DOCUMENTAÇÃO – TRABALHO PRÁTICO 2 – REDES DE COMPUTADORES
VICTOR DELLA CROCE MALTEZ – 2019042392

1. Introdução

O código foi desenvolvido com objetivo de resolver o seguinte desafio: implementar um projeto que provê um ambiente de controle de informações elétricas de uma universidade, em que tem as seguintes funcionalidades: consultar **o servidor** na Subestação de Energia (SE), que contém o estado das estações de produção de energia elétrica responsáveis por todo o monitoramento do sistema de geração de energia solar fotovoltaica; e consultar **o servidor** do Sistema de Controle de Iluminação Inteligente (SCII), que tem o objetivo de controlar os postes de iluminação inteligentes que proveem iluminação e informações em tempo real de consumo elétrico no campus. Além disso, o ambiente deve dispor de Interfaces de Controle (IC) com acesso à SE e ao SCII **(os clientes),** cuja função é consultar os dados do estado de produção de energia elétrica e os dados do sistema de iluminação inteligente no campus inteligente por meio de protocolos de comunicação, através da Internet.

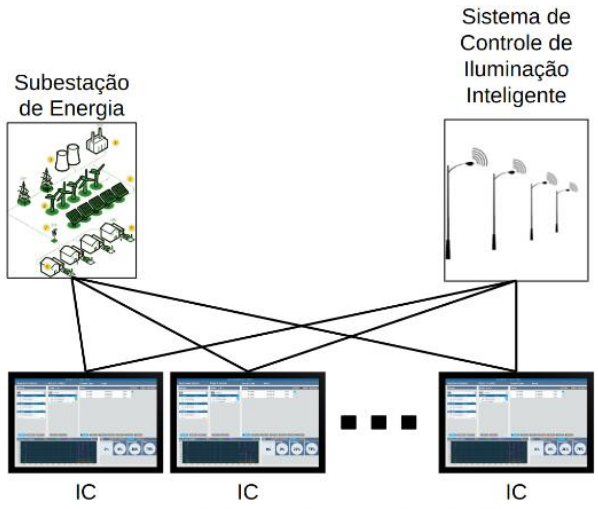

Figura 1 - Exemplo de comunicação entre as entidades do projeto

2. Arquitetura

O sistema é baseado em uma arquitetura cliente-servidor, onde o servidor gerencia as salas e os sensores, e o cliente interage com o servidor enviando solicitações e recebendo respostas. A comunicação entre cliente e servidor é feita por meio de sockets TCP/IP, garantindo a confiabilidade na transferência de dados.

Para gerenciamento de múltiplas conexões de clientes foi utilizada uma manipulação de threads a partir da biblioteca ‘pthread.h’. Com ela, na função create_thread (explicada posteriormente), há a passagem do socket do cliente como parâmetro, depois é feita a conexão, processamento de mensagens e fechamento da conexão e liberação de recursos após pedido de desconexão. Cada conexão é tratada por uma thread separada, que executa em paralelo, processando mensagens e gerando respostas apropriadas. A arquitetura garante que o servidor possa aceitar novas conexões enquanto ainda está processando outras, melhorando a escalabilidade e a capacidade de resposta do servidor.

3. Mensagens

Foram utilizadas constantes (defines) para armazenar as mensagens que são trocadas entre clientes e servidores.

a. Mensagens do cliente para o servidor:

i. **REQ_ADD**: Solicita a adição de um novo cliente.

ii. **REQ_REM**: Solicita a remoção de um cliente existente.
iii. **REQ_INFOSE**: Solicita informações sobre a produção.
iv. **REQ_INFOSCII**: Solicita informações sobre o consumo.
v. **REQ_STATUS**: Solicita o status atual da produção.
vi. **REQ_UP**, **REQ_NONE**, **REQ_DOWN**: Solicita ajustes no consumo.

b. Mensagens do servidor para o cliente:

i. **RES_ADD**: Resposta com o ID do cliente adicionado.
ii. **RES_INFOSE**, **RES_INFOSCII**: Respostas com as informações de produção e consumo.
iii. **RES_STATUS**: Resposta com o status atual da produção.
iv. **RES_UP**, **RES_NONE**, **RES_DOWN**: Respostas com os ajustes de consumo.
v. **OK01**, **ERROR01**, **ERROR02**: Respostas de status.

4. Cliente-Servidor

a. Cliente-Servidor (estruturas compartilhadas entre cliente e servidor)

i. Estrutura de Dados

1. **BUFSZ:** tamanho do buffer utilizado para enviar e receber mensagens.

b. Servidor

i. Estrutura de Dados

1. **client-count:** contador de clientes conectados
2. **clients[10]:** array que armazena o status dos clientes
3. **production:** produção de energia atual
4. **consumption:** consumo de energia atual
5. **old_consumption:** consumo de energia anterior à última alteração
6. **servername:** nome do servidor

ii. Funções

1. **generateRandomProduction():** gera um valor aleatório para a produção.
2. **generateRandomConsumption(int min, int max):** gera um valor aleatório para o consumo entre o mínimo (*min*) e máximo (*max*), passados como parâmetro.
3. **getClient(ID):** retorna um ID de cliente disponível.
4. **parse_rcv_message(char *buf, struct client_data *cdata):** processa a mensagem recebida do cliente, armazenada no *buf* passado como parâmetro, a partir do socket *cdata->csock*.
5. **client_thread(void *data):** função executada por cada thread de cliente, identificado a partir da variável *data* passada como parêmtro.

c. Cliente

i. Funções

1. **initialConenction(int _socketSE, int _socketSCII):** estabelece a conexão inicial com os servidores SE e SCII, sendo feita nela o pedido de início de comunicação com os servidores.

2. **parse_send_message(int _socketSE, int _socketSCII, char *buf, int cid):** processa e envia mensagens, armazenadas no *buf*, para os servidores ou servidor desejado.
3. **parse_rcv_message(char *buf, int _socketSCII, int cid):** processa as mensagens, armazenadas no *buf*, recebidas dos servidores.

5. Discussão

A arquitetura adotada permite a comunicação eficiente entre clientes e servidor, suportando múltiplos clientes simultaneamente através de threads. A utilização de sockets TCP garante a confiabilidade das conexões e a integridade das mensagens trocadas.

6. Conclusão

Este sistema cliente-servidor demonstra uma abordagem robusta para gerenciar e monitorar a produção e consumo de recursos em um ambiente com múltiplos clientes. A arquitetura modular e o uso de threads permitem uma escalabilidade eficiente, enquanto a comunicação via sockets TCP assegura a integridade e confiabilidade das mensagens trocadas.