Oggi andiamo avanti con il laboratorio. Passiamo alla parte UDP.

La fase iniziale dell’UDP ping è identica a quella del TCP Ping. Il Server Pong è comunque in attesa su un socket di tipo stream, e quindi il client UDP deve aprire anche lui un socket di tipo stream esattamente come funziona la versione tcp. La differenza è che questa volta nel messaggio di richiesta del client verso il server ci sarà scritto UDP al posto di TCP. Un’altra differenza è che il socket per la negoziazione sarà diverso da quello usato per fare lo scambio dei messaggi. Il Server avrà aperto un secondo socket di tipo DATAGRAM, su cui dovrà fare il bind tramite una porta. Quando il server risponde OK al client deve anche comunicargli il numero di porta effimera che ha appena predisposto per fare gli esperimenti.

Un’altra differenza è che la versione datagram non è affidabile (sia il messaggio PING che il messaggio PONG possono essere persi). Per ovviare a questo inconveniente è necessario introdurre un timeout a livello applicativo da parte del client ping. Si stabilisce un timeout ragionevolmente lungo, sulla base di quello che potrebbe essere il roundtrip time. Ci sarebbero diversi modi per realizzare questo timeout: in questo caso si lavora solo a livello applicativo e viene cambiata la modalità di funzionamento del socket stesso (di default il socket è in modalità bloccante, ma noi useremo una system call particolare per farlo diventare non bloccante -> la FCNTL).

La fcntl cambia la modalità di funzionamento da bloccante a non bloccante. Se noi lasciassimo la modalità di default, succederebbe che il client Ping manderebbe il messaggio e poi eseguirebbe la recv. Se il socket è rimasto in modalità bloccante il processo viene messo in attesa e può ripartire solo dopo che è arrivata la risposta alla recv. Questo dà ovviamente problemi nel caso in cui il messaggio sia perso, dunque, noi facciamo diventare il socket non bloccante: se quando chiamo la recv il messaggio non è ancora arrivato viene segnalata una condizione di errore che dichiara che il messaggio non è ancora arrivato.  
Poiché il processo va avanti, la recv avrà o un valore positivo che indica che il messaggio è stato ricevuto, oppure la recv restituirà il valore -1, che significa o “non è arrivato ancora nessun messaggio” o “c’è stato un problema”. In tal caso si va a vedere il contenuto della variabile errno, dove si possono trovare due messaggi particolari, EWOULDBLOCK in caso non sia ancora arrivato alcun messaggio (che vuol dire che per far arrivare il messaggio si sarebbe dovuto bloccare), EAGAIN che è la stessa cosa, ma lo standard POSIX dice che ne potrei ricevere uno qualsiasi dei due. Se in errno c’è qualcosa di diverso da questi due valori mentre recv restituisce -1, allora c’è stato un errore di qualche altro tipo.

Quando il socket è in modalità non bloccante, tutte le sue system call diventano non bloccanti, quindi non solo la recv, ma anche la send lo diventa: quando cerco di inviare il messaggio potrebbe succedere che, se questo è di grosse dimensioni, il messaggio non possa essere inviato immediatamente e anche in questo caso otteniamo come valore di ritorno -1 con errno = EWOULDBLOCK o EAGAIN. Per risolvere questa situazione bisogna richiamare nuovamente la send.

Per la parte UDP non funziona la connessione al server webdev.dibris.unige.it. Il problema è dovuto al fatto che le macchine di unige sono dentro a un firewall fatto male.

Vediamo adesso la parte Pong Server (opzionale). L’idea è di avere un processo che rimane in ascolto e che quando riceve una richiesta di connessione crea un altro processo che fa da effettivo server per il client e invia la risposta. Questo è il motivo per cui si crea un secondo socket. Il server che viene aperto da quello in ascolto è fatto per rispondere solo ad un client.

Dal punto di vista della gestione dei processi, si ha un processo principale che funziona come server pong e che crea un socket principale, quello su cui viene fatta la listen. Quando un client ping arriva e chiede di connettersi, la accept crea un secondo socket connesso con questo client. A questo punto il processo chiama la system call fork() per creare una copia identica di se stesso. In questo momento i due processi sono identici tra di loro (eseguono lo stesso codice e hanno una copia di tutte le strutture dati), in particolare il secondo socket ha anche una copia dei socket. La fork() ritorna però due valori diversi ai due processi: il processo “padre” ottiene come valore di ritorno un valore maggiore di zero (che è l’identificatore del processo figlio), mentre il processo figlio, poiché eredita il Program Counter del padre, ottiene un valore di ritorno a sua volta, che però è proprio uguale a zero.

Confrontando quindi il valore ritornato dalla fork con zero, il processo può sapere chi è (se è padre o figlio). Il processo padre torna a fare la accept sul socket principale (in modo da gestire eventuali altre richieste). Il processo che si vede ritornare 0 (il figlio) sa di essere connesso a un client e quindi fa il ping pong.   
Se arriva un altro client si ripete la stessa cosa: il server principale ri-effettua la fork, creando un figlio che eredita dal padre tutti i socket e questo figlio potrà quindi comunicare col client appena accettato.

In questo modo possiamo gestire una quantità arbitraria di client (di chiudere i socket se ne occuperanno i singoli processi server figli).

Un’altra osservazione è che nel momento in cui viene chiamata la fork, abbiamo la relazione di padre-figlio tra processo originario e processi creati. Per quanto strano possa essere ci si aspetta che il processo figlio termini sempre prima del processo padre (visto che il padre sta sempre in ascolto, mentre la vita di un server figlio dura quanto il ping pong con un singolo client). La situazione normale è che i processi figli terminino dopo un po’ di tempo, mentre il processo padre rimane attivo. C’è tutto un meccanismo per lo scambio di informazione tra padre e figli, in particolare c’è un meccanismo di segnalazione di eventi che permette al padre di sapere quando uno dei figli che ha generato termina. Quando ciò avviene il padre può guardare l’Esito del processo figlio che è terminato: tale esito è specificato nella system call “exit()”. Per convenzione exit(0) vuol dire che tutto è andato a buon fine, mente exit(valore !=0) vuol dire che c’è stato un errore: dunque appena un processo finisce non si può deallocare subito tutti i suoi dati (almeno la condizione di terminazione va mantenuta almeno finché il processo padre non la legge o finché questo non termina a sua volta).  
Questo fa sì che un processo che ha eseguito la exit non può più eseguire codice, ma deve mantenere la sua immagine in memoria: per questo motivo è stato creato lo “Stato Zombie” che indica lo stato di un processo che è terminato ma del quale il processo padre non ha ancora letto il valore di terminazione (e quindi continua a occupare risorse nel sistema). Può essere cancellato solo dopo che il padre ha letto il valore di terminazione.

Quando vengono creati tanti processi è importante che il padre si accorga in fretta della loro terminazione, in modo che legga il valore della exit e che questi processi figli terminati possano essere deallocati. Per questo motivo in genere i processi figlio mandano al processo padre delle segnalazioni quando terminano: il padre dovrà quindi contenere degli handler per le segnalazioni di terminazione dei processi figli (i Sig handler dovranno chiamare la system call wait, o waitpid per leggere il valore di terminazione).

A noi non interessa quale è questo valore di terminazione (stiamo solo facendo un ping pong), ma leggerlo è necessario per recuperare le risorse del sistema che sono occupate dai processi Zombie.

Vediamo adesso altri protocolli di tipo applicativo. Abbiamo visto quelli più utili dal punto di vista pratico (il DNS e l’NTP) per il funzionamento della rete. Vediamo adesso dei protocolli applicativi propri.   
Cominciamo con il protocollo per l’invio di messaggi di posta elettronica (l’email). Il protocollo in questione è l’SMTP, che è descritto nella sua versione attuale dall’RFC 5321. È uno dei protocolli più vecchi tra quelli attualmente in uso (la prima versione era specificata nell’RFC 821, ormai obsoleta, sebbene alcune appendici siano ancora valide).

L’SMTP si basa sul trasporto TCP e il servizio originariamente era stato definito sulla porta 25, ma successivamente sono state introdotte altre porte per tenere conto di tutte le funzionalità che sono state aggiunte al protocollo (per conoscerle guarda l’RFC).  
L’idea è che un client può connettersi a un server SMTP in modo da inviare messaggi di posta elettronica: il client manda dei messaggi di tipo testuale codificati in ascii (7-bit) che sono usati per specificare i vari attributi del messaggio di posta (tra i quali l’indirizzo del destinatario, l’oggetto del messaggio e… basta) seguiti dal contenuto del messaggio stesso. Questo messaggio viene salvato all’interno del server e può essere destinato a qualunque destinatario anche non connesso a quel server in particolare. Il destinatario deve quindi avere un Server SMTP attivo (sulla porta 25) e il server di partenza, dopo aver memorizzato il messaggio nella sua memoria interna, cerca di aprire una connessione (sempre sulla porta 25) con la macchina destinataria, basandosi sull’indirizzo. Se il messaggio arriva a destinazione, la connessione si chiude e il messaggio viene inserito in una casella di posta elettronica (che è un file su cui viene eseguita un’operazione di append). Dopodiché, l’idea è che se il destinatario ha un account sulla macchina contenente la sua casella di posta elettronica allora può leggere i messaggi che gli sono arrivati.

Il protocollo viene detto di tipo “Best Effort”, ed è di tipo asincrono. Asincrono vuol dire che il client non deve necessariamente aspettare che il destinatario sia pronto a ricevere il messaggio, questo può essere inviato anche quando la macchina destinataria è spenta: in tal caso è il server SMTP a doversi occupare di questa situazione, provando a inviare il messaggio in un secondo momento (dopo un po’ di tempo e dopo un paio di tentativi). Se il fallimento persiste (per tot. Giorni si presume), il server SMTP invia al client un messaggio sempre di posta elettronica che notifica l’impossibilità di inviare il messaggio. Ecco perché si chiama ”Best Effort”, perché prima di inviare al client la notifica di fallimento, il server fa del suo meglio per recapitare il messaggio.

Abbiamo detto che la codifica è ASCII (7 bit), quindi il messaggio è limitato nel tipo di caratteri che si possono inviare. La limitazione è dovuta al fatto che la prima versione del protocollo faceva riferimento al protocollo (applicativo) TELNET: questo protocollo era il modo per accedere a un terminale su una macchina remota e lì l’interfaccia era una shell (quindi il dialogo che avveniva tra client e server terminale usava solo ascii su 7 bit). Successivamente i messaggi sono stati standardizzati, in modo da distinguere una parte di intestazione dal messaggio vero e proprio: la parte di intestazione termina con un doppio a capo (che è ottenuto con la concatenazione del carattere \r col carattere \n). Quindi dal primo byte fino alla prima riga vuota (ottenuta coi due a capo consecutivi) è tutta intestazione (header).

Il corpo del messaggio segue l’header, questo può essere di lunghezza variabile, perciò la terminazione del messaggio è definita da un punto a inizio riga seguito da un a capo (‘\r\n.\r\n’): quindi l’unico messaggio illegale è il messaggio costituito solo da un punto a inizio riga seguito da un a capo (se si vuole mandare un punto a inizio riga da solo bisogna farlo seguire da uno spazio).

Nella parte header vengono definite delle informazioni, come l’oggetto (subject), il destinatario (to) e il mittente (from), le informazioni Cc, le Bcc (Cc ma senza rendere l’indirizzo visibile agli altri riceventi) e la Data.

Le problematiche di queste informazioni è che sono tutte scritte dal client (in particolare nessuno può sapere se nella From il client ha messo veramente il proprio indirizzo). Quindi sarebbe banalmente facile mandare messaggi con provenienza falsa. Questo ovviamente ha delle conseguenze negative sull’uso della posta elettronica, dunque come evitare di avere eccessivi messaggi di spam? Il modo che è stato messo in piedi per evitare queste problematiche è stato l’inserimento di un processo di autenticazione per l’accesso al server STMP. Al giorno d’oggi un server senza sistema di autenticazione è detto Open Relay ed è considerato illegale, perché troppo sensibile ai messaggi di Spam.

L’autenticazione può avvenire in modi diversi: quello classico è di identificare gli utenti con username e password (analogamente a come funzionava già prima per l’accesso alla casella di posta per leggere i messaggi).

Così facendo un server può tenersi un file di log dei messaggi inviati dagli utenti e, se un client viene scoperto aver inviato dei messaggi contraffatti l’amministratore del server può prendere provvedimenti contro il client incriminato.

Altro problema legati al meccanismo di posta elettronica è la seccatura che è il poter mandare messaggi solo in ascii su 7 bit; per questo motivo è stato modificato il tipo di codifica dei messaggi di posta elettronica , secondo il metodo MIME: esso permette l’aggiunta di allegati in qualsiasi formato al messaggio di posta elettronica (la codifica MIME li convertirà in ASCII durante l’invio e li riconvertirà nel formato normale una volta che il messaggio è arrivato a destinazione).