

Fury Go – come spedire una struttura in Go – pt.1

Toast: *Che stai facendo?*

Dag: *Prego.*

Toast: *Chi preghi?*

Dag: *Chiunque ci ascolti.*

Nell'ultimo articolo avevo giurato che l'argomento **Fast IPC** era, "almeno momentaneamente", chiuso. Poi, mentre riguardavo, per l'ennesima volta, lo stupendo **Mad Max: Fury Road** del Maestro **George Miller**, ho avuto un *flash* (sì, ma non vi preoccupate, è durato solo un microsecondo, mentre guardo un film sono sempre molto concentrato). Il *flash* era questo: "E se ripetessi i test IPC socket usando il Go?" L'idea era intrigante e, alla fine, ho ceduto a me stesso. E così ho anche capito che riesco, con grande facilità, a non mantenere le promesse... *avrò mica un gran futuro come politico? ah ah ah.*



...e se provassi a farlo con il Go?...

E allora veniamo al dunque: il titolo qui sopra *“come spedire una struttura in Go”* è un po’ fuorviante riguardo alla premessa iniziale: l’idea è ripetere alcuni *benchmark* usando il **Go** (**Golang** per gli amici) però per farlo bisogna scontrarsi un po’ con una delle differenze che ha questo linguaggio rispetto al **C** (e al C++); come ricorderete (e se no potreste fare una rapida rilettura **di quel vecchio post**) i test erano basati sull’invio *“a raffica”* di molti messaggi (2000000!) composti così:

```
// struttura Data per i messaggi
typedef struct { unsigned long index; // indice dei dati char
text[1024]; // testo dei dati
} Data;
```

La scelta di questa struttura non era casuale: avrei potuto, più semplicemente, inviare solo dei messaggi di testo, ma avevo deciso di inviare dati complessi (*“Data”* contiene solo due campi ma potrebbe contenerne moltissimi) per rimarcare che con il **C** è usuale trattare dati di qualsiasi tipo, e chi riceve un messaggio *“complesso”* lo può ricostruire semplicemente depositandolo in una variabile dello stesso tipo (*ah, la potenza del C...*). E poi, grazie alla struttura *“Data”* ho potuto anche mostrare come gestire un indice dei messaggi, il che non guasta mai.

E il **Go**? Come ben sapete (e ne ho parlato qui) il **Go** è un vero linguaggio ad alto livello, con tutti i pro e i contro che questo comporta. Tra i pro c’è, ovviamente, il fatto che è possibile scrivere applicazioni anche complesse con notevole semplicità e compattezza, sicuramente più del **C** (e C++). Però, quando si tratta di maneggiare dati a livello base (o meglio *binario*) il **Go** entra un po’ in difficoltà, e questo è il caso che stiamo trattando: spedire (e ricevere) messaggi complessi (strutture) non è per nulla semplice e scontato come lo è per il nostro amato **C**. Ma è, comunque, possibile: e tra poco vedremo come e con quali prestazioni.

E ora, bando alle ciance, facciamo cantare il codice! Vediamo

come sono i nostri *reader.go* e *writer.go* (equivalenti, più o meno, ai *reader.c* e *writer.c* **visti qui**). Per eseguire il *benchmark* è presente anche l'onnipresente *processes.c*, che vi risparmio perché è rimasto invariato. Vai col codice!

```
// reader.go - main processo figlio: è un reader (un server)
// su IPC socket
package main import ( "encoding/gob" "fmt" "net" "os" "time"
) // struttura Message per i messaggi
type Message struct { Index int // indice dei dati Text string
// testo dei dati
} // funzione main
func main() { // start ascolto sul file di scambio "myipcs"
(con UNIX domain socket) fmt.Printf("processo %d partito
(reader)\n", os.Getpid()) addr := net.UnixAddr{Name:
"./myipcs", Net: "unix"} lner, err := net.ListenUnix("unix",
&addr) if err != nil { // errore listen fmt.Println(err)
return } // prenoto la chiusura del listener e rimuovo
(eventualmente) il file di scambio defer lner.Close() defer
os.Remove("./myipcs") // accetta connessioni da un writer
entrante conn, err := lner.AcceptUnix() if err != nil { //
errore accept fmt.Println(err) return } // set time di
partenza per calcolare il tempo impiegato start := time.Now()
// loop di lettura messaggi dal writer n_msg := 0 var message
Message for { // set decoder e ricezione dal decoder decoder
:= gob.NewDecoder(conn) decoder.Decode(&message) // test
numero messaggi per forzare l'uscita n_msg++ if n_msg ==
2000000 { // il processo chiude la connessione ed esce per
numero raggiunto fmt.Printf("reader: ultimo messaggio
ricevuto: %s\n", message.Text) fmt.Printf("reader: processo %d
terminato (messaggi=%d tempo totale:%s)\n", os.Getpid(),
n_msg, time.Since(start).Truncate(time.Millisecond).String())
conn.Close() return } }
}
```

```
// writer.go - main processo figlio: è un writer (un client)
// su IPC socket
package main import ( "encoding/gob" "fmt" "net" "os" "time"
) // struttura Message per i messaggi
type Message struct { Index int // indice dei dati Text string
// testo dei dati
} // funzione main
```

```

func main() { // mi assicuro che il writer parta dopo il
reader fmt.Printf("processo %d partito (writer)\n",
os.Getpid()) time.Sleep(100 * time.Millisecond) // connessione
al server remoto sul file di scambio "myipcs" addr :=
net.UnixAddr{Name: "./myipcs", Net: "unix"} conn, err :=
net.DialUnix("unix", nil, &addr) if err != nil { // errore
dial fmt.Println(err) return } // loop di scrittura messaggi
per il reader var message Message message.Index = 0 for { //
test index per forzare l'uscita if message.Index == 2000000 {
// il processo chiude la connessione ed esce per indice
raggiunto fmt.Printf("writer: processo %d terminato (text=%s
messaggi=%d)\n", os.Getpid(), message.Text, message.Index)
conn.Close() return } // compongo il messaggio e lo invio
message.Index++ message.Text = fmt.Sprintf("un-messaggio-di-
test:%d", message.Index) // set encoder e spedizione
dall'encoder encoder := gob.NewEncoder(conn) err =
encoder.Encode(message) if err != nil { fmt.Println("errore di
codifica: ", err) return } }
}

```

Come avrete notato dalla descrizione nella prima linea (e *anche dal codice, spero!*) ho usato per il test gli **IPC socket** (UNIX domain socket). Poi ho ripetuto anche con i **Network Socket**, ma non mostrerò il codice perché è quasi identico. Effettivamente, per la magia del **Go**, il codice è semplicissimo rispetto alla analoga versione in **C** citata (che in questo caso era la versione "fast").

Però la complessità dell'operazione di spedire strutture complesse è mascherata dall'uso di un libreria specializzata, la **encoding/gob**, senza la quale il codice sarebbe molto più complesso (*ebbene sì, una libreria specializzata per una operazione semplice per il C ma complicata per il Go*). E, come vedremo tra poco, le prestazioni non sono eccellenti come ci si aspetterebbe (*spoiler: per colpa della encoding/gob*). Comunque il codice è **stra-commentato**, e credo che possa essere facilmente compreso anche da chi non conosce il **Go**, per cui non mi dilungherò in spiegazioni superflue.

E vabbé, so che siete curiosi, è ora di passare ai risultati!

Di seguito i risultati del *benchmark* in **Go** e, per comparazione, vi riporto anche i risultati della versione **C**:

```
aldo@Linux $ cd go-fastipcsocket/
aldo@Linux $ ./processes
sono il padre (18903): attendo la terminazione dei figli
sono il figlio 1 (18904): eseguo il nuovo processo
sono il figlio 2 (18905): eseguo il nuovo processo
processo 18905 partito (writer)
processo 18904 partito (reader)
writer: processo 18905 terminato (text=un-messaggio-di-
test:2000000 messaggi=2000000)
sono il padre (18903): figlio 18905 terminato (0)
reader: ultimo messaggio ricevuto: un-messaggio-di-
test:2000000
reader: processo 18904 terminato (messaggi=2000000 tempo
totale:13.081s)
sono il padre (18903): figlio 18904 terminato (0)
./processes: processi terminati
```

```
aldo@Linux $ cd fastipcsocket/
aldo@Linux $ ./processes
sono il padre (14990): attendo la terminazione dei figli
sono il figlio 1 (14991): eseguo il nuovo processo
sono il figlio 2 (14992): eseguo il nuovo processo
processo 14991 partito (reader)
processo 14992 partito (writer)
writer: processo 14992 terminato (text=un-messaggio-di-
test:2000000 messaggi=2000000)
sono il padre (14990): figlio 14992 terminato (0)
reader: ultimo messaggio ricevuto: un-messaggio-di-
test:2000000
reader: processo 14991 terminato (messaggi=2000000 tempo CPU:
3.309 - tempo totale:3.309s)
sono il padre (14990): figlio 14991 terminato (0)
./processes: processi terminati
```

Ebbene sì, per trattare 2000000 (!) di messaggi la versione **C** ha bisogno di **10 secondi in meno!** (13.081s vs 3.309s). Però, a questo punto, bisogna fare qualche considerazione:

1. Come versione di riferimento in **C** ho usato quella

*“fast”, visto che il meccanismo della versione **Go** è a *size variabile* ed è, quindi, somigliante. Comunque anche usando la versione **C** *“normal”* la differenza è alta: 8 secondi (13.081s vs 4.823s). (Ho scritto un sacco di benchmark... ma non ve li mostro tutti per non farvi addormentare, ah ah ah).*

2. Vi riporto, per curiosità, i risultati delle versioni con i **Network Socket**: 26.794s per il **Go** e 3.88s per il **C**. Questo era previsto, gli **IPC socket** essendo *“locali”* sono mediamente più veloci dei **Network Socket**, anche se il peggioramento della differenza **Go** vs **C** un po' sorprende.
3. Comunque, non fatevi ingannare dalle prestazioni: in termini assoluti 13.081s (e 26.794s) per 2000000 di messaggi sono, comunque, pochi! **Il Go è un linguaggio veloce!**

E, riguardo al punto 3 appena mostrato qui sopra, vi cito lo spoiler accennato poco fa (*...per colpa della “encoding/gob”...*): nella seconda parte dell'articolo (*in arrivo prossimamente su questi schermi*) vi farò vedere di che cosa è capace il **Go** quando maneggia solo testi.

Ok, per oggi può bastare: per il momento vi saluto, e vi raccomando, come sempre, di non trattenere il respiro in attesa della seconda parte (*potrebbe nuocere gravemente alla vostra salute, ah ah ah*).

Ciao, e al prossimo post!

<https://italiancoders.it/fury-go-come-spedire-una-struttura-in-go-pt-1/>