Código - Protocolo Buffer

- Protocolo de Serialização que permite que essas mensagens sejam enviadas de forma compacta antes de jogar no socket, usados para a troca e armazenamento de informações entre sistemas
  - armazenam dados em um formato binário compacto, reduzindo o tamanho dos dados transmitidos.
- Seria o equivalente ao JSON ou XML
- Desenvolvida pela Google
- Útil se você quer acessar esses dados diretamente na memória sem criar cópias adicionais
  - Através de um Objeto (como 'Bytes' ou 'Bytesarray') que utilizam uma interface de buffer para obter um ponteiro para os dados no buffer.Isso permite que você leia ou escreva diretamente nos dados.

FUNCIONAMENTO

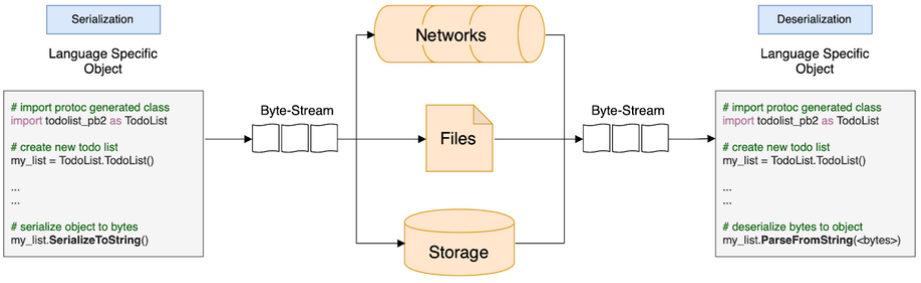

Figura 1: Essa lista de tarefas é, então, serializada e enviada pela rede, salva em um arquivo ou armazenada persistentemente em um banco de dados.O fluxo de bytes enviado é desserializado usando o método *parse*

TIPOS DE DADOS

- No Protocol Buffers (Protobuf), os tipos de dados básicos são usados para definir os campos das mensagens. Aqui estão os principais tipos de dados suportados:

Tipos de Dados Escalares

1. **Números Inteiros**
   - `int32`: Número inteiro com sinal de 32 bits.
   - `int64`: Número inteiro com sinal de 64 bits.
   - `uint32`: Número inteiro sem sinal de 32 bits.
   - `uint64`: Número inteiro sem sinal de 64 bits.
   - `sint32`: Número inteiro com sinal de 32 bits, otimizado para números negativos.
   - `sint64`: Número inteiro com sinal de 64 bits, otimizado para números negativos.
   - `fixed32`: Número inteiro sem sinal de 32 bits com tamanho fixo.
   - `fixed64`: Número inteiro sem sinal de 64 bits com tamanho fixo.
   - `sfixed32`: Número inteiro com sinal de 32 bits com tamanho fixo.
   - `sfixed64`: Número inteiro com sinal de 64 bits com tamanho fixo.
2. **Números de Ponto Flutuante**
   - `float`: Número de ponto flutuante de 32 bits.
   - `double`: Número de ponto flutuante de 64 bits.
3. **Booleano**
   - `bool`: Valor booleano (`true` ou `false`).
4. **Strings**
   - `string`: Sequência de caracteres UTF-8.
5. **Bytes**
   - `bytes`: Sequência de bytes arbitrários.

Tipos de Dados Complexos

- Além dos tipos escalares, você também pode definir tipos de dados complexos usando **mensagens** e **enums**:
- Cada `message` pode conter vários campos, e cada campo tem um nome e um número que o identifica. Esses números são usados para identificar os campos quando os dados são serializados

1. **Messages (Mensagens)**
    - Uma mensagem é um tipo de dado complexo que pode conter um ou mais campos de qualquer tipo de dados, incluindo outros tipos de mensagens.

```
message Person {
  string name = 1;        // Campo 1: Nome da pessoa
  int32 id = 2;           // Campo 2: ID da pessoa
  PhoneType phone_type = 3;  // Campo 3: Tipo de telefon
e, usa o enum PhoneType
}
```

2. **Enums**
    - Uma enumeração é usada para definir um conjunto de valores constantes.

```
enum PhoneType {
  MOBILE = 0;
  RESIDENCIAL = 1;
  TRABALHO = 2;
}
```

Repetição e Opcionalidade

- **Repetido**: Você pode usar a palavra-chave `repeated` para especificar que um campo pode conter múltiplos valores.

```
repeated string phone_numbers = 3;
```

Exemplo

1. Definindo o Formato dos Dados

Você já tem um arquivo `.proto` com a definição das mensagens e enums. Salve o código abaixo em um arquivo chamado `gerenciador.proto`.

```
syntax = "proto3";

package Gerenciador;

message Time {
    string nome = 1;
    string tecnico = 2;
    int32 pontos = 3;
    int32 qtdJogos = 4;
    map<string, Atleta> atletas = 5;

    message Tecnico {
        string nome = 1;
        int32 idade = 2;
        int32 qtdTitulos = 3;
    }
}

message Atleta {
    string nome = 1;
    Posicao posicao = 2;
    int32 numCamisa = 3;
    int32 qtdTitulos = 4;
    string time = 5;
    int32 idade = 6;

    enum Posicao {
        DEFAULT = 0;
        ZAGUEIRO = 1;
```

```
        LATERAL = 2;
        MEIO_CAMPO = 3;
        ATACANTE = 4;
        GOLEIRO = 5;
    }
}

message Message {
    int32 error = 1;
    int32 id = 2;
    string objRef = 3;
    string methodID = 4;
    bytes args = 5;
}
```

- **Instale o compilador Protobuf e o pacote Python:**

  Se ainda não tiver, instale o compilador `protoc` e o pacote `protobuf` para Python.

  ```
  pip install protobuf
  ```

- **Compile o arquivo `.proto` para gerar o código Python:**

  Execute o seguinte comando para compilar o arquivo `.proto` e gerar o código Python:

  ```
  protoc --python_out=. gerenciador.proto
  ```

  Isso criará um arquivo chamado `gerenciador_pb2.py` no diretório atual.

- **Use o código gerado para serializar e desserializar dados em Python:**

  Aqui está um exemplo de como você pode usar o código gerado para criar e manipular instâncias das mensagens `Time`, `Atleta`, e `Message`.

```
import gerenciador_pb2

# Criar uma instância de Atleta
atleta = gerenciador_pb2.Atleta(
    nome="João",
    posicao=gerenciador_pb2.Atleta.Posicao.ATACANTE,
    numCamisa=9,
    qtdTitulos=3,
    time="Time A",
    idade=25
)

# Criar uma instância de Time com um atleta
time = gerenciador_pb2.Time(
    nome="Time A",
    tecnico="Carlos",
    pontos=45,
    qtdJogos=20,
    atletas={
        "atleta_1": atleta
    }
)

# Serializar o objeto Time para um formato binário
serialized_time = time.SerializeToString()

# Desserializar o objeto Time a partir do formato binári
o
time_desserializado = gerenciador_pb2.Time()
time_desserializado.ParseFromString(serialized_time)

# Exibir os dados desserializados
print("Time desserializado:")
print(f"Nome: {time_desserializado.nome}")
print(f"Técnico: {time_desserializado.tecnico}")
print(f"Pontos: {time_desserializado.pontos}")
print(f"Quantidade de Jogos: {time_desserializado.qtdJog
os}")
```

```
for key, atleta in time_desserializado.atletas.items():
    print(f"Atleta {key}:")
    print(f"  Nome: {atleta.nome}")
    print(f"  Posição: {atleta.posicao}")
    print(f"  Número da Camisa: {atleta.numCamisa}")
    print(f"  Quantidade de Títulos: {atleta.qtdTitulo
s}")
    print(f"  Time: {atleta.time}")
    print(f"  Idade: {atleta.idade}")

# Criar uma instância de Message
message = gerenciador_pb2.Message(
    error=0,
    id=123,
    objRef="obj_1",
    methodID="method_1",
    args=b'argumentos'
)

# Serializar o objeto Message
serialized_message = message.SerializeToString()

# Desserializar o objeto Message
message_desserializado = gerenciador_pb2.Message()
message_desserializado.ParseFromString(serialized_messag
e)

# Exibir os dados da mensagem desserializada
print("Message desserializado:")
print(f"Error: {message_desserializado.error}")
print(f"ID: {message_desserializado.id}")
print(f"Objeto Referência: {message_desserializado.objRe
f}")
print(f"Método ID: {message_desserializado.methodID}")
print(f"Args: {message_desserializado.args}")
```

TIPOS DE DADOS

1. Tipos de Dados Básicos

- `string`:
  - **Uso**: Para representar texto. Use `string` para qualquer dado que seja textual, como nomes, descrições, e identificadores que são essencialmente palavras ou frases.
  - **Exemplo**: `string nome = 1;`, `string tecnico = 2;`, `string time = 5;`
- `int32`:
  - **Uso**: Para representar números inteiros que não exigem um tamanho maior que 32 bits. Use `int32` para valores inteiros pequenos a médios.
  - **Exemplo**: `int32 pontos = 3;`, `int32 qtdJogos = 4;`, `int32 numCamisa = 3;`, `int32 qtdTitulos = 4;`, `int32 idade = 6;`
- `bytes`:
  - **Uso**: Para dados binários ou bytes brutos. Use `bytes` quando você precisa armazenar dados que não são texto ou que são de tamanho variável e podem incluir qualquer tipo de dados binários.
  - **Exemplo**: `bytes args = 5;` em `Message`

Exemplos de Uso

- `string`:
  - Use para campos como `nome`, `tecnico`, e `time`, onde você precisa representar informações textuais, como o nome do time ou do técnico.
- `int32`:
  - Use para campos numéricos, como `pontos` e `qtdJogos`, onde você precisa armazenar números inteiros que não exigem valores muito grandes, como a quantidade de pontos ou jogos.
- `bytes`:
  - Use para dados que podem ser binários ou variáveis, como argumentos em `Message` que podem ser de qualquer tipo binário ou dados codificados.
- `message`:

    - Use para definir estruturas mais complexas que agrupam vários campos. Por exemplo, `Time` pode conter um `message Tecnico` para definir o técnico do time e um `map<string, Atleta>` para listar os atletas associados ao time.

- `map`:
    - Use para armazenar um conjunto de pares chave-valor. No `message Time`, o campo `atletas` é um `map` onde a chave é o nome do atleta e o valor é um `Atleta` específico.

- `enum`:
    - Use para definir um conjunto fixo de valores para uma variável. No `message Atleta`, o campo `posicao` é um `enum` que define as diferentes posições possíveis para um atleta, garantindo que o valor seja um dos valores predefinidos.