Sistema de Empréstimo de Livros via RPC

Herbert de Souza Mariano
Gabriel Perez Vargas de Vasconcelos
Merhi Osolins Daychoum

UFRRJ - 2025/1

4 de junho de 2025

Sumário

1 Introdução **2**

2 Escolha de desenvolvimento **2**

3 Dissecando o código **4**

3.1 `client.go` 4

3.1.1 4

3.1.2 5

3.1.3 6

3.1.4 7

3.1.5 8

3.2 `server.go` 9

3.2.1 Função `getLivroID` 9

3.2.2 Função `RealizarEmpréstimo` 10

3.2.3 Função `DevolverLivro` 11

3.2.4 Função `ConsultarLivro` 12

3.2.5 Função `ConsultarEmpréstimosUsuario` 13

3.2.6 Função `main` (servidor) 14

3.3 `datastore.go` 15

4 Conclusão **16**

5 Referências **17**

1 Introdução

O presente trabalho tem como objetivo o desenvolvimento de uma aplicação distribuída fundamentada no paradigma cliente/servidor, utilizando o mecanismo de chamada de procedimento remoto (Remote Procedure Call – RPC). A proposta foi concebida com base na implementação de um Sistema de Empréstimo de Livros em Biblioteca, solução que visa simular o gerenciamento de operações básicas em um acervo bibliográfico digital, incluindo o empréstimo, devolução e consulta de livros.

A aplicação contempla dois componentes principais: um servidor, responsável por armazenar de forma simplificada os dados da biblioteca e disponibilizar os métodos da API via RPC; e um cliente, capaz de consumir esses métodos remotamente por meio de uma interface textual de linha de comando. Tal estrutura permite explorar, de forma prática, os conceitos de comunicação entre processos distribuídos, abstração de serviços e desacoplamento entre camadas de responsabilidade, além de proporcionar um exercício significativo de aplicação de boas práticas de engenharia de software.

A abordagem adotada também considera princípios de organização arquitetônica inspirados na Clean Architecture, de forma a promover maior modularidade, manutenibilidade e separação de responsabilidades entre domínio, casos de uso e infraestrutura. Assim, este documento visa apresentar a fundamentação teórica da escolha arquitetural, as decisões de implementação adotadas, a dissecação das principais estruturas de código e os resultados obtidos com a execução da aplicação.

2 Escolha de desenvolvimento

A linguagem Go (Golang) foi escolhida como base para o desenvolvimento desta aplicação distribuída em virtude de suas características nativas de suporte à concorrência, simplicidade sintática, desempenho eficiente e robustez na manipulação de redes e sistemas distribuídos. Projetada pela Google, Go provê bibliotecas integradas como net/rpc, que facilitam a implementação de chamadas de procedimento remoto (RPC) de forma segura e performática, eliminando a necessidade de bibliotecas externas complexas para esse tipo de comunicação. Além disso, sua tipagem estática e gerenciamento eficiente de memória contribuem para a criação de sistemas confiáveis e com baixa incidência de erros em tempo de execução.

Para armazenamento dos dados persistentes relacionados aos livros, usuários e registros de empréstimo, optou-se pela utilização do SQLite, um sistema de gerenciamento de banco de dados relacional leve, autocontido e amplamente adotado em aplicações de pequeno e médio porte. A escolha se justifica por seu caráter embutido, o que elimina a necessidade

de configuração de servidores de banco de dados externos, além de oferecer portabilidade e facilidade de uso — atributos ideais para o escopo acadêmico e para a simplicidade proposta neste projeto. A interface entre o servidor Go e o SQLite é realizada com bibliotecas nativas de Go, garantindo integração eficiente com a lógica da aplicação.

A arquitetura desenvolvida, baseada no modelo cliente/servidor com RPC, demonstrou-se adequada ao propósito do projeto. O servidor concentra a lógica de negócio e atua como ponto central de manipulação dos dados, enquanto o cliente, executado em linha de comando, interage com a aplicação por meio de invocações remotas aos métodos definidos na API. Essa separação permite a escalabilidade do sistema, viabiliza a substituição de componentes com mínima interferência e reflete princípios de engenharia de software orientados à modularidade e reuso.

Neste contexto, apresenta-se a descrição da API RPC desenvolvida para o sistema de empréstimo de livros, a qual contempla os principais métodos remotos necessários para o funcionamento da aplicação. Entre eles, destacam-se as funcionalidades de realizar e devolver empréstimos, consultar a disponibilidade de um livro, verificar os empréstimos ativos de um usuário e acessar o histórico de empréstimos de um determinado exemplar. Cada método foi projetado para atender a uma necessidade específica do fluxo de gestão bibliotecária, permitindo que o cliente, por meio de chamadas remotas ao servidor, realize operações típicas de sistemas reais com simplicidade e eficácia.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

3 Dissecando o código

3.1 `client.go`

3.1.1

Função `realizarEmpréstimo`

```
func realizarEmpréstimo(usuario, livro, dataEmpréstimo string) {
	client, err := rpc.Dial("tcp", ":1234")
	if err != nil {
		log.Fatal("Failed to connect to server:", err)
	}

	var reply string
	err = client.Call("Server.RealizarEmpréstimo", struct{ Usuario, Livro, DataEmpréstimo string }{Usuario: usuario, Livro: livro, DataEmpréstimo: dataEmpréstimo}, &reply)
	if err != nil {
		log.Fatal(err)
	}
	fmt.Println(reply)
}
```

Figura 1: Implementação da função `realizarEmpréstimo`.

A função realizarEmpréstimo tem como finalidade estabelecer uma conexão remota com o servidor da aplicação, utilizando o protocolo RPC (Remote Procedure Call), e invocar o procedimento responsável por registrar o empréstimo de um livro a um determinado usuário, em uma data específica.

Sua lógica está organizada em três etapas principais:

- Estabelecimento da conexão RPC: A função utiliza o método rpc.Dial para se conectar ao servidor local na porta 1234. Caso essa conexão falhe, a aplicação é encerrada imediatamente por meio do método log.Fatal, que imprime a mensagem de erro.
- Envio da requisição remota: Uma estrutura anônima é criada contendo os dados necessários para o empréstimo (usuário, livro e data). Essa estrutura é passada como argumento para a chamada do método remoto Server.RealizarEmpréstimo.
- Recebimento da resposta e exibição: A resposta enviada pelo servidor é armazenada na variável reply e exibida ao usuário no terminal por meio do comando fmt.Println.

Em resumo, a função atua como intermediária entre a entrada do usuário e os serviços disponibilizados remotamente, sem executar nenhuma lógica de negócio por conta própria.

3.1.2

Função `devolverLivro`

```
func devolverLivro(usuario, livro string) {
	client, err := rpc.Dial("tcp", ":1234")
	if err != nil {
		log.Fatal("Failed to connect to server:", err)
	}

	var reply string
	err = client.Call("Server.DevolverLivro", struct{ Usuario, Livro string }{Usuario: usuario, Livro: livro}, &reply)
	if err != nil {
		log.Fatal(err)
	}
	fmt.Println(reply)
}
```

Figura 2: Implementação da função `devolverLivro`.

A função `devolverLivro` tem como finalidade estabelecer uma conexão remota com o servidor da aplicação, utilizando o protocolo RPC (Remote Procedure Call), e invocar o procedimento responsável por registrar a devolução de um livro previamente emprestado a um determinado usuário.

Sua lógica está organizada em três etapas principais:

- **Estabelecimento da conexão RPC:** A função utiliza o método `rpc.Dial` para se conectar ao servidor local na porta 1234. Caso essa conexão falhe, a execução da aplicação é interrompida com a exibição de uma mensagem de erro por meio da função `log.Fatal`.
- **Envio da requisição remota:** É criada uma estrutura anônima contendo os campos `usuario` e `livro`, que são enviados como argumentos para a chamada do método remoto `Server.DevolverLivro`.
- **Recebimento da resposta e exibição:** A resposta retornada pelo servidor é armazenada na variável `reply` e exibida no terminal utilizando o comando `fmt.Println`.

Assim como na função de empréstimo, a função `devolverLivro` atua apenas como intermediária entre o usuário e os serviços remotos da aplicação, sem executar lógica de negócio local.

3.1.3

Função `consultarLivro`

```
func consultarLivro(livro string) {
        client, err := rpc.Dial("tcp", ":1234")
        if err != nil {
                log.Fatal("Failed to connect to server:", err)
        }

        var reply string
        err = client.Call("Server.ConsultarLivro", livro, &reply)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(reply)
}
```

Figura 3: Implementação da função `consultarLivro`.

A função `consultarLivro` tem como objetivo estabelecer uma conexão remota com o servidor da aplicação e requisitar, por meio de uma chamada RPC (*Remote Procedure Call*), informações sobre a disponibilidade de um determinado livro no sistema.

Sua lógica está organizada em três etapas principais:

- **Estabelecimento da conexão RPC:** A função utiliza o método `rpc.Dial` para conectar-se ao servidor na porta 1234. Caso ocorra falha na conexão, a execução é imediatamente interrompida com a exibição de uma mensagem de erro.
- **Envio da requisição remota:** O título do livro é enviado diretamente como argumento para a chamada do método remoto `Server.ConsultarLivro`. Essa chamada visa obter a situação atual do livro (disponível ou emprestado).
- **Recebimento da resposta e exibição:** A resposta enviada pelo servidor, armazenada na variável `reply`, é exibida ao usuário por meio da função `fmt.Println`.

A função é responsável apenas pela comunicação entre o cliente e o servidor, não contendo nenhuma lógica de processamento local.

3.1.4

Função `consultarEmpréstimosUsuario`

```
func consultarEmpréstimosUsuario(usuario string) {
        client, err := rpc.Dial("tcp", ":1234")
        if err != nil {
                log.Fatal("Failed to connect to server:", err)
        }

        var reply string
        err = client.Call("Server.ConsultarEmpréstimosUsuario", usuario, &reply)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(reply)
}
```

Figura 4: Implementação da função `consultarEmpréstimosUsuario`.

A função `consultarEmpréstimosUsuario` tem como finalidade estabelecer uma conexão remota com o servidor e requisitar, por meio de uma chamada RPC (*Remote Procedure Call*), a lista de empréstimos ativos associados a um determinado usuário.

Sua lógica está organizada em três etapas principais:

- **Estabelecimento da conexão RPC:** Utiliza-se o método `rpc.Dial` para conectar-se ao servidor pela porta 1234. Caso a conexão não seja bem-sucedida, o programa é interrompido com uma mensagem de erro exibida pelo comando `log.Fatal`.
- **Envio da requisição remota:** O nome do usuário é passado como argumento para o método remoto `Server.ConsultarEmpréstimosUsuario`, com a expectativa de obter os registros de livros emprestados a ele.
- **Recebimento da resposta e exibição:** A resposta do servidor, armazenada na variável `reply`, é exibida ao usuário no terminal utilizando o comando `fmt.Println`.

A função atua como um canal de comunicação entre a entrada do usuário e a lógica do servidor, sem realizar nenhum processamento local.

3.1.5

Função `main`

```
func main() {
        // Exemplos de chamada das funções RPC
        realizarEmpréstimo("João", "Golang Programming", "2023-10-01")
        devolverLivro("João", "Golang Programming")
        consultarLivro("Golang Programming")
        consultarEmpréstimosUsuario("João")
        consultarEmpréstimosLivro("Golang Programming")
}
```

Figura 5: Função `main` com chamadas simuladas das operações RPC.

A função `main` representa o ponto de entrada da aplicação cliente e tem como finalidade realizar, de forma sequencial, chamadas simuladas às funções RPC implementadas. Cada uma das funções invocadas estabelece uma conexão com o servidor, realiza a operação correspondente e imprime o resultado no terminal.

As chamadas presentes na função estão organizadas conforme descrito abaixo:

- **realizarEmpréstimo:** Solicita ao servidor que registre o empréstimo do livro *"Golang Programming"* para o usuário *"João"*, com a data de empréstimo definida como *"2023-10-01"*.
- **devolverLivro:** Solicita a devolução do mesmo livro pelo mesmo usuário. Esta operação presume que o empréstimo anterior tenha sido registrado com sucesso.
- **consultarLivro:** Requisita informações sobre a disponibilidade do livro *"Golang Programming"*, retornando se o mesmo está disponível ou emprestado, e, se for o caso, para qual usuário.
- **consultarEmpréstimosUsuario:** Solicita a lista de livros que estão emprestados ao usuário *"João"*, informando também o status de cada empréstimo (por exemplo, se está no prazo ou em atraso).
- **consultarEmpréstimosLivro:** Requisita o histórico de empréstimos do livro *"Golang Programming"*, incluindo informações como usuário, data de empréstimo e data de devolução (se houver).

Cada uma dessas funções é executada de forma independente, utilizando conexões RPC próprias, e os resultados de cada chamada são exibidos no terminal com `fmt.Println`. Embora as chamadas estejam em sequência, não há dependência direta entre os retornos, permitindo testar e validar individualmente o comportamento de cada operação do sistema.

3.2 `server.go`

3.2.1 Função `getLivroID`

```go
func getLivroID(titulo string) (int, error) {
	row := datastore.DB().QueryRow("SELECT id FROM livros WHERE titulo = ?", titulo)

	var id int
	err := row.Scan(&id)
	if err != nil {
		return 0, fmt.Errorf("livro não encontrado")
	}

	return id, nil
}
```

Figura 6: Implementação da função `getLivroID`.

A função `getLivroID` tem como propósito buscar, no banco de dados, o identificador numérico (`id`) de um livro a partir de seu título. Trata-se de uma função auxiliar que realiza uma consulta SQL do tipo `SELECT`.

Sua lógica está organizada em três etapas:

- **Execução da consulta:** O método `QueryRow` é utilizado para executar a instrução SQL `SELECT id FROM livros WHERE titulo = ?`, substituindo o caractere `?` pelo título informado como argumento. A função `DB()` pertence ao módulo de acesso ao banco de dados.
- **Leitura do resultado:** O método `Scan` é empregado para extrair o valor da coluna `id` retornada pela consulta. Caso não haja correspondência, será retornado um erro indicando que o livro não foi encontrado.
- **Retorno do valor:** Se a operação ocorrer com sucesso, a função retorna o valor inteiro correspondente ao identificador do livro, junto a um valor nulo para o erro.

Essa função é fundamental para operações que exigem a manipulação da chave primária de livros no banco, servindo como ponto de integração entre comandos de alto nível (como empréstimos e devoluções) e a estrutura relacional de armazenamento.

3.2.2 Função `RealizarEmpréstimo`

```go
func (s *Server) RealizarEmpréstimo(args struct {
        Usuario        string
        Livro          string
        DataEmpréstimo string
}, reply *string) error {
        livroID, err := getLivroID(args.Livro)
        if err != nil {
                *reply = err.Error()
                return nil
        }

        err = datastore.RealizarEmprestimo(args.Usuario, livroID, args.DataEmpréstimo)
        if err != nil {
                *reply = err.Error()
                return nil
        }

        *reply = "Sucesso"
        return nil
}
```

Figura 7: Implementação da função `RealizarEmpréstimo` no servidor RPC.

A função `RealizarEmpréstimo` está definida no lado do servidor e é responsável por tratar remotamente uma solicitação de empréstimo de livro feita pelo cliente via chamada RPC. Ela recebe um conjunto de argumentos estruturados contendo o nome do usuário, o título do livro e a data do empréstimo, e responde com uma mensagem de sucesso ou erro.

Sua lógica é composta pelas seguintes etapas:

- **Obtenção do ID do livro:** Utiliza-se a função auxiliar `getLivroID` para obter o identificador numérico do livro com base no título informado. Caso o livro não seja encontrado, a função retorna uma mensagem de erro apropriada ao cliente.
- **Registro do empréstimo:** Em seguida, a função chama `RealizarEmprestimo` do módulo `datastore`, passando o nome do usuário, o ID do livro e a data. Se ocorrer alguma falha durante o processo de gravação no banco de dados, o erro é capturado e enviado de volta ao cliente.
- **Resposta de sucesso:** Se ambas as etapas anteriores forem executadas corretamente, a função define a resposta como `"Sucesso"`, confirmando que o empréstimo foi registrado com êxito.

A função é projetada para retornar sempre um erro nulo ao cliente RPC, uma vez que a resposta efetiva (sucesso ou mensagem de erro) é enviada por meio do ponteiro `*reply`, garantindo assim compatibilidade com a interface RPC do pacote `net/rpc`.

3.2.3 Função `DevolverLivro`

```
func (s *Server) DevolverLivro(args struct {
        Usuario string
        Livro   string
}, reply *string) error {
        livroID, err := getLivroID(args.Livro)
        if err != nil {
                *reply = err.Error()
                return nil
        }

        err = datastore.DevolverLivro(args.Usuario, livroID)
        if err != nil {
                *reply = err.Error()
                return nil
        }

        *reply = "Sucesso"
        return nil
}
```

Figura 8: Implementação da função `DevolverLivro` no servidor RPC.

A função `DevolverLivro` é executada no servidor e tem como responsabilidade processar uma solicitação de devolução de livro enviada por um cliente via chamada RPC. Os dados recebidos incluem o nome do usuário e o título do livro a ser devolvido.

Sua lógica segue três etapas principais:

- **Identificação do livro:** A função invoca `getLivroID` para obter o identificador do livro a partir de seu título. Se o livro não for encontrado no banco de dados, a mensagem de erro correspondente é retornada ao cliente por meio da variável `*reply`.
- **Execução da devolução:** Com o ID do livro obtido, a função chama o método `DevolverLivro` do pacote `datastore`, responsável por atualizar o status de disponibilidade do livro no banco. Caso ocorra alguma falha durante esse processo, a mensagem de erro é igualmente repassada ao cliente.
- **Confirmação da operação:** Se nenhuma falha for detectada, a resposta atribuída a `*reply` será a string `"Sucesso"`, indicando que a devolução foi processada corretamente.

Assim como em outras funções servidoras, o valor retornado é sempre `nil`, e eventuais

erros são comunicados exclusivamente por meio do ponteiro de resposta.

3.2.4 Função `ConsultarLivro`

```go
func (s *Server) ConsultarLivro(titulo string, reply *string) error {
        livroID, err := getLivroID(titulo)
        if err != nil {
                *reply = err.Error()
                return nil
        }

        resultado, err := datastore.ConsultarLivro(livroID)
        if err != nil {
                *reply = err.Error()
                return nil
        }

        *reply = resultado
        return nil
}
```

Figura 9: Implementação da função `ConsultarLivro` no servidor RPC.

A função `ConsultarLivro` é executada no lado do servidor e tem como objetivo atender a requisições RPC que buscam verificar a disponibilidade de um determinado livro no sistema, a partir de seu título.

A lógica da função segue as etapas descritas abaixo:

- **Recuperação do ID do livro:** Utiliza-se a função auxiliar `getLivroID` para localizar o identificador numérico associado ao título informado. Caso o livro não seja encontrado, uma mensagem de erro é atribuída à variável `*reply` e a execução é encerrada.
- **Consulta ao banco de dados:** Com o ID obtido, a função chama o método `ConsultarLivro` do módulo `datastore`, o qual realiza a consulta propriamente dita sobre a situação do livro (por exemplo, se está disponível ou emprestado, e para quem).
- **Envio da resposta ao cliente:** O resultado obtido é atribuído diretamente à variável `*reply`, sendo então transmitido ao cliente como resposta da chamada RPC.

A função retorna sempre `nil`, delegando a comunicação de falhas exclusivamente à variável de resposta `*reply`, como é padrão nas funções do servidor.

3.2.5 Função `ConsultarEmpréstimosUsuario`

```
func (s *Server) ConsultarEmpréstimosUsuario(usuario string, reply *string) error {
        resultados, err := datastore.ConsultarEmprestimosUsuario(usuario)
        if err != nil {
                *reply = err.Error()
                return nil
        }

        *reply = fmt.Sprintf("%v", resultados)
        return nil
}
```

Figura 10: Implementação da função `ConsultarEmpréstimosUsuario` no servidor RPC.

A função `ConsultarEmpréstimosUsuario` está definida no lado do servidor e é responsável por atender a chamadas RPC que solicitam a listagem de todos os empréstimos ativos relacionados a um determinado usuário.

Sua lógica é composta por duas etapas principais:

- **Consulta ao banco de dados:** O método `ConsultarEmprestimosUsuario` do módulo `datastore` é invocado com o nome do usuário como parâmetro. Caso ocorra um erro durante a consulta, uma mensagem descritiva é atribuída à variável `*reply`, e a execução é encerrada.
- **Formatação da resposta:** Os resultados obtidos são convertidos para uma representação textual utilizando `fmt.Sprintf`, e atribuídos à variável de resposta `*reply`, que será enviada de volta ao cliente solicitante.

Essa função retorna sempre `nil`, uma vez que a resposta é tratada exclusivamente via o ponteiro de retorno, respeitando o padrão das funções RPC implementadas no projeto.

3.2.6 Função `main` (servidor)

```go
func main() {
	// Inicializa o banco de dados SQLite
	if err := datastore.InitDB(); err != nil {
		log.Fatalf("Erro ao inicializar banco: %v", err)
	}

	// Registra o servidor RPC
	rpc.Register(new(Server))
	listener, err := net.Listen("tcp", ":1234")
	if err != nil {
		log.Fatal("Erro ao escutar na porta 1234:", err)
	}

	fmt.Println("Servidor escutando na porta 1234")
	for {
		conn, err := listener.Accept()
		if err != nil {
			log.Print("Erro ao aceitar conexão:", err)
			continue
		}
		go rpc.ServeConn(conn)
	}
}
```

Figura 11: Função `main` do servidor: inicialização do banco e escuta de conexões RPC.

A função `main` do servidor é responsável por inicializar os componentes essenciais da aplicação: o banco de dados local e o servidor RPC, que escuta requisições na porta `1234`.

Sua execução está organizada em três etapas principais:

- **Inicialização do banco de dados SQLite:** A função `datastore.InitDB()` é chamada para preparar a estrutura de tabelas e conexões do banco local. Caso ocorra qualquer erro nesse processo, a execução é interrompida com uma mensagem exibida via `log.Fatalf`.
- **Registro do servidor RPC e escuta de porta:** O tipo `Server` é registrado junto ao pacote `net/rpc` como provedor de métodos remotos. Em seguida, o servidor é configurado para escutar conexões TCP na porta `1234`. Se essa porta estiver indisponível ou ocorrer falha na escuta, o sistema também aborta a execução com uma mensagem crítica.
- **Aceitação de conexões:** Um loop infinito aguarda novas conexões com `listener.Accept()`. Para cada conexão aceita, é criada uma nova goroutine para lidar com o cliente

utilizando o método `rpc.ServeConn(conn)`, garantindo assim a concorrência no atendimento das requisições RPC.

Essa função é o ponto central de ativação do servidor e estabelece toda a infraestrutura necessária para que o sistema possa atender às chamadas remotas originadas pelos clientes.

3.3 `datastore.go`

O arquivo `datastore.go` concentra toda a lógica de persistência de dados da aplicação. Ele é responsável por definir as estruturas associadas aos livros e empréstimos, inicializar o banco de dados SQLite, e implementar as operações fundamentais que manipulam essas entidades. A seguir, são descritas as principais seções desse módulo:

- **Importações e variáveis globais:** O pacote importa bibliotecas para manipulação de banco de dados SQL e tratamento de tempo, utilizando o driver `modernc.org/sqlite` para acesso ao SQLite. Uma variável global `db` do tipo `*sql.DB` é utilizada para manter a conexão ativa com o banco.
- **Definições de estrutura:** Duas `structs` são definidas: `Livro`, que contém os campos `ID`, `Titulo` e `Disponivel`; e `Emprestimo`, que registra `ID`, `Usuario`, `LivroID`, `DataEmprestimo` e `DataDevolucao`.
- **Inicialização do banco de dados:** A função `InitDB` abre a conexão com o arquivo `biblioteca.db`, cria as tabelas `livros` e `emprestimos` caso ainda não existam, e insere um livro de exemplo na tabela. Esta função é chamada no início da execução do servidor.
- **Acesso à conexão ativa:** A função `DB()` retorna o ponteiro `*sql.DB`, permitindo que outros pacotes tenham acesso à conexão com o banco de dados de maneira controlada.
- **Função `ObterLivro`:** Realiza uma consulta para recuperar os dados de um livro a partir de seu `id`, preenchendo a estrutura `Livro` com as informações obtidas.
- **Função `RealizarEmprestimo`:** Verifica a disponibilidade do livro e, caso esteja disponível, inicia uma transação SQL para:
  - Inserir um novo empréstimo na tabela `emprestimos`;
  - Atualizar o status do livro para `disponivel = false`.

  Caso ocorra qualquer erro durante a transação, ela é revertida.
- **Função `DevolverLivro`:** Localiza o empréstimo ativo entre o usuário e o livro, exclui o registro da tabela `emprestimos` e atualiza o campo `disponivel` do livro para `true`, também utilizando uma transação para garantir consistência.

- **Função** `ConsultarLivro`**:** Retorna uma mensagem indicando se o livro está disponível ou, caso esteja emprestado, apresenta o nome do usuário e a data do empréstimo.
- **Função** `ConsultarEmprestimosUsuario`**:** Retorna uma lista textual com os livros atualmente emprestados a um determinado usuário, incluindo a data do empréstimo e o status (por exemplo, *"no prazo"* ou *"atrasado"*).
- **Função** `ConsultarEmprestimosLivro`**:** Fornece o histórico de empréstimos de um livro específico, apresentando os nomes dos usuários e as datas correspondentes aos registros.

O módulo `datastore.go` é essencial para abstrair o acesso ao banco de dados e garantir que as demais partes da aplicação interajam com os dados de forma segura, transacional e coesa.

4 Conclusão

O desenvolvimento da aplicação distribuída utilizando o paradigma cliente/servidor com chamadas de procedimento remoto (RPC) demonstrou-se eficaz na consolidação dos principais conceitos estudados em Sistemas Distribuídos. Por meio da implementação de um sistema de empréstimo de livros, foi possível aplicar de forma prática os conhecimentos relacionados à comunicação remota entre processos, concorrência, modularização de software e persistência de dados.

A separação entre cliente e servidor, bem como a adoção de boas práticas arquiteturais inspiradas na Clean Architecture, favoreceu a organização do código, a clareza das responsabilidades e a facilidade de manutenção. O uso da linguagem Go (Golang) proporcionou uma experiência fluida, especialmente devido ao seu suporte nativo à concorrência e ao pacote net/rpc, que facilitou a implementação das chamadas remotas.

A interface com o banco de dados SQLite garantiu persistência eficiente e compatível com os requisitos de um projeto acadêmico. Além disso, as funções implementadas permitiram simular de forma realista operações típicas de bibliotecas digitais, como empréstimos, devoluções e consultas — tudo realizado via RPC, validando o funcionamento do sistema como um todo.

Portanto, o projeto atendeu aos objetivos propostos, proporcionando um ambiente funcional que integra teoria e prática no contexto de aplicações distribuídas. A aplicação pode ser estendida futuramente com melhorias como autenticação de usuários, interface gráfica e suporte a múltiplos clientes concorrentes.

5 Referências

Referências

[1] Tanenbaum, A. S.; Van Steen, M. *Distributed Systems: Principles and Paradigms*. 2ª ed. Pearson Education, 2007.

[2] Go Documentation. *net/rpc package*. Disponível em: `https://pkg.go.dev/net/rpc`. Acesso em: 03 jun. 2025.

[3] Go Documentation. *database/sql package*. Disponível em: `https://pkg.go.dev/database/sql`. Acesso em: 03 jun. 2025.

[4] Martin, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson Education, 2017.

[5] Larson, R. R.; Kugler, J. A. *Using SQLite*. O'Reilly Media, 2011.