06-07.md

1. Lezione del 07/06 – Networking

Un contesto in cui il multithreading torna utile è quello in cui bisogni utilizzare la connessione di rete. Specifichiamo subito che la comunicazione tramite socker è spesso associata alla comunicazione su rete esterna, ma non è detto che le cose stiano così ed anzi sono molto utilizzate anche per la comunicazione fra processi.

I socket nascono infatti per far comincare più processi, nello specifico l'intento era mettere in comunicazione processi in esecuzione su due macchine differenti, ma la loro struttura permette di far comunicare processi in esecuzione sulla stessa macchina.

I socket hanno un vantaggio comunicativo rispetto alle pipe, quello di essere bidirezionali. Essi possono comunicare su 3 domini, in realtà riconducibili essenzialmente a due.

Il primo è il dominio applicativo, gestito tramite IP nellle due versioni disponibili, vale a dire IPv4 e IPv6. Quindi in questo modo il discriminante su chi riceve il messaggio sono gli indirizzi di rete.

Ovviamente noi non vogliamo comunicare con un computer in modo generico ma con un processo specifico, e qui ci viene in aiuto l'infrastruttura internet esistente in quanto i protocolli di livello trasporto offrono i numeri di porta per individuare un processo specifico sulla macchina. Questa operazione è svolta dal SO, quindi nel caso tipico dei web-server la porta comune è la porta 80, quindi le richieste in arrivo sulla porta 80 verranno reindirizzate al processo che si occupa di gestire la richiesta.

I socket permettono di definire dei domini applicativi che accompagnano la parte di indirizzamento. Tali domini applicativi sono anche chiamate famiglie di socket.

La prima, usata per la comunicazione sullo stesso computer, sono quelle dette `AF_UNIX` oopure `AF_LOCAL`, dove l'indirizzo puo' anche essere sostitutiot da un percorso per puntare un file nel file system. In quel caso faccio riferimento a un file che rappresenta il socket. Questo è come Docker gestisce i suoi container.

Nelle comunicazioni tra socket possiamo distinguere dei ruoli. Un server ad esempio apre un socket e si mette in ascolto su di esso, ma fintanto che non riceve dei messaggi esso non farà nulla, ma attenderà di ricevere una richiesta che, seppur non è scontato sia così, molto probabilmente arriverà da un client.

Gli aspetti che riguardano la sicurezza della comunicazione, non inteso in termini di crittografia ma come garanzia della consegna dei messaggi, dipende da noi. Sta quindi a noi implementare i vari meccanismi di acknolewdgement per informare che il messaggio è stato ricevuto. **Commento Personale:** mi sa che stava parlando quando usaimo i socket basati su Datagram, ma Rust ha anche quelli basati su TCP.

Sta quindi a noi implementare i vari meccanismi di acknolewdgement per informare che il messaggio è stato ricevuto. **Commento Personale:** mi sa che stava parlando quando usaimo i socket basati su Datagram, ma Rust ha anche quelli basati su TCP.

In Linux l'apertura del socket avviene tramite la funzione `socket` che ritorna un intero che serve per il bind. La funzione `bind` richiede come primo parametro il `socketfd`, il parametro ritornato dalla funzione `socket` che è effettivamete il suo File Descriptor.

La funzione `connect` non supporta eventuali cancellazioni tipiche invece delle varie librerire dei linguaggi ad alto livello. Anche in questo caso bisogna notificare tra i parametri il socket su cui ci si vuole connettere.

Al momento dell'invio dei dati, la `send` è identica alle varie open, write, etc di file binari in Linux. Questo perché la funzione accetta un `void*` per indicare un puntatore generico a dato. Anche in questo caso ho la possibilità di sapere se il tutto è stato trasferito o meno tramite il valore di ritorno, tipica delle funzioni di scrittura binarie.

Su questo aspetto si costruisce il protocollo. Se il mio messaggio è di 10 byte, e trovo che solo 8 sono state trasferite, sappiamo che c'è stato un errore.

Naturalmente la stessa identica funzionalità la si ottiene in fase di ricezione. Qui come parametri passiamo nuovamente il socket e trovandoci in un contesto di programmazione grezza viene passato un puntatore ad un buffer che non si auto ridimensiona. Quando ho costruito un protocollo orientato al messaggio, il parametro `n` che indicia la dimensione del buffer è esattamente la lunghezza del messaggio.

Posso specificare sia l'invio che la ricezione in modo puntuali. Nelle architetture di cui abbiamo parlato sono presenti un client ed un server. Il server non ha un indirizzo di invio fissato, perché si apre in ricezione. Il che significa che io, che devo gestire le API, devo dare la possibilità di inviare l'informazione specificando l'indirizzo. Questo perché se sono il server, io ricevo tanti messaggi potenzialmente da indirizzi diversi, e devo far si che la comunicazione di ritorno finisca nel posto giusto.

A ribadire quanto detto finora, poiché non vogliamo perderci niente per strada, potremmo incappare nel classico errore del file mezzo pieno. Ovvero ho dei dati da inviare che non vengono inviati perché il socket viene liberato prima e perdo un pezzo di dato. Forzando la chiusura è come se garantissi il *flush* del socket. Banalmente allora ritroveremmo che se io ho un paradigma RAII messo in pratica, allora quel che io faccio è forzare la chiusura del socket nel distruttore e la sua apertura nella costruzione.

Rust ci consente di gestire i socket che per lui sono solitamente una struttura TCP, stream o listener dipende dal lato dove ci troviamo, e che l'approccio alla programmazione è un pizzico più semplice di quanto visto finora, perchè con gli enumeratori sono in grado di gestire facilmente gli IP.

Il listener fondmantalmente modella il server, ovvero il binding e l'accettazione della connessione in ingresso, e che si sia in grado di recupare la parte di ingresso. Il vantaggio di usare Rust è che usando una `Result` ottengo in un unico blocco sia l'errore che il dato, stando attenti a notare che avendo l'ascolto la durata del processo, l'iteratore non si esaurisce mai.

```
let listener = TcpListener::bind("127.0.0.1:12345")?;

for stream in listener.incoming() {
    match stream {
        Ok(stream) => {
            println!("Connection established!");
            handle_connection(stream);
        }
        Err(e) => {
            println!("Connection failed: {}", e);
        }
    }
}
```

Qui quello che facciamo è metterci in ascolto, che vuol dire usare un indirizzo locale e gestire le richieste una per volta tramite un iteratore.

Quando leggiamo facciamo una cosa particolare. Cercando di estrarre i dati dallo stream, questi dati sono letti come fossero sequenze di byte. Quando eseguiamo una readline è come se stessimo leggendo un file grezzo, che permette di capire se la connessione ha effettivamente ha un dato.

Se non ha un dato, quindi la stringa ha size 0, la comunicazione si è interrotta, e posso quindi attendere il prossimo stream.

La connessione del client segue le stesse regole del server, dove troviamo una `wait_timeout`. Esattamente come un problema di sincronizzazione, potrei ritrovarmi in starvation. I linguaggi di alto livello offrono queste funzioni con timeout perché queste chiamate sono tipiamente bloccanti e si decide che se la connessione non viene stabilita entro un certo lasso di tempo, essa è da considerarsi fallita.