UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
Relatório do Projeto de Serviço Distribuído: Soma de Números Inteiros
Trabalho Prático Parte 1

Adilson Enio Pierog – 00158803
Andres Grendene Pacheco - 00264397
Luís Filipe Martini Gastmann – 00276150
Mateus Luiz Salvi – 00229787

Este relatório descreve a implementação de um serviço distribuído de soma de números inteiros, cujo objetivo é receber requisições de múltiplos clientes, processá-las de forma concorrente e somar os números enviados a uma variável acumuladora no servidor. O trabalho foi desenvolvido utilizando a API User Datagram Protocol (UDP) e tecnologias de sincronização como mutexes e variáveis de condição POSIX.

A) Como foi implementado cada subserviço:

Cliente (arquivo "client.c")

Função principal: Lê números inteiros da entrada padrão e envia requisições ao servidor.
Inclui uma função de subprocesso (ClientInputSubprocess) para interpretar comandos, como "enviar número" e "sair". A separação em subprocessos melhora a modularidade e simplifica a manutenção.

Servidor (arquivo "server.c")

Função principal: Gerencia threads que processam simultaneamente as requisições recebidas.
Utiliza RunServer para inicializar o servidor e criar subprocessos. A concorrência via threads garante melhor desempenho no processamento de múltiplas requisições.

Protocolo e Sincronização (arquivo "server_prot.c")

Função principal: Implementa a lógica de sincronização e tratamento das requisições.
Utiliza a estrutura REQUEST_INFO para armazenar dados de cada requisição. A sincronização é essencial para evitar condições de corrida e garantir a consistência do sistema.

Descoberta de Rede (arquivo "discovery.c")

Função principal: Identifica sub-redes e servidores disponíveis na rede. Simplifica o processo de configuração e conexão cliente-servidor.

B) Em quais áreas do código foi necessário garantir sincronização no acesso a dados:

Áreas do Código com Sincronização

- Variável Acumuladora -> Proteção com mutexes para evitar condições de corrida ao acessar ou modificar o valor acumulado.
- Lista de Requisições -> Controle de acesso concorrente para evitar inconsistências.
- Estado das Threads -> Uso de variáveis de condição para gerenciar a sincronização entre threads.

Primitivas Utilizadas:

- Mutexes (pthread_mutex_t): Protegem seções críticas no acesso a recursos compartilhados.
- Variáveis de Condição (pthread_cond_t): Sincronizam as threads em operações dependentes.

C) Descrição das principais estruturas e funções que a equipe implementou:

Estrutura REQUEST_INFO: Representa uma requisição com atributos:
request_value: Valor do número a ser somado.
request_id: Identificador único da requisição.
processed: Estado do processamento.

Funções:

Cliente: ClientInputSubprocess: Processa comandos do cliente.
RunClient: Inicializa o cliente e inicia a comunicação com o servidor.

Servidor:
RunServer: Inicializa o servidor e suas threads.
ServerInputSubprocess: Gerencia entradas administrativas no servidor.
ServerListenForSleepSubprocess: Monitora o recebimento de requisições.

Para Comunicação: Uso de funções da API UDP (sendto, recvfrom) para troca de mensagens.

Descrição dos problemas que a equipe encontrou durante a implementação e como estes foram resolvidos (ou não).

Um dos problemas enfrentados foi a tentativa de implementar o envio de mensagens em broadcast para os clientes conectados. Apesar de configurar corretamente o socket e os endereços para broadcast, o sistema retornava uma mensagem de "acesso negado" ao tentar enviar os pacotes. Isso pode ter ocorrido devido a restrições de permissões no sistema operacional, configuração inadequada da opção `SO_BROADCAST` no socket, ou mesmo por limitações impostas pela rede local, como a desativação de pacotes broadcast no roteador ou firewall. A falta de experiência com as nuances da API de sockets em C também dificultou a identificação e resolução rápida do problema, resultando na necessidade de buscar soluções alternativas ou investigar mais a fundo as permissões e configurações do ambiente de execução.

Além do acima, outro dos problemas enfrentados durante a implementação foi o gerenciamento da lista de clientes, que recebe operações de leitura e escrita de múltiplas partes do código, dificultando o controle e a garantia de consistência. Essa complexidade resultou em comportamentos inesperados, especialmente em situações onde várias threads acessam a lista simultaneamente. Além disso, a dificuldade em cadastrar novos usuários foi significativa, causada em grande parte pela falta de prática com C, especialmente no uso de structs e ponteiros. A manipulação de memória dinâmica e o entendimento detalhado das estruturas de dados específicas da linguagem demandaram um esforço adicional para evitar erros como vazamentos de memória ou acessos inválidos. Essa curva de aprendizado atrasou a implementação de funcionalidades importantes e reforçou a necessidade de maior experiência com a linguagem para lidar com operações críticas como essa.