Passiamo a vedere il protocollo HTTP. L’acronimo sta per hypertext transfer protocol, è uno dei protocolli più recenti tra quelli in largo uso. La sua definizione è reperibile dall’RFC 1945 (risalente al 1996), con la versione 1.0. Successivamente è stata introdotta la versione 1.1 (descritta dall’RFC 2616) e dopo di questa sono state introdotte anche le versioni 2 e 3. Le diverse versioni del protocollo sono tutte considerate valide (fanno tutte parte dello standard), dunque un server http deve poter rispondere a un client che usi una qualunque delle versioni di http.

Il protocollo ha una struttura di tipo client-server e, fino alla versione 2 del protocollo, è stato usato il protocollo di trasporto TCP (standardizzando la porta 80 per contattare il server http). Il dialogo tra client e server prevede che ci siano dei comandi inviabili al server e delle risposte inviabili al client. La comunicazione di comandi e risposte avviene in maniera simile agli altri protocolli che abbiamo visto, quindi, sono codificati in Ascii 7 bit. Il protocollo serve per accedere a risorse che sono localizzate sul server. Quando un client manda una richiesta il server può inviare una tra due risposte, la prima di OK (con codice numerico 200), la seconda è l’errore (il cui codice è compreso tra il 400 e il 499): un esempio classico è l’errore 404 = not found.

Il comando (o metodo) più largamente usato è il comando “get”: serve al client per ottenere la copia di un file presente sul server. La richiesta è tipicamente formata da una prima riga col nome del metodo, seguito da uno spazio e dal nome del file che si vuole reperire. A seguire ciò ci sarà, separata da uno spazio, la versione del protocollo http che si sta utilizzando (1.0, 1.1, 2, 3). La riga termina con un terminatore \r\n.

La richiesta viene analizzata dal server facendone il parsing. Alla fine della riga ci possono essere altre righe, che però sono opzionali, e che costituiscono l’header del protocollo. Le righe saranno costituite da campi “nomeOperazione” seguite dai due punti :, da uno spazio, un’informazione e un \r\n.

L’intestazione si chiude quando si trovano due a capo consecutivi \r\n\r\n.

Dopo l’intestazione ci può essere opzionalmente (a seconda delle opzioni date al comando) un payload.

Una delle opzioni che possono essere date è il nome del browser che il client sta utilizzando (questo perché non tutti i browser sono compatibili con tutte le possibilità offerte dai vari server).

Abbiamo detto che in caso di successo il server risponde con 200 = OK, dopo però sarà necessario che siano inserite le informazioni che il client aveva richiesto. Dunque, dopo la riga OK ci sarà una parte opzionale riguardante i metadati del file, terminata dal doppio a capo, con dopo il contenuto del file per payload. Una cosa interessante del protocollo http è che la richiesta di usare i caratteri ASCII a 7 bit è relativo solo alle intestazioni, il body può contenere byte da 8 bit (ciò permette di inviare come risposta qualunque file senza doverlo codificare, come invece capita col SMTP e il RADIX 64).   
Nonostante questa peculiarità del protocollo, è previsto che nella risposta ci sia un header chiamato “MIME-Type”, quindi viene usato il protocollo MIME (esattamente come con l’SMTP): lo scopo di questo è di caratterizzare in modo univoco il tipo di file binario che viene trasmesso.

Quale potrebbe essere un’alternativa all’uso del protocollo http potrebbe essere il protocollo ftp. La differenza tra i due protocolli è che in ftp c’è la seccatura di dover aprire due connessioni per trasferire i dati (quella di controllo e quella dati), il protocollo http permette di aprire una sola connessione (cioè un solo stream di comunicazione). Un’altra differenza è che l’ftp di default richiede un’autenticazione da parte del client (anche se c’è l’anonymous), mentre il protocollo http non implementa alcuna forma di autenticazione.

Un altro metodo implementato dall’http è il comando HEAD; è molto simile a get (la prima riga è uguale), ottiene come risposta 200 ok o 400...ish e ottiene come informazione i metadati del file richiesto (a differenza del get non si ottiene il contenuto, solo i metadati).

Altri comandi sono il comando PUT e il comando DELETE, che però non sono quasi mai usati. Non sono quasi mai usati perché non ha senso che un client qualsiasi senza alcuna identificazione mi possa cancellare risorse su un server o fare un upload di un file.

I dettagli descritti fin’ora sono comuni a tutte le versioni standard di http. Vediamo adesso le differenze tra le diverse versioni.

La versione 1.0 è la più semplice del protocollo, essa prevede che le connessioni TCP che vengono instaurate dal client verso il server siano non-persistenti: questo vuol dire che il client quando vuole una risorsa, apre la connessione con una richiesta e dopo che il server risponde la connessione viene chiusa (e se vuole altre informazioni il client dovrà riaprirla). La connessione non è usata come una vera connessione TCP (lo stream è usato per trasferire solo una risorsa, è più simile a un datagramma).   
Questo ovviamente ha dei costi in termini di prestazioni: quello principale è che se si vogliono tante informazioni serve aprire tante connessioni e ad ogni connessione aperta serve effettuare un three-way-handshake (che spreca un round-trip-time). Non c’è alcuna ammortizzazione del tempo perso per aprire le connessioni. Questo viene evidenziato introducendo nell’header “Connection\_close” (così il client saprà che il server ha chiuso la connessione).

Il fatto che questa versione del protocollo http riduca l’efficienza nell’accesso alle risorse del web è un po’ un controsenso (infatti http è nato solo come un’alternativa più semplice a ftp). Per questo motivo, quando l’uso di http si è diffuso in ambito commerciale, dalla versione 1.1 la restrizione è stata tolta e le connessioni sono diventate persistenti: quindi se anche la versione 1.0 non è considerata obsoleta, di fatto lo è.

Nel laboratorio di settimana prossima vedremo principalmente la versione 1.0 e la versione 1.1.

A partire dalla versione 1.1 è stato esteso anche l’insieme dei metodi utilizzabili. È stato introdotto anche il metodo POST. Questo metodo serve per permettere al client di mandare informazioni al Server. Questa esigenza nasce dal fatto che così si dà al client di poter scegliere tra più opzioni all’interno di un sito web (tramite magari degli spunti su delle checkbox, o tramite un form). Prima della versione 1.0 si usava lo sporco trucco di mettere nel nome del file della richiesta GET una codifica dei dati che il client voleva mandare al server (quindi la compilazione delle form era possibile con la versione 1.0 ma avveniva tramite un abuso del campo nomefile).

In tutte le sue versioni, il protocollo http è un protocollo senza stato. Cioè all’interno del protocollo il server, dopo aver eseguito il metodo richiesto, si dimentica della richiesta che gli è arrivata e tutto torna come prima. Questo ovviamente può generare delle difficoltà per quanto riguarda le necessità del client (se siamo su uno shop online e mettiamo un oggetto nel carrello vogliamo che quell’oggetto rimanga nel carrello anche se poi andiamo a cercare altro). Per tradurre questa necessità senza modificare i protocolli si fa in modo che sia il client a ricordarsi le cose. Per poter gestire questa comunicazione viene quindi introdotto il meccanismo dei cookies.   
Quando un client manda una richiesta al server, il server può aggiungere alla risposta un cookie. Ciascun cookie è una stringa di caratteri ascii del tipo nome = valore.  
Quando un client si vede arrivare una risposta con l’opzione Set-Cookie nell’header, questo si memorizza i cookies che ha ricevuto.

Il client può ovviamente contattare tanti servers diversi e ogni server deve poter avere i suoi cookies, quindi, nella memoria del client, ogni cookie è associato al nome del server che lo ha rilasciato.

Quando un client ha dei cookies di un server, questo li aggiunge come opzioni alle richieste successive che manda (c’è l’opzione cookie: lista di cookies), in questo modo ogni volta che gli arriva una richiesta il server vede il valore di tutti i cookies che ha settato per quel client.

Come si può usare questo meccanismo per definire uno stato delle transazioni tra client e server (tenendo conto che i client sono anonimi)? Il server legge i cookies che gli arrivano con una richiesta; se non vede nessun cookie nelle opzioni della richiesta il server può generarsi un numero casuale e dare al client un cooki del tipo ID = numero casuale, in questo modo da quel momento in poi il client potrà riprendere le transazioni mandando al server il cookie col suo ID.   
Supponiamo invece di voler realizzare un meccanismo di autenticazione basato sul protocollo http: possiamo associare un identificatore e una password ad un client e questo può mandare una richiesta POST col proprio nome utente e la password. Se l’autenticazione va a buon fine il server può inviargli un cookie che rappresenta l’id del suo account per poter effettuare delle transazioni (se il numero casuale usato per l’id è molto grande difficilmente qualcun altro lo potrà indovinare). Inoltre l’utilizzo di un cookie per muoversi nel proprio account evita all’utente di doversi autenticare ogni volta che cambia pagina all’interno dello stesso sito (gestito dallo stesso server).

Sempre nell’ottica dell’efficienza del protocollo, un modo ulteriore per aumentarla è cercare di ridurre il più possibile trasferimenti di informazioni di grandi dimensioni. Questo potrebbe essere fatto attraverso tecniche di caching. Se un utilizzatore chiede di accedere a una risorsa che era già stata aperta precedentemente (e non troppo tempo prima) si potrebbe far affidamento sulla copia della risorsa che l’utente ha ancora in memoria cache: se questa non è stata aggiornata il server può cavarsela dicendo al client “guarda che la copia che hai vale ancora” ed evitare così di inviare altri file. È possibile inserire a priori una data di scadenza nel file quando lo si invia, così da far sapere al client più o meno quando scade. Tuttavia, non sempre questo è possibile, la scadenza non è sempre prevedibile, inoltre anche inserendo la data il client probabilmente farà comunque una richiesta al server per sicurezza.   
Per questo motivo, torna comodo l’utilizzo del comando HEAD, che permette l’invio dei soli metadati del file interessato (questo permetterà al client di sapere se il file che ha in cache è scaduto o no). Però, questa verifica costa già un roundtrip time e nel caso la cache non sia aggiornata a ciò si aggiunge un altro roundtrip time per richiedere il nuovo file. Per migliorare l’efficienza, quindi, conviene inviare una richiesta di GET con l’opzione “IfModifiedSince” seguita da una data. In questo modo il server guarderà i metadati del file che ha in memoria, controllando se la versione che ha è più aggiornata di quella indicata dal campo IfModifiedSince, in tal caso il server manderà il file aggiornato come risposta alla GET. Se invece la data di ultima modifica del file corrisponde a quella dell’opzione IfModifiedSince, la risposta del server sarà di tipo 304 = Not Modified: ciò verrà interpretato dal client come “la copia che hai in cache è ancora valida”.