Continuiamo a descrivere il protocollo http, aumentando il livello di dettaglio. Vedremo la versione 1.0 e la versione 1.1.

Cominciamo con la 1.0. Abbiamo detto che uno dei suoi limiti principali sono le connessioni non persistenti; ciò significa che quando un client si connette con il server, questo chiude la connessione subito dopo aver inviato una risposta (quindi solo un’informazione). Ci possono essere situazioni in cui il client può voler chiedere più cose allo stesso server, ad esempio se ci sono delle immagini su una pagina e per ognuna di esse il client deve fare una query. Per risparmiare tempo, quindi, il client potrebbe voler aprire contemporaneamente una connessione per ciascuna query che deve fare.

Questa soluzione è però inefficiente perché il server dovrà fare tante three-way-handshake e occupare altrettanti socket per inviare le informazioni. Come possiamo aumentare la velocità di funzionamento del server? Ovviamente la soluzione sta nell’aumentare il numero di thread, in modo che ci sia un thread per ogni connessione e queste siano soddisfatte contemporaneamente.

Una via di mezzo tra avere un server non multithread e averne uno che ne realizza uno per ogni connessione è quello di predisporre un certo numero massimo di thread per le connessioni e quando uno di questi è libero va fare l’accept di una nuova connessione. Lo svantaggio è che non si sa a priori qual è la quantità ottimale di thread da attivare, il vantaggio è che si evita l’overhead di dover creare e cancellare continuamente thread.  
Questa sarà la soluzione che applicheremo nel laboratorio.

Nella versione 1.1 le connessioni vengono aperte e rimangono tali in modo da consentire l’invio di più richieste sulla stessa connessione, così facendo si ammortizza l’overhead iniziale dovuto al three-way-handshake.   
Rendere permanenti le connessioni non era l’unica modifica nel passaggio da 1.0 a 1.1. Si è introdotto anche il meccanismo di Pipelining: questo vuol dire che il client può mandare una prima richiesta, seguita da quante ne vuole senza dover necessariamente aspettare le risposte (se il client ha bisogno di 5 file può inviare direttamente tutte e 5 le risposte). Analogamente, il server dovrà mandare le risposte una dopo l’altra, mantenendole nello stesso ordine delle richieste. Connessione persistente + pipelining diventa più ragionevole dal punto di vista applicativo, perché permette di evitare tanti roundtrip time (che richiedevano alla versione 1.0 di usare il multithreading).  
Dal punto di vista del server, per supportare la versione 1.1 col pipelining può usare sempre un thread per socket (come nella versione 1.0, con la differenza che c’è un numero di socket minore), ma ancora meglio potrebbe essere di avere anche una struttura dati in cui le singole risposte sono preparate da un thread ciascuno e lasciare inviare ogni risposta solo se i thread precedenti hanno già finito (in modo da preservare l’ordine). A tale scopo si potrebbe usare una coda FIFO per salvare le richieste.

Accenniamo alla versione 2.0 del protocollo http. In questa versione si è mantenuta la consistenza delle connessioni, ma si è rimpiazzato il pipelining con un meccanismo più sofisticato: Ogni richiesta viene etichettata con un identificatore, in questo modo il server può rispondere con un ordine qualsiasi premettendo alla risposta l’identificatore della richiesta corrispondente (è sicuramente un metodo più veloce, anche se i file più grandi così arriveranno sempre sicuramente per ultimi).

Vogliamo cercare di ridurre il più possibile la complessità di realizzazione del server. Noi dobbiamo mandare nella risposta un header che contiene i metadati del body (cioè del file che stiamo inviando). Questi metadati vengono presi dalla struttura STAT (l’implementazione dell’INode del file). C’è quindi un’intera fase di preparazione della risposta prima del suo invio: io, server, ho un file system, il client mi invia un pathname (con una GET), io dal pathname devo risalire all’iNode e voglio usare le primitive standard POSIX per fare ciò. All’interno dello STAT trovo tutte le informazioni sul metadati, tranne il MIME TYPE, che va inserito nell’header della risposta. Per fare ciò abbiamo un applicazione, che si chiama “file” e che dato un pathname dice di che tipo è. Di solito, file, dice il tipo in una forma testuale facilmente comprensibile dall’utilizzatore (come “testo ascii” o “file html”) , esiste però un’opzione (-i) che gli si può dare in modo da fargli rispondere col MIME TYPE, esattamente nella forma richiesta dall’header http.

Per facilitare l’implementazione l’idea è di far partire un processo (il server), all’interno del quale sono definite due Pipe e tramite una fork sdoppio questo processo. Poiché il secondo processo condivide i file descriptors del primo posso usare la pipe per mandare messaggi tra i due processi. Il secondo processo fa una exec di file -i e inviare indietro come risposta il MIME TYPE del file richiesto. (il processo che fa la ricerca del myme type rimarrà attivo per tutta la durata del server)

Anziché fare una lettura del file e poi una riscrittura sul socket, viene usata una system call apposita chiamata “send\_file” in cui si inseriscono i file descriptor del file da inviare e del socket su cui inviarlo e fa tutto lui.  
Send file potrebbe restituire successo anche senza avere inviato tutto il file (indicando però quanti byte ha effettivamente inviato), quindi se send file non riesce a inviare completamente il file la prima volta va richiamato in modo da mandare i pezzi rimanenti.

Vediamo le problematiche di sicurezza legate al protocollo http. I nostri client ci invieranno delle richieste relative a un punto di partenza, che non sarà la root del file system, ma sarà una directory specificata nel momento in cui facciamo partire il nostro server (la cartella in genere si chiama “www”) e noi vogliamo che il server possa inviare solo i file contenuti in quella directory. Voglio in particolare che il server NON mandi file esterni a quella directory. Se io nel mio file system ho dei file personali, il server http non deve uscire e girare per il file system, dando a chiunque senza autenticazione i file sulla mia macchina.

Come si può risolvere questo problema? Un modo sarebbe di analizzare tutto il pathname ricevuto come richiesta per escludere la possibilità di uscire dalla cartella dedicata al server. Ciò però è scomodo, perché oltre a richiedere molte righe di codice per fare un parsing del pathname ci possono essere dei bug nel codice che fa il controllo, causando gravi falle di sicurezza. Per ovviare a ciò, basta usare la system call chroot. Questa è una system call molto comoda perché ci permetta di cambiare la root per un particolare processo: in questo modo siamo sicuri che il server non potrà uscire dalla sua cartella www.  
L’unico problema è che solo un programma che ha effective UID 0 può chiamare chroot, ma noi non possiamo lanciare un server e tenerlo con UID = 0 perché sennò si perde il senso di limitare i movimenti del server.

Un altro motivo per far iniziare il server con UID 0 è perché servono i privilegi di amministratore per fare la bind sulla porta 80. Quindi il server dovrà partire con UID 0, fare le sue operazioni di inizializzazione e poi ritornare a un UID non privilegiato.

Per far partire il server con UID 0 ci sono una serie di opzioni possibili: possiamo usare sudo, oppure fare login root, oppure usare Set uid bit. Il login a root è da NON considerare. Il sudo è ragionevole, ma ciò richiederebbe di avere sempre un utente abilitato al sudo lanciare il nostro server, questo non è esattamente quello che vogliamo specialmente nella fase di debugging.   
La soluzione preferibile è usare il Set uid bit, per fare ciò bisogna mettere il server come di proprietà di root e attivare il bit “set uid”, mettendo a 1 il bit di esecuzione per tutti gli altri utenti. In questo modo quando lanciamo il server, avrà real UID uguale a quello dell’utente che lo ha lanciato e Effective UID a 0. Quando il server avrà fatto tutto quello che deve, cambierà il suo Effective UID col Real UID.

In questo modo il server non sarà privilegiato durante l’esecuzione e sarà confinato in una directory del nostro file system.

Da tenere sott’occhio è anche il fatto che la richiesta get può essere condizionale (con l’opizone IfModifiedSince). La data nell’opzione sarà scritta in formato GMT (quindi indipendentemente dal meridiano). In tal caso oltre alla presenza del file dovremo controllare la data di ultima modifica (che a questo punto non diventa più un metadato da inserire nell’header e basta). Se la data di ultima modifica non fosse più recente del valore del campo IfModifiedSince la risposta dovrà essere di tipo 304 Not Modified.