Università degli studi di Pisa

Facoltà di Ingegneria

Elaborato Finale:

Sviluppo di documentazione web su alcuni aspetti dell'interfaccia CGI

realizzato da

Tucci Veneziani

Matteo

Relatore:
prof. M. Avvenuti

Introduzione

Il **world wide web** è un enorme deposito di informazioni disponibili via Internet. Gli utenti possono consultare tali risorse mediante un'apposita applicazione detta *browser*. La navigazione del WEB, come la maggior parte delle applicazioni di rete, segue il paradigma client-server e l'interazione fra il processo client (browser) e il processo server avviene mediante il protocollo HTTP. Il protocollo per il trasporto degli ipertesti, questa è la traduzione di *HyperText Transport Protocol*, è un protocollo di livello applicazione dell' architettura TCP/IP anche detta architettura internet. Nel momento in cui viene digitata una stringa URL nella barra degli indirizzi del browser oppure viene selezionato un collegamento ipertestuale, il browser diventa client di un server residente sul calcolatore specificato dalla stringa. Il server contattato invia al client un documento web che nella maggior parte dei casi corrisponde a un semplice file di testo HTML contenuto nella memoria di massa del calcolatore di classe server. Tale documento viene, una volta ricevuto, interpretato e visualizzato dal browser.

Documenti Web: Statici, Dinamici, Attivi

In realtà i documenti web non sono sempre costituiti da semplici file di testo HTML ma possono essere classificati in tre categorie a seconda delle modalità e della tempistica di aggiornamento dei contenuti.

Statici:

> I documeti *statici* sono quelli di cui abbiamo parlato precedentemente. Vengono contenuti nella memoria di massa del calcolatore su cui gira il web server (successivamente detto per brevità semplicemente *Server* anche se formalmente con questo termine si identifica un processo e non un computer) a partire da un certo punto del file system e il loro contenuto è definito una volta per tutte dall' autore. Quindi tutte le richieste di accesso a un documento statico conducano alla visualizzazione delle medesime informazioni.

Dinamici:

I documenti web *dinamici* non sono conservati all' interno della memoria di massa del *Server* e quindi non esistono in un formato predefinito ma vengono generati per mezzo di apposite applicazioni software distinte dal web server e da questo invocate ogni volta che un browser ne fa richiesta. Quindi le richieste di accesso a questo tipo di "documenti" (è stato detto che in realtà non si tratta di veri documenti ma del frutto di elaborazione remota) possono condurre alla visualizzazione di informazioni diverse di volta in volta. In questo modo è possibile fornire risposte diverse a una medesima richiesta(medesimo URL). La personalizzazione puo' essere effettuata sulla base, ad esempio dell'ora locale al *Server* alla quale la richiesta viene effettuata oppure sulla base dell' IP del richiedente.

Attivi:

I documenti web *Attivi*, come i documenti statici, sono conservati all'interno del *Server*, ma a differenza di questi non sono determinati del tutto. Sono costituiti da codice HTML ma trasportano oltre alle informazioni anche del codice eseguibile localmente al *Client*(Anche in questo caso con questo termine sarà indicato il calcolatore sul quale gira il web client: browser). Il risultato dell' esecuzione di questo codice contribuisce a formare il documento *Attivo* e il suo aspetto può variare in funzione di molti fattori come per esempio l'ora locale al *Client* e le interazioni con l'utente. Questo tipo di documenti è il solo a permettere un interazione "in tempo reale" con l'utente perchè l'esecuzione avviene senza utilizzare la rete. In questo modo è possibile realizzare anche semplici video-games. Successivamente nei due esempi piu' complicati vedremo come integrare elaborazione locale e elaborazione remota.

Vantaggi e Svantaggi dei tipi di documento

I documenti statici, come già detto sono semplici file di testo, una volta realizzati non sono soggetti a cambiamenti e quindi sono affidabili e inoltre permettono all'utente una rapida acquisizione delle informazioni dalla rete. Comunque il loro aggiornamento deve essere realizzato manualmente e quindi se le informazioni da trasportare variano frequentemente questo tipo di documenti non costituisce il mezzo più adatto.
In questi casi è bene utilizzare documenti dinamici. Questi vengono creati appositamente per mezzo di appropriate applicazioni invocate dal server web sulla base di informazioni provenienti dal client o attualmente disponibili. Per esempio sul sito web di un cinema potrebbe essere messo a disposizione degli utenti il servizio della prenotazione dei posti. Per questo servizio non possono essere utilizzate pagine statiche ma dovra' necessariamente essere utilizzata un'elaborazione lato *Server*. Quindi devono esistere delle applicazioni, una per le richieste di disponibilita' l'altra per le prenotazioni, che elaborino, quando invocate dal server, i file dove è memorizzata la situazione dei posti. E' necessario sottolineare il fatto che i browser non distinguono una pagina statica da una pagina dinamica in quanto richiedono i due tipi di documento nello stesso modo e ricevono in entrambi i casi dei caratteri. Inoltre i client ignorano se i dati che ottengono fanno parte di un file residente sul disco del *Server* o sono il risultato dall'esecuzione di un'applicazione.
Lo svantaggio principale dei documenti dinamici è il loro "costo" infatti comportano un maggior carico sul *Server*. Ogni volta che viene effettuata una richiesta che coinvolge un documento dinamico viene creato dal server almeno un nuovo processo (un solo processo se l' applicazione è compilata) il quale richiede tempo, memoria e rallenta il sistema (es. aumenta il numero di page fault, aumentano i tempi di scheduling). Quindi occorrono calcolatori più potenti e costosi per ospitare i server web. Inoltre il "costo" si fa sentire anche sul lato client in termini di tempo di attesa. Potremmo aggiungere a questi costi anche il fatto che la realizzazione di pagine dinamiche è molto più complicata perchè necessita di buone conoscenze di programmazione. Altro piccolo problema consiste nell'incapacità di mostrare l'informazione in divenire una volta che questa è stata visualizzata.
Per esempio potrebbe succedere che un utente che vuole riservare dei posti al cinema ottenga una risposta affermativa alla richiesta di disponibilità e quindi passi alla

prenotazione. A questo punto l'utente può vedere ugualmente respinta la sua richiesta. Infatti nel periodo di tempo che va dalla disponibilita' alla prenotazione altri utenti possono aver riempito i posti rimasti liberi e la pagina dinamica generata a seguito della prima richiesta non e' cambiata. Il principale vantaggio dei documenti attivi invece consiste proprio nella loro capacità di aggiornarsi continuamente e nella possibilità di realizzare una vera interattivita' in quanto tutta l'elaborazione avviene localmente al browser. Purtroppo questi documenti sono complicati da realizzare e non sono molto affidabili poichè per la loro visualizzazione occorrono browser aggiornati e macchine *Client* potenti. Inoltre i documenti attivi possono dar luogo, almeno potenzialmente, a problemi di sicurezza perche' possono importare oltre che esportare informazioni.

Web Server e documenti dinamici

Abbiamo già detto che i browser non distinguono una pagina dinamica da una statica. I server web, invece, se vogliono gestire pagine dinamiche devono essere abilitati a farlo. In primo luogo i web server devono essere in grado di riconoscere quando un URL si riferisce a una pagina dinamica. In questi casi infatti, invece di ricercare un file all'interno della memoria di massa, il server dovrà invocare un opportuna applicazione determinata in base al nome del documento richiesto e restituire l'output di tale applicazione al client. Quindi deve esistere un insieme di regole che determini il modo in cui il server web passa i parametri a tali programmi e come questi gli restituiscono i risultati. Uno dei Web Server più diffusi al mondo è Apache scaricabile gratuitamente sia per ambiete Microsoft sia per Unix/Linux. Apache è presente nelle principali distribuzioni di Linux come in Linux-Mandrake 8.0. Successivamente ci riferiremo ad un server Apache 1.3.19 in ambiente Linux Mandrake 8.0. Un server web riconosce una richiesta di una pagina dinamica in base all' URL infatti il server riceve dal client, non l'intero URL, ma soltanto quello che si dice il path virtuale. Questo deve essere tradotto dal server in un path reale che si riferisce al file system in modo da soddisfare la richiesta con una documento . Se la parte iniziale del path virtuale corrisponde a una directory con un nome particolare, solitamente */cgi-bin* ma potrebbe essere un nome qualunque definito tra le opzioni di configurazione, il server web capisce che si tratta di una pagina dinamica. Allora se il path virtuale è */cgi-bin/esempio*

Apache ricercherà, in una apposita directory "reale" specificata nei file di configurazione e corrispondente a */cgi-bin*, un file di nome esempio e lo eseguirà restituendo l'output più o meno intatto al browser che aveva fatto la richiesta. Per maggiori informazioni sugli aspetti della configurazione di Apache APPENDICE D

Interfaccia CGI

Precedentemente abbiamo detto che il server web e le applicazioni addette alla generazione di pagine dinamiche devono rispettare un insieme di regole per lo scambio dei parametri e per la restituzione dell'output. Quindi tra le due entità deve esistere un interfaccia tramite la quale avviene la comunicazione. La *Common Gateway Interface*(CGI) è uno standard ampiamente diffuso per interfacciare applicazioni esterne con un Web Server. Le applicazioni dedicate alla generazione di documenti dinamici in questo contesto prendono il nome di Programmi CGI. Una tipica interazione tra un client e un server che prevede una richiesta di un documento dinamico comprende le seguenti operazioni:

1. Il client, dopo aver stabilito una connessione TCP con un web server, tramite il protocollo HTTP, invia la richiesta di un documento dinamico con eventuali altre informazioni come parametri d'ingresso. (Successivamente sarà mostrato che un comodo strumento per passare dei dati a un web server è costituito da una FORM).
2. Una volta che la richiesta è arrivata al server questo tramite l'interfaccia CGI richiama il programma eseguibile indicato, passandogli come argomento i dati arrivati dal client.
3. A questo punto il programma CGI può compiere le sue operazioni: per esempio può interagire con un database contenuto sul disco del *Server*. Terminata l'esecuzione il programma restituisce il risultato della sua elaborazione al web server così come prescritto dall'interfaccia CGI. In questo caso il programma si comporta come un *gateway (ingresso,uscita)* tra il server e il database cioè aggancia il database al web rendendolo disponibile da tutto il mondo.

4. Il server invia al client i dati elaborati dal programma CGI tramite il protocollo HTTP

Programmi CGI

L' interfaccia CGI definisce soltanto il metodo con cui i dati sono passati dal server all' applicazione e viceversa, quindi si tratta della definizione dell' I/O. Di contro i programmatori hanno libertà di scelta su gran parte dei dettagli riguardanti l' applicazione. Il programma CGI può essere scritto in qualsiasi linguaggio che permetta ad esso di essere eseguito sul sistema:

- C/C++
- Perl
- linguaggio di shell di UNIX
- Fortran

Se viene utilizzato un linguaggio di programmazione come C/C++ o Fortran è necessario compilare il programma prima che questo possa essere depositato nella directory corrispondente alla directory virtuale /cgi-bin dove può essere eseguito dal server web. Se invece viene utilizzato un linguaggio interpretato come Perl o qualunque linguaggio di shell di UNIX il file contenente il codice dovrà risiedere nell'apposita directory. Molte persone preferiscono utilizzare linguaggi interpretati per scrivere i loro programmi CGI perchè in questo modo è più semplice effettuare il debug e la manutenzione. Per questo motivo una qualunque applicazione CGI può essere chiamata, sebbene impropriamente, script CGI. Nella scelta del linguaggio da utilizzare deve essere preso in considerazione

anche l'aspetto tempo e costo di esecuzione. E' chiaro che i programmi scritti utilizzando linguaggi compilati permettono tempi di esecuzione più bassi a parità di elaborazione rispetto a linguaggi interpretati e inoltre la richiesta da parte di un client di un documento dinamico generato da un script CGI comporta sul lato server la creazione di due processi e quindi un ulteriore overhead in termini di tempo e costo di esecuzione.

Output per i programmi CGI

I programmi CGI devono sempre restituire al server dei dati e questo viene effettuato mediante lo "standard output". L' output può essere di qualsiasi tipo, perchè lo standard CGI non impone limitazione alla natura dei documenti generati. Ad esempio, i programmi CGI possono produrre testo o immagini digitali. D'altro canto possono anche produrre soltanto istruzioni per il server in modo che questo possa recuperare un documento da inviare al client. I dati prodotti da un'applicazione devono cominciare con un piccola intestazione simile ad un header HTTP dopo la quale seguono, se esistono, i dati veri e propri. Nel caso in cui il corpo sia vuoto è nell'intestazione che il server trova le informazioni per rintracciare un documento. Esistono due forme di output che le applicazioni CGI possono fornire al server. Normalmente lo header fornito dalle applicazioni è interpretato dal server e poi l'intero output è inviato al client (Parsed header output) in altri casi invece un programma CGI può evitare l'overhead al server necessario per effettuare l'analisi dell'output e così parlare direttamente al client (Non-Parsed header output).

Non-Parsed header

Affinché sia possibile distinguere gli script che effettuano questo tipo di output dagli ordinari programmi, l'interfaccia CGI richiede che il nome di tali applicazioni inizi con nph- . Gli script che utilizzano NPH output devono restituire un' intestazione HTTP completa quindi devono conoscere qual' è la versione del protocollo utilizzata per determinare il corretto formato della risposta. Successivamente quando tratteremo l'input per i programmi CGI e le variabili di ambiente vedremo come questi vengono a conoscenza della versione corretta del protocollo da utilizzare nella risposta.

Parsed header

La risposta degli script comprende sempre un piccolo header e un body separati da una linea vuota. Eventualmente il corpo può essere nullo. L'intestazione è costituita sia da campi definiti dallo standard CGI sia da campi dell'intestazione HTTP. I primi sono interpretati dal server web e sono nello stile HTTP mentre i campi HTTP sono inviati direttamente al client e sono facoltativi infatti se non vengono forniti dall' applicazione CGI vengono inseriti dal server. Almeno uno dei campi previsti dall'interfaccia CGI deve essere fornito dallo script nello header della risposta, ogni campo non può essere utilizzato più di una volta e devono essere messi su linee differenti.

Campi dell' intestazione CGI: Attualmente esistono tre direttive per il server da includere nell' intestazione CGI:

- Content-type
- Location
- Status

Content-type: MIME type Questo primo campo è senza dubbio il piu' utilizzato dei tre e serve per indicare il tipo del documento che viene generato dallo script. Nel caso in cui l'applicazione CGI generi una pagina HTML l'output dovrà iniziare con una linea del tipo: *Content-type: text/html (new line)* invece nel caso in cui lo script generi del semplice testo la linea sarà: *Content-type: text/plain (newline)*. Se invece il programma CGI prevede di generare un immagine, come nel caso delle applicazioni che funzionano da contatori, dovremmo indicare il tipo MIME appropriato dell' immagine.

Location: URL, path, Virtual path Questo campo è utilizzato per specificare al server che lo script restituisce un riferimento ad un documento piuttosto che un documento dinamico. L' argomento di questo campo può essere un URL, un path oppure un path virtuale. Nel primo e nel secondo caso il server deve generare un messaggio di risposta HTTP del tipo "HTTP/1.0 302 redirect" a meno che lo script non fornisca esplicitamente un diverso messaggio di stato con la direttiva Status. Se invece utilizziamo un path virtuale il server deve generare la risposta che avrebbe fornito alla richiesta del documento specificato da tale path. L' ultima scelta da alcuni problemi inquanto il documento che viene visualizzato appare come se il suo URL fosse quello del documento dinamico e eventuali links potrebbero non funzionare se realizzati mediante path relativi.

Status: digit-digit-digit reason-phrase il campo "Status" è utilizzato per indicare al server quale status code, tra quelli ammessi dal protocollo HTTP, deve utilizzare nel messaggio di risposta al client. Se l'applicazione CGI non provvede a fornire nell'intestazione un campo status allora il server fornirà per default "200 OK" o "302 redirect" se viene utilizzata la direttiva *Location*. Se un programma viene utilizzato per gestire una particolare condizione incontrata dal server come per esempio l'errore "404 Not Found" allora questo dovrà utilizzare la direttiva status per propagare l'errore al client es. *Status: 404 Not Found*.

Esempio di un CGI script e il suo output

Di seguito viene riportato un semplice programma CGI scritto utilizzando il linguaggio shell di UNIX. Un componente fondamentale, per quanto riguarda il rapporto con l'utente, cioè di "alto livello", del sistema operativo, è l'interprete dei comandi. Ci sono due tipi di interprete dei comandi: ad interfaccia testuale ed ad interfaccia grafica. L'interprete dei comandi testuale anche detto "Limea di comando" prende il nome di *Shell*. Il linguaggio di shell è quindi il linguaggio con cui vengono impartiti gli ordini al sistema operativo dalla riga di comando e rappresenta un vero e proprio linguaggio di programmazione ad alto livello per mezzo del quale si possono realizzare complessi programmi detti script che in ambiente UNIX/Linux sono l'equivalente in DOS dei programmi .bat . Lo script contiene comandi scritti nello stesso formato che gli utenti userebbero per digitarli da tastiera ad eccezione della prima riga la quale è commentata e che quindi viene ignorata nell'esecuzione ma che serve al server web per capire quale interprete usare per eseguire il programma. L'omissione della prima riga provoca il non funzionamento del programma (provare per credere).

```
#!/bin/sh

echo Content-type: text/plain
echo

echo  data e ora locali al server sono : `date`
```

Il comando *echo* genera una linea di output per ogni chiamata e in assenza di argomenti genera una linea nuova. Questo è quindi il modo con cui si inviano dati sullo standard output per il server. Il comando *echo* interpreta ciò che è riportato tra accenti gravi come un comando e lo sostituisce con il risultato dell'esecuzione. Per provare questo script come applicazione CGI occorre prima di tutto copiarne il codice in un file a se stante es. *prova* e poi spostarlo nella directory indicata nei file di configurazione del web server relativa ai programmi CGI. Inoltre occorre abilitare i permessi di esecuzione del file *prova* per la categoria di utenti "other" con il comando

```
chmod o+x prova
```

Infatti il server web è considerato un utente indipendente con il nome simbolico di Apache oppure httpd. A questo punto basta immettere nella barra degli indirizzi del browser l'URL che è del tipo *http://server:porta/cgi-bin/prova* in modo che il server chiami lo script CGI e ne invii l'output al client. Se il browser da cui effettuiamo la richiesta del documento dinamico è locale al server web l'URL sarà *http://127.0.0.1:porta/cgi-bin/prova* oppure *http://localhost:porta/cgi-bin/prova*. Quello che il server restituisce al client al termine dell'esecuzione dello script varia in funzione del momento in cui viene effettuata la richiesta ed è comunque puro testo del tipo:

```
data e ora locali al server sono: Fri Aug 9 14:19:06 EDT 2002
```

Per la realizzazione dello script *nph-prova* occorre determinare il tipo di protocollo con il quale è stata fatta la richiesta. Questo è necessario per fornire una risposta corretta al client. Nel caso in cui lo script venga referenziato mediante il protocollo HTTP/1.0 allora il codice dovrebbe essere del tipo:

```
!/bin/sh
echo HTTP/1.0 200 OK
echo Server: Apache
echo Content-type: text/plain
echo

echo  data e ora locali al server sono : `date`
```

Anche nell'intestazione di *nph-prova* compare *Content-type* ma in questo caso non si tratta di una direttiva per il server ma di un campo HTTP. Successivamente nel paragrafo che parla delle variabili di ambiente verra' mostrato come determinare il tipo di risposta da utilizzare.
Maggiori chiarimenti su come provare i programmi CGI si trovano nella parte Programmare CGI in C/C++ oppure nell'appendice D.

Input per i programmi CGI

Al momento dell' esecuzione di un programma CGI il server web può passargli degli argomenti rendendolo parametrico. Questa è una tecnica utile per generare documenti dinamici diversi, personalizzati, con lo stesso programma. Gli argomenti per le applicazioni CGI possono essere forniti dai client in vari modi.
Il protocollo HTTP, di cui si parla in maniera più approfondita nell'Appendice C è un protocollo di livello applicazione della pila TCP/IP e viene utilizzato per lo scambio dei documenti tra client (browser) e web server. La richiesta di servizio del client al server è costituita da un header e da un body. Quest'ultimo, anche detto paylod, può eventualmente essere nullo. L'intestazione è composta da una parte fondamentale formata da un campo method, dall' URL di un documento senza alcun riferimento all'indirizzo del server e da un ulteriore campo che specifica la versione del protocollo da utilizzare. Il campo method definisce il tipo di operazione richiesta dal client e può assumere i seguenti valori: GET, HEAD,POST,PUT,DELETE. Il campo GET comunica al server che il browser si aspetta di ricevere un documento mentre HEAD viene utilizzato per richiedere soltanto l'intestazione di una pagina e non l'intero documento. Il comando POST serve per inviare dati al server. I comandi PUT e DELETE spesso non vengono implementati inquanto servono rispettivamente a inviare e cancellare file sul *Server*. Le richieste effettuate mediante GET,HEAD,DELETE hanno il body vuoto cioè non hanno informazioni concatenate all'intestazione mentre quelle fatte con POST e PUT hanno informazioni nel payload. Nel primo caso si tratta del contenuto di una FORM mentre nel secondo del contenuto di un file. Il server comunica con i programmi CGI principalmente in tre modi:

- Variabili di Ambiente
- Linea di Comando
- Standard input

Variabili di ambiente

Il server web può passare alle applicazioni CGI dati tramite le variabili di ambiente. Queste vengono settate dal server web al momento dell'invocazione del programma CGI. Ci sono moltissime variabili di ambiente e unadelle più utilizzate è QUERY_STRING. Le richieste dei browser avvengono di default mediante il metodo GET e quindi sono prive di payload. In questi casi i valori dei parametri possono essere forniti tramite l'aggiunta di un suffisso alla stringa URL inviata al server. Questo divide le stringhe URL ricevute in prefisso, che rappresenta un certo documento dinamico, e suffisso, che è passato al programma CGI come argomento. Sintatticamente il "?" serve da separatore tra prefisso e suffisso. Il prefisso denota un documento, e il server deve poter associare un programma CGI al prefisso ricevuto. Identificato il programma, il server lo esegue passandogli come argomenti tutto ciò che segue il punto interrogativo. I programmi CGI in questo caso ottengono i parametri utilizzando una variabile di ambiente appositamente settata dal server. La variabile di ambiente destinata al passaggio degli argomenti nel caso in cui la richiesta sia effettuata mediante il metodo GET, e' QUERY_STRING e spesso si indica con tale nome (query_sting) anche il suffisso dell'URL. Per ottenere il valore di una variabile di ambiente in un sistema di tipo UNIX dobbiamo utilizzare una funzione della libreria standard: *char *getenv(const char *name)*. Tale funzione cerca tra le variabili di ambiente quella il cui nome corrisponde alla stringa passata per argomento e restituisce un puntatore al valore della variabile di ambiente oppure NULL se non c'e' stato match. Quindi un programma CGI scritto in C++ che gestisce richieste di tipo GET può ottenere la query_string utilizzando il seguente codice:

```
int  main(){
  ...
  char* query= getenv("QUERY_STRING");
  ...
}
```

Le informazioni del suffisso possono essere aggiunte all'URL sia manualmente, digitandole all'interno della barra degli indirizzi del browser, sia per mezzo di una FORM. Maggiori informazioni su questo punto sono riportate all'interno della sezione Decodificare il contenuto di una FORM. Inoltre tali informazioni possono essere manualmente incorporate in un collegamento HTML (ancora) che riferisce uno script. In ogni caso la stringhe che costituiscano il suffisso devono essere codificate secondo lo standard URL che prevede la sostituzione degli spazi con il carattere '+' e la codifica dei caratteri speciali con %xx dove xx è la loro rappresentazione esadecimale. Per maggiori informazioni riguardanti l'URL Appendice A.

Altre importanti variabili di ambiente sono:

```
CONTENT_LENGTH
CONTENT_TYPE
GATEWAY_INTERFACE
PATH_INFO
PATH_TRASLATED
REMOTE_ADDR
REMOTE_HOST
REQUEST_METHOD
SCRIPT_NAME
SERVER_NAME
SERVER_PORT
SERVER_PROTOCOL
SERVER_SOFTWARE
```

CONTENT_LENGTH: Questa variabile d'ambiente è settata dal server web con il numero di byte presenti nel body della richiesta HTTP. Se la richiesta è effettuata con un metodo che non prevede alcun dato nel body questa variabile non è definita. Quando viene applicata la funzione getenv alla stringa CONTENT_LENGTH si ottiene il puntatore ad una stringa di cifre. Se vogliamo convertire tale stringa in intero possiamo utilizzare la funzione della libreria standard *int atoi(const char* nptr);*.

CONTENT_TYPE: Se la richiesta include un corpo allora questa variabile viene settata con il tipo MIME dell'entità nel corpo.

GATEWAY_INTERFACE: Questa variabile è impostata dal server con la versione dell'interfaccia CGI con la quale intende dialogare con le applicazioni es: CGI/1.1 .

PATH_INFO: L'interfaccia CGI permette di inviare alle applicazioni CGI extra informazioni inglobandole all'interno dell'URL dopo il path dell'applicazione e prima della eventuale query_string. L'utilizzo piu' frequente di PATH_INFO e' specificare un path che deve essere interpretato dall'applicazione. Data un'applicazione CGI di nome *prova* nella cartella relativa all'alias *cgi-bin* e' possibile utilizzare la variabile PATH_INFO per comunicarle quale e' il path relativo, per esempio a DocumentRoot, da elaborare. Se sulla barra di un browser venisse digitato il seguente URL:

```
http://myserver/cgi-bin/prova/immagini/montagna.jpg?matteo
```

il programma CGI di nome prova otterrebbe:

```
QUERY_STRING -->"matteo"
PATH_INFO -->"/immagini/montagna.jpg"
```

PATH_TRANSLATED: Il server web provvede a fornire una versione completa di PATH_INFO cioè trasforma il path virtuale in path fisico completandolo mediante DocumentRoot. Considerando l'esempio precedente se il DocumentRoot fosse /var/www/html/ allora l'applicazione CGI otterrebbe la stringa /var/www/html/immagini/montagna.jpg .

REMOTE_ADDR: Il server setta tale variabile con l'indirizzo IPv4 o IPv6 del client che ha effettuato. Tale indirizzo non è necessariamente quello dell'host dell'utente infatti potrebbe esserci un proxy web.

REMOTE_HOST: Permette allo script CGI di venire a conoscenza del nome di dominio completo del client che ha effettuato la richiesta se questo esiste altrimenti viene settata con un valore nullo.

REQUEST_METHOD: Questa variabile d'ambiente viene inizializzata dal server web con il nome del metodo con il quale viene effettuata la richiesta del documento dinamico. Tale variabile può essere molto utile quando bisogna realizzare un programma CGI con dei parametri, che sappia gestire ogni tipo di richiesta.

SCRIPT_NAME: Questa variabile viene inizializzata con il path che identifica il programma CGI all'interno del *Server*. Le stringhe PATH_INFO e QUERY_STRING non fanno parte di tale variabile d'ambiente.

SERVER_NAME: Questa variabile trasporta l'hostname del server così come viene determinato dalla self-referencing URL .

SERVER_PORT: Questa variabile contiene il numero della porta verso la quale è stata inviata la richiesta.

SERVER_PROTOCOL: Questa variabile viene inizializzata con il nome(HTTP) e la versione del protocollo con il quale è stata effettuata la richiesta. E' utile per la realizzazione di nph-script.

SERVER_SOFTWARE Tale variabile d'ambiente viene inizializzata con il nome e la versione del software del server web che gestisce le richieste ed esegue il gateway (applicazione CGI).

Per la comprensione delle variabili d'ambiente può essere utile utilizzare un piccolo programma CGI di prova come il seguente:

```
#include < iostream.h >
#include < stdlib.h >

int main(){
  cout<<"Content-type: text/plain\n\n";
  cout<<"QUERY_STRING: "<< getenv("QUERY_STRING")<< endl;
  cout<<"CONTENT_LENGTH: "<< getenv("CONTENT_LENGTH")<< endl;
  cout<<"PATH_INFO: "<< getenv("PATH_INFO")<< endl;
  cout<<"PATH_TRANSLATED: "<< getenv("PATH_TRANSLATED")<< endl;
  cout<<"SCRIPT_NAME: "<< getenv("SCRIPT_NAME")<< endl;
  cout<<"GATEWAY_INTERFACE: "<< getenv("GATEWAY_INTERFACE")<< endl;
  cout<<"SERVER_NAME: "<< getenv("SERVER_NAME")<< endl;
  cout<<"SERVER_PORT: "<< getenv("SERVER_PORT")<< endl;
  cout<<"SERVER_PROTOCOL: "<< getenv("SERVER_PROTOCOL")<< endl;
  cout<<"SERVER_SOFTWARE: "<< getenv("SERVER_SOFTWARE")<< endl;
  return 0;
}
```

Linea di comando

Se viene utilizzato un programma CGI che gestisce richieste di tipo GET e non sta decodificando il contenuto di una FORM allora la query string può essere ottenuta direttamente decodificata dalla command-line. Quindi ogni parola della query string sarà in una sezione differente del vettore di stringhe ARGV. Per esempio avendo una richiesta del tipo http://myserver/cgi-bin/prova?matteo+tucci la QUERY_STRING è *"matteo+tucci"* mentre ARGV[0]=prova, ARGV[1]=matteo e ARGV[2]=tucci. La linea di comando, per ottenere i parametri, deve essere utilizzata quando il programma CGI non elabora i risultati di una form. Il server analizza il contenuto della variabile QUERY_STRING e se trova la presenza del carattere "=", non codificato, non permette l'acquisizione dei parametri dalla linea di comando ponendo l'intero argc a 1.

Per verificare quanto sopra si possono utilizzare alcune linee di codice del tipo che segue:

```
#include < iostream.h >
#include < stdlib.h >

int main(int argc,char **argv){
  cout<<"Content-type: text/plain\n\n";
  cout<< argc<< endl;
  for(int i=0; i< argc; i++)
    cout<< argv[i]<< endl;
  return 0;
}
```

Poniamo che il risultato della compilazione di tale programma sia il file *prova*. Nel caso in cui nel suffisso dell' URL sia presente un carattere "=" es. *nome=Matteo* allora l'output interpretato da un browser, sarebbe:

```
1
prova
```

Quindi non sarebbe possibile utilizzare la linea di comando per ottenere i parametri.

Standard Input

Le richieste effettuate dai browser ai server web con i metodi POST e PUT trasportano delle informazioni nel body del pacchetto HTTP dopo l'intestazione. Tali informazioni vengono spedite alle applicazioni CGI attraverso il file descriptor dello standard input. Gli script non sono obbligati a leggere i dati e i web server non sono obbligati dallo standard a fornire una condizione di EOF quindi per determinare quanti byte leggere il programmatore dove far uso della variabile di ambiente CONTENT_LENGTH. Il programmatore ha a sua disposizione anche la variabile CONTENT_TYPE che nel caso in cui i dati inviati al programma siano il contenuto di una FORM assume il valore *application/x-www-form-urlencoded* .L'esempio che segue mostra come ottenere i dati di ingresso dallo standard input.

```
#include< iostream.h >
#include< stdlib.h >
#include< string.h >

int  main(){
  cout<<"Content-type: text/plain"<<'\n'<<'\n';
  char* query;
  char* len=getenv("CONTENT_LENGTH");
  cout<<"i caratteri nello standard input sono "<< len << endl << endl;
  int lun=atoi(len)+1;
  cout<<"i caratteri da allocare sono "<< lun << endl<< endl;
  query=new char[lun];
  cin.get(query,lun);
  cout<<"i caratteri letti sono "<< strlen(query) << endl << endl;
  cout<<"Le informazioni inviate "<< endl << query << endl;
  return 0;
}
```

La funzione membro get, della classe istream, esiste in varie versioni. In questo caso i caratteri dello stream di ingresso, provenienti dal server web, vengono letti e trasferiti in query finché non ne sono stati letti lun-1. L'ultimo byte del buffer è riservato al carattere nullo inserito automaticamente. Successivamente nel paragrafo Decodificare il contenuto di una form verrà illustrato come possono essere fatte richieste con il metodo POST attraverso un browser e come vengono solitamente passati gli argomenti agli script CGI sia con POST che con GET.

GET vs POST

Sono state specificate già alcune differenze tra i due metodi HTTP. Innanzi tutto il protocollo HTTP specifica differenti utilizzi per GET e POST. In secondo luogo i due metodi prevedono meccanismi differenti per il passaggio delle informazioni al server: GET attraverso l'intestazione della richiesta all'interno dell'URL, POST attraverso il body del pacchetto HTTP. Una piccola differenza è costituita dal fatto che quando si inviano informazioni ad uno script mediante una richiesta GET il browser non comunica all'utente che si stanno per inviare informazioni non cifrate sulla rete. Infatti queste vengono trasmesse all'interno dell'URL e il browser è inconsapevole di farlo. Quando si utilizza POST invece il browser, prima di effettuare la richiesta, genera una piccola finestra di prompt con un avvertimento per lo user. Un ulteriore importante differenza sta nella quantità di informazioni trasportabili con i due metodi. Il metodo GET passa i suoi dati come parametri della linea di comando . Alcuni sistemi UNIX hanno una limitazione di 256 caratteri sulla linea di comando e quindi se la lunghezza del suffisso dell'URL supera tale dimensione il metodo GET non è adatto e deve essere usato POST.

Decodificare il contenuto di una FORM

Come precedentemente illustrato è possibile trasmettere i parametri ai programmi CGI integrando i valori di questi all'interno dell'URL. Questa procedure, che permette la trasmissione degli argomenti con richieste di tipo GET, può essere effettuata manualmente scrivendo il valore dei parametri direttamente nella barra degli indirizzi del browser. Questa via purtroppo è molto scomoda infatti il suffisso, distinto dalla parte dell'URL che identifica un documento attivo tramite il carattere "?", deve essere codificato secondo lo standard URL. Inoltre il passaggio degli argomenti mediante la stringa URL può essere effettuato soltanto per mezzo del metodo GET che come precedentemente illustrato presenta forti limitazioni. Quindi per evitare di scrivere direttamente la query_string all'interno della barra degli indirizzi del browser e per poter effettuare anche richieste con il metodo POST, che permette la trasmissione di un maggior numero di informazioni, si utilizzano le FORM HTML. Una FORM è un modulo elettronico costituito da diversi campi che l'utente può riempire, e da almeno un bottone(submit) la cui pressione, mediante un click del mouse, provoca l'invio dell'informazioni al server. Gli elementi di una FORM sono racchiusi tra due tag :

```
< form action="URL" method="post" | "get">
                ...
< /form >
```

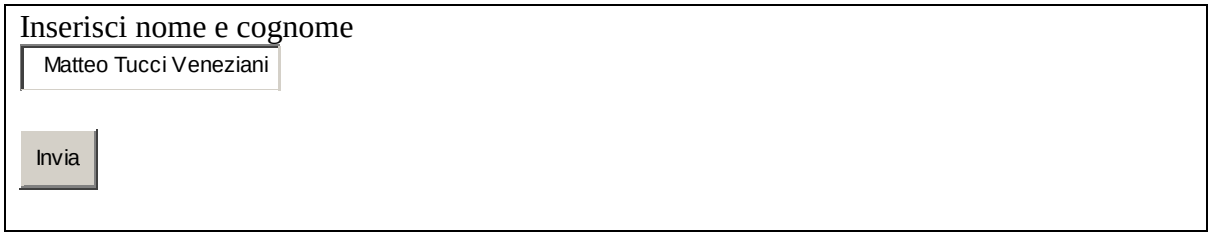
Se il valore dell'attributo method e' GET le informazioni digitate dall'utente vengono prese e attaccate all'URL dopo il carattere "?" come prevede lo standard URL e secondo un certo formato (codifica). Quindi i dati vengono spediti attraverso l'intestazione di un pacchetto HTTP ad un server il quale li rende disponibili attraverso la variabile di ambiente QUERY_STRING al programma CGI identificato dal valore dell'attributo *action*. Se il valore di questo attributo è un URL completo allora identifica anche il *Server* altrimenti se è un path virtuale allora il server web al quale effettuare la richiesta è quello che ospita la pagina con la FORM.

Se il valore del campo method è POST il risultato della FORM è il medesimo. Le informazioni digitate dall'utente vengono codificate come prevede lo standard URL e aggregate secondo un certo formato. La prima operazione adesso non sarebbe necessaria infatti la stringa che costituisce l'input per il programma CGI viene inviata al server attraverso il body HTTP. L'applicazione in questo caso viene in possesso dei dati di ingresso attraverso lo standard input.

Resta da chiarire in che modo vengono codificate le informazioni che l'utente inserisce nei campi della Form cioè in che modo si realizza un unica stringa consecutiva a partire dai dati inseriti in una Form. Tutti gli elementi di ingresso di un modulo elettronico hanno un nome definito dal valore (name) del campo attributo NAME. La stringa risultato della Form, che viene concatenata all' URL dopo il carattere "?", nel caso di richiesta GET, o inviata nell'entity body, nel caso di richiesta POST, è costituita da una sequenza di coppie name=value separate da caratteri "&". Ognuna di tali coppie e' URL encoded: gli spazi sono sostituiti con dei "+" mentre alcuni caratteri speciali (non alfabetici) dal carattere "%" seguito dalla loro codifica esadecimale. Decodificare il contenuto di una Form significa ottenere i valori originali dei singoli campi del modulo elettronico così come inseriti dall'utente a partire dalla stringa dei parametri ottenuta in ingresso dall'applicazione. Tale stringa viene spesso chiamata semplicemente query_string.
Di questo problema, chiaramente, esistono moltissime soluzioni rintracciabili sulla rete nei piu' disparati linguaggi di programmazione . La procedura standard, che è quella seguita nei due esempi di applicazioni CGI nella sezione successiva, prevede inizialmente la suddivisione della query_string attraverso i caratteri "&". Successivamente per ogni coppia name=value ottenuta da questa elaborazione è necessario procedere ad una decodifica: ogni carattere "+" deve essere sostituito da uno spazio e ogni volta che è presente un carattere "%" deve essere eliminato e i due caratteri successivi sostituiti con il carattere da essi rappresentato se interpretati come cifre esadecimali. In seguito sono esposti alcuni esempi di Form in modo da comprendere come i vari elementi dei moduli contribuiscano alla generazione della query_string.

```
< form action="/cgi-bin/prova" method="..." >
< p align="left">
Inserisci nome e cognome< br>
< input type="text" name="nomeCognome" value="Matteo Tucci Veneziani">< br>
< input type="submit" value="Invia">
< /p>
< /form>
```

Il codice HTML precedente realizza una form con un unico campo per l'introduzione di testo. Il risultato dell'interpretazione di questo codice da un brwser è riportato sotto:

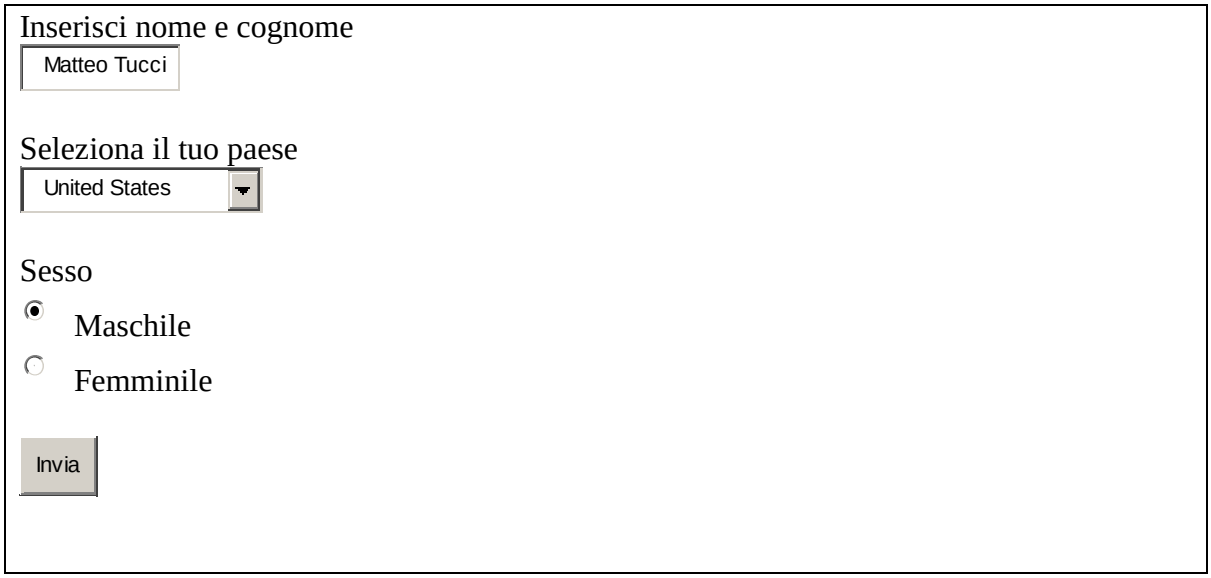


Con questo tipo di input la stringa che verrà fornita allo script sarà: **nomeCognome=Matteo+Tucci+Veneziani**.

Gli elementi di tipo textarea si comportano esattamente nello stesso modo. Adesso vengono introdotti nella form altri elementi: un menù a tendina select e un gruppo di radiobutton.

```
< form action="/cgi-bin/prova" method="..." >
< p align="left">
Inserisci nome e cognome< br>
< input type="text" name="nomeCognome" value="Matteo Tucci Veneziani">
< p align="left">
Seleziona il tuo paese< br>
< select name="paese" size="1">
< option value="United" selected >United States
< option value="Italy">Italy
< option value="Canada">Canada
< /select>
< p align="left">Sesso< br>
< input type="radio" name="sesso" value="maschio" checked>Maschile< br>
< input type="radio" name="sesso" value="femmina">Femminile< br>
< input type="submit" value="Invia">
< /p>
< /form>
```

Il risultato visualizzabile sarà:



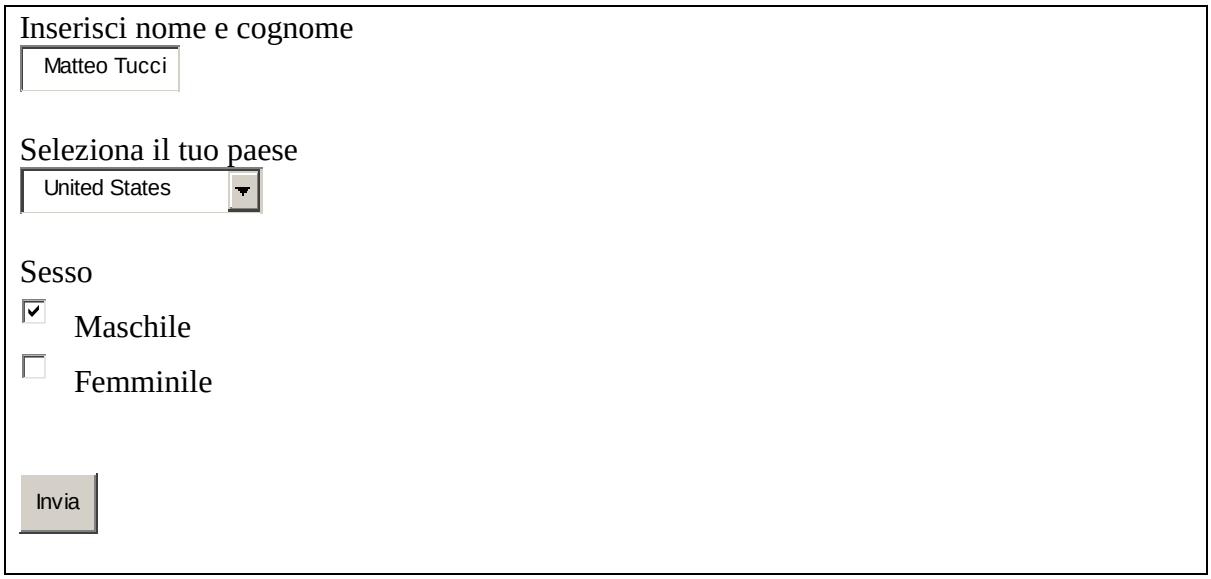
Lasciando i valori di default la stringa che verrà inviata allo script sarà **nomeCognome=Matteo+Tucci+Veneziani&paese=United&sesso=maschio**.

Nel prossimo esempio introduciamo un elemento checkbox. Questo tipo di elemento è fonte di alcuni problemi infatti se viene selezionato allora nella query_string appare una coppia name=value del tipo name=on ma altrimenti nei dati passati al programma non c'è traccia di lui. Quindi nel caso in cui in una form sia presente uno di questi elementi il numero di coppie da separare all'interno della stringa URL encoded è variabile. il codice dell' esempio è il seguente:

```
< form  >
< p align="left">
Inserisci nome e cognome< br>
<input type="text" name="nomeCognome" value="Matteo Tucci Veneziani"><br>
< p align="left">
Seleziona il tuo paese< br>
< select name="paese" size="1">
< option value="United" selected >United States
< option value="Italy">Italy
< option value="Canada">Canada
< /select>
< p align="left">Sesso
< input type="checkbox" name="maschio" checked >Maschile< br>
< input type="checkbox" name="femmina">Femminile< br>
```

```
< input type="submit" value="Invia">
< /p>
< /form>
```

Ciò che si visualizza con un browser è:



I checkbox a differenza dei radio button non sono mutamente esclusivi cioè possono essere entrambi selezionati. Quindi è chiaro che l'esempio non ha senso ma è soltanto un modo per illustrare il passaggio degli argomenti. Lasciando i valori di default la stringa che verra' inviata allo script sarà **nomeCognome=Matteo+Tucci+Veneziani&paese=United&maschio=on**.
Selezionando solo femmina invece non ci sarà traccia del primo campo (maschio): **nomeCognome=Matteo+Tucci+Veneziani&paese=United&femmina=on**.
Se i due campi vengono entrambi attivati il contenuto della form sarà **nomeCognome=Matteo+Tucci+Veneziani&paese=United&maschio=on&femmina=on** mentre se nessuno dei due è selezionato la stringa inviata sarà **nomeCognome=Matteo+Tucci+Veneziani&paese=United**.

Per verificare il risultato di una form, e quindi anche di questi esempi, è necessario modificare il codice HTML indicando per il campo *action* il nome di un'applicazione inesistente e per il campo *method* il valore "*get*" . Quindi cliccando sul bottone submit il server risponderà con un errore 404 e sarà comunque possibile vedere sulla barra degli indirizzi l'URL completo del suffisso. Nel nostro caso dato che questo sito è realizzato con dei frame la barra degli indirizzi non cambia allora è necessario visualizzare la pagina a tutto schermo senza l'indice.

Programmare CGI in C/C++

Nei due sottoparagrafi seguenti sono riportati due programmi CGI scritti in C++ utilizzando l'ambiente di sviluppo GCC. Prima di tutto è bene chiarire un punto importante: non è possibile compilare un programma CGI su una macchina e poi fare l'upload dell' eseguibile sul *Server* perchè è molto probabile che questo non funzioni. Infatti a meno che sulla prima macchina non ci sia lo stesso sistema del ISP sul quale dovrà poi girare l'applicazione questa non funzionerà. Quindi se un programma CGI deve girare su un calcolatore differente da quello sul quale è stato sviluppato è necessario effettuare la sua compilazione su questa macchina. Inoltre affinchè un programma CGI sia invocabile dal server web è necessario che questo possa eseguirlo. Quindi la directory nella quale risiedono i programmi CGI deve essere eseguibile dal server web e così pure il file risultato della compilazione. Infatti in un sistema UNIX il server web, come già anticipato, è visto come un utente indipendente con un nome e un gruppo. L' "utente" server appartiene, per il resto degli utenti del sistema, al gruppo "other" identificato dalla lettera "o". Allora è necessario eseguire le seguenti istruzioni:

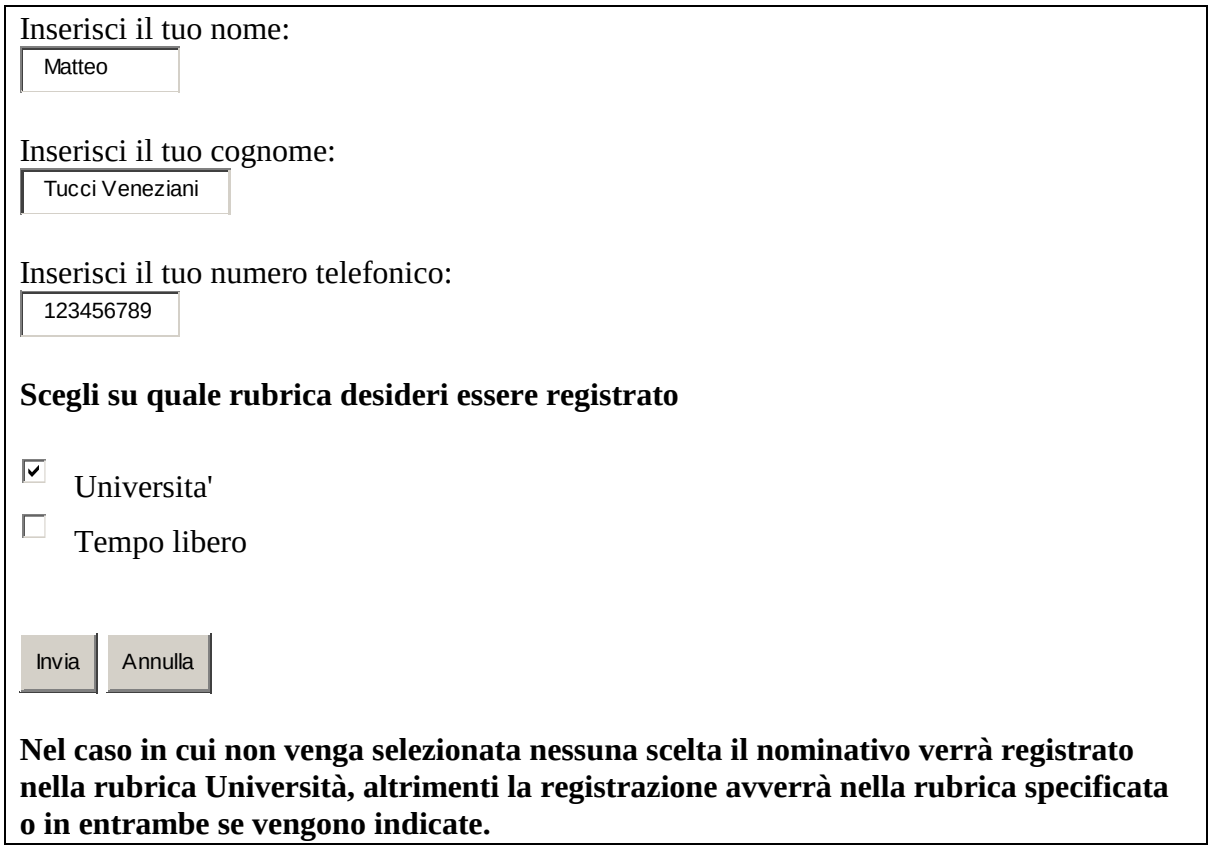
```
chmod o+x directory
chmod o+x file.cgi
```

Inoltre quando un programma viene eseguito questo gode degli stessi permessi dell'utente che lo ha invocato. Allora se l'applicazione deve leggere o scrivere file è necessario che il server web abbia i permessi di farlo.

I programmi riportati in questo paragrafo sono soltanto dei rozzi esempi sicuramente criticabili dal punto di vista della programmazione e delle prestazione. Inoltre, sebbene siano stati accuratamente testati, l'autore declina ogni responsabilità per danni a hardware ed a software derivanti dall'uso di questi. Infatti è bene ricordare che le applicazioni CGI sono un buon punto di attacco per i pirati informatici. Comunque costituiscono dei buoni modelli funzionanti per illustrare l'interfaccia CGI. Questi programmi servono rispettivamente a gestire una rubrica telefonica e una bacheca di annunci elettronica. Il codice e l'idea è molto simile. Entrambe le applicazioni non restituiscono al server un documento creato appositamente ma utilizzano la direttiva *location*. Quando ricevono una richiesta decodificano la form e scrivono i dati così ottenuti in appositi file html secondo dei formati stabiliti. La ridirezione mediante *location* avviene, nel caso della rubrica, verso il file html che contiene la form mentre, per gli annunci verso il file che contiene la bacheca . Quando un utente effettua la registrazione sulla rubrica o emette un annuncio sulla bacheca, riempiendo i campi della form e cliccando sul bottone *invia*, la rispettiva applicazione CGI estrae i dati immessi e va a scrivere sul file html dove sono conservati le informazioni per le due applicazioni. L'utente, grazie alla direttiva location, nel primo caso, continua a visualizzare il documento contenente la form, nel secondo visualizza il file contenente la bacheca.

Esempio: Rubrica Telefonica

Il programma rubrica telefonica è stato realizzato per servire richieste di tipo GET e quindi riceve la query_string mediante la variabile di ambiente omonima. La scelta di get per questo tipo di problema è appropriata in quanto i byte da trasferire sono esigui. I dati di ingresso sono il risultato di una form del tipo:



Tale modulo è costituito da 5 campi: 3 campi *text* e 2 *chekbox*. Il codice HTML per questo tipo di form è:

```
< form action="/cgi-bin/rubrica" method="GET">
< p align="left">
Inserisci il tuo nome:< br>
< input type="text" name="nome" maxlength="20">
< p align="left">
Inserisci il tuo cognome:< br>
< input type="text" name="cognome" maxlength="20">
< p align="left">
Inserisci il tuo numero telefonico:< br>
< input type="text" name="tel" maxlength="20" >
```

```
< p align="left">
<b>Scegli su  quale rubrica desideri essere registrato</b>< br>< br>< br>
< input type="checkbox" name="uni">Universita'< br>
< input type="checkbox" name="lib">Tempo libero < br>< br>< br>
< input type="submit" value="Invia">
< input type="reset" value="Annulla">
< /form>
```

Quindi la query_string generata da questa form con i dati dell'esempio è **nome=Matteo&cognome=Tucci+Veneziani&tel=123456789&uni=on**. Quando l'utente preme il bottone invia, il programma CGI, lanciato dal server, decodifica la form per ottenere al piu' 5 stringhe distinte. Poi in base a quale checkbox e' stato selezionato, scrive sul file universita.html o libero.html, che si trovano all'interno del sotto albero del file system la cui radice e' documentRoot, i dati estratti dalla stringa di ingresso. Per consentire all'utente di visualizzare la rubrica e' possibile predisporre nella stessa pagina della form un collegamento ai due documenti html. Chiaramente e' necessario fornire i permessi di lettura e scrittura dei due file all'intero gruppo "other". Adesso verrà analizzata la parte del programma che effettua la decodifica della form.

Questa parte è stata realizzata interamente all'esterno del main ed è indipendente dal tipo di form utilizzata e dal numero di campi che questa possiede. Questo ne e' il codice:

```
inline int
hexdigit(register int c)
{
        c &= 0x7f;

        if (c >= 'a' && c <= 'f')
                return (c - 'a' + 10);

        if (c >= 'A' && c <= 'F')
                return (c - 'A' + 10);

        return (c - '0');
}

void sostituisci(char* s,int i,int n){
  int primo,secondo;
   primo=hexdigit(s[i+1]);
   secondo=hexdigit(s[i+2]);
   primo<<=4;
   s[i]=primo|secondo;
   for(int j=i+1;j< n-1;j++)
      s[j]=s[j+2];
}
```

```
void parse(char* s){
  int n=strlen(s);
  for(int i=0;i< n;i++)
    if(s[i]=='+')
      if(i!=n-1)
        s[i]=' ';
      else
        s[i]=0;
    else
    if(s[i]=='%')
      sostituisci(s,i,n);
}

void estrai(char* query,char** parole,int nform){
  int n=strlen(query);
  int nparole=0;
  int a=0;
  int i=0;
  for( i=0;i< n;i++){
    if(query[i]=='&' && nparole< nform-1 ){
      parole[nparole]=new char[i+1-a];
      a=i+1;
      nparole++;
    }
    else
      if(nparole==nform-1){
        parole[nparole]=new char[n-i+1];
        break;
      }
  }
  i=0;
  for(int in=0; in< nform; in++){
    a=0;
    while(query[i]!='&' && i< n ){
           parole[in][a]=query[i];
           i++;
           a++;
    }
    parole[in][a]='\0';
    i++;
  }
}

void decodifica(char* query, char** parole, int& n){
    n=1;
    char* c;
    c=strchr(query,'&');
    while(c!=0){
      n++;
      c=strchr((c+1),'&');
    }
    estrai(query,parole,n);
    for(int i=0;i< n;i++)
       parse(parole[i]);
}
```

```
int  main(){

 cout<<"Location: path del documento su cui risiede la form"<<'\n'<<'\n';
 char* query= getenv("QUERY_STRING");

 char* parole[5] ;
 int n;
 decodifica(query,parole,n);
  ...
}
```

All'interno del main la variabile *parole* costituisce un vettore di puntatori a carattere. Questi puntatori verranno utilizzati per indirizzare le coppie name=value, che in questo caso sono al piu' 5, estratte dalla query_string. La funzione *decodifica* calcola, come prima cosa, il numero di coppie presenti nella stringa di ingresso contando i caratteri "&". Successivamente chiama la funzione *estrai* la quale ottiene le stringhe name=value dalla query_string assegnando l'indirizzo del loro primo elemento ai puntatori parole[*i*]. Queste vengono poi analizzate con la funzione *parse* che effettua l'URL decoding. Quindi alla fine dell'esecuzione della funzione *decodifica* ogni elemento del vettore *parole* rappresenta una stringa del tipo name=value. La funzione estrai è la funzione piu' complicata. Questa scorre una prima volta la stringa query per tutta la sua lunghezza e quando trova il carattere "&" alloca tanti byte quanti sono i caratteri che ha contato dall'inizio o dal precedente. Quando sono già stati trovati 4 caratteri "&" è inutile cercare il quinto perchè questo non esiste. Quindi quando la funzione analizza un carattere e il numero dei vettori allocati è 4 allora alloca subito il quinto di dimensioni pari al numero dei caratteri restanti. La funzione effettua poi un secondo passaggio nel quale comincia a copiare i caratteri in una stringa parole[*i*] fino a che non incontra un "&" oppure l'ultimo carattere di *query*. Nel primo caso, assegna all'ultimo byte di parole[*i*] un '\0' e passa a riempire i byte allocati al puntatore parole[*i+1*] nel secondo assegna il carattere '\0' all'ultimo byte della stringa e termina. La funzione *parse* chiamata all'interno di *decodifica* effettua la ricerca di caratteri "+" o "%". Quando trova un "+" questo viene sostituito con un carattere spazio mentre nell'altro caso utilizza la funzione *sostituisci* per sostituire il carattere "%" e i due successivi con quello da quest'ultimi rappresentato.

Adesso verra' trattata la parte del programma che effettua l'elaborazione dei dati. Questa parte è stata realizzata all'interno del corpo del *main*, sebbene fosse comunque possibile incapsularla all'interno di una funzione distinta, e consiste nella scrittura dei dati in alcuni file. Ecco il codice:

```
 int main(){

    ...

    fstream fi;
    if(n==3){
      fi.open("/www/universita.html",ios::in|ios::out);
      fi.seekp(-26,ios::end);
      fi<<"< tr>< td>"<<&parole[0][5]<<"< td>"<<&parole[1][8];
      fi<<"< td>"<<&parole[2][4]<<"< /tr>\n\n";
      fi<<"< /table>\n< /body>\n< /html>\n";
      fi.close();
    }else
      if(n==4){
        if(parole[3][0]=='u')
          fi.open("/wwww/universita.html",ios::in|ios::out);
        else
          fi.open("/www/libero.html",ios::in|ios::out);
        fi.seekp(-26,ios::end);
        fi<<"< tr>< td>"<<&parole[0][5]<<"< td>"<<&parole[1][8];
        fi<<"< td>"<<&parole[2][4]<<"< /tr>\n\n";
        fi<<"< /table>\n< /body>\n< /html>\n";
        fi.close();
      }
      else{
        fi.open("/www/universita.html",ios::in|ios::out);
        fi.seekp(-26,ios::end);
        fi<<"< tr>< td>"<<&parole[0][5]<<"< td>"<<&parole[1][8];
        fi<<"< td>"<<&parole[2][4]<<"< /tr>\n\n";
        fi<<"< /table>\n< /body>\n< /html>\n";
        fi.close();

        fi.open("/www/libero.html",ios::in|ios::out);
        fi.seekp(-26,ios::end);
        fi<<"< tr>< td>"<<&parole[0][5]<<"< td>"<<&parole[1][8];
        fi<<"< td>"<<&parole[2][4]<<"< /tr>\n\n";
        fi<<"< /table>\n< /body>\n< /html>\n";
        fi.close();
      }
     return 0;
}
```

A questo punto non rimane che scrivere nel file che conserva la rubrica. Tale file viene determinato sulla base del checkbox selezionato come indicato all'interno della form. Il programma si aspetta che i due file siano strutturati in un certo modo:

```
< html>< head>< title>Rubrica di Matteo< /title>
< /head>
< body>
< h1 style='color:red;'>La Rubrica di MatteoWeb< /h1>
< table align='center' border='2' cellpadding='3'>
< tr>< th>Nome< th>Cognome< th>Tel< /tr>

< /table>
< /body>
< /html>
```

Ogni volta che l'applicazione deve scrivere i dati di un utente, apre uno dei file in lettura-scrittura. Poi mediante la funzione *seekp* posiziona il puntatore alla posizione corrente per la scrittura, alla fine del file e risale di 26 caratteri effettuando la successiva scrittura a partire dalla riga vuota. I file contenenti la rubrica avranno un aspetto del genere:

La Rubrica di MatteoWeb

Nome	**Cognome**	**Tel**
Paolo	Rossi	797986599
Matteo	Tucci Veneziani	123456789

In questo programma non si effettuano controlli sui dati inseriti dall'utente. Per esempio potrebbe essere digitata una stringa nel campo dedicato al numero di telefono e il programma CGI continuerebbe a funzionare come se niente fosse. Solitamente tali controlli sulla correttezza dei dati vengono effettuati sul lato client in modo da non impegnare ulteriormente il server. Questo può essere effettuato utilizzando script locali scritti in Javascript o altri linguaggi. In questo caso dovremo inserire nello header del documento tra i tag < script> il seguente codice:

```
< script language="javascript">
< !--
function chiama(){
    document.forms[0].method="get";
    document.forms[0].action="/cgi-bin/rubrica";
    var nome=document.forms[0].elements[0].value.length;
    var cognome=document.forms[0].elements[1].value.length;
    var tel=document.forms[0].elements[2].value.length;

  if(nome!=0 && cognome!=0 && tel!=0){
        for(var i=0; i'9'){
                   alert("Numero non corretto");
                    return;
                   }
         document.forms[0].submit();
         }
   else
       if(nome==0){
         alert("Inserisci il nome");
         return 0;
         }
         else
           if(cognome==0){
              alert("Inserisci il cognome");
              return 0;
              }
              else{
                alert("Inserisci il telefono");
                return 0;
                }
}
//-->
< /script>
```

Inoltre è necessario sostituire il bottone *sumbit* con un semplice campo *button*:

```
< input type="button" value="invia" onClick="chiama()">
```

In questo modo quando il bottone viene premuto, i dati inseriti nella FORM vengono inviati al server soltanto se tutti i campi sono stati compilati e il formato del numero telefonico è corretto. Questo è un esempio di interazione tra elaborazione locale e remota nel web.

Esempio: Bacheca di annunci

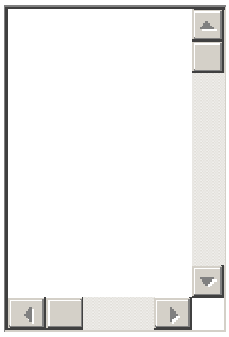
Il programma *bacheca* funziona in modo molto simile all'esempio precedente. E' stato realizzato per servire richieste GET e POST e quindi riceve la stringa query_string sia mediante la variabile di ambiente omonima sia mediante lo standard input. Per determinare quale dei due meccanismi utilizzare per ottenere i dati di ingresso l'applicazione fa uso della variabile di ambiente REQUEST_METHOD. Per questo tipo di problema è più adatto il metodo di trasmissione POST perchè in grado di trasportare un numero maggiore di informazioni. Il programma *bacheca* accetta le richieste da una FORM del tipo:

Inserisci il tuo nome

Inserisci il tuo cognome

Inserisci il numero di telefono

Inserisci il tuo annuncio



Definisci il tipo di annuncio che intendi fare

- (•) Vendo
- ( ) Compro

Invia Annulla

Tale modulo è costituito da 6 campi: 3 text, 1 textarea e 2 radio-button. Il codice HTML per questo tipo di FORM è:

```
< form action="/cgi-bin/bacheca" method="GET | POST">
< p align="left">
Inserisci il tuo nome < br>
< input type="text" name="nome" maxlength="20" >
< p align="left">
inserisci il tuo cognome< br>
< input type="text" name="cognome" maxlength="20" >
< p align="left">
inserisci il numero di telefono< br>
< input type="text" name="tel" maxlength="20" >
< p align="left">
Inserisci il tuo annuncio< br>
< textarea name="annu" cols="16" rows="10" wrap >< /textarea>
< p align="left">
< b>Definisci il tipo di annuncio che intendi fare< /b>< br>
< input type="radio" name="tipo" value="vendo" checked > Vendo < br>
< input type="radio" name="tipo" value="compro"> Compro< br>< br>
< input type="submit" value="Invia">
< input type="reset" value="Annulla">
< /p>
< /form>
```

Quindi la query_string generata da questa FORM sarà del tipo **nome=Matteo&cognome=Tucci+Veneziani&tel=123456789&annu=vendo+vw+Golf+1.4+16v+prezzo%3A+10000+euro&tipo=vendo**. La parte del programma che effettua la decodifica della form è la stessa dell'esempio precedente anche se in questo caso il problema è più semplice visto che il numero di coppie name=value all'interno della query_string è costante. Rispetto al programma *rubrica* quello che cambia è costituito dal modo nel quale vengono acquisiti i dati:

```
int  main(){
  char* met=getenv("REQUEST_METHOD");
  char* query;
  if(strcasecmp(met,"post")==0){
    char* len=getenv("CONTENT_LENGTH");
    int lun=atoi(len)+1;
    query=new char[lun];
    cin.get(query,lun-1);
  }
 else
  query=getenv("QUERY_STRING");
  ...
}
```

Se il valore della variabile REQUEST_METHOD è la stringa "post" i dati vengono letti dallo standard input mentre nel caso contrario sono acquisiti mendiante la variabile QUERY_STRING. Anche la parte che costituisce l'elaborazione è abbastanza simile al programma precedente :

```
int main(){
    ...

    fstream fi;
    if(parole[4][5]=='v'){
       fi.open("/www/vendo.html",ios::in|ios::out);
       cout<<"Location: /www/vendo.html"<<'\n'<<'\n';
       }
    else{
       fi.open("/www/compro.html",ios::in|ios::out);
       cout<<"Location: /www/compro.html"<<'\n'<<'\n';
    }
    fi.seekp(-31,ios::end);
    fi<<"< tr>< th>< li>Annuncio scritto da "<<&parole[0][5];
    fi<<" "<<&parole[1][8]<<" Tel: "<<&parole[2][4]<<"</li></th></tr>\n";
    fi<<"< tr>< td>"<<&parole[3][5]<<"< /td>< /tr>\n";
    fi<<"< /table>\n< /ul>\n< /body>\n< /html>\n";
    fi.close();
    return 0;
}
```

Il programma, una volta decodificata la form, apre il file vendo.html o compro.html, entrambi disponibili dalla rete, in base al valore del campo *tipo*. L'applicazione si aspetta che questi file abbiano una certa forma e siano del tipo:

```
< html>< head>< title>Annunci di MatteoWeb< /title>
< /head>
< body>
< h1 style='color:red;'>La bacheca di MatteoWeb:< /h1>
< ul>
< table border='0' align='center' width='80%' cellspacing='7' >

< /table>
< /ul>
< /body>
< /html>
```

Ogni volta che il programma deve scrivere i dati di un utente, apre uno dei file in lettura-scrittura. Poi mediante la funzione *seekp* posiziona il puntatore alla posizione corrente per la scrittura, alla fine del file e risale di 31 caratteri effettuando la successiva scrittura a partire dalla riga vuota. I file contenenti la bacheca avranno un aspetto del genere:

La bacheca di MatteoWeb:

- **Annuncio scritto da Matteo Tucci Veneziani Tel: 123456789**

  vendo vw Golf 1.4 16v prezzo 10000 euro

Gli annunci vengono scritti all'interno di due righe di una tabella per gestire automaticamente il ritorno carrello e inoltre per avere uno spazio verticale costante. Anche per questo caso può essere realizzato un controllo locale dei dati mediante uno script in javascript.

APPENDICE A: URL

URL è l'acronimo di ***Uniform Resource Locator*** ovvero: Identificatore Uniforme delle Risorse. Esso costituisce l'estensione del concetto di *filename* o di path per la rete. Mediante un Url si può riferire un file che si trova all'interno della memoria di massa di un calcolatore connesso alla rete. Si tratta di una stringa del tipo:

```
protocollo://nome_calcolatore:porta/path_virtuale
```

Il campo *protocollo* indica quale deve essere il protocollo per ottenere il documento. Il secondo campo: *nome_calcolatore* rappresenta il nome di dominio della macchina contenente il file richiesto e può essere costituito anche dall'indirizzo IP nella notazione Dot Notation. La parte dell'URL chiamata *porta* indica il numero di porta che identifica il processo server incaricato di gestire il protocollo specificato sulla macchina individuata dal nome di dominio. Tale campo è facoltativo e se omesso viene utilizzato il numero di porta standard per quella applicazione. L'ultimo campo indicato come *path virtuale* si riferisce ad un "documento" presente sul calcolatore remoto. Questa è la parte dell'URL che viene inserita, ad esempio, nella richiesta HTTP effettuata al server.
La stringa URL seguente rappresenta un esempio di HTTP URL:
http://digilander.iol.it/MatteoVale/index.html
Il protocollo, chiaramente, è http cioè il protocollo di trasporto degli ipertesti. La stringa *digilander.iol.it* è un dominio di terzo livello ed individua una macchina fisica. Come nella maggior parte dei casi la porta non viene indicata inquanto si presuppone che nel calcolatore indirizzato il server web giri sulla porta 80 che per il web è quella di default. Il path virtuale che viene spedito al server servirà ad individuare un documento all'interno del calcolatore remoto. La stringa seguente rappresenta un esempio di FTP URL:

```
ftp://ftp.esempio.com/public/prova.txt
```

Anche in questo caso il numero di porta non viene riportato in quanto il server ftp sulla macchina ftp.esempio.com è in esecuzione sulla porta 21 che è la porta di default per questo tipo di applicazione. Ci sono moltissimi altri tipi di URL tra cui Gopher URL , News URL e mail URL:

```
mail:matucciv@tin.it
```

La sintassi dello standard URL è molto complicata e permette soltanto un piccolo insieme dei caratteri ASCII, circa 60. Comunque questa definisce un modo per codificare anche i caratteri non ammessi: "%" seguito dalle due cifre esadecimali della codifica ASCII. Anche lo spazio non può essere inserito in una stringa URL e viene sostituito dal carattere "+". Per il programmatore di applicazioni CGI questo non è un grosso problema infatti basta che sia cosciente del fatto che i parametri vengono codificati secondo lo standard URL e che quindi una volta ottenuti necessitano di essere decodificati.

APPENDICE B: FORM HTML

Il comando necessario per creare un modulo è < FORM>; tutto ciò che segue questo tag e che precede il comando di chiusura < /FORM> fa parte del modulo. < FORM> richiede sempre due attributi: il primo è ACTION= seguito dall'URL, a cui il modulo invierà i dati racchiuso tra virgolette. Il secondo attributo è METHOD= e indica il metodo utilizzato nella trasmissione dei dati al server. I vari campi del modulo vengono inseriti tramite il comando < INPUT>. Quest'ultimo tag può avere molti attributi:

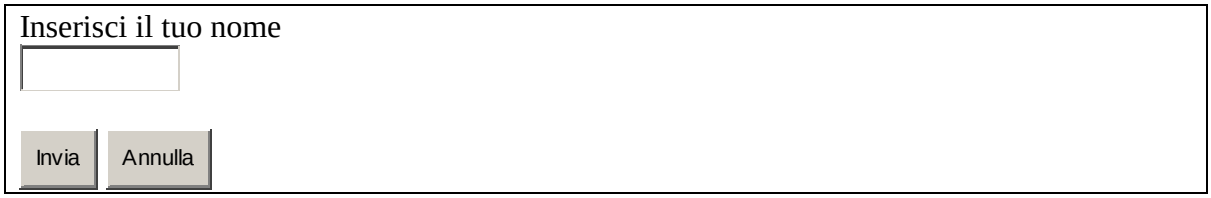
```
< input type="..." name="stringa" [value="stringa"] ...>
```

L'attributo TYPE definisce il tipo di campo creato mentre il valore di NAME costituisce il nome dell'elemento inserito. VALUE viene spesso utilizzato per assegnare dei valori di default.
Ci sono molti elementi di input che possono essere inseriti in una FORM. Una delle form più semplici è quella costituita da un unico elemento di testo e due bottoni *submit* e *reset*. Ecco quali sono i tag necessari per realizzarla:

```
< Form action="..." method="get | post">
< p class="para">
Inserisci il tuo nome< br>
< input type="text" name="nome">< br>< br>
< input type="submit" value="Invia">
```

```
< input type="reset" value="Annulla">
< /p>
< /form>
```

Il risultato visualizzabile sullo schermo è il seguente:



Per il campo *text* è possibile utilizzare l'attributo SIZE per definire le dimensioni dell'elemento di testo, e l'attributo MAXLENGTH per stabilire il massimo numero di caratteri. Inoltre VALUE può essere utilizzato per definire un valore di default. Per quanto riguarda i due bottoni, il campo VALUE definisce la stringa che appare sopra gli elementi grafici. La pressione del pulsante *Invia* comporta l'invio dell'informazioni al server mentre *Annulla* annulla i dati inseriti fino a quel momento. Un elemento molto simile a *text* è il campo *password* che permette di creare degli elementi di testo nei quali i caratteri digitati appaiono sottoforma di asterischi. Sebbene la stringa digitata non sia leggibile questo non comporta la presenza di alcun supporto alla sicurezza della trasmissione. Due importanti elementi di un modulo elettronico sono i *radiobutton* e i *checkbox*. I primi servono a costringere l'utente a fare delle scelte mutuamente esclusive. Per realizzarli è sufficiente assegnare a TYPE il valore *radio*. I radiobutton che costituiscono lo stesso gruppo devono avere lo stesso valore dell'attributo NAME e si distinguono fra loro per il campo VALUE. Ecco un esempio: una serie di comandi di questo tipo

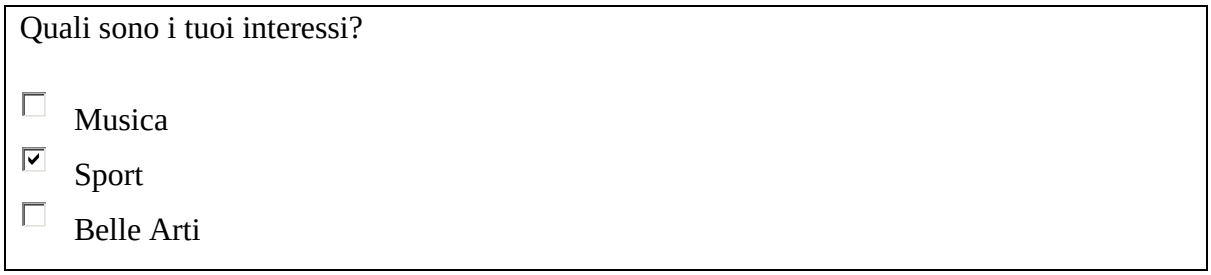
```
< p align="left">
Sesso:< br>
< input type="radio" name="sesso" value="maschio" checked>Uomo< br>
< input type="radio" name="sesso" value="femmina">Donna
< /p>
```

Produrra' sullo schermo il seguente risultato:

L'attributo CHECKED permette di realizzare una scelta di default. L'uso dei *checkbox* permette di creare delle caselle di tipo si/no:

```
< p align="left">
Quali sono i tuoi interessi?
< input type="checkbox" name="musica">Musica< br>
< input type="checkbox" name="sport" checked>Sport< br>
< input type="checkbox" name="arte">Belle Arti< br>
< /p>
```

Questo e' il risultato sul monitor:



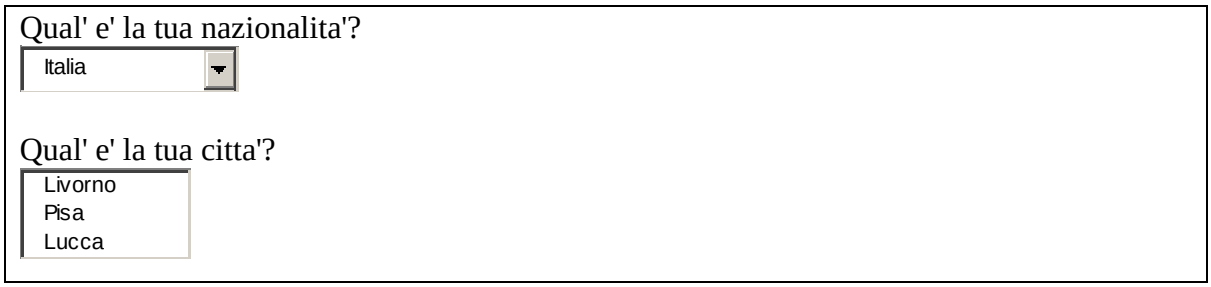
La differenza rispetto ai *radiobutton* è che in questo caso l'utente può operare più di una scelta.

Un altro elemento di ingresso, che consente di effettuare delle scelte, può essere inserito mediante il comando SELECT il quale da luogo a delle finestre contenenti delle opzioni tra cui scegliere. Si possono realizzare delle finestre contenenti tutte le opzioni oppure dei menù a tendina. L'utilizzo dell'attributo MULTIPLE dà la possibilità all'utente di effettuare più di una scelta:

Ecco un esempio

```
< p align="left">
Qual'e' la  tua nazionalita'?< br>
< select name="paese" size="1">
< option value="ita">Italia
< option  value="fra">Francia
< option value="usa">Stati Uniti
< /select>
< p align=left>
Qual'e' la tua citta'?< br>
< select name="town" size="3">
< option value="li">Livorno
< option value="pi">Pisa
< option value="lu">Lucca
< /select>
< /p>
```

Il risultato dell'interpretazione di questo codice da parte del browser sarà del tipo:



Un'altro campo molto usato è la *textarea*. Per inserire questo elemento in una FORM è necessario utilizzare il comando TEXTAREA. Questo tag ha l'attributo NAME e due ulteriori attributi per definire le sue dimensioni: ROWS e COLS. Affinchèil testo inserito dall'utente scorra automaticamente nella finestra senza che le stringhe debbano essere spezzate manualmente può essere utilizzato l'attributo WRAP:

```
< form>
< p align="left">
Inserisci i tui suggerimenti:< br>
< textarea name="consigli" rows="10" cols="30" wrap>< /textarea>
< /form>
```

Il risultato di questo codice sarà:

Inizio modulo

Inserisci i tui suggerimenti:

APPENDICE C: HTTP

La navigazione del WEB, come la maggior parte delle applicazioni di rete, segue il paradigma client-server. L'interazione tra il processo client, che in questo caso viene detto browser, e il processo server avviene mediante il protocollo HTTP. Questo è un protocollo di livello applicazione della pila TCP/IP e quindi funziona al disopra del protocollo TCP. Mentre quest'ultimo è *session-oriented*, HTTP non ha idea di che cosa sia una sessione. Una volta che una richiesta è stata spedita e la relativa risposta è stata ricevuta i due processi remoti si scordano l'uno dell'altro e per questo si parla di protocollo *state-less*. Per comunicare attraverso il web con il protocollo HTTP occorre innanzi tutto che il client attivi una connessione con il server web indicato nella parte iniziale dell'URL introdotta dall'utente. Successivamente, utilizzando la connessione attivata, il client effettua la richiesta del documento specificato. I passi successivi consistono nella risposta del server e nella chiusura della connessione.

Browser Request

La richiesta è costituita dall'intestazione e dal corpo di un pacchetto HTTP. Il corpo anche detto payload può essere vuoto mentre l'intestazione ha una forma standard:

```
Method URL Protocol/version
Other Information
```

Method:GET, HEAD, POST, PUT, DELETE. Il primo valore per method è certamente il più comune e viene utilizzato dal client per chiedere che gli venga inviato un certo documento. Head invece richiede soltanto l'intestazione della pagina e non i dati. Post viene utilizzato per trasmettere le informazioni risultato di una FORM al server. Le ultime due possibilità servono rispettivamente per inviare e cancellare un file sulla memoria di massa di un *Server* e per questo sono implementate molto raramente.

Other Information La richiesta viene rifinita dal browser mediante l'aggiunta di più comandi. Tipicamente il client aggiunge il comando *Accept* con il quale indica il tipo dei dati che può maneggiare. Il browser può anche aggiungere il nome dell'applicazione utilizzando il comando *User-Agent*. Un campo molto importante è *If-Modified-Since*. Questo ha come argomento una data e permette di specificare al server di inviare il documento solo se è stato modificato successivamente a questa.
E' possibile anche inviare al server la preferenza sulla lingua nella quale sono scritti i documenti nel caso in cui sia presente un'alternativa.
Se l'URL digitato dall'utente nella barra degli indirizzi e' *http://myserver/ricerca/rmi.html* il browser risolvera', con il servizio DNS, il nome *myserver* in un indirizzo IP e aprirà una connessione verso tale indirizzo e la porta 80. Una volta stabilito un collegamento TCP il client invierà sul pipe il pacchetto HTTP un byte per volta:

```
----------------------------
GET /ricerca/rmi.html HTTP/1.0
User-Aggent: Mozzilla 4.0
Accept *.*
Language: ...

----------------------------
```

All'interno dell'intestazione non c'e', quindi traccia dell'indirizzo del server. Una volta che il client ha ricevuto il documento desiderato questo chiude la connessione e se è necessario acquisire un ulteriore documento dal server, deve aprirne una nuova. L'ultima versione del protocollo, HTTP/1.1, da questo punto di vista e' piu' efficiente in quanto mantiene aperta la connessione con il server per un certo intervallo di tempo.

Server Response

La risposta del server è costituita dall'intestazione e dal corpo di un pacchetto HTTP. Il corpo può eventualmente essere vuoto nel caso in cui la richiesta non abbia buon fine. L'intera risposta ha una forma del tipo:

```
-----------
Status Line
Server: ...
MIME-Version: ...
MetaData
-----------

Data
```

Status-line

Il primo campo dell'intestazione comunica al browser l'esito della sua richiesta per mezzo di un codice. La status line ha una forma del genere:

```
HTTP/1.0 200 OK
```

In questo caso la richiesta del browser ha avuto successo e il documento richiesto segue l'intestazione. Un web server puo' inviare al client molti codici. Alcuni di questi vengono mostrati dal browser ma raramente gli utenti conoscono il loro significato. Questi sono raggruppati logicamente secondo il loro significato: i codici da 200-299 indicano che la richiesta ha avuto successo, da 300-399 che la pagina richiesta può essere stata spostata. I codici all'interno del range 400-499 mostrano un errore del client e quelli da 500-599 un

errore sul server. La versione del protocollo usata nella risposta deve essere la medesima con la quale è stata fatta la richiesta.

MetaData

Queste sono delle informazioni incluse dal server che riguardano i dati spediti al client. Le più comuni sono riportate di seguito: *Content-type:*Tipo MIME delle informazioni
*Content-length:*Numero di byte inviati
*Last-Modified:*Data dell'ultima modifica
*Content-Language:*Linguaggio naturale in cui è scritto il documento.

Di seguito è riportata una tipica risposta nella quale il file è stato trovato e spedito al client:

```
HTTP/1.0 200 OK
Server: Apache/1.3
MIME-Version: 1.0
Content-Type: text/plain
Content-length: 9

Ciao Ciao
```

Nella comunicazione non vi è traccia dell'indirizzo del client perchè il server utilizza la stessa connessione aperta dal browser e ottiene il suo indirizzo quando accetta l'apertura di questa.

APPENDICE D: Apache e CGI

All'interno di una richiesta HTTP il client invia al server il path virtuale del documento richiesto. Questo per poter fornire una risposta al browser deve risolvere il path virtuale in un path reale relativo al suo file system. Apache è uno dei server web più diffusi al mondo ed è disponibile gratuitamente sulla rete e in molte distribuzioni di Linux. La configurazione di Apache avviene mediante la modifica di alcuni file. Nella grande maggioranza dei casi le direttive vengono concentrate tutte in uno solo: *httpd.conf*. I file di configurazione vengono posizionati all'interno della memoria di massa in modo dipendente dal tipo di sistema ma solitamente si trovano all'interno dell'albero la cui radice è /etc. Apache è un server concorrente cioè può erogare contemporaneamente lo stesso servizio a più client senza che questi debbano necessariamente mettersi in coda. Quando l'applicazione viene lanciata vengono creati anche un certo numero di processi figli e ogni volta che il server, cioè il processo padre, riceve una richiesta la fa gestire a uno dei suoi figli e ne crea uno nuovo. Attraverso i file di configurazione è possibile gestire la creazione dei processi stabilendo quanti ne debbono essere creati, quanti debbono essere al più quelli liberi e il loro numero minimo. Inoltre quando viene lanciato un processo, questo ha gli stessi privilegi dell'utente che lo ha invocato. Il web server viene solitamente lanciato da root e quindi il processo padre gode di ogni permesso. I figli ereditano i permessi dal padre di default ma questo è un comportamento che dà dei problemi di sicurezza e quindi è necessario farli girare come se fossero lanciati da un utente con pochi privilegi. Questo utente, il cui nome può essere determinato mediante le direttive :

```
user: apache
group: apache
```

appartiene al gruppo "other" per tutto il resto degli utenti del sistema. Quindi mediante i file di configurazione si possono specificare molte opzioni di funzionamento. Una direttiva molto importante è DocumentRoot. Questa ha come valore una directory che costituisce la radice dell'albero del file system all'interno del quale vengono conservati i documenti disponibili via web. Di default tutte i path virtuali che il server web ottiene dalle richieste vengono completati con DocumentRoot:

```
DocumentRoot: /var/www/html
```

Spesso però è necessario rendere una directory esterna al sottoalbero di documentRoot disponibile via Internet e si usa la direttiva Alias. Questa ha due campi:

```
Alias: /icons/ /var/www/icons
```

Se la parte iniziale del path virtuale coincide con la prima stringa il path reale non viene risolto con DocumentRoot ma con il secondo campo.Una direttiva molto simile a quest'ultima è ScriptAlias:

```
ScriptAlias: /cgi-bin/ /var/www/cgi-bin/
```

Se la parte iniziale del path virtuale coincide con la prima stringa allora il web server ricerca il file indicato all'interno della directory individuata dal percorso al secondo campo e lo esegue. Quindi le richieste dirette a applicazioni CGI non devono necessariamente contenere la stringa cgi-bin nell'URL ma quella che appare al primo campo della direttiva ScriptAlias.
Una volta installato il server web conviene modificare il file di configurazione come Super User in modo che i campi DocumentRoot e ScriptAlias puntino a directory presenti nella propria home. Infatti in questo modo è più comodo sviluppare applicazioni CGI perchè si hanno tutti i permessi senza dover lavorare come Root. A questo punto però è necessario che la documentRoot abbia i permessi di lettura e esecuzione abilitati per il gruppo "other" e che la cartella che contiene gli script abbia quelli di esecuzione.

Bibliografia

1. The WWW Common Gateway Interface RFC version 1.1

2. Web programming. Building Internet Application 2° ed. *Chris Bates* ed: John Willey & Sons

3. Internet e Reti di Calcolatori *Douglas Commer* ed: Addison Wesley