# Relazione Progetto
# VITO BARRA

## Struttura

Ho utilizato una struttura multi-client; all'inizio dell'esecuzione il main imposta i parametri opzionali e gli handler dei segnali e successivamente forka, generando cosi un processo figlio **Collector** che farà da server.

### Multi-Client

Il flusso principale del main genera la quantità di thread richiesta come impostato nei parametri. Ogni thread proverà ad aprire una connesione al server creando così una corrispondenza uno ad uno *thread-client*.

### Server

Il server è sequenziale ed utilizza una *select* per ottimizzare l'esecuzione del codice attendendo che ci sia quacosa da leggere su almeno uno dei client presi in esame nella *FD_SET*.

## Gestione Sincronizzazione

Per gestire la sincronizzazione ho scritto una struct e delle funzioni di interfaccia in un apposito file: SyncData.c

```
typedef struct SyncData
{
    SyncCounter_t *shm_SyncCounter;
    pthread_cond_t ServerOpened;
    pthread_mutex_t mut;
    int ServerInit;
} SyncData_t;
```

Un'istanza di questa struttura è codivisa tra i processi **MainWorker** e **Collector** tramite *SharedMemory.*

### Server-client

Per assicurarmi che il server sia aperto e pronto a ricevere connessioni ho utilizzato una *contidition variable* su un valore "booleano" di inizializzazione. Prima che i client tentino di aprire una connessione al server attendono sulla sua inizializzazione.

Il server comunica il completamento dell'inizializzazione ai client in attesa mediante un broadcast su ServerOpened.

Una volta assicuratosi che il server è stato inizializzato, il client richiede una connessione e se viene rifiutata ritenta dopo un secondo. Se durante la richesta dovessero finire le task, il server, e quindi il socket si chiudono; il client smette di richere l'accesso controllando sull'errore "NO SUCH FILE OR DIRECTORY"

## Segnali

I segnali sono implementati per gestire la chiusura del programma svuotando la coda dei task, se ancora piena, al momento della ricezione del segnale. Lo svuotamento viene gestito con un counter  condiviso tra client e server che contiene la quantità di task da eseguire. Questo viene incrementato quando il main aggiunge una task e decrementato quando una task viene terminata dal server.

```
typedef struct SyncCounter
{
    pthread_cond_t OnAzeration;
    pthread_mutex_t mut;
    int counter;
} SyncCounter_t;
```

Quando viene ricevuto uno dei segnale tra SIGHUP, SIGINT, SIGQUIT o SIGTERM, il programma avvia un handler di segnali che attende su OnAzeration; una volta che il server avrà decrementato il counter portandolo a zero sapremo che avrà finito le task caricate, e allora l'handler del segnale procederà liberando la memoria delle variabili utilizzate.

le SIGPIPE viene gestito con una stampa d'errore ma non interrompe l'esecuzione del programma.

## Terminazione

Quando i due processi **main** e **collector** finiscono, chiamano rispettivamente le due funzioni

```
void AT_exitCallBack_client();
void AT_exitCallBack_server();
```

che gestiscono le ultime cose prima di chiudere il programma e in caso di chiusura brusca. In particolare fanno l'*unlink* del socket e rilasciano la memoria della SharedMemory.