The Fastsocket – come velocizzare le POSIX IPC in C (e C++) – pt.2

Enigmista: *Enigma numero due! Se sei giustizia, non mentire è tuo dovere. Qual'è il tuo prezzo per fingere di non vedere?*
Colson: *"Il prezzo"?* ***Batman:*** *Le mazzette. Le ha chiesto quanto ha preso per chiudere un occhio.*

E rieccoci qui, siamo di nuovo sul pezzo. Devo dirlo, non avevo previsto una seconda parte per **l'articolo sulle Fastpipe**; però la chiusura che avevo scritto, quella sulla (presunta) complessità si usare il meccanismo dei messaggi a lunghezza variabile su altre **POSIX IPC** (oltre alle **Named Pipe**) mi ha fatto sentire un po' in colpa, e quindi ho deciso di estendere un il discorso (*prima che qualcuno mi facesse notare che avevo esagerato sulle difficoltà, ah ah ah*). E quindi rispolveriamo il nostro film ispiratore della prima parte, il bel bel **The Batman**, del bravo **Matt Reeves**. La frase citata sopra è molto attuale (*e vabbè, più che attuale direi che è, ahimè, quasi una costante della storia…*) ma, per quanto riguarda l'articolo precedente, vi assicuro che nessuno mi ha pagato per dire che era complicata una cosa che non lo è: è stata solo una svista (*o uno scherzetto da Enigmista, fate voi, ah ah ah*).



…tu non hai idea di quanto sia veloce un Fastsocket…

Ok: per riprendere il discorso ho deciso di mostrare come si può applicare il meccanismo dei messaggi a lunghezza variabile sui **socket**: visto che si parla di **POSIX IPC** l'ho fatto con gli **IPC socket** (**UNIX domain socket**, per gli amici), anche se sarebbe stato più comodo farlo direttamente con i **Network socket** (**Internet domain socket**, per gli amici), tanto sono praticamente la stessa cosa (*si, non storcete il naso, è così*).

Vi ricordo quale era il problema dei *socket* che rendeva impossibile al primo colpo l'implementazione usata con le *pipe:* la frammentazione dei messaggi tipica dei protocolli usati (il TCP/IP per i *Network socket*, e un protocollo simile per gli *IPC socket* che usano i *Kernel socket* invece dei classici *socket* di rete). Questo problemino si può risolvere, in realtà, abbastanza semplicemente (*e spoiler: anche in modo ultra-semplice, ma questo alla fine dell'articolo. Aspettate senza trattenere il fiato, mi raccomando!*).

Bene, proseguiamo. L'obbiettivo è, quindi, fare un nuovo *benchmark* per confrontare le prestazioni degli **IPC socket** in modo *"classic"* con quelle del modo *"fast". V*i mostrerò,

direttamente, la versione *“fast”*, visto che la versione *“classic”* si differenzia solo nell’uso delle funzioni **send(2)** e **recv(2)** al posto, rispettivamente, delle nostre **fastWrite()** e **fastRead()**, più l’incapsulamento dei dati nel tipo *Message* (insomma, le differenze sono le stesse che ci sono tra *pipe* e *Fastpipe* **come visto nell’ultimo articolo**). Vai col codice!

```
// processes.c - main processo padre
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/wait.h>
#include "data.h" // funzione main()
int main(int argc, char* argv[])
{ // crea i processi figli pid_t pid1, pid2; (pid1 = fork())
&& (pid2 = fork()); // test pid processi if (pid1 == 0) { //
sono il figlio 1 printf("sono il figlio 1 (%d): eseguo il
nuovo processo\n", getpid()); char *pathname = "reader"; char
*newargv[] = { pathname, NULL }; execv(pathname, newargv);
exit(EXIT_FAILURE); // exec non ritorna mai } else if (pid2 ==
0) { // sono il figlio 2 printf("sono il figlio 2 (%d): eseguo
il nuovo processo\n", getpid()); char *pathname = "writer";
char *newargv[] = { pathname, NULL }; execv(pathname,
newargv); exit(EXIT_FAILURE); // exec non ritorna mai } else
if (pid1 > 0 && pid2 > 0) { // sono il padre printf("sono il
padre (%d): attendo la terminazione dei figli\n", getpid());
int status; pid_t wpid; while ((wpid = wait(&status)) > 0)
printf("sono il padre (%d): figlio %d terminato (%d)\n",
getpid(), (int)wpid, status); // esco printf("%s: processi
terminati\n", argv[0]); exit(EXIT_SUCCESS); } else { // errore
nella fork(): esco printf("%s: fork error (%s)\n", argv[0],
strerror(errno)); exit(EXIT_FAILURE); }
}
```

```
// writer.c - main processo figlio
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
```

```
#include <unistd.h>
#include <sys/socket.h>
#include <sys/un.h>
#include "data.h"
#include "message.h" // funzione main()
int main(int argc, char *argv[])
{ // creo il socket in modo IPC e Stream printf("processo %d partito (writer)\n", getpid()); int sock; if ((sock = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) { // errore di creazione printf("%s: non posso creare il socket (%s)\n", argv[0], strerror(errno)); exit(EXIT_FAILURE); } // prepara la struttura sockaddr_un per il reader (è un server) remoto struct sockaddr_un reader; memset(&reader, 0, sizeof(reader)); reader.sun_family = AF_UNIX; strcpy(reader.sun_path, IPCS_PATH); // connessione al server remoto if (connect(sock, (struct sockaddr *)&reader, sizeof(reader)) < 0) { // errore connect fprintf(stderr, "%s: errore connect (%s)\n", argv[0], strerror(errno)); close(sock); return EXIT_FAILURE; } // loop di scrittura messaggi per il reader Message message; Data *my_data = &message.data; my_data->index = 0; do { // test index per forzare l'uscita if (my_data->index == N_MESSAGES) { // il processo chiude il socket ed esce per indice raggiunto printf("processo %d terminato (text=%s messaggi=%ld)\n", getpid(), my_data->text, my_data->index); close(sock); exit(EXIT_SUCCESS); } // compongo il messaggio e lo invio my_data->index++; snprintf(my_data->text, sizeof(my_data->text), "un-messaggio-di-test:%ld", my_data->index); } while (fastWrite(sock, &message) != -1); // il processo chiude il socket ed esce per altro motivo (errore) printf("processo %d terminato con errore (%s)\n", getpid(), strerror(errno)); close(sock); exit(EXIT_FAILURE);
}
```

```
// reader.c - main processo figlio
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <time.h>
#include <sys/time.h>
#include <sys/socket.h>
```

```
#include <sys/un.h>
#include "data.h"
#include "message.h" #define BACKLOG 10 // per listen() // funzione main()
int main(int argc, char *argv[])
{ // creo il socket in modo IPC e Stream printf("processo %d partito (reader)\n", getpid()); int sock; if ((sock = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) { // errore di creazione printf("%s: non posso creare il socket (%s)\n", argv[0], strerror(errno)); exit(EXIT_FAILURE); } // prepara la struttura sockaddr_un per questo reader (è un server) struct sockaddr_un reader; memset(&reader, 0, sizeof(reader)); reader.sun_family = AF_UNIX; strcpy(reader.sun_path, IPCS_PATH); // associa l'indirizzo del reader al socket (questo crea il file IPCS_PATH) unlink(IPCS_PATH); // rimuovo un eventuale file già creato if (bind(sock, (struct sockaddr *)&reader, sizeof(reader)) == -1) { // errore bind printf("%s: errore bind (%s)\n", argv[0], strerror(errno)); exit(EXIT_FAILURE); } // start ascolto con una coda di max BACKLOG connessioni if (listen(sock, BACKLOG) < 0) { // errore listen printf("%s: errore listen (%s)\n", argv[0], strerror(errno)); close(sock); exit(EXIT_FAILURE); } // accetta connessioni da un writer entrante socklen_t socksize = sizeof(struct sockaddr_un); struct sockaddr_un writer; // (remote) writer socket info (è un client) int writer_sock; if ((writer_sock = accept(sock, (struct sockaddr *)&writer, &socksize)) < 0) { // errore accept printf("%s: errore accept (%s)\n", argv[0], strerror(errno)); close(sock); exit(EXIT_FAILURE); } // chiude il socket non più in uso close(sock); // set clock e time per calcolare il tempo di CPU e il tempo di sistema clock_t t_start = clock(); struct timeval tv_start; gettimeofday(&tv_start, NULL); // loop di lettura messaggi dal writer int n_msg = 0; Message message; Data *my_data = &message.data; while (fastRead(writer_sock, &message) != -1) { // test numero messaggi per forzare l'uscita if (++n_msg == N_MESSAGES) { // get clock e time per calcolare il tempo di CPU e il tempo di sistema clock_t t_end = clock(); double t_passed = ((double)(t_end - t_start)) / CLOCKS_PER_SEC; struct timeval tv_end, tv_elapsed; gettimeofday(&tv_end, NULL); timersub(&tv_end, &tv_start, &tv_elapsed); // il processo chiude il socket, cancella il
```

```
file ed esce per numero raggiunto printf("reader: ultimo messaggio ricevuto: %s\n", my_data->text); printf("reader: processo %d terminato (messaggi=%d tempo CPU: %.3f - " "tempo totale:%ld.%ld)\n", getpid(), n_msg, t_passed, tv_elapsed.tv_sec, tv_elapsed.tv_usec / 1000); close(writer_sock); unlink(IPCS_PATH); exit(EXIT_SUCCESS); } } // il processo chiude il socket, cancella il file ed esce per altro motivo (errore) printf("processo %d terminato con errore (%s)\n", getpid(), strerror(errno)); close(writer_sock); unlink(IPCS_PATH); exit(EXIT_FAILURE);
}
```

```
// data.h - header per dati per mini-libreria IPC con IPC socket
#ifndef DATA_H
#define DATA_H // path del file per ipc socket
#define IPCS_PATH "myipcs" // numero di messaggi da scambiare per il benchmark
#define N_MESSAGES 2000000 // struttura Data per i messaggi
typedef struct { unsigned long index; // indice dei dati char text[16384]; // testo dei dati
} Data; #endif // DATA_H
```

```
#include <unistd.h>
#include <sys/socket.h>
#include "data.h"
#include "message.h" // prototipi locali
ssize_t myRecv(int sockfd, void *buf, size_t len, int flags);
// fastwrite - scrittura con size
ssize_t fastWrite( int fd, // il socket descriptor per la send() della libc Message *buf) // il buffer che contiene i byte da trasmettere
{ // set size reale (somma di tutte le dimensioni dei membri del // tipo Message (eccetto il membro size)) buf->size = sizeof(buf->data.index) + strlen(buf->data.text); // invio il messaggio completo: size + real-size (size + somma di // tutte le dimensioni dei membri del tipo Message (eccetto il membro size)) return send(fd, buf, SIZEOFP(buf), 0);
} // fastread - lettura con size
ssize_t fastRead( int fd, // il socket descriptor per la recv() della libc Message *buf) // il buffer su cui scrivere i byte ricevuti
```

```
{ // legge il size da usare nella successiva read ssize_t size_rcvd; if ((size_rcvd = myRecv(fd, &buf->size, sizeof(size_t), 0)) > 0) { // return la read successiva return myRecv(fd, &buf->data, buf->size, 0); } // ritorna nessun byte letto o errore return size_rcvd;
} // myRecv - una recv() speciale per pacchetti frammentati
ssize_t myRecv( int sockfd, // il socket descriptor per la recv() della libc void *buf, // il buffer su cui scivere i byte ricevuti size_t len, // la quantità di byte da ricevere int flags) // i flag per la recv() della libc
{ // loop per ricevere completamente un messaggio (forse) spezzettato int bytes_recvd = 0; // byte totali ricevuti int bytes_pending = len; // byte totali mancanti char* address = buf; while (bytes_pending > 0) { // ricevo un buffer (normalmente il primo e unico buffer che contiene // il messaggio intero) address += bytes_recvd; int tmp_recvd; // byte ricevuti if ((tmp_recvd = recv(sockfd, address, bytes_pending, flags)) > 0) { // aggiorno i contatori bytes_recvd += tmp_recvd; bytes_pending -= tmp_recvd; } } return bytes_recvd;
}
```

```
// message.h - header per read/write per mini-libreria IPC con IPC socket
#ifndef MESSAGE_H
#define MESSAGE_H #include "data.h" // struttura Data per i messaggi
typedef struct { size_t size; // size reale (somma di tutte le dimensioni dei membri del // tipo Message (eccetto il membro size)) Data data; // campo dati del messaggio
} Message; // size reale del messaggio
#define SIZEOFP(X) (sizeof(size_t) + sizeof(X->data.index) + strlen(X->data.text)) // prototipi globali
ssize_t fastRead(int fd, Message *buf);
ssize_t fastWrite(int fd, Message *buf); #endif // MESSAGE_H
```

Ok, il nuovo codice è **ampiamente commentato** (come sempre), ma comunque è il caso di aggiungere qualche dettaglio. È formato da sei file, quindi due in più di quello di riferimento, per cui la struttura è questa:

1. Il *main* di un processo padre: **processes.c**. Crea ed

esegue due processi figli con *fork + exec*. I due processi figli si chiameranno *writer* e *reader.*
2. Il *main* del processo *writer:* **writer.c**.
3. Il *main* del processo *reader:* **reader.c**.
4. Un *header file* per *reader* e *writer:* **data.h**.
5. Un nuovo sorgente che serve a gestire la trasmissione di messaggi con lunghezza variabile: **message.c**.
6. Un nuovo *header file* per il file message.c: **message.h**.

A questo punto vi risparmio ulteriori descrizioni che sarebbero identiche a quelle dello scorso articolo (*e i più attenti avranno notato che qui sopra ho fatto un copia-e-incolla di intere frasi, ah ah ah*).

È il caso, invece, di soffermarsi sull'unica differenza reale rispetto alla implementazione delle *Fastpipe:* nel file *message.c* la funzione **fastRead()** invece di chiamare internamente la **recv(2)** della *libc* chiama una nuova funzione locale, la **myRecv()**, delegando a quest'ultima le chiamate alla **recv(2)**. E cosa fa di speciale questa funzione? Semplice, visto che il problema da risolvere è lo *"spezzettamento"* (o frammentazione, come preferite) dei messaggi in arrivo, la **myRecv()** si occupa di ricevere tutti i frammenti di un messaggio e di restituire al chiamante il messaggio intero… problema risolto!

E come lavora la **myRecv()**? Direi che il codice è sufficientemente compatto e lineare da rendere l'idea a prima vista: viene fatto un *loop* di **recv(2)** e, di volta in volta vengono aggiornati dei contatori di *byte* ricevuti e di *byte* mancanti alla ricezione completa del messaggio (evidentemente la ricezione è completa quando si ricevono *<len> byte).* Nella migliore delle ipotesi il *loop* interno non farà nulla: alla prima ricezione si ottiene già il messaggio intero e lo si ritorna al chiamante, però in alcuni casi (magari frequenti) il messaggio verrà costruito chiamando più volta la **recv(2)** nel *loop.* Semplice, no?

E i risultati del test **IPC socket** *vs* **IPC Fastsocket** quali sono? Anche questa volta (come per le *pipe)* erano abbastanza scontati, ma è, comunque, il caso di mostrarli:

```
aldo@Linux $ cd ipcsocket/
aldo@Linux $ ./processes
sono il padre (17317): attendo la terminazione dei figli
sono il figlio 1 (17318): eseguo il nuovo processo
sono il figlio 2 (17319): eseguo il nuovo processo
processo 17318 partito (reader)
processo 17319 partito (writer)
processo 17319 terminato (text=un-messaggio-di-test:2000000 messaggi=2000000)
reader: ultimo messaggio ricevuto: un-messaggio-di-test:2000000
reader: processo 17318 terminato (messaggi=2000000 tempo CPU: 4.803 - tempo totale:4.823)
sono il padre (17317): figlio 17318 terminato (0)
sono il padre (17317): figlio 17319 terminato (0)
./processes: processi terminati aldo@Linux $ cd ../fastipcsocket/
aldo@Linux $ ./processes
sono il figlio 1 (17325): eseguo il nuovo processo
sono il padre (17324): attendo la terminazione dei figli
sono il figlio 2 (17326): eseguo il nuovo processo
processo 17325 partito (reader)
processo 17326 partito (writer)
processo 17326 terminato (text=un-messaggio-di-test:2000000 messaggi=2000000)
reader: ultimo messaggio ricevuto: un-messaggio-di-test:2000000
reader: processo 17325 terminato (messaggi=2000000 tempo CPU: 3.456 - tempo totale:3.456)
sono il padre (17324): figlio 17326 terminato (0)
sono il padre (17324): figlio 17325 terminato (0)
./processes: processi terminati
```

Il miglioramento nell'invio di 2000000 messaggi è buono, 3.456 secondi invece di 4.823 secondi, ed era, come detto sopra, scontato, visto che invece di trattare pacchetti (a lunghezza fissa) di 16 KB trattiamo pacchetti di qualche decina di byte;

magari ci si poteva aspettare qualcosa di più, ma è evidente che in un *benchmark* di questo tipo anche solo l'avvio di una operazione di read o write ha il suo peso, e contribuisce al tempo totale a prescindere dalla quantità di dati trattati (*oops… un altro copia-e-incolla, ah ah ah*).

E aggiungo, solo come curiosità: ho fatto la stessa operazione anche coi **Network socket** (vi risparmio il codice, non voglio dilungarmi troppo) e il miglioramento *"fast" vs "classic"* nell'invio di 2000000 messaggi è superiore a quello ottenuto con gli **IPC Socket**: 3.114 secondi invece di 6.899 secondi.

Siamo arrivati alla fine dell'articolo. Manca qualcosa? Ah, si, dimenticavo, avevo *spoilerato* un trucco finale! Ecco, dovete sapere che la **fastRead()** in realtà si può scrivere in maniera decisamente più semplice, rinunciando addirittura all'uso della **myRead()**, vediamo come:

```
// fastread - lettura con size
ssize_t fastRead( int fd, // il socket descriptor per la recv() della libc Message *buf) // il buffer su cui scrivere i byte ricevuti
{ // legge il size da usare nella successiva read ssize_t size_rcvd; if ((size_rcvd = recv(fd, &buf->size, sizeof(size_t), MSG_WAITALL)) > 0) { // return la read successiva return recv(fd, &buf->data, buf->size, MSG_WAITALL); } // ritorna nessun byte letto o errore return size_rcvd;
}
```

Come potete ben vedere, questa nuova versione della *fastRead()* è ultra semplice, ed è quasi identica a quella usata con la **Fastpipe**. Sfrutta il flag **MSG_WAITALL** della **recv(2)** (da passare nell'apposito campo <*flags*> della funzione; con questo flag attivo la ricezione si blocca fino a quando non sono arrivati tutti i blocchi che compongono il messaggio: fa esattamente il lavoro della **myRead()** vista sopra, però lo fa già a livello interno di libreria, quindi è sicuramente il metodo preferente. Comunque ho implementato la **myRead()** non

per perdere tempo, ma per mostrare come funzionano internamente i misteriosi *flag* delle funzioni di libreria, per cui scriverla è stato un utile esercizio (*ma nel Software di produzione usate **MSG_WAITALL**, mi raccomando!*).

E questo ultimo trucchetto ci porta indirettamente a un altro quesito: e se volessimo implementare **Fastsocket** usando **Datagram** (ossia aprendo il socket in modo **SOCK_DGRAM** invece che **SOCK_STREAM**)? Ecco, in questo caso, come per le **POSIX Message Queue** citate nell'ultimo articolo, l'esercizio non è possibile: per le MQ il problema era il meccanismo di base che è a lunghezza fissa, mentre per i *Datagram* (o modo **UDP**, per i **Network socket**) il meccanismo di lettura in due passi (prima la lunghezza del *buffer* e dopo il *buffer* stesso) non può funzionare, perché con i *Datagram* non viene garantito né l'arrivo né l'ordine di arrivo dei messaggi (*quindi funziona solo se siamo molto ma molto fortunati, ah ah ah*). Ah, guarda caso, come dice il manuale della **recv(2)** il flag **MSG_WAITALL** non è disponibile per i *Datagram socket* (*oh, che sorpresa!*).

E per oggi può bastare, abbiamo dimostrato che il metodo "*fast*" si può applicare anche agli **IPC socket** e ai **Network socket** (ma non coi *Datagram*, occhio!). Con questa seconda parte dichiaro concluso, almeno momentaneamente, l'argomento **Fast IPC**. Cosa ci riserverà il futuro? Boh, non lo so ancora, ma vi garantisco che sarà interessante!

Ciao, e al prossimo post!

https://italiancoders.it/the-fastsocket-come-velocizzare-le-posix-ipc-in-c-e-c-pt-2/