
# Midterm Project

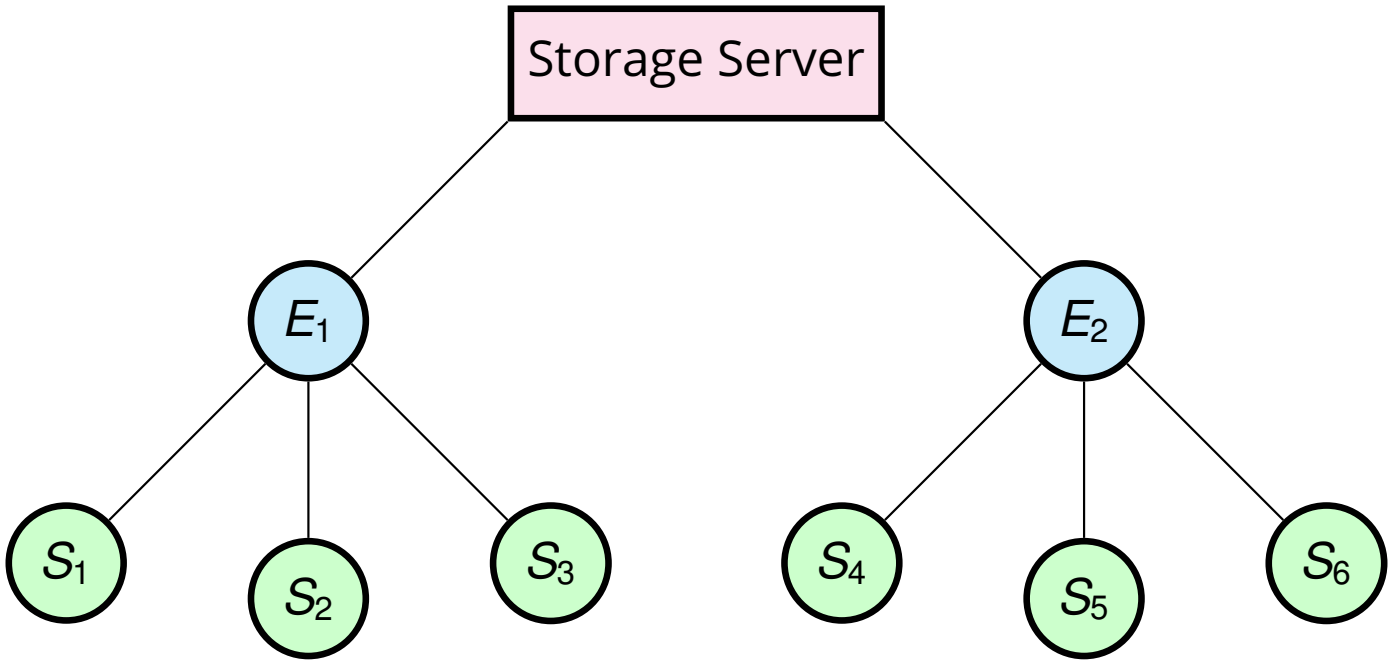
April 21, 2024

# Presentation Overview

# Premessa

*Nella presentazione sono inclusi degli snippet di codice che contengono solo i passaggi fondamentali per comprendere il funzionamento del progetto. È quindi importante notare che alcune componenti cruciali, come la gestione degli errori, sono state omesse per motivi di spazio.*

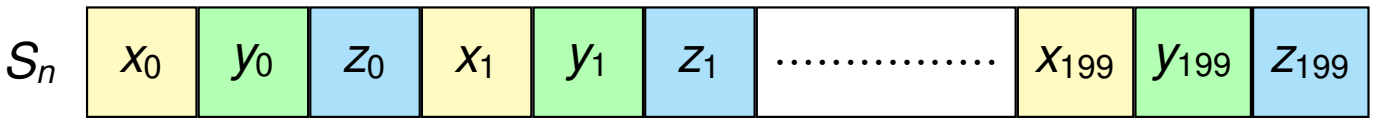
# Struttura generale del progetto



# Requisiti

- Ogni edge deve gestire i dati provenienti da tre sensori;
- Su di essi, deve essere calcolato l'RMS;
- I dati e l'RMS devono essere inviati al server di storage;
- Inoltre, i dati devono essere memorizzati in locale su file CSV;

# Formato dei dai ricevuti



# Gestione dei dati

1. Viene creato un nuovo thread per ogni connessione;
2. I dati vengono inseriti all'interno di una struct formata da tre array;
3. Si calcola l'RMS e viene aggiunto alla struct;
4. Mentre i dati vengono inviati al server di storage, i segnali vengono memorizzati su file CSV.

# Implementazione messa in ascolto dell'edge

```
1    while (1) {
2        pthread_t client_thread;
3        struct sockaddr_in client_addr;
4        int *client_sock = (int*)malloc(sizeof(int));
5        int client_size = sizeof(client_addr);
6
7        *client_sock = accept(server_socket, (struct sockaddr*) &client_addr, &
     client_size);
8        if (*client_sock != -1){
9            pthread_create(&client_thread, NULL, handle_client, (void*)
     client_sock);
10           pthread_join(client_thread, NULL);
11           free(client_sock);
12       } else {
13           printf("There was an error while accepting the client.\n");
14           exit(0);
15       }
16   }
17
```

# Implementazione ricezione dei dati

```
1   while (offset < bytes_received) {
2       for (int i = 0; i < SERIES_LENGTH; i++) {
3           float value = *((float*)(client_message + offset));
4           switch (i){
5           case 0:
6               values.x[index] = value;
7               break;
8           case 1:
9               values.y[index] = value;
10              break;
11          case 2:
12              values.z[index] = value;
13              break;
14          }
15          offset += sizeof(float);
16      }
17      index++;
18  }
19
```

- Tramite l'offset è possibile scorrere i byte ricevuti dai sensori;
- Tramite l'index si vanno a posizionare i valori all'interno della struct.

# Cos'è l'RMS

*RMS* sta per **Root Mean Square** o, in italiano, **valore efficace**. Quando un segnale è variabile nel tempo, come una forma d'onda sinusoidale che oscilla da positiva a negativa, la sua potenza istantanea cambia continuamente. L'RMS è una misura che tiene conto di questa variazione nel tempo, fornendo un valore che rappresenta la quantità di potenza effettivamente dissipata o trasmessa dal segnale.
La formula per calcolarlo su un segnale discreto è la seguente:

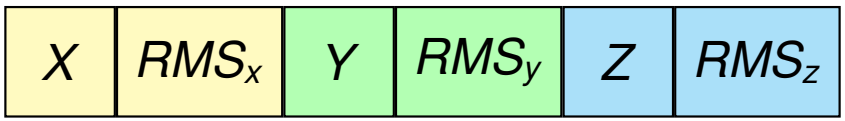
$$x_{rms} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$$

# Calcolo RMS e invio al server di storage

```
1    rms_values rms = root_mean_square(values);
2    values.x[ROWS_BEFORE_SENDING] = rms.x;
3    values.y[ROWS_BEFORE_SENDING] = rms.y;
4    values.z[ROWS_BEFORE_SENDING] = rms.z;
5
6    pthread_create(&thread, NULL, send_to_server, (void*) &values);
7    write_to_file(values);
8    pthread_join(thread, NULL);
9
```

- `root_mean_square` calcola l'RMS per ognuno dei tre segnali;
- `send_to_server` invia i dati al server di storage;
- `write_to_file` scrive i segnali su file CSV.

# Formato dei dati inviati al server




$$X = \{x_0, x_1, ..., x_{199}\}$$

$$Y = \{y_0, y_1, ..., y_{199}\}$$

$$Z = \{z_0, z_1, ..., z_{199}\}$$

# Confronto con TCP

Rispetto a TCP, per gestire i pacchetti inviati tramite socket raw si hanno a disposizione solamente `recv` o `recvfrom`.
Questo comporta la presenza di maggiori difficoltà:

- Bisogna prima leggere i byte per differenziare i vari pacchetti;
- Come conseguenza, è più complesso gestire il multi-threading, non avendo la coda del metodo `listen`.

# Risoluzione dei problemi

Per risolvere i problemi appena menzionati, sono state adottate le seguenti soluzioni:

- L'utilizzo della libreria **libcap**, scelta per la maggiore flessibilità dei metodi offerti rispetto all'uso dei socket raw.
- L'implementazione di una coda dalla quale i thread possono estrarre i pacchetti e analizzarli senza bloccare la ricezione di nuovi pacchetti.

# Utilizzo di pcap: messa in ascolto dell'edge

```
1    handle = pcap_open_live(device, snapshot_length, 1, 10000, error_buffer);
2    pcap_compile(handle, &filter, filter_exp, 0, PCAP_NETMASK_UNKNOWN);
3    pcap_setfilter(handle, &filter);
4    while(1) {
5        pcap_loop(handle, 0, queue_packet, NULL);
6    }
7
```

- `pcap_open_live` viene utilizzato per recuperare l'handle di un determinato network device;
- `pcap_compile` e `pcap_setfilter` applicano dei filtri ai pacchetti recuperati;
- `pcap_loop` gestisce ogni pacchetto tramite la funzione target `queue_packet`.

# Utilizzo di pcap: recuperare l'Ethernet header

```
1     node_t* client = (node_t*) args;
2
3     struct ether_header *eth_header;
4     eth_header = (struct ether_header *) client->packet;
5
6     if (ntohs(eth_header->ether_type) != ETHERTYPE_IP) {
7         printf("Not an IP packet. Skipping...\n\n");
8         return;
9     }
10
```

# Utilizzo di pcap: recuperare gli header IP e TCP

```
1   ip_header = client->packet + ethernet_header_length;
2   ip_header_length = ((*ip_header) & 0x0F);
3   ip_header_length = ip_header_length * 4;
4   printf("IP header length (IHL) in bytes: %d\n", ip_header_length);
5
6   u_char protocol = *(ip_header + 9);
7   if (protocol != IPPROTO_TCP) {
8       printf("Not a TCP packet. Skipping...\n\n");
9       return;
10  }
11
12  tcp_header = client->packet + ethernet_header_length + ip_header_length;
13  tcp_header_length = ((*(tcp_header + 12)) & 0xF0) >> 4;
14  tcp_header_length = tcp_header_length * 4;
15  printf("TCP header length in bytes: %d\n", tcp_header_length);
16
```

# Utilizzo di pcap: recuperare il payload

```
1    int total_headers_size = ethernet_header_length+ip_header_length+
     tcp_header_length;
2
3    payload_length = client->header->caplen - total_headers_size;
4    printf("Payload size: %d bytes\n", payload_length);
5
6    payload = client->packet + total_headers_size;
7    printf("Memory address where payload begins: %p\n\n", payload);
8
```

Dopo aver recuperato il payload, si può procedere nello stesso modo in cui si procedeva con la socket TCP, ovvero scorrendo i byte e posizionando i valori all'interno della struct.

# Gestione del multi-threading (1)

Per gestire il multi-threading, nel main vengono creati dei thread che non verranno più terminati:

```
1     pthread_t thread_ids[ACCEPTED_CLIENTS];
2
3     for(int i = 0; i < ACCEPTED_CLIENTS; i++){
4         pthread_create(&thread_ids[i], NULL, threads_function, NULL);
5     }
6
```

# Gestione del multi-threading (2)

Il compito di ogni thread è estrarre elementi dalla coda quando sono presenti:

```
1   void* threads_function(void* args){
2       while(1) {
3           pthread_mutex_lock(&queue_mutex);
4           node_t* client = dequeue();
5           pthread_mutex_unlock(&queue_mutex);
6           if (client != NULL) {
7               handle_client((void*) client);
8           } else {
9               pthread_mutex_lock(&queue_mutex);
10              pthread_cond_wait(&cond, &queue_mutex);
11              pthread_mutex_unlock(&queue_mutex);
12          }
13      }
14  }
15
```

Notiamo come nel caso in cui la coda dovesse essere vuota, vengono messi in wait tramite la condizione.

# Gestione del multi-threading (3)

Per essere sbloccati, ogni volta che un pacchetto viene messo in coda viene sbloccata la condizione:

```
1    void queue_packet(u_char *args, const struct pcap_pkthdr *header, const u_char
       *packet){
2        pthread_mutex_lock(&queue_mutex);
3        enqueue(args, header, packet);
4        pthread_cond_signal(&cond);
5        pthread_mutex_unlock(&queue_mutex);
6    }
7
```

# Implementazione della coda: enqueue

```
1  void enqueue(u_char *args, const struct pcap_pkthdr *header, const u_char *
   packet) {
2      node_t* new_node = malloc(sizeof(node_t));
3      new_node->args = args;
4      new_node->header = header;
5      new_node->packet = packet;
6      new_node->next = NULL;
7
8      if(tail == NULL) {
9          head = new_node;
10     } else {
11         tail->next = new_node;
12     }
13     tail = new_node;
14 }
15
```

# Implementazione della coda: dequeue

```
1   node_t* dequeue(){
2       if(head == NULL) {
3           return NULL;
4       } else {
5           node_t* result = head;
6           node_t* temp = head;
7           head = head->next;
8           if (head == NULL) {
9               tail = NULL;
10          }
11          free(temp);
12          return result;
13      }
14  }
15
```

Come possiamo notare dal codice, si tratta di una coda FIFO.

# Implementazione della coda: dequeue

Per poter analizzare in maniera più dettagliata il codice, è possibile recarsi alla seguente repo:

`https://github.com/gabriele-agosta/edgeserver`

# Grazie per l'attenzione