Midterm Project

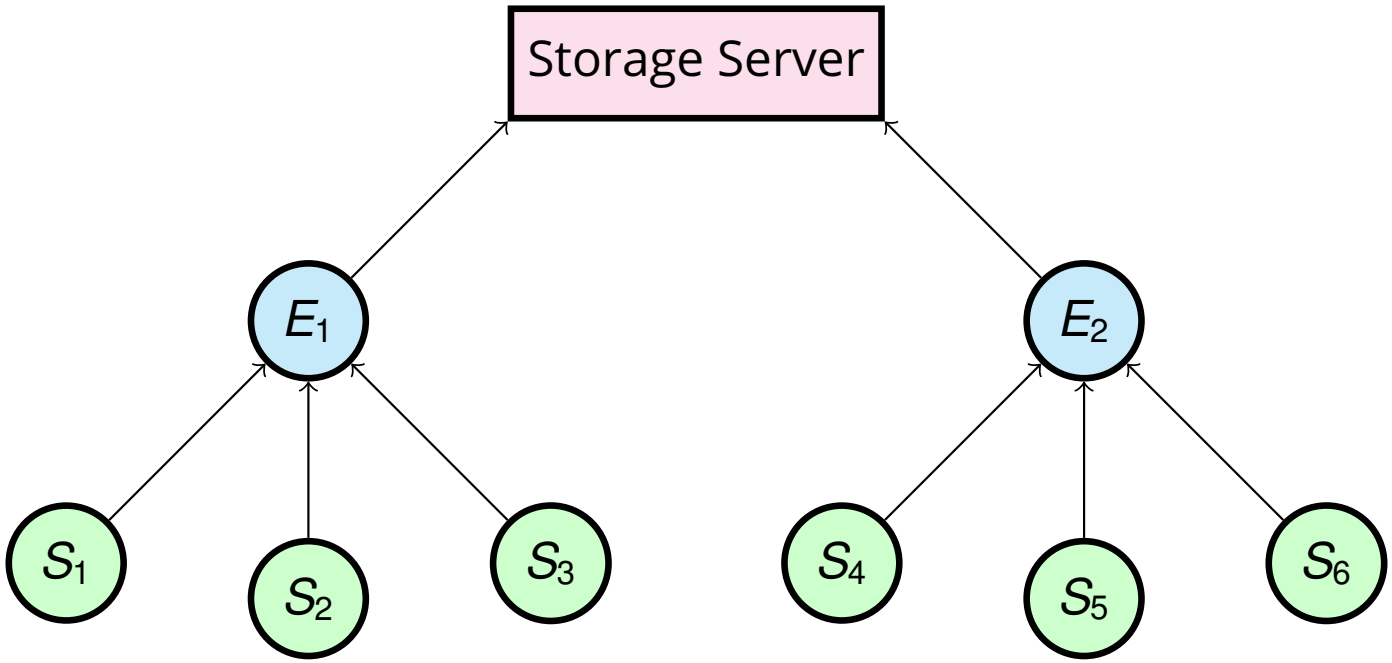
April 22, 2024

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.

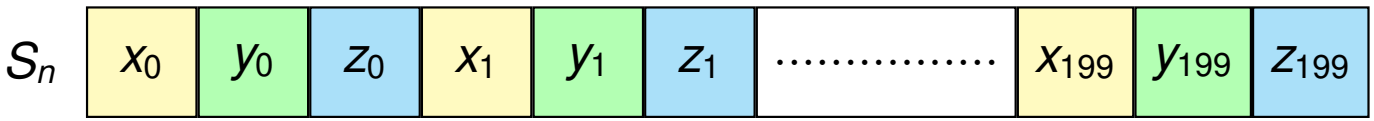
Struttura generale del progetto



Requisiti

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

Formato dei dati ricevuti



Gestione dei dati

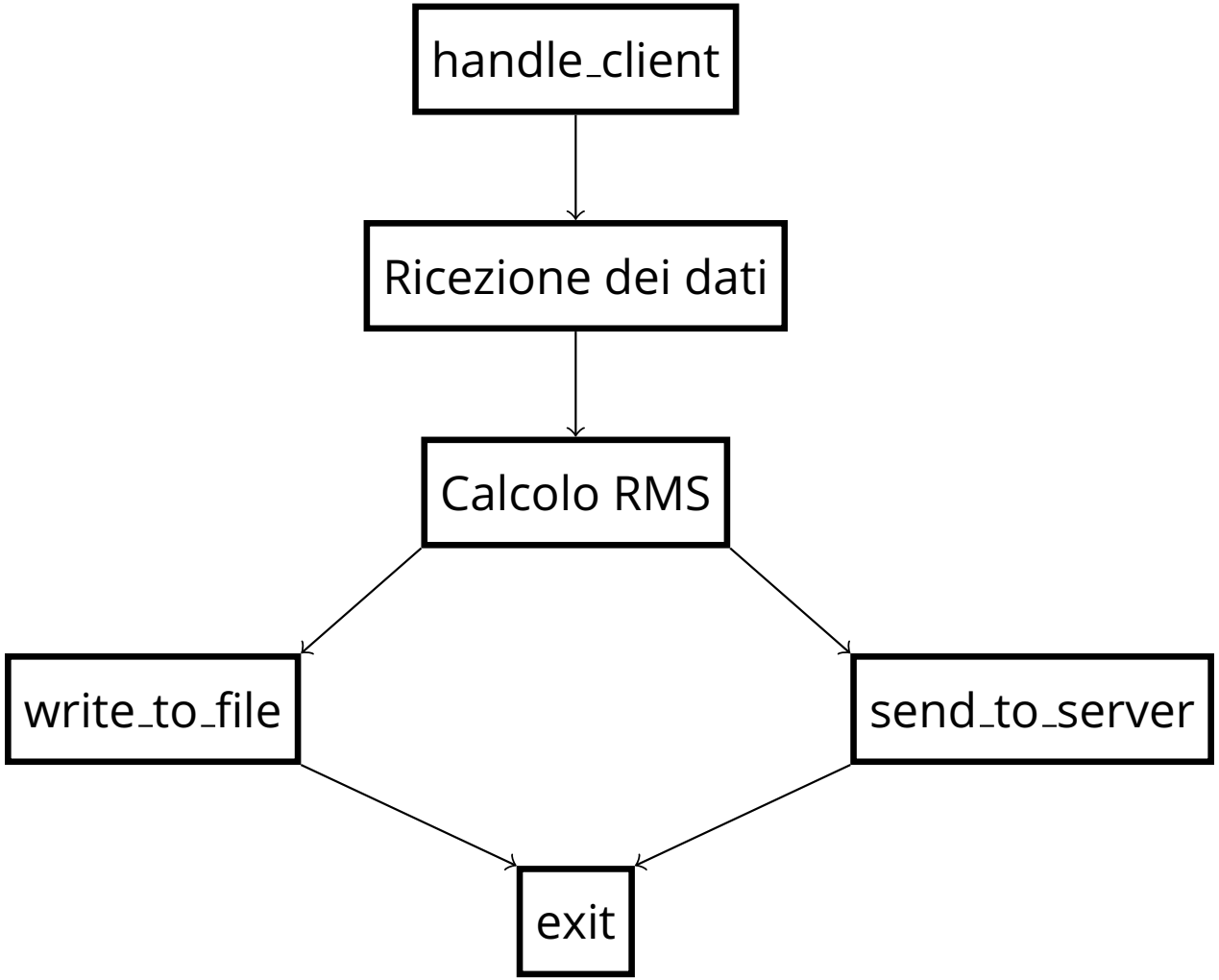
1. Viene creato un nuovo thread per ogni connessione;
2. I valori dei segnali vengono inseriti all'interno di una struct formata da tre array;
3. Si calcola il RMS e viene aggiunto alla struct;
4. Mentre i dati vengono inviati al server di storage, i segnali vengono memorizzati su file CSV.

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
```

- Il server rimane indefinitivamente in ascolto;
- Quando un sensore si connette, viene creato un thread per gestirlo.

Schematizzazione di handle client



Implementazione

```
void* handle_client(void *args) {
    ...
    bytes_received = recv(client_sock, ..., ..., MSG_WAITALL);
    while (offset < bytes_received) {
        for (int i = 0; i < SERIES_LENGTH; i++) {
            float value = *((float*)(client_message + offset));
            switch (i){
            case 0:
                values.x[index] = value;
                break;
            case 1:
                values.y[index] = value;
                break;
            case 2:
                values.z[index] = value;
                break;
            }
            offset += sizeof(float);
        }
        index++;
    }

```

- Il thread, dopo aver ricevuto la socket come argomento, scorre i dati ricevuti tramite l'offset.

Operazioni in seguito alla ricezione

```
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    close(client_sock);
10   pthread_exit(0);
11
```

- `root_mean_square` calcola il 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.

Cos'è il RMS

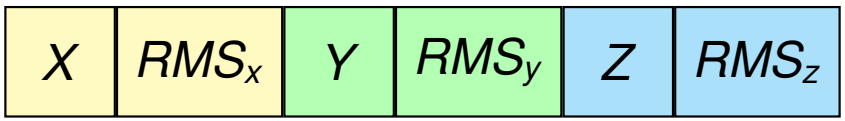
RMS sta per **Root Mean Square** o, in italiano, **valore efficace**. Quando un segnale è variabile nel tempo, come una forma d'onda sinusoidale che oscilla da positiva a negativa, la sua potenza istantanea cambia continuamente. Il RMS è una misura che tiene conto di questa variazione nel tempo, fornendo 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:

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

Implementazione calcolo del RMS

```
1     for (int i = 0; i < ROWS_BEFORE_SENDING; i++){
2         rms.x += pow(values.x[i], 2);
3         rms.y += pow(values.y[i], 2);
4         rms.z += pow(values.z[i], 2);
5     }
6
7     rms.x = sqrt(rms.x / ROWS_BEFORE_SENDING);
8     rms.y = sqrt(rms.y / ROWS_BEFORE_SENDING);
9     rms.z = sqrt(rms.z / ROWS_BEFORE_SENDING);
10
```

Formato dei dati inviati al server



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

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

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

Implementazione invio dei dati al server

```
1  float message[(ROWS_BEFORE_SENDING * SERIES_LENGTH) + SERIES_LENGTH];
2  ...
3  destination_socket = socket(AF_INET, SOCK_STREAM, 0);
4  ...
5  connect(destination_socket, (struct sockaddr*) &destination_addr, sizeof(
     destination_addr))
6  send(destination_socket, message, sizeof(message), 0)
7
```

- Si inseriscono i dati tutti all'interno di un unico messaggio;
- Si crea la socket;
- Dopo aver instaurato la connessione, viene effettuata l'operazione di invio.

Scrittura su file

```
1   pthread_mutex_lock(&file_mutex);
2   fp = fopen(filename, "a");
3   if (fp != NULL) {
4       fprintf(fp, "\n");
5       for (int i = 0; i < ROWS_BEFORE_SENDING; i++){
6           fprintf(fp, "%f, %f, %f\n", values.x[i], values.y[i], values.z[i]);
7       }
8       fclose(fp);
9   } else {
10      printf("There was an error while opening the file: %s\n", strerror(errno))
     ;
11  }
12  pthread_mutex_unlock(&file_mutex);
13
```

- Per evitare che più thread possano scrivere contemporaneamente su file, si fa utilizzo di un mutex.

Repo github

Per poter analizzare in maniera più dettagliata il codice, è possibile recarsi ad una delle seguenti repo:

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

Operazioni necessarie per eseguirlo

Gli adattamenti necessari per poter eseguire il codice in contesti diversi dal laboratorio sono:

- Adattare le direttive del preprocessore SERIES_LENGTH, ROWS_BEFORE_SENDING, SERVER_PORT, DESTINATION_PORT alle proprie necessità;
- Modificare gli indirizzi IP dell'edge e del server di destinazione

Grazie per l'attenzione