package main

import (
	"fmt"
	 "math/rand"
	"time")

func f(s string) {
	for i := 0; i < 30; i++ {
		fmt.Print(s)
		   amt := time.Duration(rand.Intn(250))
		time.Sleep(time.Millisecond * amt)
		    }}
```

Sleep opreste executia corutinei

[time package - time - Go Packages]

```
func main() {
	go f("A")
	go f("B")

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



```
PS C:\Users\igleu\Documents\DIR\ICLP\GO\conc> go run concAB.go
BABBBAAAABBABAABBABBAAABABAABBABABBBABAABAABAABAABAABBBABABBe
gata
PS C:\Users\igleu\Documents\DIR\ICLP\GO\conc> go run concAB.go
ABBAAABABAABAAABABABBABBABABBBBAABBAAABBABBAAABABABABBABBABAe
gata
```

sincronizare folosind **sync.WaitGroup**

```
package main

import (
	"fmt"
	"sync"
)

var wg sync.WaitGroup

func f(s string) {
	defer wg.Done()
	for i := 0; i < 5; i++ {
		fmt.Print(s)
		}}
```

```
func main() {
	wg.Add(1)
	go f("A")

	wg.Wait()
	fmt.Println("gata")
}
```

```
> go run concABW.go
AAAAAgata
```

defer amana xecutia unei functii pana cand functia parinte isi termina executia

sincronizare folosind **sync.WaitGroup**

```
package main

import (
        "fmt"
        "sync")

var wg sync.WaitGroup

func f(s string) {
        defer wg.Done()
        for i := 0; i < 5; i++ {
                fmt.Print(s)
                }}

  func main() {
        wg.Add(1)
        go f("A")

        wg.Wait()
        fmt.Println("gata")
  }
```

A WaitGroup waits for a collection of goroutines to finish.

The main goroutine calls [WaitGroup.Add] to set the number of goroutines to wait for. Typically the calls to Add should execute before the statement creating the goroutine or other event to be waited for.

Then each of the goroutines runs and calls [WaitGroup.Done] when finished.

At the same time, [WaitGroup.Wait] can be used to block until all goroutines have finished.

- The Go Programming Language

sincronizare folosind **sync.WaitGroup**

```
package main

import (
	"fmt"
	"sync"
)

var wg sync.WaitGroup

func f(s string) {
	defer wg.Done()
	for i := 0; i < 5; i++ {
		fmt.Print(s)
		}
}
```

```
func main() {
	wg.Add(1)
	go f("A")
	wg.Add(1)
	go f("B")

	wg.Wait()
	fmt.Println("gata")
}
```

```
> go run concABW.go
BBBBBAAAAAgata
```

Transmiterea ca parametru a unui **WaitGroup** se face prin referinta

```
package main

import (
    "fmt"
    "sync"
)

func f(s string, wg  *sync.WaitGroup) {
    defer wg.Done()
    for i := 0; i < 5; i++ {
        fmt.Print(s)
        }
}
```

```
func main() {
    var wg sync.WaitGroup
    wg.Add(1)
    go f("A", &wg)
    wg.Add(1)
    go f("B", &wg)

    wg.Wait()
    fmt.Println("gata")
}
```

```
> go run concABW.go
BBBBBAAAAAgata
```

wg actioneaza ca un contor care este incrementat cu valoarea transmisa prin **Add**. **Wait** blocheaza executia pana cand contorul are valoarea 0.

```
package main

import (
    "fmt"
    "sync"
)

func f(s string, wg  *sync.WaitGroup) {
    defer wg.Done()
    for i := 0; i < 5; i++ {
        fmt.Print(s)
        }
}
```

```
func main() {
    var wg sync.WaitGroup
    wg.Add(2)
    go f("A", &wg)
    go f("B", &wg)

    wg.Wait()
    fmt.Println("gata")
}
```

```
> go run concABW.go
BBBBBAAAAAgata
```

Closures

```
func main() {
    var wg sync.WaitGroup
    for _, salutation := range []string{"hello",
                            "greetings", "good day"} {
        wg.Add(1)
        go func() {
            defer wg.Done()
            fmt.Println(salutation)
            }()
}
    wg.Wait()}
```

```
> go run gocl.go
greetings
hello
good day

> go run gocl.go
good day
hello
greetings

> go run gocl.go
greetings
good day
hello
```

https://go.dev/wiki/LoopvarExperiment

Observatie: Closures si Wait

```
import (
      "fmt"
        "sync")

var salutation string
var nume string = "Ioana"
var wg sync.WaitGroup

func g() func(s string) {
      var salutation string
      defer wg.Done()
      salutation = "welcome"
            return func(s string) {
                  fmt.Println(salutation, s, nume)}}
```

```
func main() {
      salutation = "hello"
      wg.Add(1)
      go g()(" you")
      wg.Wait()
      fmt.Println(salutation, nume)}
```

```
> go run closure1.go
welcome  you Ioana
hello Ioana

> go run closure1.go
hello Ioana
```

nu s-a executat functia closure

Observatie: Closures si Wait

```
import (
      "fmt"
        "sync")

var salutation string
var nume string = "Ioana"
var wg sync.WaitGroup

func g() func(s string) {
      var salutation string
      salutation = "welcome"
            return func(s string) {
            defer wg.Done()
            fmt.Println(salutation, s, nume)}}
```

```
func main() {
      salutation = "hello"
      wg.Add(1)
      go g()(" you")
      wg.Wait()
      fmt.Println(salutation, nume)}
```

```
> go run closure1.go
welcome  you Ioana
hello Ioana

> go run closure1.go
hello Ioana
```

- Gorutinele comunica intre ele folosind **canale**

- Canalele pot fi cu capacitate (buffered) sau fara capacitate (unbuffered, cu capacitate 0)
  c := make(chan int) (este echivalent cu **make(chan int,0)**)
  c := make(chan int, 3)

- Operatiile cu canale:
  c <- x // valoarea x este trimisa pe canal (scriere)
  x = <-c // x primeste o valoare de pe canal (citire)
  close(c) // canalul este inchis; se pot face citiri, dar valoarea citita va fi 0 (valoarea nula a tipului respectiv;
  nu se pot face scrieri

- Comunicarea pe canalele fara capacitate este **sincrona**: un mesaj este transmis (scris) numai daca exista si cineva care va primi (citi) acel mesaj; in caz contrar, gorutina care trimite mesajul este blocata pana cand cea care primeste este disponibila.

- Comunicarea pe canalele cu capacitate este **asincrona**. Accesul la canale este blocant: o gorutina care incearca sa citeasca dintr-un canal gol va astepta pana cand canalul continue o valoare si orice gorutina care vrea sa scrie pe un canal va astepta pana cand exista o locatie libera.

Sincronizare folosind canale in loc de sync.WaitGroup

```
func f(s string, c chan string) {
    for i := 0; i < 30; i++ {
        fmt.Print(s)
            }
    c <- "gata"
}
```

```
func main() {
    done := make(chan string)
    go f("A", done)
    s := <-done
    fmt.Println(s)
}
```

Comunicare folosind canale

```
func f(s string, c chan string) {
    defer close(c)
    for i := 0; i < 30; i++ {
        c <- s
    }
    }
```

```
func main() {
    buf := make(chan string)
    go f("A", buf)
    s, ok := <-buf
    for ok {
        fmt.Println(s)
        s, ok = <-buf
    }
}
```

ok va fi **false** cand canalul este inchis

Comunicare folosind canale

```
func f(s string, c chan string) {
    defer close(c)
    for i := 0; i < 30; i++ {
        c <- s
    }
    }
```

```
func main() {
    buf := make(chan string)
    go f("A", buf)

    for s := range buf {
        fmt.Println(s)

    }}
```

se poate folosi **range** pentru a itera prin valorile unui canal

Comunicare folosind canale unidirectionale: chan<- si **<-chan**

```
func f(s string, c chan<- string) {
      defer close(c)
      for i := 0; i < 3; i++ {
            c <- s
      }}
```

```
func main() {
      buf := make(chan string)
      go f("A", buf)

      for s := range buf {
            fmt.Println(s)

      }}
```

canalele pot fi unidirectionale:
parametrul functiei f poate fi folosit numai pentru trimiterea mesajelor,
Iar incercare de a-l folosi pentru citire va da eroare

Comunicare folosind canale: gorutine pentru citire si scriere

```
func f(s string, c chan<- string) {
    defer close(c)
    for i := 0; i < 3; i++ {
        c <- s
    }}
```

```
func main() {
    bufA := make(chan string)
    bufB := make(chan string)
    go f("A", bufA)
    go f("B", bufB)
    go func() {
        for a := range bufA {
            fmt.Println(a)}}()
    go func() {
        for b := range bufB {
            fmt.Println(b)}}()
    var input string
    fmt.Scanln(&input)}
```

apelul unei functii anonime

Este necesar pentru a ne asigura ca sunt executate gorutinele.
Se poate folosi un alt canal sau Wait

Comunicare folosind canale: gorutine pentru citire si scriere

```
func f(s string, c chan<- string) {
    defer close(c)
    for i := 0; i < 5; i++ {
        c <- s
    }}
```

```
func main() {
    bufA := make(chan string)
    bufB := make(chan string)
    go f("A", bufA)
    go f("B", bufB)
    for j := 0; j < 15; j++ {
    select {
        case a := <-bufA:
            fmt.Println(a)
        case b := <-bufB:
            fmt.Println(b)
        default:
            fmt.Println("    .")
            }}
```

Toate conditiile de pe ramurile instuctiunii select sunt analizate in paralel. Daca niciuna nu poate fi executata se executa **default**, iar daca default lipseste gorutina este blocata pana cand o conditie este indeplinita.

Comunicare folosind canale: gorutine pentru citire si scriere

```
> go run chan5.go
   .BAAAABABBB    .    .    .    .

> go run chan5.go
   .AAAAABBBBB    .    .    .    .

> go run chan5.go
   .BABBBBAA   .AA    .    .    .
```

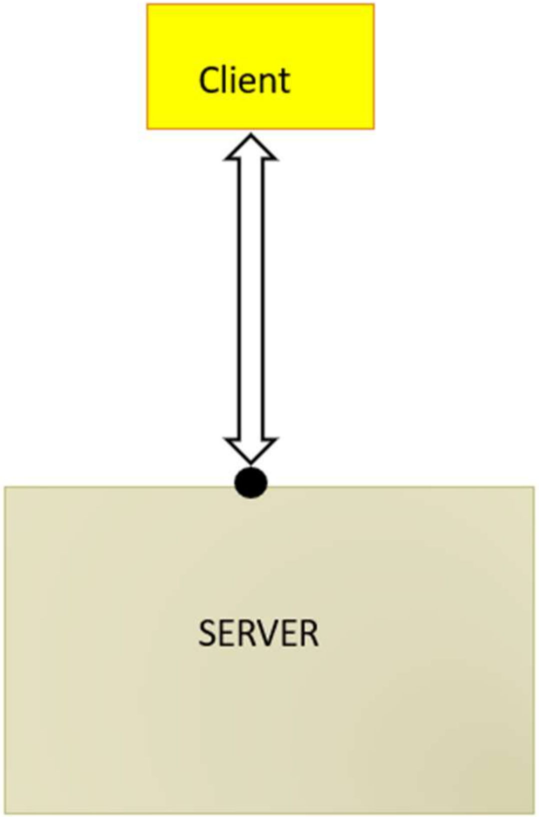
```
func main() {
    bufA := make(chan string)
    bufB := make(chan string)
    go f("A", bufA)
    go f("B", bufB)
    for j := 0; j < 15; j++ {
    select {
        case a := <-bufA:
            fmt.Print(a)
        case b := <-bufB:
            fmt.Print(b)
        default:
            fmt.Print("    .")
            }}
```

Comunicare folosind canale: gorutine pentru citire si scriere

```
func f(s string, c chan<- string) {
    defer close(c)
    for i := 0; i < 5; i++ {
        c <- s
    }}
```

Functia **time.After (t)** intoarce un canal care va trimite timpul curent dupa perioada t.

```
func main() {
    bufA := make(chan string)
    bufB := make(chan string)
    go f("A", bufA)
    go f("B", bufB)
    for j := 0; j < 15; j++ {
    ok := true
    for ok {
        select {
        case a := <-bufA:  fmt.Println(a)
        case b := <-bufB: fmt.Println(b)
        case <-time.After(5 * time.Second):
                ok = false
        }}
```



Implementarea unui server

https://pkg.go.dev/net

Implementarea serverului

```
package main

import (
    "fmt"
    "net"
)
```

```
func main() {
    // listener este socketul serverului
    listener, err := net.Listen("tcp", "localhost:8081")
     //prelucrare eroare

    fmt.Println("Server is listening")
    defer listener.Close()

    for {
        // acceptarea conexiuni
        conn, err := listener.Accept()
           //prelucrare eroare
        fmt.Println("New client")

        // gorutina pentru interactiunea cu clientul
        go handleClient(conn)}}
```

Implementarea serverului

```
package main
import (
       "fmt"
       "net")
func main() {
     listener, err := net.Listen("tcp", "localhost:8081")
     fmt.Println("Server is listening")
       defer listener.Close()
       for {
                    conn, err := listener.Accept()
                    fmt.Println("New client")
                   go handleClient(conn)
       }}
```

conn.Read si **conn.Write** sunt folosite pentru comunicarea dintre server si client. Ele folosesc date de tip **[]byte** (pentru alte tipuri trebuie facuta conversia)

```
func handleClient(conn net.Conn) {
       defer conn.Close()

       // transmiterea unui mesaj catre client
    conn.Write([]byte("Hello client!"))

       // primirea unui mesaj trimis de client
       buffer := make([]byte, 1024)
       n, err := conn.Read(buffer)
       //prelucrarea erorii

       // procesarea datelor primite de la client
       fmt.Printf("Received: %s\n", buffer[:n])
}
```

Implementarea serverului

```
func main() {
        listener, err := net.Listen("tcp", "localhost:8080")
        if err != nil {
                fmt.Println("Error:", err)
                return}
          fmt.Println("Server is listening")

        defer listener.Close()

        for {
                conn, err := listener.Accept()
                if err != nil {
                     fmt.Println("Error:", err)
                     continue}
                fmt.Println("New client")
                go handleClient(conn)
        }}
```

```
func handleClient(conn net.Conn) {
        defer conn.Close()

        conn.Write([]byte("Hello Client!"))

        buffer := make([]byte, 1024)
        n, err := conn.Read(buffer)
        if err != nil {
                fmt.Println("Error:", err)
                return}

        // procesarea  datelor primate de la client
        fmt.Printf("Received: %s\n", buffer[:n])
}
```

Implementarea clientului

```
package main
import (
	"fmt"
	"net")
func main() {
	// conectarea la server
	conn, err := net.Dial("tcp",
"localhost:8081")
	//prelucrarea erorii
	defer conn.Close()

	// citirea datelor trimise de server
	buf := make([]byte, 1024)
	_, err = conn.Read(buf)
      //prelucrarea erorii
	fmt.Printf("Received: %s\n", buf)
```

```
// trimiterea datelor catre server

	var mes string
	fmt.Scanf("%s\n", &mes)
	data := []byte(mes)
	_, err = conn.Write(data)
      // prelucrarea erorii

} //end  main
```

conn.Read si **conn.Write** sunt folosite pentru comunicarea dintre server si client. Ele folosesc date de tip **[]byte** (pentru alte tipuri trebuie facuta conversia)

Implementarea clientului

```
package main
import (
        "fmt"
        "net")
func main() {
        // conectarea la server
        conn, err := net.Dial("tcp", "localhost:8081")
        if err != nil {
                fmt.Println("Error:", err)
                return}
        defer conn.Close()

        // citirea datelor trimise de server
        buf := make([]byte, 1024)
        _, err = conn.Read(buf)
        if err != nil {
                fmt.Println(err)
                return}
        fmt.Printf("Received: %s\n", buf)
```

```
// trimiterea datelor catre server
        var mes string
        fmt.Scanf("%s\n", &mes)
        data := []byte(mes)
        _, err = conn.Write(data)
        if err != nil {
                fmt.Println("Error:", err)
                return
        }
} \\ end  main
```

```
func handleClient(conn net.Conn) { …}
```

```
func main() {
        // Listen for incoming connections
        listener, err := net.Listen("tcp", "localhost:8080")
        fmt.Println("Server is listening")

        defer listener.Close()

        for {
                // Accept incoming connections
                conn, err := listener.Accept()

               fmt.Println("New client")

                // Handle client connection in a goroutine
                go handleClient(conn)
        }
}
```

```
> go run simpleserver.go
Server is listening
New client
Received: ioana
New client
Received: ana
New client
New client
Received: ion
Received: petre
```

```
> go run simpleclient.go
Hello client
ioana
```

```
> go run simpleclient.go
Hello client
ana
```

```
> go run simpleclient.go
Hello client
ion
```

```
> go run simpleclient.go
Hello client
petre
```

Clientii sunt prelucrati concurent!

Implementarea clientului

```
package main
import (
    "fmt"
    "net")
func main() {
    // conectarea la server
    conn, err := net.Dial("tcp", "localhost:8081")
    //prelucrarea erorii
    defer conn.Close()

    // citirea datelor trimise de server
    buf := make([]byte, 1024)
    _, err = conn.Read(buf)
   //prelucrarea erorii
    fmt.Printf("Received: %s\n", buf)
```

```
// trimiterea datelor catre server iterativ

    var mes string
    fmt.Scanf("%s\n", &mes)
    for mes != "end" {
        data := []byte(mes)
        _, err = conn.Write(data)
       //prelucrarea erorii
        fmt.Scanf("%s\n", &mes)
    }
} //end  main
```

```
> go run simpleclient.go
Ioana
mesaj1
mesaj2
```

Implementarea serverului

```
package main
import (
        "fmt"
        "net")
func main() {
     listener, err := net.Listen("tcp", "localhost:8081")
    //prelucrarea erorii
     fmt.Println("Server is listening")
        defer listener.Close()
        for {
                        conn, err := listener.Accept()
                        //prelucrarea erorii
                        fmt.Println("New client")
                        go handleClient(conn)
        }}
```

```
func handleClient(conn net.Conn) {
      defer conn.Close()

   // primirea mesajelor trimise de client
      buffer := make([]byte, 1024)
    for {
      n, err := conn.Read(buffer)
      if err != nil {
                  fmt.Println("Error:", err)
                  return}
     // procesarea datelor primite de la client
      fmt.Printf("Received: %s\n", buffer[:n])
}}
```

```
func handleClient(conn net.Conn) {
        defer conn.Close()

   // primirea mesajelor trimise de client
        buffer := make([]byte, 1024)
     for {
        n, err := conn.Read(buffer)
        if err != nil {
                        fmt.Println("Error:", err)
                        return}
        // procesarea datelor primite de la client
        fmt.Printf("Received: %s\n", buffer[:n])
}}
```

```
> go run simpleserverl.go
Server is listening
Received: ioana1
Received: ioana2
Received: ana1
Received: ioana3
Received: ana2
Received: ana3
Error: EOF
Received: ioana4
Error: EOF
```

```
> go run simpleclientl.go
ioana1
ioana2
ioana3
ioana4
end
```

```
> go run simpleclientl.go
ana1
ana2
ana3
end
```

Clientii sunt prelucrati concurent!

Sabloane (patterns)

- Generator
- Pipeline
- Worker Pool
- Quit Channel
- Multiplexing (FanIn)

Vor fi exemplificate pe modelul producator-consumator.

- https://go.dev/talks/2012/concurrency.slide
- https://reliasoftware.com/blog/golang-concurrency-patterns

Sablonul "generator": o functie care intoarce un canal

```
generator := func(...) <-chan type {
        results := make(chan type)

        go func() {
            defer close(results)
            for  ….  {
                results <-  item
            }
        }()     // gorutina este lansata in interiorul functiei

        return results
    }

changen := generator(…) // crearea canalului
```

Sablonul "generator": o functie care intoarce un canal

```
producer := func(s string) <-chan string {
        results := make(chan string)
        go func() {
            defer close(results)
            for i := 0; i <= 9; i++ {
                results <- s
            }
        }()
        return results
    }


results := producer("A")
```

Sablonul "generator": o functie care intoarce un canal

```
func main() {
producer := func(s string) <-chan string {
        results := make(chan string)
        go func() {
            defer close(results)
            for i := 0; i <= 9; i++ {
                results <- s
            }
        }()
        return results
    }
```

```
consumer := func(results <-chan string) {
        for result := range results {
            fmt.Printf("Received: %s\n", result)}
        fmt.Println("Done receiving!")
    }

    results := producer("A")  // generarea
    consumer(results)   }  // end main
```

Sablonul "pipeline": prelucrare in mai multi pasi

Exemplu: A A A A A   => a a a a a => aa aa aa aa aa

```
data := producer("A")                      // "A"  "A"  "A"
results1 := processdata1(data)         // "a" "a" "a"
results   := processdata2(results1)  // "aa" "aa" "aa"
consumer(results)  }
```

```
processdata1 := func(data <-chan string) <-chan string {
        results := make(chan string)
        go func() {
            defer close(results)
            for item := range data {
                  // procesare data
                results <- strings.ToLower(item)
            }
        }()
        return results}
```

```
processdata2 := func(data <-chan string) <-chan string {
        results := make(chan string)
        go func() {
            defer close(results)
            for item := range data {
                // procesare data
                results <- item + item
            }
        }()
        return results}
```

Sablonul "pipeline": prelucrare in mai multi pasi

```
func main() {
producer := func(s string) <-chan string {
        results := make(chan string)
        go func() {
            defer close(results)
            for i := 0; i <= 9; i++ {
                results <- s
            }
        }()
        return results}

processdata1 := func(data <-chan string) <-chan string {
        results := make(chan string)
        go func() {
            defer close(results)
            for item := range data {
                results <- strings.ToLower(item) // process item
            }
        }()
        return results}
```

```
processdata2 := func(data <-chan string) <-chan string {
        results := make(chan string)
        go func() {
            defer close(results)
            for item := range data {
                results <- item + item // process item
            }
        }()
        return results}

   consumer := func(results <-chan string) {
           for result := range results {
               fmt.Printf("Received: %s\n", result)}
           fmt.Println("Done receiving!")}

      data := producer("A") // "A"  "A"  "A"
       results1 := processdata1(data)  // "a" "a" "a"
       results := processdata2(results1) // "aa" "aa" "aa"
       consumer(results)  }  // end main
```

Sablonul "quit channel": gorutinele copil primesc mesaj de terminare de la parinte

```
func main() {
    done := make(chan bool)
    producer := func(s string, done chan bool) <-chan string {
        results := make(chan string)
        go func() {
            defer close(results)
            for {
                select {
                case <-done:
                    return
                case results <- s:
                }
            }
        }()
        return results
    }
```

```
consumer := func(results <-chan string) {…}

results := producer("A", done) //
go consumer(results)

time.Sleep(1 * time.Second)
done <- true  // generarea este intrerupta
}
```

Sablonul Piscina ("worker pool"): prelucrarile sunt facute de mai multe gorutine "worker"
care sunt sincronizate cu WaitGroup

```
 worker := func(i int, results <-chan string,  wg *sync.WaitGroup) {
            defer wg.Done()
            for result := range results { // se prelucreaza result
                                          fmt.Printf("%d received: %s\n", i, result)}
            fmt.Println("Done!")
           }


   results := producer("A")
   var wg sync.WaitGroup
   var nrworker int = 3              // results e prelucrat de 3 gorutine "worker"
    for i := 0; i < nrworker; i++ {
              wg.Add(1)
              go worker(i+1,results, &wg)} //  consumatorii sunt gorutine "worker"
   wg.Wait()
```

Sablonul FanIn (multiplexing): se reunesc concurrent mai multe canale

```
func fanIn(c1, c2 <-chan string) <-chan string {
    c := make(chan string)
    go func() {
        for {
            select {
            case s := <-c1:  c <- s
            case s := <-c2:  c <- s
            }
        }
    }()
    return c}
```

ideea generala

Sablonul FanIn (multiplexing): se reunesc concurrent mai multe canale

- Doua tipuri de producatori , fiecare are canalul lui, se obtine un canal comun folosind fanIn

```
var nA, nB int
    fmt.Print("nA=")
    fmt.Scan(&nA)
    fmt.Print("nB=")
    fmt.Scan(&nB)

    c1 := producer("A", nA)
    c2 := producer("B", nB)
    results := fanIn(c1, c2)
    consumer(results)
```

```
func fanIn(c1, c2 <-chan string) <-chan string {
  c := make(chan string)
  go func() {
    for {
      select {
      case s := <-c1:  c <- s
      case s := <-c2:  c <- s
      }
    }
  }()
  return c}
```

Functia generala nu functioneaza corect, se pierd date!

Sablonul FanIn (multiplexing): se reunesc concurrent mai multe canale

```
fanIn := func(c1, c2 <-chan string) <-chan string {
        c := make(chan string)
        go func() {
            defer close(c)
for (c1 != nil) || (c2 != nil) {
                select {
                case s, ok1 := <-c1:
                    if ok1 {c <- s}
                        else {c1 = nil}
                case s, ok2 := <-c2:
                    if ok2 {c <- s}
                        else {c2 = nil}
                }}}()
        return c}
```

https://reliasoftware.com/blog/golang-concurrency-patterns

```
func fanIn(c1, c2 <-chan string) <-chan string {
  c := make(chan string)
  go func() {
    for {
      select {
      case s := <-c1:  c <- s
      case s := <-c2:  c <- s
      }
    }
  }()
  return c}
```

Functia generala nu functioneaza corect, se pierd date!

Sablonul FanIn (multiplexing): se reunesc concurrent mai multe canale

```
fanIn := func(c1, c2 <-chan string) <-chan string {
        c := make(chan string)
        go func() {
            defer close(c)
for (c1 != nil) || (c2 != nil) {
                select {
                case s, ok1 := <-c1:
                    if ok1 {c <- s}
                        else {c1 = nil}
                case s, ok2 := <-c2:
                    if ok2 {c <- s}
                        else {c2 = nil}
                }}}()
        return c}
```

Canalele cu valoarea nil nu sunt selectate niciodata (spre deosebire de cele inchise)!

"Since communication on nil channels can never proceed, a select with only nil channels and no default case blocks forever."
https://go.dev/ref/spec#Select_statements

https://reliasoftware.com/blog/golang-concurrency-patterns