Sistemas Distribuídos - Relatório Detalhado

Universidade Federal do Ceará – Campus de Quixadá
Professor: Rafael Braga
Trabalho 1: Comunicação entre processos
Tema: Gestão de Supermercado
Dupla: Mônica Maria Rodrigues
Maryana Moraes

1. Introdução

Este relatório descreve de forma detalhada a implementação em C++ do Trabalho 1 da disciplina Sistemas Distribuídos — Comunicação entre processos (Sockets e Streams). O projeto implementa a aplicação 'Gestão de Supermercado' conforme o enunciado fornecido em PDF, adaptando os conceitos Java (OutputStream/InputStream) para C++.

2. Resumo do Projeto

Arquitetura: cliente-servidor TCP. O cliente serializa (empacota) objetos representando funcionários e os envia ao servidor. O servidor desserializa (desempacota) os bytes recebidos, reconstrói os objetos, adiciona-os à agregação 'Supermercado' e responde ao cliente com confirmação. Também há suporte para gravação em arquivo binário e escrita em stdout (equivalente a testar OutputStream de destino).

3. Estrutura de arquivos

A implementação está organizada da seguinte forma (pasta raiz do trabalho): include/ - funcionario.h - subclasses.h - supermercado.h - serializer.h

src/ - funcionario.cpp - subclasses.cpp - supermercado.cpp - serializer.cpp - servidor.cpp - cliente.cpp Makefile

4. Design de classes (POJOs)

As classes principais e suas responsabilidades:

- Escala (interface abstrata): define o método virtual puro mostrarEscala().
- Funcionario (superclasse): contém id, nome, salario; implementa Escala com uma implementação padrão; métodos: getters, exibirInfo(), tipo().
- Subclasses: Vendedor (comissao), Balconista (setor), Caixa (numeroCaixa), Gerente (bonus). Cada uma sobrescreve tipo(), exibirInfo() e mostrarEscala().
- Supermercado: agregador que mantém vector>; métodos adicionar() e listar().

Trecho: definição resumida de Funcionario (cabeçalho)

```
class Funcionario : public Escala {
protected:
        int id;
         std::string nome;
         double salario;
public:
        Funcionario();
        Funcionario(int id_, const std::string &nome_, double salario_);
        virtual ~Funcionario() = default;
        int getId() const;
        std::string getNome() const;
         double getSalario() const;
        virtual std::string tipo() const { return "Funcionario"; }
         virtual void exibirInfo(std::ostream &os = std::cout) const;
         std::string mostrarEscala() const override;
};
```

5. Serialização e Protocolo

A serialização foi implementada manualmente na classe Serializer (serializer.h / serializer.cpp). Formato por objeto (binário) usado: - tipo_len (1 byte) - tipo (tipo_len bytes) - id (int32) - nome_len (int32) - nome (nome_len bytes) - salario (double) - extras_len (int32) - extras (extras_len bytes) — string com pares chave=valor para atributos das subclasses (ex: comissao=150.0)

Antes dos objetos, no fluxo de socket, é enviado o número de objetos (int32, em network byte order). Para cada objeto é enviada a quantidade de bytes do objeto (int32) seguida do bloco de bytes do objeto — isso cumpre o requisito que pede enviar o número de bytes para cada objeto.

Trecho: empacotamento (packFuncionario) — resumo

```
// Exemplo resumido do empacotamento:
std::ostringstream os(std::ios::binary);
uint8_t tipo_len = tipo.size();
os.write((char*)&tipo_len, sizeof(tipo_len));
 os.write(tipo.c_str(), tipo_len);
int32_t id = f.getId();
```

```
os.write((char*)&id, sizeof(id));
int32_t nome_len = nome.size();
os.write((char*)&nome_len, sizeof(nome_len));
os.write(nome.c_str(), nome_len);
double s = f.getSalario(); os.write((char*)&s, sizeof(s));
int32_t extras_len = extras.size();
os.write((char*)&extras_len, sizeof(extras_len));
if (extras_len>0) os.write(extras.c_str(), extras_len);
```

6. Comportamento do Cliente

O cliente cria uma lista de funcionários (exemplo com Vendedor, Caixa, Balconista, Gerente). Em seguida: - escreve a lista em stdout (Serializer::writeToStream) — corresponde ao teste i do enunciado; - grava a lista em arquivo binário funcs.bin (writeToStream) — corresponde ao teste ii; - conecta ao servidor TCP (127.0.0.1:9090) e envia a lista via Serializer::writeToSocket — corresponde ao teste iii (envio a servidor remoto). Após enviar, o cliente lê uma resposta simples: número de funcionários recebidos (int32 em network byte order).

7. Comportamento do Servidor

O servidor abre a porta 9090, aceita conexões e cria uma thread para cada cliente (multithreaded). A thread chama Serializer::readFromSocket para reconstruir os objetos, cria um Supermercado, adiciona os funcionários e chama listar() para exibir as informações. Em seguida envia um reply (int32) de confirmação ao cliente.

8. Saída de Teste (exemplo real obtido)

Saída do servidor ao receber o cliente:

```
Servidor aguardando conexoes na porta 9090...
Cliente conectado (thread)
Supermercado: Central - Lista de funcionarios:
Vendedor - ID: 1 | Nome: Maria | Salario: 2500.5 | Comissao: 150 | Tipo: Vendedor | Escala: Vendedor
Caixa - ID: 2 | Nome: Joao | Salario: 1800.75 | NumCaixa: 3 | Tipo: Caixa | Escala: Caixa
Balconista - ID: 3 | Nome: Carla | Salario: 2000 | Setor: Padaria | Tipo: Balconista | Escala: Balconista
Gerente - ID: 4 | Nome: Ana | Salario: 5000 | Bonus: 700 | Tipo: Gerente | Escala: Gerente
Conexao com cliente encerrada.
```

9. Arquivo gerado (funcs.bin)

O cliente grava o arquivo funcs.bin em formato binário. Esse arquivo pode ser lido pelo método Serializer::readFromStream (por exemplo, redirecionando o conteúdo para um programa leitor ou criando um utilitário para ler o arquivo e imprimir as estruturas).

10. Mapeamento dos itens do enunciado para a implementação

- Item 1 (POJOs e classes): implementado em include/funcionario.h e include/subclasses.h.
- Item 2 (OutputStream customizado): implementado como Serializer::writeToStream / writeToSocket (testes em stdout, arquivo e TCP).
- Item 3 (InputStream customizado): implementado como Serializer::readFromStream / readFromSocket (stdin/file/socket podem ser usados como origem).
- Item 4 (cliente-servidor via sockets): implementado em src/cliente.cpp e src/servidor.cpp com empacotamento e desempacotamento.
- Item 5 (representação externa): foi usada serialização binária manual; Protobuf/JSON seriam alternativas opcionais.

11. Observações Técnicas e Boas Práticas

- A serialização manual requer cuidado com endianness. No código, enviados campos que representam tamanhos/contadores (int32) em network byte order (htonl/ntohl) para compatibilidade entre máquinas diferentes.
- Funções send_all/recv_all garantem a transmissão de todos os bytes.
- Uso de shared_ptr mantém gerenciamento automático de memória.
- O formato extras (string chave=valor) foi escolhido para simplicidade; para produção use Protobuf/JSON para interoperabilidade.

12. Instruções de Execução (passo a passo)

1) No terminal, compile: make

2) Em um terminal, execute o servidor:

./servidor

3. Em outro terminal, execute o cliente:

./cliente

4. Verifique as saídas em ambos os terminais e o arquivo funcs.bin gerado.

13. Melhorias sugeridas (opções para nota extra)

- Implementar um protocolo de controle com comandos (ADD, REMOVE, LIST, LOGIN) para que o cliente possa interagir dinamicamente com o servidor.

- Implementar autenticação (login) e sessão para o eleitor/votação caso adaptar para sistema de votação.
- Implementar mensagens multicast UDP para notificações (conforme item 5 do enunciado original sobre votações).
- Substituir a serialização manual por Protocol Buffers (protoc) para garantir robustez e documentação do formato.

14. Conclusão

O trabalho implementado em C++ atende aos requisitos do enunciado: criação de classes, serialização manual, testes em stdout/arquivo/TCP, e comunicação cliente-servidor multithreaded. O código está organizado e pronto para entrega.