Esercitazione 4
Uso di go avanzato: trattamento delle priorità

22 novembre 2024

Go

Realizzazione di politiche basate su priorità

Esempio: il ponte a senso unico alternato

Si consideri un ponte che collega le 2 rive di un fiume (riva Nord, riva Sud).

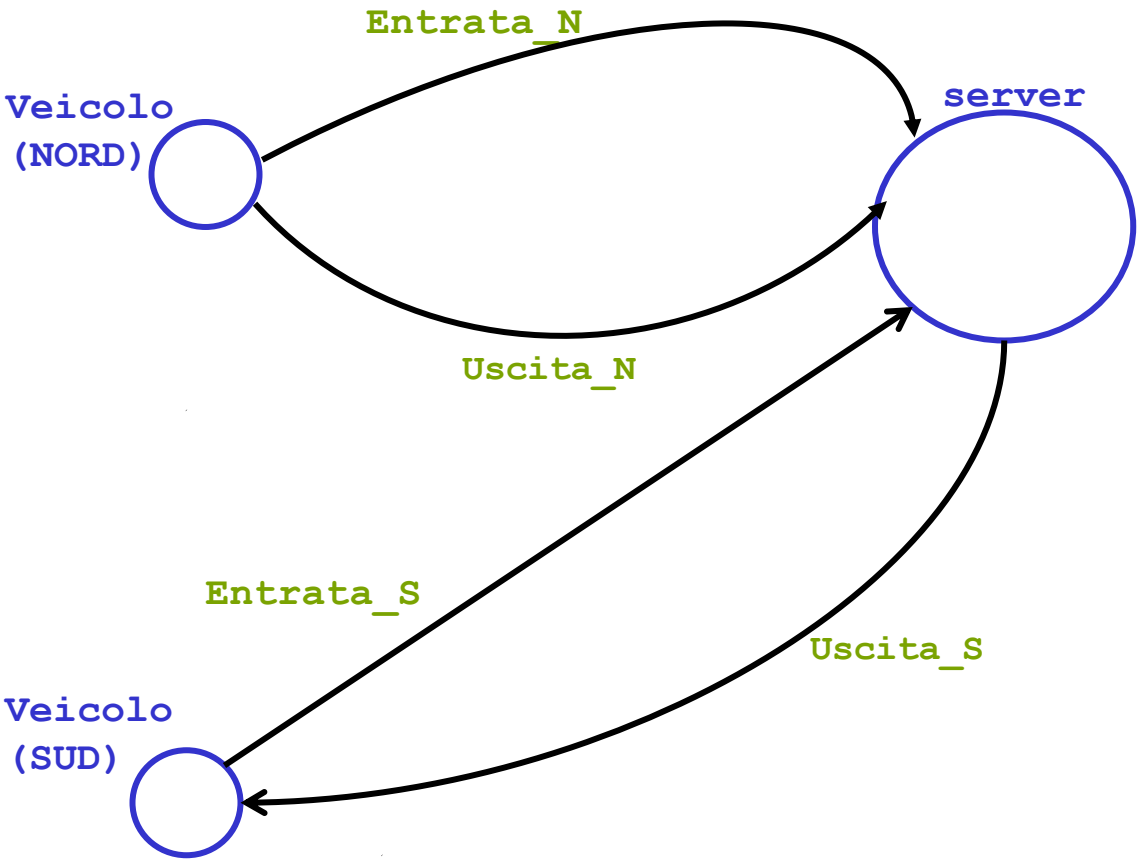
Il ponte è stretto, pertanto può essere percorso solo a senso unico alternato. Ciò significa che un veicolo non può accedere al ponte se su di esso vi è almeno un veicolo in direzione opposta.

Inoltre il ponte ha una capacità limitata a **MAX**, che esprime il massimo numero di veicoli che possono attraversarlo contemporaneamente.

Si realizzi un programma GO, nel quale la gestione del ponte sia affidata ad una goroutine server, e ogni veicolo sia rappresentato da una goroutine concorrente.

Il server deve controllare accessi e uscite dal ponte rispettando i vincoli dati ed, inoltre, **nell'accesso al ponte** dia la **priorità ai veicoli provenienti da NORD**.

Impostazione



Impostazione

- **Quali e quanti canali?**
  - Un canale per le richieste di accesso da NORD(veicoli -> server):

```
var entrataN chan int
```

  - Un canale per le richieste di accesso da SUD(veicoli -> server):

```
var entrataS chan int
```

  - Un canale per le richieste di uscita da NORD (veicoli -> server):

```
var uscitaN chan int
```

  - Un canale per le richieste di uscita da SUD (veicoli -> server):

```
var uscitaS chan int
```

Priorità

Un veicolo che accede da sud non può entrare se ci sono processi da nord in attesa.

Problema: come fare per privilegiare le richieste di accesso da NORD?

E' necessario considerare **il numero dei processi in attesa** per ogni livello di priorità.

- Con gli strumenti **go** visti finora: dovremmo accettare ogni richiesta di entrata e valutare solo dopo la receive se la richiesta :
  - può essere **accettata,** perchè soddisfa tutti i vincoli, oppure
  - deve essere messa in attesa, ad esempio **perchè ci sono processi più prioritari in attesa**. → gestione di strutture dati interne al server che registrino anche i processi in attesa nei diversi livelli di priorità.
  - Problema dei **risvegli**: quando e quanti/quali processi riattivare? In che ordine?

- **Alternativa**: uso della funzione **len()**

Funzione len

func len(v Type) int

La funzione built in **len** restituisce la «lunghezza» di v, dipendentemente dal suo tipo Type.

Tipi possibili:

- **Array**: restituisce il numero di elementi di v;
- **Pointer to array**: restituisce il numero di elementi in *v;
- **Slice, or map**: restituisce il numero di elementi di v; se v ha valore nil, len(v) restituisce zero.
- **String**: restituisce il numero di bytes in v.
- **Chan:** restituisce il **numero di messaggi accodati** (non ancora letti) nel buffer del canale; se v ha valore nil, len(v) restituisce zero.

Chan & len()

In go, **per ogni canale bufferizzato**, è possibile ottenere il numero di messaggi accodati nel canale tramite la funzione di libreria **len()**:

```
var C=make(chan int, 100)      // il canale è bufferizzato
                               // ->  send asincrone
…
Num_msg:=len(C)   // Num_msg: numero dei messaggi
                  // attualmente accodati nel canale C
```

Applichiamo al nostro problema:

Quali e quanti canali?

- Due canali **bufferizzati** per le richieste di accesso da NORD e da SUD:

```
var entrataN = make(chan int, MAXBUFF)
var entrataS = make(chan int, MAXBUFF)
```

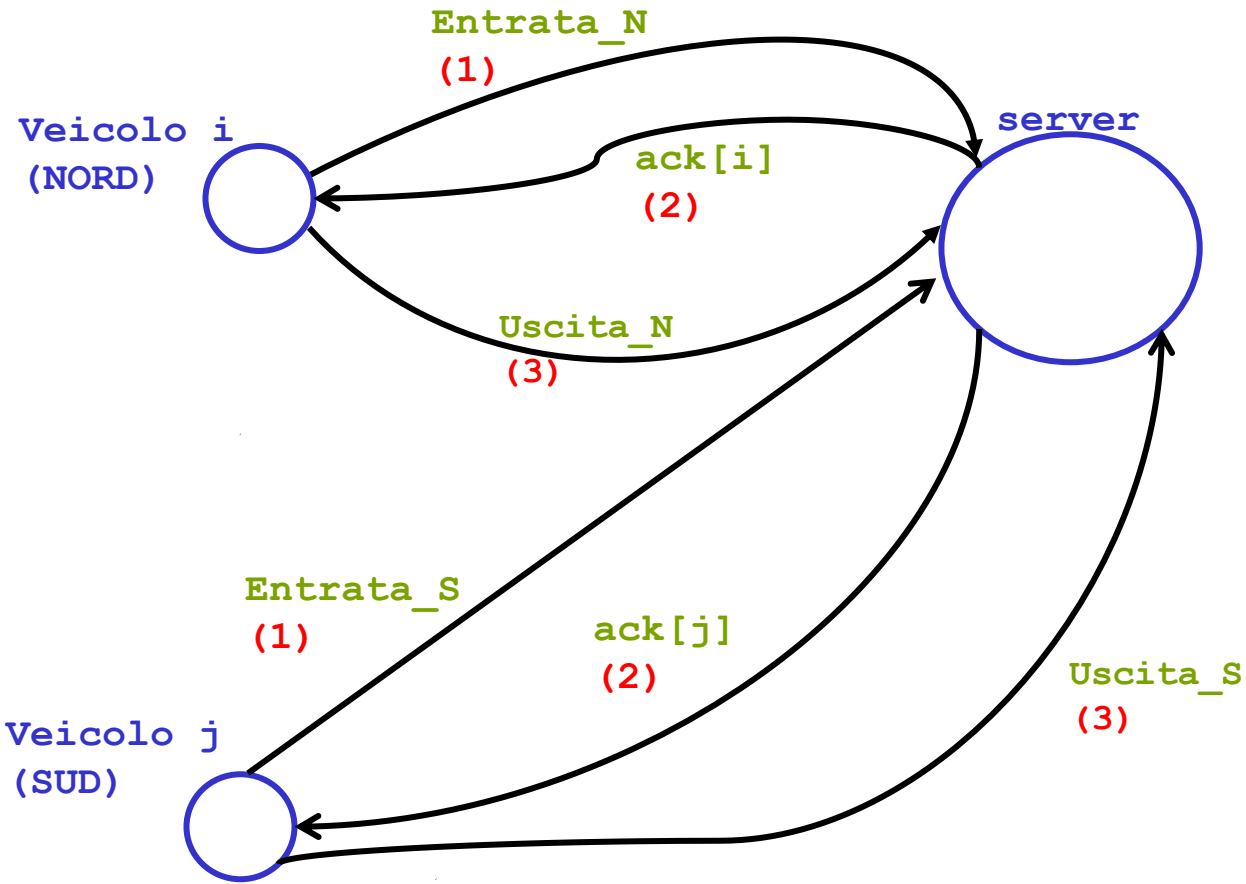
- Due canali (la sincronizzazione è ininfluente, ad es. non bufferizzati) per le richieste di uscita:

```
var uscitaN = make(chan int)
var uscitaS = make(chan int)
```

- Inoltre, poichè la comunicazione su entrataN e entrataS è di tipo **asincrono**, è necessario prevedere un canale per ogni veicolo (server-> veicolo) per la sincronizzazione server->clienti.

```
var ACK[MAXPROC]chan int
```

Impostazione



Soluzione

```
package main
import (
     "fmt"
     "math/rand"
     "time" )
var done = make(chan bool)
var termina = make(chan bool)
var entrataN = make(chan int, MAXBUFF)
var entrataS = make(chan int, MAXBUFF)
var uscitaN = make(chan int)
var uscitaS = make(chan int)
var ACK [MAXPROC]chan int //risposte client
```

```
func veicolo(myid int, dir int) {
     var tt int
     tt = rand.Intn(5) + 1
     time.Sleep(time.Duration(tt) * time.Second)
     if dir == N {
             entrataN <- myid // send asincrona
             <-ACK [myid]     // attesa x sincronizzazione
             tt = rand.Intn(5)
             time.Sleep(time.Duration(tt) * time.Second)
             uscitaN <- myid // send sincrona
     } else {
             entrataS <- myid
             <-ACK [myid] // attesa x sincronizzazione
             tt = rand.Intn(5)
             time.Sleep(time.Duration(tt) * time.Second)
             uscitaS <- myid  // send sincrona
     }
     done <- true }
```

```
func server() {
var contN int = 0
var contS int = 0
for {
   select {
   case x := <-when((contN < MAX) && (contS == 0), entrataN): // da nord, max prio
         contN++
         ACK[x] <- 1 // fine "call"
   case x := <-when((contS < MAX) && (contN == 0) && (len(entrataN) == 0), entrataS):
         contS++
         ACK[x] <- 1 // fine"call"
   case <-uscitaN:
         contN--
   case <-uscitaS:
         contS--
   case <-termina: //tutti i processi clienti hanno finito
         done <- true
         return
   }
}}
```

```
func main() {
      var VN int
      var VS int
      fmt.Printf("\n quanti veicoli NORD (max %d)? ", MAXPROC/2)
      fmt.Scanf("%d", &VN)
      fmt.Printf("\n quanti veicoli SUD (max %d)? ", MAXPROC/2)
      fmt.Scanf("%d", &VS)
      for i := 0; i < VS+VN; i++ {
            ACK[i] = make(chan int, MAXBUFF)
      }
      rand.Seed(time.Now().Unix())
      go server()
      for i := 0; i < VS; i++ {
            go veicolo(i, S)
      }
      for j := i; j < VN; j++ {
            go veicolo(j, N)
      } // continua
```

```
//..continua
     for i := 0; i < VN+VS; i++ {
             <-done
     }
     termina <- true
     <-done
     fmt.Printf("\n HO FINITO ")
}

func when(b bool, c chan int) chan int {
     if !b {
             return nil
     }
     return c
}

// fine programma
```

Esercizio 1 (da svolgere):

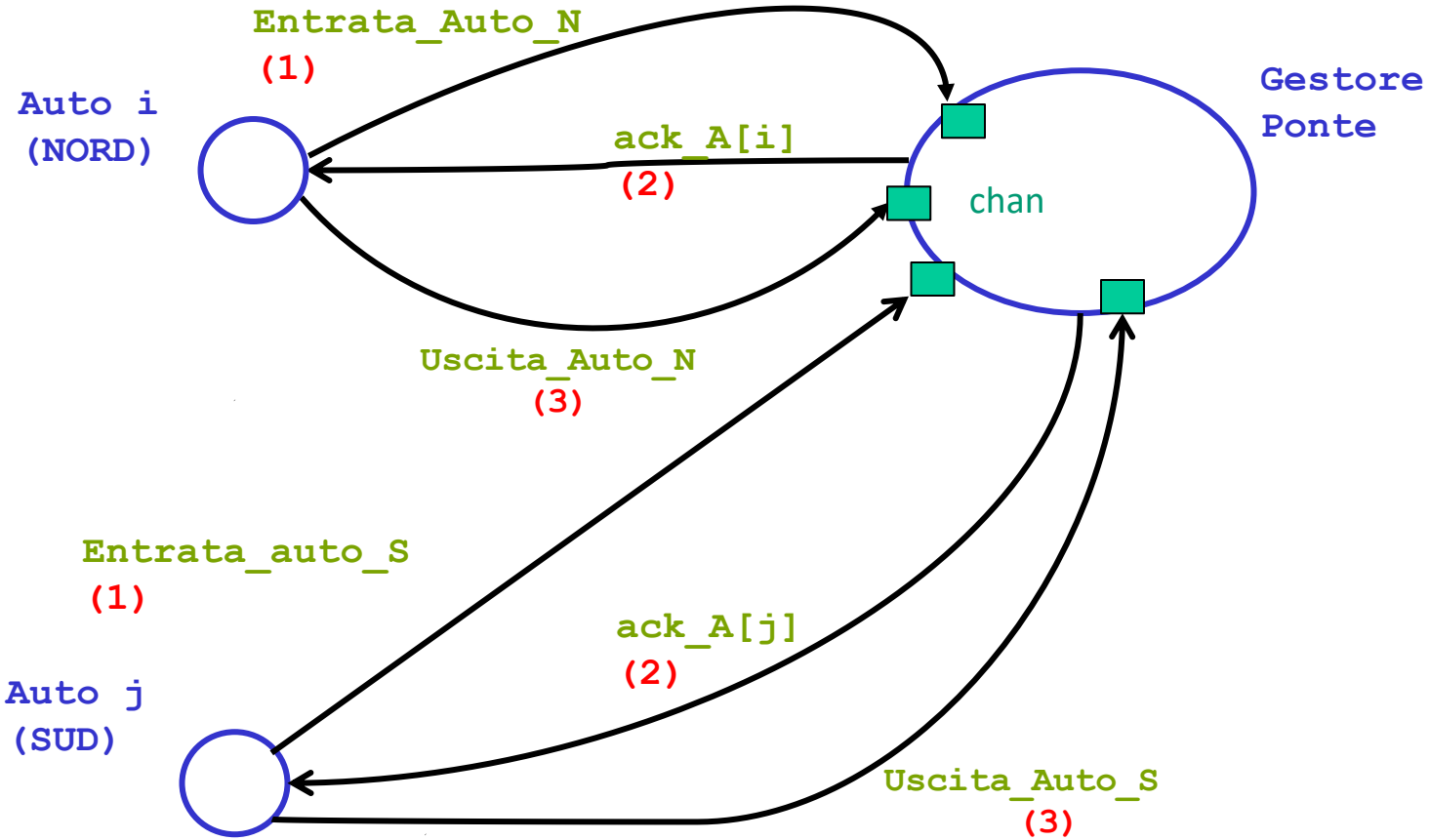
Si consideri un ponte che collega le due rive di un fiume.

- Al ponte possono accedere due tipi di utenti: **auto** e **pedoni**.
- Il ponte ha una **capacita` massima MAX** che esprime il numero massimo di «**persone equivalenti**» che possono transitare contemporaneamente su di esso; a questo scopo si assuma che ogni pedone equivalga a una «persona equivalente», e ogni auto equivalga a 10 «persone equivalenti».
- Il ponte è talmente stretto che il transito di un'auto in una particolare direzione d impedisce l'accesso al ponte di ogni altro utente (auto o pedone) in direzione opposta a d.

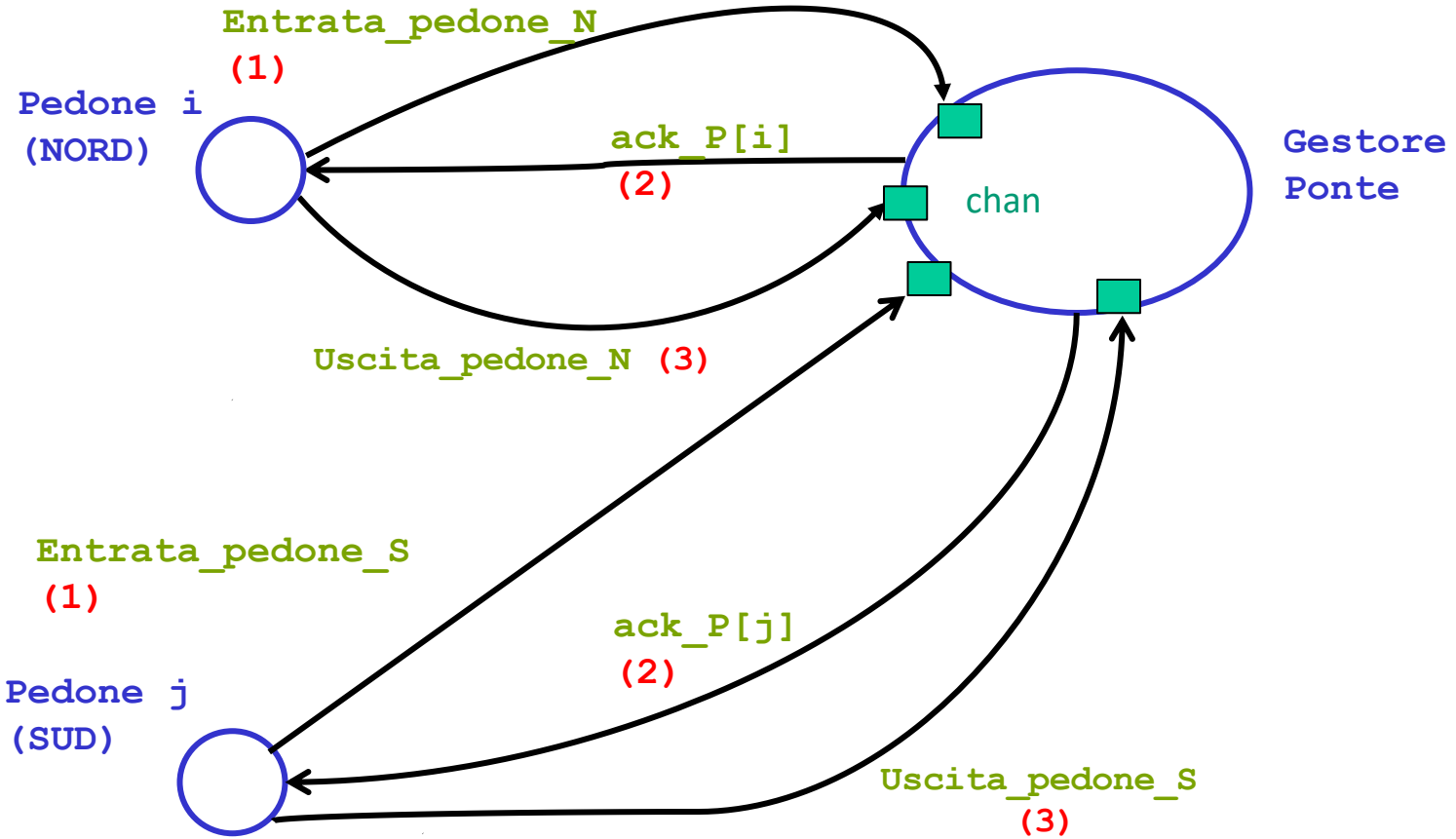
  Realizzare una politica di sincronizzazione delle entrate e delle uscite dal ponte che tenga conto delle specifiche date, e che **nell'accesso al ponte dia la precedenza ai pedoni rispetto alle auto:**

  - **tra i pedoni,** vengano favoriti quelli **provenienti da Sud;**
  - **tra le auto** vengano favorite quelle **provenienti da Sud.**

Impostazione

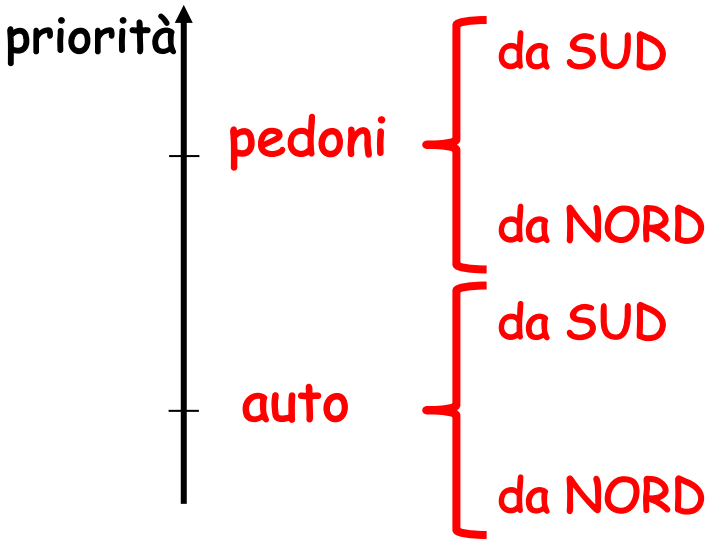


Impostazione



Vincoli

- **Capacità**: l'accesso al ponte di un utente non deve causare il superamento del limite massimo MAX
- **Direzione**: se c'è almeno un'auto sul ponte in direzione d, nessun utente può percorrere il ponte in direzione opposta a d.
- **Priorità**: un utente non può accedere al ponte se vi sono utenti più prioritari in attesa:



Esercizio 2 (primo appello AA 2019-20)

Si consideri lo stabilimento di produzione di una casa automobilistica.

La produzione di ogni automobile avviene attraverso varie fasi in sequenza e si realizza in modo completamente automatico, grazie all'attività di sistemi automatici e robotici che eseguono le varie fasi.

Si consideri, in particolare, la fase di **montaggio delle ruote**, che prevede l'installazione di **4 cerchi e 4 pneumatici** per ogni auto.

Nello stabilimento considerato vengono prodotti **2 modelli di auto** (**modelloA** e **modelloB**) che montano ruote e cerchi diversi. In particolare:

- Il modello A monta cerchi del tipo **CA** e pneumatici del tipo **PA**
- Il modello B monta cerchi del tipo **CB** e pneumatici del tipo **PB**

La fornitura di cerchi e pneumatici viene eseguita da 4 nastri trasportatori dedicati, che consegnano tali elementi a un unico **deposito**, da quale verranno successivamente prelevati dai sistemi robotici dedicati al montaggio.

Il deposito è caratterizzato da capacità limitate pari a :

- **MaxP**: numero massimo di pneumatici (di qualunque tipo) che possono essere contemporaneamente stoccati nel deposito;
- **MaxC**: numero massimo di cerchi (di qualunque tipo) che possono essere contemporaneamente stoccati nel deposito;

In particolare, gli pneumatici vengono immessi nel deposito da **due nastri trasportatori**:

- uno dedicato al trasporto degli pneumatici PA
- l'altro per il trasporto degli pneumatici PB.

Analogamente, i cerchi vengono immessi da due nastri trasportatori:

- uno dedicato al trasporto dei cerchi CA;
- l'altro per il trasporto dei CB.

Ogni nastro consegna a istanti **non predicibili** un oggetto alla volta (cerchio o pneumatico) al deposito.

Il montaggio di cerchi e pneumatici è svolto da due sistemi robotici (o robot): uno per il modello A e uno per il modello B.

Ogni robot è attrezzato con **un braccio** in grado di montare sia cerchi che pneumatici.

Per il **montaggio delle ruote di un'auto***, ognuno dei due robot procede quindi come segue. Per ognuna delle 4 quattro ruote:

- preleva un cerchio e lo monta;
- preleva uno pneumatico e lo monta.

Il deposito dovrà gestire gli accessi da parte dei nastri e dei robot in modo da soddisfare i vincoli dati, ed inoltre i seguenti vincoli di priorità:

- nell'immissione **venga data la precedenza a pneumatici e cerchi del tipo corrispondente al modello di auto** con il minor numero di montaggi ruote completati;
- nel prelievo **venga data la precedenza a pneumatici e cerchi del tipo corrispondente al modello di auto** con il minor numero di montaggi ruote completati;

Realizzare un'applicazione concorrente da sviluppare in go nella quale i nastri trasportatori e i robot siano rappresentati da goroutine concorrenti, ed il deposito sia una risorsa a disposizione degli utenti.

La sincronizzazione tra le goroutine dovrà tenere conto di tutti i vincoli dati.

L'applicazione dovrà terminare dopo il montaggio delle ruote di TOT auto.

**** Per "montaggio ruote" si intende il montaggio di tutte e 4 le ruote (cerchi e pneumatici) di un'auto.***

Impostazione: goroutine e servizi

- **Server**:
  - **deposito**: è il **server** che gestisce cerchi e pneumatici pronti per essere montati; offre i servizi:
    - inserimenti di cerchi e pneumatici: **insCA**, **insCB**, **insPA**, **insPB**
    - prelievi di cerchi e pneumatici: **prelCA**, **prelCB**, **prelPA**, **prelPB**
- **Clienti:**

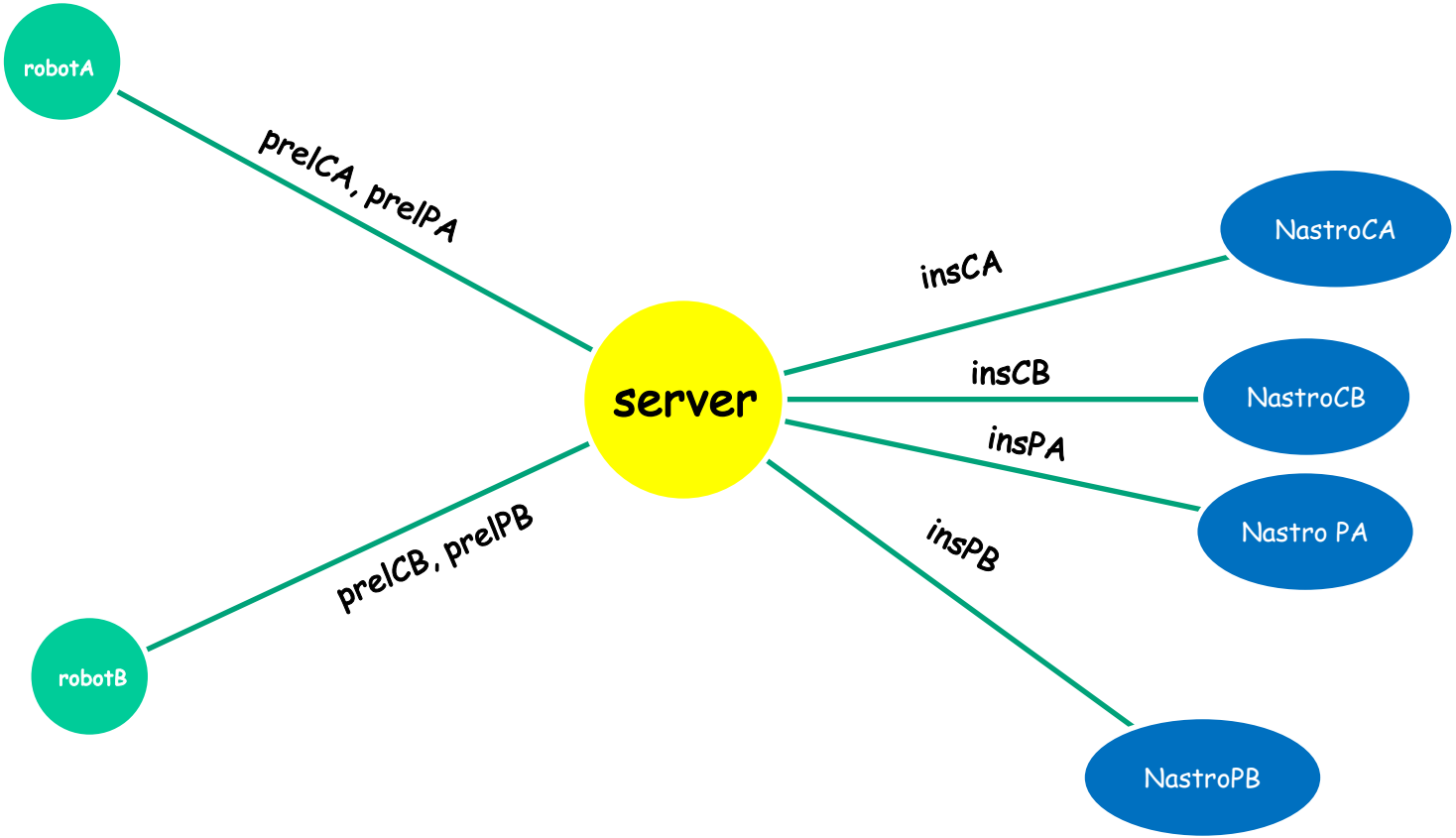
Nastri trasportatori:

  - **nastroCA:** deposita periodicamente un cerchione di tipo A (richiesta al server su insCA)
  - **nastroCB:** deposita periodicamente un cerchione di tipo B (richiesta al server su insCB)
  - **nastroPA:** deposita periodicamente uno pneumatico di tipo A (richiesta al server su insPA)
  - **nastroPB:** deposita periodicamente uno pneumatico di tipo B (richiesta al server su insPB)

Robot:

  - **robotA**: dedicato al montaggio delle ruote delle auto modelloA; per ogni auto ripete per 4 volte:
    - preleva cerchio A (richiesta al server su prelCA), preleva pneumatico A (richiesta al server su prelPA).
  - **robotB**: dedicato al montaggio delle ruote delle auto modelloB; per ogni auto ripete per 4 volte:
    - preleva cerchio B (richiesta al server su prelCB), preleva pneumatico A (richiesta al server su prelPB).

Schema


robotA
prelCA, prelPA
server
insCA
NastroCA
insCB
NastroCB
insPA
Nastro PA
insPB
NastroPB
robotB
prelCB, prelPB