1 Lezione del 29-09-25

Continuiamo a vedere nel dettaglio l'ambito delo *sviluppo di applicazioni*.

1.1 Comunicazione fra processi

Abbiamo visto come, sebbene debbano girare su una vasta gamma di dispositivi ed interagire con svariate tecnologie di collegamento, queste possono essere riassunte nei paradigmi client-server e *peer-to-peer*. Inoltre, abbiamo visto come maggior parte della complessità dell'infrastruttura di rete sia astratta via nei moderni S/O dietro il meccanismo dei *socket*.

In particolare, vediamo i nostri applicativi come composti da **processi** in esecuzione su macchine fisiche, da cui **processi client** su macchine client e **processi server** su macchine server. Le modalità secondo la quale questi processi si scambieranno **messaggi** saranno rappresentate dalla cosiddetta **IPC** (*Inter Process Communication*).

Notiamo che non è necessario che la IPC sia su macchine diverse, due processi nella stessa macchina possono infatti comunicare fra di loro come se si trovassero su macchine diverse collegate in rete.

Quello che vanno ad implementare i **socket**, standard *de facto* apparso in origine nei sistemi operativi BSD, è appunto l'IPC fra più processi.

1.1.1 Modalità di indirizzamento

Per ricevere messaggi, i processi devono avere degli *identificatori*. Questi sono rappresentati da **indirizzi IP** su 32 bit. L'indirizzo IP della macchina su cui i processi girano non basta, in quanto chiaramente una macchina può avere più di un processo in esecuzione.

1.1.2 Protocolli livello application

Un protocollo di livello *application* dovrà a questo punto definire diverse specifiche sull'IPC:

- Il **tipo** di messaggi scambiati: questi possono solitamente essere *richieste* e *risposte*;
- **Sintassi** dei messaggi: quali campi presentano i messaggi e come i campi sono delineati nella struttura del messaggio;
- **Semantica** dei messaggi: il significato dell'informazione contenuta nei campi;
- **Regole**: su come e quando i processi possono ricevere o inviare messaggi.

Esistono diversi protocolli application, sia **aperti** (interoperabili per applicazioni di più sviluppatori, lo sono il protcollo HTTP, i protocolli di posta, ecc...) che **proprietari** (lo sono ad esempio i protocolli associati a prodotti software proprietari come Skype, ecc...). In particolare, i protocolli aperti possono essere sia **ufficiali** (supportati da agenzie come l'RFC), o **non ufficiali** (basati su documenti non ufficiali ma di pubblico accesso, diventati standard *de facto* dopo grande adozione, ad esempio BitTorrent).

1.1.3 Servizi ad applicazioni

Iniziamo a vedere quali tipi di servizi potrebbero servire ad un applicazione in rete.

- **Integrità dati**: alcune applicazioni (trasferimento dati, transazioni, ecc...) potrebbero richiedere un trasferimento sicuro al 100%. Altre (videogiochi, streaming, ecc...) potrebbero invece poter tollerare perdite paraziale dei dati in fase di trasmissione.
- **Throughput**: alcune applicazioni (multimedia, ecc...) richiedono una quantità minima di throughput per essere efficaci. Altre (le cosiddette applicazioni *elastiche*) ne prendono quanto non hanno a disposizione.
- **Tempo**: alcune applicazioni (ancora videogiochi, telefonia online, ecc...) potrebbero richiedere delay temporali molto contenuti per essere efficaci.
- **Sicurezza**: servizi più o meno critici richiedono livelli variabili di sicurezza (crittografia, ecc...).

Su questa base, potremmo classificare alcune applicazioni sulla base dei requisiti di trasporto che hanno:

Applicazione	Perdita dati	Throughput	Tempo	Sicurezza
Trasferimento file/download	Nessuna	Elastico	Indifferente	Dipende
E-mail	Nessuna	Elastico	Indifferente	Dipende
Web	Nessuna	Elastico	Indifferente	Dipende
Streaming	Tollerante	5 Kbps - 5 Mbps	Pochi secondi	Non importante
Videogiochi	Tollerante	5 Kbps - 5 Mbps	Pochi millisecondi	Dipende
Messaggistica istantanea	Nessuna	Elastico	Perlopiù indifferente	Critica

1.1.4 Protocolli livello transport

Potrebbe essere utile, per capire come realizzare le specifiche sopra descritte, studiare ad alto livello i due protocolli di trasporto principali:

- **TCP** (*Transmission Control Protocol*): assicura il trasporto *affidabile* di messaggi tra processi, controllo del *flusso* e delle *congestioni*, ma ha promesse più scarse nel campo della temporizzazione e del throughput (sebbene assicuri un *throughput minimo*). Orientato alla *connessione* fra processi client e server, è più sicuro dell'alternativa;
- **UDP** (*User Datagram Protocol*): meno sicuro e affidabile, non fornisce servizi di controllo flussi o congestioni, né throughput minimo. Questo lo rende adatto per soluzioni a basso overhead, dove le prestazioni sono più significative della correttezza dei dati.

Il controllo del flusso e delle congestioni si collega a quanto visto nella sezione 2.2:

- Il controllo del **flusso** si assicura che il mittente non potrà sopraffare il destinatario con una mole troppo grande di messaggi;
- Il controllo delle **congestioni** assicura che i ruoter nell'infrastruttura fra mittente e destinatario non vengano sovraccaricati, o se lo sono l'informazione venga deviata in modo da assicurare che i pacchetti arrivino in maniera affidabile.

Possiamo quindi assegnare ad ognuna delle applicazioni viste nella tabella di sezione 4.13 un protocollo adatto:

Applicazione	Protocollo application	Protocollo transport
Trasferimento file/download	FTP	TCP
E-mail	SMTP	TCP
Web	HTTP	TCP
Streaming	HTTP, DASH	TCP
Videogiochi	WOW, FPS o proprietario	TCP o UDP
Messaggistica istantanea	HTTP o proprietario	TCP o UDP (telefonia)

1.1.5 Sicurezza in TCP

I socket TCP e UDP di base non forniscono particolari funzionalità di sicurezza: i dati sono trasmessi senza crittografia, per cui dati sensibili sono visibili in chiaro.

Si può sfruttare il protocollo (in verità protocollo *middleware*, che sta fra livello transport e livello application) **TLS** (*Transport Layer Security*) per fornire connessioni TCP crittografate, con integrità dei dati assicurata e autenticazione del destinatario.

Possiamo sfruttare TLS in 2 modi principali:

- TSL implementato in applicazione, che a sua volta interagisce con socket TCP;
- API che fornisce socket TLS, che ricevono dati in chiaro e li inviano su Internet crittografati.

La manifestazione più comune del protocollo TLS si vede negli URL delle pagine web, che iniziano con `http://` quando si usa HTTP su TCP puro, e con `https://` quando si usa HTTP su TCP con TLS.

1.2 Web e HTTP

Veniamo quindi a dettagliare la più famosa applicazione sviluppata su Internet, cioè il Web. Come abbiamo visto il Web è supportato dal protocollo **HTTP** (*HyperText Transfer Protocol*).

Ricordiamo quindi che una **pagina** Web consiste di oggetti di diversi formati che possono essere allocati su più Web server. Le pagine in sé per se sono anch'esse oggetti, consistono di file **HTML** (*Hypertext Markup Language*), indirizzabili assieme come tutti gli altri oggetti che le compongono da un **URL** (*Uniform Resource Locator*), in forma:

```
<schema>//<nome-host>/<percorso_risorsa>
```

ad esempio, `https://www.bittorrent.org/index.html`.

1.2.1 Protocollo HTTP

Il protocollo HTTP è basato sul modello client-server (richiede un protocollo di livello transport che supporti connessioni client-server, quasi sempre TCP). In questo, il client è rappresentato dal *browser*, che compila richieste per ottenere pagine web, mentre il server è rappresentato dal *Web server*, che riceve le richieste dei browser e risponde inviando oggetti.

Nello specifico, una connessione HTTP si svolge come segue:

- Il client inizia la connessione TCP (lato applicazione, crea il socket) col server, alla porta 80;
- Il server accetta la connessione TCP del client;
- Messaggi HTTP vengono scambiati fra client (browser) e server sulla linea TCP;
- La connessione TCP viene chiusa.

Il protocollo HTTP è *privo di stato*, cioè non si mantiene nessuna informazione riguardo alle richieste passate del client.

Esistono 2 tipi di connessioni HTTP:

- HTTP **non persistente**: si apre la connessione TCP e si invia al più un oggetto sulla connessione prima di chiudere. In questo caso scaricare più oggetti richiede più connessioni TCP;
- HTTP **persistente**: si apre la connessione TCP e si inviano più oggetti sulla connessione prima di chiudere.

Definiamo il **RTT** (*Round-Trip Time*) come il tempo che un piccolo pacchetto impiega per viaggiare da client a server e ritorno.

- Nel caso dell'HTTP non persistente, richiediamo un RTT per iniziare la connessione TCP, un RTT per richiedere il file, più il tempo necessario a trasferire il file vero e proprio, per cui si impiega:

  $$T_{\text{non-pers}} = 2\text{RTT} + T_{\text{tras}}$$

  per ottenere ogni file.
- Nel caso dell'HTTP persistente, dovremmo comunque usare 2 RTT per iniziare la connessione e chiedere il primo file, ma ogni file successivo richiederà solamente il tempo RTT necessario a richiedere la risorsa, per cui risparmieremo tempo.

  Chiaramente in questo caso avremo il problema di dover capire *quando* chiudere la trasmissione: magari dopo n richieste, dopo un tempo T (*Timeout*), ecc...

1.2.2 Richieste HTTP

Abbiamo visto come i messaggi HTTP appartengono a 2 tipi, *richieste* e *risposte*. I messaggi sono in formato ASCII, quindi leggibile dall'uomo.

In particolare, l'header di una richiesta HTTP ha la seguente forma generale:

```
GET /index.html HTTP/1.1
Host: www-net.cs.umass.edu
User-Agent: Firefox/3.6.10
Accept: text/html,application/xhtml+xml
Accept-Language: en-US,en
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Ogni riga è terminata da `\r\n`, cartteri di ritorno carrello e nuova linea, non riportati nell'esempio.

La prima riga definisce il **tipo** di richiesta (GET, POST, HEAD, ecc...), la **risorsa** richiesta e la **versione** del protocollo che il client vuole utilizzare. La seconda linea contiene

poi l'host del server richiesto, e la terza il processo (qui il browser Firefox) che effettua la richiesta. Le successive rige definiscono il tipo di oggetto che il client è disposto ad ottenere (tipo MIME, lingua, codifica e compressione, ecc...). Infine, l'ultima riga stabilisce le regole di connessione richieste dal client, in questo caso `keep-alive` (quindi HTTP persistente).

Un doppio ritorno carrello e nuova linea segnala la fine delle linee di header e l'inizio dei dati veri e propri.