Scambio di messaggi: socket

Il meccanismo più potente e flessibile per lo scambio delle informazioni attualmente in uso è quello fornito da dai protocolli TDP e UDP, che sono il fondamento di Internet così come la conosciamo oggi. Il principale vantaggio è quello di poter trasferire dati tra processi in computer diversi, mentre in tutti gli altri casi il sistema è limitato a un solo computer.

Lo scambio di dati utilizzando il protocollo TCP è un po' complicato e fa riferimento a concetti che verranno pienamente affrontati il prossimo anno, come client-server, servizi con e senza connessione e altro ancora. Per ora quindi ci concentreremi sull'uso del protocollo UDP, più simile ai sistemi che abbiamo già visto e dalla realizzazione più semplice.

Un **socket** (PRESA DI CORRENTE) è un **punto di comunicazione tra due processi**, ed è formato dall'unione di un indirizzo e da una porta.

L'indirizzo identifica a quale host deve essere consegnato il messaggio, mentre porta identifica lo specifico processo che deve ricevere il messaggio.

Perché una comunicazione sia possibile, occorrono **due** socket, sorgente e destinazione. La creazione quindi di un sistema di comunicazione è un processo formato da più passi:

1. **Creare il socket**
   Si effettua con la funzione socket(). Nel nostro caso la chiamata tipica è
   `s = socket(AF_INET, SOCK_DGRAM, 0)`
2. **Identificare il socket** (o “dargli un nome”)
   Il socket appena creato è sostanzialmente vuoto, più simile a un buco nel muro che a una presa di corrente. Come sappiamo, dobbiamo fornigli un indirizzo IP e una porta, e inserirlo in un'apposita struttura chiamata `struct sockaddr_in indirizzo;`
   In particolare, useremo il nostro indirizzo ip e chiederemo al sistema operativo di scegliere una qualsiasi porta libera, a meno che non abbiamo preferenze per una porta particolare. La chiamata assumerà un aspetto simile a

```
bind(s, (struct sockaddr *)&indirizzo,
sizeof(indirizzo);
```

3. Sul trasmittente (client), **spedire un messaggio**
   E' il punto più interessante, ma anche il più complicato, che si realizza con la funzione sendto(). La funzione è dotata di parecchi parametri: il socket appena creato, indirizzo e porta di destinazione, un buffer contenente il messaggio da trasferire e qualche flag di controllo. Un aspetto tipico potrebbe essere
   ```
   sendto(s, messaggio, strlen(messaggio), 0,
   (struct sockaddr *)&dati_server,
   sizeof(dati_server))
   ```

4. Sul ricevente (server), **ricevere un messaggio**
   Come nel caso di una ricetrasmittente radio, il ricevente deve essere in ascolto sul "canale" giusto per poter ricevere un messaggio e questo avverrà usando bind() settato su una specifica porta. Dopo di che, userà la funzione recvfrom(), che ha una struttura di parametri analoga a sendto()
   ```
   recvfrom(s, buffer, sizeof(buffer), 0, (struct
   sockaddr *)&dati_client, &dimensione_ind);
   ```
   La funzione **non è bloccante**: occorre controllare che resistuisca una valore diverso da zero per sapere se ha effettivamente ricevuto un pacchetto.

5. **Proseguire nel dialogo**
   Dato che il server ha ricevuto indirizzo IP e porta utilizzata dal client nella struttura dati_cliente, può spedire messaggi in risposta al client utilizzando sendto. Il "dialogo" può proseguire nei due sensi senza limitazioni.

6. **Chiudere il socket**
   Anche se non strettamente necessarrio, dato che non esiste una vera e propria connessione tra server e client, è opportuno chiudere il socket per liberare le strutture di gestione nel sistema operativo.
   ```
   close(fd);
   ```

FARE IMMAGINE ESPLICATIVA

Qui trovate un esempio completo e funzionante di trasmissione UDP, basato su quanto è stato spiegato. Occorre dire che il linguaggio C in questo caso risulta estremamente verboso e ripetitivo, ma quasi tutti i linguaggi di programmazione moderni sono dotati di librerie che rendono la programmazione dei socket estremamete compatta e comprensibile. Per esempio, in Java, potete guardare un server e un client oppure, se volete una versione multithread, qui. In C#, potete guardare qui.

Esercizi

Quiz

Multiple choice

Telcomando/telecomandato

Shared memory

Programmazione multithread

Nel caso appena visto, due processi possono cooperare, ma con una certa difficoltà: i metodi di comunicazione risultano difficili da utilizzare, con limitato scambio di informazione, problemi di sincronizzazione, spesso tutti contemporaneamente. Molti di queste limitazioni possono essere evitate quando si utilizzano i thread.

Ricordiamo dall'anno scorso, che i thread (o processi leggeri) sono linee di esecuzione all'interno dello stesso processo: come tali, i thread condividono interamente la memoria, riducendo la necessità dei casi di diavolerie come pipe, e socket dato che una memoria è condivisa, una variabile globale sarà accessibile da tutti! In più, dato che i processori sono in grado di gestire più thread anche sullo stesso core, il cambio di contesto più avvenire in modo molto rapido.

Anche in questo caso però, non è sempre rose e viole: il fatto di poter accedere e modificare le variabili in modo non prevedibile introduce una serie di problemi,