PCD ASS03 (3)

Federico Bravetti
federico.bravetti2@studio.unibo.it

I. Analisi del Problema

Il progetto in questione riguarda lo sviluppo del gioco *Guess the Number*. In particolare sarà utilizzato il linguaggio *Go* per implementare un modello di comunicazione a canali sincroni.

A. Architettura

Si propongono due architetture, una prima versione più semplice riportata in Fig. 1, e una seconda versione con GUI (implementata utilizzando la libreria *fyne*) riportata in Fig. 2. In questo caso la GUI è stata utilizzata sia per la gestione del gioco sia come console per il log delle varie entità.

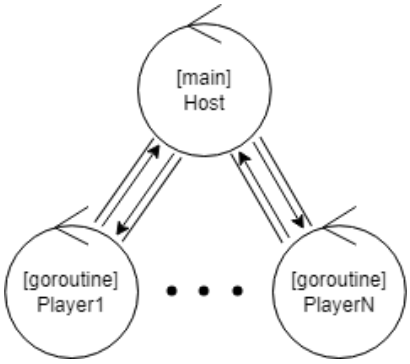

Figure 1. Architettura

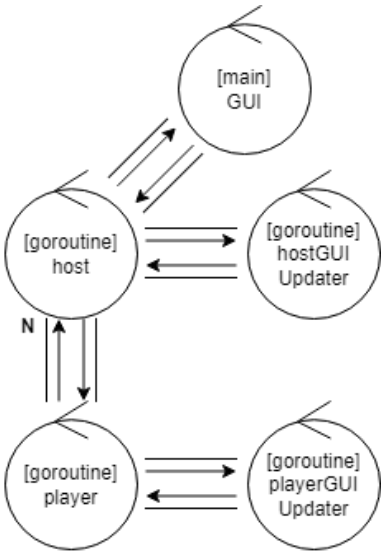

Figure 2. Architettura con GUI

II. Comportamento del Sistema

Di seguito viene analizzato il comportamento del sistema andando a considerare le interazioni tra le varie entità per poi discutere i vantaggi nell'utilizzo di un modello di comunicazione sincrono.

A. Interazione tra Entità

In primo luogo si vuole analizzare l'evoluzione temporale della versione base del gioco, osservando il diagramma riportato in Fig. 3.

In questa versione si è scelto di adottare una struttura molto semplice ed essenziale, di fatti una volta avviato il programma questo viene eseguito immediatamente e per valutare la corretta terminazione della partita si sfrutta la meccanica di chiusura dei canali, non essendo questi più necessari.

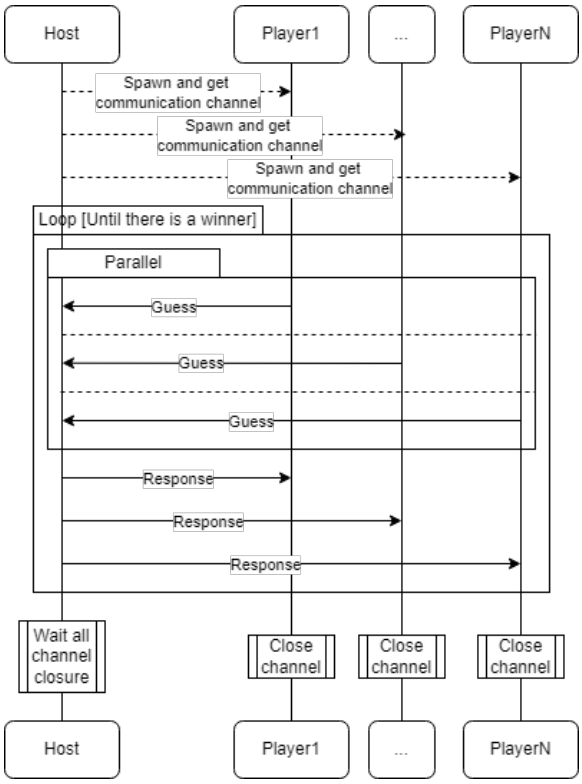

Figure 3. Evoluzione del gioco

Invece nella versione con GUI, il cui diagramma è riportato in Fig. 4, è stata implementata un'interazione più raffinata utilizzando dei messaggi di controllo aggiuntivi.

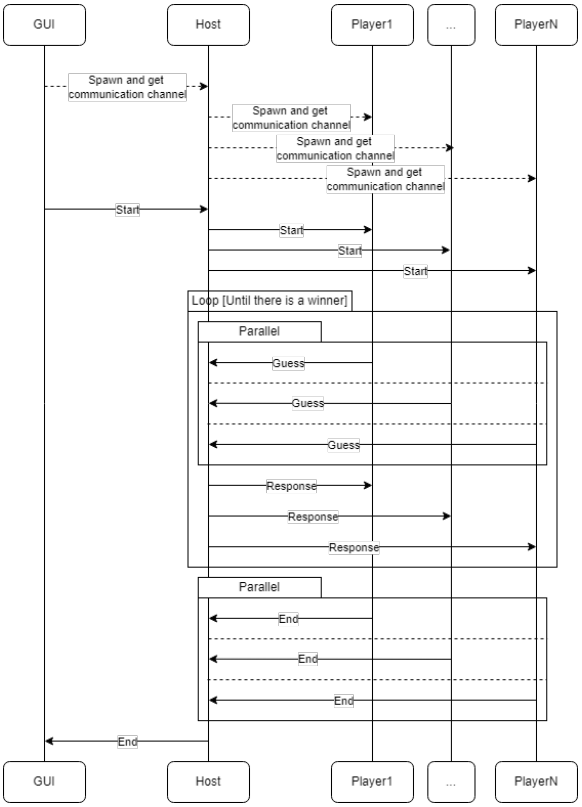

Figure 4. Evoluzione del gioco con GUI

Inoltre, in aggiunta ai messaggi riportati nel grafico, sia l'host che i vari player inviano vari messaggi alla propria GUI di log attraverso un canale dedicato.

B. *Vantaggi del modello Sincrono*

Uno dei vantaggi e delle meccaniche che è possibile utilizzare nel caso di un modello a scambio di messaggi sincrono è quella riportata nel codice proposto di seguito.

Listing 1. Esempio di comunicazione

```
1 func main() {
2  ch := makePlayer()
3  ch <- "Ping"
4  for msg := range ch {
5     switch msg {
6     case "Pong":
7        ch <- "Ping"
8     }
9  }
10 }
11
12 func makePlayer() chan string {
13  ch := make(chan string)
14  go run(ch)
15  return ch
16 }
17
18 func run(ch chan string) {
19  for msg := range ch {
20     switch msg {
21     case "Ping":
22        ch <- "Pong"
23     }
24  }
25 }
```

In particolare si può notare che i messaggi inviati a righe 3 e 7, essendo la send bloccante, saranno sicuramente ricevuti nella goroutine run e, viceversa, quello inviato a riga 22 sarà sicuramente ricevuto dalla main goroutine, quindi non si genera concorrenza nella ricezione dei messaggi.

Questa feature è stata utilizzata in diversi punti del codice essendo il programma strutturato unicamente con canali one-to-one in cui si ha sempre una struttura Request-Response.

Nel caso in cui si ha la necessità di raggruppare messaggi provenienti da diversi canali per elaborarli in un solo punto, ad esempio nel caso di più player e un solo host che deve sincronizzare i vari turni, è necessario fare attenzione all'implementazione che si adotta.

Infatti il codice riportato di seguito fa uso di una tecnica per aggregare messaggi che consiste nell'associare ad ogni entità Player un'entità Aggregator che rilancia i messaggi in un canale di *fanIn*.

Listing 2. Problema di concorrenza

```
1 func main() {
2  aggregatedPong := make(chan string)
3  players := []chan
   string{makePlayer(), makePlayer()}
4  aggregatePong(aggregatedPong,
   players)
5  pong := 0
6  for msg := range aggregatedPong {
7     switch msg {
8     case "Pong":
9        pong++
10       if pong == len(players) {
11          pong = 0
12          for _, ch := range players {
13             ch <- "Ping"
14          }
15       }
16     }
17  }
18 }
19
20 func aggregatePong(aggregatedPong
   chan string, players []chan string)
   {
26   for _, ch := range players {
27    ch <- "Ping"
28    go func(c chan string) {
29       for msg := range c {
30          aggregatedPong <- msg
31       }
32    }(ch)
33  }
```

Come mostrato in Fig. 5 questo crea concorrenza tra l'Aggregator e il Player, infatti il messaggio di Ping inviato dall'Host può essere intercettato da entrambi causando anche possibili deadlock.

La soluzione che è stata adottata fa uso della libreria di utilità *reflect* che evita questo problema.

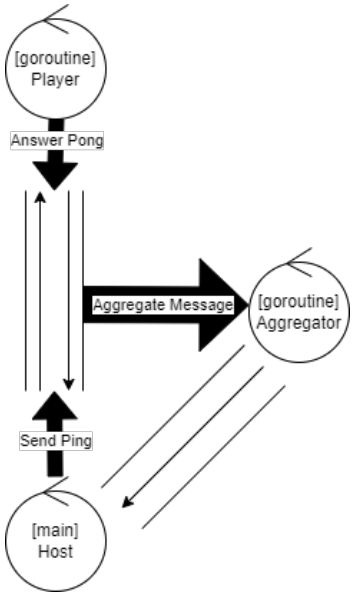

Figure 5. Concorrenza

NOTA: nella versione base del programma l'unica garanzia che si ha nella ricezione di tutti gli eventi di chiusura dei canali è data dalla fairness della funzione Select della libreria reflect.

> "Select executes a select operation described by the list of cases. Like the Go select statement, it blocks until at least one of the cases can proceed, makes a uniform pseudo-random choice, and then executes that case."

Che, in base a quanto quanto riportato nella documentazione del metodo che è qui sopra citata, dovrebbe essere assicurata.