Relatório do Projeto

Sistema de Histórias Interativas Multijogador com RPC

Disciplina: Sistemas Distribuídos e Tecnologias (CC5SDT) **Semestre:** 2025-2 **Professor:** Rafael Keller Tesser **Alunos:** Jorge Daniel Ristow de Camargo 2521679

Sumário

---

1. Introdução ao Módulo RPC

1.1 JSON-RPC 2.0

JSON-RPC 2.0 é um protocolo de chamada de procedimento remoto (RPC) stateless e leve que utiliza JSON (JavaScript Object Notation) como formato de dados. É especificado na RFC 4627 (https://www.jsonrpc.org/specification) e permite que um cliente solicite a execução de procedimentos em um servidor remoto de forma simples e eficiente.

Características Principais

- **Stateless:** Cada requisição é independente e autocontida
- **Formato JSON:** Utiliza JSON para serialização de dados, facilitando interoperabilidade
- **Sobre HTTP/HTTPS:** Pode ser transportado via HTTP, facilitando atravessar firewalls
- **Leve e Simples:** Especificação minimalista sem complexidade desnecessária
- **Suporte a Batch:** Permite múltiplas chamadas em uma única requisição
- **Tratamento de Erros Padronizado:** Códigos de erro bem definidos

Estrutura de uma Requisição JSON-RPC 2.0

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "login",
  "params": {
    "username": "usuario1",
    "password": "senha123"
  }
}
```

Campos:

- `jsonrpc`: String indicando a versão do protocolo (sempre "2.0")
- `id`: Identificador único da requisição (usado para correlacionar resposta)
- `method`: Nome do método a ser invocado no servidor
- `params`: Parâmetros do método (objeto ou array)

Estrutura de uma Resposta de Sucesso

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    "user": {
      "id": "user_123",
      "username": "usuario1",
      "role": "USER"
    },
    "expiresIn": 86400
  }
}
```

Estrutura de uma Resposta de Erro

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32001,
    "message": "Credenciais inválidas",
    "data": {
      "reason": "Usuário ou senha incorretos"
    }
  }
}
```

Códigos de Erro Padronizados

Código	Nome	Descrição
-32700	Parse error	JSON inválido
-32600	Invalid Request	Requisição mal formada
-32601	Method not found	Método não existe
-32602	Invalid params	Parâmetros inválidos
-32603	Internal error	Erro interno do servidor
-32000 a -32099	Server error	Erros customizados da aplicação

1.2 Justificativa da Escolha

A escolha do **JSON-RPC 2.0** como protocolo de comunicação para o sistema de histórias interativas foi baseada em diversos fatores técnicos e práticos:

1. **Simplicidade e Clareza**

- Protocolo minimalista com especificação clara e concisa
- Fácil de implementar tanto no cliente quanto no servidor
- Curva de aprendizado suave para desenvolvedores
- Menos propenso a erros de implementação comparado a protocolos mais complexos

2. **Interoperabilidade**

- JSON é amplamente suportado em todas as linguagens modernas
- Facilita integração com diferentes plataformas e tecnologias
- Permite clientes em navegadores web (JavaScript), mobile, desktop, etc.
- Não requer bibliotecas especializadas complexas

3. **Leveza e Performance**

- Overhead mínimo de protocolo

- Payload compacto em comparação com XML-RPC ou SOAP
- Parsing eficiente de JSON
- Adequado para aplicações em tempo real com votação colaborativa

4. **Compatibilidade Web**

- Funciona nativamente sobre HTTP/HTTPS
- Facilita atravessar firewalls corporativos
- Compatível com CORS para aplicações web distribuídas
- Integração natural com APIs REST quando necessário

5. **Adequação ao Domínio**

- Sistema multiplayer requer múltiplas chamadas de procedimento (votar, criar personagem, enviar mensagem, etc.)
- RPC é mais natural que REST para operações procedurais
- Facilita implementação de long polling para atualizações em tempo real
- Permite agrupar múltiplas operações relacionadas

6. **Ferramental e Documentação**

- Integração fácil com Swagger/OpenAPI para documentação
- Suporte nativo em frameworks modernos (Fastify, Express)
- Bibliotecas maduras e testadas disponíveis
- Comunidade ativa e ampla base de conhecimento

1.3 Comparação com Outras Tecnologias

JSON-RPC 2.0 vs Java RMI

Aspecto	JSON-RPC 2.0	Java RMI
Linguagem	Agnóstico	Java apenas
Transporte	HTTP/HTTPS	Protocolo próprio (JRMP)
Serialização	JSON (texto)	Serialização binária Java
Firewall	Atravessa facilmente	Problemas com firewalls
Complexidade	Simples	Complexo (stubs, skeletons)
Web Browser	□ Suporta	□ Não suporta

Veredito: JSON-RPC 2.0 vence em interoperabilidade e simplicidade.

JSON-RPC 2.0 vs gRPC

Aspecto	JSON-RPC 2.0	gRPC
Formato	JSON (texto)	Protocol Buffers (binário)
Performance	Boa	Excelente
Browser	□ Nativo	⚠ Requer gRPC-Web
Debugging	Fácil (texto legível)	Difícil (binário)

Aspecto	JSON-RPC 2.0	gRPC
Streaming	□ Não	□ Sim (bidirecional)
Setup	Simples	Complexo (protobuf, geração de código)

Veredito: Para aplicação web com requisitos moderados de performance, JSON-RPC 2.0 é mais simples e adequado.

JSON-RPC 2.0 vs XML-RPC

Aspecto	JSON-RPC 2.0	XML-RPC
Formato	JSON	XML
Tamanho	Compacto	Verboso
Parsing	Rápido	Lento
Legibilidade	Alta	Baixa
Adoção	Moderna	Legado

Veredito: JSON-RPC 2.0 supera XML-RPC em todos os aspectos para aplicações modernas.

JSON-RPC 2.0 vs Python RpyC/Pyro

Aspecto	JSON-RPC 2.0	RpyC/Pyro
Linguagem	Agnóstico	Python apenas
Transparência	RPC explícito	RPC transparente
Segurança	JWT, validação manual	Mecanismos próprios
Debugging	Claro	Pode ser confuso
Frontend Web	□ Ideal	□ Requer adaptador

Veredito: JSON-RPC 2.0 é preferível para sistemas heterogêneos com frontend web.

---

2. Desenvolvimento da Aplicação

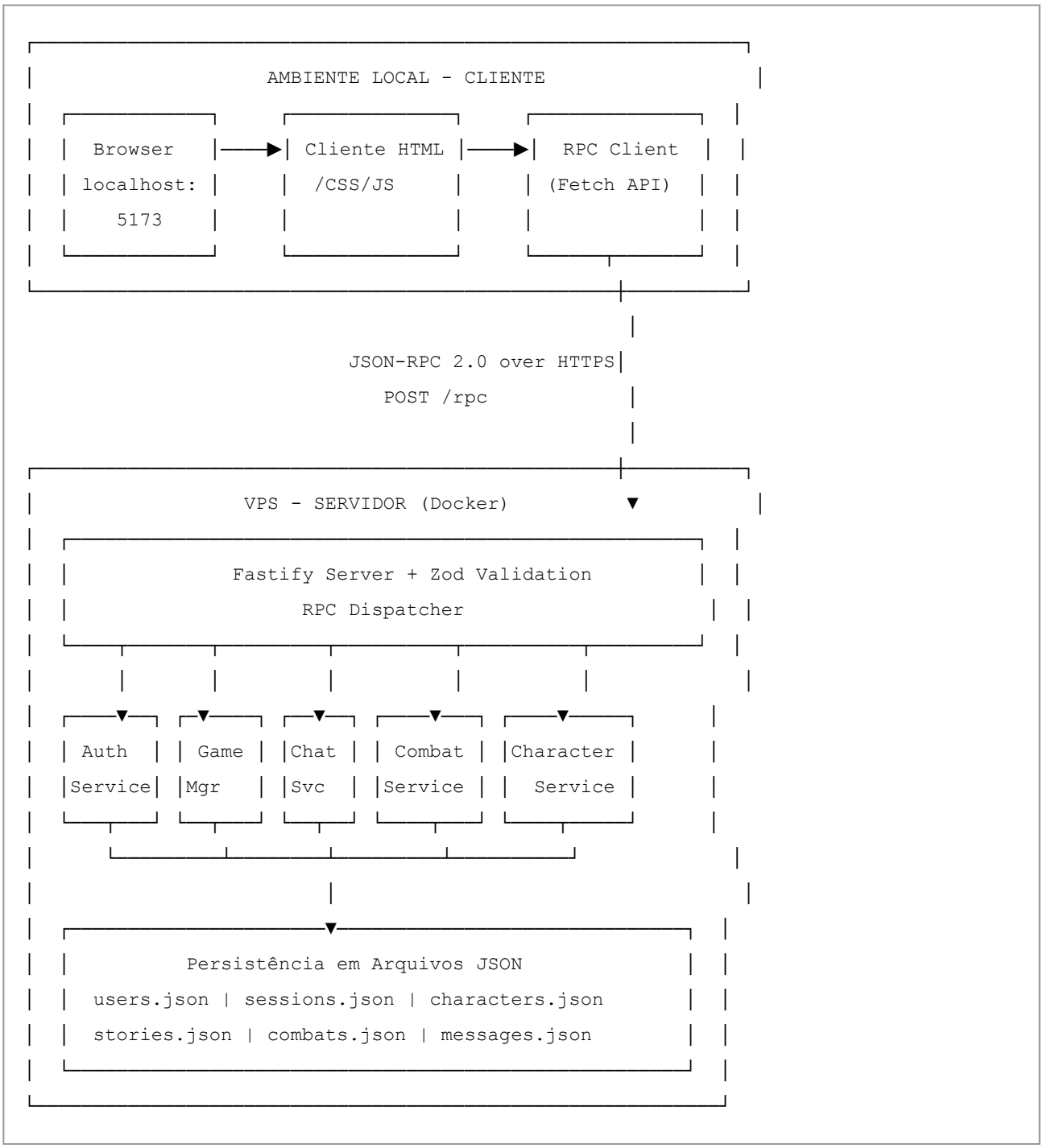
2.1 Visão Geral do Sistema

O **RPC Interactive Stories** é um sistema distribuído de histórias interativas multiplayer que permite a jogadores colaborarem em aventuras estilo "escolha sua própria aventura" inspiradas em D&D (Dungeons & Dragons). O sistema implementa:

- **Autenticação JWT:** Login seguro com tokens de sessão
- **Criação de Personagens D&D:** Sistema completo de atributos, raças, classes e background
- **Sessões Multiplayer:** Salas de jogo com código de acesso
- **Votação Colaborativa:** Decisões por voto com timer automático
- **Sistema de Combate:** Mecânicas D&D com D20, iniciativa, ataques e dano
- **Chat em Tempo Real:** Comunicação entre jogadores via long polling
- **Painel Administrativo:** Gerenciamento de usuários, sessões e histórias
- **Parser de Histórias:** Import de histórias em formato Mermaid

2.2 Arquitetura Distribuída

O sistema utiliza arquitetura **cliente-servidor distribuída pura**:



Fluxo de Comunicação

1. **Cliente** executa `npm start` localmente
2. **Browser** abre em `localhost:5173`
3. **Cliente RPC** se conecta ao servidor VPS via HTTPS
4. Todas as operações (login, criar personagem, votar, chat) são **chamadas RPC**
5. **Servidor** processa requisições, valida com Zod, executa lógica de negócio

6. **Resposta JSON-RPC** retorna ao cliente com resultado ou erro

2.3 Stack Tecnológica

Backend (Servidor VPS)

- **Runtime:** Node.js 18+
- **Framework Web:** Fastify (alta performance, baixa latência)
- **Protocolo RPC:** JSON-RPC 2.0 puro
- **Validação:** Zod schemas (type-safe)
- **Autenticação:** JWT (jsonwebtoken) + bcrypt
- **Documentação:** Swagger UI via @fastify/swagger
- **Container:** Docker + Docker Compose
- **Persistência:** Arquivos JSON (File-based storage)
- **Parser:** Mermaid para histórias em flowchart

Frontend (Cliente Local)

- **Runtime:** Node.js (apenas para servidor de desenvolvimento)
- **Servidor Local:** Express (serve arquivos estáticos)
- **Interface:** HTML5 + CSS3 + Vanilla JavaScript (sem frameworks)
- **RPC Client:** Fetch API (nativo do browser)
- **Estado:** LocalStorage (apenas token JWT)
- **Arquitetura:** Componentes modulares em `src/ui/`

Ferramental de Desenvolvimento

- **Linguagem:** TypeScript (backend), JavaScript ES6+ (frontend)
- **Linter:** ESLint
- **Build:** TypeScript compiler (tsc)
- **Dev Server:** tsx watch (hot reload backend)
- **Versionamento:** Git + GitHub
- **Deploy:** Docker + VPS

2.4 Implementação Backend

Estrutura de Pastas

```
backend/
├── src/
│   ├── models/                    # Zod schemas e tipos
│   │   ├── auth_schemas.ts
│   │   ├── character_schemas.ts
│   │   ├── game_schemas.ts
│   │   ├── combat_schemas.ts
│   │   └── update_schemas.ts
│   ├── services/                  # Lógica de negócio
│   │   ├── auth_service.ts
│   │   ├── character_service.ts
│   │   ├── game_service.ts
│   │   ├── combat_service.ts
│   │   └── vote_service.ts
│   ├── stores/                    # Persistência JSON
│   │   ├── user_store.ts
│   │   ├── session_store.ts
│   │   ├── character_store.ts
│   │   └── combat_store.ts
│   ├── rpc/
│   │   ├── handlers/
│   │   │   ├── rpc_methods/       # Handlers RPC
│   │   │   │   ├── auth_methods.ts
│   │   │   │   ├── game_methods.ts
│   │   │   │   ├── combat_methods.ts
│   │   │   │   └── index.ts       # Registry
│   │   │   └── wrappers/          # REST wrappers (Swagger)
│   │   │       ├── auth_wrappers.ts
│   │   │       ├── game_wrappers.ts
│   │   │       └── combat_wrappers.ts
│   │   ├── openapi/               # Documentação OpenAPI
│   │   │   ├── paths/
│   │   │   │   ├── auth_paths.ts
│   │   │   │   ├── game_paths.ts
│   │   │   │   └── combat_paths.ts
│   │   │   └── registry.ts
│   │   ├── middleware/
│   │   │   ├── auth.ts
│   │   │   ├── cors.ts
│   │   │   └── error.ts
│   │   └── server.ts              # Ponto de entrada
│   └── utils/
│       ├── jwt.ts
```

```
│       └── errors.ts
├── data/                           # Arquivos JSON
├── stories/                        # Histórias Mermaid
├── Dockerfile
├── docker-compose.yml
└── package.json
```

Padrão de Implementação (6 Passos)

Toda funcionalidade backend segue este padrão rigoroso:

1. Definir Schemas Zod (`src/models/{domain}_schemas.ts`)

```
import { z } from 'zod';
import { extendZodWithOpenApi } from '@asteasolutions/zod-to-openapi';

extendZodWithOpenApi(z);

export const LoginParamsSchema = z.object({
  username: z.string().min(3).openapi({
    example: 'usuario1',
    description: 'Nome de usuário'
  }),
  password: z.string().min(6).openapi({
    example: 'senha123',
    description: 'Senha do usuário'
  })
});

export const LoginResponseSchema = z.object({
  token: z.string().openapi({
    example: 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...',
    description: 'JWT token de autenticação'
  }),
  user: UserSchema,
  expiresIn: z.number().openapi({
    example: 86400,
    description: 'Tempo de expiração em segundos (24h)'
  })
});

export type LoginParams = z.infer<typeof LoginParamsSchema>;
export type LoginResponse = z.infer<typeof LoginResponseSchema>;
```

2. Criar Service (`src/services/{domain}_service.ts`)

```
import bcrypt from 'bcrypt';
import { verifyToken, generateToken } from '../utils/jwt';
import { userStore } from '../stores/user_store';
import { JSON_RPC_ERRORS } from '../utils/errors';
import type { LoginParams, LoginResponse } from '../models/auth_schemas';

export async function login(params: LoginParams): Promise<LoginResponse> {
  const { username, password } = params;

  // Buscar usuário
  const user = userStore.findByUsername(username);
  if (!user) {
    throw {
      ...JSON_RPC_ERRORS.UNAUTHORIZED,
      message: 'Credenciais inválidas',
      data: { reason: 'Usuário não encontrado' }
    };
  }

  // Verificar senha
  const passwordMatch = await bcrypt.compare(password, user.passwordHash);
  if (!passwordMatch) {
    throw {
      ...JSON_RPC_ERRORS.UNAUTHORIZED,
      message: 'Credenciais inválidas',
      data: { reason: 'Senha incorreta' }
    };
  }

  // Gerar token
  const token = generateToken({
    userId: user.id,
    username: user.username,
    role: user.role
  });

  // Atualizar último login
  userStore.update(user.id, { lastLogin: new Date().toISOString() });

  return {
    token,
    user: {
      id: user.id,
```

```
      username: user.username,
      role: user.role
    },
    expiresIn: 86400 // 24h em segundos
  };
}
```

3. Adicionar Método RPC (`src/rpc/handlers/rpc_methods/auth_methods.ts`)

```
import { LoginParamsSchema } from '../../../models/auth_schemas';
import { login as loginService } from '../../../services/auth_service';

export async function login(params: unknown) {
  // Validar params com Zod
  const validatedParams = LoginParamsSchema.parse(params);

  // Chamar service (service faz validação de negócio)
  return await loginService(validatedParams);
}
```

4. Registrar no Registry (`src/rpc/handlers/rpc_methods/index.ts`)

```
import { login } from './auth_methods';
import { vote } from './game_methods';
import { rollAttack } from './combat_methods';

export const methodRegistry = {
  'login': login,
  'register': register,
  'vote': vote,
  'rollAttack': rollAttack,
  // ... todos os métodos RPC
};
```

5. Criar Wrapper REST (`src/rpc/handlers/wrappers/auth_wrappers.ts`)

```
import { FastifyPluginAsync } from 'fastify';
import { ZodTypeProvider } from 'fastify-type-provider-zod';
import { LoginParamsSchema, LoginResponseSchema } from '../../../models/auth_schemas';
import { login as loginService } from '../../../services/auth_service';

export const authWrappers: FastifyPluginAsync = async (app) => {
  app.withTypeProvider<ZodTypeProvider>().route({
    method: 'POST',
    url: '/rpc/login',
    schema: {
      tags: ['Authentication'],
      summary: 'Autentica usuário e retorna JWT token',
      description: 'Endpoint de login. Retorna token JWT válido por 24h.',
      body: LoginParamsSchema,
      response: {
        200: LoginResponseSchema
      }
    },
    handler: async (request, reply) => {
      const result = await loginService(request.body);
      return reply.send(result);
    }
  });
};
```

6. Registrar OpenAPI Path (`src/rpc/openapi/paths/auth_paths.ts`)

```
import { registry } from '../registry';
import { LoginParamsSchema, LoginResponseSchema } from '../../models/auth_schemas';

registry.registerPath({
  method: 'post',
  path: '/rpc/login',
  tags: ['Authentication'],
  summary: 'Autentica usuário e retorna JWT token',
  request: {
    body: {
      content: {
        'application/json': {
          schema: LoginParamsSchema
        }
      }
    }
  },
  responses: {
    200: {
      description: 'Login bem-sucedido',
      content: {
        'application/json': {
          schema: LoginResponseSchema
        }
      }
    }
  }
});
```

Servidor RPC Principal

```
// src/rpc/server.ts
import Fastify from 'fastify';
import cors from '@fastify/cors';
import swagger from '@fastify/swagger';
import swaggerUi from '@fastify/swagger-ui';
import { methodRegistry } from './handlers/rpc_methods';
import { authWrappers } from './handlers/wrappers/auth_wrappers';
import { gameWrappers } from './handlers/wrappers/game_wrappers';
import { combatWrappers } from './handlers/wrappers/combat_wrappers';

const app = Fastify({ logger: true });

// CORS para clientes remotos
app.register(cors, {
  origin: '*',
  credentials: true
});

// Swagger UI
app.register(swagger, {
  openapi: {
    info: {
      title: 'RPC Interactive Stories API',
      version: '1.0.0',
      description: 'Sistema de histórias interativas multiplayer com JSON-RPC 2.0'
    },
    servers: [
      { url: 'http://173.249.60.72:8443', description: 'Produção (VPS)' },
      { url: 'http://localhost:8443', description: 'Desenvolvimento' }
    ]
  }
});

app.register(swaggerUi, {
  routePrefix: '/docs'
});

// Registrar wrappers REST (apenas para Swagger)
app.register(authWrappers);
app.register(gameWrappers);
app.register(combatWrappers);

// Endpoint RPC REAL (usado pelo frontend)
```

```
app.post('/rpc', async (request, reply) => {
  const { jsonrpc, id, method, params } = request.body as any;

  // Validar JSON-RPC 2.0
  if (jsonrpc !== '2.0') {
    return reply.send({
      jsonrpc: '2.0',
      id,
      error: {
        code: -32600,
        message: 'Invalid Request',
        data: { reason: 'jsonrpc field must be "2.0"' }
      }
    });
  }

  // Buscar método no registry
  const rpcMethod = methodRegistry[method];
  if (!rpcMethod) {
    return reply.send({
      jsonrpc: '2.0',
      id,
      error: {
        code: -32601,
        message: 'Method not found',
        data: { method }
      }
    });
  }

  try {
    // Executar método RPC
    const result = await rpcMethod(params || {});

    return reply.send({
      jsonrpc: '2.0',
      id,
      result
    });
  } catch (error: any) {
    // Tratar erro estruturado
    return reply.send({
      jsonrpc: '2.0',
      id,
```

```
      error: {
        code: error.code || -32603,
        message: error.message || 'Internal error',
        data: error.data
      }
    });
  }
});

// Health check
app.get('/health', async () => ({
  status: 'ok',
  uptime: process.uptime(),
  timestamp: new Date().toISOString()
}));

// Iniciar servidor
const start = async () => {
  try {
    await app.listen({ port: 8443, host: '0.0.0.0' });
    console.log('□ Servidor RPC rodando em http://0.0.0.0:8443');
    console.log('□ Documentação disponível em http://0.0.0.0:8443/docs');
  } catch (error) {
    app.log.error(error);
    process.exit(1);
  }
};

start();
```

2.5 Implementação Frontend

Estrutura de Pastas

```
frontend/
├── public/
│   ├── index.html
│   ├── login.html
│   ├── register.html
│   ├── home.html
│   ├── character-form.html
│   ├── waiting-room.html
│   └── gameplay.html
├── src/
│   ├── rpc/
│   │   └── client.js            # Cliente RPC
│   ├── services/                # Wrappers de RPC
│   │   ├── authService.js
│   │   ├── gameService.js
│   │   ├── characterService.js
│   │   └── combatService.js
│   ├── ui/
│   │   ├── auth/
│   │   │   ├── login.js
│   │   │   └── register.js
│   │   ├── characters/
│   │   │   └── character-form.js
│   │   ├── sessions/
│   │   │   └── waiting-room.js
│   │   ├── game/
│   │   │   └── gameplay.js
│   │   └── shared/
│   │       ├── utils.js
│   │       ├── constants.js
│   │       └── navigation.js
│   ├── styles/                  # CSS por página
│   │   ├── main.css
│   │   ├── login.css
│   │   ├── character-form.css
│   │   └── gameplay.css
│   ├── utils/
│   │   └── auth.js              # getToken, requireAuth, logout
│   └── server.js                # Express local
├── .env
└── package.json
```

Cliente RPC

```
// src/rpc/client.js
export default class RpcClient {
  constructor(endpoint) {
    this.endpoint = endpoint || 'http://173.249.60.72:8443';
    this.requestId = 1;
  }

  async call(method, params = {}) {
    const payload = {
      jsonrpc: '2.0',
      id: this.requestId++,
      method,
      params
    };

    try {
      const response = await fetch(`${this.endpoint}/rpc`, {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify(payload)
      });

      if (!response.ok) {
        throw new Error(`HTTP ${response.status}: ${response.statusText}`);
      }

      const data = await response.json();

      if (data.error) {
        const error = new Error(data.error.message);
        error.code = data.error.code;
        error.data = data.error.data;
        throw error;
      }

      return data.result;
    } catch (error) {
      console.error('RPC Error:', error);
      throw error;
    }
```

```
    }
}
```

Service Layer (Exemplo: authService.js)

```
// src/services/authService.js
import RpcClient from '../rpc/client.js';

const client = new RpcClient();

export async function login(username, password) {
  try {
    const result = await client.call('login', { username, password });

    // Salvar token no localStorage
    localStorage.setItem('auth_token', result.token);
    localStorage.setItem('user', JSON.stringify(result.user));

    return result;
  } catch (error) {
    console.error('Login error:', error);
    throw error;
  }
}

export async function register(username, password, confirmPassword) {
  return await client.call('register', { username, password, confirmPassword });
}

export function getToken() {
  return localStorage.getItem('auth_token');
}

export function getUser() {
  const userJson = localStorage.getItem('user');
  return userJson ? JSON.parse(userJson) : null;
}

export function logout() {
  localStorage.removeItem('auth_token');
  localStorage.removeItem('user');
  window.location.href = '/login.html';
}

export function requireAuth() {
  if (!getToken()) {
    window.location.href = '/login.html';
    return false;
```

```
    }
    return true;
}
```

Componente de Interface (Exemplo: login.js)

```
// src/ui/auth/login.js
import { login } from '../../services/authService.js';
import { handleError } from '../shared/utils.js';

document.addEventListener('DOMContentLoaded', () => {
  const form = document.getElementById('login-form');
  const usernameInput = document.getElementById('username');
  const passwordInput = document.getElementById('password');
  const submitButton = document.getElementById('submit-btn');
  const errorMessage = document.getElementById('error-message');

  form.addEventListener('submit', async (e) => {
    e.preventDefault();

    // Limpar mensagens de erro
    errorMessage.textContent = '';
    errorMessage.style.display = 'none';

    // Desabilitar botão durante requisição
    submitButton.disabled = true;
    submitButton.textContent = 'Entrando...';

    try {
      const username = usernameInput.value.trim();
      const password = passwordInput.value;

      const result = await login(username, password);

      console.log('Login bem-sucedido:', result);

      // Redirecionar para home
      window.location.href = '/home.html';
    } catch (error) {
      errorMessage.textContent = error.message || 'Erro ao fazer login';
      errorMessage.style.display = 'block';

      submitButton.disabled = false;
      submitButton.textContent = 'Entrar';
    }
  });
});
```

Página HTML (Exemplo: login.html)

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Login - Histórias Interativas</title>
  <link rel="stylesheet" href="/src/styles/main.css">
  <link rel="stylesheet" href="/src/styles/login.css">
</head>
<body>
  <div class="container">
    <div class="login-card">
      <h1>Histórias Interativas</h1>
      <p class="subtitle">Faça login para começar sua aventura</p>

      <form id="login-form">
        <div class="form-group">
          <label for="username">Usuário</label>
          <input
            type="text"
            id="username"
            name="username"
            required
            minlength="3"
            placeholder="Digite seu usuário"
          >
        </div>

        <div class="form-group">
          <label for="password">Senha</label>
          <input
            type="password"
            id="password"
            name="password"
            required
            minlength="6"
            placeholder="Digite sua senha"
          >
        </div>

        <div id="error-message" class="error-message" style="display: none;"></div>

        <button type="submit" id="submit-btn" class="btn-primary">
```

```
          Entrar
        </button>
      </form>

      <p class="register-link">
        Não tem uma conta?
        <a href="/register.html">Cadastre-se</a>
      </p>
    </div>
  </div>

  <script type="module" src="/src/ui/auth/login.js"></script>
</body>
</html>
```

2.6 Comunicação RPC

Fluxo Completo de uma Chamada RPC

Exemplo: Votação em uma Opção

1. **Cliente** (gameplay.js) chama método vote:

```
import { submitVote } from '../../services/gameService.js';
import { getToken } from '../../utils/auth.js';

async function handleVote(optionId) {
  try {
    const token = getToken();
    const sessionId = getCurrentSessionId();
    const characterId = getCurrentCharacterId();

    const result = await submitVote(token, sessionId, characterId, optionId);

    if (result.sucesso) {
      updateUI({
        message: 'Voto registrado!',
        timerStarted: result.timerStarted,
        timeRemaining: result.timeRemaining
      });
    }
  } catch (error) {
    showError(error.message);
  }
}
```

2. **Service Layer** (gameService.js) faz chamada RPC:

```
export async function submitVote(token, sessionId, characterId, opcaoId) {
  const client = new RpcClient();
  return await client.call('vote', { token, sessionId, characterId, opcaoId });
}
```

3. **RPC Client** envia payload JSON-RPC:

```
POST http://173.249.60.72:8443/rpc

{
  "jsonrpc": "2.0",
  "id": 123,
  "method": "vote",
  "params": {
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    "sessionId": "session_789",
    "characterId": "char_456",
    "opcaoId": "entrar"
  }
}
```

4. **Servidor RPC** recebe requisição em `/rpc`:

```
app.post('/rpc', async (request, reply) => {
  const { jsonrpc, id, method, params } = request.body;

  // Buscar método "vote" no registry
  const rpcMethod = methodRegistry['vote'];

  // Executar método
  const result = await rpcMethod(params);

  return reply.send({ jsonrpc: '2.0', id, result });
});
```

5. **RPC Method** valida e chama service:

```
// src/rpc/handlers/rpc_methods/game_methods.ts
export async function vote(params: unknown) {
  const validatedParams = SubmitVoteSchema.parse(params);
  return await voteService.submitVote(validatedParams);
}
```

6. **Service** executa lógica de negócio:

```
// src/services/vote_service.ts
export async function submitVote(params: SubmitVote): Promise<SubmitVoteResponse> {
  const { token, sessionId, characterId, opcaoId } = params;

  // 1. Validar token
  const decoded = verifyToken(token);
  const userId = decoded.userId;

  // 2. Buscar sessão
  const session = sessionStore.findById(sessionId);
  if (!session) {
    throw { ...JSON_RPC_ERRORS.NOT_FOUND, message: 'Sessão não encontrada' };
  }

  // 3. Validar personagem pertence ao usuário
  const character = characterStore.findById(characterId);
  if (character.userId !== userId) {
    throw { ...JSON_RPC_ERRORS.FORBIDDEN, message: 'Personagem não é seu' };
  }

  // 4. Registrar voto
  session.votes[characterId] = opcaoId;

  // 5. Verificar se deve iniciar timer
  const isFirstVote = Object.keys(session.votes).length === 1;
  let timerStarted = false;
  let timeRemaining = 0;

  if (isFirstVote) {
    session.votingTimer = {
      startedAt: new Date(),
      duration: 60000, // 1 minuto
      autoFinish: true
    };
    timerStarted = true;
    timeRemaining = 60;
  }

  // 6. Salvar sessão
  sessionStore.update(sessionId, session);

  // 7. Criar evento para long polling
  eventQueue.push({
```

```
    type: 'VOTE_RECEIVED',
    sessionId,
    data: { characterId, opcaoId }
  });

  return {
    sucesso: true,
    timerStarted,
    timeRemaining
  };
}
```

7. **Servidor** retorna resposta JSON-RPC:

```
{
  "jsonrpc": "2.0",
  "id": 123,
  "result": {
    "sucesso": true,
    "timerStarted": true,
    "timeRemaining": 60
  }
}
```

8. **Cliente** recebe resultado e atualiza UI:

```
// Atualiza contador de votos
updateVoteCounter();

// Inicia timer visual
startVotingTimer(60);

// Mostra feedback
showSuccessMessage('Voto registrado!');
```

Long Polling para Atualizações em Tempo Real

O sistema usa **long polling** via RPC para notificações:

```
// Frontend: polling contínuo
async function pollUpdates() {
  const token = getToken();
  const sessionId = getCurrentSessionId();
  let lastUpdateId = null;

  while (true) {
    try {
      const result = await client.call('checkGameUpdates', {
        token,
        sessionId,
        lastUpdateId
      });

      if (result.updates.length > 0) {
        // Processar atualizações
        result.updates.forEach(update => {
          handleUpdate(update);
        });

        lastUpdateId = result.lastUpdateId;
      }

      // Aguardar antes da próxima requisição
      await sleep(2000);
    } catch (error) {
      console.error('Polling error:', error);
      await sleep(5000); // Backoff em caso de erro
    }
  }
}

function handleUpdate(update) {
  switch (update.type) {
    case 'VOTE_RECEIVED':
      updateVoteDisplay(update.data);
      break;
    case 'CHAPTER_CHANGED':
      loadNewChapter(update.data.newChapter);
      break;
    case 'PLAYER_STATUS_CHANGED':
      updatePlayerTiles(update.data.playerUpdates);
      break;
```

```
  }
}
```

---

3. Instruções de Instalação e Uso

3.1 Instalação do Backend (Servidor)

Pré-requisitos

- Node.js 18+ (Download (https://nodejs.org/))
- npm (incluído com Node.js)
- Git (Download (https://git-scm.com/))

Passos de Instalação

1. Clone o repositório:

```
git clone https://github.com/seu-usuario/RPC-Interactive-Stories-Game.git
cd RPC-Interactive-Stories-Game
```

2. Navegue até o diretório do backend:

```
cd backend
```

3. Instale as dependências:

```
npm install
```

4. Configure as variáveis de ambiente:

```
cp .env.example .env
```

Edite o arquivo `.env` conforme necessário:

```
NODE_ENV=development
PORT=8443
HOST=0.0.0.0

# JWT
JWT_SECRET=seu-secret-key-super-seguro-aqui
JWT_EXPIRES_IN=24h

# CORS
CORS_ORIGIN=*

# Logs
LOG_LEVEL=info
```

5. Execute em modo desenvolvimento:

```
npm run dev
```

O servidor estará rodando em `http://localhost:8443`.

6. Acesse a documentação Swagger:

Abra seu navegador em `http://localhost:8443/docs` para ver a documentação interativa da API.

Build para Produção

```
# Compilar TypeScript
npm run build

# Executar build
npm start
```

3.2 Instalação do Frontend (Cliente)

Pré-requisitos

- Node.js 16+ (Download (https://nodejs.org/))
- npm (incluído com Node.js)

Passos de Instalação

1. Navegue até o diretório do frontend:

```
cd frontend
```

2. Instale as dependências:

```
npm install
```

3. Configure a URL do servidor:

O arquivo `.env` já está configurado:

```
SERVER_URL=http://173.249.60.72:8443
PORT=5173
```

Para desenvolvimento local, altere para:

```
SERVER_URL=http://localhost:8443
PORT=5173
```

4. Execute o servidor de desenvolvimento:

```
npm run dev
```

O navegador abrirá automaticamente em `http://localhost:5173`.

3.3 Deploy em Produção

Deploy do Backend (VPS com Docker)

1. Conecte-se à VPS via SSH:

```
ssh root@173.249.60.72
```

2. Clone o repositório na VPS:

```
git clone https://github.com/seu-usuario/RPC-Interactive-Stories-Game.git
cd RPC-Interactive-Stories-Game/backend
```

3. Configure variáveis de ambiente para produção:

```
cp .env.example .env.production
nano .env.production
```

4. Build e execute com Docker Compose:

```
docker-compose up -d --build
```

5. Verifique o status:

```
docker-compose ps
docker-compose logs -f
```

O servidor estará acessível em `http://173.249.60.72:8443`.

Comandos Docker Úteis

```
# Ver logs
docker-compose logs -f --tail=50

# Reiniciar servidor
docker-compose restart

# Parar servidor
docker-compose down

# Rebuild e restart
docker-compose up -d --build

# Ver métricas
docker stats
```

Backup dos Dados

```
# Criar backup
tar -czf backup_$(date +%Y%m%d).tar.gz ./data

# Restaurar backup
tar -xzf backup_20250201.tar.gz
```

3.4 Guia de Uso

1. Registro de Usuário

1. Acesse `http://localhost:5173/register.html`
2. Preencha:
    - **Usuário:** Mínimo 3 caracteres
    - **Senha:** Mínimo 6 caracteres
    - **Confirmar Senha:** Deve ser igual à senha
3. Clique em "Cadastrar"
4. Você será redirecionado para a tela de login

2. Login

1. Acesse `http://localhost:5173/login.html`
2. Digite seu usuário e senha
3. Clique em "Entrar"
4. Após o login, você será redirecionado para a **Home**

3. Criar Sessão de Jogo

1. Na **Home**, clique em "Criar Nova Sessão"
2. Selecione uma história do catálogo
3. Configure:
    - **Nome da Sessão:** Ex: "Aventura do Grupo A"
    - **Máximo de Jogadores:** 2-10
4. Clique em "Criar"
5. Você receberá um **código de sessão** (ex: "ABC123")
6. Compartilhe o código com outros jogadores

4. Entrar em Sessão Existente

1. Na **Home**, clique em "Entrar em Sessão"
2. Digite o **código da sessão** (ex: "ABC123")
3. Clique em "Entrar"
4. Você será levado para a **sala de espera**

5. Criar Personagem D&D

Na **sala de espera**, você deve criar seu personagem:

1. **Informações Básicas:**
    - **Nome:** Ex: "Aragorn"
    - **Raça:** Humano, Elfo, Anão ou Halfling
    - **Classe:** Guerreiro, Mago, Ladino ou Clérigo
2. **Atributos (3-18 cada):**
    - Força (Strength)

- Destreza (Dexterity)
- Constituição (Constitution)
- Inteligência (Intelligence)
- Sabedoria (Wisdom)
- Carisma (Charisma)

3. **Background (mínimo 10 caracteres cada):**

   - **História:** Background do personagem
   - **Aparência:** Descrição física
   - **Personalidade:** Traços de personalidade
   - **Medos:** Do que o personagem tem medo
   - **Objetivos:** Metas e motivações

4. **Equipamento:**

   - Adicione até 10 itens (ex: "Espada longa", "Escudo", "Armadura de couro")

5. Clique em "Criar Personagem"

6. Iniciar Jogo

Quando todos os jogadores criarem seus personagens:

1. O dono da sessão verá o botão "Iniciar Jogo"
2. Clique para começar a aventura
3. Todos serão redirecionados para a **tela de jogo**

7. Gameplay - Votação

Na tela de jogo:

1. Leia o texto do capítulo atual
2. Veja as opções disponíveis
3. Clique na opção desejada para votar
4. Quando alguém vota primeiro, um **timer** inicia (padrão: 1 minuto)
5. Todos devem votar antes do timer acabar
6. A opção mais votada vence
7. A história avança para o próximo capítulo

8. Sistema de Combate

Quando o grupo encontrar inimigos:

Fase 1: Iniciativa

1. Cada jogador clica em "Rolar Iniciativa"
2. Sistema rola 1d20 + modificadores
3. Ordem de turno é determinada

Fase 2: Turnos de Combate

1. No seu turno:
    - Selecione um alvo (inimigo)
    - Clique em "Atacar"
2. Sistema rola ataque (1d20 + modificadores)
3. Se acertar, rola dano
4. HP do alvo diminui
5. Próximo na ordem age

Mecânicas Especiais:

- **Acerto Crítico (20 natural):** Dano × 2
- **Falha Crítica (1 natural):** Atacante toma 1d4 de dano
- **Morte:** HP = 0, personagem cai
- **Ressurreição:** Rolar 2d10, sucesso se resultado ≥ 11 (máximo 3 tentativas)

Fim do Combate:

- Quando todos os inimigos morrem: **Vitória**
- Grupo recebe XP e possível loot
- História continua

9. Chat

Durante o jogo:

1. Use o painel de chat à direita
2. Digite mensagens para discutir com o grupo
3. Mensagens aparecem em tempo real (long polling)

10. Painel Administrativo (ADMIN)

Se você é administrador:

1. Acesse "Painel Admin" na Home
2. Funções disponíveis:
    - **Gerenciar Usuários:** Ver todos os usuários, promover/rebaixar, deletar
    - **Gerenciar Sessões:** Ver todas as sessões, forçar mudanças de estado, deletar
    - **Gerenciar Histórias:** Upload de arquivos Mermaid, ativar/desativar histórias
    - **Estatísticas:** Ver estatísticas do sistema

11. Criar História (ADMIN)

Para adicionar uma nova história:

1. Crie um arquivo `.mmd` com flowchart Mermaid:

flowchart TD inicio["Você chega a uma caverna misteriosa..."] inicio -->|Entrar na caverna| dentro["Você está dentro da caverna escura"] inicio -->|Voltar para cidade| cidade["Você retorna à cidade"] dentro -->|Explorar| combate{{"Combate: Troll da Caverna"}} dentro -->|Sair| inicio combate -->|Vitória| tesouro["Você encontra um baú de tesouro!"] combate -->|Derrota| fim_derrota["FIM - Derrota"] tesouro --> fim_vitoria["FIM - Vitória"] cidade --> fim_desistencia["FIM - Desistência"]

2. Acesse o painel admin
3. Clique em "Upload História"
4. Faça upload do arquivo `.mmd`
5. Preencha metadados:
    - Título
    - Descrição
    - Gênero
    - Número recomendado de jogadores
    - Dificuldade
6. A história estará disponível no catálogo

---

4. Conclusão

O projeto **RPC Interactive Stories** demonstra com sucesso a aplicação de **JSON-RPC 2.0** em um sistema distribuído complexo de histórias interativas multiplayer. A escolha do protocolo JSON-RPC 2.0 se mostrou adequada para o domínio, proporcionando:

Benefícios Alcançados

1. **Simplicidade de Implementação**
    - Protocolo leve e fácil de entender
    - Redução de complexidade comparado a alternativas (gRPC, Java RMI)
    - Facilita manutenção e evolução do código
2. **Interoperabilidade**
    - Cliente JavaScript web pode se conectar ao servidor Node.js sem problemas
    - Possibilidade de criar clientes em outras linguagens facilmente
    - Formato JSON é universalmente suportado
3. **Performance Adequada**
    - Latência baixa para operações RPC
    - Payload compacto
    - Suficiente para aplicação multiplayer em tempo real
4. **Facilidade de Debugging**
    - Requisições e respostas em formato texto legível (JSON)

- Ferramentas de rede do navegador mostram tráfego claramente
- Swagger UI facilita testes manuais

5. **Documentação Automática**

    - Integração natural com Swagger/OpenAPI
    - Documentação sempre sincronizada com código
    - Interface interativa para explorar API

Desafios Enfrentados

1. **Ausência de Type Hints Nativos**

    - Solução: Uso de Zod para validação e inferência de tipos TypeScript

2. **Falta de Streaming Bidirecional**

    - Solução: Long polling para atualizações em tempo real
    - Alternativa futura: Considerar WebSockets para eventos

3. **Gerenciamento de Estado Distribuído**

    - Solução: Arquivos JSON com locking para evitar race conditions
    - Alternativa futura: Migrar para banco de dados (PostgreSQL, MongoDB)

Aprendizados

- **RPC é ideal para operações procedurais** (login, votar, atacar) em vez de REST
- **JSON-RPC 2.0 é excelente para MVPs** e sistemas pequenos/médios
- **Arquitetura cliente-servidor distribuída** funciona bem com RPC sobre HTTP
- **Validação rigorosa** (Zod schemas) é essencial para robustez
- **Documentação automática** (Swagger) economiza tempo e evita dessincronia

Possíveis Melhorias Futuras

1. **WebSockets para Eventos em Tempo Real**

    - Substituir long polling por WebSocket para menor latência
    - Eventos push do servidor para clientes

2. **Banco de Dados Relacional**

    - Migrar de arquivos JSON para PostgreSQL
    - Transações ACID para operações críticas

3. **Autenticação Avançada**

    - OAuth 2.0 / OpenID Connect

- Refresh tokens para sessões longas

4. **Cache Distribuído**
    - Redis para cache de sessões e estados
    - Reduzir carga no armazenamento
5. **Escalabilidade Horizontal**
    - Múltiplas instâncias do servidor com load balancer
    - Sessões compartilhadas via Redis
6. **Testes Automatizados**
    - Testes unitários para serviços
    - Testes de integração para RPC endpoints
    - Testes E2E para fluxos completos

Conclusão Final

O projeto alcançou seus objetivos de implementar um sistema distribuído funcional usando **JSON-RPC 2.0 puro**. A arquitetura cliente-servidor com comunicação RPC se mostrou eficiente e escalável para o domínio de histórias interativas multiplayer. O uso de tecnologias modernas (Fastify, Zod, TypeScript, Docker) em conjunto com o protocolo JSON-RPC 2.0 resultou em uma aplicação robusta, bem documentada e fácil de manter.

A experiência adquirida no desenvolvimento deste sistema pode ser aplicada em outros contextos de sistemas distribuídos, especialmente aqueles que requerem comunicação cliente-servidor com operações procedurais.

---

5. Referências

- **JSON-RPC 2.0 Specification:** https://www.jsonrpc.org/specification (https://www.jsonrpc.org/specification)
- **Fastify Documentation:** https://www.fastify.io/ (https://www.fastify.io/)
- **Zod Documentation:** https://zod.dev/ (https://zod.dev/)
- **TypeScript Handbook:** https://www.typescriptlang.org/docs/ (https://www.typescriptlang.org/docs/)
- **Docker Documentation:** https://docs.docker.com/ (https://docs.docker.com/)
- **JWT (JSON Web Tokens):** https://jwt.io/ (https://jwt.io/)
- **OpenAPI Specification:** https://swagger.io/specification/ (https://swagger.io/specification/)
- **Mermaid Flowcharts:** https://mermaid.js.org/ (https://mermaid.js.org/)
- **D&D 5e SRD:** https://www.dndbeyond.com/sources/basic-rules (https://www.dndbeyond.com/sources/basic-rules)
- **Repositório do Projeto:** https://github.com/16Jorgecamargo/RPC-Interactive-Stories-Game (https://github.com/16Jorgecamargo/RPC-Interactive-Stories-Game)
