Appunti di Cybersecurity: Meccanismi TCP e Attacchi

Appunti dalla Lezione

30 ottobre 2025

Indice

1 Evoluzione dei Protocolli: Da TCP a UDP (HTTP/3)

Si osserva un'evoluzione nei protocolli web, come HTTP/3 (riferito in lezione come HTTP 4.0), che segna un passaggio significativo dal protocollo TCP (Transmission Control Protocol) a UDP (User Datagram Protocol).

In questo nuovo modello, è l'applicazione stessa a dover gestire l'affidabilità dei dati. Questo contrasta con l'approccio tradizionale di TCP, che gestisce nativamente l'affidabilità (controllo di sequenza, ritrasmissione dei pacchetti persi, controllo di congestione), ma la cui rigida gestione dell'ordine può creare inefficienze (es. *Head-of-Line Blocking*). L'uso di UDP permette all'applicazione (es. QUIC per HTTP/3) di implementare meccanismi di affidabilità più flessibili e ottimizzati per il web moderno.

2 Analisi del Funzionamento di TCP

Per comprendere le vulnerabilità, è essenziale analizzare come TCP gestisce la comunicazione, l'affidabilità e l'ordinamento dei dati.

2.1 Segmentazione e Riassemblaggio

Quando un'applicazione invia un flusso di dati (es. 4000 byte), TCP deve gestirlo tenendo conto dei limiti della rete sottostante. La **Maximum Transmission Unit (MTU)** definisce la dimensione massima di un pacchetto (spesso 1500 byte).

Se Peer 1 invia 4000 byte a Peer 2, TCP (o IP) segmenterà (o frammenterà) i dati:

- **Frammento 1 (F1):** byte da 0 a 1500
- **Frammento 2 (F2):** byte da 1500 a 3000
- **Frammento 3 (F3):** byte da 3000 a 4000

A livello TCP, si esegue una singola `send(4000)`. Il protocollo TCP al lato ricevente deve garantire che i dati siano riassemblati nell'ordine corretto, anche se i pacchetti arrivano in ordine errato (es. F2, F1, F3).

2.2 Handshake: Sequence (SEQ) e Acknowledgment (ACK)

Il riordinamento è possibile grazie ai numeri di sequenza (**SEQ**) e di riscontro (**ACK**), negoziati durante la fase di *challenge* (handshake).

1. **Peer 1 (Client):** Invia un pacchetto `SYN` (Synchronize) per iniziare la connessione. Questo pacchetto contiene un numero di sequenza iniziale randomico.

```
Peer 1 -> Peer 2: SYN (SEQ = 7254)
```

2. **Peer 2 (Server):** Risponde con un flag `SYN-ACK`. Invia il proprio numero di sequenza (es. 10) e imposta l'ACK al numero di sequenza del client + 1, confermando la ricezione.

```
Peer 2 -> Peer 1: SYN-ACK (SEQ = 10, ACK = 7255)
```

3. **Peer 1 (Client):** Completa l'handshake inviando un `ACK` finale.

```
Peer 1 -> Peer 2: ACK (SEQ = 7255, ACK = 11)
```

Una comunicazione TCP è quindi completamente definita non solo dalla 5-tupla (IP Sorgente, Porta Sorgente, IP Destinazione, Porta Destinazione, Protocollo) ma anche dalla relazione $SEQ \leftrightarrow ACK$ tra i due peer.

2.3 Controllo di Flusso, Buffering e Ritrasmissione

La comunicazione prosegue con la convalida dei dati inviati tramite i numeri di sequenza, che agiscono come *offset* nel flusso di byte.

Se P1 invia F1 (1500 byte) partendo da $SEQ = 7255$:

- fuori ordine: Se P2 riceve prima F2 (byte 1500-3000), non può ancora consegnare i dati all'applicazione. Inserisce F2 in uno "storage" (buffer di ricezione).
- : Quando P2 riceve F1 (byte 0-1500), riempie il buffer con i primi 1500 byte. Ora può "attaccare" F2, che era nello storage, all'offset corretto (1500).
- Cumulativo: A questo punto, P2 conferma a P1 di aver ricevuto correttamente i primi 3000 byte (F1+F2) e svuota la coda (consegnando i dati all'applicazione). L'ACK inviato sarà $SEQ_{P1} + 3000$.

Grazie all'uso degli offset (numeri di sequenza), non si possono avere duplicati (dati duplicati con lo stesso SEQ vengono scartati).

Gestione delle Perdite

Se P2 non riceve F3 entro un certo timeout (es. $>$ 10 millisecondi), non può confermare i dati oltre i 3000 byte. P2 potrebbe quindi rimandare lo stesso pacchetto `ACK` (ACK duplicato) comunicando che gli sono arrivati solo i primi due segmenti. Questo segnala a P1 di ritrasmettere F3.

Se la connessione attraversa reti con MTU diverse (es. una rete ATM) che frammentano ulteriormente i pacchetti (es. da 1000 a 500 byte), il peer può rispondere con un nuovo ACK specifico per il re-invio (*re-transmitted*) dei dati persi.

La connessione TCP è una **macchina a stati finiti** e questi meccanismi di affidabilità sono attivi finché la connessione non viene chiusa.

3 Superficie d'Attacco in TCP

3.1 Man-in-the-Middle (MITM) e Sequence Number Attack

Dato che la comunicazione è definita dai numeri SEQ e ACK, un attore malevolo (Man-in-the-Middle, MITM) che può osservare e modificare i pacchetti può sabotare la connessione.

Esempio (su rete locale):

1. P1 invia `SYN` con $SEQ = 30$.
2. Il MITM intercetta il pacchetto e lo modifica, inoltrando a P2 un `SYN` con $SEQ = 40$.
3. P2 risponde (al MITM, pensando sia P1) con `SYN-ACK` e $ACK = 41$.
4. Il MITM può continuare la conversazione con P2, mentre P1 non riceverà mai una risposta corretta al suo $SEQ = 30$ originale, sabotando l'intera conversazione.

3.2 Vulnerabilità lungo il Percorso di Rete

Quando un peer si connette a un sito web (es. `ftp.com`), la comunicazione attraversa molteplici macchine (router, switch) intermedi (es. 20 hop).

- **S**uperficie d'attacco: Ognuna di queste 20 macchine è un potenziale punto di attacco. È impossibile conoscere a priori tutte le vulnerabilità presenti lungo l'intera catena.
- **D**ifesa applicativa: La componente applicativa (es. TLS/SSL) deve agire per difendere il *percorso* (garantendo crittografia e integrità dei dati end-to-end).
- **L**imiti della difesa: Tuttavia, la difesa applicativa non protegge dal basso livello di rete.

L'introduzione di pacchetti anomali (spoofing) da parte di un attaccante e la gestione dei "non-care" (pacchetti inattesi o malformati) possono portare alla mancata ricezione del messaggio originale ed effettivo (Denial of Service) o all'iniezione di dati malevoli. Questa immagine mostra il **diagramma a stati finiti (Finite State Machine, FSM)** del protocollo TCP. È fondamentalmente la "mappa logica" che ogni computer utilizza per gestire il ciclo di vita di una connessione TCP, dall'apertura al trasferimento dati fino alla chiusura.

Gli ovali rappresentano gli **stati**, mentre le frecce rappresentano le **transizioni**. Le transizioni sono attivate da eventi (es. un comando dall'applicazione, `app_`, o un pacchetto ricevuto, `recv_`) e possono causare l'invio di un pacchetto (es. `send_`).

4 Apertura della Connessione (Handshake a 3 vie)

Questo processo coinvolge un lato "attivo" (il client, che inizia la connessione) e un lato "passivo" (il server, che ascolta).

4.1 Lato Server (Apertura Passiva)

1. `CLOSED`: Lo stato di partenza. Non c'è connessione.
2. `LISTEN`: Il server esegue un `app_passive open` (es. un server web che si avvia) e si mette in ascolto su una porta, in attesa di richieste.
3. `SYN_RCVD`: Il server (in `LISTEN`) riceve un pacchetto `SYN` (Synchronize) dal client. Risponde inviando il proprio `SYN` e l' `ACK` (Acknowledgment) del `SYN` del client. Ora attende l'`ACK` finale.

4.2 Lato Client (Apertura Attiva)

1. `CLOSED`: Stato di partenza.
2. `SYN_SENT`: L'applicazione (es. il browser) esegue un `app_active open` per connettersi. Invia un pacchetto `SYN` al server e attende la risposta `SYN_ACK`.

4.3 Completamento dell'Handshake

- Quando il Client (in `SYN_SENT`) riceve il `SYN_ACK` dal server, invia l'ultimo `ACK` e passa allo stato `ESTABLISHED`.
- Quando il Server (in `SYN_RCVD`) riceve l'ultimo `ACK` dal client, passa anch'esso allo stato `ESTABLISHED`.

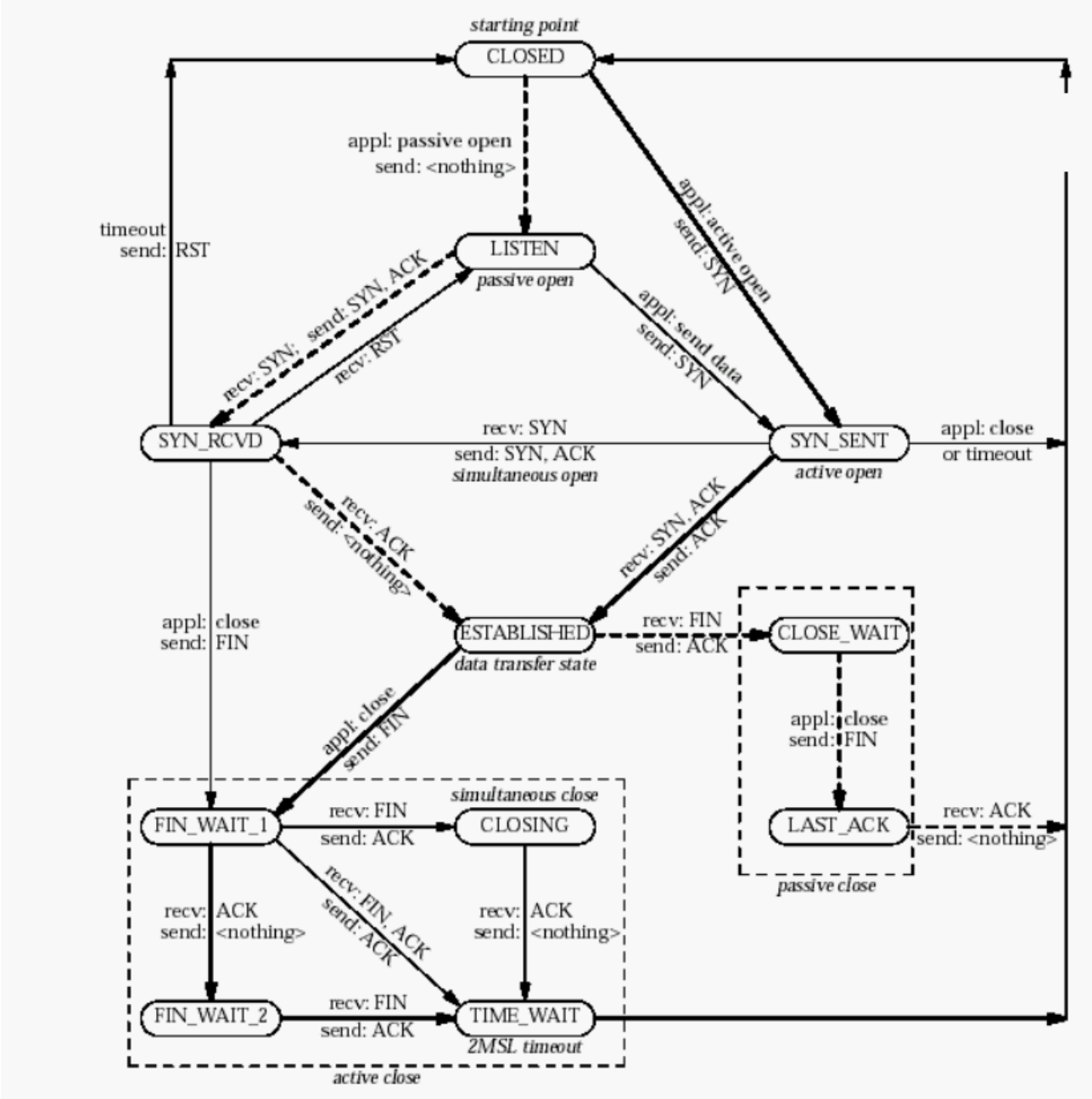

Figura 1: Diagramma a Stati Finiti del Protocollo TCP

5 Trasferimento Dati

- `ESTABLISHED`: Questo è lo stato principale della connessione. L'handshake è completo e i due peer possono scambiarsi liberamente dati (pacchetti PSH+ACK). È lo stato in cui avvengono la segmentazione e il riassemblaggio.

6 Chiusura della Connessione (Handshake a 4 vie)

La chiusura di TCP è gestita in modo che ogni lato possa terminare la sua parte di invio ("half-close") indipendentemente.

6.1 Lato che inizia la chiusura (Chiusura Attiva)

1. `ESTABLISHED`: L'applicazione decide di chiudere (`app_close`).
2. `FIN_WAIT_1`: Il sistema invia un pacchetto `FIN` (Finish) e attende l' `ACK` da parte dell'altro peer.
3. `FIN_WAIT_2`: Riceve l' `ACK` per il suo `FIN`. Ora sa che l'altro peer ha ricevuto la sua richiesta di chiusura. Rimane in attesa del `FIN` dall'altro lato (che indica che anche l'altro peer ha finito di inviare dati).
4. `TIME_WAIT`: Riceve il `FIN` dall'altro peer, invia l'ultimo `ACK` e entra in questo stato. È uno stato di "quarantena" (che dura `2MSL`, *Two Maximum Segment Lifetime*) per assicurarsi che l'ultimo `ACK` sia arrivato e per evitare che pacchetti "orfani" di questa connessione vengano confusi con una nuova connessione.

6.2 Lato che riceve la richiesta di chiusura (Chiusura Passiva)

1. `ESTABLISHED`: Riceve un `FIN` dal peer che ha iniziato la chiusura.
2. `CLOSE_WAIT`: Invia un `ACK` per confermare la ricezione del `FIN`. Informa la propria applicazione che la connessione è in chiusura (l'altro lato non invierà più dati). L'applicazione locale deve finire le sue operazioni e chiudere a sua volta.
3. `LAST_ACK`: L'applicazione locale esegue `app_close`. Il sistema invia il proprio `FIN` e attende l'ultimo `ACK`.
4. `CLOSED`: Riceve l'ultimo `ACK` (quello inviato dal peer che è entrato in `TIME_WAIT`) e chiude definitivamente la connessione.

Infine, anche il peer in `TIME_WAIT` scadrà il suo timer e tornerà a `CLOSED`.

7 `traceroute` (o `tracert`)

7.1 Definizione

`traceroute` (noto come `tracert` su sistemi Windows) è un programma diagnostico a riga di comando per reti di computer. Il suo scopo è tracciare il percorso (la rotta) che i pacchetti IP seguono da un host sorgente a un host destinazione.

7.2 Utilizzo e Funzionamento

L'uso principale di `traceroute` è la diagnosi dei problemi di rete. Permette di:

- Identificare ogni singolo "salto" (hop), solitamente un router, che un pacchetto attraversa per raggiungere la sua destinazione.
- Misurare il tempo di latenza (Round Trip Time, RTT) per ogni hop, aiutando a individuare colli di bottiglia o punti in cui la connessione è lenta.
- Rilevare la perdita di pacchetti lungo il percorso.

Principio di Funzionamento

`traceroute` opera inviando una serie di pacchetti (tipicamente pacchetti ICMP Echo Request o UDP) verso la destinazione. La sua peculiarità è quella di manipolare il campo **Time-To-Live (TTL)** nell'intestazione IP.

1. Inizia inviando un pacchetto con $TTL = 1$. Il primo router lungo il percorso decrementa il TTL a 0, scarta il pacchetto e invia un messaggio ICMP `Time Exceeded` (Tempo scaduto) all'host sorgente.
2. La sorgente registra l'indirizzo IP del primo router (dall'intestazione del messaggio ICMP) e il tempo impiegato.
3. Ripete l'operazione con $TTL = 2$. Il primo router inoltra il pacchetto, ma il secondo router lo scarta (decrementando il TTL da 1 a 0) e invia un ICMP `Time Exceeded`.
4. Questo processo continua, incrementando il TTL di 1 per ogni "giro", fino a quando il pacchetto raggiunge la destinazione finale.
5. Quando la destinazione viene raggiunta, essa risponde con un messaggio diverso (es. ICMP `Port Unreachable` se si usano pacchetti UDP, o `Echo Reply` se si usa ICMP), segnalando a `traceroute` che il percorso è stato completato.

8 `netstat` (Network Statistics)

8.1 Definizione

`netstat` è un'utility a riga di comando che fornisce informazioni e statistiche sulle connessioni di rete attive, le tabelle di instradamento e le interfacce di rete. È disponibile sulla maggior parte dei sistemi operativi, inclusi Windows, Linux e macOS.

8.2 Utilizzo

`netstat` è uno strumento fondamentale per il monitoraggio e la diagnosi dello stato della rete locale di un host. I suoi casi d'uso più comuni includono:

- connessioni attive: Mostra tutte le connessioni TCP e UDP attive, includendo l'indirizzo IP locale, l'indirizzo IP straniero (remoto) e le rispettive porte.
- porte in ascolto: Permette di vedere quali porte sul computer sono aperte e in stato `LISTENING`, ovvero in attesa di connessioni in entrata (es. un server web sulla porta 443).
- connessioni ai processi: Su molti sistemi, con le opzioni appropriate (es. `-b` su Windows, `-p` su Linux), può mostrare quale programma o processo (PID) sta utilizzando una determinata porta o connessione.
- tabelle di routing: Mostra la tabella di routing IP locale, utile per diagnosticare problemi di instradamento.
- statistiche di interfaccia: Fornisce statistiche sul traffico di rete per ciascuna interfaccia (pacchetti inviati/ricevuti, errori, ecc.).

È particolarmente utile in cybersecurity per una rapida analisi: se si sospetta una connessione non autorizzata, `netstat` può rivelare se il computer sta comunicando con indirizzi IP sospetti.

9 `base64`

9.1 Definizione

`Base64` non è un protocollo o un comando diagnostico di rete, ma uno **schema di codifica**. È un metodo standardizzato per convertire dati binari (come immagini, file eseguibili, o qualsiasi sequenza di byte) in un formato di testo ASCII.

Il nome deriva dal fatto che utilizza un set di 64 caratteri (A-Z, a-z, 0-9, `+` e `/`) per rappresentare i dati binari.

9.2 Utilizzo e Funzionamento

L'obiettivo principale di `Base64` è garantire che i dati binari possano essere trasmessi o memorizzati in sistemi che sono progettati per gestire esclusivamente dati testuali.

- **A**llegati E-mail (MIME): Storicamente, il protocollo SMTP (per le email) era progettato solo per testo ASCII a 7-bit. `Base64` è usato per codificare gli allegati (immagini, PDF) in modo che possano viaggiare come testo nel corpo del messaggio email.
- **I**ncorporamento in URL e HTML: A volte è utile incorporare piccole immagini o dati direttamente in un file HTML o CSS (*Data URI*) senza dover fare una richiesta di rete separata.
- **A**PI e JSON/XML: I formati JSON e XML sono testuali. Se un'API deve restituire un file binario (es. un certificato) all'interno di una risposta JSON, lo codificherà in `Base64`.

Principio di Funzionamento

`Base64` funziona prendendo 3 byte di dati binari (3 byte = 24 bit) e rappresentandoli come 4 caratteri ASCII.

1. Prende 3 byte (24 bit) di input.
2. Suddivide questi 24 bit in 4 gruppi da 6 bit ciascuno.
3. Ogni gruppo da 6 bit può rappresentare $2^6 = 64$ valori diversi (da 0 a 63).
4. Mappa ognuno di questi 64 valori a un carattere specifico della tabella `Base64`.
5. Se i dati di input non sono un multiplo di 3 byte, viene aggiunto un padding (carattere `=`) alla fine della stringa per indicare quanti byte mancavano.

Importante: La codifica `Base64` *non* è crittografia e non offre alcuna sicurezza. È facilmente reversibile (decodificabile). Il suo unico scopo è la rappresentazione dei dati. Comporta inoltre un aumento della dimensione dei dati di circa il 33%.

10 Il Protocollo HTTP (HyperText Transfer Protocol)

10.1 Definizione e Contesto

Il protocollo HTTP è un protocollo di comunicazione a livello applicativo. La sua struttura si compone di una **parte di controllo** (le intestazioni o headers) e una **parte dati** (il corpo o body). Questo messaggio HTTP viene incapsulato, a sua volta, nella parte dati di un segmento TCP.

La porta "suggerita" (standard) per il protocollo HTTP è la porta **80**.

10.2 Struttura di una Richiesta HTTP

Una comunicazione HTTP è basata su un modello client-server. Il client invia una richiesta (request) e il server risponde (response).

La struttura di una richiesta client è la seguente:

1. **Riga di Richiesta (Request Line):**
   - `Metodo`: L'azione richiesta (es. GET).
   - `URI (Uniform Resource Identifier)`: L'identificativo della risorsa (es. /index.html).
   - `Versione Protocollo`: Solitamente `HTTP/1.1` o superiore.
   - Terminatore di riga (CRLF).
2. **Intestazioni (Headers):**
   - Una serie di coppie `chiave: valore` che forniscono informazioni aggiuntive.
   - **Campo Host:** Obbligatorio dalla versione 1.1. Indica il dominio a cui è destinata la richiesta (es. `Host: ftp.com`).
   - Altre intestazioni comuni: `User-Agent`, `Accept`, `Content-Length`, ecc.
   - Ogni header è terminato da CRLF.
3. **Riga Vuota (CRLF):** Una riga vuota (solo CRLF) che separa gli header dal corpo del messaggio.
4. **Corpo (Body):** (Opzionale) Contiene i dati inviati al server, ad esempio con un metodo POST.

Sia nella richiesta che nella risposta, i dati (body) seguono sempre la riga vuota. Nelle prime versioni di HTTP (1.0), il client doveva attendere che il server chiudesse la connessione per sapere che i dati erano terminati. Con `HTTP/1.1` è stato introdotto l'header `Content-Length`, che specifica la lunghezza esatta del corpo del messaggio, permettendo connessioni persistenti.

10.3 Metodi HTTP

I metodi HTTP definiscono l'azione che si desidera eseguire sulla risorsa specificata:

- **GET**: Richiede una rappresentazione della risorsa. È il metodo più comune (es. scaricare una pagina web).
- **POST**: Invia dati al server (es. compilare un modulo) affinché li processi. I dati sono contenuti nel corpo (body) della richiesta.
- **PUT**: Carica una risorsa sul server, sostituendo qualsiasi versione esistente all'URI specificato.
- **HEAD**: Identico a GET, ma richiede solo gli header della risposta, senza il corpo. Utile per controllare la validità di una risorsa senza scaricarla.
- **DELETE**: Rimuove la risorsa specificata.

- **OPTIONS**: Richiede informazioni sulle opzioni di comunicazione disponibili per una risorsa (es. quali metodi sono supportati).
- **CONNECT**: Stabilisce un tunnel verso il server identificato dall'URI (usato principalmente per proxy HTTPS).

10.4 Sintassi Completa della URI

L'URI (Uniform Resource Identifier) ha una sintassi complessa che può includere diverse parti:

```
HTTP://username:password@domain:porta/path?chiave=valore#tag
```

- `HTTP://`: Schema del protocollo.
- `username:password@`: (Obsoleto) Credenziali per l'autenticazione di base.
- `domain`: Il dominio o l'indirizzo IP del server.
- `:porta`: La porta su cui il server è in ascolto (opzionale, default 80 per HTTP).
- `/path`: Il percorso della risorsa sul server.
- `?chiave=valore`: La "query string", usata per passare parametri (es. in un GET).
- `#tag`: Un frammento o "anchor", usato dal client (browser) per posizionarsi in un punto specifico della pagina (non viene inviato al server).

10.5 Protocollo Stateless (Senza Stato)

HTTP è un protocollo **stateless**: ogni richiesta è indipendente e il server non mantiene memoria delle richieste precedenti. Se una risorsa è "pregiata" (protetta), il client deve inviare le credenziali (es. `username:password`) per *ogni* richiesta di accesso.

Storicamente, software come Apache Web Server permettevano di limitare l'accesso a specifiche directory o pagine tramite file `.htaccess`, ma oggi questa forma di autenticazione (HTTP Basic Auth) è considerata insicura se non usata su HTTPS, poiché le credenziali viaggiano in chiaro.

10.6 Evoluzione: da HTTP 1.0 a 1.1 e il Campo Host

Con HTTP 1.0, per accedere a un sito (es. `ftp.com`), un client doveva:

1. Risolvere il dominio `ftp.com` in un indirizzo IP tramite una query DNS.
2. Effettuare una `connect(IP)` sulla porta 80.

Questo modello presupponeva una relazione 1:1 tra indirizzo IP e dominio.

Il protocollo HTTP 1.0 **non prevedeva il campo Host**. Con l'avvento del *virtual hosting* (più siti web sullo stesso server, con lo stesso IP), è diventato necessario specificare quale sito si intendeva visitare.

Con **HTTP 1.1**, l'header `Host` è diventato obbligatorio. Dopo la connessione TCP, il client invia:

```
GET / HTTP/1.1
Host: ftp.com
...
```

Il server legge l'header `Host` e può così instradare la richiesta al sito corretto (es. `ftp.com`) tra i tanti che ospita su quell'IP.

Tutta questa comunicazione, se avviene su HTTP (porta 80), è una **connessione in chiaro**: chiunque in ascolto sulla rete (es. un Man-in-the-Middle) può leggere l'intera richiesta e risposta, inclusi eventuali username, password o cookie.