Protocolli applicativi

Protocolli applicativi

Nel contesto delle reti di calcolatori, il termine **protocollo** è interpretato come un insieme di regole che regola il formato dei messaggi scambiati tra computer [Popovic, 2018].

Tre specifiche fondamentali:

- Procedure (e regole) di gestione dei messaggi
- Processamento errori
- Formato dei messaggi

Gestione dello stato del protocollo

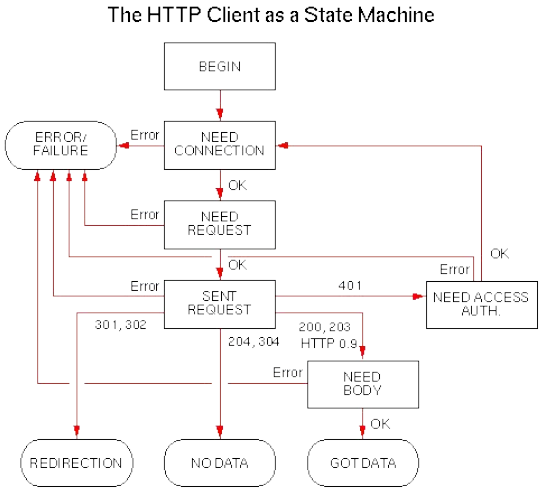
Il progettista di applicazioni distribuite deve affrontare il problema della gestione dello stato di un protocollo applicativo

Nel paradigma di comunicazione a scambio messaggi, è utile ragionare in termini di messaggi come eventi che cambiano lo stato del sistema

Uso di diversi strumenti:

- Reti di Petri
- Macchine a stati finiti
- Linguaggi formali di specifica (es. TLA)
- ecc.

Esempio: Stato Client HTTP

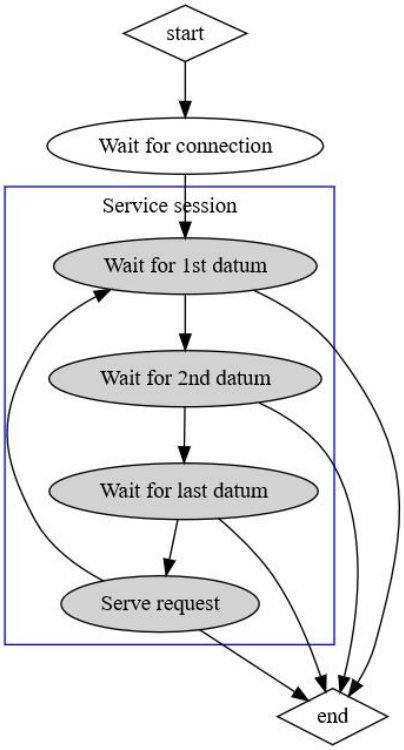


Courtesy: https://www.w3.org/Library/User/Architecture/HTTPFeatures.html

Macchine a stati

Il progetto di protocolli applicativi complessi esula dallo scopo dell'insegnamento

Tipicamente, noi lavoreremo con protocolli con procedure di gestione messaggi molto semplici, che corrispondono a una macchina a stati con un flusso sostanzialmente lineare, come quella in figura



Per approfondire il tema protocolli

- M. Popovic, "Communication Protocol Engineering", 2nd Edition, CRC Press, 2018
- Hartmut Koenig, "Protocol Engineering", Springer, 2012

Formato messaggi

- Oltre alla codifica stessa delle informazioni (problema che abbiamo già trattato nella lezione sull'eterogeneità) il progettista deve occuparsi di organizzazione di informazioni in messaggi che vengono scambiati tra Client e Server
- Spazio di progetto abbastanza ampio

Parsing messaggi

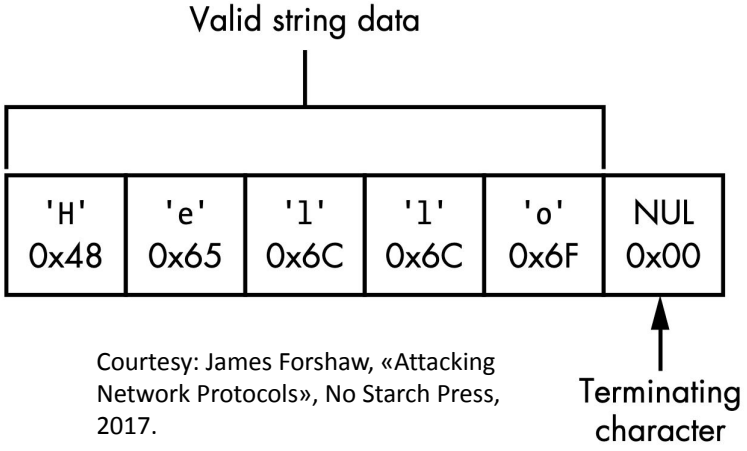
- Il parsing dei messaggi di input rappresenta un aspetto critico dello sviluppo di applicazioni distribuite da molti punti di vista (correttezza, robustezza, sicurezza, performance)
- Parsing è ***estremamente error prone***. Caldamente consigliabile utilizzare strumenti dedicati per la generazione del codice di parsing dei messaggi, a partire dai **parser combinator**.
- Possibile adozione di strumenti di testing (es. fuzzying) e di soluzioni per la verifica di correttezza formale
- Sfortunatamente, non abbiamo tempo per trattare sistematicamente questo argomento, assai complesso, nel nostro insegnamento. Adotteremo pertanto delle soluzioni ad hoc con un ragionevole compromesso tra qualità di design del protocollo e correttezza, robustezza, sicurezza, performance del codice di parsing.

Parsing messaggi

- Per approfondire:
  - S. Bratus et al., “Curing the Vulnerable Parser: Design Patterns for Secure Input Handling”, login Usenix Mag. 42(1) (2017)
  - P. Chifflier and G. Couprie, "Writing Parsers Like it is 2017," 2017 IEEE Security and Privacy Workshops (SPW), San Jose, CA, USA, 2017, pp. 80-92, doi: 10.1109/SPW.2017.39.
  - https://github.com/UpstandingHackers/hammer
  - https://github.com/dloss/binary-parsing

Terminated data

Una possibilità è quella di definire uno speciale terminatore che indica la fine di una informazione a lunghezza variabile



Courtesy: James Forshaw, «Attacking Network Protocols», No Starch Press, 2017.

Terminated data

- Pros:
  - Uso di testo UTF-8 terminato da «\n» modo semplice e human-friendly di realizzare protocolli (anche multilinea, come HTTP 1.* ed email/RFC822)
  - Implementazione relativamente semplice sender-side indipendentemente dal terminatore
- Cons:
  - Notevole overhead di gestione buffer (scansione per trovare il terminatore, eventuale spostamento di memoria, ecc.)
  - Terminatore non può essere usato come informazione valida (uso doppio terminatore o escaping)
  - Attenzione alla definizione di una dimensione massima per i messaggi!
  - In C non è disponibile una funzione come readLine di Java

Esempio lettura messaggio null-terminated - 1/2

```
uint8_t buffer[BUFSIZE]; /* buffer di appoggio */
uint8_t value[BUFSIZE]; /* buffer in cui metterò l'informazione */
int bytes_received = 0;
uint8_t *term_ptr;
size_t left = sizeof(buffer);
do {
   /* controllo se ho ricevuto il terminatore '\0' */
   if ((term_ptr = memchr(buffer, '\0', bytes_received)) != NULL)
       break;
   /* esco se leggo dato di dimensioni maggiori di BUFSIZE */
   if (left == 0) {
       fprintf(stderr, "Protocol error!\n");
       exit(EXIT_FAILURE);
   }
   cc = read(ns, buffer + bytes_received, left);
   if (cc < 0) { perror("read"); close(ns); exit(EXIT_FAILURE); }
   bytes_received += cc; left -= cc;
} while (1);
```

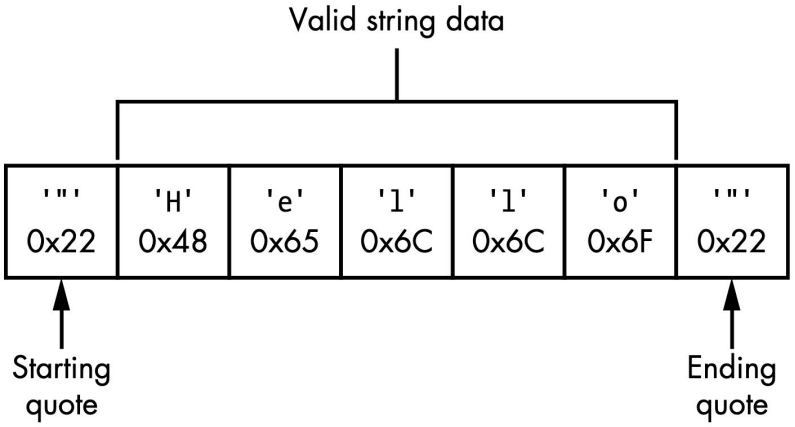
Esempio lettura messaggio null-terminated - 2/2

```
/* calcolo dimensione informazione ricevuta */
int inform_size = term_ptr - buffer;

/* sposto informazione da buffer a value */
memcpy(value, buffer, inform_size);

/* sposto dati letti ma non ancora processari all'inizio di buffer */
memcpy(buffer, term_ptr + 1,
       bytes_received - inform_size - 1);
```

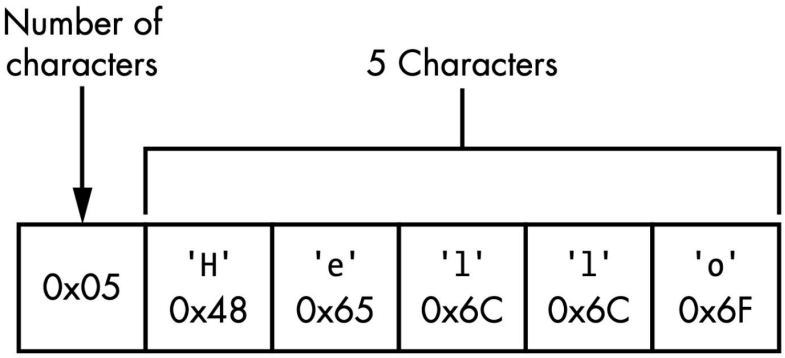
Terminated data con doppio terminatore



Courtesy: James Forshaw, «Attacking Network Protocols», No Starch Press, 2017.

Length-prefixed data

Una seconda possibilità è quella di anteporre a ogni informazione a lunghezza variabile un campo che ne indica la dimensione



Courtesy: James Forshaw, «Attacking Network Protocols», No Starch Press, 2017.

Length-prefixed data

- Pros:
  - Implementazione relativamente semplice receiver-side
  - Gestione dei buffer significativamente più semplice rispetto a terminated data
- Cons:
  - Leggero aumento di complessità sender-side rispetto ad approcci terminated data
  - Impossibile discriminare informazioni tra loro

Esempio lettura length-prefixed data in C

```
/* leggo dimensione campo e la ricostruisco in field_length */
cc = read(ns, buffer, 2); /* uso buffer appoggio per leggere field_length */
if (cc < 0) {
        perror("read"); exit(EXIT_FAILURE);
} else if (cc != 2) {
        fputs("Wrong protocol: cannot read next field length!", stderr);
        exit(EXIT_FAILURE);
}
field_length = (buffer[1] << 0) | (buffer[0] << 8); /* network byte order */

/* leggo field_length bytes dalla socket e li metto nel buffer value_buf */
to_read = field_length; index = 0;

while (to_read > 0) {/* equivalente della funzione read_n dello stevens */
   cc = read(ns, value_buf + index, to_read);
   if (cc < 0) { perror("read"); exit(EXIT_FAILURE); }
   to_read -= cc; index += cc;
}
```

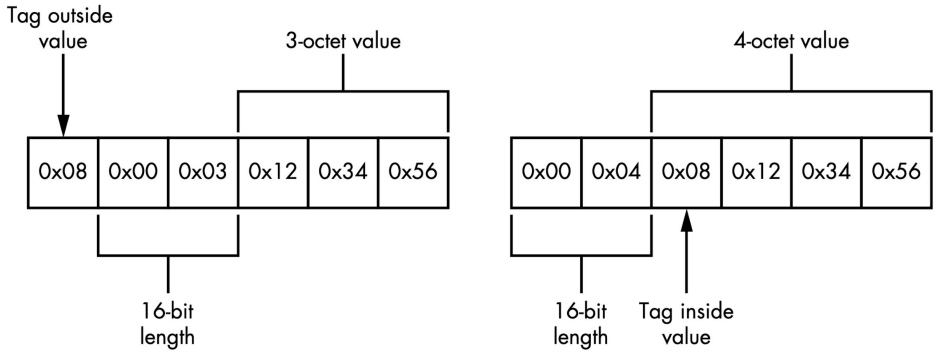
Esempio scrittura length-prefixed data in Java

```
/* scrivo su Socket toServer la stringa anno in codifica UTF-8 */
byte[] anno_utf8 = anno.getBytes("UTF-8");
int anno_len = anno_utf8.length;
byte[] len = new byte[2];
len[0] = (byte)((anno_len & 0xFF00) >> 8);
len[1] = (byte)(anno_len & 0xFF);

toServer.write(len);
toServer.write(anno_utf8);
```

Tag (o Type), Length, and Value

Una seconda possibilità è quella di anteporre a ogni informazione a lunghezza variabile un campo che ne indica la dimensione



Courtesy: James Forshaw, «Attacking Network Protocols», No Starch Press, 2017.

Tag (o Type), Length, and Value

- Pros:
  - Sostanzialmente gli stessi di length-prefixed data
  - Possibile discriminare informazioni tra loro
- Cons:
  - Sostanzialmente gli stessi di length-prefixed data
  - Aumento di complessità rispetto a length-prefixed data
  - Attenzione a non sottodimensionare il campo type

Formati standard

- Infine, è possibile ricorrere a formati standard:
- Testuali
  - (Simplified) Canonical S-expressions (per un esempio si veda la entry https://en.wikipedia.org/wiki/Canonical_S-expressions di Wikipedia)
  - XML
  - JSON
- Binari
  - Google Protocol Buffer (Protobuf)
  - Mempack
  - CBOR