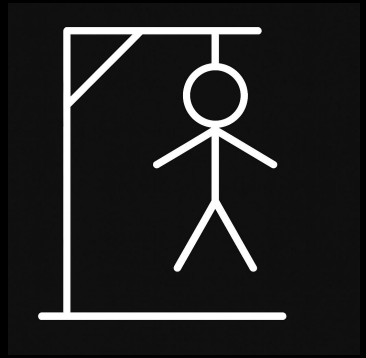
Jogo da Forca - Aplicação Cliente-Servidor com RabbitMQ e gRPC

Amanda Quirino Rodrigues Dos Santos (aqrs)
Emanoel Rafael Melo Ferreira da Silva (ermfs)
Rebecca Lima Sousa (rls7)



Centro de Informática
UFPE

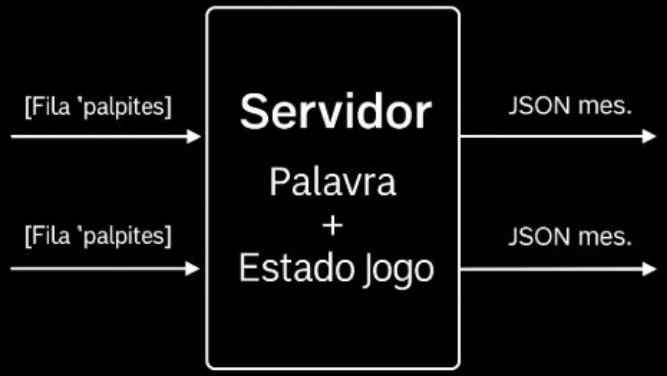
Objetivo

- Desenvolver o jogo da forca multiplayer com comunicação entre clientes e servidor
- Aplicar conceitos de sistemas distribuídos e mensageria assíncrona
- Utilizar a linguagem Go com o RabbitMQ e gRPC

Tecnologias Utilizadas

- Linguagem: Go
- Broker de Mensagens: RabbitMQ
- Protocolo: AMQP
- Editor: Visual Studio Code
- Execução local com Docker
- gRPC

RabbitMQ


[Fila 'palpites]
[Fila 'palpites]
Servidor
Palavra
+
Estado Jogo
JSON mes.
JSON mes.

Arquitetura da Aplicação (Cliente - Servidor)

- Um servidor central controla o estado do jogo
- Dois clientes se conectam via filas do RabbitMQ
- O servidor envia o estado do jogo aos clientes
- Os clientes enviam palpites ao servidor por uma fila compartilhada

Funcionamento do Jogo

- O servidor define uma palavra secreta
- Os clientes se identificam como cliente1 e cliente2
- Os palpites são enviados por turnos e processados no servidor
- O estado do jogo é atualizado e enviado a ambos os clientes
- Vitória ou derrota é determinada com base no número de erros ou acertos

Comunicação Cliente-Servidor: Início

- Dial: Estabelece conexão com o servidor
- Channel: Canal de comunicação que permite a comunicação entre o cliente e o RabbitMQ
- QueueDeclare: Fila onde as mensagens serão consumidas pelo cliente e publicadas pelo servidor. Caso a fila não exista ela será adicionada automaticamente.
- Consume: Inicia o consumo de mensagens

```go
conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
failOnError(err, "Conexão RabbitMQ")
defer conn.Close()

ch, err := conn.Channel()
failOnError(err, "Canal")
defer ch.Close()

ch.QueueDeclare(clienteID, false, false, false, false, nil)
msgs, err := ch.Consume(clienteID, "", true, false, false, false, nil)
failOnError(err, "Consumo")
```

Comunicação Cliente-Servidor: Envio de Palpites

- Publish: Envia a mensagem para a fila criada pelo servidor
  a. “”: Indica que a mensagem será enviada diretamente para a fila em vez de uma troca (exchange)
  b. “palpites”: Nome da fila para onde o palpite será enviado
  c. false: Indica que a fila deve ser durável
  d. false: Indica que a fila não será exclusiva
  e. amqp.Publishing: estrutura onde ficam os dados e metadados da mensagem

```
err := ch.Publish("", "palpites", false, false, amqp.Publishing{
    ContentType: "text/plain",
    Body:        []byte(letra),
    Headers: amqp.Table{
        "player": clienteID,
    },
})
```

Comunicação Cliente-Servidor: Envios do Servidor

- Publish: Envia a mensagem para a fila criada pelo servidor
  a. “”: Indica que a mensagem será enviada diretamente para a fila em vez de uma troca (exchange)
  b. cliente: Variável contendo o nome da fila ao qual a mensagem será enviada
  c. false: Indica que a fila deve ser durável
  d. false: Indica que a fila não será exclusiva
  e. amqp.Publishing: Estrutura onde ficam as informações da mensagem a ser publicada

```
ch.Publish("", cliente, false, false, amqp.Publishing{
    ContentType: "application/json",
    Body:        estadoBytes,
})
```

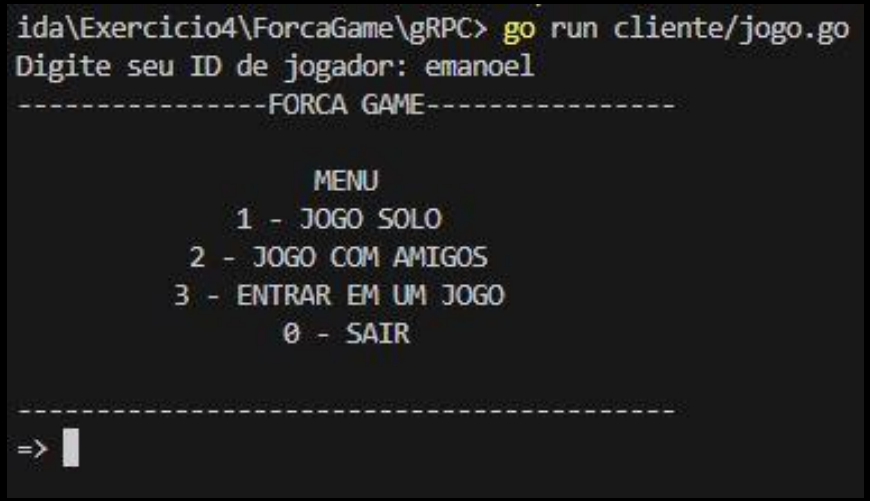
gRPC

Arquitetura da Aplicação (Cliente - Servidor)

- Servidor
  a. Gerencia a criação do jogo
  b. Administra os jogos em andamento
  c. Gerencia a sincronização
- Cliente
  a. Se conectam ao servidor para entrar em um jogo
  b. Obtém iterativamente o estado do jogo atual ao qual participam
  c. Jogam enviando palpites
- Contrato
  a. Define a interface de serviços e estruturas utilizadas na comunicação via gRPC

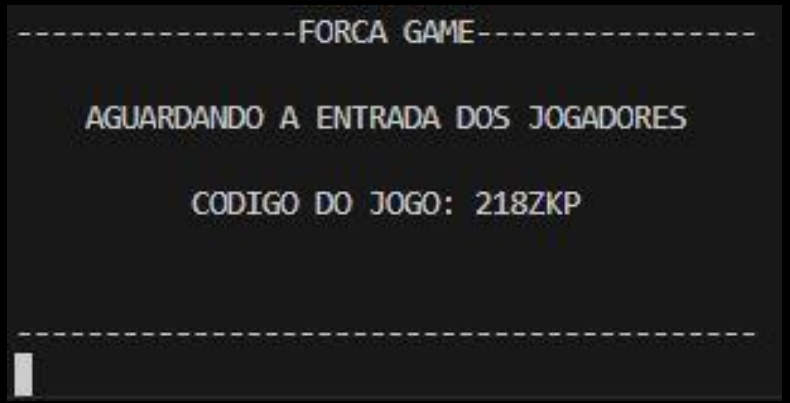
Funcionamento do Jogo

- O cliente se conecta ao servidor para iniciar um jogo.
- O cliente escolhe a modalidade do jogo.



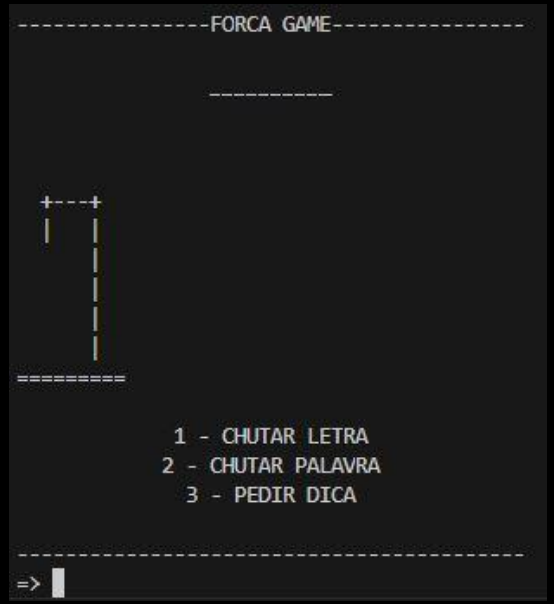
Funcionamento do Jogo

- Caso a modalidade do jogo for multiplayer o cliente compartilha o código do jogo para outras pessoas entrarem.



Funcionamento do Jogo

- Quando o jogo se inicia os jogadores podem enviar palpite de letra, da palavra completa ou ainda pedir uma dica.
- A ordem de jogada é a ordem de ingresso no jogo.

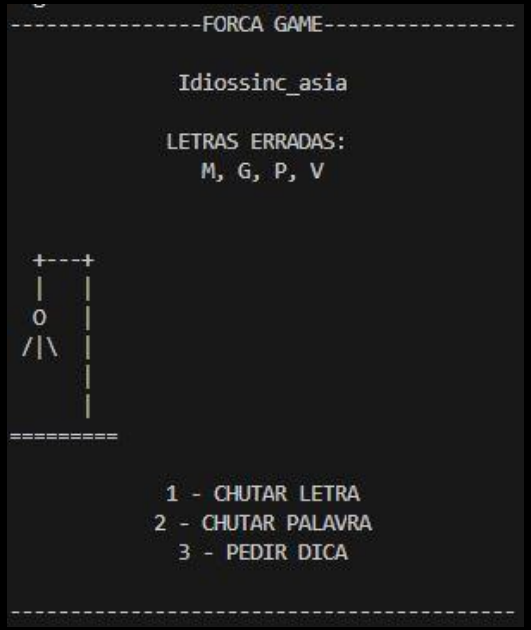


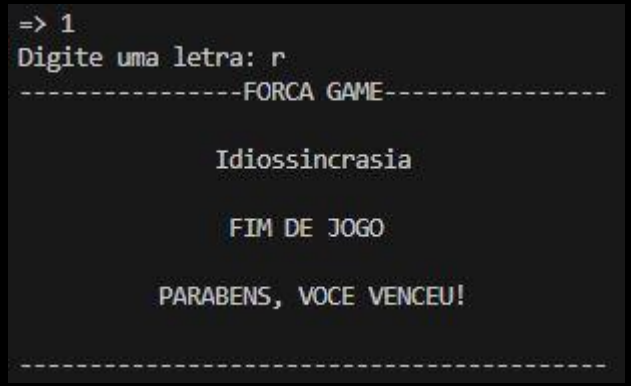
Funcionamento do Jogo

- Se errar 6 vezes o jogador perde

Funcionamento do Jogo

- Se ao realizar alguma jogada o jogador revelar a palavra inteira ele vence.





Código - Contrato Serviços

```
service GameService {
  rpc CriarJogo (CriarJogoRequest) returns (CriarJogoResponse);
  rpc EntrarJogo (EntrarJogoRequest) returns (EntrarJogoResponse);

  rpc PalpitarLetra (PalpitarLetraRequest) returns (AtualizacaoResponse);
  rpc PalpitarPalavra (PalpitarPalavraRequest) returns (AtualizacaoResponse);
  rpc PedirDica (DicaRequest) returns (AtualizacaoResponse);

  rpc ObterEstado (EstadoRequest) returns (AtualizacaoResponse);
}
```

Código - Contrato Mensagens

```
message CriarJogoRequest {···
}

message CriarJogoResponse {···
}

message EntrarJogoRequest {···
}

message EntrarJogoResponse {···
}

message PalpitarLetraRequest {···
}

message PalpitarPalavraRequest {···
}

message DicaRequest {···
}

message EstadoRequest {···
}

message AtualizacaoResponse {···
}
```

```
message EstadoRequest {
  string codigo_jogo = 1;
  string jogador_id = 2;
}

message AtualizacaoResponse {
  string palavra_visivel = 1;
  repeated string letras_erradas = 2;
  int32 erros_jogador = 3;
  string jogador_da_vez = 4;
  string mensagem = 5;
  int32 jogo_status = 6;
  string vencedor_id = 7;
}
```

Código - Servidor


```
type GameServer struct {···
}

func NewGameServer() *GameServer {···
}

func (s *GameServer) monitorarTimeout(codigo string) {···
}

func (s *GameServer) CriarJogo(ctx context.Context, req *pb.CriarJogoRequest) (*pb.CriarJogoResponse, error) {···
}

func (s *GameServer) EntrarJogo(ctx context.Context, req *pb.EntrarJogoRequest) (*pb.EntrarJogoResponse, error) {···
}

func (s *GameServer) PalpitarLetra(ctx context.Context, req *pb.PalpitarLetraRequest) (*pb.AtualizacaoResponse, error) {···
}

func (s *GameServer) PalpitarPalavra(ctx context.Context, req *pb.PalpitarPalavraRequest) (*pb.AtualizacaoResponse, error) {···
}

func (s *GameServer) PedirDica(ctx context.Context, req *pb.DicaRequest) (*pb.AtualizacaoResponse, error) {···
}

func (s *GameServer) ObterEstado(ctx context.Context, req *pb.EstadoRequest) (*pb.AtualizacaoResponse, error) {···
}
```

Código - Servidor

```go
func NewGameServer() *GameServer {
    return &GameServer{
        jogos: make(map[string]*Jogo),
    }
}
```

```go
lis, err := net.Listen("tcp", ":50051")
if err != nil {
    log.Fatalf("Erro ao escutar: %v", err)
}

grpcServer := grpc.NewServer()
gameSrv := NewGameServer()
pb.RegisterGameServiceServer(grpcServer, gameSrv)

fmt.Println("Servidor gRPC rodando na porta 50051...")
if err := grpcServer.Serve(lis); err != nil {
    log.Fatalf("Erro ao rodar servidor: %v", err)
}
```

Inicialização do servidor como um servidor gRPC

Código - Servidor

```go
func (s *GameServer) PalpitarLetra(ctx context.Context, req *pb.PalpitarLetraRequest) (*pb.AtualizacaoResponse, error) {
    s.mu.Lock()
    defer s.mu.Unlock()

    jogo, existe := s.jogos[req.CodigoJogo]
    if !existe || jogo.Status == FINALIZADO {
        jogo.UltimaJogada = time.Now()
        return &pb.AtualizacaoResponse{Mensagem: "Jogo não encontrado ou já finalizado"}, nil
    }

    if jogo.JogadorDaVez != req.JogadorId {
        jogo.UltimaJogada = time.Now()
        return &pb.AtualizacaoResponse{Mensagem: "Não é sua vez"}, nil
    }
```

Implementação dos serviços do contrato

Código - Servidor

```go
func trocarTurno(j *Jogo) {
    jogadores := j.Jogadores
    atual := j.JogadorDaVez
    var idx int
    for i, id := range jogadores {
        if id == atual {
            idx = i
            break
        }
    }

    for {
        idx = (idx + 1) % len(jogadores)
        if !j.Eliminados[jogadores[idx]] {
            j.JogadorDaVez = jogadores[idx]
            break
        }
    }
}
```

Função que define o próximo jogador

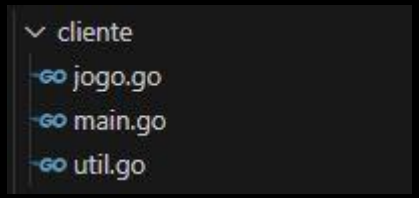
Código - Servidor

```go
func (s *GameServer) monitorarTimeout(codigo string) {
    for {
        time.Sleep(5 * time.Second)

        s.mu.Lock()
        jogo, ok := s.jogos[codigo]
        if !ok || jogo.Status != EM_CURSO {
            s.mu.Unlock()
            return
        }

        if time.Since(jogo.UltimaJogada) > 60*time.Second {
            jogador := jogo.JogadorDaVez
            fmt.Printf("Jogador %s foi eliminado por inatividade\n", jogador)
            jogo.Eliminados[jogador] = true
            jogo.Erros[jogador] = 6

            restantes := jogadoresRestantes(jogo)
            if len(restantes) == 1 {
                jogo.Status = FINALIZADO
                jogo.VencedorID = restantes[0]
                s.mu.Unlock()
                return
            } else if len(restantes) == 0 {
                jogo.Status = FINALIZADO
                jogo.VencedorID = "nil"
                s.mu.Unlock()
                return
            }
```

Problema - Usuários que ficarem muito tempo sem jogar bloqueiam outros usuários de jogar.
Solução - Usuários que ficarem inativos por muito tempo são eliminados

```
----------FORCA GAME----------

          _________

          FIM DE JOGO

          VOCE PERDEU!

  +---+
  |   |
  O   |
 /|\  |
 / \  |
      |
=========

------------------------------------
```

Código - Cliente




```go
type Jogo struct {
    Codigo          string
    PalavraVisivel  []rune
    Erros           int
    DicaUsada       bool
    LetrasErradas   []string
    JogadorDaVez    string
    VencedorID      string
    Status          int
}
```

Arquivo jogo.go

```go
const widthGame = 44
const MENU = 1
const AGUARDANDO_JOGADORES = 2
const INGRESSO = 3
const EM_ANDAMENTO = 4
const FIM_DE_JOGO = 5

const NAO_INICIADO = 0
const PENDENTE_JOGADORES = 1
const EM_CURSO = 2
const FINALIZADO = 3

func desenharBoneco(erros int) {···
}

func printLinhaGame(msg string, placeholder rune) {···
}

func printGame() string {···
}
```

Arquivo util.go

Código - Cliente

```go
conn, err := grpc.Dial("localhost:50051", grpc.WithInsecure())
if err != nil {
    log.Fatalf("Erro ao conectar: %v", err)
}
defer conn.Close()
client := pb.NewGameServiceClient(conn)
```

Abertura da conexão gRPC

```go
resp, err := client.CriarJogo(ctx, &pb.CriarJogoRequest{
    JogadorId: jogadorId,
    Solo:      true,
})
if err != nil {
    log.Println("Erro:", err)
    continue
}
codigoJogo = resp.CodigoJogo
jogo.Codigo = resp.CodigoJogo
gameStage = EM_ANDAMENTO
```

```go
fmt.Print("Digite uma letra: ")
letra, _ := reader.ReadString('\n')
letra = strings.TrimSpace(letra)

_, err := client.PalpitarLetra(ctx, &pb.PalpitarLetraRequest{
    JogadorId:  jogadorId,
    CodigoJogo: codigoJogo,
    Letra:      letra,
})
if err != nil {
    log.Println("Erro:", err)
    continue
}
```

Utilização dos serviços por meio de chamadas gRPC

Código - Cliente

```
if gameStage == AGUARDANDO_JOGADORES || gameStage == EM_ANDAMENTO {
    resp, err := client.ObterEstado(ctx, &pb.EstadoRequest{
        CodigoJogo: codigoJogo,
        JogadorId:  jogadorId,
    })
    if err != nil {
        log.Println("Erro:", err)
        continue
    }

    jogo.PalavraVisivel = []rune(resp.PalavraVisivel)
    jogo.Erros = int(resp.ErrosJogador)
    jogo.LetrasErradas = resp.LetrasErradas
    jogo.JogadorDaVez = resp.JogadorDaVez
    jogo.Status = int(resp.JogoStatus)
    jogo.VencedorID = resp.VencedorId
    if jogo.Status == FINALIZADO {
        gameStage = FIM_DE_JOGO
    }
}
```

Obtenção do status atual do jogo

Obrigado!