Better Call Go – come scrivere Server e Client TCP in Go e C – pt.2

signora: *Ed io che pensavo che tutti gli avvocati fossero idioti!*
Jimmy McGill (Saul Goodman): *Solo la metà sono idioti, l'altra sono truffatori.*

Dove eravamo rimasti? Ah, si, **nell'ultimo articolo** vi avevo proposto un confronto tra un **Server TCP** scritto in **C** (*il nostro amato **C***) e uno scritto in **Go** (*la nostra ultima fiamma*). L'ispirazione proveniva dalla *"vulcanicità"* di **Jimmy"Saul"McGill**, l'avvocato *"multi-piattaforma"* delle (oramai) mitiche serie **Breaking Bad** e **Better Call Saul**: Saul ci insegna che non bisogna fossilizzarci sulle nostre abitudini, che dobbiamo esplorare nuovo orizzonti e avere sempre nuove ispirazioni, che bisogna essere dei *"vulcani di idee"*. E quindi: il **C** è un grande e insostituibile linguaggio, ma se in alcuni campi il **Go** ci permette di scrivere più rapidamente e altrettanto bene la stessa applicazione perché non usarlo? Io poi lo trovo divertente e leggero, mi sento veramente a mio agio e senza preoccupazioni e restrizioni quando lo uso.

(…esattamente il contrario, ma proprio il contrario, di quando sono costretto a mettere le mani su codice che è stato scritto da qualche fanatico della programmazione generica in C++… ebbene si, in quel caso riesco a lanciare molte maledizioni con grande ritmo e fluidità. Ma questa è un'altra storia…).



…e allora ti spiego: questo è un Server e quello è un Client…
Veniamo al dunque: i due **Server TCP** li abbiamo già visti. Ci mancano i **Client TCP**, e cominciamo anche stavolta con l'esempio in **C**, che anche in questo caso è, praticamente , identico a quello che avevo descritto **in un vecchio articolo**: vai col codice!

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <arpa/inet.h> #define MYBUFSIZE 1024 int main(int argc, char *argv[])
{ // test argomenti if (argc != 3) { // errore di chiamata printf("%s: numero argomenti errato\n", argv[0]); printf("uso: %s host port [e.g.: %s 127.0.0.1 9999]\n", argv[0], argv[0]); return EXIT_FAILURE; } // creo il socket in modo internet/TCP int cli_sock; if ((cli_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == -1) { // errore socket() printf("%s: non posso creare il socket (%s)\n", argv[0], strerror(errno)); return EXIT_FAILURE; } // preparo la struttura sockaddr_in per
```

```
il server remoto struct sockaddr_in server; memset(&server, 0, sizeof(struct sockaddr_in)); server.sin_family = AF_INET; // set famiglia di indirizzi server.sin_addr.s_addr = inet_addr(argv[1]); // set indirizzo del server server.sin_port = htons(atoi(argv[2])); // set port del server // mi connetto al server remoto if (connect(cli_sock, (struct sockaddr *)&server, sizeof(server)) == -1) { // errore connect() printf("%s: errore connect (%s)\n", argv[0], strerror(errno)); close(cli_sock); return EXIT_FAILURE; } // loop di comunicazione col server remoto for (;;) { // compongo un messaggio per il server remoto char my_msg[MYBUFSIZE]; printf("Scrivi un messaggio per il Server remoto: "); scanf("%s", my_msg); // invio il messaggio al server remoto if (send(cli_sock, my_msg, strlen(my_msg), 0) == -1) { // errore send() printf("%s: errore send (%s)\n", argv[0], strerror(errno)); close(cli_sock); return EXIT_FAILURE; } // ricevo una risposta dal server remoto memset(my_msg, 0, MYBUFSIZE); if (recv(cli_sock, my_msg, MYBUFSIZE, 0) == -1) { // errore recv() printf("%s: errore recv (%s)\n", argv[0], strerror(errno)); close(cli_sock); return EXIT_FAILURE; } // mostro la risposta printf("%s: il server risponde: %s\n", argv[0], my_msg); } // esco con Ok return EXIT_SUCCESS;
}
```

E, come sempre, il codice è **esageratamente commentato** e decisamente auto-esplicativo, per cui non mi dilungherò sulle singole istruzioni e/o gruppi di istruzioni (leggete i commenti! sono li per quello!), ma aggiungerò solo una descrizione del flusso che è quello classico ed elementare di un **Client TCP**:

1. creo il socket in modo internet/TCP
2. preparo la struttura *sockaddr_in* per il server remoto
3. mi connetto al server remoto
4. loop di comunicazione col server remoto
   - compongo un messaggio per il server remoto
   - invio il messaggio al server remoto
   - ricevo e mostro la risposta del server remoto

ovviamente esistono varianti di questa struttura, ma questa è

quella classica. Come avrete notato nel loop di comunicazione c'è l'attesa del messaggio di risposta, e questo ci aiuta, durante l'esecuzione, a osservare il corretto funzionamento della coppia Client/Server che metteremo in prova.

Anche in questo caso (come per il Server) il codice è compatto e ben leggibile, ed è, quindi, facilissimo da manutenere. Ma… e la versione in **Go**? Vediamola!

```
package main import ( "bufio" "fmt" "net" "os"
) func main() { // test argomenti if len(os.Args) != 3 { // errore di chiamata fmt.Printf("%s: numero argomenti errato\n", os.Args[0]) fmt.Printf("uso: %s host port [e.g.: %s 127.0.0.1 9999]\n", os.Args[0], os.Args[0]) return } // mi connetto al server remoto addr := os.Args[1] + ":" + os.Args[2] // set indirizzo (i.e.: "host:port") conn, err := net.Dial("tcp", addr) // set dial con network di tipo TCP if err != nil { // errore di connessione fmt.Println(err) return } // loop di comunicazione col server remoto connrdr := bufio.NewReader(conn) // reader sulla connessione stdinrdr := bufio.NewReader(os.Stdin) // reader sullo standard input for { // compongo un messaggio per il server remoto fmt.Print("Scrivi un messaggio per il Server remoto: ") client_msg, _ := stdinrdr.ReadString('\n') // invio il messaggio al server remoto _, err = conn.Write([]byte(client_msg)) // scrivo sulla connessione if err != nil { // errore di invio fmt.Println(err) return } // ricevo una risposta dal server remoto server_msg, err := connrdr.ReadString('\n') // leggo con il conn reader if err != nil { // errore di lettura fmt.Println(err) return } // mostro la risposta fmt.Printf("%s: il server risponde: %s", os.Args[0], server_msg) }
}
```

Io questa versione la trovo fantastica! I passi sono esattamente gli stessi dell'esempio in **C** (e ci mancherebbe che non lo siano), ma è tutto più semplice e lineare, una vera sciccheria!

E, per chi non conoscesse ancora il **Go**, faccio notare che buona parte della semplificazione è dovuta alla semplicità

intrinseca del linguaggio (solo **25** ***keywords***, mentre **C99** ne ha **37** e **C++11** ne ha **84** e, *prevedo, C++2099 ne avrà 9999…*), ma è anche dovuta al meccanismo dei *"Package"*, che sono una specie di via di mezzo tra gli *"#include"* del **C**, la **STL** del **C++** e le librerie di **C** e **C++**: una volta identificato quale *Package* esegue la funzionalità che ci serve, lo si importa (l'istruzione *"import"* all'inizio del sorgente) e lo si usa. Tutto qua! Un buon esempio è l'uso del ***Package fmt*** (formatted I/O) nel classicissimo programma ***"hello world"*** in **Go**:

```
package main import "fmt" // importa il Package fmt (formatted I/O) func main() { fmt.Println("hello, world")
}
```

la successiva operazione di *"build"* dell'eseguibile fa tutto in automatico (scordatevi delle operazioni di *link* da mettere nei *makefile* di **C** e **C++**): basta importare, usare e *"buildare"*. **Go** è un gran linguaggio, è veramente ad alto livello ma mantiene allo stesso tempo dettagli da *"basso livello"* (scusate il gioco di parole) che ti permettono un range di utilizzazione enorme.

E, prima dei saluti di rito, vi lascio con una nuova citazione del mitico **Rob Pike** (l'altra la trovate **qui**) sulle idee alla base del **Go**:

"Notice that Robert [Griesmer] said C was the starting point, not C++. I'm not
certain but I believe he meant C proper, especially because Ken [Thompson] was there. But it's also true that, in the end, we didn't really start from C. We built from scratch, borrowing only minor things like operators and brace brackets and a few common keywords. (And of course we also borrowed ideas from other languages we knew.) In any case, I see now that we reacted to C++ by going back down to basics, breaking it all down and starting over. We weren't trying to design a better C++, or even a better C. It was to be a better language overall for the kind of software we cared about." da "Less is exponentially more", Rob Pike, 2012

Ciao, e al prossimo post!

https://italiancoders.it/better-call-go-come-scrivere-server-e-client-tcp-in-go-e-c-pt-2/