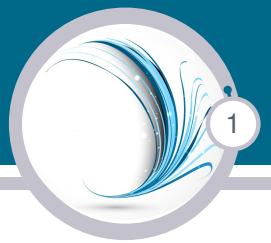
Facoltà di Ingegneria Informatica
Università degli Studi di Roma Tor Vergata

Progetto del corso Ingegneria di Internet e del Web a.a. 2017-2018

Andrea Graziani - matricola 0189326

26 febbraio 2018

Architettura dell'applicazione
Un'applicazione client-server con architettura a 2 livelli



L'applicazione di rete sviluppata è di tipo **client-server con architettura a 2 livelli** in cui è prevista l'esistenza di:

- Un **client** ossia l'entità che, incorporando tutta la logica applicativa che rende operativa l'applicazione (*business logic*), accede ai servizi o alle risorse di un'altra componente, detta server.
- Un **server** ovvero l'entità incaricata di soddisfare le richieste di servizio provenienti dai client.

L'impraticabilità di un server iterativo

Sebbene di semplice implementazione, un server di tipo **iterativo** non è adatto quando l'elaborazione delle richieste dei client richiede "*molto tempo*" perché ciò comporta **tempi di attesa molto lunghi**.

I vantaggi di un server concorrente

In un server di tipo **concorrente** il compito di gestire e soddisfare le richieste ricevute dai client viene affidata ad un processo (o *thread*) **distinto** da quello principale. Ciò consente di:

- Gestire più richieste contemporaneamente.
- Ridurre i tempi di attesa.
- Permettere al processo server di poter continuare a ricevere ed elaborare nuove richieste.

Architettura dell'applicazione
Il processo server

Se si analizza il codice della funzione `start_server()` è facile intuire che il lavoro svolto dal **processo server** si articola in:

Inizializzazione della socket di ascolto Necessaria per poter ricevere messaggi di richiesta dai client;

Caricamento metadati Questa fase prevede il caricamento di una serie di informazioni utili a soddisfare richieste di tipo `list`;

Fase di ascolto Il processo server rimane in attesa dell'arrivo di un messaggio di richiesta sulla socket di ascolto precedentemente creata.

Creazione del thread Alla ricezione di un messaggio di richiesta viene creato un nuovo thread a cui affidare il compito di gestire e soddisfare la richiesta ricevuta.
Successivamente il processo server ritorna in fase di ascolto.

Architettura dell'applicazione
Il processo server - Inizializzazione della socket di ascolto

Da un punto di vista implementativo la **fase di inizializzazione della socket di ascolto** avviene attraverso l'esecuzione della funzione `__server_initialization` che prevede:

1. L'esecuzione della chiamata di sistema `socket(int domain, int type, int protocol)` dove:

   1.1 Il parametro `domain` indica il **dominio di comunicazione** cioè specifica la famiglia dei protocolli che verranno usati nella comunicazione.
   Il dominio scelto è `AF_INET` che designa il **protocollo IPv4**.

   1.2 Il parametro `type` specifica il *tipo di socket* ed è utilizzato per specificare **la semantica della comunicazione**.
   Il tipo scelto è stato `SOCK_DGRAM` che una forma di comunicazione **basata su datagrammi**, **senza connessione**, **inaffidabile** e che può inviare **messaggi di lunghezza prefissata**.

   1.3 Il parametro **protocol** specifica il particolare protocollo da utilizzare con la socket specificata.

Architettura dell'applicazione

Il processo server - Inizializzazione della socket di ascolto

Una volta creata una socket mediante la chiamata di sistema `socket` occorre assegnarle un indirizzo mediante la chiamata di sistema `bind` che permette di assegnare alla socket l'indirizzo specificato da una struttura dati di tipo `sockaddr`.

```
1  // Set address family that is used to designate the type of addresses that socket
2  // can communicate with: in this case, Internet Protocol v4 addresses */
3  servaddr.sin_family = AF_INET;
4  /* Server accepts request on any network interface */
5  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
6
7  /* Specify port 'SERVER_PORT' converting them from host byte order to network byte order. */
8  if (main_process) {
9          servaddr.sin_port = htons(SERVER_COMMAND_PORT);
10
11         // allow reuse of local address if there is not an active listening
12         // socket bound to the address
13         int reuse = 1;
14         if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(int)) == -1)
15                 exit_failure("setsockopt");
16
17 } else
18         servaddr.sin_port = 0;
19
20 /* Bind the socket to server address */
21 if (bind(sockfd, (struct sockaddr*) &servaddr, sizeof(servaddr)) == -1)
22         exit_failure("bind");
```

Messaggi di richiesta

Un client può inoltrare richieste di servizio al server attraverso l'invio di un apposito **messaggio di richiesta** sfruttando la socket di ascolto.

Un messaggio di richiesta è modellato dalla struttura dati di tipo `RequestPacket` la quale specifica il tipo, i parametri e le modalità di esecuzione di una data richiesta.

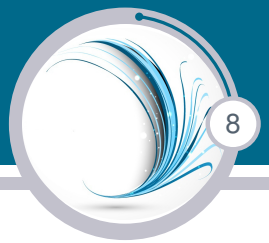
Listing 1: Implementazione della struttura `RequestPacket`

```
1  typedef struct request_client_packet {
2
3          Settings client_setting;
4          char request_type;
5          char request_type_argument[NAME_MAX];
6
7  } RequestPacket;
```

Architettura dell'applicazione
Il processo client

Analizzando la funzione `client_start` è facile intuire che l'invio di un messaggio di richiesta e dunque l'inizio di una trasmissione prevede:

- L'inizializzazione di una struttura dati denominata `RequestPacket`.
- L'invio della suddetta struttura al server attraverso l'ausilio di una struttura di tipo `DataNetwork`.
- Attesa della risposta dal server; in caso di mancata risposta occorre rinviare un nuovo messaggio di richiesta.

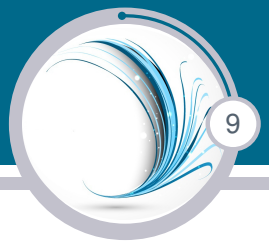


La struttura dati `DataNetwork`

Una struttura dati di tipo `DataNetwork` rappresenta tecnicamente un contenitore per tutte le informazioni necessarie per comunicare con un host tra cui l'**indirizzo IP** e il **numero di porta**.

Listing 2: Implementazione della struttura `DataNetwork`

```
1  typedef struct data_network {
2
3          /* Socket file descriptor */
4          int socket_fd;
5          /* Internet socket address */
6          struct sockaddr_in address;
7          /* Internet socket address length */
8          socklen_t address_len;
9
10 } DataNetwork;
```



Problema

Il server UDP deve poter scambiare più di un pacchetto con il client per poter soddisfarne le richieste. Il problema è che **l'unico numero di porta noto** al client per poter comunicare con il server è, sfortunatamente, **la stessa porta utilizzata da quest'ultimo per ricevere nuove richieste**.

Soluzione

I thread incaricati di soddisfare le richieste hanno il compito di **creare un'apposita socket di supporto** eseguendo il **binding su una porta effimera** e utilizzando quel socket per inviare e ricevere i successivi messaggi con un client.

Il client deve guardare **il numero di porta della prima risposta del server** e dunque inviare i pacchetti successivi a quella porta.

Da un punto di vista implementativo ciò è reso possibile dalla funzione `__receive_datagram` la quale provvede ad **aggiornare il contenuto della struttura** `DataNetwork` dopo la ricezione della prima risposta. Tutto ciò permette al client di **ignorare tutti quei pacchetti non provenienti dal processo server con cui interagisce**.

Architettura dell'applicazione
Schema di funzionamento

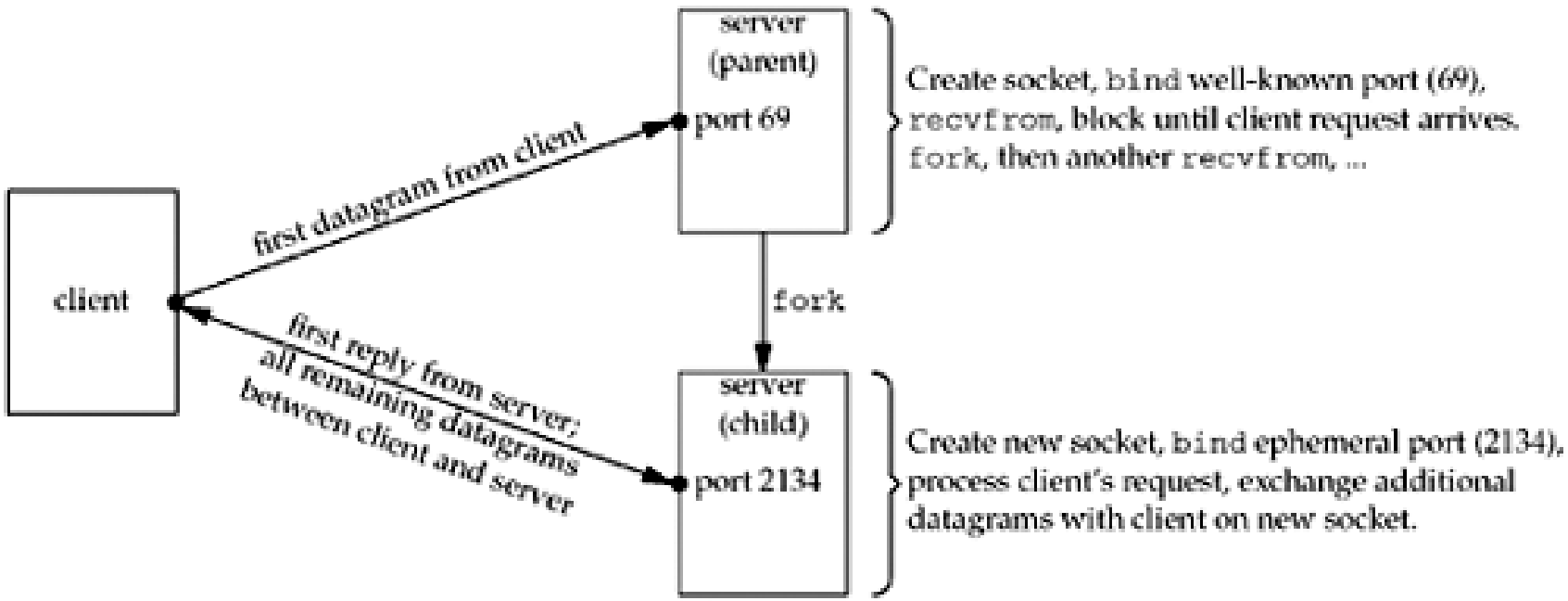


Figura: Schema del funzionamento di un server concorrente.

Servizio di trasmissione affidabile

Obiettivo

Per i scopi della nostra applicazione abbiamo bisogno di realizzare un **protocollo applicativo** che fornisca un **servizio di trasmissione dati affidabile** basandosi, tuttavia, su un protocollo di trasporto che offra, invece, un servizio di comunicazione inaffidabile (detto anche *best effort*).

Garantire un servizio di trasmissione dati affidabile richiede di:

- Gestire in modo adeguato la presenza di errori sui bit dei pacchetti trasmessi.
- Riuscire a rilevare e poi a gestire lo smarrimento di uno o più pacchetti.

Servizio di trasmissione affidabile

Come in moltissimi protocolli di comunicazione, il mittente e il destinatario devono essere in grado di potersi scambiare appositi **messaggi di notifica** per gestire situazioni di errore o smarrimento dei dati trasmessi.

Infatti questi messaggi di notifica risultano fondamentali per due importanti motivi:

- Garantire il corretto funzionamento del protocollo applicativo il quale, in accordo alle specifiche progettuali, deve simulare un protocollo di trasmissione a **ripetizione selettiva**. Infatti **l 'unico modo che ha il mittente per sapere se un pacchetto sia stato ricevuto correttamente o meno consiste nel *feedback* esplicito del destinatario.**
- Individuare un pacchetto smarrito in basa alla mancata o eccessivamente ritardata ricezione del messaggio di notifica proveniente dal destinatario.

I messaggi di controllo

Nella terminologia usata dal nostro protocollo i messaggi di notifica scambiati dagli host sono rappresentati dai cosiddetti **messaggi di controllo**.

Da un punto di vista implementativo i messaggi di controllo, modellati dalla struttura `ControlDatagram`, vengono utilizzati per:

1. Inviare al mittente un riscontro (o ACK, anche detto *acknowledgement*) per i pacchetti ricevuti in modo corretto compatibilmente con il protocollo di trasmissione SR.
2. Gestire le procedure di instaurazione e chiusura delle trasmissioni.

Listing 3: Implementazione della struttura `ControlDatagram`

```
1  typedef struct control_datagram {
2          char type;
3          char n;
4  } ControlDatagram;
```

Come rilevare un pacchetto smarrito?

Come abbiamo detto in precedenza se dopo la trasmissione di un certo pacchetto il mittente non ricevesse alcun riscontro da parte del destinatario dopo un certo *lasso di tempo*, il mittente considera il suddetto pacchetto come *smarrito* e procede alla *ritrasmissione* di quest'ultimo.

Quanto deve essere lungo questo *lasso di tempo*?

Ovviamente l'intervallo di timeout dovrebbe essere più grande del tempo di andata e ritorno del segnale sulla connessione per evitare timeout prematuri o troppo ritardati.
L'algoritmo usato per il calcolo della lunghezza dell'intervallo di timeout è analogo a quello usato nel protocollo di trasporto TPC.

Servizio di trasmissione affidabile
Le ritrasmissioni

Dovendo simulare un protocollo di comunicazione SR, dobbiamo associare ad ogni pacchetto da trasmettere un timer dedicato capace di segnalare al mittente l'avvenuta scadenza di un dato lasso di tempo.

Inoltre il mittente deve essere in grado di:

- Inizializzare un timer ogni volta che invia un pacchetto.
- Rispondere adeguatamente all'interrupt generato dal timer in caso di timeout procedendo quindi alla ritrasmissione del pacchetto smarrito e reinizializzare il timer.
- Disabilitare il timer quando il mittente riceve un riscontro positivo da parte del destinatario.

Servizio di trasmissione affidabile
La struttura `DatagramTimer`

L'associazione tra uno specifico pacchetto con un timer avviene mediante l'ausilio di un'apposita struttura dati denominata `DatagramTimer`.

Listing 4: Implementazione della struttura `DatagramTimer`

```
1   typedef struct datagram_timer {
2           void *Datagram_ptr;
3           DataNetwork *DataNetwork_ptr;
4
5           char *abort_trasmission_ptr;
6           char datagram_type;
7
8           timer_t timer;
9           char times_retrasmitted;
10          struct itimerspec timervals;
11          struct timespec sending_time;
12          struct timespec receiving_time;
13  } DatagramTimer;
```

Servizio di trasmissione affidabile
Inizializzazione della struttura `DatagramTimer`

L'associazione di uno pacchetto con un timer consiste nell'eseguire una procedura di inizializzazione di un oggetto `DatagramTimer` che prevede le seguenti fasi:

- Associazione del pacchetto;
- Configurazione handler ossia dichiarazione della procedura da seguire in caso di timeout;
- Configurazione della struttura `sigevent` durante la quale viene specificata le modalità attraverso cui l'applicazione debba essere avvisata dell'avvenuto timeout;
- Creazione del timer logico in accordo alle API POSIX;

Servizio di trasmissione affidabile

Inizializzazione della struttura `DatagramTimer`


```
1   void DatagramTimer_init(DatagramTimer *DatagramTimer_obj, void **Datagram_ptr, DataNetwork **
        DataNetwork_ptr, char *abort_trasmission_ptr,
2                   char datagram_type, void (*handler)(int, siginfo_t*, void*)) {
3
4           struct sigevent sev;
5           struct sigaction sa;
6
7           /* Configuration 'DatagramTimer' */
8           if (DataNetwork_ptr != NULL)
9                   DatagramTimer_obj->DataNetwork_ptr = *DataNetwork_ptr;
10          if (Datagram_ptr != NULL)
11                  DatagramTimer_obj->Datagram_ptr = *Datagram_ptr;
12
13          DatagramTimer_obj->abort_trasmission_ptr = abort_trasmission_ptr;
14          DatagramTimer_obj->datagram_type = datagram_type;
15
16          /* Establish handler for notification signal */
17          sa.sa_flags = SA_SIGINFO;
18          sa.sa_sigaction = handler;
19          sigemptyset(&sa.sa_mask);
20          sigaddset(&sa.sa_mask, SIGRTMAX);
21          if (sigaction(SIGRTMAX, &sa, NULL) == -1)
22                  exit_failure("sigaction");
23
24          /* Configuration structure to transport application-defined values with signals. */
25          sev.sigev_notify = SIGEV_THREAD_ID;
26          sev._sigev_un._tid = syscall(SYS_gettid);
27          sev.sigev_signo = SIGRTMAX;
28          sev.sigev_value.sival_ptr = DatagramTimer_obj;
29
30          /* Create new per-process timer using CLOCK_ID. */
31          if (timer_create(CLOCK_REALTIME, &sev, &DatagramTimer_obj->timer) == -1)
32                  exit_failure("timer_create");
33  }
```

Fasi della trasmissione dati

Per la trasmissione dei dati fra host il protocollo applicativo prevede l'esecuzione delle seguenti operazioni:

- Inizializzazione della trasmissione;
- Instaurazione della trasmissione;
- Trasmissione dati;
- Chiusura della trasmissione;

Gestione della trasmissione dati
Inizializzazione della trasmissione

Durante questa fase devono essere allocate e inizializzate tutte le strutture dati necessarie per eseguire la trasmissione tra cui:

- Un *buffer di ricezione* (lato client) e un *buffer di invio* (lato server) entrambi di dimensione N. Ricordiamo che la relazione ottimale tra la dimensione del buffer N e la dimensione dell'apertura della finestra di spedizione W è:

$$N = 2W$$

- Un buffer di dimensione N usato per ospitare oggetti di tipo `DatagramTimer` necessari per l'associazione tra timer e pacchetti.
- Un oggetto di tipo `TimeInfo`.
- Un oggetto di tipo `NetworkStatistics`.

Questa fase è necessaria per segnalare l'inizio di una trasmissione ed è utile per distinguere correttamente più trasmissioni consecutive eseguite tra una stessa coppia di host.

Questa fase, similmente a quanto avviene nella fase di connessione in TCP, prevede la trasmissione di tre opportuni messaggi di controllo: si può parlare dunque di **handshake a tre vie** (*three-way handshake*).

Gestione della trasmissione dati
Trasmissione dati

Il protocollo suddivide la trasmissione dei dati nelle seguenti operazioni:

- Lettura dal *file system* e incapsulamento dati;
- Invio pacchetto dati previa attivazione timer ad esso associato rispettando le regole del protocollo SR;
- Gestione dei riscontri ricevuti previa disattivazione dei timer associati a pacchetti inviati con successo e aggiornamento delle variabili della trasmissione compatibilmente con il protocollo SR;
- Eventuale ritrasmissione di pacchetti smarriti;

Gestione della trasmissione dati
Chiusura della trasmissione

La chiusura della trasmissione avviene solo quando sono entrambe verificate le seguenti condizioni:

- Il mittente non ha più dati da inviare al destinatario;
- Il mittente rileva che anche l'ultimo pacchetto inviato ha ricevuto un riscontro positivo;

Il protocollo prevede l'invio di appositi messaggi di controllo (`FIN`, `FIN_ACK`, `CLS`) per terminare correttamente la trasmissione.

Grazie per l’attenzione!