Faculdade de Ciências Exatas e da Engenharia

Licenciatura em Engenharia Informática

Sistemas Operativos

Projeto Prático - 3ª Fase

Sudoku

05/01/2025

Docentes:

Prof.º Eduardo Marques

Prof.º Fernando Martins

Discentes:

2022622 Pedro Brito

2150821 Cláudio Pinto

2118722 João Tomás

Índice

Lista de Figuras

1 Introdução

Este relatório descreve o desenvolvimento de um jogo Sudoku baseado numa arquitetura cliente-servidor, com ênfase nos desafios de sincronização em ambientes *multiplayer*. O sistema permite que exista competição entre múltiplos jogadores para resolver tabuleiros de Sudoku, enquanto o servidor gere o estado do jogo e as interações.

A implementação utiliza comunicação baseada no protocolo *TCP* e distingue jogadores *premium* e não *premium*, garantindo vantagens exclusivas aos primeiros, como prioridade na entrada de salas e maior controlo sobre jogadas. Os desafios de sincronização incluem a gestão concorrente do tabuleiro através das soluções clássicas de leitores-escritores, produtores-consumidores, loja do barbeiro, prioridades estáticas e dinâmicas, utilização de barreiras e filas.

Este relatório analisa as soluções adotadas, as limitações identificadas e sugestões para melhorias futuras.

O código-fonte do projeto pode ser encontrado no *GitHub* no seguinte repositório: Sudoku_SO.

2 Lógica do Jogo e Decisões do Grupo

2.1 Descrição do Jogo

O jogo Sudoku é baseado numa arquitetura cliente-servidor, que permite a participação simultânea de múltiplos jogadores no mesmo ambiente. O objetivo é proporcionar uma experiência interativa onde os jogadores podem selecionar ou iniciar jogos de Sudoku, seja *singleplayer* ou em modo *multiplayer*, através de menus.

Na inicialização do servidor existem parâmetros que podem ser alterados como: a porta do servidor; caminho relativo dos jogos; caminho relativo dos logs registados; número máximo de salas que podem estar instanciadas ao mesmo tempo; número máximo de jogadores por sala; número máximo de clientes conectados ao servidor e o número máximo de espera (em modo de jogo com prioridades dinâmicas).

Por outro lado, na inicialização do cliente existem os seguintes parâmetros: IP do servidor; porta do servidor; *hostname* do servidor (ainda não implementado); caminho relativo para os logs; switch para selecionar se o cliente resolve o tabuleiro de foram manual ou automática; switch para selecionar se o cliente é *premium* ou não-*premium* e, por fim, um parâmetro para definir a eficiência de resolução do cliente se este resolve de forma automática (de 1 (fácil) a 3 (difícil)).

No modo *singleplayer*, os jogadores têm a possibilidade de escolher um jogo aleatório, selecionado automaticamente, ou um jogo específico, através de uma lista de opções fornecida pelo servidor.

No modo *multiplayer*, vários jogadores competem para resolver o mesmo tabuleiro, preenchendo o maior número possível de células. Contudo, existe um limite de jogadores por sala, que é definido no momento da inicialização do servidor. Durante o jogo, as ações de cada jogador têm impacto direto no estado do tabuleiro, tornando essencial que todos os participantes tenham uma visão atualizada e coerente do progresso. Na criação de uma sala *multiplayer*, o jogador criador da sala deve escolher o modo de sincronização para esse jogo. Os modos de sincronização que este sistema oferece são: leitores-escritores, loja do barbeiro com prioridades estáticas, loja do barbeiro com prioridades dinâmicas e loja do barbeiro com filas FIFO.

Os jogos disponíveis estão armazenados num ficheiro contendo informações essenciais,

como o tabuleiro inicial, a solução correta, o tempo recorde e o número máximo de acertos registados. Durante um jogo, o cliente deve tentar resolver o tabuleiro linha a linha. Para tal, o cliente envia para o servidor uma tentativa de preenchimento de uma linha completa do tabuleiro. O servidor analisa esta linha, célula por célula, comparando-a com a solução do jogo. Caso uma das células enviadas pelo cliente esteja correta, o servidor atualiza o tabuleiro com o valor correspondente. Se a linha completa estiver correta, o servidor avança para a linha seguinte, enviando o tabuleiro atualizado de volta aos clientes.

Adicionalmente, existe uma distinção entre jogadores *premium* e não *premium*, sendo que os primeiros têm acesso a vantagens exclusivas, como:

- **Prioridade na entrada da sala:** Jogadores *premium* são colocados à frente de jogadores não *premium* em todos os modos de jogo *multiplayer*.
- **Prioridade na leitura e escrita no tabuleiro:** Jogadores *premium* têm prioridade a jogar em relação aos jogadores não *premium* dependendo do modo de jogo em que se encontram.

Para cada jogo ficam guardadas estatísticas, como o tempo total necessário para completar o tabuleiro e o número de jogadas realizadas por cada jogador. No final do jogo, estas informações podem ser consultadas pelos participantes. Caso o desempenho ultrapasse os recordes previamente registados, os ficheiros do jogo são atualizados com os novos valores.

Também registou-se o número de leituras e escritas de cada jogador com o objetivo de se poder comparar os diferentes métodos de sincronização utilizadas para jogos *multiplayer*.

Devido à forma como as jogadas são verificadas e ao impacto que cada tentativa tem no estado do tabuleiro, é imprescindível assegurar consistência e sincronização entre os jogadores e o servidor, garantindo uma experiência fluida e sem conflitos, especialmente no modo *multiplayer*.

2.2 Formatos de comunicação

A comunicação entre o cliente e o servidor no sistema implementado baseia-se na utilização de *sockets TCP* para troca de mensagens. Este protocolo permite uma comu-

nicação bidirecional entre ambas as partes.

2.2.1 Comunicação no Cliente

No lado do cliente, a lógica de comunicação é responsável por:

- **Envio de Pedidos ao Servidor:** O cliente utiliza a função `send` para enviar mensagens ao servidor.
  Exemplo: Quando o cliente solicita estatísticas, envia a mensagem `"GET_STATS"`.
- **Receção de Respostas do Servidor:** O cliente utiliza a função `recv` para receber dados do servidor.
  As respostas recebidas são armazenadas num *buffer* para posterior processamento e exibição.
- **Fluxo Geral de Comunicação:** O cliente inicia um pedido e aguarda até receber uma resposta antes de continuar com outras operações (comunicação síncrona).

2.2.2 Comunicação no Servidor

No lado do servidor, a lógica é mais complexa, dado que este vai gerir múltiplos clientes simultaneamente:

- **Gestão de Conexões:** O servidor utiliza *threads* para lidar com vários clientes ao mesmo tempo.
  A função `pthread` é utilizada para criar novas *threads*, garantindo que cada cliente é tratado de forma independente.
  Métodos como `select` ajudam a monitorizar múltiplas conexões de forma eficiente.
- **Processamento de Pedidos:** O servidor interpreta as mensagens recebidas e toma as ações apropriadas.
  Exemplo: Ao receber o pedido `"GET_STATS"`, o servidor acede ao ficheiro `room_stats.log` para obter os dados solicitados.
- **Envio de Respostas ao Cliente:** Após processar o pedido, o servidor utiliza a função `send` para enviar uma resposta.

Exemplo: Em caso de erro, o servidor retorna uma mensagem de erro, como `"Erro: Não foi possível abrir o ficheiro de estatísticas."`.

2.3 Estrutura e Funcionamento Geral

Modelo de Comunicação: O cliente inicia a comunicação através de pedidos específicos, e o servidor responde com os dados necessários ou mensagens de erro.

Gestão de Recursos: O servidor gere os recursos como ficheiros de logs (`room_stats.log`) para responder às solicitações dos clientes.

Concorrência: A arquitetura é projetada para suportar múltiplos clientes ao mesmo tempo, utilizando *threads* e métodos de sincronização para evitar conflitos.

2.4 Justificação da Escolha

2.4.1 *Sockets TCP*

A utilização de *sockets* para a comunicação cliente-servidor foi uma escolha devido à sua:

- **Eficiência:** Permite a troca direta de mensagens com baixa latência.
- **Flexibilidade:** Suporta comunicação em rede local.
- **Escalabilidade:** A gestão de múltiplos clientes é realizada de forma eficiente através de *threads*.

2.4.2 Barreira

A utilização de barreiras para a sincronização das *threads* foi devido à sua:

- **Consistência:** Garante que todas as *threads* atingem um ponto comum de execução antes de avançarem, evitando situações de inconsistência.
- **Simplicidade:** Simplifica a lógica de sincronização em cenários onde múltiplas *threads* precisam de sincronizar o progresso por fases.

- **Eficácia:** Reduz a necessidade de sincronização individual entre *threads*, melhorando o desempenho geral.

2.4.3 Filas de Espera com Prioridade

A utilização de filas de espera foi devido à sua:

- **Justiça:** Garante uma ordenação clara com base na prioridade, respeitando o estatuto de jogadores *premium* e não *premium*.
- **Flexibilidade:** Permite implementar diferentes lógicas de acesso (prioridades estáticas ou dinâmicas).
- **Eficácia:** Organiza de forma eficiente os pedidos.

2.4.4 Produtores e Consumidores - *Logs*

A implementação do modelo de Produtores e Consumidores para a escrita de *logs* foi uma escolha devido à sua:

- **Eficiência:** Permite que diversas *threads* produzam mensagens enquanto que apenas uma *thread* fica dedicada para a escrita.
- **Consistência:** Evita problemas de concorrência, controlando o acesso ao *buffer* de *logs*.
- **Escalabilidade:** Permite cenários com uma elevada produção de *logs*, minimizando a perda de desempenho.

2.4.5 *Barber Shop*

A adoção do modelo *Barber Shop* para a gestão de filas foi uma escolha devido à sua:

- **Versatilidade:** Suporta diversas lógicas de gestão de filas (ex.: FIFO, prioridades estáticas e dinâmicas).
- **Conveniência na implementação:** Simplificou a implementação de cenários com múltiplos clientes.
- **Relevância:** Alinha-se com a necessidade de priorização dinâmica do projeto.

2.4.6 Leitores e Escritores

A utilização dos Leitores e Escritores com prioridade para Escritores foi uma escolha devido à sua:

- **Eficiência:** Permite o acesso concorrente por diversos Leitores, garantindo a exclusividade para Escritores.
- **Consistência:** Protege a integridade dos dados durante operações de escrita.
- **Adaptabilidade:** Permitiu ajustar a prioridade entre Leitores e Escritores de acordo com as decisões do grupo para o cenário apresentado.

3 Sincronização

3.1 Partilhada entre Modos

3.1.1 Barreira

O sistema implementa uma barreira de sincronização para sincronizar as múltiplas *threads* (clientes) na sala de jogo. Este mecanismo garante que todas as *threads* atinjam um ponto específico de execução antes de avançarem para a próxima fase, sincronizando tanto a entrada como a saída da zona crítica.

Este sistema foi utilizado para o início e final dos jogos de modo a garantir que todos os jogadores iniciam e acabam o jogo ao mesmo tempo.

- **acquireTurnsTileSemaphore**: Esta função assegura que todas as *threads* chegam à barreira antes de avançar para a zona crítica. Cada *thread* incrementa um contador partilhado quando chega à barreira. Quando todas as *threads* estão prontas, a barreira é desbloqueada, permitindo que todas avancem simultaneamente.
- **releaseTurnsTileSemaphore**: Após a conclusão da zona crítica, esta função garante que nenhuma *thread* sai da barreira até que todas as outras também tenham

terminado. À medida que as *threads* completam a sua execução, o contador partilhado é decrementado. Quando todas as *threads* terminam, a barreira é desbloqueada.

3.1.2 Filas de Espera com Prioridade para Entrar no Jogo

O sistema implementa filas com prioridade para organizar os clientes que pretendem entrar na sala de jogo. Estas filas asseguram que os clientes *premium* têm prioridade sobre os restantes.

- ***Struct*** `PriorityQueue`: A *Struct* `PriorityQueue` faz parte da *Struct* da sala (*`Room`*) e é utilizada para gerir a fila de clientes com base na prioridade.
- **Função** `enqueue`: Adiciona um cliente à fila, ordenando-o com base na sua prioridade (*premium* ou não).
- **Função** `dequeue`: Remove o próximo cliente da fila, começando pelos clientes *premium*.

3.1.3 Produtores e Consumidores - *Logs*

O sistema implementa produtores e consumidores para gerir a escrita de *logs*. Este mecanismo permite que múltiplas *threads* produzam mensagens de *log* enquanto apenas uma *thread* (consumidor) escreve essas mensagens no ficheiro de *log*.

- `logBuffer`: *Buffer* responsável por armazenar as mensagens de *log* temporariamente antes de serem escritas no ficheiro.
- `mutexLogSemaphore`: Semáforo que garante acesso exclusivo ao `logBuffer`, permitindo que apenas um produtor ou consumidor o utilize de cada vez.
- `itemsLogSemaphore`: Conta o número de mensagens disponíveis para consumo no `logBuffer`.
- `spacesSemaphore`: Conta o número de espaços disponíveis no `logBuffer` para que os produtores possam adicionar mensagens.

- `produceLog`: Função utilizada para adicionar mensagens ao `logBuffer`. Antes de produzir um *log*, verifica se existe espaço disponível no *buffer* através do `spacesSemaphore`.
- `consumeLog`: Função responsável por processar as mensagens do `logBuffer` e escrevê-las no ficheiro. Antes de consumir um *log*, verifica se há mensagens disponíveis no `itemsLogSemaphore`.

3.2 Modo de Jogo - 1

3.2.1 Leitores e Escritores (Prioridade para Escritores)

O sistema implementa o padrão da sincronização de Leitores-Escritores com prioridade para escritores, garantindo que os escritores têm acesso exclusivo ao recurso partilhado quando necessário, enquanto os leitores podem aceder simultaneamente, desde que não existam escritores ativos ou em espera.

- `writeSemaphore`: Semáforo que garante exclusividade para escritores durante a operação de escrita.
- `readSemaphore`: Semáforo que controla o acesso dos leitores, bloqueando-os quando existem escritores ativos ou em espera.
- `readMutex`: *Mutex* utilizado para proteger o contador de leitores (`readerCount`), evitando condições de concorrência durante a sua atualização.
- `writeMutex`: *Mutex* utilizado para proteger o contador de escritores (`writerCount`), garantindo que múltiplos escritores não interferem entre si.
- `readerCount`: Contador que mantém o número de leitores atualmente ativos no sistema.
- `writerCount`: Contador que mantém o número de escritores ativos ou em espera.
- `acquireReadLock`: Função que sincroniza os leitores, garantindo que não existem escritores ativos ou em espera antes de permitir o acesso à leitura.

- `releaseReadLock`: Função chamada por um leitor quando termina a sua leitura. Caso seja o último leitor ativo, permite a entrada de escritores.
- `acquireWriteLock`: Função que assegura exclusividade para um escritor, bloqueando o acesso de leitores e outros escritores.
- `releaseWriteLock`: Função chamada por um escritor ao terminar a sua operação, permitindo que leitores ou outros escritores aguardando possam aceder ao recurso.

3.3 Modo de Jogo - 2

3.3.1 *Barber Shop*

O sistema implementa a lógica do problema *Barber Shop*, adaptado para a gestão de *threads* numa fila única com opção de diferentes tipos de prioridade para os clientes. O comportamento da fila é ajustável a três modos distintos:

- **Prioridade Estática (0)**: Os clientes *premium* têm sempre prioridade sobre os não *premium*, mantendo a ordem entre si.
- **Prioridade Dinâmica (1)**: Os clientes *premium* têm prioridade inicial, mas a prioridade dos clientes pode ser ajustada dinamicamente com base no tempo de espera.
- **FIFO (2)**: A fila segue o princípio de *First In, First Out*, sem qualquer tipo de prioridade.

Sincronização

São utilizados semáforos e mutexes para gerir o acesso à fila e sincronizar as operações:

- `costumerSemaphore`: Indica que um cliente está à espera de ser atendido.
- `costumerDoneSemaphore`: Assinala que o cliente foi atendido e está pronto para sair.
- `barberDoneSemaphore`: Indica que o *Barber* terminou de atender o cliente.

- `barberShopMutex`: Garante acesso exclusivo à fila, evitando condições de concorrência.

Barber (*Thread*)

Casa sala de jogo inclui uma *thread* dedicada ao *Barber*, que vai gerir o atendimento dos clientes na fila, retirando-os e processando-os conforme o tipo de fila configurado.

Funções Principais

- `enterBarberShop(Room *room, Client *client)`: Adiciona o cliente à fila utilizando a lógica correspondente ao tipo de fila configurado (Prioridade Estática, Dinâmica ou FIFO) e aguarda que o semáforo `costumerSemaphore` seja assinalado.
- `leaveBarberShop(Room *room, Client *client)`: O cliente assinala que terminou o atendimento utilizando o semáforo `costumerDoneSemaphore` e espera que o *Barber* assinale o semáforo `barberDoneSemaphore` para sair.
- `barberCut(Room *room)`: Retira o próximo cliente da fila com base na lógica do tipo de prioridade configurado.
- `barberIsDone(Room *room)`: Indica que o *Barber* terminou o atendimento e permite que o cliente saia.

Tipos de Fila e Configuração

O tipo de fila é configurado no momento da criação da sala através do parâmetro `synchronizationType`:

- **0**: Utiliza Leitores-Escritores, uma lógica independente do Barber Shop.
- **1**: Barber Shop com Prioridade Estática.
- **2**: Barber Shop com Prioridade Dinâmica.
- **3**: Barber Shop com FIFO.

4 Reflexão sobre as Soluções de Sincronização

4.1 Qual a forma de resolução de tabuleiros foi mais eficiente e porquê?

Os testes foram realizados no servidor da Universidade da Madeira (UMA), utilizando um cliente não *premium*, três clientes *premium* e o jogo associado à *Board ID 3*.

A forma de resolução dos tabuleiros mais eficiente foi observada nos modelos de Leitores e Escritores e *FIFO*. Ambos demonstraram ser os mais eficientes devido à sua capacidade de permitir que todos os jogadores, *premium* e não *premium*, contribuíssem de forma equilibrada na resolução do tabuleiro.

4.1.1 Leitores e Escritores

Este modelo foi eficiente porque permitiu diversos jogadores realizarem operações de leitura simultaneamente, enquanto assegurou exclusividade para as operações de escrita. Resultando numa gestão equilibrada e justa no tabuleiro, garantindo que todos os jogadores tivessem oportunidades iguais de contribuição.

4.1.2 *Barber FIFO*

O modelo *Barber FIFO* foi igualmente eficiente devido à simplicidade na abordagem "primeiro a chegar, primeiro a sair", eliminando assim qualquer tipo de prioridades entre jogadores. Esta estratégia gerou um fluxo contínuo de tentativas e acertos, maximizando a utilização do tabuleiro.

4.2 Testes Realizados

4.2.1 Leitores e Escritores

Neste modelo de sincronização, todos os jogadores, independentemente de serem *premium* ou não *premium*, têm igualdade de oportunidades para aceder ao tabuleiro. O

padrão de Leitores e Escritores garante que diversos jogadores podem ler simultaneamente, enquanto as escritas são realizadas de forma exclusiva.

Como resultado, verificou-se que os jogadores *premium* e não *premium* apresentaram valores iguais de *Reads*, *Writes*, tentativas e acertos, refletindo uma distribuição justa de recursos entre todos os jogadores.

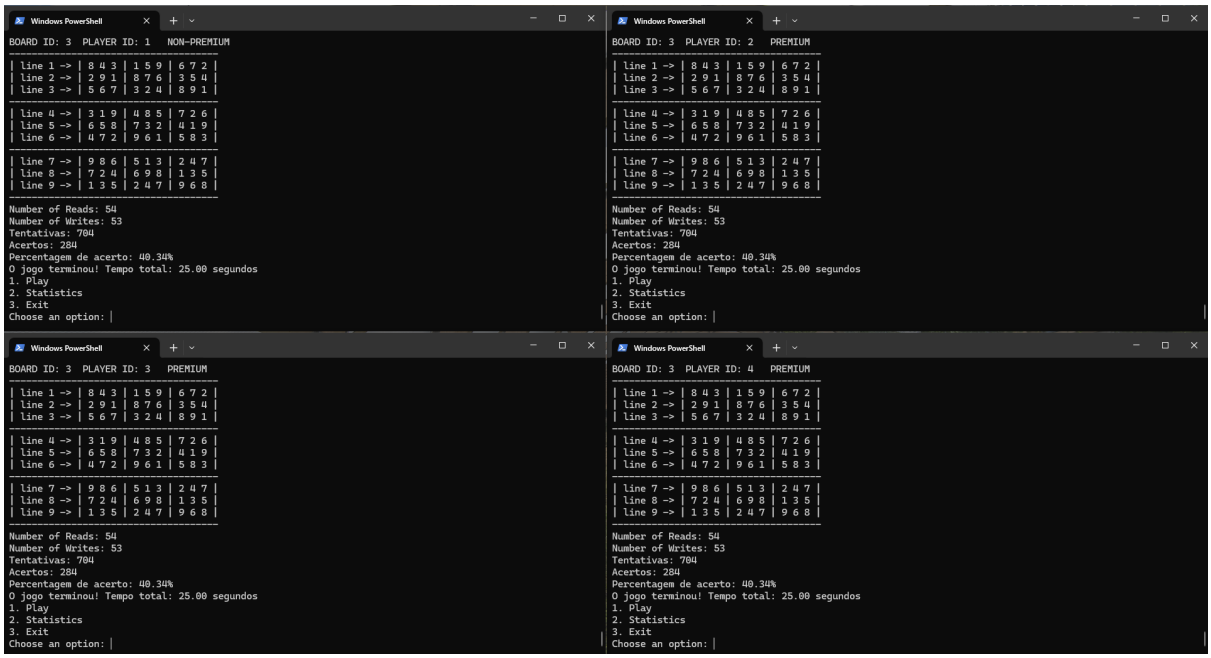

Figura 1: Leitores e Escritores

4.2.2 *Barber* - Prioridades dinâmicas

O parâmetro (*maxWaitingTime*) garante que todos os clientes, *premium* ou não *premium*, sejam atendidos de forma justa, evitando *starvation*. Quando o tempo de espera (*timeInQueue*) de um cliente atinge este limite, ele é movido para o início da fila, ignorando a prioridade habitual.

Abaixo, compara-se o impacto deste mecanismo com valores de 5 e 10 ciclos para ilustrar o seu efeito no equilíbrio entre justiça e prioridade.

***Barber* - Dinâmico (10 ciclos):** Neste tipo de sincronização observou-se um número superior de *Reads*, *Writes*, tentativas e acertos dos jogadores *premiums*, em comparação com os jogadores não *premiums*. Esta diferença pode resultar a uma maior injustiça para o jogador não *premium* ao preencher o tabuleiro, devido ao menor número de tentativas para resolverem o jogo.

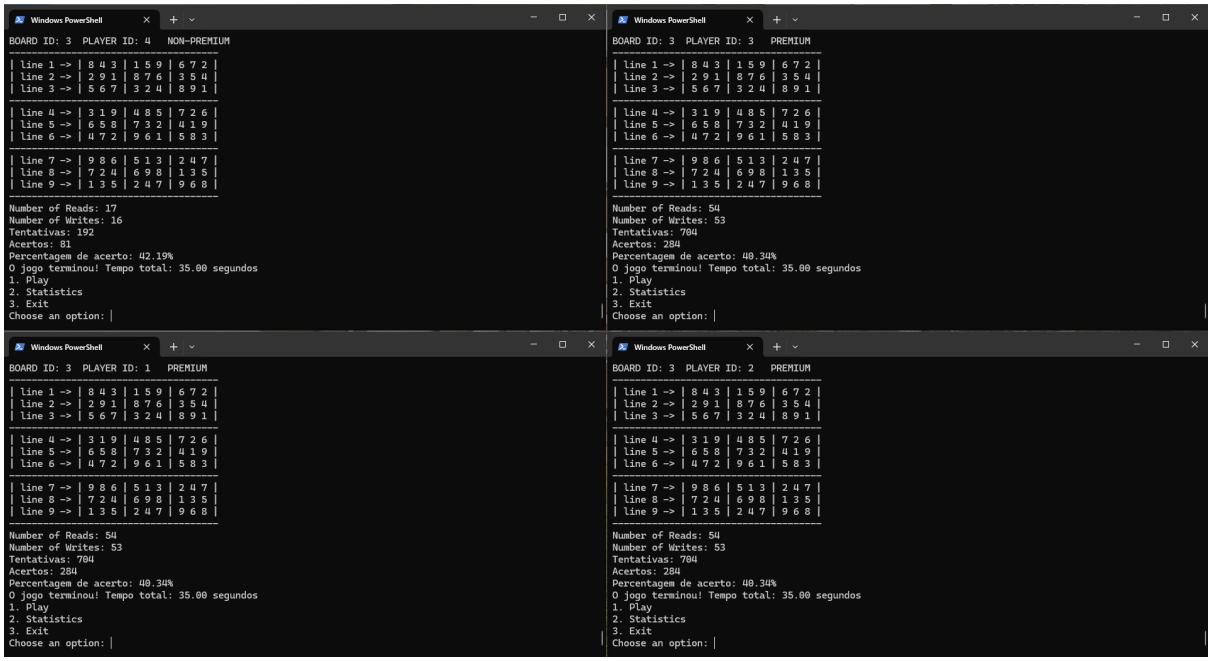

Figura 2: *Barber* - (10 *maxWaitingTime*)

***Barber* - Dinâmico (5 ciclos):** Neste tipo de sincronização, ao reduzirmos o número de ciclos de 10 para 5 no *Barber* dinâmico, observou-se uma redução no tempo total de jogo e uma menor percentagem de acertos por parte dos jogadores não *premium* em comparação com os jogadores *premium*.

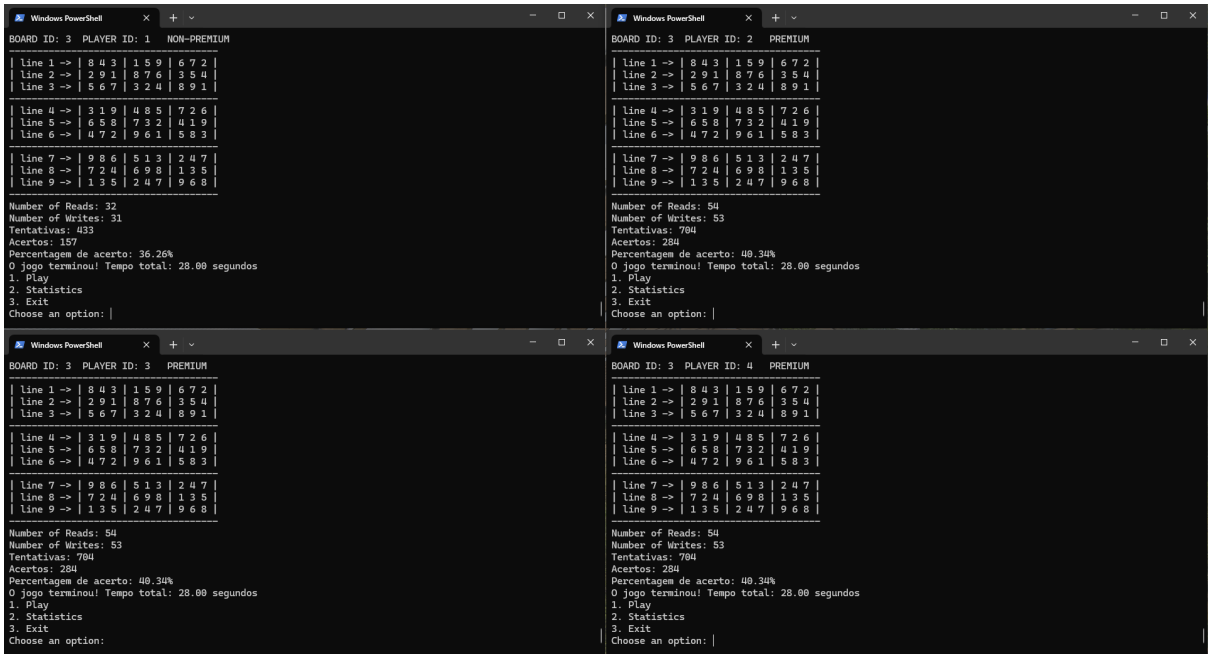

Figura 3: *Barber* - (5 *maxWaitingTime*)

f

4.2.3 *Barber* - Estático

Neste tipo de sincronização, concluímos que é injusta para os jogadores não *premium*, pois os jogadores *premium* têm sempre vantagem no preenchimento do tabuleiro. Os resultados mostram que os jogadores não *premium* não conseguem realizar nenhuma tentativa de preenchimento, enquanto que os jogadores *premium* preenchem todas as linhas do tabuleiro.

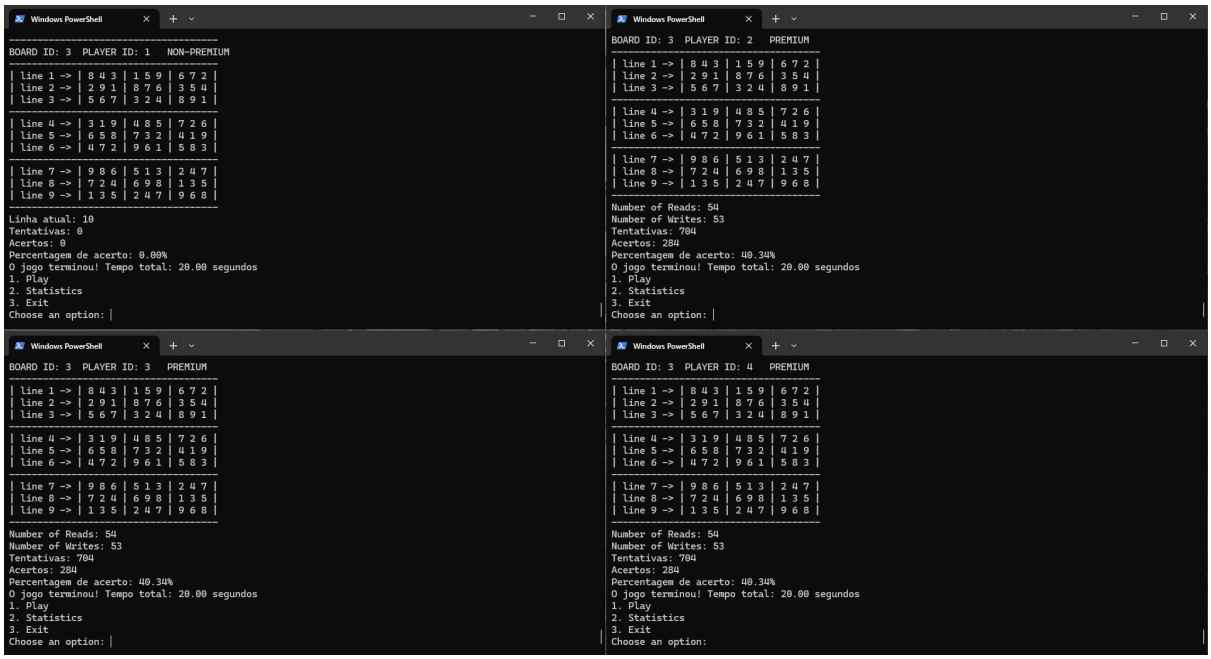

Figura 4: *Barber* - (Estático)

4.2.4 *Barber* - *FIFO*

Neste tipo de sincronização, não existem prioridades entre os diferentes tipos de jogadores, uma vez que é utilizado um algoritmo de primeiro a chegar, primeiro a sair. A análise aos resultados dos testes revelou um equilíbrio justo entre os jogadores *premium* e não *premium*, dado que ambos apresentam números iguais de *Reads*, *Writes*, tentativas e acertos.

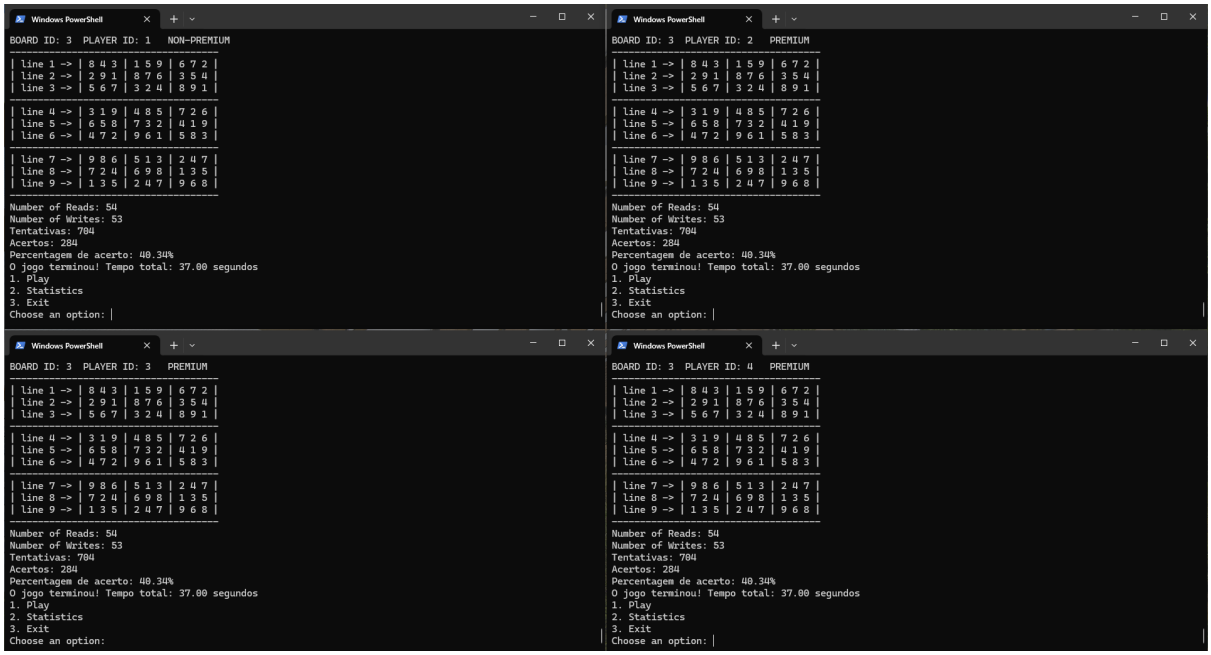

Figura 5: *Barber* - (FIFO)

4.3 As soluções apresentadas apresentam uma maior preocupação no uso justo e equilibrado dos recursos ou na eficiência geral do sistema?

Podemos afirmar que a solução apresentada tem principal foco no uso justo e equilibrado dos recursos, como demonstrado anteriormente com a gestão de prioridades. Apesar disso, o grupo teve em conta a preocupação de garantir um desempenho aceitável, utilizando *threads* e comunicação *TCP* que permitisse diversos clientes em simultâneo.

4.3.1 Uso Justo e Equilibrado dos Recursos

Prioridade entre Jogadores *Premium* e não *Premium*:

- A solução implementa filas com prioridades (estáticas e dinâmicas) para garantir que os jogadores *premium* tenham uma vantagem significativa, mas sem bloquear completamente o acesso aos jogadores não *premium*.
- Existe um equilíbrio visível nas implementações de sincronização (Leitores e Escritores e *Barber Shop*), que permitem um tratamento diferenciado mas respeitando a participação de todos os jogadores.

Sincronização Consistente:

- A utilização de barreiras e *mutexes* demonstra uma preocupação em assegurar o acesso sincronizado de todos os jogadores ao estado atualizado do tabuleiro, evitando conflitos e vantagem.

4.3.2 Eficiência Geral do Sistema

***Sockets TCP*:**

- A escolha dos *sockets TCP* garantem uma comunicação fiável entre cliente e servidor, minimizando perdas de informação na troca de pedidos.
- A criação de *threads* para gerir diversos clientes permitiu processamento paralelo eficaz, apesar de ser uma solução menos escalável.

4.4 Limitações das soluções apresentadas

4.4.1 Bloqueios Excessivos

Grande parte das situações os jogadores não *premium* perdem a prioridade para os jogadores *premium* resultando em bloqueios excessivos. Alem disso em cenários que exista muita concorrência, onde diversos clientes competem simultaneamente pelo acesso a recursos partilhados, como o tabuleiro ou as filas pode levar a condições de *deadlock* em casos extremos se a lógica de sincronização não for gerida corretamente.

4.4.2 Tempos de espera longos

Os jogadores não *premium* podem ter tempos de espera longos, para entrarem em salas de certos modos de jogo, devido à prioridade atribuída aos jogadores *premium*. Nos modos de prioridade dinâmica, a lógica de reorganização baseada no *maxWaitingTime* vai alterando continuamente a ordem natural da fila.

4.4.3 Atraso na Sincronização entre Jogadores

A utilização de barreiras e mecanismos de sincronização, como *mutexes* e semáforos garante a consistência do estado do tabuleiro e a sincronização para todos os jogado-

res. Contudo, estes mecanismos podem criar *bottlenecks* resultando em *delay* podendo comprometer a sincronização global.

5 Conclusão

Este projeto, revelou-se crucial para pôr em prática os conhecimentos adquiridos em aula sobre sincronização. Ao longo do desenvolvimento foram aplicados diferentes mecanismos de sincronização, permitindo-nos compreender a sua importância na sincronização de tarefas e na garantia da consistência dos dados. Ao longo deste projeto, enfrentamos algumas dificuldades com a sincronização, o que nos levou a repensar estratégias e adotar abordagens alternativas para superar os desafios. Este trabalho não só consolidou as nossas bases teóricas, como também aprofundou os nossos conhecimentos, preparando-nos melhor para projetos futuros.

Referências

[1] José Alves Marques, Paulo Ferreira, Carlos Ribeiro, Luís Veiga, and Rodrigo Rodrigues. *Sistemas Operativos*. FCA - Editora de Informática, Lisboa, Portugal, 2 edition, 2023.

A Anexo A: Código-Fonte do Projeto

A.1 Makefile

```
CC = gcc
CFLAGS = -g -c -Wall -Werror

# Paths
CLIENT_SRC = client/src
CLIENT_CONFIG = client/config
CLIENT_LOGS = client/logs
SERVER_SRC = server/src
SERVER_CONFIG = server/config
SERVER_LOGS = server/logs
UTILS_LOGS = utils/logs
UTILS_PARSON = utils/parson
UTILS_NETWORK = utils/network
UTILS_QUEUES = utils/queues

# Object files for client, server, and utilities
CLIENT_OBJS = $(CLIENT_SRC)/client.o $(CLIENT_SRC)/client-comms.o $(
    CLIENT_SRC)/client-game.o $(CLIENT_SRC)/client-menus.o $(
    CLIENT_CONFIG)/config.o $(CLIENT_LOGS)/logs.o
SERVER_OBJS = $(SERVER_SRC)/server.o $(SERVER_SRC)/server-comms.o $(
    SERVER_SRC)/server-game.o $(SERVER_SRC)/server-barber.o $(SERVER_SRC
    )/server-barrier.o $(SERVER_SRC)/server-readerWriter.o $(SERVER_SRC)
    /server-statistics.o $(SERVER_CONFIG)/config.o $(SERVER_LOGS)/logs.o
UTIL_OBJS = $(UTILS_LOGS)/logs-common.o $(UTILS_PARSON)/parson.o $(
    UTILS_NETWORK)/network.o $(UTILS_QUEUES)/queues.o

# Targets
all: server client

# Client build
client: $(CLIENT_OBJS) $(UTIL_OBJS)
    $(CC) -o client.exe $(CLIENT_OBJS) $(UTIL_OBJS) -lpthread

```

```
28 # Compile client object files
29 $(CLIENT_SRC)/client.o: $(CLIENT_SRC)/client.c $(CLIENT_CONFIG)/config.
       h
30     $(CC) $(CFLAGS) $(CLIENT_SRC)/client.c -o $@
31
32 $(CLIENT_SRC)/client-comms.o: $(CLIENT_SRC)/client-comms.c $(CLIENT_SRC
       )/client-comms.h
33     $(CC) $(CFLAGS) $(CLIENT_SRC)/client-comms.c -o $@
34
35 $(CLIENT_SRC)/client-game.o: $(CLIENT_SRC)/client-game.c $(CLIENT_SRC)/
       client-game.h
36     $(CC) $(CFLAGS) $(CLIENT_SRC)/client-game.c -o $@
37
38 $(CLIENT_SRC)/client-menus.o: $(CLIENT_SRC)/client-menus.c $(CLIENT_SRC
       )/client-menus.h
39     $(CC) $(CFLAGS) $(CLIENT_SRC)/client-menus.c -o $@
40
41 $(CLIENT_CONFIG)/config.o: $(CLIENT_CONFIG)/config.c $(CLIENT_CONFIG)/
       config.h
42     $(CC) $(CFLAGS) $(CLIENT_CONFIG)/config.c -o $@
43
44 $(CLIENT_LOGS)/logs.o: $(CLIENT_LOGS)/logs.c $(CLIENT_LOGS)/logs.h
45     $(CC) $(CFLAGS) $(CLIENT_LOGS)/logs.c -o $@
46
47 # Server build
48 server: $(SERVER_OBJS) $(UTIL_OBJS)
49     $(CC) -o server.exe $(SERVER_OBJS) $(UTIL_OBJS) -lpthread
50
51 # Compile server object files
52 $(SERVER_SRC)/server.o: $(SERVER_SRC)/server.c $(SERVER_CONFIG)/config.
       h $(SERVER_SRC)/server-comms.h $(SERVER_SRC)/server-game.h
53     $(CC) $(CFLAGS) $(SERVER_SRC)/server.c -o $@
54
55 $(SERVER_SRC)/server-comms.o: $(SERVER_SRC)/server-comms.c $(SERVER_SRC
       )/server-comms.h
56     $(CC) $(CFLAGS) $(SERVER_SRC)/server-comms.c -o $@
57
58 $(SERVER_SRC)/server-game.o: $(SERVER_SRC)/server-game.c $(SERVER_SRC)/
       server-game.h
```

```
59      $(CC) $(CFLAGS) $(SERVER_SRC)/server-game.c -o $@
60
61  $(SERVER_SRC)/server-barber.o: $(SERVER_SRC)/server-barber.c $(
        SERVER_SRC)/server-barber.h
62      $(CC) $(CFLAGS) $(SERVER_SRC)/server-barber.c -o $@
63
64  $(SERVER_SRC)/server-barrier.o: $(SERVER_SRC)/server-barrier.c $(
        SERVER_SRC)/server-barrier.h
65      $(CC) $(CFLAGS) $(SERVER_SRC)/server-barrier.c -o $@
66
67  $(SERVER_SRC)/server-readerWriter.o: $(SERVER_SRC)/server-readerWriter.
        c $(SERVER_SRC)/server-readerWriter.h
68      $(CC) $(CFLAGS) $(SERVER_SRC)/server-readerWriter.c -o $@
69
70  $(SERVER_SRC)/server-statistics.o: $(SERVER_SRC)/server-statistics.c $(
        SERVER_SRC)/server-statistics.h
71      $(CC) $(CFLAGS) $(SERVER_SRC)/server-statistics.c -o $@
72
73  $(SERVER_CONFIG)/config.o: $(SERVER_CONFIG)/config.c $(SERVER_CONFIG)/
        config.h
74      $(CC) $(CFLAGS) $(SERVER_CONFIG)/config.c -o $@
75
76  $(SERVER_LOGS)/logs.o: $(SERVER_LOGS)/logs.c $(SERVER_LOGS)/logs.h
77      $(CC) $(CFLAGS) $(SERVER_LOGS)/logs.c -o $@
78
79  # Compile utility object files
80  $(UTILS_LOGS)/logs-common.o: $(UTILS_LOGS)/logs-common.c $(UTILS_LOGS)/
        logs-common.h
81      $(CC) $(CFLAGS) $(UTILS_LOGS)/logs-common.c -o $@
82
83  $(UTILS_PARSON)/parson.o: $(UTILS_PARSON)/parson.c $(UTILS_PARSON)/
        parson.h
84      $(CC) $(CFLAGS) $(UTILS_PARSON)/parson.c -o $@
85
86  $(UTILS_NETWORK)/network.o: $(UTILS_NETWORK)/network.c $(UTILS_NETWORK)
        /network.h
87      $(CC) $(CFLAGS) $(UTILS_NETWORK)/network.c -o $@
88
```

```
89 $(UTILS_QUEUES)/queues.o: $(UTILS_QUEUES)/queues.c $(UTILS_QUEUES)/
       queues.h
90     $(CC) $(CFLAGS) $(UTILS_QUEUES)/queues.c -o $@
91
92 # Clean up
93 clean:
94     rm -f $(SERVER_SRC)/*.o $(SERVER_CONFIG)/*.o $(SERVER_LOGS)/*.o
          server.exe $(CLIENT_SRC)/*.o $(CLIENT_CONFIG)/*.o $(CLIENT_LOGS)
          /*.o client.exe $(UTILS_LOGS)/*.o $(UTILS_PARSON)/*.o $(
          UTILS_NETWORK)/*.o $(UTILS_QUEUES)/*.o
```

A.2 Readme

```
1  # Sudoku_SO
2  To compile the project:
3  make all
4  make server (compile server only)
5  make client (compile client only)
6
7  To start the server:
8  ./server.exe server/config/server.conf
9
10 Server Variables (server.conf):
11 -SERVER_PORT - port on where the server gets hosted
12 -GAME_PATH - path for storing sudoku games (leave this by default)
13 -SERVER_LOG_PATH - path for logging (leave this by default)
14 -MAX_ROOMS - maximum number of rooms that can be created
15 -MAX_PLAYERS_PER_ROOM - maximum number of players in each room created
16 -MAX_PLAYERS_ON_SERVER - maximum number of players that can connect to
       the server
17 -MAX_WAITING_TIME - maximum time that a player waits on queue (for
       barber shop with dynamic priorities)
18
19 To start the client:
20 ./client.exe client/config/client.conf
21
22 Client Variables (client.conf):
23 SERVER_IP - ip where to connect to the server
```

```
SERVER_PORT - port where to connect to the server (same as in server.
    conf)
SERVER_HOSTNAME - hostname to be translated (not yet implemented)
LOG_PATH - path for client logging (leave this by default)
IS_MANUAL - switch to make client solve sudoku manually or
    automatically (0/1)
IS_PREMIUM - switch to make client premium or not premium (0/1)
DIFFICULTY = expertise of client (1-easy, 2-normal, 3-hard)

Known bugs:
Check if client/data exists. If not create data inside client.

Check memory leaks:
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --
    verbose --log-file=valgrind-out-server.txt ./server.exe server/
    config/server.conf
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --
    verbose --log-file=valgrind-out-client.txt ./client.exe client/
    config/client.conf
```

A.3 Server

A.3.1 config/server.conf

```
SERVER_PORT = 8080
GAME_PATH = server/data/games.json
SERVER_LOG_PATH = server/data/logs.json
MAX_ROOMS = 5
MAX_PLAYERS_PER_ROOM = 4
MAX_PLAYERS_ON_SERVER = 20
MAX_WAITING_TIME = 5
```

A.3.2 config/config.h

```
#ifndef CONFIG_H
#define CONFIG_H

```

```c
#include <stdbool.h>
#include <semaphore.h>
#include <pthread.h>

#include "../../utils/queues/queues.h"

/**
 * Estrutura que representa um jogo, incluindo o tabuleiro e a
    solu   o correta.
 *
 * @param id O identificador  nico  do jogo.
 * @param board O tabuleiro de jogo, representado como uma matriz de 9
    x9.
 * @param solution A solu   o correta do jogo, tamb m representada
    como uma matriz de 9x9.
 * @param currentLine O n mero da linha atual a resolver no jogo.
 */

typedef struct {
    int id;
    char board[9][9];
    char solution[9][9];
    int currentLine;
} Game;

// Estrutura que cont m dados do cliente, incluindo o descritor de
   socket e a configura  o do servidor.
typedef struct {
    int socket_fd;
    int clientID;
    bool isPremium;
    bool startAgain;
    // self semaphore to be used on barber shop
    sem_t selfSemaphore;
} Client;

/**
 * Estrutura que representa uma sala de jogo.
 *
```

```
39  * @param id O identificador  nico  da sala.
40  * @param maxClients O n mero  m ximo de jogadores que a sala pode
        conter.
41  * @param numClients O n mero atual de jogadores na sala.
42  * @param Clients Um pointer para um array que cont m os IDs dos
        jogadores na sala.
43  * @param game Um pointer para o jogo associado    sala.
44  */
45
46 typedef struct {
47     int id;
48     int maxClients;
49     int numClients;
50     Client **clients;
51     int timer;
52     bool isGameRunning;
53     bool isSinglePlayer;
54     bool isFinished;
55     Game *game;
56     time_t startTime;
57     double elapsedTime;
58
59     pthread_mutex_t timerMutex;
60     pthread_mutex_t mutex;
61
62     // Priority Queue
63     PriorityQueue *enterRoomQueue;
64
65     // Reader-writer locks
66     pthread_mutex_t readMutex;
67     pthread_mutex_t writeMutex;
68     sem_t writeSemaphore;
69     sem_t readSemaphore;
70     sem_t nonPremiumWriteSemaphore;
71     int readerCount;
72     int writerCount;
73
74     // Priority queue barbershop
75     int customers;
```

```
76      pthread_mutex_t barberShopMutex;
77      sem_t costumerSemaphore;
78      sem_t costumerDoneSemaphore;
79      sem_t barberDoneSemaphore;
80      PriorityQueue *barberShopQueue;
81      pthread_t barberThread;
82
83      // bool to decide if the game is reader-writer or barbershop
84      bool isReaderWriter;
85      int priorityQueueType; // 0 static priority, 1 dynamic priority, 2
            FIFO
86      int maxWaitingTime;
87
88      // barrier to start the game and end the game
89      int waitingCount;
90      sem_t mutexSemaphore;
91      sem_t turnsTileSemaphore1;
92      sem_t turnsTileSemaphore2;
93
94      bool savedStatistics;
95
96  } Room;
97
98
99  /**
100  * Estrutura que representa a configura  o do servidor.
101  *
102  * @param serverPort O n mero da porta que o servidor utiliza para
         comunica  o.
103  * @param gamePath O caminho para o ficheiro que cont m os dados do
         jogo.
104  * @param logPath O caminho para o ficheiro onde os logs do servidor
         s  o guardados.
105  * @param maxRooms O n mero  m ximo de salas que o servidor pode gerir
         .
106  * @param maxClientsPerRoom O n mero  m ximo de jogadores que cada
         sala pode conter.
107  * @param numRooms O n mero atual de salas criadas no servidor.
108  * @param rooms Um pointer para um array de pointers de `Room`,
```

```
109  * que representa as salas de jogo geridas pelo servidor.
110  */
111
112 typedef struct {
113     int serverPort;
114     char gamePath[256];
115     char logPath[256];
116     int maxRooms;
117     int maxClientsPerRoom;
118     int maxClientsOnline;
119     int numClientsOnline;
120     int numRooms;
121     int maxWaitingTime;
122
123     Room **rooms;
124     Client **clients;
125
126     // producer-consumer for writing logs
127     sem_t mutexLogSemaphore; // mutex to grant exclusive access
128     sem_t itemsLogSemaphore; // sempaphore to signal when there are
            items to consume
129     sem_t spacesSemaphore; // semaphore to signal when there are spaces
             to produce
130
131     char logBuffer[10][256]; // buffer to store log messages
132
133     // mutex
134     pthread_mutex_t mutex;
135 } ServerConfig;
136
137 typedef struct {
138     Client *client;
139     ServerConfig *config;
140 } client_data;
141
142
143 // Obter a configura  o do servidor
144 ServerConfig *getServerConfig(char *configPath);
145
```

```
146 // Adicionar um cliente    lista de clientes online
147 void addClient(ServerConfig *config, Client *client);
148
149 // Remover um cliente da lista de clientes online
150 void removeClient(ServerConfig *config, Client *client);
151
152 #endif // CONFIG_H
```

A.3.3 config/config.c

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <errno.h>
6  #include "config.h"
7  #include "../logs/logs.h"
8  #include "../../utils/logs/logs-common.h"
9
10 /**
11  * L   as configura  es do servidor a partir de um ficheiro de
        configura  o e
12  * inicializa uma estrutura `ServerConfig` com os valores lidos.
13  *
14  * @param configPath O caminho para o ficheiro de configura  o que
        cont m as defini  es do servidor.
15  * @return Um pointer para uma estrutura `ServerConfig` inicializada,
        ou NULL se a aloca  o de mem ria falhar.
16  *
17  * @details Esta fun  o faz o seguinte:
18  * - Aloca mem ria para uma estrutura `ServerConfig` e inicializa os
        seus campos com zeros.
19  * - Abre o ficheiro de configura  o especificado em modo de leitura.
20  * - L   as defini  es do ficheiro, como a porta do servidor, o
        caminho do jogo, o caminho do log,
21  *   o n mero m ximo de salas, e o n mero m ximo de jogadores por
        sala,
22  *   e preenche os respetivos campos da estrutura.
```

```
23  * - Aloca mem ria para um array de pointers de ‘Room‘ e inicializa
        cada pointer a NULL.
24  * - Regista o evento de in cio do servidor no ficheiro de log.
25  * - Imprime as configura  es do servidor na consola.
26  * - Se ocorrer um erro ao abrir o ficheiro ou a alocar mem ria,
27 *    imprime uma mensagem de erro e termina o programa.
28  */
29
30 ServerConfig *getServerConfig(char *configPath) {
31
32     ServerConfig *config = (ServerConfig *)malloc(sizeof(ServerConfig))
           ;
33     if (config == NULL) {
34         fprintf(stderr, "Memory allocation failed\n");
35         return NULL;
36     }
37     memset(config, 0, sizeof(ServerConfig));  // Initialize Room struct
38
39     // Abre o ficheiro 'config.txt' em modo de leitura
40     FILE *file;
41     file = fopen(configPath, "r");
42
43     // Se o ficheiro n o existir, imprime uma mensagem de erro e
           termina o programa
44     if (file == NULL) {
45         fprintf(stderr, "Couldn't open %s: %s\n", configPath, strerror(
               errno));
46         exit(1);
47     }
48
49     char line[256];
50
51     if (fgets(line, sizeof(line), file) != NULL) {
52         // Remover a nova linha, se houver
53         line[strcspn(line, "\n")] = 0;
54         sscanf(line, "SERVER_PORT = %d", &config->serverPort);
55     }
56
57     if (fgets(line, sizeof(line), file) != NULL) {
```

```
58          // Remover a nova linha, se houver
59          line[strcspn(line, "\n")] = 0;
60          sscanf(line, "GAME_PATH = %s", config->gamePath);
61      }
62
63      if (fgets(line, sizeof(line), file) != NULL) {
64          // Remover a nova linha, se houver
65          line[strcspn(line, "\n")] = 0;
66          sscanf(line, "SERVER_LOG_PATH = %s", config->logPath);
67      }
68
69      if (fgets(line, sizeof(line), file) != NULL) {
70          // Remover a nova linha, se houver
71          line[strcspn(line, "\n")] = 0;
72          sscanf(line, "MAX_ROOMS = %d", &config->maxRooms);
73      }
74
75      if (fgets(line, sizeof(line), file) != NULL) {
76          // Remover a nova linha, se houver
77          line[strcspn(line, "\n")] = 0;
78          sscanf(line, "MAX_PLAYERS_PER_ROOM = %d", &config->
                maxClientsPerRoom);
79      }
80
81      if (fgets(line, sizeof(line), file) != NULL) {
82          // Remover a nova linha, se houver
83          line[strcspn(line, "\n")] = 0;
84          sscanf(line, "MAX_PLAYERS_ON_SERVER = %d", &config->
                maxClientsOnline);
85      }
86
87      if (fgets(line, sizeof(line), file) != NULL) {
88          // Remover a nova linha, se houver
89          line[strcspn(line, "\n")] = 0;
90          sscanf(line, "MAX_WAITING_TIME = %d", &config->maxWaitingTime);
91      }
92
93      // Fecha o ficheiro
94      fclose(file);
```

```
95
96      // Inicializa as salas
97      config->rooms = (Room **)malloc(config->maxRooms * sizeof(Room));
98      if (config->rooms == NULL) {
99          fprintf(stderr, "Memory allocation failed for rooms\n");
100         exit(1);  // Handle the error as appropriate
101     }
102
103     // initialize each Room pointer to NULL
104     for (int i = 0; i < config->maxRooms; i++) {
105         config->rooms[i] = NULL;  // Initialize each pointer
106     }
107
108     // initialize clients
109     config->clients = (Client **)malloc(config->maxClientsOnline *
            sizeof(Client));
110     if (config->clients == NULL) {
111         fprintf(stderr, "Memory allocation failed for clients\n");
112         exit(1);  // Handle the error as appropriate
113     }
114
115     // initialize each Client pointer to NULL
116     for (int i = 0; i < config->maxClientsOnline; i++) {
117         config->clients[i] = NULL;  // Initialize each pointer
118     }
119
120     // Inicializa o n mero de salas e jogadores online
121     config->numRooms = 0;
122     config->numClientsOnline = 0;
123
124     // producer-consumer for writing logs
125     sem_init(&config->mutexLogSemaphore, 0, 1); // mutex to grant
            exclusive access
126     sem_init(&config->itemsLogSemaphore, 0, 0); // semaphore to signal
            when there are items to consume
127     sem_init(&config->spacesSemaphore, 0, 10); // semaphore to signal
            when there are spaces to produce
128
129     // mutex
```

```
130     pthread_mutex_init(&config->mutex, NULL);
131
132     // produce log message
133     writeLogJSON(config->logPath, 0, 0, "Server started");
134
135     // Imprime as configura  es do servidor na consola
136     printf("PORTA DO SERVIDOR: %d\n", config->serverPort);
137     printf("PATH DO JOGO: %s\n", config->gamePath);
138     printf("PATH DO LOG: %s\n", config->logPath);
139     printf("MAXIMO DE JOGADORES POR SALA: %d\n", config->
            maxClientsPerRoom);
140     printf("MAXIMO DE SALAS: %d\n", config->maxRooms);
141     printf("MAXIMO DE JOGADORES ONLINE: %d\n", config->maxClientsOnline
            );
142     printf("MAXIMO DE TEMPO DE ESPERA: %d\n", config->maxWaitingTime);
143
144     // Retorna a vari vel config
145     return config;
146 }
147
148 void addClient(ServerConfig *config, Client *client) {
149
150     // add client to clients array
151     for (int i = 0; i < config->maxClientsOnline; i++) {
152         if (config->clients[i] == NULL) {
153             config->clients[i] = client;
154             break;
155         }
156     }
157
158     config->numClientsOnline++;
159     //printf("NUMERO DE JOGADORES ONLINE: %d\n", config->
            numClientsOnline);
160 }
161
162 void removeClient(ServerConfig *config, Client *client) {
163
164     // remove client from clients array
165     for (int i = 0; i < config->maxClientsOnline; i++) {
```

```
166         if (config->clients[i] != NULL && config->clients[i]->socket_fd
                == client->socket_fd) {
167             free(config->clients[i]);
168             config->clients[i] = NULL;
169             break;
170         }
171     }
172
173     // shift clients
174     for (int i = 0; i < config->maxClientsOnline; i++) {
175         if (config->clients[i] == NULL && config->clients[i + 1] !=
               NULL) {
176             config->clients[i] = config->clients[i + 1];
177             config->clients[i + 1] = NULL;
178         }
179     }
180
181     // decrement number of clients online
182     config->numClientsOnline--;
183 }
```

A.3.4 logs/logs.h

```
1  #ifndef LOGS_H
2  #define LOGS_H
3
4  #include "../config/config.h"
5
6  // Fun  o externa para registar um erro no log e terminar o programa.
7  void err_dump(ServerConfig *config, int idJogo, int idJogador, char *
       msg, char *event);
8
9  // add message to log buffer
10 void addLogMessage(ServerConfig *config, char *message);
11
12 // get message from log buffer
13 char *getLogMessage(ServerConfig *config);
14
15 // remove message from log buffer
```

```
16 void removeLogMessage(ServerConfig *config);
17
18 // consume log message
19 void *consumeLog(void *arg);
20
21 // produce log message
22 void produceLog(ServerConfig *config, char *msg, char* event, int
       idJogo, int idJogador);
23
24 #endif // LOGS_H
```

A.3.5 logs/logs.c

```
1  #include <string.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include "logs.h"
5  #include "../../utils/logs/logs-common.h"
6
7
8  void err_dump(ServerConfig * config, int idJogo, int idJogador, char *
       msg, char *event) {
9
10     // produce log message
11     produceLog(config, msg, event, idJogo, idJogador);
12
13     // imprime a mensagem de erro e termina o programa
14     perror(msg);
15     exit(1);
16 }
17
18 // add message to log buffer
19 void addLogMessage(ServerConfig *config, char *message) {
20
21     // add message to log buffer
22     for (int i = 0; i < 10; i++) {
23         if (config->logBuffer[i][0] == '\0') {
24             strcpy(config->logBuffer[i], message);
25             break;
```

```
26          }
27      }
28  }
29
30  // get message from log buffer
31  char *getLogMessage(ServerConfig *config) {
32
33      // get message from log buffer
34      for (int i = 0; i < 10; i++) {
35          if (config->logBuffer[i][0] != '\0') {
36              return config->logBuffer[i];
37          }
38      }
39
40      return NULL;
41  }
42
43  // remove message from log buffer
44  void removeLogMessage(ServerConfig *config) {
45
46      // remove message from log buffer
47      for (int i = 0; i < 10; i++) {
48          if (config->logBuffer[i][0] != '\0') {
49              memset(config->logBuffer[i], 0, sizeof(config->logBuffer[i
                    ]));
50              break;
51          }
52      }
53  }
54
55
56  void *consumeLog(void *arg) {
57
58      while (1) {
59
60          ServerConfig* config = (ServerConfig*) arg; // get config
61
62          sem_wait(&config->itemsLogSemaphore);   // check if there are
                items to consume
```

```
63
64          //printf("SERVER detects a log to write!\n");
65
66          sem_wait(&config->mutexLogSemaphore);   // lock the buffer
67
68          //printf("SERVER writing a log message\n");
69
70          char *message = getLogMessage(config);  // get message from
                buffer
71          if (message != NULL) {
72
73              // break message into gameID, playerID and message through
                    \n
74              char *token = strtok(message, "\n");
75              int gameID = atoi(token);
76              token = strtok(NULL, "\n");
77              int playerID = atoi(token);
78              token = strtok(NULL, "\n");
79              char *message = token;
80
81              writeLogJSON(config->logPath, gameID, playerID, message);
                    // write the message on the log
82              removeLogMessage(config);           // remove message from
                    buffer
83          }
84          sem_post(&config->mutexLogSemaphore);   // unlock the buffer
85
86          //printf("SERVER finished writing a log message\n");
87          sem_post(&config->spacesSemaphore);     // signal that there
                are spaces to produce
88         // printf("SERVER signaled that there are spaces to produce\n");
89      }
90
91      return NULL;
92  }
93
94  void produceLog(ServerConfig *config, char *msg, char* event, int
        idJogo, int idJogador) {
95
```

```
96      //printf("CLIENT %d : wants to write a log message\n", idJogador);
97
98      sem_wait(&config->spacesSemaphore);     // check if there is space
            to produce
99
100     //printf("THERE's space to CLIENT %d write a log message!\n",
            idJogador);
101     sem_wait(&config->mutexLogSemaphore);   // lock the buffer
102
103     //printf("CLIENT %d : writing a log message\n", idJogador);
104     char *message = concatenateInfo(msg, event, idJogo, idJogador);  //
            concatenate info
105     addLogMessage(config, message);         // add message to buffer
106     free(message);                          // free message
107
108     sem_post(&config->mutexLogSemaphore);   // unlock the buffer
109     sem_post(&config->itemsLogSemaphore);   // signal that there are
            items to consume
110 }
```

A.3.6 src/server-barber.h

```
1  #ifndef SERVER_BARBER_H
2  #define SERVER_BARBER_H
3
4  #include "../config/config.h"
5
6  void enterBarberShop(Room *room, Client *client);
7
8  void leaveBarberShop(Room *room, Client *client);
9
10 void barberCut(Room *room);
11
12 void barberIsDone(Room *room);
13
14 void *handleBarber(void *arg);
15
16 #endif // SERVER_BARBER_H
```

A.3.7 src/server-barber.c

```
#include <stdio.h>
#include "server-barber.h"

void enterBarberShop(Room *room, Client *client) {

    // initialize self semaphore
    sem_init(&client->selfSemaphore, 0, 0);

    // lock the barber shop mutex
    pthread_mutex_lock(&room->barberShopMutex);

    // increment the number of customers
    room->customers++;

    //printf("CLIENT %d ENTERED THE BARBER SHOP\n", client->clientID);
    // add the client to the barber shop queue
    if (room->priorityQueueType == 0) { // static priority
        enqueueWithPriority(room->barberShopQueue, client->clientID,
            client->isPremium);
    } else if (room->priorityQueueType == 1) { // dynamic priority
        enqueueWithPriority(room->barberShopQueue, client->clientID,
            client->isPremium);
        updateQueueWithPriority(room->barberShopQueue, room->
            maxWaitingTime);
    } else { // FIFO
        enqueueFifo(room->barberShopQueue, client->clientID);
    }

    // unlock the barber shop mutex
    pthread_mutex_unlock(&room->barberShopMutex);

    //printf("CLIENT %d IS WAITING FOR THE BARBER1\n", client->clientID
        );

    // wait for the costumer semaphore
    sem_post(&room->costumerSemaphore);

```

```
34      //printf("CLIENT %d IS WAITING FOR THE BARBER2\n", client->clientID
            );
35
36      // wait for the self semaphore to be unlocked by the dequeue
            function
37      sem_wait(&client->selfSemaphore);
38  }
39
40  void leaveBarberShop(Room *room, Client *client) {
41
42      //printf("CLIENT %d IS DONE WITH THE BARBER\n", client->clientID);
43
44      // signal that the costumer is done
45      sem_post(&room->costumerDoneSemaphore);
46
47      //printf("CLIENT %d IS WAITING FOR THE BARBER TO BE DONE\n", client
            ->clientID);
48
49      // wait for the barber to be done
50      sem_wait(&room->barberDoneSemaphore);
51
52      // lock the barber shop mutex
53      pthread_mutex_lock(&room->barberShopMutex);
54
55      // decrement the number of customers
56      room->customers--;
57
58      // unlock the barber shop mutex
59      pthread_mutex_unlock(&room->barberShopMutex);
60  }
61
62  void barberCut(Room *room) {
63
64      //printf("BARBER IS WAITING FOR A COSTUMER\n");
65
66
67
68      // wait for the costumer semaphore
69      sem_wait(&room->costumerSemaphore);
```

```
70
71      // lock the barber shop mutex
72      pthread_mutex_lock(&room->barberShopMutex);
73
74      //printf("NUMBER OF CUSTOMERS: %d\n", room->customers);
75
76      // dequeue the client from the barber shop queue
77      int clientID = dequeue(room->barberShopQueue);
78      Client *client;
79
80      if (clientID == -1) {
81          printf("NO CLIENTS IN THE BARBER SHOP\n");
82          // unlock the barber shop mutex
83          pthread_mutex_unlock(&room->barberShopMutex);
84          return;
85      }
86
87      // unlock the self semaphore
88      for (int i = 0; i < room->maxClients; i++) {
89          if (room->clients[i]->clientID == clientID) {
90              client = room->clients[i];
91              break;
92          }
93      }
94
95      // unlock the barber shop mutex
96      pthread_mutex_unlock(&room->barberShopMutex);
97
98      // post the self semaphore
99      sem_post(&client->selfSemaphore);
100
101     // delete the self semaphore
102     sem_destroy(&client->selfSemaphore);
103 }
104
105 void barberIsDone(Room *room) {
106
107     //printf("BARBER IS DONE WITH THE COSTUMER1\n");
108     // wait for the costumer done semaphore
```

```
109     sem_wait(&room->costumerDoneSemaphore);
110
111     //printf("BARBER IS DONE WITH THE COSTUMER2\n");
112
113     // signal that the barber is done
114     sem_post(&room->barberDoneSemaphore);
115 }
116
117 void *handleBarber(void *arg) {
118
119     while (1) {
120
121         Room *room = (Room *)arg;
122
123         barberCut(room);
124         barberIsDone(room);
125     }
126 }
```

A.3.8 src/server-barrier.h

```
1  #ifndef SERVER_BARRIER_H
2  #define SERVER_BARRIER_H
3
4  #include "../config/config.h"
5
6  void acquireTurnsTileSemaphore(Room *room, Client *client);
7
8  void releaseTurnsTileSemaphore(Room *room, Client *client);
9
10 #endif // SERVER_BARRIER_H
```

A.3.9 src/server-barrier.c

```
1  #include <stdio.h>
2  #include "server-barrier.h"
3
4  void acquireTurnsTileSemaphore(Room *room, Client *client) {
```

```
5
6       // lock the mutex
7       sem_wait(&room->mutexSemaphore);
8
9       //printf("CLIENT %d ARRIVED AT THE TILE SEMAPHORE\n", client->
            clientID);
10
11      // increment the waiting count
12      room->waitingCount++;
13
14      // if all players are waiting unlock the turns tile semaphore
15      if (room->waitingCount == room->numClients) {
16
17          //printf("CLIENT %d ARRIVED AND IT'S THE LAST ONE\n", client->
                clientID);
18          //printf("RELEASING TURNS TILE SEMAPHORE\n");
19
20          // need to loop n times to unlock the turns tile semaphore
21          for (int i = 0; i < room->numClients; i++) {
22              sem_post(&room->turnsTileSemaphore1);
23          }
24      }
25
26      // unlock the mutex
27      sem_post(&room->mutexSemaphore);
28
29      //printf("CLIENT %d IS WAITING FOR TURNS TILE SEMAPHORE\n", client
            ->clientID);
30      // wait for the turns tile semaphore
31      sem_wait(&room->turnsTileSemaphore1);
32
33      //printf("CLIENT %d IS DONE WAITING FOR TURNS TILE SEMAPHORE\n",
            client->clientID);
34  }
35
36  void releaseTurnsTileSemaphore(Room *room, Client *client) {
37
38      // lock the mutex
39      sem_wait(&room->mutexSemaphore);
```

```
40
41      //printf("CLIENT %d ARRIVED AT THE TILE SEMAPHORE\n", client->
            clientID);
42
43      // decrement the waiting count
44      room->waitingCount--;
45
46      // if all players are done unlock the turns tile semaphore
47      if (room->waitingCount == 0) {
48
49          //printf("CLIENT %d ARRIVED AND IT'S THE LAST ONE\n", client->
                clientID);
50          //printf("RELEASING TURNS TILE SEMAPHORE\n");
51
52          // need to loop n times to unlock the turns tile semaphore
53          for (int i = 0; i < room->maxClients; i++) {
54              sem_post(&room->turnsTileSemaphore2);
55          }
56      }
57
58      // unlock the mutex
59      sem_post(&room->mutexSemaphore);
60
61      //printf("CLIENT %d IS WAITING FOR TURNS TILE SEMAPHORE\n", client
            ->clientID);
62      // wait for the turns tile semaphore
63      sem_wait(&room->turnsTileSemaphore2);
64      //printf("CLIENT %d IS DONE WAITING FOR TURNS TILE SEMAPHORE\n",
            client->clientID);
65
66  }
```

A.3.10 src/server-comms.h

```
1  #ifndef SERVER_COMMS_H
2  #define SERVER_COMMS_H
3
4  #include <stdbool.h>
5  #include "server-game.h"
```

```
6
7  // Gera um ID  nico  para um cliente.
8  int generateUniqueClientId();
9
10 // Fun  o para lidar com a comunica  o com um cliente.
11 void *handleClient(void *arg);
12
13 // Inicializa o socket do servidor e associa-o a um endere o.
14 void initializeSocket(struct sockaddr_in *serv_addr, int *sockfd,
       ServerConfig *config);
15
16 #endif
```

A.3.11 src/server-comms.c

```
1  #include <pthread.h>
2  #include <sys/select.h>
3  #include <time.h>
4  #include "../../utils/parson/parson.h"
5  #include "../../utils/network/network.h"
6  #include "../../utils/logs/logs-common.h"
7  #include "server-comms.h"
8  #include "../logs/logs.h"
9
10
11 static int nextClientID = 1;
12
13
14 /**
15  * Gera um ID de cliente  nico .
16  *
17  * @return Um ID de cliente  nico , incrementando o valor de '
       nextClientID'.
18  *
19  * @details Esta fun  o usa uma vari vel global 'nextClientID' para
       gerar IDs  nicos  para clientes.
20  * Cada vez que a fun  o  chamada, o valor de 'nextClientID'
       retornado e, em seguida, incrementado.
21  * Isto garante que cada cliente receba um ID  nico  e sequencial.
```

```
22  */
23
24  int generateUniqueClientId() {
25      return nextClientID++;
26  }
27
28  /**
29   * Fun   o que gere a comunica  o com um cliente ligado ao servidor.
30   *
31   * @param arg Um pointer gen rico (void *) que    convertido para um
         pointer 'client',
32   * contendo as informa  es do cliente e a configura  o do servidor.
33   * @return Retorna NULL (a fun   o    utilizada como um manipulador de
          threads,
34   * por isso o valor de retorno n o    usado).
35   *
36   * @details Esta fun   o faz o seguinte:
37   * - Obt m os dados do cliente a partir do argumento 'arg' e
         inicializa as vari veis necess rias.
38   * - Recebe o pedido de ID do cliente, gera um ID   nico , e envia-o de
         volta ao cliente.
39   * - Entra num ciclo principal onde recebe comandos do cliente e
         executa as a   es correspondentes:
40   *    - Criar um novo jogo single Client ou MultiPlayer.
41   *    - Listar e selecionar jogos ou salas existentes.
42   *    - Receber e enviar dados relacionados com o estado do jogo e
         a   es do cliente.
43   * - Gere a comunica  o com o cliente, incluindo o envio e a
         rece   o de mensagens,
44   *    e regista eventos no ficheiro de log.
45   * - Trata erros de forma apropriada, terminando a conex o e a thread
         se necess rio.
46   * - Fecha a liga   o com o cliente, liberta a mem ria alocada e
         termina a execu   o da thread.
47   */
48
49  void *handleClient(void *arg) {
50
51      // Obter dados do cliente
```

```
52      client_data *data = (client_data *)arg;
53      Client *client = data->client;
54      ServerConfig *serverConfig = data->config;
55      int newSockfd = client->socket_fd;
56
57      // receber buffer do cliente
58      char buffer[BUFFER_SIZE];
59      memset(buffer, 0, sizeof(buffer));
60
61      Room *room;
62      int currentLine = 1;
63
64      //printf("A AGUARDAR POR PEDIDOS DO CLIENTE\n");
65
66      // receber premium status
67      if (recv(newSockfd, buffer, sizeof(buffer), 0) < 0) {
68          // erro ao receber status premium
69          err_dump(serverConfig, 0, client->clientID, "can't receive
                premium status", EVENT_MESSAGE_SERVER_NOT_RECEIVED);
70      } else if (strcmp(buffer, "premium") == 0) {
71          client->isPremium = true;
72      } else {
73          client->isPremium = false;
74      }
75
76      // send id to client
77      client->clientID = generateUniqueClientId();
78      sprintf(buffer, "%d", client->clientID);
79
80      if (send(newSockfd, buffer, strlen(buffer), 0) < 0) {
81          // erro ao enviar ID do jogador
82          err_dump(serverConfig, 0, 0, "can't send client ID",
                EVENT_MESSAGE_SERVER_NOT_SENT);
83      } else {
84          printf("ID atribuido ao novo cliente: %d (%s)\n", client->
                clientID, client->isPremium ? "premium" : "not premium");
85          produceLog(serverConfig, "Conexao estabelecida com um novo
                cliente", EVENT_CONNECTION_SERVER_ESTABLISHED, 0, client->
                clientID);
```

```
86      }
87
88      bool continueLoop = true;
89
90      while (continueLoop) {
91
92          //printf("A aguardar por pedidos do cliente %d\n", client->
                clientID);
93
94          client->startAgain = false;
95          memset(buffer, 0, sizeof(buffer));
96
97          // Receber menu status
98          if (recv(newSockfd, buffer, sizeof(buffer), 0) < 0) {
99              // erro ao receber modo de jogo
100             err_dump(serverConfig, 0, client->clientID, "can't receive
                    menu status", EVENT_MESSAGE_SERVER_NOT_RECEIVED);
101         } else {
102
103             //printf("BUFFER RECEBIDO: %s\n", buffer);
104
105             // cliente quer ver as estatisticas
106             if(strcmp(buffer, "GET_STATS") == 0){
107
108                 pthread_mutex_lock(&serverConfig->mutex);
109
110                 sendRoomStatistics(serverConfig, client);
111                 client->startAgain = true;
112
113                 pthread_mutex_unlock(&serverConfig->mutex);
114
115             } else if (strcmp(buffer, "newSinglePlayerGame") == 0) {
116
117                 pthread_mutex_lock(&serverConfig->mutex);
118
119                 // criar novo jogo single player
120                 room = createRoomAndGame(serverConfig, client, true,
                        true, 0, 0);
121
```

```
122                 pthread_mutex_unlock(&serverConfig->mutex);
123
124             } else if(strcmp(buffer, "newMultiPlayerGameReadersWriters"
                    ) == 0) {
125
126                 printf("Cliente %d solicitou um novo jogo rando
                        multiplayer game com readers-writers\n", client->
                        clientID);
127
128                 // lock mutex
129                 pthread_mutex_lock(&serverConfig->mutex);
130
131                 room = createRoomAndGame(serverConfig, client, false,
                        true, 0, 0);
132
133                 // unlock mutex
134                 pthread_mutex_unlock(&serverConfig->mutex);
135
136                 // adicionar timer
137                 handleTimer(serverConfig, room, client);
138
139             } else if (strcmp(buffer, "
                    newMultiPlayerGameBarberShopStaticPriority") == 0) {
140
141
142                 pthread_mutex_lock(&serverConfig->mutex);
143
144                 room = createRoomAndGame(serverConfig, client, false,
                        true, 0, 1);
145
146                 pthread_mutex_unlock(&serverConfig->mutex);
147
148                 // adicionar timer
149                 handleTimer(serverConfig, room, client);
150
151             } else if (strcmp(buffer, "
                    newMultiPlayerGameBarberShopDynamicPriority") == 0) {
152
```

```
153                 printf("Cliente %d solicitou um novo jogo rando
                        multiplayer game com barber shop dynamic priority\n"
                        , client->clientID);
154
155                 // lock mutex
156                 pthread_mutex_lock(&serverConfig->mutex);
157
158                 room = createRoomAndGame(serverConfig, client, false,
                        true, 0, 2);
159
160                 // unlock mutex
161                 pthread_mutex_unlock(&serverConfig->mutex);
162
163                 // adicionar timer
164                 handleTimer(serverConfig, room, client);
165
166             } else if (strcmp(buffer, "newMultiPlayerGameBarberShopFIFO
                    ") == 0) {
167
168                 printf("Cliente %d solicitou um novo jogo rando
                        multiplayer game com barber shop fifo\n", client->
                        clientID);
169
170                 // lock mutex
171                 pthread_mutex_lock(&serverConfig->mutex);
172
173                 room = createRoomAndGame(serverConfig, client, false,
                        true, 0, 3);
174
175                 // unlock mutex
176                 pthread_mutex_unlock(&serverConfig->mutex);
177
178                 // adicionar timer
179                 handleTimer(serverConfig, room, client);
180
181             } else if (strcmp(buffer, "selectSinglePlayerGames") == 0
                    || strcmp(buffer, "selectMultiPlayerGames") == 0) {
182
```

```
183                 bool isSinglePlayer = strncmp(buffer, "
                        selectSinglePlayerGames", strlen("
                        selectSinglePlayerGames")) == 0;
184                 // Receber jogos existentes
185                 char buffer[BUFFER_SIZE];
186                 memset(buffer, 0, sizeof(buffer));
187
188                 // lock mutex
189                 pthread_mutex_lock(&serverConfig->mutex);
190                 // Obter jogos existentes
191                 char *games = getGames(serverConfig);
192
193                 //printf("Jogos existentes: %s\n", games);
194
195                 // unlock mutex
196                 pthread_mutex_unlock(&serverConfig->mutex);
197
198                 bool leave = false;
199
200                 while (!leave) {
201                     // Enviar jogos existentes ao cliente
202                     if (send(newSockfd, games, strlen(games), 0) < 0) {
203                         free(games);
204                         err_dump(serverConfig, 0, client->clientID, "
                                can't send existing games to client",
                                EVENT_MESSAGE_SERVER_NOT_SENT);
205                     } else {
206                         produceLog(serverConfig, "Jogos enviados para o
                                 cliente", EVENT_SERVER_GAMES_SENT, 0,
                                client->clientID);
207                     }
208
209                     memset(buffer, 0, sizeof(buffer));
210
211                     // receber ID do jogo
212                     if (recv(newSockfd, buffer, sizeof(buffer), 0) < 0)
                             {
213                         free(games);
```

```
214                     err_dump(serverConfig, 0, client->clientID, "
                            can't receive game ID from client",
                            EVENT_MESSAGE_SERVER_NOT_RECEIVED);
215                 } else if (atoi(buffer) == 0) {
216                     printf("Cliente %d voltou atras no menu\n",
                            client->clientID);
217                     leave = true;
218                     client->startAgain = true;
219                 } else {
220                       printf("Cliente %d escolheu o jogo com o ID:
                              %s\n", client->clientID, buffer);
221
222                     // obter ID do jogo
223                     int gameID = atoi(buffer);
224
225                     int synchronizationType;
226
227                     // need to receive synchronization type
228                     if (!isSinglePlayer) {
229
230                         // receive synchronization type
231                         memset(buffer, 0, sizeof(buffer));
232
233                         if (recv(newSockfd, buffer, sizeof(buffer),
                                0) < 0) {
234                             err_dump(serverConfig, 0, client->
                                    clientID, "can't receive
                                    synchronization type from client
                                    ",
                                    EVENT_MESSAGE_SERVER_NOT_RECEIVED
                                    );
235                         } else {
236                             //printf("BUFFER RECEBIDO PARA
                                    SYNCHRONIZATION TYPE: %s\n", buffer)
                                    ;
237                             if (strcmp(buffer, "
                                    newMultiPlayerGameReadersWriters")
                                    == 0) {
```

```
238                         printf("Cliente %d escolheu o jogo
                                com readers-writers\n", client->
                                clientID);
239                         synchronizationType = 0;
240                     } else if (strcmp(buffer, "
                            newMultiPlayerGameBarberShopStaticPriority
                            ") == 0) {
241                         printf("Cliente %d escolheu o jogo
                                com barber shop priority\n",
                                client->clientID);
242                         synchronizationType = 1;
243                     } else if (strcmp(buffer, "
                            newMultiPlayerGameBarberShopDynamicPriority
                            ") == 0) {
244                         printf("Cliente %d escolheu o jogo
                                com barber shop dynamic priority
                                \n", client->clientID);
245                         synchronizationType = 2;
246                     } else if (strcmp(buffer, "
                            newMultiPlayerGameBarberShopFIFO")
                            == 0) {
247                         printf("Cliente %d escolheu o jogo
                                com barber shop fifo\n", client
                                ->clientID);
248                         synchronizationType = 3;
249                     } else if (strcmp(buffer, "0") == 0) {
250                         printf("Cliente %d voltou atras no
                                menu\n", client->clientID);
251                         leave = true;
252                         client->startAgain = true;
253                         break;
254                     }
255                 }
256
257                 //printf("Synchronization type: %d\n",
                        synchronizationType);
258             }
259
260             // lock mutex
```

```
261                     pthread_mutex_lock(&serverConfig->mutex);
262
263                     room = createRoomAndGame(serverConfig, client,
                            isSinglePlayer, false, gameID,
                            synchronizationType);
264
265                     // unlock mutex
266                     pthread_mutex_unlock(&serverConfig->mutex);
267
268                     if (!isSinglePlayer) {
269                         // adicionar timer
270                         handleTimer(serverConfig, room, client);
271                     }
272
273                     leave = true;
274                 }
275             }
276
277             free(games);
278             games = NULL;
279
280         } else if (strcmp(buffer, "existingRooms") == 0) {
281               // Receber jogos existentes
282             char buffer[BUFFER_SIZE];
283             memset(buffer, 0, sizeof(buffer));
284             // Obter salas existentes
285
286             pthread_mutex_lock(&serverConfig->mutex);
287
288             char *rooms = getRooms(serverConfig);
289
290             //printf("Rooms existentes: %s\n", rooms);
291
292             pthread_mutex_unlock(&serverConfig->mutex);
293
294             bool leave = false;
295
296             while (!leave) {
297                 // Enviar salas existentes ao cliente
```

```
298                     if (send(newSockfd, rooms, strlen(rooms), 0) < 0) {
299                         free(rooms);
300                         err_dump(serverConfig, 0, client->clientID, "
                                can't send existing games to client",
                                EVENT_MESSAGE_SERVER_NOT_SENT);
301                     } else {
302                         produceLog(serverConfig, "Jogos enviados para o
                                 cliente", EVENT_SERVER_GAMES_SENT, 0,
                                client->clientID);
303                     }
304                     //printf("RECEBER IDS DAS ROOMS\n");
305                     memset(buffer, 0, sizeof(buffer));
306
307                     // receber ID da sala
308                     if (recv(newSockfd, buffer, sizeof(buffer), 0) < 0)
                            {
309                         free(rooms);
310                         err_dump(serverConfig, 0, client->clientID, "
                                can't receive room ID from client",
                                EVENT_MESSAGE_SERVER_NOT_RECEIVED);
311                     } else if (atoi(buffer) == 0) {
312                         printf("Cliente %d voltou atras no menu\n",
                                client->clientID);
313                         leave = true;
314                         client->startAgain = true;
315                     } else {
316                         int roomID = atoi(buffer);
317
318                         //printf("ROOM ID NO SERVIDOR: %d\n", roomID);
319
320                         // lock mutex
321                         pthread_mutex_lock(&serverConfig->mutex);
322
323                         // get the room
324                         room = getRoom(serverConfig, roomID, client->
                                clientID);
325
326                         // unlock mutex
327                         pthread_mutex_unlock(&serverConfig->mutex);
```

```
328
329                     // add the Client to the queue
330                     //printf("ENQUEING CLIENT %d which is %s\n",
                            client->clientID, client->isPremium ? "
                            premium" : "not premium");
331                     enqueueWithPriority(room->enterRoomQueue,
                            client->clientID, client->isPremium);
332
333                     //printf("BEFORE SLEEP\n");
334
335                     // sleep thread for test
336                     sleep(5);
337
338                     //printf("AFTER SLEEP\n");
339
340                     // Join room all clients in queue
341                     pthread_mutex_lock(&room->mutex);
342
343
344                     for (int i = 0; i < room->maxClients * 2; i++)
                            {
345
346                         if (room->enterRoomQueue->front == NULL) {
347                             break;
348
349                         } else {
350
351                             Client *clientTemp;
352
353                             int clientIDJoin = dequeue(room->
                                    enterRoomQueue);
354
355                             // get client socket
356                             for (int i = 0; i < serverConfig->
                                    numClientsOnline; i++) {
357                                 if (serverConfig->clients[i] !=
                                        NULL && serverConfig->clients[i
                                        ]->clientID == clientIDJoin) {
```

```
358                                 clientTemp = serverConfig->
                                        clients[i];
359                                 break;
360                             }
361                         }
362
363                         // check if room is full so the client
                            doesnt join
364                         //printf("CHECKING IF ROOM IS FULL FOR
                            CLIENT %d\n", clientTemp->clientID);
365                         if (room->numClients >= room->
                            maxClients) {
366
367                             // send message to client
368                             if (send(clientTemp->socket_fd, "
                                Room is full", strlen("Room is
                                full"), 0) < 0) {
369                                 err_dump(serverConfig, 0,
                                    client->clientID, "can't
                                    send message to client",
                                    EVENT_MESSAGE_SERVER_NOT_SENT
                                    );
370                             } else {
371                                 produceLog(serverConfig, "Room
                                    is full",
                                    EVENT_ROOM_NOT_JOIN, 0,
                                    clientTemp->clientID);
372                             }
373
374                             clientTemp->startAgain = true;
375
376                         } else {
377
378                             joinRoom(serverConfig, room,
                                clientTemp);
379                         }
380                     }
381                 }
382
```

```
383                     pthread_mutex_unlock(&room->mutex);
384
385                     // atualizar tempo de espera para este Client
386                     handleTimer(serverConfig, room, client);
387
388                     leave = true;
389
390                 }
391             }
392             //printf("Rooms pointer before free: %p\n", (void*)
                    rooms);
393             free(rooms);
394             //rooms = NULL;
395
396         } else if (strcmp(buffer, "closeConnection") == 0) {
397             continueLoop = false;
398             break;
399         } else if (strcmp(buffer, "0") == 0) {
400             printf("Cliente %d voltou atras no menu\n", client->
                    clientID);
401             client->startAgain = true;
402         }
403
404         if (!client->startAgain) {
405             room->isGameRunning = true;
406             room->startTime = time(NULL);
407
408             // barreira para come ar o jogo
409             if (!room->isSinglePlayer) {//Se o jogo for multiplayer
410                 acquireTurnsTileSemaphore(room, client);//Garante
                        que todas as threads estao prontas antes de
                        avan ar para a fase critica
411             }
412
413             // receber linhas do cliente
414             receiveLines(serverConfig, room, client, &currentLine);
415
416             if (!room->isSinglePlayer) {
417                 // barreira para terminar o jogo
```

```
418                     releaseTurnsTileSemaphore(room, client);
419                 }
420 
421                 // mutex para terminar o jogo
422                 pthread_mutex_lock(&serverConfig->mutex);
423 
424                 // terminar o jogo
425                 finishGame(serverConfig, room, &newSockfd);
426 
427                 // unlock mutex
428                 pthread_mutex_unlock(&serverConfig->mutex);
429             }
430         }
431     }
432 
433     // fechar a liga  o com o cliente
434     close(newSockfd);
435     printf("Conexao terminada com o cliente %d\n", client->clientID);
436     produceLog(serverConfig, "Conexao terminada com o cliente",
        EVENT_SERVER_CONNECTION_FINISH, 0, client->clientID);
437 
438     // libertar a mem ria alocada
439     removeClient(serverConfig, client);
440     free(data);
441 
442     // terminar a thread
443     pthread_exit(NULL);
444 }
445 
446 
447 
448 /**
449  * Inicializa o socket do servidor, associando-o a um endere o e porta
        especificados na configura  o.
450  *
451  * @param serv_addr Um pointer para a estrutura 'sockaddr_in' que ser
        preenchida com as informa  es do endere o do servidor.
452  * @param sockfd Um pointer para o descritor de socket, que ser
        inicializado e associado ao endere o.
```

```
453  * @param config Um pointer para a estrutura ‘ServerConfig‘ que cont m
         as configura  es do servidor, incluindo a porta.
454  *
455  * @details Esta fun  o faz o seguinte:
456  * - Cria um socket do tipo ‘SOCK_STREAM‘ para comunica  es baseadas
         em TCP.
457  * - Limpa a estrutura do socket com ‘memset‘ para garantir que todos
         os campos s o inicializados corretamente.
458  * - Preenche a estrutura do socket com a fam lia de endere os ‘
         AF_INET‘,
459  *   o endere o ‘INADDR_ANY‘ (para aceitar conex es de qualquer
         endere o) e a porta especificada na configura  o do servidor.
460  * - Usa ‘bind‘ para associar o socket ao endere o e porta definidos.
         Se a opera  o falhar,
461  *   a fun  o regista um erro no log e termina o programa.
462  * - Coloca o socket em modo de escuta com a fun  o ‘listen‘,
         permitindo que o servidor aceite conex es.
463  */
464
465 void initializeSocket(struct sockaddr_in *serv_addr, int *sockfd,
       ServerConfig *config) {
466
467     // Criar socket
468     if ((*sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
469         err_dump(config, 0, 0, "can't open stream socket",
                EVENT_CONNECTION_SERVER_ERROR);
470     }
471
472     // Limpar a estrutura do socket
473     memset((char *) serv_addr, 0, sizeof(*serv_addr));
474
475     // Preencher a estrutura do socket
476     serv_addr->sin_family = AF_INET;
477     serv_addr->sin_addr.s_addr = htonl(INADDR_ANY);
478     serv_addr->sin_port = htons(config->serverPort);
479
480     // Associar o socket a um endereco qualquer
481     if (bind(*sockfd, (struct sockaddr *)serv_addr, sizeof(*serv_addr))
            < 0) {
```

```
482         err_dump(config, 0, 0, "can't bind local address",
                EVENT_CONNECTION_SERVER_ERROR);
483     }
484
485     // Ouvir o socket
486     listen(*sockfd, 1);
487 }
```

A.3.12 src/server-game.h

```
1  #ifndef SERVER_GAME_H
2  #define SERVER_GAME_H
3
4  #include "../config/config.h"
5  #include "server-barber.h"
6  #include "server-barrier.h"
7  #include "server-readerWriter.h"
8  #include "server-statistics.h"
9
10 // Gera um ID  nico  para uma sala.
11 int generateUniqueId();
12
13 // Verifica se a linha inserida pelo jogador est  correta.
14 int verifyLine(ServerConfig *config, Game *game, char *solutionSent,
       int insertLine[9], int playerID);
15
16 // Verifica se uma linha do tabuleiro est  correta.
17 bool isLineCorrect(Game *game, int row);
18
19 // Cria uma sala e um jogo, configurando-os com base nos par metros
       fornecidos.
20 Room *createRoomAndGame(ServerConfig *config, Client *client, bool
       isSinglePlayer, bool isRandom, int gameID, int synchronizationType);
21
22 // Cria uma nova sala de jogo.
23 Room *createRoom(ServerConfig *config, int playerID, bool
       isSinglePlayer, int synchronizationType);
24
25 // Obt m uma sala de jogo a partir do ID.
```

```
26 Room *getRoom(ServerConfig *config, int roomID, int playerID);
27
28 // Junta um jogador a uma sala existente.
29 void joinRoom(ServerConfig *config, Room *room, Client *client);
30
31 // delete room
32 void deleteRoom(ServerConfig *config, int roomID);
33
34 // Obt m uma lista das salas de jogo dispon veis.
35 char *getRooms(ServerConfig *config);
36
37 // Obt m uma lista de IDs dos jogos dispon veis.
38 char *getGames(ServerConfig *config);
39
40 // Carrega um jogo espec fico a partir do ficheiro 'games.json'.
41 Game *loadGame(ServerConfig *config, int gameID, int playerID);
42
43 // Carrega um jogo aleat rio do ficheiro 'games.json'.
44 Game *loadRandomGame(ServerConfig *config, int playerID);
45
46 // Envia o tabuleiro atual ao cliente em formato JSON.
47 void sendBoard(ServerConfig *config, Room* room, Client *client);
48
49 // Recebe as linhas enviadas pelo cliente e processa-as.
50 void receiveLines(ServerConfig *config, Room *room, Client *client, int
       *currentLine);
51
52 // Termina o jogo e limpa os recursos associados    sala.
53 void finishGame(ServerConfig *config, Room *room, int *socket);
54
55 // Trata o temporizador de espera do cliente.
56 void handleTimer(ServerConfig *config, Room *room, Client *client);
57
58 // Envia uma mensagem de atualiza  o do temporizador ao cliente.
59 void sendTimerUpdate(ServerConfig *config, Room *room, Client *client);
60
61 #endif
```

A.3.13 src/server-game.c

```c
#include <stdio.h>  // Usar FILE, fopen(), fclose()
#include <stdlib.h>
#include <string.h> // Usar strings (strcpy(), strcmp())
#include <time.h>   // Usar time()
#include <stdbool.h> // Usar bool
#include <pthread.h>
#include <unistd.h>
#include "../../utils/logs/logs-common.h"
#include "../../utils/parson/parson.h"
#include "../../utils/network/network.h"
#include "server-game.h"
#include "../logs/logs.h"

static int nextRoomID = 1;

/**
 * Gera um identificador  nico  para uma nova sala de jogo.
 *
 * @return Um identificador  nico  para a sala, incrementando o valor
    de 'nextRoomID'.
 *
 * @details Esta fun  o utiliza uma vari vel est tica 'nextRoomID'
    para gerar IDs  nicos
 * e sequenciais para as salas de jogo.
 * Cada vez que a fun  o    chamada, o valor de 'nextRoomID'
    retornado e, em seguida, incrementado,
 * garantindo que cada sala criada tenha um ID exclusivo.
 */

int generateUniqueId() {
    return nextRoomID++;
}

/**
 * Carrega um jogo a partir do ficheiro 'games.json' com base no ID do
    jogo.
 *
```

```
34  * @param config Um pointer para a estrutura ‘ServerConfig‘ que cont m
        o caminho
35  * para o ficheiro ’games.json’ e o ficheiro de log.
36  * @param gameID O identificador do jogo que se pretende carregar.
37  * @param playerID O identificador do jogador que est  a tentar
        carregar o jogo,
38  * usado para o registo no log.
39  * @return Um pointer para a estrutura ‘Game‘ carregada, ou NULL se o
        jogo n o for encontrado ou ocorrer um erro.
40  *
41  * @details Esta fun  o faz o seguinte:
42  * - Aloca mem ria para a estrutura ‘Game‘ e inicializa-a a zeros.
43  * - Abre o ficheiro ’games.json’ e l  o seu conte do. Se ocorrer um
        erro, regista-o no log e devolve NULL.
44  * - Faz o parse do conte do JSON e percorre o array de jogos para
        encontrar o jogo com o ID correspondente.
45  * - Se o jogo for encontrado, preenche o tabuleiro (‘board‘) e a
        solu  o (‘solution‘) na estrutura ‘Game‘.
46  * - Regista o carregamento bem-sucedido no log e devolve o pointer
        para o jogo.
47  * - Se o jogo n o for encontrado, regista o erro no log, imprime uma
        mensagem de erro no terminal, e devolve NULL.
48  * - Liberta a mem ria alocada para o conte do JSON e a estrutura ‘
        Game‘ em caso de falha.
49  */
50
51  Game *loadGame(ServerConfig *config, int gameID, int playerID) {
52
53      Game *game = (Game *)malloc(sizeof(Game));
54      if (game == NULL) {
55          fprintf(stderr, "Memory allocation failed\n");
56          return NULL;
57      }
58      memset(game, 0, sizeof(Game));  // Initialize game struct
59
60      FILE *file = fopen(config->gamePath, "r");
61
62      if (file == NULL) {
```

```
63          produceLog(config, "Erro ao abrir o ficheiro de jogos.",
                EVENT_GAME_NOT_LOAD, gameID, playerID);
64          free(game);  // Free allocated memory before exiting
65          return NULL;
66      }
67
68      // Ler o ficheiro JSON
69      fseek(file, 0, SEEK_END);
70      long file_size = ftell(file);
71      fseek(file, 0, SEEK_SET);
72
73      char *file_content = malloc(file_size + 1);
74      if (file_content == NULL) {
75          fclose(file);
76          fprintf(stderr, "Memory allocation failed\n");
77          return NULL;
78      }
79
80      fread(file_content, 1, file_size, file);
81      file_content[file_size] = '\0';
82      fclose(file);
83
84      // Parse do JSON
85      JSON_Value *root_value = json_parse_string(file_content);
86      JSON_Object *root_object = json_value_get_object(root_value);
87      free(file_content);
88
89      // Get the "games" array
90      JSON_Array *games_array = json_object_get_array(root_object, "games
            ");
91
92      // Iterate over the games array to find the matching ID
93      for (int i = 0; i < json_array_get_count(games_array); i++) {
94
95          // get json object
96          JSON_Object *game_object = json_array_get_object(games_array, i
                );
97
98          // get current id from Json object
```

```
99          int currentID = (int)json_object_get_number(game_object, "id");
100
101         // se o game for encontrado
102         if (currentID == gameID) {
103
104             // set game id as currentID
105             game->id = currentID;
106
107             // set current line as 1
108             game->currentLine = 1;
109
110              // Obter o board e a solution do JSON
111             for (int row = 0; row < 9; row++) {
112
113                 JSON_Array *board_row = json_array_get_array(
                        json_object_get_array(game_object, "board"), row);
114                 JSON_Array *solution_row = json_array_get_array(
                        json_object_get_array(game_object, "solution"), row)
                        ;
115
116                 for (int col = 0; col < 9; col++) {
117                     game->board[row][col] = (int)json_array_get_number(
                            board_row, col);
118                     game->solution[row][col] = (int)
                            json_array_get_number(solution_row, col);
119                 }
120             }
121
122             // game has been loaded successfully
123             produceLog(config, "Jogo carregado com sucesso",
                    EVENT_GAME_LOAD, gameID, playerID);
124
125             // if game has been found leave loop
126             json_value_free(root_value);
127             return game; // Return the game immediately
128         }
129     }
130
131     // Game not found
```

```
132     if (game->id == 0) {
133 
134         // cria mensagem mais detalhada
135         char logMessage[100];
136         snprintf(logMessage, sizeof(logMessage), "game com ID %d nao
                encontrado", gameID);
137         produceLog(config, logMessage, EVENT_GAME_NOT_FOUND, gameID,
                playerID);
138 
139         // mostra no terminal e encerra programa
140         fprintf(stderr, "game com ID %d nao encontrado.\n", gameID);
141         free(game);
142         return NULL;
143     }
144 
145     // Limpar memoria alocada ao JSON
146     json_value_free(root_value);
147 
148     return game;
149 }
150 
151 
152 /**
153  * Carrega um jogo aleat rio a partir do ficheiro 'games.json'.
154  *
155  * @param config Um pointer para a estrutura `ServerConfig` que cont m
        o caminho
156  * para o ficheiro 'games.json' e o caminho do log.
157  * @param playerID O identificador do jogador que est  a solicitar um
        jogo aleat rio,
158  * usado para o registo no log.
159  * @return Um pointer para a estrutura `Game` carregada, ou NULL se
        ocorrer um erro.
160  *
161  * @details Esta fun  o faz o seguinte:
162  * - Abre o ficheiro 'games.json' para leitura. Se o ficheiro n o
        puder ser aberto,
163  * regista um erro no log e termina o programa.
```

```
164  * - L   o conte do do ficheiro e garante que a string termina com um
         caractere nulo.
165  * - Faz o parse do conte do JSON e obt m o array de jogos.
166  * - Usa uma semente baseada no tempo atual para gerar um ID de jogo
         aleat rio.
167  * - Chama a fun   o `loadGame` para carregar o jogo aleat rio
         selecionado.
168  * - Liberta a mem ria alocada para o conte do JSON e retorna o jogo
         carregado.
169  */
170
171 Game *loadRandomGame(ServerConfig *config, int playerID) {
172
173     // open the file
174     FILE *file = fopen(config->gamePath, "r");
175
176     // check if file is opened
177     if (file == NULL) {
178         err_dump(config, 0, playerID, "Erro ao abrir o ficheiro de
                jogos.", EVENT_GAME_NOT_LOAD);
179     }
180
181     // get the size of the file
182     fseek(file, 0, SEEK_END);
183     long file_size = ftell(file);
184     fseek(file, 0, SEEK_SET);
185
186     // read the file content
187     char *file_content = malloc(file_size + 1);
188     fread(file_content, 1, file_size, file);
189
190     // add null terminator to the end of the file content
191     file_content[file_size] = '\0';
192
193     // close the file
194     fclose(file);
195
196     // parse the JSON
197     JSON_Value *root_value = json_parse_string(file_content);
```

```
198     JSON_Object *root_object = json_value_get_object(root_value);
199
200     // free the file content
201     free(file_content);
202
203     // get the games array
204     JSON_Array *games_array = json_object_get_array(root_object, "games
            ");
205
206     // get the number of games
207     int numberOfGames = json_array_get_count(games_array);
208
209     // get seed for random number generator
210     srand(time(NULL));
211
212     // get a random game ID
213     int randomGameID = rand() % numberOfGames + 1;
214     printf("Random game ID selected: %d from %d games\n", randomGameID,
            numberOfGames);
215
216     // free the JSON object
217     json_value_free(root_value);
218
219     // return the loaded game
220     return loadGame(config, randomGameID, playerID);
221 }
222
223
224 /**
225  * Verifica se uma linha inserida pelo jogador est  correta em
        rela  o  solu  o do jogo.
226  *
227  * @param logFileName O caminho para o ficheiro de log onde os eventos
        s o registados.
228  * @param solutionSent Uma string que representa a solu  o enviada
        pelo jogador.
229  * @param game Um pointer para a estrutura 'Game' que cont m o estado
        atual do tabuleiro e a solu  o correta.
```

```
230  * @param insertLine Um array de 9 inteiros que representa a linha
         inserida pelo jogador.
231  * @param lineNumber O n mero da linha a ser verificada (0-indexado).
232  * @param playerID O identificador do jogador que enviou a linha.
233  * @return 1 se a linha estiver correta, 0 se estiver incorreta ou
         incompleta.
234  *
235  * @details Esta fun  o faz o seguinte:
236  * - Regista no log a solu  o enviada pelo jogador, incluindo o ID do
         jogador, o ID do jogo, e o n mero da linha.
237  * - Itera sobre os 9 valores da linha inserida e compara cada valor
         com a solu  o do jogo.
238  * - Se o valor estiver correto, atualiza o tabuleiro do jogo com o
         valor inserido.
239  * - Imprime no terminal a posi  o da linha, o valor esperado, e o
         valor recebido para facilitar o debug.
240  * - Verifica se a linha est  correta usando a fun  o `isLineCorrect
         `.
241  * - Regista no log se a linha foi validada como correta ou incorreta e
         devolve 1 ou 0, respetivamente.
242  */
243
244 int verifyLine(ServerConfig *config, Game *game, char * solutionSent,
       int insertLine[9], int playerID) {
245
246     char logMessage[100];
247
248     sprintf(logMessage, "O jogador %d no jogo %d para a linha %d: %s",
           playerID, game->id, game->currentLine, solutionSent);
249     produceLog(config, logMessage, EVENT_SOLUTION_SENT, game->id,
           playerID);
250
251     for (int j = 0; j < 9; j++) {
252
253         // verifica se o valor inserido    igual ao valor da solu  o
254         if (insertLine[j] == game->solution[game->currentLine - 1][j])
               {
255
256             // se for igual, atualiza o tabuleiro
```

```
257             game->board[game->currentLine - 1][j] = insertLine[j];
258
259         // se o valor inserido for diferente do valor da solu   o
260         }
261
262         //printf("Posi   o %d: esperado %d, recebido %d\n", j + 1,
                game->solution[game->currentLine - 1][j], insertLine[j]);
263     }
264
265     // limpa logMessage
266     memset(logMessage, 0, sizeof(logMessage));
267
268     // verifica se a linha est   correta
269     if (isLineCorrect(game, game->currentLine - 1)) {
270
271         // linha correta
272         snprintf(logMessage, sizeof(logMessage), "Linha enviada (%s)
                validada como CERTA", solutionSent);
273         produceLog(config, logMessage, EVENT_SOLUTION_CORRECT, game->id
                , playerID);
274         return 1;
275
276     } else {
277
278         // linha incorreta
279         snprintf(logMessage, sizeof(logMessage), "Linha enviada (%s)
                validada como ERRADA/INCOMPLETA", solutionSent);
280         produceLog(config, logMessage, EVENT_SOLUTION_INCORRECT, game->
                id, playerID);
281         return 0;
282     }
283 }
284
285 /**
286  * Verifica se uma linha espec fica do tabuleiro do jogo est   correta
        em rela   o    solu   o.
287  *
288  * @param game Um pointer para a estrutura 'Game' que cont m o
        tabuleiro atual e a solu   o correta.
```

```
289  * @param row O n mero da linha (0-indexado) que deve ser verificada.
290  * @return `true` se todos os valores da linha estiverem corretos, `
         false` caso contr rio.
291  *
292  * @details Esta fun  o percorre os 9 valores da linha especificada
         no tabuleiro do jogo.
293  * Se algum valor da linha atual n o corresponder ao valor na
         solu  o , a fun  o devolve `false`.
294  * Se todos os valores forem iguais, a fun  o devolve `true`,
         indicando que a linha est  correta.
295  */
296
297 bool isLineCorrect(Game *game, int row) {
298     for (int i = 0; i < 9; i++) {
299         if (game->board[row][i] != game->solution[row][i]) {
300             return false;
301         }
302     }
303     return true;
304 }
305
306
307 /**
308  * Cria uma nova sala de jogo e inicializa os seus campos.
309  *
310  * @param config Um pointer para a estrutura `ServerConfig` que cont m
         as configura  es do servidor,
311  * incluindo o n mero  m ximo de salas e o caminho para o ficheiro de
         log.
312  * @return Um pointer para a nova estrutura `Room` criada, ou NULL se a
         aloca  o de mem ria falhar.
313  *
314  * @details Esta fun  o faz o seguinte:
315  * - Aloca mem ria para uma nova estrutura `Room` e inicializa os seus
         campos a zeros.
316  * - Gera um identificador  nico  para a sala usando `generateUniqueId
         `.
317  * - Aloca mem ria para o array de jogadores da sala, com o tamanho
         m ximo definido em `config`.
```

```
318  * - Inicializa o array `config->rooms` a zeros para garantir que todos
         os campos est o corretamente definidos.
319  * - Incrementa o n mero de salas ativas no servidor.
320  * - Regista a cria  o da sala no ficheiro de log.
321  * - Devolve o pointer para a sala criada, ou NULL se a aloca  o de
         mem ria falhar.
322  */
323
324 Room *createRoom(ServerConfig *config, int playerID, bool
        isSinglePlayer, int synchronizationType) {
325
326     Room *room = (Room *)malloc(sizeof(Room));
327     if (room == NULL) {
328         err_dump(config, 0, playerID, "Memory allocation failed",
                EVENT_ROOM_NOT_LOAD);
329         return NULL;
330     }
331     memset(room, 0, sizeof(Room));  // Initialize Room struct
332
333     room->id = generateUniqueId();
334     printf("Room ID na cria  o da room: %d\n", room->id);
335     room->timer = 60;
336     room->isGameRunning = false;
337     room->isFinished = false;
338     room->isSinglePlayer = isSinglePlayer;
339     room->maxClients = room->isSinglePlayer ? 1 : config->
            maxClientsPerRoom;
340     room->clients = (Client **)malloc(sizeof(Client *) * room->
            maxClients);
341     room->maxWaitingTime = config->maxWaitingTime;
342     room->savedStatistics = false;
343
344     // we dont need synchronization for single player games
345     if (!room->isSinglePlayer) {
346
347         // Inicializar priority queue
348         room->enterRoomQueue = (PriorityQueue *)malloc(sizeof(
                PriorityQueue));
349         initPriorityQueue(room->enterRoomQueue, room->maxClients * 2);
```

```
350
351         // Initialize reader-writer locks
352         sem_init(&room->writeSemaphore, 0, 1); // Inicializar sem foro
                para escrita e come a a aceitar 1 escritor
353         sem_init(&room->readSemaphore, 0, 1); // Inicializar sem foro
               para leitura e come a a aceitar 1 leitor
354         sem_init(&room->nonPremiumWriteSemaphore, 0, 0); //
355         pthread_mutex_init(&room->readMutex, NULL); // Inicializar
               mutex para leitura
356         pthread_mutex_init(&room->writeMutex, NULL); // Inicializar
               mutex para escrita
357         room->readerCount = 0;
358         room->writerCount = 0;
359
360         // Inicializar barreira para come ar e terminar o jogo
361         room->waitingCount = 0;
362         sem_init(&room->mutexSemaphore, 0, 1);
363         sem_init(&room->turnsTileSemaphore1, 0, 0);
364         sem_init(&room->turnsTileSemaphore2, 0, 0);
365
366         // barber shop initialization
367         room->customers = 0;
368         pthread_mutex_init(&room->barberShopMutex, NULL);
369         sem_init(&room->costumerSemaphore, 0, 0);
370         sem_init(&room->costumerDoneSemaphore, 0, 0);
371         sem_init(&room->barberDoneSemaphore, 0, 0);
372         room->barberShopQueue = (PriorityQueue *)malloc(sizeof(
               PriorityQueue));
373         initPriorityQueue(room->barberShopQueue, room->maxClients);
374
375         // check if the game is reader-writer or barber shop
376         if (synchronizationType == 0) {
377             room->isReaderWriter = true;
378         }
379
380         if (synchronizationType == 1) { // barber shop with static
               priority
381             room->isReaderWriter = false;
382             room->priorityQueueType = 0;
```

```
383         }
384
385         if (synchronizationType == 2) { // barber shop with dynamic
                priority
386             room->isReaderWriter = false;
387             room->priorityQueueType = 1;
388         }
389
390         if (synchronizationType == 3) { // barber shop with FIFO
391             room->isReaderWriter = false;
392             room->priorityQueueType = 2;
393         }
394
395         if (!room->isReaderWriter) {
396             // initialize thread for barber
397             if(pthread_create(&room->barberThread, NULL, handleBarber,
                    (void *)room)) {
398                 err_dump(config, 0, playerID, "Erro ao criar a thread
                        do barbeiro", EVENT_THREAD_NOT_CREATE);
399             }
400             produceLog(config, "Barbeiro criado com sucesso",
                    EVENT_BARBER_CREATED, room->id, playerID);
401         }
402
403         // initialize mutexes
404         pthread_mutex_init(&room->mutex, NULL);
405         pthread_mutex_init(&room->timerMutex, NULL);
406     }
407
408     // add the room to the list of rooms and increment the number of
            rooms
409     config->rooms[config->numRooms++] = room;
410
411     // log room creation
412     produceLog(config, "Sala criada com sucesso", EVENT_ROOM_LOAD, room
            ->id, playerID);
413     return room;
414 }
415
```

```
416 Room *getRoom(ServerConfig *config, int roomID, int playerID) {
417
418     // check if roomID is valid
419     if (roomID < 1) {
420         err_dump(config, 0, playerID, "Room ID is invalid",
                EVENT_ROOM_NOT_JOIN);
421         return NULL;
422     }
423
424     // get the room by looping through the rooms
425     Room *room = NULL;
426     for (int i = 0; i < config->numRooms; i++) {
427         //printf("i: %d\n", i);
428         //printf("NR rooms: %d\n", config->numRooms);
429         //printf("ROOM ID: %d\n", config->rooms[i]->id);
430         if (config->rooms[i]->id == roomID) {
431             room = config->rooms[i];
432             break;
433         }
434     }
435
436     return room;
437 }
438
439 void deleteRoom(ServerConfig *config, int roomID) {
440     for (int i = 0; i < config->numRooms; i++) {
441         if (config->rooms[i]->id == roomID) {
442             printf("FREEING MEMORY FOR ROOM %d\n", roomID);
443
444             free(config->rooms[i]->clients);
445
446             free(config->rooms[i]->game);
447
448             // Destruir mutexes e sem foros
449             if (!config->rooms[i]->isSinglePlayer) {
450                 pthread_mutex_destroy(&config->rooms[i]->mutex);
451                 pthread_mutex_destroy(&config->rooms[i]->timerMutex);
452                 pthread_mutex_destroy(&config->rooms[i]->readMutex);
453                 pthread_mutex_destroy(&config->rooms[i]->writeMutex);
```

```
454                 pthread_mutex_destroy(&config->rooms[i]->
                        barberShopMutex);
455                 sem_destroy(&config->rooms[i]->mutexSemaphore);
456                 sem_destroy(&config->rooms[i]->turnsTileSemaphore1);
457                 sem_destroy(&config->rooms[i]->turnsTileSemaphore2);
458                 sem_destroy(&config->rooms[i]->writeSemaphore);
459                 sem_destroy(&config->rooms[i]->readSemaphore);
460                 sem_destroy(&config->rooms[i]->nonPremiumWriteSemaphore
                        );
461                 sem_destroy(&config->rooms[i]->costumerSemaphore);
462                 sem_destroy(&config->rooms[i]->costumerDoneSemaphore);
463                 sem_destroy(&config->rooms[i]->barberDoneSemaphore);
464
465                 // destroy barber
466                 if (!config->rooms[i]->isReaderWriter) {
467                     pthread_cancel(config->rooms[i]->barberThread);
468                 }
469
470                 // free priority queue
471                 freePriorityQueue(config->rooms[i]->enterRoomQueue);
472                 freePriorityQueue(config->rooms[i]->barberShopQueue);
473             }
474
475             free(config->rooms[i]);
476
477             // shift all rooms to the left
478             for (int j = i; j < config->numRooms - 1; j++) {
479                 config->rooms[j] = config->rooms[j + 1];
480             }
481
482             // decrement number of rooms
483             config->numRooms--;
484
485             // log room deletion
486             produceLog(config, "Sala eliminada com sucesso",
                        EVENT_ROOM_DELETE, roomID, 0);
487
488             break;
489         }
```

```
490     }
491
492
493 }
494
495 /**
496  * Obt m uma lista de identificadores de jogos a partir do ficheiro '
        games.json'.
497  *
498  * @param config Um pointer para a estrutura `ServerConfig` que cont m
499  * o caminho para o ficheiro 'games.json' e o caminho do log.
500  * @return Uma string que cont m os IDs de todos os jogos, formatados
        com
501  * "Game ID: [ID]", ou NULL se ocorrer um erro na leitura do ficheiro.
502  *
503  * @details Esta fun   o faz o seguinte:
504  * - Abre o ficheiro 'games.json' e l   todo o seu conte do. Se o
        ficheiro n o puder ser aberto,
505  * regista o erro no log e termina o programa.
506  * - Faz o parse do conte do JSON para obter o array de jogos.
507  * - Itera sobre cada jogo no array e extrai o ID do jogo.
508  * - Concatena todos os IDs numa string, cada um formatado como "Game
        ID: [ID]".
509  * - Liberta a mem ria alocada para o conte do JSON e devolve a
        string que cont m os IDs.
510  *
511  * @note O tamanho da string `games`    limitado a 1024 caracteres, o
        que pode ser ajustado
512  * conforme necess rio para evitar overflow.
513  */
514
515 char *getGames(ServerConfig *config) {
516
517     // open the file
518     FILE *file = fopen(config->gamePath, "r");
519
520     // check if file is opened
521     if (file == NULL) {
```

```
522         err_dump(config, 0, 0, "Erro ao abrir o ficheiro de jogos.",
                EVENT_GAME_NOT_LOAD);
523     }
524
525     // get the size of the file
526     fseek(file, 0, SEEK_END);
527     long file_size = ftell(file);
528     fseek(file, 0, SEEK_SET);
529
530     // read the file content
531     char *file_content = malloc(file_size + 1);
532     fread(file_content, 1, file_size, file);
533
534     // add null terminator to the end of the file content
535     file_content[file_size] = '\0';
536
537     // close the file
538     fclose(file);
539
540     // get ids of the games
541     JSON_Value *root_value = json_parse_string(file_content);
542     JSON_Object *root_object = json_value_get_object(root_value);
543
544     // free the file content
545     free(file_content);
546
547     // get the games array
548     JSON_Array *games_array = json_object_get_array(root_object, "games
            ");
549
550     // get the number of games
551     int numberOfGames = json_array_get_count(games_array);
552
553     // create a string to store the ids of the games
554     char *games = (char *)malloc(1024);
555     memset(games, 0, 1024);
556
557     // iterate over the games array
558     for (int i = 0; i < numberOfGames; i++) {
```

```
559         JSON_Object *game_object = json_array_get_object(games_array, i
                );
560         int gameID = (int)json_object_get_number(game_object, "id");
561         char gameIDString[10];
562         sprintf(gameIDString, "%d", gameID);
563         strcat(games, "Game ID: ");
564         strcat(games, gameIDString);
565         strcat(games, "\n");
566     }
567 
568     // free the JSON object
569     json_value_free(root_value);
570 
571     // return the games
572     return games;
573 }
574 
575 /**
576  * Obt m uma lista de salas de jogo ativas no servidor, incluindo o ID
        da sala, o n mero de jogadores,
577  * o n mero  m ximo de jogadores, e o ID do jogo associado.
578  *
579  * @param config Um pointer para a estrutura `ServerConfig` que cont m
        as informa  es sobre as salas de jogo.
580  * @return Uma string que cont m a lista de salas de jogo formatada,
581  * ou uma mensagem "No rooms available\n" se n o houver salas ativas.
582  *
583  * @details Esta fun  o faz o seguinte:
584  * - Verifica se h  salas dispon veis. Se o n mero de salas for zero
        , devolve a mensagem "No rooms available\n".
585  * - Cria uma string para armazenar as informa  es de cada sala,
        incluindo o ID da sala, o n mero de jogadores atuais,
586  *   o n mero  m ximo de jogadores, e o ID do jogo.
587  * - Itera sobre todas as salas e concatena as informa  es formatadas
        de cada sala na string `rooms`.
588  * - Devolve a string `rooms` contendo as informa  es de todas as
        salas. A mem ria alocada para esta
589  * string deve ser libertada pelo chamador.
590  *
```

```
591  * @note O tamanho da string `rooms`    limitado a 1024 caracteres, o
         que pode ser ajustado
592  * se houver um n mero elevado de salas.
593  */
594
595 char *getRooms(ServerConfig *config) {
596
597         // create a string to store the rooms
598         char *rooms = (char *)malloc(BUFFER_SIZE);
599         memset(rooms, 0, 1024);
600
601         // iterate over the rooms
602         if (config->numRooms == 0) {
603             strcpy(rooms, "No rooms available\n");
604         }
605
606         for (int i = 0; i < config->numRooms; i++) {
607             if (config->rooms[i] == NULL) {
608                 continue;
609             }
610             // can only join rooms that are not running
611             if (config->rooms[i]->isGameRunning == false) {
612                 // show number of players in the room, max players in
                        the room and the game ID
613                 char roomString[100];
614                 sprintf(roomString, "Room ID: %d, Players: %d/%d, Game
                        ID: %d\n", config->rooms[i]->id, config->rooms[i]->
                        numClients, config->rooms[i]->maxClients, config->
                        rooms[i]->game->id);
615                 strcat(rooms, roomString);
616             }
617         }
618
619         // if no rooms are available
620         if (strlen(rooms) == 0) {
621             strcpy(rooms, "No rooms available\n");
622         }
623
624         // return the rooms
```

```
625         return rooms;
626 }
627
628 /**
629  * Cria uma nova sala de jogo e carrega um jogo associado, atribuindo-o
        a um jogador.
630  *
631  * @param newSockfd Um pointer para o descritor de socket do cliente,
       usado para enviar mensagens.
632  * @param config Um pointer para a estrutura ‘ServerConfig‘ que cont m
        as configura  es do servidor.
633  * @param ClientID O identificador do jogador que ir  participar na
       sala.
634  * @param isSinglePlayer Um valor booleano que indica se o jogo
       single Client (true) ou MultiPlayer (false).
635  * @param isRandom Um valor booleano que indica se o jogo deve ser
       carregado aleatoriamente (true),
636  * ou se um jogo espec fico deve ser carregado (false).
637  * @param gameID O identificador do jogo a ser carregado, usado se ‘
       isRandom‘ for false.
638  * @return Um pointer para a estrutura ‘Room‘ criada, ou NULL se n o
       houver salas dispon veis.
639  *
640  * @details Esta fun  o faz o seguinte:
641  * - Verifica se o n mero m ximo de salas foi atingido.
642  *   Se n o houver salas dispon veis, envia uma mensagem ao cliente e
        regista o evento no ficheiro de log.
643  * - Cria uma nova sala e adiciona-a    lista de salas do servidor.
644  * - Carrega um jogo, seja de forma aleat ria ou usando o ‘gameID‘
       especificado.
645  * - Se o jogo for carregado com sucesso, associa-o    sala criada.
646  * - Define o n mero m ximo de jogadores para 1 se for um jogo single
        Client,
647  *   ou para o valor definido na configura  o se for MultiPlayer.
648  * - Adiciona o jogador    sala e inicializa o n mero de jogadores da
       sala.
649  * - Imprime uma mensagem de confirma  o e retorna a sala criada.
650  */
651
```

```
652 Room *createRoomAndGame(ServerConfig *config, Client *client, bool
        isSinglePlayer, bool isRandom, int gameID, int synchronizationType)
    {
653
654     // check if there's config->rooms is full by looping
655     if (config->numRooms >= config->maxRooms) {
656         // send message to client
657         send(client->socket_fd, "No rooms available", strlen("No rooms
                available"), 0);
658         // write log
659         produceLog(config, "No rooms available", EVENT_ROOM_NOT_CREATED
                , 0, client->clientID);
660         return NULL;
661     }
662
663     // create room
664     Room *room = createRoom(config, client->clientID, isSinglePlayer,
            synchronizationType);
665
666     // load game
667     Game *game;
668
669     if (isRandom) {
670         game = loadRandomGame(config, client->clientID);
671     } else {
672         game = loadGame(config, gameID, client->clientID);
673     }
674
675     if (game == NULL) {
676         fprintf(stderr, "Error loading random game\n");
677         exit(1);
678     }
679
680     // add game to room
681     room->game = game;
682
683     joinRoom(config, room, client);
684
685     char buffer[BUFFER_SIZE];
```

```
686     memset(buffer, 0, sizeof(buffer));
687
688     strcpy(buffer, "");
689
690     if (!room->isSinglePlayer && !room->isReaderWriter) {
691         if (room->priorityQueueType == 0) {
692             strcpy(buffer, " and STATIC PRIORITIES queue");
693         } else if (room->priorityQueueType == 1) {
694             strcpy(buffer, " and DYNAMIC PRIORITIES queue");
695         } else if (room->priorityQueueType == 2) {
696             strcpy(buffer, " and FIFO queue");
697         }
698     }
699
700     // Mensagem de cria  o da sala
701     printf("New game created by client %d with game %d and room is
            synchronized by %s%s. Client %d is %s.\n",
702             client->clientID, room->game->id,
703             !room->isSinglePlayer ? (room->isReaderWriter ? "READER-
                WRITER" : "BARBER SHOP") : "",
704             buffer,
705             client->clientID, client->isPremium ? "Premium" : "Non-
                premium");
706
707     if (!room->isSinglePlayer) {
708         printf("Waiting for more Clients to join in room %d\n", room->
               id);
709     }
710
711
712     return room;
713 }
714
715 /**
716  * Adiciona um jogador a uma sala de jogo existente.
717  *
718  * @param config Um pointer para a estrutura `ServerConfig` que cont m
        as informa  es
719  * de configura  o do servidor, incluindo a lista de salas.
```

```
720  * @param roomID O identificador da sala que o jogador pretende entrar.
721  * @param ClientID O identificador do jogador que deseja juntar-se
         sala.
722  * @return Um pointer para a sala `Room` se o jogador for adicionado
         com sucesso,
723  * ou NULL se a sala estiver cheia ou se o `roomID` for inv lido.
724  *
725  * @details Esta fun   o faz o seguinte:
726  * - Verifica se o `roomID` fornecido    v lido (dentro dos limites
         das salas existentes).
727  * - Obt m a sala correspondente a partir da lista de salas no
         servidor.
728  * - Verifica se a sala j   est  cheia. Se estiver, devolve NULL e
         imprime uma mensagem de erro no terminal.
729  * - Se houver espa o na sala, adiciona o jogador ao array de
         jogadores da sala e incrementa o n mero de jogadores.
730  * - Regista no ficheiro de log que o jogador entrou na sala.
731  * - Devolve um pointer para a sala `Room` atualizada.
732  */
733 void joinRoom(ServerConfig *config, Room *room, Client *client) {
734
735     // check if room is full
736     if (room->numClients == room->maxClients) {
737         err_dump(config, 0, client->clientID, "Room is full",
                EVENT_ROOM_NOT_JOIN);
738         return;
739     }
740
741     // check if room is running
742     if (room->isGameRunning) {
743         err_dump(config, 0, client->clientID, "Room is running",
                EVENT_ROOM_NOT_JOIN);
744         return;
745     }
746
747     //printf("Client %d joined room %d\n", client->clientID, room->id);
748
749     // add client to room
750     room->clients[room->numClients] = client;
```

```
751
752     // increment number of clients
753     room->numClients++;
754
755     // Log de entrada do jogador na sala
756     char logMessage[256];
757     snprintf(logMessage, sizeof(logMessage), "Client %d joined room %d"
            , client->clientID, room->id);
758     produceLog(config, logMessage, EVENT_ROOM_JOIN, room->game->id,
            client->clientID);
759
760     printf("Client %d (Premium: %s) joined room %d with socket %d\n",
761             client->clientID, client->isPremium ? "Yes" : "No", room->id
                , client->socket_fd);
762 }
763
764 /**
765  * Envia o tabuleiro do jogo ao cliente em formato JSON.
766  *
767  * @param socket Um pointer para o descritor de socket utilizado para
        enviar o tabuleiro ao cliente.
768  * @param game Um pointer para a estrutura 'Game' que cont m o
        tabuleiro a ser enviado.
769  * @param config Um pointer para a estrutura 'ServerConfig' que cont m
         a configura  o do servidor,
770  * incluindo o caminho do ficheiro de log.
771  *
772  * @details Esta fun  o faz o seguinte:
773  * - Cria uma estrutura JSON que representa o tabuleiro do jogo,
        incluindo o ID do jogo.
774  * - Converte o tabuleiro 9x9 num array de arrays em formato JSON.
775  * - Serializa o objeto JSON para uma string e envia-a ao cliente
        atrav s do socket.
776  * - Em caso de erro ao enviar o tabuleiro, regista a mensagem de erro
        no log e termina a execu  o da fun  o.
777  * - Liberta a mem ria alocada para a string serializada e o objeto
        JSON.
778  */
779
```

```
780 void sendBoard(ServerConfig *config, Room* room, Client *client) {
781
782     // Enviar board ao cliente em formato JSON
783     JSON_Value *root_value = json_value_init_object();
784     JSON_Object *root_object = json_value_get_object(root_value);
785     json_object_set_number(root_object, "id", room->game->id);
786     JSON_Value *board_value = json_value_init_array();
787     JSON_Array *board_array = json_value_get_array(board_value);
788
789     for (int i = 0; i < 9; i++) {
790         JSON_Value *linha_value = json_value_init_array();
791         JSON_Array *linha_array = json_value_get_array(linha_value);
792         for (int j = 0; j < 9; j++) {
793             json_array_append_number(linha_array, room->game->board[i][
                    j]);
794         }
795         json_array_append_value(board_array, linha_value);
796     }
797
798     json_object_set_value(root_object, "board", board_value);
799     char *serialized_string = json_serialize_to_string(root_value);
800
801     //adicionar a linha atual como um inteiro    string
802     char buffer[10];
803     sprintf(buffer, "\n%d", room->game->currentLine);
804     char *temp = malloc(strlen(serialized_string) + strlen(buffer) + 1)
        ;
805     strcpy(temp, serialized_string);
806     strcat(temp, buffer);
807
808     //printf("Enviando tabuleiro ao cliente %d do jogo %d\n", client->
        clientID, room->game->id);
809     //printf("Enviando board e linha atual: %s\n", temp);
810     // Enviar tabuleiro e linha atual ao cliente
811     if (send(client->socket_fd, temp, strlen(temp), 0) < 0) {
812         err_dump(config, room->game->id, 0, "can't send board and line
            to client", EVENT_MESSAGE_SERVER_NOT_SENT);
813         return;
814     }
```

```
815
816     // escrever no log
817     produceLog(config, "Tabuleiro enviado ao cliente",
            EVENT_MESSAGE_SERVER_SENT, room->game->id, client->clientID);
818
819     free(temp);
820     json_free_serialized_string(serialized_string);
821     json_value_free(root_value);
822 }
823
824
825 /**
826  * Recebe linhas do cliente, valida-as, e atualiza o tabuleiro do jogo.
827  *
828  * @param newSockfd Um pointer para o descritor de socket usado para
        comunica  o com o cliente.
829  * @param game Um pointer para a estrutura `Game` que cont m o estado
        atual do tabuleiro.
830  * @param ClientID O identificador do jogador que est  a enviar as
        linhas.
831  * @param config Um pointer para a estrutura `ServerConfig` que cont m
         a configura  o do servidor,
832  * incluindo o caminho do ficheiro de log.
833  *
834  * @details Esta fun  o faz o seguinte:
835  * - Itera sobre as 9 linhas do tabuleiro, recebendo e validando cada
        linha enviada pelo cliente.
836  * - Recebe cada linha do cliente como uma string de 9 caracteres e
        converte-a em valores inteiros.
837  * - Usa a fun  o `verifyLine` para validar a linha recebida. Se a
        linha estiver correta,
838  *   continua para a pr xima; caso contr rio, solicita ao cliente que
         envie novamente.
839  * - Se ocorrer um erro ao receber uma linha, regista o erro no
        ficheiro de log e termina a execu  o da fun  o.
840  * - Ap s cada valida  o, envia o tabuleiro atualizado ao cliente.
841  * - Adiciona um atraso de 1 segundo (`sleep(1)`) antes de enviar o
        tabuleiro para garantir que o cliente tem
842  *   tempo para processar as atualiza  es.
```

```
843  */
844
845 void receiveLines(ServerConfig *config, Room *room, Client *client, int
        *currentLine) {
846
847     // pre condition reader
848     if (!room->isSinglePlayer) {
849         if (room->isReaderWriter) {
850             acquireReadLock(room);
851         } else {
852             enterBarberShop(room, client);
853         }
854     }
855
856     //printf("-------------------------------------------\n");
857     //printf("IN CIO DA SEC   O CR TICA DE LEITURA para o cliente %d
            na sala %d\n", client->clientID, room->id);
858
859     int correctLine = 0;
860
861     // critical section reader
862     sendBoard(config, room, client);
863
864     //printf("FIM DA SEC   O CR TICA DE LEITURA para o cliente %d na
           sala %d\n", client->clientID, room->id);
865     //printf("-------------------------------------------\n");
866
867     // post condition reader
868     if (!room->isSinglePlayer) {
869         if (room->isReaderWriter) {
870             releaseReadLock(room);
871         } else {
872             leaveBarberShop(room, client);
873         }
874     }
875
876     // Receber e validar as linhas do cliente
877     while (room->game->currentLine <= 9) {
878
```

```
879         //printf("Recebendo linha %d do cliente %d\n", *currentLine,
                ClientID);
880
881         char line[10];
882
883         // Limpar linha
884         memset(line, '0', sizeof(line));
885
886         // Receber linha do cliente
887         if (recv(client->socket_fd, line, sizeof(line), 0) < 0) {
888             err_dump(config, room->game->id, client->clientID, "can't
                    receive line from client",
                    EVENT_MESSAGE_SERVER_NOT_RECEIVED);
889             return;
890         } else {
891
892             printf("Cliente %d %s quer resolver a linha %d na sala %d
                    com o jogo %d\n",
893             client->clientID, client->isPremium ? "(PREMIUM)" : "(NOT
                    PREMIUM)",
894             room->id, room->game->id, room->game->currentLine);
895
896             // pre condition writer
897             if (!room->isSinglePlayer) {
898                 if (room->isReaderWriter) {
899                     acquireWriteLock(room, client);
900                 } else {
901                     enterBarberShop(room, client);
902                 }
903             }
904
905             //printf
                    ("-----------------------------------------------------\
                    n");
906             //printf("IN CIO DA SEC  O CR TICA DE ESCRITA para o
                    cliente %d na sala %d\n", client->clientID, room->id);
907
908             // **Adicionei print para verificar a linha recebida**
```

```
909             //printf("Linha recebida do cliente %d: %s\n", client->
                    clientID, line);
910
911             // Converte a linha recebida em valores inteiros
912             int insertLine[9];
913             for (int j = 0; j < 9; j++) {
914                 insertLine[j] = line[j] - '0';
915             }
916
917             printf("Verificando linha %d do cliente %d na sala %d com o
                    o jogo %d\n",
918             room->game->currentLine, client->clientID, room->id, room->
                    game->id);
919             // critical section writer
920             // Verificar a linha recebida com a fun   o verifyLine
921             correctLine = verifyLine(config, room->game, line,
                    insertLine, client->clientID);
922
923             if (correctLine == 1) {
924                 // linha correta
925                 //printf("Linha %d correta enviada pelo cliente %d\n",
                        room->game->currentLine, client->clientID);
926                 (room->game->currentLine)++;
927
928             } else {
929                 // linha incorreta
930                 //printf("Linha %d incorreta enviada pelo cliente %d\n
                        ", room->game->currentLine, client->clientID);
931             }
932
933             //printf("FIM DA SEC   O CR TICA DE ESCRITA para o
                    cliente %d na sala %d\n", client->clientID, room->id);
934             //printf
                    ("-----------------------------------------------------\
                    n");
935
936             // post condition writer
937             if (!room->isSinglePlayer) {
938                 if (room->isReaderWriter) {
```

```
939                releaseWriteLock(room, client);
940            } else {
941                leaveBarberShop(room, client);
942            }
943        }
944
945        // add a random delay to appear more natural
946        //int delay = rand() % 3;
947        //sleep(delay);
948
949        // pre condition reader
950        if (!room->isSinglePlayer) {
951            if (room->isReaderWriter) {
952                acquireReadLock(room);
953            } else {
954                enterBarberShop(room, client);
955            }
956        }
957
958        sendBoard(config, room, client);
959        printf("
               ----------------------------------------------------\n"
           );
960        // post condition reader
961        if (!room->isSinglePlayer) {
962            if (room->isReaderWriter) {
963                releaseReadLock(room);
964            } else {
965                leaveBarberShop(room, client);
966            }
967        }
968    }
969  }
970 }
971
972
973 /**
974  * Termina o jogo e limpa os recursos associados    sala de jogo.
975  *
```

```
976  * @param socket Um pointer para o descritor de socket utilizado para
         comunica  o com o cliente
977  * (n o usado diretamente na fun  o).
978  * @param room Um pointer para a estrutura 'Room' que representa a sala
         de jogo que deve ser terminada.
979  * @param ClientID O identificador do jogador (n o usado diretamente
         na fun  o).
980  * @param config Um pointer para a estrutura 'ServerConfig' que cont m
         a configura  o do servidor,
981  * incluindo o n mero atual de salas.
982  *
983  * @details Esta fun  o faz o seguinte:
984  * - Remove todos os jogadores da sala, definindo os IDs dos jogadores
         para 0.
985  * - Restaura o n mero de jogadores da sala e o n mero m ximo de
         jogadores a 0.
986  * - Liberta a mem ria alocada para o jogo e a sala, prevenindo fugas
         de mem ria.
987  * - Decrementa o contador do n mero de salas no servidor na
         configura  o.
988  */
989
990 void finishGame(ServerConfig *config, Room *room, int *socket) {
991
992     // lock mutex
993     pthread_mutex_lock(&room->mutex);
994
995     if (room == NULL) {
996         return;
997     }
998
999     if (!room->isFinished) {
1000        time_t endTime = time(NULL);
1001        room->elapsedTime = difftime(endTime, room->startTime);
1002
1003        // check if is single Client
1004        if (!room->isSinglePlayer) {
1005            // Substract 60 seconds from elapsed time
1006            room->elapsedTime = room->elapsedTime;
```

```
1007            }
1008
1009            printf("Jogo na sala %d terminou. Tempo total: %.2f segundos\n"
                    , room->id, room->elapsedTime);
1010
1011            // for every Client in the room
1012            for (int i = 0; i < room->numClients; i++) {
1013
1014                // get accuracy from client
1015                char accuracy[10];
1016                memset(accuracy, 0, sizeof(accuracy));
1017                if (recv(room->clients[i]->socket_fd, accuracy, sizeof(
                        accuracy), 0) < 0) {
1018                    // erro ao receber accuracy
1019                    err_dump(config, room->game->id, room->clients[i]->
                            clientID, "can't receive accuracy from client",
                            EVENT_MESSAGE_SERVER_NOT_RECEIVED);
1020                }
1021
1022                printf("A accuracy recebida foi de: %s\n", accuracy);
1023
1024                // convert accuracy to float
1025                float accuracyFloat = atof(accuracy);
1026
1027                char timeMessage[256];
1028                snprintf(timeMessage, sizeof(timeMessage), "A accuracy
                        recebida foi de: %.2f %%\n", accuracyFloat);
1029                produceLog(config, timeMessage,
                        EVENT_MESSAGE_SERVER_RECEIVED, room->game->id, room->
                        clients[i]->clientID);
1030
1031                // Envia o tempo decorrido ao cliente
1032
1033                snprintf(timeMessage, sizeof(timeMessage), "O jogo terminou
                        ! Tempo total: %.2f segundos\n", room->elapsedTime);
1034                if (send(room->clients[i]->socket_fd, timeMessage, strlen(
                        timeMessage), 0) < 0) {
1035                    // erro ao enviar mensagem
```

```
1036                    err_dump(config, room->game->id, room->clients[i]->
                            clientID, "can't send time message to client",
                            EVENT_MESSAGE_SERVER_NOT_SENT);
1037                }
1038
1039                // escrever no log timeMessage with
                        EVENT_MESSAGE_SERVER_SENT
1040                snprintf(timeMessage, sizeof(timeMessage), "Time elapsed:
                        %.2f seconds", room->elapsedTime);
1041                produceLog(config, timeMessage, EVENT_MESSAGE_SERVER_SENT,
                        room->game->id, room->clients[i]->clientID);
1042
1043                // save on games.json the record
1044                updateGameStatistics(config, room->game->id, room->
                        elapsedTime, accuracyFloat);
1045
1046                // Save room statistics in log
1047                if (!room->savedStatistics) {
1048                    saveRoomStatistics(room->id, room->elapsedTime);
1049                    room->savedStatistics = true;
1050                }
1051            }
1052
1053            // set room as finished
1054            room->isFinished = true;
1055
1056            // remove room
1057            deleteRoom(config, room->id);
1058        }
1059
1060        // unlock mutex
1061        pthread_mutex_unlock(&room->mutex);
1062
1063    }
1064
1065    void handleTimer(ServerConfig *config, Room *room, Client *client) {
1066        fd_set readfds;
1067        struct timeval tv;
1068
```

```
1069        // lock mutex
1070        pthread_mutex_lock(&room->timerMutex);
1071
1072        if (room->timer == 0) {
1073            pthread_mutex_unlock(&room->timerMutex);
1074            return;
1075        }
1076
1077        while (room->timer > 0) {
1078            FD_ZERO(&readfds);
1079            FD_SET(client->socket_fd, &readfds);
1080            tv.tv_sec = 1;
1081            tv.tv_usec = 0;
1082
1083            int result = select(FD_SETSIZE, &readfds, NULL, NULL, &tv);
1084            //printf("Timer: %d\n", room->timer);
1085            //printf("Result: %d\n", result);
1086            //printf("Current thread: %ld\n", pthread_self());
1087
1088            if (result > 0) {
1089                // Recebe mensagem do cliente (se necess rio)
1090            } else if (result == 0) {
1091                if (room->timer == 0) {
1092                    break;
1093                }
1094
1095                // Verificar se todos os jogadores se juntaram
1096                if (room->numClients == room->maxClients) {
1097                    pthread_mutex_lock(&room->mutex);
1098                    room->timer = 0;
1099                    printf("All Clients have joined the room %d\n", room->
                        id);
1100                    printf("Starting game in room %d\n", room->id);
1101
1102                    // Enviar atualiza  o do timer para todos os
                        jogadores
1103                    for (int i = 0; i < room->numClients; i++) {
1104
```

```
1105                    //printf("Sending timer update to Client %d with
                            the socket: %d\n", room->Clients[i], room->
                            clientSockets[i]);
1106                    sendTimerUpdate(config, room, room->clients[i]);
1107                    //printf("Sent timer update to Client %d on socket
                            %d\n", room->clients[i]->clientID, room->clients
                            [i]->socket_fd);
1108
1109                }
1110                pthread_mutex_unlock(&room->mutex);
1111                break;
1112            }
1113
1114            // Enviar atualiza  o do timer a cada 10 segundos ou
                    quando o timer for inferior a 5 segundos
1115            if (room->timer % 10 == 0 || room->timer <= 5) {
1116                //pthread_mutex_lock(&room->mutex);
1117                for (int i = 0; i < room->numClients; i++) {
1118                    //printf("Sending timer update to Client %d with
                            the socket: %d\n", room->Clients[i], room->
                            clientSockets[i]);
1119                    sendTimerUpdate(config, room, room->clients[i]);
1120                }
1121                //pthread_mutex_lock(&room->mutex);
1122            }
1123
1124            // Decrementa o timer
1125            room->timer--;
1126
1127        } else {
1128            perror("select");
1129        }
1130    }
1131
1132    // Iniciar jogo
1133    printf("Jogo na sala %d iniciado  s  %s\n", room->id, ctime(&room->
            startTime));
1134
1135    pthread_mutex_unlock(&room->timerMutex);
```

```
1136 }
1137
1138 // Fun   o que envia atualiza  es do timer para o cliente,
     considerando o status premium
1139 void sendTimerUpdate(ServerConfig *config, Room *room, Client *client)
     {
1140
1141     // Preparar a mensagem de atualiza  o do timer
1142     char buffer[BUFFER_SIZE];
1143     memset(buffer, 0, sizeof(buffer));
1144
1145     sprintf(buffer, "TIMERUPDATE\n%d\n%d\n%d\n%d\n",
1146             room->timer, room->id, room->game->id, room->numClients);
1147
1148     // Enviar a mensagem de atualiza  o
1149     if (send(client->socket_fd, buffer, strlen(buffer), 0) < 0) {
1150         // erro ao enviar mensagem
1151         err_dump(config, room->game->id, client->clientID, "can't send
                update to client", EVENT_MESSAGE_SERVER_NOT_SENT);
1152     } else {
1153         // Escrever no log a atualiza  o enviada, considerando o
                status premium
1154         char logMessage[256];
1155         snprintf(logMessage, sizeof(logMessage), "Sent update to Client
                %d %s - Time left: %d seconds - Room ID: %d - Game ID: %d -
                Clients joined: %d",
1156                 client->clientID, client->isPremium ? "(Premium User)"
                        : "(Non Premium User)", room->timer, room->id, room
                        ->game->id, room->numClients);
1157
1158         produceLog(config, logMessage, EVENT_MESSAGE_SERVER_SENT, room
                ->game->id, client->clientID);
1159         printf("%s\n", logMessage);
1160     }
1161 }
```

A.3.14 src/server-readerWriter.h

```h
#ifndef SERVER_READERWRITER_H
#define SERVER_READERWRITER_H

#include "../config/config.h"

void acquireReadLock(Room *room);

void releaseReadLock(Room *room);

void acquireWriteLock(Room *room, Client *client);

void releaseWriteLock(Room *room, Client *client);

#endif // SERVER_READERWRITER_H
```

A.3.15 src/server-readerWriter.c

```c
#include <stdio.h>
#include "server-readerWriter.h"

void acquireReadLock(Room *room) {
    // indicate that a reader is entering the room
    sem_wait(&room->readSemaphore);

    // lock mutex
    pthread_mutex_lock(&room->readMutex);

    // increment reader count
    room->readerCount++;

    // if first reader lock the write semaphore
    if (room->readerCount == 1) {
        sem_wait(&room->writeSemaphore); //(decrementa o sem foro)
    }

    // unlock mutex
    pthread_mutex_unlock(&room->readMutex);

    // unlock the reader semaphore
```

```
23      sem_post(&room->readSemaphore);
24  }
25
26  void releaseReadLock(Room *room) {
27      // lock mutex
28      pthread_mutex_lock(&room->readMutex);
29
30      // decrement reader count
31      room->readerCount--;
32
33      // if last reader unlock the write semaphore
34      if (room->readerCount == 0) {
35          sem_post(&room->writeSemaphore);//(incrementa o sem foro)
36      }
37
38      // unlock mutex
39      pthread_mutex_unlock(&room->readMutex);
40  }
41
42  void acquireWriteLock(Room *room, Client *client) {
43
44      // add a random delay to appear more natural
45      //int delay = rand() % 5;
46      //sleep(delay);
47
48      pthread_mutex_lock(&room->writeMutex);
49
50      // increment writer count
51      room->writerCount++;
52
53      //printf("WRITER COUNT: %d from client %d\n", room->writerCount,
            client->clientID);
54
55      // if first writer lock the read semaphore to block readers
56      if (room->writerCount == 1) {
57          //printf("WRITER %d IS LOCKING READ SEMAPHORE\n", client->
                clientID);
58          sem_wait(&room->readSemaphore);
59      }
```

```
60
61      // unlock the writer mutex
62      pthread_mutex_unlock(&room->writeMutex);
63
64      // lock the write semaphore
65      //printf("WRITER %d IS WAITING FOR WRITE SEMAPHORE\n", client->
            clientID);
66      sem_wait(&room->writeSemaphore);
67  }
68
69  void releaseWriteLock(Room *room, Client *client) {
70
71      sem_post(&room->writeSemaphore);
72
73      // lock the writer mutex
74      pthread_mutex_lock(&room->writeMutex);
75
76      // decrement writer count
77      room->writerCount--;
78
79      // if last writer unlock the read semaphore
80      if (room->writerCount == 0) {
81          sem_post(&room->readSemaphore);
82      }
83
84      //printf("WRITER %d IS DONE WRITING\n", client->clientID);
85
86      // unlock the writer mutex
87      pthread_mutex_unlock(&room->writeMutex);
88
89  }
```

A.3.16 src/server-statistics.h

```
1  #ifndef SERVER_STATISTICS_H
2  #define SERVER_STATISTICS_H
3
4  #include "../config/config.h"
5
```

```
6  // Guarda as estat sticas de uma sala no ficheiro 'room_stats.log'.
7  void saveRoomStatistics(int roomId, double elapsedTime);
8
9  // update game statistics
10 void updateGameStatistics(ServerConfig *config, int roomID, int
       elapsedTime, float accuracy);
11
12 // send message with statistics to client
13 void sendRoomStatistics(ServerConfig *config, Client *client);
14
15 #endif // SERVER_STATISTICS_H
```

A.3.17 src/server-statistics.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include "../../utils/logs/logs-common.h"
5  #include "../../utils/parson/parson.h"
6  #include "../../utils/network/network.h"
7  #include "../logs/logs.h"
8  #include "server-statistics.h"
9
10
11 void saveRoomStatistics(int roomId, double elapsedTime) {
12     FILE *file = fopen("room_stats.log", "a");  // Abre o ficheiro em
           modo de append
13     if (file != NULL) {
14         fprintf(file, "Sala %d - Tempo de resolu  o: %.2f segundos\n"
               , roomId, elapsedTime);
15         fclose(file);
16     } else {
17         printf("Erro ao abrir o ficheiro de estat sticas.\n");
18     }
19 }
20
21 void updateGameStatistics(ServerConfig *config, int gameID, int
       elapsedTime, float accuracy) {
22
```

```
23      FILE *file = fopen(config->gamePath, "r");
24
25      // Ler o ficheiro JSON
26      fseek(file, 0, SEEK_END);
27      long file_size = ftell(file);
28      fseek(file, 0, SEEK_SET);
29
30      char *file_content = malloc(file_size + 1);
31      if (file_content == NULL) {
32          fclose(file);
33          err_dump(config, gameID, 0, "memory allocation failed when
                updating statistics", MEMORY_ERROR);
34          return;
35      }
36
37      fread(file_content, 1, file_size, file);
38      file_content[file_size] = '\0';
39      fclose(file);
40
41      // Parse do JSON
42      JSON_Value *root_value = json_parse_string(file_content);
43      JSON_Object *root_object = json_value_get_object(root_value);
44      free(file_content);
45
46      // Get the "games" array
47      JSON_Array *games_array = json_object_get_array(root_object, "games
            ");
48
49      // iterate over the games array
50      for (int i = 0; i < json_array_get_count(games_array); i++) {
51
52          JSON_Object *game_object = json_array_get_object(games_array, i
                );
53          int currentID = (int)json_object_get_number(game_object, "id");
54
55          // if the game is found
56          if (currentID == gameID) {
57
```

```
58              int previousTimeRecord = (int)json_object_get_number(
                    game_object, "timeRecord");
59              float previousAccuracyRecord = (float)
                    json_object_get_number(game_object, "accuracyRecord");
60
61              // if elapsed time is less than the current one store it
62              if (elapsedTime < previousTimeRecord || previousTimeRecord
                    == 0) {
63                  json_object_set_number(game_object, "timeRecord",
                        elapsedTime);
64
65                  // write to log
66                  char logMessage[100];
67                  snprintf(logMessage, sizeof(logMessage), "Tempo recorde
                         atualizado para %d segundos no jogo %d",
                        elapsedTime, gameID);
68                  produceLog(config, logMessage, EVENT_NEW_RECORD, gameID
                        , 0);
69
70                  printf("Tempo recorde atualizado de %d para %d segundos
                         no jogo %d\n", previousTimeRecord, elapsedTime,
                        gameID);
71              }
72
73              // if accuracy is greater than the current one store it
74              if (accuracy > previousAccuracyRecord ||
                    previousAccuracyRecord == 0) {
75                  json_object_set_number(game_object, "accuracyRecord",
                        accuracy);
76
77                  // write to log
78                  char logMessage[100];
79                  snprintf(logMessage, sizeof(logMessage), "Precis o
                        recorde atualizada para %f no jogo %d", accuracy,
                        gameID);
80                  produceLog(config, logMessage, EVENT_NEW_RECORD, gameID
                        , 0);
81
```

```
82                  printf("Precis o recorde atualizada de %.2f para %.2f
                        no jogo %d\n", previousAccuracyRecord, accuracy,
                        gameID);
83              }
84              break;
85          }
86      }
87
88      // write the updated JSON to the file
89      file = fopen(config->gamePath, "w");
90      if (file == NULL) {
91          err_dump(config, gameID, 0, "can't open file to write updated
                statistics", MEMORY_ERROR);
92          json_value_free(root_value);
93          return;
94      }
95
96      char *serialized_string = json_serialize_to_string_pretty(
            root_value);
97      fwrite(serialized_string, 1, strlen(serialized_string), file);
98      fclose(file);
99      free(serialized_string);
100
101     // free the JSON object
102     json_value_free(root_value);
103 }
104
105 void sendRoomStatistics(ServerConfig *config, Client *client) {
106     FILE *file = fopen("room_stats.log", "r");
107     if (file == NULL) {
108         const char *errorMsg = "Erro: N o foi poss vel abrir o
                ficheiro de estat sticas.\n";
109         if (send(client->socket_fd, errorMsg, strlen(errorMsg), 0) < 0)
                {
110             // erro ao enviar mensagem de erro
111             err_dump(config, 0, client->clientID, "can't send error
                    message to client", EVENT_MESSAGE_SERVER_NOT_SENT);
112         } else {
```

```
113             printf("Erro: N o foi poss vel abrir o ficheiro de
                    estat sticas\n");
114             produceLog(config, "Erro: N o foi poss vel abrir o
                    ficheiro de estat sticas", EVENT_MESSAGE_SERVER_SENT,
                    0, client->clientID);
115         }
116         return;
117     }
118
119     char line[256];
120     char stats[1024] = "";  // Buffer para enviar todas as
            estat sticas
121     while (fgets(line, sizeof(line), file) != NULL) {
122         strncat(stats, line, sizeof(stats) - strlen(stats) - 1);  //
                Adiciona cada linha ao buffer
123     }
124     fclose(file);
125
126     // if there are no statistics send "No statistics available"
127     if (strlen(stats) == 0) {
128         strcpy(stats, "No statistics available");
129     }
130
131
132     // Envia as estat sticas ao cliente
133     if (send(client->socket_fd, stats, strlen(stats), 0) < 0) {
134         // erro ao enviar estat sticas
135         err_dump(config, 0, client->clientID, "can't send statistics to
                client", EVENT_MESSAGE_SERVER_NOT_SENT);
136     } else {
137         printf("Estat sticas enviadas ao cliente\n");
138         produceLog(config, "Estat sticas enviadas ao cliente",
                EVENT_MESSAGE_SERVER_SENT, 0, client->clientID);
139     }
140 }
```

A.3.18 src/server.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <signal.h>
#include "../../utils/logs/logs-common.h"
#include "../../utils/network/network.h"
#include "../config/config.h"
#include "server-comms.h"
#include "server-game.h"
#include "../logs/logs.h"


/**
 * Fun   o principal que configura e inicia o servidor, aceita
    conex es de clientes,
 * e cria threads para gerir essas conex es.
 *
 * @param argc O n mero de argumentos passados na linha de comando.
 * @param argv Um array de strings que cont m os argumentos da linha
    de comando.
 * O primeiro argumento (argv[1]) deve ser o caminho para o ficheiro de
     configura   o do servidor.
 * @return Retorna 0 se o servidor terminar normalmente ou 1 se faltar
    o argumento de configura   o.
 *
 * @details Esta fun   o realiza as seguintes opera   es:
 * - Verifica se o argumento de configura   o foi fornecido.
 * Se n o for, imprime uma mensagem de erro e termina o programa.
 * - Carrega as configura   es do servidor a partir do ficheiro de
    configura   o especificado.
 * - Inicializa o socket do servidor e configura-o para aceitar
    conex es de clientes.
 * - Entra num loop infinito para aceitar conex es:
 *   - Aceita uma liga   o do cliente e cria uma estrutura `Client`
    que armazena as
 * informa   es necess rias para a nova conex o.
 *   - Cria uma nova thread para gerir cada cliente, usando a fun   o
    `handleClient` para
```

```
32  * processar a comunica  o com o cliente.
33  *   - Detacha a thread para que possa ser gerida automaticamente pelo
        sistema operativo,
34  * sem necessidade de jun  o manual.
35  * - Se houver um erro ao aceitar uma conex o ou criar uma thread, a
        fun  o regista o erro
36  * no ficheiro de log especificado na configura  o.
37  *
38  * @note O loop principal executa indefinidamente at  que o servidor
        seja manualmente interrompido.
39  * O socket principal    fechado no final.
40  */
41
42 ServerConfig* svConfig;
43
44 void handleSigInt(int sig) {
45     printf("Server shutting down...\n");
46     free(svConfig->clients);
47     free(svConfig->rooms);
48
49     // destroy semaphores
50     sem_destroy(&svConfig->mutexLogSemaphore);
51     sem_destroy(&svConfig->itemsLogSemaphore);
52     sem_destroy(&svConfig->spacesSemaphore);
53
54     // destroy mutex
55     pthread_mutex_destroy(&svConfig->mutex);
56
57     free(svConfig);
58
59     exit(0);
60 }
61
62 int main(int argc, char *argv[]) {
63
64     if (argc < 2) {
65         printf("Erro: Faltam argumentos de configuracao!\n");
66         return 1;
67     }
```

```
68
69
70      //signal(SIGINT, handleSigInt);
71
72      printf("Server starting...\n");
73
74
75      // Carrega a configuracao do servidor
76      svConfig = getServerConfig(argv[1]);
77
78      // Inicializa vari veis para socket
79      int sockfd, newSockfd;
80      struct sockaddr_in serv_addr;
81
82      // Inicializar o socket
83      initializeSocket(&serv_addr, &sockfd, svConfig);
84
85      // Aguardar por conex es indefinidamente
86      for (;;) {
87
88          // Aceitar liga   o
89          if ((newSockfd = accept(sockfd, (struct sockaddr *) 0, 0)) < 0)
                {
90              // erro ao aceitar ligacao
91              err_dump(svConfig, 0, 0, "accept error",
                    EVENT_CONNECTION_SERVER_ERROR);
92          } else {
93
94              // elements to pass to thread: config, playerID, newSockfd
95              Client *client = (Client *) malloc(sizeof(Client));
96              if (client == NULL) {
97                  // erro ao alocar memoria
98                  err_dump(svConfig, 0, 0, "can't allocate memory",
                        MEMORY_ERROR);
99                  close(newSockfd);
100                 continue;
101             }
102
103             client->socket_fd = newSockfd;
```

```
104             addClient(svConfig, client);
105
106             client_data *data = (client_data *) malloc(sizeof(
                    client_data));
107             if (data == NULL) {
108                 // erro ao alocar memoria
109                 err_dump(svConfig, 0, 0, "can't allocate memory",
                        MEMORY_ERROR);
110                 removeClient(svConfig, client);
111                 free(client);
112                 close(newSockfd);
113                 continue;
114             }
115
116             data->config = svConfig;
117             data->client = client;
118
119             // create a new thread to handle the client
120             pthread_t thread;
121             if (pthread_create(&thread, NULL, handleClient, (void *)
                    data) != 0) {
122                 // erro ao criar thread
123                 err_dump(svConfig, 0, 0, "can't create client thread",
                        EVENT_SERVER_THREAD_ERROR);
124                 removeClient(svConfig, client);
125                 free(client);
126                 free(data);
127                 close(newSockfd);
128                 continue;
129             }
130
131             // detach the thread
132             pthread_detach(thread);
133         }
134
135         // Create a thread to handle consume for logs
136         pthread_t logThread;
137         if (pthread_create(&logThread, NULL, consumeLog, (void *)
                svConfig) != 0) {
```

```
138             // erro ao criar thread
139             err_dump(svConfig, 0, 0, "can't create log consumer thread"
                    , EVENT_THREAD_NOT_CREATE);
140             continue;
141         }
142     }
143
144     close(sockfd);
145     free(svConfig);
146     return 0;
147 }
```

A.4 Client

A.4.1 config/client.conf

```
SERVER_IP = 127.0.0.1
SERVER_PORT = 8080
SERVER_HOSTNAME = vpn.uma.pt
LOG_PATH = client/data/
IS_MANUAL = 0
IS_PREMIUM = 0
DIFFICULTY = 1
```

A.4.2 config/config.h

```
#ifndef CONFIG_H
#define CONFIG_H

#include <stdbool.h>

/**
 * Estrutura que armazena as configura  es do cliente, incluindo
    informa  es de rede, identifica  o,
 * e op  es de jogo.
 */
typedef struct {
```

```
11      char serverIP[256];          /**< Endere o IP do servidor. */
12      int serverPort;              /**< Porta do servidor. */
13      char serverHostName[256];    /**< Nome do host do servidor. */
14      int clientID;                /**< ID  nico  do cliente. */
15      char sourceLogPath[256];        /**< Caminho para o ficheiro de log.
            */
16      char logPath[512];           /**< Caminho para o ficheiro de log. */
17      bool isManual;               /**< Define se o jogo ser  jogado em
           modo manual. */
18      bool isPremium;              /**< Define se o jogador    premium. */
19      int difficulty;              /**< N vel de dificuldade do jogo (ex
           .: 1 - f cil , 2 - m dio , 3 - dif cil). */
20      int readsCount;              /**< N mero de leituras de mensagens
           do servidor. */
21      int writesCount;             /**< N mero de escritas de mensagens
           para o servidor. */
22  } clientConfig;
23
24  // Carrega as configura  es do cliente a partir de um ficheiro de
       configura  o.
25  clientConfig *getClientConfig(char *configPath);
26
27  #endif // CONFIG_H
```

A.4.3 config/config.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <errno.h>
5   #include "config.h"
6
7   /**
8    * Carrega as configura  es do cliente a partir de um ficheiro de
       configura  o especificado.
9    *
10   * @param configPath O caminho para o ficheiro de configura  o que
       cont m as defini  es do cliente.
```

```
11  * @return Uma estrutura `clientConfig` preenchida com as
        configura  es do cliente.
12  *
13  * @details A fun  o realiza as seguintes opera  es:
14  * - Abre o ficheiro de configura  o em modo de leitura. Se o
        ficheiro n o puder ser aberto, o programa termina.
15  * - L  cada linha do ficheiro e extrai as configura  es usando `
        sscanf`:
16  *   - Endere o IP do servidor (`SERVER_IP`).
17  *   - Porta do servidor (`SERVER_PORT`).
18  *   - Nome do host do servidor (`SERVER_HOSTNAME`).
19  *   - Caminho para o ficheiro de log (`LOG_PATH`).
20  *   - Modo de jogo (manual ou autom tico) convertido para booleano (`
        IS_MANUAL`).
21  *   - N vel de dificuldade do jogo (`DIFFICULTY`).
22  * - Remove caracteres de nova linha de cada linha lida para garantir
        que os dados s o processados corretamente.
23  * - Imprime as configura  es carregadas no terminal.
24  * - Fecha o ficheiro de configura  o e retorna a estrutura `
        clientConfig`.
25  */
26
27 clientConfig *getClientConfig(char *configPath) {
28
29     // Cria uma vari vel do tipo clientConfig
30     clientConfig *config = (clientConfig *)malloc(sizeof(clientConfig))
           ;
31     if (config == NULL) {
32         fprintf(stderr, "Memory allocation failed\n");
33         return NULL;
34     }
35     memset(config, 0, sizeof(clientConfig));  // Inicializa a estrutura
           clientConfig
36
37     // Abre o ficheiro 'config.txt' em modo de leitura
38     FILE *file;
39     file = fopen(configPath, "r");
40
```

```
41      // Se o ficheiro n o existir, imprime uma mensagem de erro e
            termina o programa
42      if (file == NULL) {
43          //fprintf(stderr, "Couldn't open %s: %s\n", configPath,
                strerror(errno));
44          exit(1);
45      }
46
47      char line[256];
48      if (fgets(line, sizeof(line), file) != NULL) {
49          // Remover a nova linha, se houver
50          line[strcspn(line, "\n")] = 0;
51          sscanf(line, "SERVER_IP = %s", config->serverIP);
52      }
53
54      if (fgets(line, sizeof(line), file) != NULL) {
55          // Remover a nova linha, se houver
56          line[strcspn(line, "\n")] = 0;
57          sscanf(line, "SERVER_PORT = %d", &config->serverPort);
58      }
59
60      if (fgets(line, sizeof(line), file) != NULL) {
61          // Remover a nova linha, se houver
62          line[strcspn(line, "\n")] = 0;
63          sscanf(line, "SERVER_HOSTNAME = %s", config->serverHostName);
64      }
65
66      if (fgets(line, sizeof(line), file) != NULL) {
67          // Remover a nova linha, se houver
68          line[strcspn(line, "\n")] = 0;
69          sscanf(line, "LOG_PATH = %s", config->sourceLogPath);
70      }
71
72      // add temporary client id 0
73      char logPath[512];
74      snprintf(logPath, sizeof(logPath), "%sclient-%d-logs.json", config
            ->sourceLogPath, 0);
75      // set logPath to the new logPath
76      strcpy(config->logPath, logPath);
```

```
77
78      if (fgets(line, sizeof(line), file) != NULL) {
79          int isManual;
80          // Remover a nova linha, se houver
81          line[strcspn(line, "\n")] = 0;
82          sscanf(line, "IS_MANUAL = %d", &isManual);
83
84          // Converte o valor lido para booleano
85          config->isManual = isManual == 1 ? true : false;
86      }
87
88      if (fgets(line, sizeof(line), file) != NULL) {
89          int isPremium;
90          // Remover a nova linha, se houver
91          line[strcspn(line, "\n")] = 0;
92          sscanf(line, "IS_PREMIUM = %d", &isPremium);
93
94          // Converte o valor lido para booleano
95          config->isPremium = isPremium == 1 ? true : false;
96      }
97
98      if (fgets(line, sizeof(line), file) != NULL) {
99          // Remover a nova linha, se houver
100         line[strcspn(line, "\n")] = 0;
101         sscanf(line, "DIFFICULTY = %d", &config->difficulty);
102     }
103
104     // id to 0
105     config->clientID = 0;
106
107     // Fecha o ficheiro
108     fclose(file);
109
110     printf("IP do servidor: %s\n", config->serverIP);
111     printf("Porta do servidor: %d\n", config->serverPort);
112     printf("Hostname do servidor: %s\n", config->serverHostName);
113     printf("Log path do cliente: %s\n", config->logPath);
114     printf("Modo: %s\n", config->isManual ? "manual" : "automatico");
115     printf("Cliente %s premium.\n", config->isPremium ? "SIM" : "NAO");
```

```
116
117     if (config->difficulty == 1) {
118         printf("Dificuldade: Facil\n");
119     } else if (config->difficulty == 2) {
120         printf("Dificuldade: Medio\n");
121     } else {
122         printf("Dificuldade: Dificil\n");
123     }
124
125     // Retorna a vari vel config
126     return config;
127 }
```

A.4.4 logs/logs.h

```
1  #ifndef LOGS_H
2  #define LOGS_H
3
4  #include "../config/config.h"
5
6  // Fun   o externa para registar um erro no log e terminar o programa.
7  void err_dump_client(char *filePath, int idJogo, int idJogador, char *
       msg, char *event);
8
9
10 #endif // LOGS_H
```

A.4.5 logs/logs.c

```
1  #include <string.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include "logs.h"
5  #include "../../utils/logs/logs-common.h"
6
7
8  void err_dump_client(char *filePath, int idJogo, int idJogador, char *
       msg, char *event) {
```

```
9
10      // produce log message
11      writeLogJSON(filePath, idJogo, idJogador, msg);
12
13      // imprime a mensagem de erro e termina o programa
14      perror(msg);
15      exit(1);
16  }
```

A.4.6 src/client-comms.h

```
1   #ifndef CLIENT_COMMS_H
2   #define CLIENT_COMMS_H
3
4   #include <stdbool.h>
5   #include "../../utils/network/network.h"
6   #include "../config/config.h"
7
8   // Estabelece uma liga  o TCP com o servidor.
9   void connectToServer(struct sockaddr_in *serv_addr, int *socketfd,
        clientConfig *config);
10
11  // Envia uma mensagem para fechar a conex o com o servidor.
12  void closeConnection(int *socketfd, clientConfig *config);
13
14  #endif // CLIENT_COMMS_H
```

A.4.7 src/client-comms.c

```
1   #include "../../utils/parson/parson.h"
2   #include "../../utils/network/network.h"
3   #include "../../utils/logs/logs-common.h"
4   #include "../logs/logs.h"
5   #include "client-comms.h"
6
7   /**
8    * Estabelece uma liga  o TCP ao servidor especificado na
         configura  o do cliente.
```

```
9  *
10 * @param serv_addr Um pointer para a estrutura ‘sockaddr_in‘ que ser
       configurada
11 * com o endere o e a porta do servidor.
12 * @param socketfd Um pointer para o descritor de socket que ser
       criado e usado para a conex o.
13 * @param config A estrutura ‘clientConfig‘ que cont m as
       informa  es de configura  o do cliente,
14 * incluindo o IP e a porta do servidor.
15 *
16 * @details Esta fun  o realiza os seguintes passos:
17 * - Inicializa a estrutura ‘serv_addr‘ com ‘memset‘ para garantir que
       todos os campos
18 * est o corretamente definidos.
19 * - Converte o endere o IP do servidor de formato textual para
       bin rio com ‘inet_pton‘.
20 * Se falhar, regista o erro e termina.
21 * - Configura a porta do servidor e cria um socket TCP (do tipo ‘
       SOCK_STREAM‘).
22 * Se a cria  o do socket falhar, regista o erro.
23 * - Tenta estabelecer uma liga  o ao servidor com ‘connect‘. Se a
       liga  o falhar, regista o erro.
24 * - Se a conex o for bem-sucedida, imprime uma mensagem de
       confirma  o e regista o evento de
25 * liga  o estabelecida no ficheiro de log.
26 */
27
28 void connectToServer(struct sockaddr_in *serv_addr, int *socketfd,
       clientConfig *config) {
29     /* Primeiro uma limpeza preventiva! memset    mais eficiente que
           bzero
30        Dados para o socket stream: tipo */
31
32     memset(serv_addr, 0, sizeof(*serv_addr));
33     serv_addr->sin_family = AF_INET; // endere os iternet DARPA
34
35     /* Converter serverIP para binario*/
36     if (inet_pton(AF_INET, config->serverIP, &serv_addr->sin_addr) <=
           0) {
```

```
37          // erro ao converter serverIP para binario
38          err_dump_client(config->logPath, 0, config->clientID, "can't
                get server address", EVENT_CONNECTION_CLIENT_NOT_ESTABLISHED
                );
39      }
40
41      /* Dados para o socket stream: porta do servidor */
42      serv_addr->sin_port = htons(config->serverPort);
43
44      /* Cria socket tcp (stream) */
45      if ((*socketfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
46          // erro ao abrir socket
47          err_dump_client(config->logPath, 0, config->clientID, "can't
                open datagram socket",
                EVENT_CONNECTION_CLIENT_NOT_ESTABLISHED);
48      }
49
50      /* Estabelece liga  o com o servidor */
51      if (connect(*socketfd, (struct sockaddr *)serv_addr, sizeof(*
            serv_addr)) < 0) {
52          // erro ao conectar ao servidor
53          err_dump_client(config->logPath, 0, config->clientID, "can't
                connect to server", EVENT_CONNECTION_CLIENT_NOT_ESTABLISHED)
                ;
54      }
55
56      /* Print conexao estabelecida */
57      printf("Conexao estabelecida com o servidor %s:%d\n", config->
            serverIP, config->serverPort);
58      writeLogJSON(config->logPath, 0, config->clientID,
            EVENT_CONNECTION_CLIENT_ESTABLISHED);
59  }
60
61
62  /**
63   * Envia uma mensagem ao servidor para fechar a conex o e regista o
        evento no log.
64   *
```

```
65  * @param socketfd Um pointer para o descritor de socket usado para a
        comunica  o com o servidor.
66  * @param config A estrutura `clientConfig` que cont m as
        configura  es do cliente,
67  * incluindo o caminho do log e o ID do cliente.
68  *
69  * @details A fun  o realiza o seguinte:
70  * - Envia uma mensagem ao servidor indicando que a conex o deve ser
        fechada.
71  * - Se o envio falhar, regista o erro no ficheiro de log utilizando `
        err_dump_client`.
72  * - Se o envio for bem-sucedido, imprime uma mensagem a indicar que a
        conex o est  a ser encerrada e regista o evento de encerramento
        no log.
73  */
74
75  void closeConnection(int *socketfd, clientConfig *config) {
76
77      // send close connection message to the server
78      if (send(*socketfd, "closeConnection", strlen("closeConnection"),
            0) < 0) {
79          err_dump_client(config->logPath, 0, config->clientID, "can't
                send close connection message to server",
                EVENT_MESSAGE_CLIENT_NOT_SENT);
80      } else {
81          printf("Closing connection...\n");
82          writeLogJSON(config->logPath, 0, config->clientID,
                EVENT_CONNECTION_CLIENT_CLOSED);
83      }
84
85      // close the socket
86      close(*socketfd);
87
88      // free config
89      free(config);
90
91      // exit the program
92      exit(0);
93  }
```

A.4.8 src/client-game.h

```c
#ifndef CLIENT_GAME_H
#define CLIENT_GAME_H

#include <stdbool.h>
#include "client-menus.h"

// Estrutura para armazenar estat sticas da resolu  o de uma linha
typedef struct {
    double tempoResolucao;
    int tentativas;
    int acertos;
    double percentagemAcerto;
} EstatisticasLinha;

// Fun  o para verificar a linha no buffer
int verifyLine(char *buffer);

// Fun  o para resolver uma linha
void resolveLine(char *buffer, char *line, int row, int difficulty,
    EstatisticasLinha *estatisticas);

// Fun  o para verificar se um n mero pode ser colocado numa c lula
    espec fica do tabuleiro
bool isValid(JSON_Array *board_array, int row, int col, int num, int
    difficulty);

// Envia linhas de jogo ao servidor e processa o tabuleiro atualizado.
void playGame(int *socketfd, clientConfig *config);

// Exibe o tabuleiro de jogo recebido do servidor.
char *showBoard(int *socketfd, clientConfig *config);

// Acaba o jogo
void finishGame(int *socketfd, clientConfig *config, EstatisticasLinha
    *estatisticas);
```

```
32
33 #endif // CLIENT_GAME_H
```

A.4.9 src/client-game.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <time.h>
5  #include "../../utils/parson/parson.h"
6  #include "../../utils/logs/logs-common.h"
7  #include "../logs/logs.h"
8  #include "client-game.h"
9
10 /**
11  * Verifica se a linha cont m exatamente 9 d gitos num ricos (de '0'
        a '9').
12  *
13  * @param buffer Uma string que cont m a linha a ser verificada.
14  * @return -1 se a linha n o tiver exatamente 9 caracteres ou se
       contiver caracteres n o num ricos.
15  *         0 se a linha for v lida (cont m exatamente 9 d gitos
       num ricos).
16  *
17  * @details A fun   o faz o seguinte:
18  * - Verifica se o comprimento da string `buffer`    exatamente 9.
19  * - Percorre cada car cter na string para verificar se    um d gito
       num rico.
20  * - Se qualquer car cter n o for um d gito ou se o comprimento for
       diferente de 9,
21  * retorna -1; caso contr rio, retorna 0.
22  */
23
24 int verifyLine(char *buffer) {
25     if (strlen(buffer) != 9) {
26         return -1;
27     }
28     for (int i = 0; i < 9; i++) {
29         if (buffer[i] < '0' || buffer[i] > '9') {
```

```
30              return -1;
31          }
32      }
33      return 0;
34  }
35
36  /**
37   * Resolve uma linha do tabuleiro de jogo, preenchendo c lulas vazias
        com n meros v lidos.
38   *
39   * @param buffer Uma string JSON que cont m o estado atual do
        tabuleiro.
40   * @param line Uma string de 10 caracteres onde a linha resolvida ser
         armazenada (9 d gitos + terminador nulo).
41   * @param row O n mero da linha (0-indexado) que ser  resolvida.
42   * @param difficulty O n vel de dificuldade usado para validar os
        n meros inseridos.
43   *
44   * @details A fun  o faz o seguinte:
45   * - Inicializa o gerador de n meros aleat rios.
46   * - Faz o parse da string JSON para obter o tabuleiro de jogo.
47   * - Itera por cada c lula da linha especificada:
48   *   - Se a c lula estiver vazia (valor 0), tenta n meros de 1 a 9
        at  encontrar um v lido.
49   *   - Se a c lula j  tiver um valor, copia-o para a string `line`.
50   * - Usa a fun  o `isValid` para verificar se um n mero  v lido
        para a posi  o dada, tendo em conta a dificuldade.
51   * - Termina a string `line` com o caractere nulo (`'\0'`) e liberta a
        mem ria alocada para o objeto JSON.
52   * - Imprime a linha gerada no terminal.
53   */
54
55  void resolveLine(char *buffer, char *line, int row, int difficulty,
       EstatisticasLinha *estatisticas) {
56      // Inicializar as estat sticas
57      // Seed random number generator
58      srand(time(NULL));
59
60      printf("Resolvendo linha %d...\n", row + 1);
```

```
61      //printf("Buffer recebido: %s\n", buffer);
62
63      // Parse the JSON object from the buffer
64      JSON_Value *root_value = json_parse_string(buffer);
65      JSON_Object *root_object = json_value_get_object(root_value);
66      JSON_Array *board_array = json_object_get_array(root_object, "board
            ");
67
68      // Get the line array from the board array
69      JSON_Array *linha_array = json_array_get_array(board_array, row);
70
71      // Iterate through each cell in the line
72      for (int i = 0; i < 9; i++) {
73          int cell_value = (int)json_array_get_number(linha_array, i);
74
75          // If the cell is empty (value is 0)
76          if (cell_value == 0) {
77              // Try different numbers until a valid one is found
78              for (int num = 1; num <= 9; num++) {
79                  estatisticas->tentativas++; // Aumenta o n mero de
                        tentativas a cada tentativa de n mero
80
81                  // Check if the number is valid for this cell
82                  if (isValid(board_array, row, i, num, difficulty)) {
83                      // Set the cell value to the number
84                      line[i] = num + '0'; // Convert to character for
                            string representation
85
86                      // Se o n mero for o correto, aumenta o n mero de
                            acertos
87                      if (num == (int)json_array_get_number(linha_array,
                            i)) {
88                          estatisticas->acertos++;
89                      }
90                      break; // Se for encontrado um n mero v lido para
                            de aumentar os acertos
91                  }
92              }
93          } else {
```

```
94              // The cell is already filled, copy the value to the line
95              line[i] = cell_value + '0';
96              estatisticas->acertos++; // Aumenta o n mero de acertos
97          }
98      }
99
100     // Terminate the string
101     line[9] = '\0';
102
103     // Free the JSON objects
104     json_value_free(root_value);
105
106     // Calcular a percentagem de acerto
107     estatisticas->percentagemAcerto = (estatisticas->acertos * 100) / (
            float)estatisticas->tentativas;
108
109     // Exibir as estat sticas
110     //printf("Linha gerada: %s\n", line);
111     printf("Tentativas: %d\n", estatisticas->tentativas);
112     printf("Acertos: %d\n", estatisticas->acertos);
113     printf("Percentagem de acerto: %.2f%%\n", estatisticas->
            percentagemAcerto);
114 }
115
116
117
118 /**
119  * Verifica se um n mero pode ser colocado numa c lula espec fica do
        tabuleiro de acordo
120  * com as regras do jogo e o n vel de dificuldade.
121  *
122  * @param board_array Um pointer para o array JSON que representa o
        tabuleiro de jogo.
123  * @param row  ndice da linha da c lula a ser verificada.
124  * @param col  ndice da coluna da c lula a ser verificada.
125  * @param num  n mero a ser verificado.
126  * @param difficulty  n vel de dificuldade que determina as regras de
        verifica  o (1, 2 ou 3).
```

```
127  * @return `true` se o n mero puder ser colocado na c lula sem violar
         as regras, `false` caso contr rio.
128  *
129  * @details A fun  o verifica a validade do n mero nas seguintes
         condi  es:
130  * - **Linha**: O n mero n o pode j  existir na mesma linha (exceto
         na coluna atual).
131  * - **Coluna**: Se a dificuldade for 2 ou superior, o n mero n o
         pode j  existir na mesma coluna (exceto na linha atual).
132  * - **Subgrade 3x3**: Se a dificuldade for 3, o n mero n o pode j
         existir na mesma subgrade 3x3 (exceto na c lula atual).
133  *
134  * @note A fun  o ajusta a complexidade da verifica  o com base no
         n vel de dificuldade fornecido:
135  * - Dificuldade 1: Apenas verifica a linha.
136  * - Dificuldade 2: Verifica a linha e a coluna.
137  * - Dificuldade 3: Verifica a linha, a coluna, e a subgrade 3x3.
138  */
139
140 bool isValid(JSON_Array *board_array, int row, int col, int num, int
        difficulty) {
141
142     // Check row
143     for (int i = 0; i < 9; i++) {
144         if (i != col && json_array_get_number(json_array_get_array(
                board_array, row), i) == num) {
145             return false;
146         }
147     }
148
149     // Check column
150     if (difficulty >=2) {
151         for (int i = 0; i < 9; i++) {
152             if (i != row && json_array_get_number(json_array_get_array(
                    board_array, i), col) == num) {
153                 return false;
154             }
155         }
156     }
```

```
157
158     // Check 3x3 subgrid
159     if (difficulty == 3) {
160         int startRow = (row / 3) * 3;
161         int startCol = (col / 3) * 3;
162         for (int i = 0; i < 3; i++) {
163             for (int j = 0; j < 3; j++) {
164                 if (i + startRow != row && j + startCol != col &&
165                     json_array_get_number(json_array_get_array(
                        board_array, i + startRow), j + startCol) == num
                        ) {
166                     return false;
167                 }
168             }
169         }
170     }
171     return true;
172 }
173
174 void playGame(int *socketfd, clientConfig *config) {
175
176     // buffer for the board
177     char buffer[BUFFER_SIZE];
178     memset(buffer, 0, sizeof(buffer));
179
180     printf("A jogar...\n");
181
182     // set default values of reads and writes
183     config->readsCount = 0;
184     config->writesCount = 0;
185
186     char *board;
187     board = showBoard(socketfd, config);
188
189     // get the current line
190     char *boardSplit = strtok(board, "\n");
191     char *token = strtok(NULL, "\n");
192     int currentLine = atoi(token);
193
```

```
194     char tempString[BUFFER_SIZE]; // Allocate a temporary buffer
195     strcpy(tempString, boardSplit); // Copy the original board data
196 
197     printf("Linha atual: %d\n", currentLine);
198 
199     EstatisticasLinha *estatisticas;
200     estatisticas = (EstatisticasLinha *)malloc(sizeof(EstatisticasLinha
        ));
201     estatisticas->tentativas = 0; // Iniciar com 0 tentativas
202     estatisticas->acertos = 0;
203     estatisticas->percentagemAcerto = 0.0;
204     estatisticas->tempoResolucao = 0.0;
205 
206     free(board);
207     writeLogJSON(config->logPath, 0, config->clientID, "Started playing
          the game");
208 
209     // Enviar linhas inseridas pelo utilizador e receber o board
        atualizado
210     while (currentLine <= 9) {
211 
212         int validLine = 0;  // Vari vel para controlar se a linha
             est  correta
213 
214         // line to send to server
215         char line[10];
216 
217         // inicializa o buffer com '0' e terminador nulo
218         memset(line, '0', sizeof(line));
219 
220         while (!validLine) {
221 
222             if (config->isManual) {
223 
224                 printf("Insira valores para a linha %d do board (
                    exactamente 9 digitos):\n", currentLine);
225                 scanf("%s", line);
226                 char logMessage[256];
```

```
227                 snprintf(logMessage, sizeof(logMessage), "Manual input
                        for board line %d", currentLine);
228                 writeLogJSON(config->logPath, 0, config->clientID,
                        logMessage);
229
230             } else {
231
232                 // Passa a vari vel estatisticas para a fun   o
                        resolveLine
233                 resolveLine(tempString, line, currentLine - 1, config->
                        difficulty, estatisticas);
234                 char logMessage[256];
235                 snprintf(logMessage, sizeof(logMessage), "Auto-solving
                        the board line %d", currentLine);
236                 writeLogJSON(config->logPath, 0, config->clientID,
                        logMessage);
237
238             }
239
240             // Enviar a linha ao servidor
241             if (send(*socketfd, line, sizeof(line), 0) < 0) {
242                 err_dump_client(config->logPath, 0, config->clientID, "
                        can't send lines to server",
                        EVENT_MESSAGE_CLIENT_NOT_SENT);
243                 continue;
244             } else {
245                 printf("Linha enviada: %s\n", line);
246                 // Incrementa o contador de escritas
247                 config->writesCount++;
248                 char logMessage[256];
249                 snprintf(logMessage, sizeof(logMessage), "Sent line %d
                        to server", currentLine);
250                 writeLogJSON(config->logPath, 0, config->clientID,
                        logMessage);
251             }
252
253             char *board;
254             board = showBoard(socketfd, config);
255             // get the current line
```

```
256             boardSplit = strtok(board, "\n");
257             char *token = strtok(NULL, "\n");
258             int serverLine = atoi(token);
259 
260             strcpy(tempString, boardSplit); // Copy the original board
                    data
261 
262             //printf("Linha do servidor: %d\n", serverLine);
263             if (serverLine > currentLine) {
264                 validLine = 1;
265                 currentLine = serverLine;
266                 char logMessage[256];
267                 snprintf(logMessage, sizeof(logMessage), "Received line
                        %d from server", currentLine);
268                 writeLogJSON(config->logPath, 0, config->clientID,
                        logMessage);
269             } else {
270                 printf("Linha %d incorreta. Tente novamente.\n",
                        currentLine);
271                 char logMessage[256];
272                 snprintf(logMessage, sizeof(logMessage), "Received
                        incorrect line %d from server", currentLine);
273                 writeLogJSON(config->logPath, 1, config->clientID,
                        logMessage);
274             }
275 
276             // print read and write counts
277             printf("Number of Reads: %d\n", config->readsCount);
278             printf("Number of Writes: %d\n", config->writesCount);
279 
280             free(board);
281         }
282     }
283 
284     finishGame(socketfd, config, estatisticas);
285     writeLogJSON(config->logPath, 0, config->clientID, "Game finished")
            ;
286 }
287 
```

```
288 /**
289  * Exibe o tabuleiro de jogo a partir de uma string JSON e regista o
        evento no log.
290  *
291  * @param buffer Uma string JSON que cont m o estado do tabuleiro.
292  * @param logFileName O caminho para o ficheiro de log onde o evento
        ser  registado.
293  * @param playerID O identificador do jogador que est  a visualizar o
        tabuleiro.
294  *
295  * @details A fun  o faz o seguinte:
296  * - Faz o parse da string JSON para obter o objeto ‘board‘ e o ‘gameID
        ‘.
297  * - Imprime o tabuleiro no formato de uma grelha 9x9 com separadores
        visuais.
298  * - Regista o evento de visualiza  o do tabuleiro no ficheiro de log
        .
299  * - Liberta a mem ria alocada para o objeto JSON ap s a opera  o.
300  */
301
302 char *showBoard(int *socketfd, clientConfig *config) {
303
304     // buffer for the board
305     // Allocate memory for buffer on the heap
306     char *buffer = (char *)malloc(BUFFER_SIZE);
307     if (buffer == NULL) {
308         perror("Failed to allocate memory");
309         return NULL;
310     }
311     memset(buffer, 0, BUFFER_SIZE);
312
313     printf("Received board from server...\n");
314
315     // receive the board from the server
316     if (recv(*socketfd, buffer, BUFFER_SIZE, 0) < 0) {
317         // error receiving board from server
318         err_dump_client(config->logPath, 0, config->clientID, "can't
            receive board from server",
            EVENT_MESSAGE_CLIENT_NOT_RECEIVED);
```

```
319         free(buffer);
320
321     } else {
322
323         if (strcmp(buffer, "No rooms available") == 0) {
324             printf("No rooms available\n");
325             free(buffer);
326             return NULL;
327         }
328     }
329
330     //printf("Board received: %s\n", buffer);
331
332     char *board = strtok(buffer, "\n");
333     char *token = strtok(NULL, "\n");
334     int serverLine = atoi(token);
335     //printf("Linha do servidor: %d\n", serverLine);
336
337     // get the JSON object from the buffer
338     JSON_Value *root_value = json_parse_string(board);
339     JSON_Object *root_object = json_value_get_object(root_value);
340
341     // get the game ID
342     int gameID = (int)json_object_get_number(root_object, "id");
343
344     // print the board
345     printf("----------------------------------------\n");
346     printf("BOARD ID: %d  PLAYER ID: %d   %s\n", gameID, config->
        clientID, config->isPremium ? "PREMIUM" : "NON-PREMIUM");
347     printf("----------------------------------------\n");
348
349     // get the board array from the JSON object
350     JSON_Array *board_array = json_object_get_array(root_object, "board
        ");
351
352     for (int i = 0; i < json_array_get_count(board_array); i++) {
353
354         // get the line array from the board array
355         JSON_Array *linha_array = json_array_get_array(board_array, i);
```

```
356
357         printf("| line %d -> | ", i + 1);
358
359         // print the line array
360         for (int j = 0; j < 9; j++) {
361             printf("%d ", (int)json_array_get_number(linha_array, j));
362             if ((j + 1) % 3 == 0) {
363                 printf("| ");
364             }
365         }
366         printf("\n");
367         if ((i + 1) % 3 == 0) {
368             printf("-------------------------------------\n");
369         }
370     }
371
372     // Concatenate board and server line
373     char tempString[BUFFER_SIZE]; // Allocate a temporary buffer
374     strcpy(tempString, board); // Copy the original board data
375     strcat(tempString, "\n"); // Concatenate newline
376     char serverLineStr[10];
377     sprintf(serverLineStr, "%d", serverLine);
378     strcat(tempString, serverLineStr); // Concatenate server line
379
380     // Copy the modified string to the original buffer
381     strcpy(buffer, tempString);
382
383     // Free the JSON object
384     json_value_free(root_value);
385
386     writeLogJSON(config->logPath, gameID, config->clientID,
        EVENT_BOARD_SHOW);
387
388     // increase the reads count
389     config->readsCount++;
390
391     return buffer;
392 }
393
```

```
394 void finishGame(int *socketfd, clientConfig *config, EstatisticasLinha
        *estatisticas) {
395
396     char buffer[BUFFER_SIZE];
397     memset(buffer, 0, sizeof(buffer));
398
399     // send the accuracy to the server
400     char accuracyString[10];
401     sprintf(accuracyString, "%.2f", estatisticas->percentagemAcerto);
402
403     // send the accuracy to the server
404     if (send(*socketfd, accuracyString, strlen(accuracyString), 0) < 0)
            {
405
406         // error sending accuracy to server
407         err_dump_client(config->logPath, 0, config->clientID, "can't
                send accuracy to server", EVENT_MESSAGE_CLIENT_NOT_SENT);
408     } else {
409
410         // show the accuracy sent
411         //printf("Accuracy sent: %s\n", accuracyString);
412          // Log accuracy sent
413         char logMessage[256];
414         snprintf(logMessage, sizeof(logMessage), "%s: sent accuracy: %s
                ", EVENT_MESSAGE_CLIENT_SENT, accuracyString);
415         writeLogJSON(config->logPath, 0, config->clientID, logMessage);
                // Log de envio de precis o
416     }
417
418     printf("Tentativas: %d\n", estatisticas->tentativas);
419     printf("Acertos: %d\n", estatisticas->acertos);
420     printf("Percentagem de acerto: %.2f%%\n", estatisticas->
            percentagemAcerto);
421
422
423
424     // receive the final message from the server
425     if (recv(*socketfd, buffer, sizeof(buffer), 0) < 0) {
426
```

```
427             // error receiving final board from server
428             err_dump_client(config->logPath, 0, config->clientID, "can't
                    receive final board from server",
                    EVENT_MESSAGE_CLIENT_NOT_RECEIVED);
429
430         } else {
431
432             // show the final message
433             printf("%s", buffer);
434         }
435
436         free(estatisticas);
437     }
```

A.4.10 src/client-menus.h

```
1   #ifndef CLIENT_MENUS_H
2   #define CLIENT_MENUS_H
3
4   #include <stdbool.h>
5   #include "../config/config.h"
6   #include "client-comms.h"
7
8   #define INTERFACE_MENU "1. Play\n2. Statistics\n3. Exit\nChoose an
        option: "
9   #define INTERFACE_PLAY_MENU "1. Singleplayer\n2. Multiplayer\n3. Back\
        n4. Exit\nChoose an option: "
10  #define INTERFACE_SELECT_SINGLEPLAYER_GAME "1. New Random SinglepLayer
        Game\n2. New Specific Singleplayer Game\n3. Back\n4. Exit\nChoose an
         option: "
11  #define INTERFACE_SELECT_MULTIPLAYER_GAME "1. New Random Multiplayer
        Game\n2. New Specific Multiplayer Game\n3. Back\n4. Exit\nChoose an
        option: "
12  #define INTERFACE_SELECT_MULTIPLAYER_MENU "1. Create a New Multiplayer
        Game\n2. Join a Multiplayer Game\n3. Back\n4. Exit\nChoose an option
        : "
13  #define INTERFACE_POSSIBLE_SYNCHRONIZATION "1. Readers-Writers\n2.
        Barber-Shop with static priority\n3. Barber-shop with dynamic
```

```
    priority\n4. Barber-Shop with FIFO\n5. Back\n6. Exit\nChoose an
    option: "
14
15  // Exibe o menu principal e processa as op   es do utilizador.
16  void showMenu(int *socketfd, clientConfig *config);
17
18  // Exibe o menu de op   es de jogo e processa as escolhas.
19  void showPlayMenu(int *socketfd, clientConfig *config);
20
21  // Exibe o menu para iniciar um novo jogo single player.
22  void showSinglePLayerMenu(int *socketfd, clientConfig *config);
23
24  // Exibe o menu de estatisticas
25  void showStatisticsMenu(int *socketfd, clientConfig *client);
26
27  // Exibe o menu de op   es multiplayer.
28  void showMultiPlayerMenu(int *socketfd, clientConfig *config);
29
30  // Permite ao utilizador criar ou entrar num novo jogo multiplayer.
31  void createNewMultiplayerGame(int *socketfd, clientConfig *config);
32
33  // shows menu for possible synchronizations
34  void showPossibleSynchronizations(int *socketfd, clientConfig *config);
35
36  // Exibe o menu de op   es multiplayer.
37  void showMultiPlayerMenu(int *socketfd, clientConfig *config);
38
39  // Solicita e exibe as salas multiplayer dispon veis.
40  void showMultiplayerRooms(int *socketfd, clientConfig *config);
41
42  // Exibe jogos dispon veis para o utilizador selecionar.
43  void showGames(int *socketfd, clientConfig *config, bool isSinglePlayer
    );
44
45  // Recebe um temporizador do servidor e atualiza o tempo restante.
46  void receiveTimer(int *socketfd, clientConfig *config);
47
48  // Inicia um jogo single player aleat rio.
49  void playSinglePlayerGame(int *socketfd, clientConfig *config);
```

```
50
51 // Inicia um jogo multiplayer aleat rio.
52 void playMultiPlayerGame(int *socketfd, clientConfig *config, char *
       synchronization);
53
54 // Fun   o para mostrar timer update
55 int showTimerUpdate(char *buffer, int timeLeft);
56
57 #endif // CLIENT_MENUS_H
```

A.4.11 src/client-menus.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include "../../utils/logs/logs-common.h"
4  #include "../logs/logs.h"
5  #include "client-menus.h"
6
7  /*
8   * Exibe o menu principal do cliente e processa as op   es
        selecionadas pelo utilizador.
9   *
10  * @param socketfd Um pointer para o descritor de socket usado para a
        comunica   o com o servidor.
11  * @param config A estrutura `clientConfig` que cont m as
        configura   es do cliente.
12  *
13  * @details A fun   o faz o seguinte:
14  * - Exibe o menu principal e solicita ao utilizador que escolha uma
        op   o.
15  * - Valida a entrada do utilizador para garantir que    um n mero.
16  * - Processa a op   o escolhida:
17  *    - Op   o 1: Chama a fun   o `showPlayMenu` para exibir o menu de
        jogo.
18  *    - Op   o 2: Chama a fun   o (comentada) para exibir as
        estat sticas (ainda por implementar).
19  *    - Op   o 3: Fecha a conex o com o servidor.
20  * - Repete o loop at   que o utilizador escolha uma op   o v lida (1
        a 3).
```

```
21  */
22
23  void showMenu(int *socketfd, clientConfig *config) {
24
25      int option;
26
27      do {
28          // Print the menu
29          printf(INTERFACE_MENU);
30
31          // Get the option from the user
32          if (scanf("%d", &option) != 1) {
33              printf("Invalid input. Please enter a number.\n");
34              fflush(stdin); // Clear the input buffer
35              continue;
36          }
37
38          // Process the option
39          switch (option) {
40              case 1:
41                  showPlayMenu(socketfd, config);
42                  break;
43              case 2:
44                  showStatisticsMenu(socketfd, config);
45                  showMenu(socketfd, config);
46                  break;
47              case 3:
48                  closeConnection(socketfd, config);
49              default:
50                  printf("Invalid option. Please try again.\n");
51          }
52      } while (option < 1 || option > 3);
53  }
54
55  /**
56   * Exibe o menu de jogo e processa as op   es selecionadas pelo
        utilizador.
57   *
```

```
58  * @param socketfd Um pointer para o descritor de socket usado para a
        comunica  o com o servidor.
59  * @param config A estrutura `clientConfig` que cont m as
        configura  es do cliente.
60  *
61  * @details A fun  o faz o seguinte:
62  * - Exibe o menu de op  es de jogo e solicita ao utilizador que
        escolha uma op  o.
63  * - Valida a entrada do utilizador para garantir que  um n mero.
64  * - Processa a op  o escolhida:
65  *   - Op  o 1: Chama a fun  o `showSinglePlayerMenu` para mostrar
        o menu de jogo single player.
66  *   - Op  o 2: Chama a fun  o `showMultiPlayerMenu` para mostrar o
        menu de jogo multiplayer.
67  *   - Op  o 3: Retorna ao menu principal chamando `showMenu`.
68  *   - Op  o 4: Fecha a conex o com o servidor e termina o programa.
69  * - Repete o loop at  que o utilizador escolha uma op  o v lida (1
        a 4).
70  */
71
72  void showPlayMenu(int *socketfd, clientConfig *config) {
73
74      int option;
75
76      do {
77          // print the play menu
78          printf(INTERFACE_PLAY_MENU);
79
80          // Get the option from the user
81          if (scanf("%d", &option) != 1) {
82              printf("Invalid input. Please enter a number.\n");
83              fflush(stdin); // Clear the input buffer
84              continue;
85          }
86
87          // switch the option
88          switch (option) {
89              case 1:
90                  showSinglePLayerMenu(socketfd, config);
```

```
91                  break;
92              case 2:
93                  showMultiPlayerMenu(socketfd, config);
94                  break;
95              case 3:
96                  showMenu(socketfd, config);
97                  break;
98              case 4:
99                  closeConnection(socketfd, config);
100                 break;
101             default:
102                 printf("Invalid option\n");
103                 break;
104         }
105     } while (option < 1 || option > 4);
106 }
107
108
109 /**
110  * Exibe o menu de jogo single player e processa as op   es
        selecionadas pelo utilizador.
111  *
112  * @param socketfd Um pointer para o descritor de socket usado para a
        comunica   o com o servidor.
113  * @param config A estrutura `clientConfig` que cont m as
        configura   es do cliente.
114  *
115  * @details A fun   o faz o seguinte:
116  * - Exibe o menu de sele   o de jogo single player e solicita ao
        utilizador que escolha uma op   o.
117  * - Valida a entrada do utilizador para garantir que    um n mero.
118  * - Processa a op   o escolhida:
119  *    - Op   o 1: Chama a fun   o `playRandomSinglePlayerGame` para
        jogar um jogo aleat rio.
120  *    - Op   o 2: Chama a fun   o `showGames` para mostrar os jogos
        dispon veis.
121  *    - Op   o 3: Retorna ao menu de jogo chamando `showPlayMenu`.
122  *    - Op   o 4: Fecha a conex o com o servidor e termina o programa.
```

```
123  * - Repete o loop at  que o utilizador escolha uma op  o v lida (1
         a 4).
124  */
125
126 void showSinglePLayerMenu(int *socketfd, clientConfig *config) {
127
128     int option;
129
130     do {
131         // ask for choosing a random game or a specific game
132         printf(INTERFACE_SELECT_SINGLEPLAYER_GAME);
133
134         // Get the option from the user
135         if (scanf("%d", &option) != 1) {
136             printf("Invalid input. Please enter a number.\n");
137             fflush(stdin); // Clear the input buffer
138             continue;
139         }
140
141         // switch the option
142         switch (option) {
143             case 1:
144                 // choose a random game
145                 playSinglePlayerGame(socketfd, config);
146                 break;
147             case 2:
148                 // choose a specific single player game
149                 showGames(socketfd, config, true);
150                 break;
151             case 3:
152                 showPlayMenu(socketfd, config);
153                 break;
154             case 4:
155                 closeConnection(socketfd, config);
156                 break;
157             default:
158                 printf("Invalid option\n");
159                 break;
160         }
```

```
161     } while (option < 1 || option > 4);
162 }
163
164
165 /**
166  * Joga um jogo aleat rio single player.
167  *
168  * @param socketfd Um pointer para o descritor de socket usado para a
        comunica  o com o servidor.
169  * @param config A estrutura ‘clientConfig‘ que cont m as
        configura  es do cliente.
170  *
171  * @details A fun  o faz o seguinte:
172  * - Envia um pedido ao servidor para um novo jogo aleat rio single
        player.
173  * - Recebe o tabuleiro do servidor.
174  * - Se o tabuleiro for "No rooms available", exibe uma mensagem e
        retorna.
175  * - Chama a fun  o ‘showBoard‘ para exibir o tabuleiro.
176  */
177
178 void showStatisticsMenu(int *socketfd, clientConfig *client) {
179     // Envia pedido de estat sticas ao servidor
180     const char *request = "GET_STATS";
181     if (send(*socketfd, request, strlen(request), 0) < 0) {
182         // erro ao enviar pedido de estat sticas
183         err_dump_client(client->logPath, 0, client->clientID, "can't
              send statistics request to server",
              EVENT_MESSAGE_CLIENT_NOT_SENT);
184     } else {
185         printf("Pedido de estat sticas enviado ao servidor\n");
186         char logMessage[256];
187         sprintf(logMessage, "%s: Pedido de estat sticas enviado ao
              servidor", EVENT_MESSAGE_CLIENT_SENT);
188         writeLogJSON(client->logPath, 0, client->clientID, logMessage);
189     }
190
191     // Recebe e exibe as estat sticas do servidor
192     char buffer[1024];
```

```
193     memset(buffer, 0, sizeof(buffer));
194
195     if (recv(*socketfd, buffer, sizeof(buffer), 0) < 0) {
196         // erro ao receber estat sticas
197         err_dump_client(client->logPath, 0, client->clientID, "can't
                receive statistics from server",
                EVENT_MESSAGE_CLIENT_NOT_RECEIVED);
198     } else {
199         printf("Estat sticas do servidor:\n%s\n", buffer);
200     }
201 }
202
203 /**
204  * Verifica se existem jogos dispon veis no servidor.
205  *
206  * @param socketfd Um pointer para o descritor de socket usado para a
        comunica  o com o servidor.
207  * @param config A estrutura 'clientConfig' que cont m as
        configura  es do cliente.
208  *
209  * @details A fun  o faz o seguinte:
210  * - Envia um pedido ao servidor para obter os jogos dispon veis.
211  * - Recebe a lista de jogos do servidor.
212  * - Exibe a lista de jogos.
213  */
214
215 /**
216  * Exibe o menu de jogo multiplayer e processa as op  es selecionadas
        pelo utilizador.
217  *
218  * @param socketfd Um pointer para o descritor de socket usado para a
        comunica  o com o servidor.
219  * @param config A estrutura 'clientConfig' que cont m as
        configura  es do cliente.
220  *
221  * @details A fun  o faz o seguinte:
222  * - Exibe o menu de op  es de jogo multiplayer e solicita ao
        utilizador que escolha uma op  o.
223  * - Valida a entrada do utilizador para garantir que  um n mero.
```

```
224  * - Processa a op   o escolhida:
225  *   - Op   o 1: Chama a fun   o `createNewMultiplayerGame` para
         criar um novo jogo multiplayer.
226  *   - Op   o 2: Chama a fun   o `showMultiplayerRooms` para mostrar
         as salas de jogo dispon veis.
227  *   - Op   o 3: Retorna ao menu de jogo chamando `showPlayMenu`.
228  *   - Op   o 4: Fecha a conex o com o servidor e termina o programa.
229  * - Repete o loop at   que o utilizador escolha uma op   o v lida (1
         a 4).
230  */
231
232 void showMultiPlayerMenu(int *socketfd, clientConfig *config) {
233
234     int option;
235
236     do {
237         // ask for creating a room or joining a room
238         printf(INTERFACE_SELECT_MULTIPLAYER_MENU);
239
240         // Get the option from the user
241         if (scanf("%d", &option) != 1) {
242             printf("Invalid input. Please enter a number.\n");
243             fflush(stdin); // Clear the input buffer
244             continue;
245         }
246
247         // switch the option
248         switch (option) {
249             case 1:
250                 // create a room
251                 createNewMultiplayerGame(socketfd, config);
252                 break;
253             case 2:
254                 // join a room
255                 showMultiplayerRooms(socketfd, config);
256                 break;
257             case 3:
258                 showPlayMenu(socketfd, config);
259                 break;
```

```
260             case 4:
261                 closeConnection(socketfd, config);
262                 break;
263             default:
264                 printf("Invalid option\n");
265                 break;
266         }
267     } while (option < 1 || option > 4);
268 }
269
270 void createNewMultiplayerGame(int *socketfd, clientConfig *config) {
271     // create a new multiplayer game
272
273     int option;
274
275     do {
276         // ask for creating a room or joining a room
277         printf(INTERFACE_SELECT_MULTIPLAYER_GAME);
278
279         // Get the option from the user
280         if (scanf("%d", &option) != 1) {
281             printf("Invalid input. Please enter a number.\n");
282             fflush(stdin); // Clear the input buffer
283             continue;
284         }
285
286         // switch the option
287         switch (option) {
288             case 1:
289                 // create a new random multiplayer game
290                 showPossibleSynchronizations(socketfd, config);
291                 //playRandomMultiPlayerGame(socketfd, config);
292                 break;
293             case 2:
294                 // show new specific multiplayer game
295                 showGames(socketfd, config, false);
296                 break;
297             case 3:
298                 showMultiPlayerMenu(socketfd, config);
```

```
299                 break;
300             case 4:
301                 closeConnection(socketfd, config);
302                 break;
303             default:
304                 printf("Invalid option\n");
305                 break;
306         }
307     } while (option < 1 || option > 4);
308 }
309
310
311 /**
312  * Solicita ao servidor um jogo multiplayer aleat rio e exibe o
        tabuleiro recebido.
313  *
314  * @param socketfd Um pointer para o descritor de socket usado para a
        comunica  o com o servidor.
315  * @param config A estrutura ‘clientConfig‘ que cont m as
        configura  es do cliente,
316  * incluindo o caminho do log e o ID do cliente.
317  *
318  * @details A fun  o faz o seguinte:
319  * - Envia um pedido ao servidor para iniciar um novo jogo multiplayer
        aleat rio.
320  * - Se o envio do pedido falhar, regista o erro no log e termina.
321  * - Se o pedido for bem-sucedido, aguarda a rece  o do tabuleiro do
        jogo:
322  *   - Se n o houver salas dispon veis, informa o utilizador.
323  *   - Se o tabuleiro for recebido corretamente, chama ‘showBoard‘ para
         exibir o tabuleiro.
324  * - Regista quaisquer erros de comunica  o (envio ou rece  o) no
        ficheiro de log.
325  */
326
327 void showPossibleSynchronizations(int *socketfd, clientConfig *config)
        {
328
329     int option;
```

```
330
331     do {
332         // ask for creating a room or joining a room
333         printf(INTERFACE_POSSIBLE_SYNCHRONIZATION);
334
335         // Get the option from the user
336         if (scanf("%d", &option) != 1) {
337             printf("Invalid input. Please enter a number.\n");
338             fflush(stdin); // Clear the input buffer
339             continue;
340         }
341
342         // switch the option
343         switch (option) {
344             case 1:
345                 // create a new random multiplayer game with readers-
                        writers synchronization
346                 playMultiPlayerGame(socketfd, config, "readersWriters")
                        ;
347                 writeLogJSON(config->logPath, 0, config->clientID, "
                        Started multiplayer game with readers-writers
                        synchronization");
348
349                 break;
350             case 2:
351                 // create a new random multiplayer game with barber
                        shop synchronization with priority queues
352                 playMultiPlayerGame(socketfd, config, "
                        barberShopStaticPriority");
353                 writeLogJSON(config->logPath, 0, config->clientID, "
                        Started multiplayer game with
                        barberShopStaticPriority synchronization");
354                 break;
355             case 3:
356                 // create a new random multiplayer game with barber
                        shop synchronization with dynamic priority queues
357                 playMultiPlayerGame(socketfd, config, "
                        barberShopDynamicPriority");
```

```
358                 writeLogJSON(config->logPath, 0, config->clientID, "
                        Started multiplayer game with
                        barberShopDynamicPriority synchronization");
359                 break;
360             case 4:
361                 // create a new random multiplayer game with barber
                        shop synchronization with a FIFO queue
362                 playMultiPlayerGame(socketfd, config, "barberShopFIFO")
                        ;
363                 writeLogJSON(config->logPath, 0, config->clientID, "
                        Started multiplayer game with barberShopFIFO
                        synchronization");
364
365                 break;
366             case 5:
367
368                 // send 0 to the server
369                 if (send(*socketfd, "0", strlen("0"), 0) < 0) {
370                     err_dump_client(config->logPath, 0, config->
                            clientID, "can't send return to menu to server",
                             EVENT_MESSAGE_CLIENT_NOT_SENT);
371                 } else {
372                     char logMessage[256];
373                     snprintf(logMessage, sizeof(logMessage), "%s: sent
                            0 to return to multiplayer menu",
                            EVENT_MESSAGE_CLIENT_SENT);
374                     writeLogJSON(config->logPath, 0, config->clientID,
                            logMessage);
375                 }
376
377                 // back to the multiplayer menu
378                 createNewMultiplayerGame(socketfd, config);
379                 writeLogJSON(config->logPath, 0, config->clientID, "
                        Returned to multiplayer menu");
380                 break;
381             case 6:
382                 // close the connection
383                 closeConnection(socketfd, config);
384                 break;
```

```
385             default:
386                 printf("Invalid option\n");
387                 break;
388         }
389     } while (option < 1 || option > 6);
390
391 }
392
393 /**
394  * Solicita ao servidor a lista de salas multiplayer existentes e
        permite ao utilizador
395  * escolher uma sala ou voltar atr s.
396  *
397  * @param socketfd Um pointer para o descritor de socket usado para a
        comunica  o com o servidor.
398  * @param config A estrutura ‘clientConfig‘ que cont m as
        configura  es do cliente,
399  * incluindo o caminho do log e o ID do cliente.
400  *
401  * @details A fun  o faz o seguinte:
402  * - Envia um pedido ao servidor para obter a lista de salas
        multiplayer dispon veis.
403  * - Se o pedido falhar, regista o erro no log e termina.
404  * - Se o pedido for bem-sucedido, recebe a lista de salas do servidor
        e exibe-a.
405  * - Permite ao utilizador escolher uma sala pelo ID ou voltar ao menu
        anterior:
406  *   - Se o utilizador escolher 0, envia o pedido de retorno ao
        servidor e exibe o menu multiplayer.
407  *   - Se for escolhido um ID de sala, envia o ID ao servidor e aguarda
        o tabuleiro da sala.
408  * - Regista todos os erros de comunica  o e eventos importantes no
        ficheiro de log.
409  */
410
411 void showMultiplayerRooms(int *socketfd, clientConfig *config) {
412     // ask server for existing rooms
413     if (send(*socketfd, "existingRooms", strlen("existingRooms"), 0) <
        0) {
```

```
414             err_dump_client(config->logPath, 0, config->clientID, "can't
                    send existing rooms request to server",
                    EVENT_MESSAGE_CLIENT_NOT_SENT);
415
416         } else {
417             printf("Requesting existing multiplayer rooms...\n");
418             writeLogJSON(config->logPath, 0, config->clientID, "Sent
                    existing rooms request to server");
419
420             // receive the rooms from the server
421             char buffer[BUFFER_SIZE];
422             memset(buffer, 0, sizeof(buffer));
423
424             if (recv(*socketfd, buffer, sizeof(buffer), 0) < 0) {
425
426                 // error receiving rooms from server
427                 err_dump_client(config->logPath, 0, config->clientID, "can'
                        t receive rooms from server",
                        EVENT_MESSAGE_CLIENT_NOT_RECEIVED);
428
429             } else {
430
431                 // show the rooms
432                 printf("Existing rooms:\n%s\n", buffer);
433                 printf("0 - Back\n");
434                 writeLogJSON(config->logPath, 0, config->clientID, "
                        Received existing rooms from server");
435
436                 // ask for the game ID
437                 int roomID;
438                 printf("Choose an option: ");
439
440                 // Get the game ID from the user
441                 if (scanf("%d", &roomID) != 1) {
442                     printf("Invalid input. Please enter a number.\n");
443                     fflush(stdin); // Clear the input buffer
444                     return;
445                 }
446
```

```
447             if (roomID == 0) {
448
449                 // send 0 to the server
450                 if (send(*socketfd, "0", strlen("0"), 0) < 0) {
451                     err_dump_client(config->logPath, 0, config->
                            clientID, "can't send return to menu to server",
                             EVENT_MESSAGE_CLIENT_NOT_SENT);
452                 } else {
453                     char logMessage[256];
454                     snprintf(logMessage, sizeof(logMessage), "%s: sent
                            0 to return to multiplayer menu",
                            EVENT_MESSAGE_CLIENT_SENT);
455                     writeLogJSON(config->logPath, 0, config->clientID,
                            logMessage);
456                 }
457
458                 // show the multiplayer menu
459                 showMultiPlayerMenu(socketfd, config);
460
461
462             } else {
463
464                 // send the room ID to the server
465                 char roomIDString[10];
466                 sprintf(roomIDString, "%d", roomID);
467
468                 if (send(*socketfd, roomIDString, strlen(roomIDString),
                        0) < 0) {
469                     err_dump_client(config->logPath, 0, config->
                            clientID, "can't send room ID to server",
                            EVENT_MESSAGE_CLIENT_NOT_SENT);
470                 } else {
471                     printf("Requesting room with ID %s...\n",
                            roomIDString);
472                 }
473
474                 //printf("NOW RECEIVING TIMER\n");
475                 // receive timer from server
476                 receiveTimer(socketfd, config);
```

```
477             }
478         }
479     }
480 }
481
482 /**
483  * Solicita ao servidor a lista de jogos existentes (single player ou
        multiplayer) e
484  * permite ao utilizador escolher um jogo ou voltar ao menu.
485  *
486  * @param socketfd Um pointer para o descritor de socket usado para a
        comunica  o com o servidor.
487  * @param config A estrutura ‘clientConfig‘ que cont m as
        configura  es do cliente,
488  * incluindo o caminho do log e o ID do cliente.
489  * @param isSinglePlayer Um valor booleano que indica se o utilizador
        quer jogos
490  * single player (‘true‘) ou multiplayer (‘false‘).
491  *
492  * @details A fun  o realiza as seguintes opera  es:
493  * - Envia um pedido ao servidor para obter a lista de jogos
        dispon veis,
494  * com base no tipo de jogo (single player ou multiplayer).
495  * - Se o pedido falhar, regista o erro no log e termina.
496  * - Se o pedido for bem-sucedido, recebe e exibe a lista de jogos.
497  * - Permite ao utilizador escolher um jogo pelo ID ou voltar ao menu
        anterior:
498  *   - Se o utilizador escolher 0, envia a escolha ao servidor e exibe
        o menu apropriado (single player ou multiplayer).
499  *   - Se for escolhido um ID de jogo, envia o ID ao servidor, recebe o
        tabuleiro e chama ‘showBoard‘ para o exibir.
500  * - Regista quaisquer erros de comunica  o e eventos importantes no
        ficheiro de log.
501  */
502 void showGames(int *socketfd, clientConfig *config, bool isSinglePlayer
        ) {
503
504     // message to send to the server
505     char message[256];
```

```
506     if (isSinglePlayer) {
507         snprintf(message, sizeof(message), "selectSinglePlayerGames");
508     } else {
509         snprintf(message, sizeof(message), "selectMultiPlayerGames");
510     }
511 
512     // ask server for existing games
513     if (send(*socketfd, message, strlen(message), 0) < 0) {
514         err_dump_client(config->logPath, 0, config->clientID, "can't
            send existing games request to server",
            EVENT_MESSAGE_CLIENT_NOT_SENT);
515     } else {
516         writeLogJSON(config->logPath, 0, config->clientID, "Requested
            existing games from server");
517         printf("Requesting existing games...\n");
518 
519         // receive the games from the server
520         char buffer[BUFFER_SIZE];
521         memset(buffer, 0, sizeof(buffer));
522 
523         if (recv(*socketfd, buffer, sizeof(buffer), 0) < 0) {
524 
525             // error receiving games from server
526             err_dump_client(config->logPath, 0, config->clientID, "can'
                t receive games from server",
                EVENT_MESSAGE_CLIENT_NOT_RECEIVED);
527 
528         } else {
529 
530             // show the games
531             printf("Existing games:\n%s", buffer);
532             printf("0 - Back\n");
533             writeLogJSON(config->logPath, 0, config->clientID, "
                Received and displayed existing games");
534             // ask for the game ID
535             int gameID;
536             printf("Choose an option: ");
537 
538             // Get the game ID from the user
```

```
539             if (scanf("%d", &gameID) != 1) {
540                 printf("Invalid input. Please enter a number.\n");
541                 fflush(stdin); // Clear the input buffer
542                 return;
543             }
544 
545             if (gameID == 0) {
546 
547                 // send 0 to the server
548                 if (send(*socketfd, "0", strlen("0"), 0) < 0) {
549                     err_dump_client(config->logPath, 0, config->
                            clientID, "can't send return to menu to server",
                             EVENT_MESSAGE_CLIENT_NOT_SENT);
550                 } else {
551                     char logMessage[256];
552                     snprintf(logMessage, sizeof(logMessage), "%s: sent
                            0 to return to menu", EVENT_MESSAGE_CLIENT_SENT)
                            ;
553                     writeLogJSON(config->logPath, 0, config->clientID,
                            logMessage);
554                 }
555 
556                 // show the single player menu
557                 if (isSinglePlayer) {
558                     showSinglePLayerMenu(socketfd, config);
559                 } else {
560                     showMultiPlayerMenu(socketfd, config);
561                 }
562 
563             } else {
564                 // send the game ID to the server
565                 char gameIDString[10];
566                 sprintf(gameIDString, "%d", gameID);
567 
568                 if (send(*socketfd, gameIDString, strlen(gameIDString),
                        0) < 0) {
569                     err_dump_client(config->logPath, 0, config->
                            clientID, "can't send game ID to server",
                            EVENT_MESSAGE_CLIENT_NOT_SENT);
```

```
570                 } else {
571                     printf("Requesting game with ID %s...\n",
                            gameIDString);
572                 }
573 
574                 if (!isSinglePlayer) {
575                     // now need to choose synchronization
576                     showPossibleSynchronizations(socketfd, config);
577                     writeLogJSON(config->logPath, 0, config->clientID,
                            "Selected synchronization type for multiplayer
                            game");
578                 }
579             }
580         }
581     }
582 }
583 
584 void receiveTimer(int *socketfd, clientConfig *config) {
585 
586     char buffer[BUFFER_SIZE];
587     memset(buffer, 0, sizeof(buffer));
588 
589     // receive at 60, 50, 40, 30, 20, 10, 5, 4, 3, 2, 1 seconds
590     int timeLeft = 60;
591 
592     bool isRoomFull = false;
593 
594     //printf("IN RECEIVING TIMER\n");
595 
596     while (timeLeft > 0) {
597         if (recv(*socketfd, buffer, sizeof(buffer), 0) < 0) {
598 
599             // error receiving timer from server
600             err_dump_client(config->logPath, 0, config->clientID, "can'
                    t receive timer from server",
                    EVENT_MESSAGE_CLIENT_NOT_RECEIVED);
601 
602         } else {
603 
```

```
604             //printf("Buffer: %s\n", buffer);
605
606             // check if buffer is "Room is full"
607             if (strcmp(buffer, "Room is full") == 0) {
608
609                 isRoomFull = true;
610                 char logMessage[256];
611                 snprintf(logMessage, sizeof(logMessage), "%s: room is
                        full", EVENT_MESSAGE_CLIENT_RECEIVED);
612                 writeLogJSON(config->logPath, 0, config->clientID,
                        logMessage);
613                 break;
614             }
615
616             timeLeft = showTimerUpdate(buffer, timeLeft);
617
618             char logMessage[256];
619             snprintf(logMessage, sizeof(logMessage), "%s: time left: %d
                    ", EVENT_MESSAGE_CLIENT_RECEIVED, timeLeft);
620             writeLogJSON(config->logPath, 0, config->clientID,
                    logMessage);
621         }
622     }
623
624     if (isRoomFull) {
625         //debug
626         writeLogJSON(config->logPath, 0, config->clientID, "Room is
                full. Returning to multiplayer menu.");
627         printf("Room is full\n");
628         // show the multiplayer menu
629         showMultiPlayerMenu(socketfd, config);
630     }
631 }
632
633 void playSinglePlayerGame(int *socketfd, clientConfig *config) {
634
635     // ask server for a random game
636     if (send(*socketfd, "newSinglePlayerGame", strlen("
            newSinglePlayerGame"), 0) < 0) {
```

```
637         err_dump_client(config->logPath, 0, config->clientID, "can't
                send game request to server", EVENT_MESSAGE_CLIENT_NOT_SENT)
                ;
638     } else {
639         printf("Requesting a new game...\n");
640     }
641 }
642
643 void playMultiPlayerGame(int *socketfd, clientConfig *config, char *
        synchronization) {
644
645     // buffer
646     char buffer[BUFFER_SIZE];
647     memset(buffer, 0, sizeof(buffer));
648
649     // check for synchronization
650     if (strcmp(synchronization, "readersWriters") == 0) {
651
652         // buffer for readersWriters
653         strcpy(buffer, "newMultiPlayerGameReadersWriters");
654      // Log evento de solicita  o
655         writeLogJSON(config->logPath, 0, config->clientID, "Requesting
                multiplayer game with readers-writers synchronization");
656     } else if (strcmp(synchronization, "barberShopStaticPriority") ==
            0) {
657
658         // buffer for barberShopPriority
659         strcpy(buffer, "newMultiPlayerGameBarberShopStaticPriority");
660          // Log evento de solicita  o
661         writeLogJSON(config->logPath, 0, config->clientID, "Requesting
                multiplayer game with barber shop static priority
                synchronization");
662
663     } else if (strcmp(synchronization, "barberShopDynamicPriority") ==
            0) {
664
665         // buffer for barberShopDynamicPriority
666         strcpy(buffer, "newMultiPlayerGameBarberShopDynamicPriority");
667          // Log evento de solicita  o
```

```
668         writeLogJSON(config->logPath, 0, config->clientID, "Requesting
                multiplayer game with barber shop dynamic priority
                synchronization");
669
670     } else if (strcmp(synchronization, "barberShopFIFO") == 0) {
671
672         // buffer for barberShopFIFO
673         strcpy(buffer, "newMultiPlayerGameBarberShopFIFO");
674         writeLogJSON(config->logPath, 0, config->clientID, "Requesting
                multiplayer game with barber shop FIFO synchronization");
675
676     } else {
677         printf("Invalid synchronization option\n");
678         writeLogJSON(config->logPath, 1, config->clientID, "Invalid
                synchronization option requested");
679
680         return;
681     }
682
683     // ask server for a
684     if (send(*socketfd, buffer, BUFFER_SIZE, 0) < 0) {
685         err_dump_client(config->logPath, 0, config->clientID, "can't
                send multiplayer game request to server",
                EVENT_MESSAGE_CLIENT_NOT_SENT);
686     } else {
687         printf("Requesting a new multiplayer game...\n");
688
689         // receive timer from server
690         receiveTimer(socketfd, config);
691
692         printf("JOGO INICIADO\n");
693     }
694 }
695
696 int showTimerUpdate(char *buffer, int timeLeft) {
697
698     //printf("Recebido: %s\n", buffer);
699     // Parse the received message
700     strtok(buffer, "\n");
```

```
701     timeLeft = atoi(strtok(NULL, "\n"));
702     //printf("Tempo restante: %d segundos\n", timeLeft);
703     int roomId = atoi(strtok(NULL, "\n"));
704     //printf("ID da sala: %d\n", roomId);
705     int gameId = atoi(strtok(NULL, "\n"));
706     //printf("ID do jogo: %d\n", gameId);
707     int numPlayers = atoi(strtok(NULL, "\n"));
708     //printf("Jogadores na sala: %d\n", numPlayers);
709 
710     // show the timer update
711     printf("Time left: %d seconds - Room ID: %d - Game ID: %d - Players
            joined: %d\n", timeLeft, roomId, gameId, numPlayers);
712 
713     return --timeLeft;
714 }
```

A.5 Utils

A.5.1 logs/logs-common.h

```
1  #ifndef LOGS_COMMON_H
2  #define LOGS_COMMON_H
3  
4  /* server */
5  #define EVENT_SERVER_START                    "Server inicializado"
6  #define EVENT_GAME_LOAD                       "Jogo carregado"
7  #define EVENT_GAME_NOT_LOAD                   "Jogo nao carregado"
8  #define EVENT_GAME_NOT_FOUND                  "Jogo nao encontrado"
9  #define EVENT_BOARD_SHOW                      "Tabuleiro do jogo
       mostrado"
10 #define EVENT_SOLUTION_SENT                   "Solucao recebida"
11 #define EVENT_SOLUTION_CORRECT                "Solucao correta"
12 #define EVENT_SOLUTION_INCORRECT              "Solucao errada"
13 #define EVENT_GAME_OVER                       "Jogo terminado"
14 #define EVENT_MESSAGE_SERVER_SENT             "Mensagem enviada para
       o cliente"
15 #define EVENT_MESSAGE_SERVER_NOT_SENT         "Erro ao enviar
       mensagem para o cliente"
```

```
16 #define EVENT_MESSAGE_SERVER_RECEIVED           "Mensagem recebida do
       cliente"
17 #define EVENT_MESSAGE_SERVER_NOT_RECEIVED       "Erro ao receber
       mensagem do cliente"
18 #define EVENT_CONNECTION_SERVER_ERROR           "Erro na conexao do
       servidor"
19 #define EVENT_CONNECTION_SERVER_ESTABLISHED     "Conexao estabelecida
       com o cliente"
20 #define EVENT_SERVER_THREAD_ERROR               "Erro ao criar thread"
21 #define EVENT_SERVER_CONNECTION_FINISH          "Conexao terminada com
       o cliente"
22 #define EVENT_SERVER_GAMES_SENT                 "Jogos enviados para o
       cliente"
23 #define EVENT_ROOM_NOT_CREATED                        "Sala nao
       criada"
24 #define EVENT_ROOM_LOAD                         "Sala carregada"
25 #define EVENT_ROOM_NOT_LOAD                     "Sala nao carregada"
26 #define EVENT_ROOM_JOIN                         "Jogador entrou na sala
       "
27 #define EVENT_ROOM_NOT_JOIN                     "Jogador nao entrou na
       sala"
28 #define EVENT_ROOM_DELETE                       "Sala eliminada"
29 #define EVENT_ROOM_NOT_DELETE                   "Sala nao eliminada"
30 #define EVENT_NEW_RECORD                        "Novo recorde"
31 #define EVENT_THREAD_NOT_CREATE                  "Erro ao criar a
       thread"
32 #define EVENT_BARBER_CREATED                    "Barbeiro criado"
33
34 /* client */
35 #define EVENT_CONNECTION_CLIENT_ESTABLISHED     "Conexao estabelecida
       com o servidor"
36 #define EVENT_CONNECTION_CLIENT_NOT_ESTABLISHED "Erro ao estabelecer
       conexao com o servidor"
37 #define EVENT_CONNECTION_CLIENT_CLOSED          "Conexao com o servidor
        fechada"
38 #define EVENT_MESSAGE_CLIENT_SENT               "Mensagem enviada para
       o servidor"
39 #define EVENT_MESSAGE_CLIENT_NOT_SENT           "Erro ao enviar
       mensagem para o servidor"
```

```
40 #define EVENT_MESSAGE_CLIENT_RECEIVED           "Mensagem recebida do
       servidor"
41 #define EVENT_MESSAGE_CLIENT_NOT_RECEIVED       "Erro ao receber
       mensagem do servidor"
42 #define EVENT_GAME_STARTED                      "Game started"
43
44
45 #define MEMORY_ERROR                            "Erro de memoria"
46
47 #define BUFFER_SIZE 1024
48
49 struct log {
50     int id;
51     char event[32];
52     char log[256];
53 };
54
55 // wirte log in JSON format
56 void writeLogJSON(const char *filename, int gameID, int playerID, const
       char *logMessage);
57
58 // concatenate info
59 char *concatenateInfo(char *msg, char* event, int idJogo, int idJogador
       );
60
61 #endif // LOGS_COMMON_H
```

A.5.2 logs/logs-common.c

```
1  #include <stdio.h>  // Usar FILE, fopen(), fclose()
2  #include <stdlib.h>
3  #include <time.h>  // Usar time_t, time(), ctime()
4  #include <string.h>
5  #include "../parson/parson.h"
6  #include "logs-common.h"
7
8  /**
9   * Escreve uma mensagem de log num ficheiro JSON, incluindo o ID do
       jogo,
```

```
10  * o ID do jogador e a data/hora em que o evento ocorreu.
11  *
12  * @param filename O caminho do ficheiro JSON onde o log ser  escrito.
13  * @param gameID O identificador do jogo associado ao log.
14  * @param playerID O identificador do jogador associado ao log.
15  * @param logMessage A mensagem de log a ser registada.
16  */
17
18 void writeLogJSON(const char *filename, int gameID, int playerID, const
       char *logMessage) {
19     // Abrir o ficheiro JSON existente
20     JSON_Value *rootValue = json_parse_file(filename);
21     JSON_Object *rootObject = NULL;
22     JSON_Value *logsArrayValue = NULL;
23     JSON_Array *logsArray = NULL;
24
25     if (rootValue == NULL) {
26         // Se o ficheiro nao existe ou esta vazio, cria uma nova
              estrutura JSON
27         rootValue = json_value_init_object();
28         rootObject = json_value_get_object(rootValue);
29         // Criar um array vazio para os logs
30         logsArrayValue = json_value_init_array();
31         // Adicionar o array ao objeto raiz
32         json_object_set_value(rootObject, "logs", logsArrayValue);
33
34     } else {
35         // Se o ficheiro JSON ja existe, carregar os dados existentes
36         rootObject = json_value_get_object(rootValue);
37         logsArrayValue = json_object_get_value(rootObject, "logs");
38     }
39      // Obter o array de logs
40     logsArray = json_value_get_array(logsArrayValue);
41
42     // Obter a data e hora atual
43     time_t t = time(NULL);
44     struct tm tm = *localtime(&t);
45     char timestamp[72];
```

```
46      // Aqui alterei o formato da data para nao ser necessario dar
            escape da / no ficheiro JSON
47      sprintf(timestamp, "%02d-%02d-%04d %02d:%02d:%02d",
48              tm.tm_mday, tm.tm_mon + 1, tm.tm_year + 1900,
49              tm.tm_hour, tm.tm_min, tm.tm_sec);
50
51      // Cria um novo ficheiro JSON para o log mesmo que nao tenha sido
            criado antes
52      JSON_Value *logValue = json_value_init_object();
53      JSON_Object *logObject = json_value_get_object(logValue);
54
55      // Preencher os campos do log
56      json_object_set_string(logObject, "timestamp", timestamp);
57      json_object_set_number(logObject, "gameID", gameID);
58      json_object_set_number(logObject, "playerID", playerID);
59      json_object_set_string(logObject, "message", logMessage);
60
61      // Adicionar o novo log ao array de logs
62      json_array_append_value(logsArray, logValue);
63
64      // Gravar o ficheiro JSON atualizado
65      if (json_serialize_to_file_pretty(rootValue, filename) !=
            JSONSuccess) {
66          printf("Erro ao gravar o ficheiro JSON: %s\n", filename);
67      }
68
69      // Limpar a mem ria JSON
70      json_value_free(rootValue);
71  }
72
73  /**
74   * Regista uma mensagem de erro no log, imprime a mensagem de erro no
        stderr,
75   * e termina o programa com um c digo de erro.
76   *
77   * @param logPath O caminho do ficheiro de log onde a mensagem de erro
        ser  escrita.
78   * @param idJogo O identificador do jogo associado ao erro.
79   * @param idJogador O identificador do jogador associado ao erro.
```

```
 * @param msg A mensagem de erro a ser registada e impressa.
 * @param event O evento espec fico associado    mensagem de erro.
 */

char* concatenateInfo(char *msg, char* event, int idJogo, int idJogador
    ) {
    char* logMessage = (char*)malloc(BUFFER_SIZE);
    if (logMessage == NULL) {
        // Handle memory allocation failure
        fprintf(stderr, "Memory allocation failed in concatenateInfo\n"
            );
        return NULL;
    }
    memset(logMessage, 0, BUFFER_SIZE);

    // add event to msg
    snprintf(logMessage, BUFFER_SIZE, "%s: %s", event, msg);

    // add other info to msg
    snprintf(logMessage, BUFFER_SIZE, "%d\n%d\n%s\n", idJogo, idJogador
        , msg);

    return logMessage;
}
```

A.5.3 network/network.h

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>

// Declara  es de vari veis externas usadas para opera  es de I/O.
```

```
13 extern int fd;
14 extern char *ptr;
15 extern int nbytes;
16
17 // Fun   o externa para ler nbytes de um descritor de ficheiro.
18 extern int readn(int fd, char *ptr, int nbytes);
19
20 // Fun   o externa para escrever nbytes num descritor de ficheiro.
21 extern int writen(int fd, char *ptr, int nbytes);
22
23 // Fun   o externa para ler uma linha de um descritor de ficheiro.
24 extern int readline(int fd, char *ptr, int maxlen);
25
26 // Fun   o externa para gerir a comunica   o cliente-servidor.
27 extern void str_cli(FILE *fp, int sockfd);
28
29 // Fun   o externa para ecoar dados recebidos de um cliente.
30 extern void str_echo(int sockfd);
```

A.5.4 network/network.c

```
1  #include <stdio.h>
2  #include <errno.h>
3  #include "../logs/logs-common.h"
4  #include "network.h"
5
6  /* Fun   es utilit rias retiradas de "UNIX Networking Programming" */
7
8
9  /**
10  * L   exatamente nbytes de um ficheiro/socket.
11  *
12  * @param fd O descritor de ficheiro (socket ou ficheiro) de onde ser
        feita a leitura.
13  * @param ptr Um pointer para o buffer onde os dados lidos ser o
       armazenados.
14  * @param nbytes O n mero de bytes a serem lidos.
15  * @return O n mero de bytes lidos ou um valor negativo em caso de
       erro.
```

```
16  */
17
18  int readn(int fd, char *ptr, int nbytes)
19  {
20      int nleft, nread;
21
22      nleft = nbytes;
23      while (nleft > 0)
24      {
25          nread = read(fd, ptr, nleft);
26          if (nread < 0)
27              return (nread);
28          else if (nread == 0)
29              break;
30
31          nleft -= nread;
32          ptr += nread;
33      }
34      return (nbytes - nleft);
35  }
36
37
38  /**
39   * Escreve exatamente nbytes num ficheiro/socket.
40   *
41   * @param fd O descritor de ficheiro (socket ou ficheiro) onde ser
         feita a escrita.
42   * @param ptr Um pointer para o buffer que cont m os dados a serem
         escritos.
43   * @param nbytes O n mero de bytes a serem escritos.
44   * @return O n mero de bytes escritos ou um valor negativo em caso de
         erro.
45   */
46
47  int writen(int fd, char *ptr, int nbytes)
48  {
49      int nleft, nwritten;
50
51      nleft = nbytes;
```

```
52      while (nleft > 0)
53      {
54          nwritten = write(fd, ptr, nleft);
55          if (nwritten <= 0)
56              return (nwritten);
57
58          nleft -= nwritten;
59          ptr += nwritten;
60      }
61      return (nbytes - nleft);
62  }
63
64
65  /**
66   * L   uma linha de um ficheiro/socket (at   encontrar '\n', atingir
        maxlen ou ocorrer um erro).
67   *
68   * @param fd O descritor de ficheiro (socket ou ficheiro) de onde ser
         feita a leitura.
69   * @param ptr Um pointer para o buffer onde a linha lida ser
        armazenada.
70   * @param maxlen O n mero m ximo de caracteres a serem lidos,
        incluindo o terminador '\0'.
71   * @return O n mero de caracteres lidos, incluindo '\n', ou -1 em caso
         de erro. Retorna 0
72   * se atingir o fim do ficheiro antes de ler qualquer car cter.
73   */
74
75  int readline(int fd, char *ptr, int maxlen)
76  {
77      int n, rc;
78      char c;
79
80      for (n = 1; n < maxlen; n++)
81      {
82          if ((rc = read(fd, &c, 1)) == 1)
83          {
84              *ptr++ = c;
85              if (c == '\n')
```

```
86                  break;
87          }
88          else if (rc == 0)
89          {
90              if (n == 1)
91                  return (0);
92              else
93                  break;
94          }
95          else
96              return (-1);
97      }
98
99      /* N o esquecer de terminar a string */
100     *ptr = 0;
101
102     /* Note-se que n foi incrementado de modo a contar
103        com o \n ou \0 */
104     return (n);
105 }
```

A.5.5 queues/queues.h

```
1  #ifndef QUEUES_H
2  #define QUEUES_H
3
4  #include <stdbool.h>
5  #include <semaphore.h>
6  #include <pthread.h>
7
8  typedef struct Node {
9      int clientID;
10     int timeInQueue;
11     bool isPremium;
12     struct Node* next;
13 } Node;
14
15 // Define a structure for the queue
16 typedef struct PriorityQueue {
```

```
17      Node* front;
18      Node* rear;
19      pthread_mutex_t mutex;
20      sem_t empty;
21      sem_t full;
22  } PriorityQueue;
23
24  // create a new node
25  Node *createNode(int clientID, bool isPremium);
26
27  // initialize the queue
28  void initPriorityQueue(PriorityQueue *queue, int queueSize);
29
30  // enqueue an element
31  void enqueueWithPriority(PriorityQueue *queue, int clientID, bool
        isPremium);
32
33  // update the priority of the elements in the queue
34  void updatePriority(PriorityQueue *queue);
35
36  // update the queue with the priority
37  void updateQueueWithPriority(PriorityQueue *queue, int maxWaitingTime);
38
39  // enqueue an element in a FIFO way
40  void enqueueFifo(PriorityQueue *queue, int clientID);
41
42  // dequeue an element
43  int dequeue(PriorityQueue *queue);
44
45  // free the queue
46  void freePriorityQueue(PriorityQueue *queue);
47
48  #endif // QUEUES_H
```

A.5.6 queues/queues.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include "queues.h"
```

```
4
5  Node *createNode(int clientID, bool isPremium) {
6      Node *newNode = (Node *)malloc(sizeof(Node));
7      if (newNode == NULL) {
8          fprintf(stderr, "Memory allocation failed\n");
9          return NULL;
10     }
11     newNode->clientID = clientID;
12     newNode->isPremium = isPremium;
13     newNode->next = NULL;
14     newNode->timeInQueue = 0;
15     return newNode;
16 }
17
18 void initPriorityQueue(PriorityQueue *queue, int queueSize) {
19     queue->front = NULL;
20     queue->rear = NULL;
21     pthread_mutex_init(&queue->mutex, NULL);
22     sem_init(&queue->empty, 0, queueSize);
23     sem_init(&queue->full, 0, 0);
24 }
25
26 void enqueueWithPriority(PriorityQueue *queue, int clientID, bool
       isPremium) {
27     //printf("CLIENT %d WANT TO JOIN THE ROOM\n", clientID);
28
29     Node *newNode = createNode(clientID, isPremium); // create a new
           node
30     //printf("CREATING A NODE FOR CLIENt %d WHICH IS %s\n", clientID,
           isPremium ? "PREMIUM" : "NOT PREMIUM");
31
32     sem_wait(&queue->empty); // wait for empty space (fica a espera que
           haja espa o na fila para adicionar um novo cliente)
33
34     //printf("CLIENT %d JOINING THE QUEUE\n", clientID);
35
36     pthread_mutex_lock(&queue->mutex); // lock the queue (garante a
           exclus o m tua se dois clientes tentarem aceder
           simultaneamente    fila)
```

```
37
38      if (queue->front == NULL) { // if the queue is empty
39          //printf("QUEUE IS EMPTY: ADDING CLIENT %d TO THE FIRST NODE\n
                ", clientID);
40          queue->front = newNode;
41          queue->rear = newNode;
42      } else {                    // if the queue is not empty
43
44          //print order of the queue
45          //Node *temp1 = queue->front;
46          //printf("--------------------\n");
47          //printf("QUEUE ORDER BEFORE ENQUEING:\n");
48          //while (temp1 != NULL) {
49          //    printf("Client %d isPremium: %s\n", temp1->clientID,
                temp1->isPremium ? "true" : "false");
50          //    temp1 = temp1->next;
51          //}
52          //printf("--------------------\n");
53          // order by isPremium
54          Node *temp = queue->front;
55          Node *prev = NULL;
56
57          // find last premium
58          while (temp != NULL && temp->isPremium) {
59              prev = temp;
60              temp = temp->next;
61          }
62
63          // if not premium, find last non-premium
64          if (!isPremium) {
65              while (temp != NULL && !temp->isPremium) {
66                  prev = temp;
67                  temp = temp->next;
68              }
69          }
70
71          if (prev == NULL) { // if the new node is the first
72              //printf("CLIENT %d IS THE FIRST NODE\n", clientID);
73              newNode->next = queue->front;
```

```
74              queue->front = newNode;
75          } else if (temp == NULL) { // if the new node is the last
76              //printf("CLIENT %d IS THE LAST NODE\n", clientID);
77              queue->rear->next = newNode;
78              queue->rear = newNode;
79          } else { // if the new node is in the middle
80              //printf("CLIENT %d IS IN THE MIDDLE\n", clientID);
81              prev->next = newNode;
82              newNode->next = temp;
83          }
84      }
85
86      // print order of the queue
87      //Node *temp2 = queue->front;
88      //printf("--------------------\n");
89      //printf("QUEUE ORDER AFTER ENQUEUING:\n");
90      //while (temp2 != NULL) {
91      //    printf("Client %d isPremium: %s\n", temp2->clientID, temp2->
            isPremium ? "true" : "false");
92      //    temp2 = temp2->next;
93      //}
94      //printf("FRONT: %d\n", queue->front->clientID);
95      //printf("REAR: %d\n", queue->rear->clientID);
96      //printf("--------------------\n");
97
98      // increment timeonQueue for all clients in the queue
99      updatePriority(queue);
100
101     pthread_mutex_unlock(&queue->mutex);
102     sem_post(&queue->full);
103 }
104
105 void updatePriority(PriorityQueue *queue) {
106     Node *temp = queue->front;
107     while (temp != NULL) {
108         temp->timeInQueue++;
109         temp = temp->next;
110     }
111 }
```

```
112
113 void updateQueueWithPriority(PriorityQueue *queue, int maxWaitingTime)
    {
114
115     // lock the queue
116     pthread_mutex_lock(&queue->mutex);
117
118     // print order of the queue
119     //Node *temp1 = queue->front;
120     //printf("--------------------\n");
121     //printf("QUEUE ORDER BEFORE UPDATEING PRIORITY:\n");
122     //while (temp1 != NULL) {
123     //    printf("Client %d isPremium: %s TIME:%d\n", temp1->clientID,
        temp1->isPremium ? "true" : "false", temp1->timeInQueue);
124     //    temp1 = temp1->next;
125     //}
126     //printf("FRONT: %d\n", queue->front->clientID);
127     //printf("REAR: %d\n", queue->rear->clientID);
128     //printf("--------------------\n");
129
130     Node *temp = queue->front;
131     Node *prev = NULL;
132     while (temp != NULL) {
133         // if time in queue has reached the max waiting time put it in
            the front of the queue
134         if (temp->timeInQueue >= maxWaitingTime) {
135
136             // reset time in queue
137             temp->timeInQueue = 0;
138
139             // if the node is already in the front of the queue
140             if (prev == NULL) {
141                 temp = temp->next; // move to the next node
142             } else if (temp->next == NULL) { // if the node is the last
                node
143                 prev->next = NULL;              //
144                 temp->next = queue->front;
145                 queue->front = temp;
146                 queue->rear = prev;
```

```
147             } else { // if the node is in the middle of the queue
148                 prev->next = temp->next;
149                 temp->next = queue->front;
150                 queue->front = temp;
151                 temp = prev->next;
152             }
153 
154         } else { // move to the next node
155             prev = temp;
156             temp = temp->next;
157         }
158 
159     }
160 
161     // print order of the queue
162     //Node *temp2 = queue->front;
163     //printf("--------------------\n");
164     //printf("QUEUE ORDER AFTER UPDATING PRIORITY:\n");
165     //while (temp2 != NULL) {
166     //    printf("Client %d isPremium: %s TIME:%d\n", temp2->clientID,
        temp2->isPremium ? "true" : "false", temp2->timeInQueue);
167     //    temp2 = temp2->next;
168     //}
169     //printf("FRONT: %d\n", queue->front->clientID);
170     //printf("REAR: %d\n", queue->rear->clientID);
171     //printf("--------------------\n");
172 
173 
174     // unlock the queue
175     pthread_mutex_unlock(&queue->mutex);
176 }
177 
178 void enqueueFifo(PriorityQueue *queue, int clientID) {
179 
180     Node *newNode = createNode(clientID, false); // create a new node
181 
182     sem_wait(&queue->empty); // wait for empty space(se for >0 continua
        a decrementar se for =0 fica a espera que haja espa o na fila
        para adicionar um novo cliente)
```

```
183
184     pthread_mutex_lock(&queue->mutex); // lock the queue
185
186     if (queue->front == NULL) { // if the queue is empty(ao adicionar 1
            cliente passa para a fun   o de baixo porque a  fila j   n o
            est  vazia)
187         queue->front = newNode;
188         queue->rear = newNode;
189     } else {                    // if the queue is not empty(adiciona o
            cliente no fim da fila)
190         queue->rear->next = newNode;
191         queue->rear = newNode;
192     }
193
194     pthread_mutex_unlock(&queue->mutex);
195
196     sem_post(&queue->full);//(incrementa o sem foro full, indicando
            que h  um item na fila para ser consumido)
197
198 }
199
200
201 int dequeue(PriorityQueue *queue) {
202     //printf("CLIENT WANTS TO BE REMOVED FROM THE QUEUE\n");
203
204     sem_wait(&queue->full); // wait for full queue (decr)
205
206     //printf("CLIENT REMOVING FROM THE QUEUE\n");
207
208     pthread_mutex_lock(&queue->mutex); // lock the queue
209
210     Node *temp = queue->front;//(guarda o primeiro n  em temp do
            cliente que vai ser removido)
211
212     if (queue->front == NULL) {
213         printf("Queue is empty\n");
214         pthread_mutex_unlock(&queue->mutex);
215         sem_post(&queue->empty);
216         return -1;
```

```
217     }
218
219     queue->front = queue->front->next;//(avan a o front para o
            pr ximo cliente)
220
221     if (queue->front == NULL) { // if the queue is empty
222         queue->rear = NULL;
223     }
224
225     int clientID = temp->clientID;
226
227     free(temp);
228
229     pthread_mutex_unlock(&queue->mutex);
230     sem_post(&queue->empty);// incrementa o sem foro empty, indicando
            que h  um espa o na fila para ser preenchido
231     //printf("CLIENT REMOVED FROM THE QUEUE\n");
232
233     return clientID;
234 }
235
236 void freePriorityQueue(PriorityQueue *queue) {
237     Node *temp = queue->front;
238     while (temp != NULL) {
239         Node *next = temp->next;
240         free(temp);
241         temp = next;
242     }
243     pthread_mutex_destroy(&queue->mutex);
244     sem_destroy(&queue->empty);
245     sem_destroy(&queue->full);
246     free(queue);
247 }
```

A.5.7 parson/parson.h

```
1 /*
2  SPDX-License-Identifier: MIT
3
```

```
 Parson 1.5.3 (https://github.com/kgabis/parson)
 Copyright (c) 2012 - 2023 Krzysztof Gabis

 Permission is hereby granted, free of charge, to any person obtaining a copy
 of this software and associated documentation files (the "Software"), to deal
 in the Software without restriction, including without limitation the rights
 to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 copies of the Software, and to permit persons to whom the Software is
 furnished to do so, subject to the following conditions:

 The above copyright notice and this permission notice shall be included in
 all copies or substantial portions of the Software.

 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 THE SOFTWARE.
*/

#ifndef parson_parson_h
#define parson_parson_h

#ifdef __cplusplus
extern "C"
{
#endif
```

```
#if 0
} /* unconfuse xcode */
#endif

#define PARSON_VERSION_MAJOR 1
#define PARSON_VERSION_MINOR 5
#define PARSON_VERSION_PATCH 3

#define PARSON_VERSION_STRING "1.5.3"

#include <stddef.h>   /* size_t */

/* Types and enums */
typedef struct json_object_t JSON_Object;
typedef struct json_array_t  JSON_Array;
typedef struct json_value_t  JSON_Value;

enum json_value_type {
    JSONError   = -1,
    JSONNull    = 1,
    JSONString  = 2,
    JSONNumber  = 3,
    JSONObject  = 4,
    JSONArray   = 5,
    JSONBoolean = 6
};
typedef int JSON_Value_Type;

enum json_result_t {
    JSONSuccess = 0,
    JSONFailure = -1
};
typedef int JSON_Status;

typedef void * (*JSON_Malloc_Function)(size_t);
typedef void   (*JSON_Free_Function)(void *);

/* A function used for serializing numbers (see
   json_set_number_serialization_function).
```

```
71      If 'buf' is null then it should return number of bytes that would've
            been written
72      (but not more than PARSON_NUM_BUF_SIZE).
73  */
74  typedef int (*JSON_Number_Serialization_Function)(double num, char *buf
        );
75
76  /* Call only once, before calling any other function from parson API.
        If not called, malloc and free
77      from stdlib will be used for all allocations */
78  void json_set_allocation_functions(JSON_Malloc_Function malloc_fun,
        JSON_Free_Function free_fun);
79
80  /* Sets if slashes should be escaped or not when serializing JSON. By
        default slashes are escaped.
81   This function sets a global setting and is not thread safe. */
82  void json_set_escape_slashes(int escape_slashes);
83
84  /* Sets float format used for serialization of numbers.
85      Make sure it can't serialize to a string longer than
            PARSON_NUM_BUF_SIZE.
86      If format is null then the default format is used. */
87  void json_set_float_serialization_format(const char *format);
88
89  /* Sets a function that will be used for serialization of numbers.
90      If function is null then the default serialization function is used.
            */
91  void json_set_number_serialization_function(
        JSON_Number_Serialization_Function fun);
92
93  /* Parses first JSON value in a file, returns NULL in case of error */
94  JSON_Value * json_parse_file(const char *filename);
95
96  /* Parses first JSON value in a file and ignores comments (/ * * / and
        //),
97      returns NULL in case of error */
98  JSON_Value * json_parse_file_with_comments(const char *filename);
99
```

```
100 /*  Parses first JSON value in a string, returns NULL in case of error
    */
101 JSON_Value * json_parse_string(const char *string);
102
103 /*  Parses first JSON value in a string and ignores comments (/ * * /
    and //),
104     returns NULL in case of error */
105 JSON_Value * json_parse_string_with_comments(const char *string);
106
107 /* Serialization */
108 size_t      json_serialization_size(const JSON_Value *value); /*
    returns 0 on fail */
109 JSON_Status json_serialize_to_buffer(const JSON_Value *value, char *buf
    , size_t buf_size_in_bytes);
110 JSON_Status json_serialize_to_file(const JSON_Value *value, const char
    *filename);
111 char *      json_serialize_to_string(const JSON_Value *value);
112
113 /* Pretty serialization */
114 size_t      json_serialization_size_pretty(const JSON_Value *value); /*
     returns 0 on fail */
115 JSON_Status json_serialize_to_buffer_pretty(const JSON_Value *value,
    char *buf, size_t buf_size_in_bytes);
116 JSON_Status json_serialize_to_file_pretty(const JSON_Value *value,
    const char *filename);
117 char *      json_serialize_to_string_pretty(const JSON_Value *value);
118
119 void        json_free_serialized_string(char *string); /* frees string
    from json_serialize_to_string and json_serialize_to_string_pretty */
120
121 /* Comparing */
122 int  json_value_equals(const JSON_Value *a, const JSON_Value *b);
123
124 /* Validation
125    This is *NOT* JSON Schema. It validates json by checking if object
       have identically
126    named fields with matching types.
127    For example schema {"name":"", "age":0} will validate
128    {"name":"Joe", "age":25} and {"name":"Joe", "age":25, "gender":"m"},
```

```
129     but not {"name":"Joe"} or {"name":"Joe", "age":"Cucumber"}.
130     In case of arrays, only first value in schema is checked against all
            values in tested array.
131     Empty objects ({}) validate all objects, empty arrays ([]) validate
            all arrays,
132     null validates values of every type.
133  */
134 JSON_Status json_validate(const JSON_Value *schema, const JSON_Value *
        value);
135
136 /*
137  * JSON Object
138  */
139 JSON_Value  * json_object_get_value  (const JSON_Object *object, const
        char *name);
140 const char  * json_object_get_string (const JSON_Object *object, const
        char *name);
141 size_t        json_object_get_string_len(const JSON_Object *object,
        const char *name); /* doesn't account for last null character */
142 JSON_Object * json_object_get_object (const JSON_Object *object, const
        char *name);
143 JSON_Array  * json_object_get_array  (const JSON_Object *object, const
        char *name);
144 double        json_object_get_number (const JSON_Object *object, const
        char *name); /* returns 0 on fail */
145 int           json_object_get_boolean(const JSON_Object *object, const
        char *name); /* returns -1 on fail */
146
147 /* dotget functions enable addressing values with dot notation in
        nested objects,
148  just like in structs or c++/java/c# objects (e.g. objectA.objectB.
            value).
149  Because valid names in JSON can contain dots, some values may be
            inaccessible
150  this way. */
151 JSON_Value  * json_object_dotget_value  (const JSON_Object *object,
        const char *name);
152 const char  * json_object_dotget_string (const JSON_Object *object,
        const char *name);
```

```
153 size_t        json_object_dotget_string_len(const JSON_Object *object,
    const char *name); /* doesn't account for last null character */
154 JSON_Object * json_object_dotget_object (const JSON_Object *object,
    const char *name);
155 JSON_Array  * json_object_dotget_array  (const JSON_Object *object,
    const char *name);
156 double        json_object_dotget_number (const JSON_Object *object,
    const char *name); /* returns 0 on fail */
157 int           json_object_dotget_boolean(const JSON_Object *object,
    const char *name); /* returns -1 on fail */
158
159 /* Functions to get available names */
160 size_t        json_object_get_count   (const JSON_Object *object);
161 const char  * json_object_get_name    (const JSON_Object *object,
    size_t index);
162 JSON_Value  * json_object_get_value_at(const JSON_Object *object,
    size_t index);
163 JSON_Value  * json_object_get_wrapping_value(const JSON_Object *object)
    ;
164
165 /* Functions to check if object has a value with a specific name.
    Returned value is 1 if object has
166  * a value and 0 if it doesn't. dothas functions behave exactly like
     dotget functions. */
167 int json_object_has_value        (const JSON_Object *object, const char
     *name);
168 int json_object_has_value_of_type(const JSON_Object *object, const char
     *name, JSON_Value_Type type);
169
170 int json_object_dothas_value        (const JSON_Object *object, const
    char *name);
171 int json_object_dothas_value_of_type(const JSON_Object *object, const
    char *name, JSON_Value_Type type);
172
173 /* Creates new name-value pair or frees and replaces old value with a
    new one.
174  * json_object_set_value does not copy passed value so it shouldn't be
     freed afterwards. */
```

```
175 JSON_Status json_object_set_value(JSON_Object *object, const char *name
        , JSON_Value *value);
176 JSON_Status json_object_set_string(JSON_Object *object, const char *
        name, const char *string);
177 JSON_Status json_object_set_string_with_len(JSON_Object *object, const
        char *name, const char *string, size_t len);  /* length shouldn't
        include last null character */
178 JSON_Status json_object_set_number(JSON_Object *object, const char *
        name, double number);
179 JSON_Status json_object_set_boolean(JSON_Object *object, const char *
        name, int boolean);
180 JSON_Status json_object_set_null(JSON_Object *object, const char *name)
        ;
181
182 /* Works like dotget functions, but creates whole hierarchy if
        necessary.
183  * json_object_dotset_value does not copy passed value so it shouldn't
         be freed afterwards. */
184 JSON_Status json_object_dotset_value(JSON_Object *object, const char *
        name, JSON_Value *value);
185 JSON_Status json_object_dotset_string(JSON_Object *object, const char *
        name, const char *string);
186 JSON_Status json_object_dotset_string_with_len(JSON_Object *object,
        const char *name, const char *string, size_t len); /* length shouldn
        't include last null character */
187 JSON_Status json_object_dotset_number(JSON_Object *object, const char *
        name, double number);
188 JSON_Status json_object_dotset_boolean(JSON_Object *object, const char
        *name, int boolean);
189 JSON_Status json_object_dotset_null(JSON_Object *object, const char *
        name);
190
191 /* Frees and removes name-value pair */
192 JSON_Status json_object_remove(JSON_Object *object, const char *name);
193
194 /* Works like dotget function, but removes name-value pair only on
        exact match. */
195 JSON_Status json_object_dotremove(JSON_Object *object, const char *key)
        ;
```

```
196
197 /* Removes all name-value pairs in object */
198 JSON_Status json_object_clear(JSON_Object *object);
199
200 /*
201  *JSON Array
202  */
203 JSON_Value  * json_array_get_value  (const JSON_Array *array, size_t
       index);
204 const char  * json_array_get_string (const JSON_Array *array, size_t
       index);
205 size_t        json_array_get_string_len(const JSON_Array *array, size_t
        index); /* doesn't account for last null character */
206 JSON_Object * json_array_get_object (const JSON_Array *array, size_t
       index);
207 JSON_Array  * json_array_get_array  (const JSON_Array *array, size_t
       index);
208 double        json_array_get_number (const JSON_Array *array, size_t
       index); /* returns 0 on fail */
209 int           json_array_get_boolean(const JSON_Array *array, size_t
       index); /* returns -1 on fail */
210 size_t        json_array_get_count  (const JSON_Array *array);
211 JSON_Value  * json_array_get_wrapping_value(const JSON_Array *array);
212
213 /* Frees and removes value at given index, does nothing and returns
       JSONFailure if index doesn't exist.
214  * Order of values in array may change during execution.  */
215 JSON_Status json_array_remove(JSON_Array *array, size_t i);
216
217 /* Frees and removes from array value at given index and replaces it
       with given one.
218  * Does nothing and returns JSONFailure if index doesn't exist.
219  * json_array_replace_value does not copy passed value so it shouldn't
        be freed afterwards. */
220 JSON_Status json_array_replace_value(JSON_Array *array, size_t i,
       JSON_Value *value);
221 JSON_Status json_array_replace_string(JSON_Array *array, size_t i,
       const char* string);
```

```
222 JSON_Status json_array_replace_string_with_len(JSON_Array *array,
       size_t i, const char *string, size_t len); /* length shouldn't
       include last null character */
223 JSON_Status json_array_replace_number(JSON_Array *array, size_t i,
       double number);
224 JSON_Status json_array_replace_boolean(JSON_Array *array, size_t i, int
        boolean);
225 JSON_Status json_array_replace_null(JSON_Array *array, size_t i);
226
227 /* Frees and removes all values from array */
228 JSON_Status json_array_clear(JSON_Array *array);
229
230 /* Appends new value at the end of array.
231  * json_array_append_value does not copy passed value so it shouldn't
        be freed afterwards. */
232 JSON_Status json_array_append_value(JSON_Array *array, JSON_Value *
       value);
233 JSON_Status json_array_append_string(JSON_Array *array, const char *
       string);
234 JSON_Status json_array_append_string_with_len(JSON_Array *array, const
       char *string, size_t len); /* length shouldn't include last null
       character */
235 JSON_Status json_array_append_number(JSON_Array *array, double number);
236 JSON_Status json_array_append_boolean(JSON_Array *array, int boolean);
237 JSON_Status json_array_append_null(JSON_Array *array);
238
239 /*
240  *JSON Value
241  */
242 JSON_Value * json_value_init_object (void);
243 JSON_Value * json_value_init_array  (void);
244 JSON_Value * json_value_init_string (const char *string); /* copies
       passed string */
245 JSON_Value * json_value_init_string_with_len(const char *string, size_t
        length); /* copies passed string, length shouldn't include last
       null character */
246 JSON_Value * json_value_init_number (double number);
247 JSON_Value * json_value_init_boolean(int boolean);
248 JSON_Value * json_value_init_null   (void);
```

```
249 JSON_Value * json_value_deep_copy   (const JSON_Value *value);
250 void          json_value_free        (JSON_Value *value);
251
252 JSON_Value_Type json_value_get_type   (const JSON_Value *value);
253 JSON_Object *   json_value_get_object (const JSON_Value *value);
254 JSON_Array  *   json_value_get_array  (const JSON_Value *value);
255 const char  *   json_value_get_string (const JSON_Value *value);
256 size_t          json_value_get_string_len(const JSON_Value *value); /*
       doesn't account for last null character */
257 double          json_value_get_number (const JSON_Value *value);
258 int             json_value_get_boolean(const JSON_Value *value);
259 JSON_Value  *   json_value_get_parent (const JSON_Value *value);
260
261 /* Same as above, but shorter */
262 JSON_Value_Type json_type   (const JSON_Value *value);
263 JSON_Object *   json_object (const JSON_Value *value);
264 JSON_Array  *   json_array  (const JSON_Value *value);
265 const char  *   json_string (const JSON_Value *value);
266 size_t          json_string_len(const JSON_Value *value); /* doesn't
       account for last null character */
267 double          json_number (const JSON_Value *value);
268 int             json_boolean(const JSON_Value *value);
269
270 #ifdef __cplusplus
271 }
272 #endif
273
274 #endif
```

A.5.8 parson/parson.c

```
1 /*
2  SPDX-License-Identifier: MIT
3
4  Parson 1.5.3 (https://github.com/kgabis/parson)
5  Copyright (c) 2012 - 2023 Krzysztof Gabis
6
7  Permission is hereby granted, free of charge, to any person obtaining
       a copy
```

```
 of this software and associated documentation files (the "Software"), to deal
 in the Software without restriction, including without limitation the rights
 to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 copies of the Software, and to permit persons to whom the Software is
 furnished to do so, subject to the following conditions:

 The above copyright notice and this permission notice shall be included in
 all copies or substantial portions of the Software.

 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 THE SOFTWARE.
*/
#ifdef _MSC_VER
#ifndef _CRT_SECURE_NO_WARNINGS
#define _CRT_SECURE_NO_WARNINGS
#endif /* _CRT_SECURE_NO_WARNINGS */
#endif /* _MSC_VER */

#include "parson.h"

#define PARSON_IMPL_VERSION_MAJOR 1
#define PARSON_IMPL_VERSION_MINOR 5
#define PARSON_IMPL_VERSION_PATCH 3

#if (PARSON_VERSION_MAJOR != PARSON_IMPL_VERSION_MAJOR)\
```

```
38 || (PARSON_VERSION_MINOR != PARSON_IMPL_VERSION_MINOR)\
39 || (PARSON_VERSION_PATCH != PARSON_IMPL_VERSION_PATCH)
40 #error "parson version mismatch between parson.c and parson.h"
41 #endif
42
43 #include <stdarg.h>
44 #include <stdio.h>
45 #include <stdlib.h>
46 #include <string.h>
47 #include <ctype.h>
48 #include <math.h>
49 #include <errno.h>
50
51 /* Apparently sscanf is not implemented in some "standard" libraries,
      so don't use it, if you
52  * don't have to. */
53 #ifdef sscanf
54 #undef sscanf
55 #define sscanf THINK_TWICE_ABOUT_USING_SSCANF
56 #endif
57
58 /* strcpy is unsafe */
59 #ifdef strcpy
60 #undef strcpy
61 #endif
62 #define strcpy USE_MEMCPY_INSTEAD_OF_STRCPY
63
64 #define STARTING_CAPACITY 16
65 #define MAX_NESTING       2048
66
67 #ifndef PARSON_DEFAULT_FLOAT_FORMAT
68 #define PARSON_DEFAULT_FLOAT_FORMAT "%1.17g" /* do not increase
      precision without incresing NUM_BUF_SIZE */
69 #endif
70
71 #ifndef PARSON_NUM_BUF_SIZE
72 #define PARSON_NUM_BUF_SIZE 64 /* double printed with "%1.17g" shouldn'
      t be longer than 25 bytes so let's be paranoid and use 64 */
73 #endif
```

```
74
75  #ifndef PARSON_INDENT_STR
76  #define PARSON_INDENT_STR "    "
77  #endif
78
79  #define SIZEOF_TOKEN(a)       (sizeof(a) - 1)
80  #define SKIP_CHAR(str)        ((*str)++)
81  #define SKIP_WHITESPACES(str) while (isspace((unsigned char)(**str))) {
        SKIP_CHAR(str); }
82  #define MAX(a, b)             ((a) > (b) ? (a) : (b))
83
84  #undef malloc
85  #undef free
86
87  #if defined(isnan) && defined(isinf)
88  #define IS_NUMBER_INVALID(x) (isnan((x)) || isinf((x)))
89  #else
90  #define IS_NUMBER_INVALID(x) (((x) * 0.0) != 0.0)
91  #endif
92
93  #define OBJECT_INVALID_IX ((size_t)-1)
94
95  static JSON_Malloc_Function parson_malloc = malloc;
96  static JSON_Free_Function parson_free = free;
97
98  static int parson_escape_slashes = 1;
99
100 static char *parson_float_format = NULL;
101
102 static JSON_Number_Serialization_Function
       parson_number_serialization_function = NULL;
103
104 #define IS_CONT(b) (((unsigned char)(b) & 0xC0) == 0x80) /* is utf-8
       continuation byte */
105
106 typedef int parson_bool_t;
107
108 #define PARSON_TRUE 1
109 #define PARSON_FALSE 0
```

```
110
111 typedef struct json_string {
112     char *chars;
113     size_t length;
114 } JSON_String;
115
116 /* Type definitions */
117 typedef union json_value_value {
118     JSON_String  string;
119     double       number;
120     JSON_Object *object;
121     JSON_Array  *array;
122     int          boolean;
123     int          null;
124 } JSON_Value_Value;
125
126 struct json_value_t {
127     JSON_Value      *parent;
128     JSON_Value_Type  type;
129     JSON_Value_Value value;
130 };
131
132 struct json_object_t {
133     JSON_Value    *wrapping_value;
134     size_t        *cells;
135     unsigned long *hashes;
136     char         **names;
137     JSON_Value   **values;
138     size_t        *cell_ixs;
139     size_t         count;
140     size_t         item_capacity;
141     size_t         cell_capacity;
142 };
143
144 struct json_array_t {
145     JSON_Value  *wrapping_value;
146     JSON_Value **items;
147     size_t       count;
148     size_t       capacity;
```

```
149 };
150
151 /* Various */
152 static char * read_file(const char *filename);
153 static void   remove_comments(char *string, const char *start_token,
      const char *end_token);
154 static char * parson_strndup(const char *string, size_t n);
155 static char * parson_strdup(const char *string);
156 static int    parson_sprintf(char * s, const char * format, ...);
157
158 static int    hex_char_to_int(char c);
159 static JSON_Status parse_utf16_hex(const char *string, unsigned int *
      result);
160 static int         num_bytes_in_utf8_sequence(unsigned char c);
161 static JSON_Status   verify_utf8_sequence(const unsigned char *string,
      int *len);
162 static parson_bool_t is_valid_utf8(const char *string, size_t
      string_len);
163 static parson_bool_t is_decimal(const char *string, size_t length);
164 static unsigned long hash_string(const char *string, size_t n);
165
166 /* JSON Object */
167 static JSON_Object * json_object_make(JSON_Value *wrapping_value);
168 static JSON_Status   json_object_init(JSON_Object *object, size_t
      capacity);
169 static void          json_object_deinit(JSON_Object *object,
      parson_bool_t free_keys, parson_bool_t free_values);
170 static JSON_Status   json_object_grow_and_rehash(JSON_Object *object);
171 static size_t        json_object_get_cell_ix(const JSON_Object *object,
       const char *key, size_t key_len, unsigned long hash, parson_bool_t
      *out_found);
172 static JSON_Status   json_object_add(JSON_Object *object, char *name,
      JSON_Value *value);
173 static JSON_Value  * json_object_getn_value(const JSON_Object *object,
      const char *name, size_t name_len);
174 static JSON_Status   json_object_remove_internal(JSON_Object *object,
      const char *name, parson_bool_t free_value);
175 static JSON_Status   json_object_dotremove_internal(JSON_Object *object
      , const char *name, parson_bool_t free_value);
```

```
176 static void           json_object_free(JSON_Object *object);
177
178 /* JSON Array */
179 static JSON_Array * json_array_make(JSON_Value *wrapping_value);
180 static JSON_Status  json_array_add(JSON_Array *array, JSON_Value *value
    );
181 static JSON_Status  json_array_resize(JSON_Array *array, size_t
    new_capacity);
182 static void         json_array_free(JSON_Array *array);
183
184 /* JSON Value */
185 static JSON_Value * json_value_init_string_no_copy(char *string, size_t
     length);
186 static const JSON_String * json_value_get_string_desc(const JSON_Value
    *value);
187
188 /* Parser */
189 static JSON_Status   skip_quotes(const char **string);
190 static JSON_Status   parse_utf16(const char **unprocessed, char **
    processed);
191 static char *        process_string(const char *input, size_t input_len
    , size_t *output_len);
192 static char *        get_quoted_string(const char **string, size_t *
    output_string_len);
193 static JSON_Value *  parse_object_value(const char **string, size_t
    nesting);
194 static JSON_Value *  parse_array_value(const char **string, size_t
    nesting);
195 static JSON_Value *  parse_string_value(const char **string);
196 static JSON_Value *  parse_boolean_value(const char **string);
197 static JSON_Value *  parse_number_value(const char **string);
198 static JSON_Value *  parse_null_value(const char **string);
199 static JSON_Value *  parse_value(const char **string, size_t nesting);
200
201 /* Serialization */
202 static int json_serialize_to_buffer_r(const JSON_Value *value, char *
    buf, int level, parson_bool_t is_pretty, char *num_buf);
203 static int json_serialize_string(const char *string, size_t len, char *
    buf);
```

```
204
205 /* Various */
206 static char * read_file(const char * filename) {
207     FILE *fp = fopen(filename, "r");
208     size_t size_to_read = 0;
209     size_t size_read = 0;
210     long pos;
211     char *file_contents;
212     if (!fp) {
213         return NULL;
214     }
215     fseek(fp, 0L, SEEK_END);
216     pos = ftell(fp);
217     if (pos < 0) {
218         fclose(fp);
219         return NULL;
220     }
221     size_to_read = pos;
222     rewind(fp);
223     file_contents = (char*)parson_malloc(sizeof(char) * (size_to_read +
          1));
224     if (!file_contents) {
225         fclose(fp);
226         return NULL;
227     }
228     size_read = fread(file_contents, 1, size_to_read, fp);
229     if (size_read == 0 || ferror(fp)) {
230         fclose(fp);
231         parson_free(file_contents);
232         return NULL;
233     }
234     fclose(fp);
235     file_contents[size_read] = '\0';
236     return file_contents;
237 }
238
239 static void remove_comments(char *string, const char *start_token,
       const char *end_token) {
240     parson_bool_t in_string = PARSON_FALSE, escaped = PARSON_FALSE;
```

```
241     size_t i;
242     char *ptr = NULL, current_char;
243     size_t start_token_len = strlen(start_token);
244     size_t end_token_len = strlen(end_token);
245     if (start_token_len == 0 || end_token_len == 0) {
246         return;
247     }
248     while ((current_char = *string) != '\0') {
249         if (current_char == '\\' && !escaped) {
250             escaped = PARSON_TRUE;
251             string++;
252             continue;
253         } else if (current_char == '\"' && !escaped) {
254             in_string = !in_string;
255         } else if (!in_string && strncmp(string, start_token,
                start_token_len) == 0) {
256             for(i = 0; i < start_token_len; i++) {
257                 string[i] = ' ';
258             }
259             string = string + start_token_len;
260             ptr = strstr(string, end_token);
261             if (!ptr) {
262                 return;
263             }
264             for (i = 0; i < (ptr - string) + end_token_len; i++) {
265                 string[i] = ' ';
266             }
267             string = ptr + end_token_len - 1;
268         }
269         escaped = PARSON_FALSE;
270         string++;
271     }
272 }
273
274 static char * parson_strndup(const char *string, size_t n) {
275     /* We expect the caller has validated that 'n' fits within the
            input buffer. */
276     char *output_string = (char*)parson_malloc(n + 1);
277     if (!output_string) {
```

```
278             return NULL;
279         }
280         output_string[n] = '\0';
281         memcpy(output_string, string, n);
282         return output_string;
283     }
284 
285     static char * parson_strdup(const char *string) {
286         return parson_strndup(string, strlen(string));
287     }
288 
289     static int parson_sprintf(char * s, const char * format, ...) {
290         int result;
291         va_list args;
292         va_start(args, format);
293 
294         #if defined(__APPLE__) && defined(__clang__)
295             #pragma clang diagnostic push
296             #pragma clang diagnostic ignored "-Wdeprecated-declarations"
297         #endif
298             result = vsprintf(s, format, args);
299         #if defined(__APPLE__) && defined(__clang__)
300             #pragma clang diagnostic pop
301         #endif
302 
303         va_end(args);
304         return result;
305 
306     }
307 
308     static int hex_char_to_int(char c) {
309         if (c >= '0' && c <= '9') {
310             return c - '0';
311         } else if (c >= 'a' && c <= 'f') {
312             return c - 'a' + 10;
313         } else if (c >= 'A' && c <= 'F') {
314             return c - 'A' + 10;
315         }
316         return -1;
```

```
317 }
318
319 static JSON_Status parse_utf16_hex(const char *s, unsigned int *result)
        {
320     int x1, x2, x3, x4;
321     if (s[0] == '\0' || s[1] == '\0' || s[2] == '\0' || s[3] == '\0') {
322         return JSONFailure;
323     }
324     x1 = hex_char_to_int(s[0]);
325     x2 = hex_char_to_int(s[1]);
326     x3 = hex_char_to_int(s[2]);
327     x4 = hex_char_to_int(s[3]);
328     if (x1 == -1 || x2 == -1 || x3 == -1 || x4 == -1) {
329         return JSONFailure;
330     }
331     *result = (unsigned int)((x1 << 12) | (x2 << 8) | (x3 << 4) | x4);
332     return JSONSuccess;
333 }
334
335 static int num_bytes_in_utf8_sequence(unsigned char c) {
336     if (c == 0xC0 || c == 0xC1 || c > 0xF4 || IS_CONT(c)) {
337         return 0;
338     } else if ((c & 0x80) == 0) {    /* 0xxxxxxx */
339         return 1;
340     } else if ((c & 0xE0) == 0xC0) { /* 110xxxxx */
341         return 2;
342     } else if ((c & 0xF0) == 0xE0) { /* 1110xxxx */
343         return 3;
344     } else if ((c & 0xF8) == 0xF0) { /* 11110xxx */
345         return 4;
346     }
347     return 0; /* won't happen */
348 }
349
350 static JSON_Status verify_utf8_sequence(const unsigned char *string,
      int *len) {
351     unsigned int cp = 0;
352     *len = num_bytes_in_utf8_sequence(string[0]);
353
```

```
354     if (*len == 1) {
355         cp = string[0];
356     } else if (*len == 2 && IS_CONT(string[1])) {
357         cp = string[0] & 0x1F;
358         cp = (cp << 6) | (string[1] & 0x3F);
359     } else if (*len == 3 && IS_CONT(string[1]) && IS_CONT(string[2])) {
360         cp = ((unsigned char)string[0]) & 0xF;
361         cp = (cp << 6) | (string[1] & 0x3F);
362         cp = (cp << 6) | (string[2] & 0x3F);
363     } else if (*len == 4 && IS_CONT(string[1]) && IS_CONT(string[2]) &&
        IS_CONT(string[3])) {
364         cp = string[0] & 0x7;
365         cp = (cp << 6) | (string[1] & 0x3F);
366         cp = (cp << 6) | (string[2] & 0x3F);
367         cp = (cp << 6) | (string[3] & 0x3F);
368     } else {
369         return JSONFailure;
370     }
371
372     /* overlong encodings */
373     if ((cp < 0x80    && *len > 1) ||
374         (cp < 0x800   && *len > 2) ||
375         (cp < 0x10000 && *len > 3)) {
376         return JSONFailure;
377     }
378
379     /* invalid unicode */
380     if (cp > 0x10FFFF) {
381         return JSONFailure;
382     }
383
384     /* surrogate halves */
385     if (cp >= 0xD800 && cp <= 0xDFFF) {
386         return JSONFailure;
387     }
388
389     return JSONSuccess;
390 }
391
```

```
392 static int is_valid_utf8(const char *string, size_t string_len) {
393     int len = 0;
394     const char *string_end =  string + string_len;
395     while (string < string_end) {
396         if (verify_utf8_sequence((const unsigned char*)string, &len) !=
                JSONSuccess) {
397             return PARSON_FALSE;
398         }
399         string += len;
400     }
401     return PARSON_TRUE;
402 }
403
404 static parson_bool_t is_decimal(const char *string, size_t length) {
405     if (length > 1 && string[0] == '0' && string[1] != '.') {
406         return PARSON_FALSE;
407     }
408     if (length > 2 && !strncmp(string, "-0", 2) && string[2] != '.') {
409         return PARSON_FALSE;
410     }
411     while (length--) {
412         if (strchr("xX", string[length])) {
413             return PARSON_FALSE;
414         }
415     }
416     return PARSON_TRUE;
417 }
418
419 static unsigned long hash_string(const char *string, size_t n) {
420 #ifdef PARSON_FORCE_HASH_COLLISIONS
421     (void)string;
422     (void)n;
423     return 0;
424 #else
425     unsigned long hash = 5381;
426     unsigned char c;
427     size_t i = 0;
428     for (i = 0; i < n; i++) {
429         c = string[i];
```

```
430         if (c == '\0') {
431             break;
432         }
433         hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
434     }
435     return hash;
436 #endif
437 }
438 
439 /* JSON Object */
440 static JSON_Object * json_object_make(JSON_Value *wrapping_value) {
441     JSON_Status res = JSONFailure;
442     JSON_Object *new_obj = (JSON_Object*)parson_malloc(sizeof(
        JSON_Object));
443     if (new_obj == NULL) {
444         return NULL;
445     }
446     new_obj->wrapping_value = wrapping_value;
447     res = json_object_init(new_obj, 0);
448     if (res != JSONSuccess) {
449         parson_free(new_obj);
450         return NULL;
451     }
452     return new_obj;
453 }
454 
455 static JSON_Status json_object_init(JSON_Object *object, size_t
    capacity) {
456     unsigned int i = 0;
457 
458     object->cells = NULL;
459     object->names = NULL;
460     object->values = NULL;
461     object->cell_ixs = NULL;
462     object->hashes = NULL;
463 
464     object->count = 0;
465     object->cell_capacity = capacity;
466     object->item_capacity = (unsigned int)(capacity * 7/10);
```

```
467
468     if (capacity == 0) {
469         return JSONSuccess;
470     }
471
472     object->cells = (size_t*)parson_malloc(object->cell_capacity *
            sizeof(*object->cells));
473     object->names = (char**)parson_malloc(object->item_capacity *
            sizeof(*object->names));
474     object->values = (JSON_Value**)parson_malloc(object->item_capacity
            * sizeof(*object->values));
475     object->cell_ixs = (size_t*)parson_malloc(object->item_capacity *
            sizeof(*object->cell_ixs));
476     object->hashes = (unsigned long*)parson_malloc(object->
            item_capacity * sizeof(*object->hashes));
477     if (object->cells == NULL
478         || object->names == NULL
479         || object->values == NULL
480         || object->cell_ixs == NULL
481         || object->hashes == NULL) {
482         goto error;
483     }
484     for (i = 0; i < object->cell_capacity; i++) {
485         object->cells[i] = OBJECT_INVALID_IX;
486     }
487     return JSONSuccess;
488 error:
489     parson_free(object->cells);
490     parson_free(object->names);
491     parson_free(object->values);
492     parson_free(object->cell_ixs);
493     parson_free(object->hashes);
494     return JSONFailure;
495 }
496
497 static void json_object_deinit(JSON_Object *object, parson_bool_t
        free_keys, parson_bool_t free_values) {
498     unsigned int i = 0;
499     for (i = 0; i < object->count; i++) {
```

```
500         if (free_keys) {
501             parson_free(object->names[i]);
502         }
503         if (free_values) {
504             json_value_free(object->values[i]);
505         }
506     }
507 
508     object->count = 0;
509     object->item_capacity = 0;
510     object->cell_capacity = 0;
511 
512     parson_free(object->cells);
513     parson_free(object->names);
514     parson_free(object->values);
515     parson_free(object->cell_ixs);
516     parson_free(object->hashes);
517 
518     object->cells = NULL;
519     object->names = NULL;
520     object->values = NULL;
521     object->cell_ixs = NULL;
522     object->hashes = NULL;
523 }
524 
525 static JSON_Status json_object_grow_and_rehash(JSON_Object *object) {
526     JSON_Value *wrapping_value = NULL;
527     JSON_Object new_object;
528     char *key = NULL;
529     JSON_Value *value = NULL;
530     unsigned int i = 0;
531     size_t new_capacity = MAX(object->cell_capacity * 2,
        STARTING_CAPACITY);
532     JSON_Status res = json_object_init(&new_object, new_capacity);
533     if (res != JSONSuccess) {
534         return JSONFailure;
535     }
536 
537     wrapping_value = json_object_get_wrapping_value(object);
```

```
538     new_object.wrapping_value = wrapping_value;
539 
540     for (i = 0; i < object->count; i++) {
541         key = object->names[i];
542         value = object->values[i];
543         res = json_object_add(&new_object, key, value);
544         if (res != JSONSuccess) {
545             json_object_deinit(&new_object, PARSON_FALSE, PARSON_FALSE)
                ;
546             return JSONFailure;
547         }
548         value->parent = wrapping_value;
549     }
550     json_object_deinit(object, PARSON_FALSE, PARSON_FALSE);
551     *object = new_object;
552     return JSONSuccess;
553 }
554 
555 static size_t json_object_get_cell_ix(const JSON_Object *object, const
    char *key, size_t key_len, unsigned long hash, parson_bool_t *
    out_found) {
556     size_t cell_ix = hash & (object->cell_capacity - 1);
557     size_t cell = 0;
558     size_t ix = 0;
559     unsigned int i = 0;
560     unsigned long hash_to_check = 0;
561     const char *key_to_check = NULL;
562     size_t key_to_check_len = 0;
563 
564     *out_found = PARSON_FALSE;
565 
566     for (i = 0; i < object->cell_capacity; i++) {
567         ix = (cell_ix + i) & (object->cell_capacity - 1);
568         cell = object->cells[ix];
569         if (cell == OBJECT_INVALID_IX) {
570             return ix;
571         }
572         hash_to_check = object->hashes[cell];
573         if (hash != hash_to_check) {
```

```
574                 continue;
575             }
576             key_to_check = object->names[cell];
577             key_to_check_len = strlen(key_to_check);
578             if (key_to_check_len == key_len && strncmp(key, key_to_check,
                    key_len) == 0) {
579                 *out_found = PARSON_TRUE;
580                 return ix;
581             }
582         }
583         return OBJECT_INVALID_IX;
584     }
585
586     static JSON_Status json_object_add(JSON_Object *object, char *name,
            JSON_Value *value) {
587         unsigned long hash = 0;
588         parson_bool_t found = PARSON_FALSE;
589         size_t cell_ix = 0;
590         JSON_Status res = JSONFailure;
591
592         if (!object || !name || !value) {
593             return JSONFailure;
594         }
595
596         hash = hash_string(name, strlen(name));
597         found = PARSON_FALSE;
598         cell_ix = json_object_get_cell_ix(object, name, strlen(name), hash,
                &found);
599         if (found) {
600             return JSONFailure;
601         }
602
603         if (object->count >= object->item_capacity) {
604             res = json_object_grow_and_rehash(object);
605             if (res != JSONSuccess) {
606                 return JSONFailure;
607             }
608             cell_ix = json_object_get_cell_ix(object, name, strlen(name),
                    hash, &found);
```

```
609     }
610 
611     object->names[object->count] = name;
612     object->cells[cell_ix] = object->count;
613     object->values[object->count] = value;
614     object->cell_ixs[object->count] = cell_ix;
615     object->hashes[object->count] = hash;
616     object->count++;
617     value->parent = json_object_get_wrapping_value(object);
618 
619     return JSONSuccess;
620 }
621 
622 static JSON_Value * json_object_getn_value(const JSON_Object *object,
        const char *name, size_t name_len) {
623     unsigned long hash = 0;
624     parson_bool_t found = PARSON_FALSE;
625     size_t cell_ix = 0;
626     size_t item_ix = 0;
627     if (!object || !name) {
628         return NULL;
629     }
630     hash = hash_string(name, name_len);
631     found = PARSON_FALSE;
632     cell_ix = json_object_get_cell_ix(object, name, name_len, hash, &
            found);
633     if (!found) {
634         return NULL;
635     }
636     item_ix = object->cells[cell_ix];
637     return object->values[item_ix];
638 }
639 
640 static JSON_Status json_object_remove_internal(JSON_Object *object,
        const char *name, parson_bool_t free_value) {
641     unsigned long hash = 0;
642     parson_bool_t found = PARSON_FALSE;
643     size_t cell = 0;
644     size_t item_ix = 0;
```

```
645     size_t last_item_ix = 0;
646     size_t i = 0;
647     size_t j = 0;
648     size_t x = 0;
649     size_t k = 0;
650     JSON_Value *val = NULL;
651 
652     if (object == NULL) {
653         return JSONFailure;
654     }
655 
656     hash = hash_string(name, strlen(name));
657     found = PARSON_FALSE;
658     cell = json_object_get_cell_ix(object, name, strlen(name), hash, &
            found);
659     if (!found) {
660         return JSONFailure;
661     }
662 
663     item_ix = object->cells[cell];
664     if (free_value) {
665         val = object->values[item_ix];
666         json_value_free(val);
667         val = NULL;
668     }
669 
670     parson_free(object->names[item_ix]);
671     last_item_ix = object->count - 1;
672     if (item_ix < last_item_ix) {
673         object->names[item_ix] = object->names[last_item_ix];
674         object->values[item_ix] = object->values[last_item_ix];
675         object->cell_ixs[item_ix] = object->cell_ixs[last_item_ix];
676         object->hashes[item_ix] = object->hashes[last_item_ix];
677         object->cells[object->cell_ixs[item_ix]] = item_ix;
678     }
679     object->count--;
680 
681     i = cell;
682     j = i;
```

```
683     for (x = 0; x < (object->cell_capacity - 1); x++) {
684         j = (j + 1) & (object->cell_capacity - 1);
685         if (object->cells[j] == OBJECT_INVALID_IX) {
686             break;
687         }
688         k = object->hashes[object->cells[j]] & (object->cell_capacity -
                1);
689         if ((j > i && (k <= i || k > j))
690          || (j < i && (k <= i && k > j))) {
691             object->cell_ixs[object->cells[j]] = i;
692             object->cells[i] = object->cells[j];
693             i = j;
694         }
695     }
696     object->cells[i] = OBJECT_INVALID_IX;
697     return JSONSuccess;
698 }
699
700 static JSON_Status json_object_dotremove_internal(JSON_Object *object,
       const char *name, parson_bool_t free_value) {
701     JSON_Value *temp_value = NULL;
702     JSON_Object *temp_object = NULL;
703     const char *dot_pos = strchr(name, '.');
704     if (!dot_pos) {
705         return json_object_remove_internal(object, name, free_value);
706     }
707     temp_value = json_object_getn_value(object, name, dot_pos - name);
708     if (json_value_get_type(temp_value) != JSONObject) {
709         return JSONFailure;
710     }
711     temp_object = json_value_get_object(temp_value);
712     return json_object_dotremove_internal(temp_object, dot_pos + 1,
           free_value);
713 }
714
715 static void json_object_free(JSON_Object *object) {
716     json_object_deinit(object, PARSON_TRUE, PARSON_TRUE);
717     parson_free(object);
718 }
```

```
719
720 /* JSON Array */
721 static JSON_Array * json_array_make(JSON_Value *wrapping_value) {
722     JSON_Array *new_array = (JSON_Array*)parson_malloc(sizeof(
        JSON_Array));
723     if (new_array == NULL) {
724         return NULL;
725     }
726     new_array->wrapping_value = wrapping_value;
727     new_array->items = (JSON_Value**)NULL;
728     new_array->capacity = 0;
729     new_array->count = 0;
730     return new_array;
731 }
732
733 static JSON_Status json_array_add(JSON_Array *array, JSON_Value *value)
        {
734     if (array->count >= array->capacity) {
735         size_t new_capacity = MAX(array->capacity * 2,
            STARTING_CAPACITY);
736         if (json_array_resize(array, new_capacity) != JSONSuccess) {
737             return JSONFailure;
738         }
739     }
740     value->parent = json_array_get_wrapping_value(array);
741     array->items[array->count] = value;
742     array->count++;
743     return JSONSuccess;
744 }
745
746 static JSON_Status json_array_resize(JSON_Array *array, size_t
       new_capacity) {
747     JSON_Value **new_items = NULL;
748     if (new_capacity == 0) {
749         return JSONFailure;
750     }
751     new_items = (JSON_Value**)parson_malloc(new_capacity * sizeof(
        JSON_Value*));
752     if (new_items == NULL) {
```

```
753             return JSONFailure;
754         }
755         if (array->items != NULL && array->count > 0) {
756             memcpy(new_items, array->items, array->count * sizeof(
                    JSON_Value*));
757         }
758         parson_free(array->items);
759         array->items = new_items;
760         array->capacity = new_capacity;
761         return JSONSuccess;
762     }
763 
764     static void json_array_free(JSON_Array *array) {
765         size_t i;
766         for (i = 0; i < array->count; i++) {
767             json_value_free(array->items[i]);
768         }
769         parson_free(array->items);
770         parson_free(array);
771     }
772 
773     /* JSON Value */
774     static JSON_Value * json_value_init_string_no_copy(char *string, size_t
        length) {
775         JSON_Value *new_value = (JSON_Value*)parson_malloc(sizeof(
                JSON_Value));
776         if (!new_value) {
777             return NULL;
778         }
779         new_value->parent = NULL;
780         new_value->type = JSONString;
781         new_value->value.string.chars = string;
782         new_value->value.string.length = length;
783         return new_value;
784     }
785 
786     /* Parser */
787     static JSON_Status skip_quotes(const char **string) {
788         if (**string != '\"') {
```

```
789         return JSONFailure;
790     }
791     SKIP_CHAR(string);
792     while (**string != '\"') {
793         if (**string == '\0') {
794             return JSONFailure;
795         } else if (**string == '\\') {
796             SKIP_CHAR(string);
797             if (**string == '\0') {
798                 return JSONFailure;
799             }
800         }
801         SKIP_CHAR(string);
802     }
803     SKIP_CHAR(string);
804     return JSONSuccess;
805 }
806
807 static JSON_Status parse_utf16(const char **unprocessed, char **
    processed) {
808     unsigned int cp, lead, trail;
809     char *processed_ptr = *processed;
810     const char *unprocessed_ptr = *unprocessed;
811     JSON_Status status = JSONFailure;
812     unprocessed_ptr++; /* skips u */
813     status = parse_utf16_hex(unprocessed_ptr, &cp);
814     if (status != JSONSuccess) {
815         return JSONFailure;
816     }
817     if (cp < 0x80) {
818         processed_ptr[0] = (char)cp; /* 0xxxxxxx */
819     } else if (cp < 0x800) {
820         processed_ptr[0] = ((cp >> 6) & 0x1F) | 0xC0; /* 110xxxxx */
821         processed_ptr[1] = ((cp)      & 0x3F) | 0x80; /* 10xxxxxx */
822         processed_ptr += 1;
823     } else if (cp < 0xD800 || cp > 0xDFFF) {
824         processed_ptr[0] = ((cp >> 12) & 0x0F) | 0xE0; /* 1110xxxx */
825         processed_ptr[1] = ((cp >> 6)  & 0x3F) | 0x80; /* 10xxxxxx */
826         processed_ptr[2] = ((cp)       & 0x3F) | 0x80; /* 10xxxxxx */
```

```
827             processed_ptr += 2;
828         } else if (cp >= 0xD800 && cp <= 0xDBFF) { /* lead surrogate (0
                xD800..0xDBFF) */
829             lead = cp;
830             unprocessed_ptr += 4; /* should always be within the buffer,
                    otherwise previous sscanf would fail */
831             if (*unprocessed_ptr++ != '\\' || *unprocessed_ptr++ != 'u') {
832                 return JSONFailure;
833             }
834             status = parse_utf16_hex(unprocessed_ptr, &trail);
835             if (status != JSONSuccess || trail < 0xDC00 || trail > 0xDFFF)
                    { /* valid trail surrogate? (0xDC00..0xDFFF) */
836                 return JSONFailure;
837             }
838             cp = ((((lead - 0xD800) & 0x3FF) << 10) | ((trail - 0xDC00) & 0
                    x3FF)) + 0x010000;
839             processed_ptr[0] = (((cp >> 18) & 0x07) | 0xF0); /* 11110xxx */
840             processed_ptr[1] = (((cp >> 12) & 0x3F) | 0x80); /* 10xxxxxx */
841             processed_ptr[2] = (((cp >> 6)  & 0x3F) | 0x80); /* 10xxxxxx */
842             processed_ptr[3] = (((cp)       & 0x3F) | 0x80); /* 10xxxxxx */
843             processed_ptr += 3;
844         } else { /* trail surrogate before lead surrogate */
845             return JSONFailure;
846         }
847         unprocessed_ptr += 3;
848         *processed = processed_ptr;
849         *unprocessed = unprocessed_ptr;
850         return JSONSuccess;
851 }
852
853
854 /* Copies and processes passed string up to supplied length.
855 Example: "\u006Corem ipsum" -> lorem ipsum */
856 static char* process_string(const char *input, size_t input_len, size_t
        *output_len) {
857     const char *input_ptr = input;
858     size_t initial_size = (input_len + 1) * sizeof(char);
859     size_t final_size = 0;
860     char *output = NULL, *output_ptr = NULL, *resized_output = NULL;
```

```
861     output = (char*)parson_malloc(initial_size);
862     if (output == NULL) {
863         goto error;
864     }
865     output_ptr = output;
866     while ((*input_ptr != '\0') && (size_t)(input_ptr - input) <
            input_len) {
867         if (*input_ptr == '\\') {
868             input_ptr++;
869             switch (*input_ptr) {
870                 case '\"': *output_ptr = '\"'; break;
871                 case '\\': *output_ptr = '\\'; break;
872                 case '/':  *output_ptr = '/';  break;
873                 case 'b':  *output_ptr = '\b'; break;
874                 case 'f':  *output_ptr = '\f'; break;
875                 case 'n':  *output_ptr = '\n'; break;
876                 case 'r':  *output_ptr = '\r'; break;
877                 case 't':  *output_ptr = '\t'; break;
878                 case 'u':
879                     if (parse_utf16(&input_ptr, &output_ptr) !=
                            JSONSuccess) {
880                         goto error;
881                     }
882                     break;
883                 default:
884                     goto error;
885             }
886         } else if ((unsigned char)*input_ptr < 0x20) {
887             goto error; /* 0x00-0x19 are invalid characters for json
                    string (http://www.ietf.org/rfc/rfc4627.txt) */
888         } else {
889             *output_ptr = *input_ptr;
890         }
891         output_ptr++;
892         input_ptr++;
893     }
894     *output_ptr = '\0';
895     /* resize to new length */
896     final_size = (size_t)(output_ptr-output) + 1;
```

```
897     /* todo: don't resize if final_size == initial_size */
898     resized_output = (char*)parson_malloc(final_size);
899     if (resized_output == NULL) {
900         goto error;
901     }
902     memcpy(resized_output, output, final_size);
903     *output_len = final_size - 1;
904     parson_free(output);
905     return resized_output;
906 error:
907     parson_free(output);
908     return NULL;
909 }
910 
911 /* Return processed contents of a string between quotes and
912    skips passed argument to a matching quote. */
913 static char * get_quoted_string(const char **string, size_t *
    output_string_len) {
914     const char *string_start = *string;
915     size_t input_string_len = 0;
916     JSON_Status status = skip_quotes(string);
917     if (status != JSONSuccess) {
918         return NULL;
919     }
920     input_string_len = *string - string_start - 2; /* length without
        quotes */
921     return process_string(string_start + 1, input_string_len,
        output_string_len);
922 }
923 
924 static JSON_Value * parse_value(const char **string, size_t nesting) {
925     if (nesting > MAX_NESTING) {
926         return NULL;
927     }
928     SKIP_WHITESPACES(string);
929     switch (**string) {
930         case '{':
931             return parse_object_value(string, nesting + 1);
932         case '[':
```

```
933             return parse_array_value(string, nesting + 1);
934         case '\"':
935             return parse_string_value(string);
936         case 'f': case 't':
937             return parse_boolean_value(string);
938         case '-':
939         case '0': case '1': case '2': case '3': case '4':
940         case '5': case '6': case '7': case '8': case '9':
941             return parse_number_value(string);
942         case 'n':
943             return parse_null_value(string);
944         default:
945             return NULL;
946     }
947 }
948
949 static JSON_Value * parse_object_value(const char **string, size_t
        nesting) {
950     JSON_Status status = JSONFailure;
951     JSON_Value *output_value = NULL, *new_value = NULL;
952     JSON_Object *output_object = NULL;
953     char *new_key = NULL;
954
955     output_value = json_value_init_object();
956     if (output_value == NULL) {
957         return NULL;
958     }
959     if (**string != '{') {
960         json_value_free(output_value);
961         return NULL;
962     }
963     output_object = json_value_get_object(output_value);
964     SKIP_CHAR(string);
965     SKIP_WHITESPACES(string);
966     if (**string == '}') { /* empty object */
967         SKIP_CHAR(string);
968         return output_value;
969     }
970     while (**string != '\0') {
```

```
971             size_t key_len = 0;
972             new_key = get_quoted_string(string, &key_len);
973             /* We do not support key names with embedded \0 chars */
974             if (!new_key) {
975                 json_value_free(output_value);
976                 return NULL;
977             }
978             if (key_len != strlen(new_key)) {
979                 parson_free(new_key);
980                 json_value_free(output_value);
981                 return NULL;
982             }
983             SKIP_WHITESPACES(string);
984             if (**string != ':') {
985                 parson_free(new_key);
986                 json_value_free(output_value);
987                 return NULL;
988             }
989             SKIP_CHAR(string);
990             new_value = parse_value(string, nesting);
991             if (new_value == NULL) {
992                 parson_free(new_key);
993                 json_value_free(output_value);
994                 return NULL;
995             }
996             status = json_object_add(output_object, new_key, new_value);
997             if (status != JSONSuccess) {
998                 parson_free(new_key);
999                 json_value_free(new_value);
1000                json_value_free(output_value);
1001                return NULL;
1002            }
1003            SKIP_WHITESPACES(string);
1004            if (**string != ',') {
1005                break;
1006            }
1007            SKIP_CHAR(string);
1008            SKIP_WHITESPACES(string);
1009            if (**string == '}') {
```

```
1010                break;
1011            }
1012        }
1013        SKIP_WHITESPACES(string);
1014        if (**string != '}') {
1015            json_value_free(output_value);
1016            return NULL;
1017        }
1018        SKIP_CHAR(string);
1019        return output_value;
1020    }
1021
1022    static JSON_Value * parse_array_value(const char **string, size_t
        nesting) {
1023        JSON_Value *output_value = NULL, *new_array_value = NULL;
1024        JSON_Array *output_array = NULL;
1025        output_value = json_value_init_array();
1026        if (output_value == NULL) {
1027            return NULL;
1028        }
1029        if (**string != '[') {
1030            json_value_free(output_value);
1031            return NULL;
1032        }
1033        output_array = json_value_get_array(output_value);
1034        SKIP_CHAR(string);
1035        SKIP_WHITESPACES(string);
1036        if (**string == ']') { /* empty array */
1037            SKIP_CHAR(string);
1038            return output_value;
1039        }
1040        while (**string != '\0') {
1041            new_array_value = parse_value(string, nesting);
1042            if (new_array_value == NULL) {
1043                json_value_free(output_value);
1044                return NULL;
1045            }
1046            if (json_array_add(output_array, new_array_value) !=
                JSONSuccess) {
```

```
1047                json_value_free(new_array_value);
1048                json_value_free(output_value);
1049                return NULL;
1050            }
1051            SKIP_WHITESPACES(string);
1052            if (**string != ',') {
1053                break;
1054            }
1055            SKIP_CHAR(string);
1056            SKIP_WHITESPACES(string);
1057            if (**string == ']') {
1058                break;
1059            }
1060        }
1061        SKIP_WHITESPACES(string);
1062        if (**string != ']' || /* Trim array after parsing is over */
1063            json_array_resize(output_array, json_array_get_count(
                    output_array)) != JSONSuccess) {
1064                json_value_free(output_value);
1065                return NULL;
1066        }
1067        SKIP_CHAR(string);
1068        return output_value;
1069    }
1070
1071    static JSON_Value * parse_string_value(const char **string) {
1072        JSON_Value *value = NULL;
1073        size_t new_string_len = 0;
1074        char *new_string = get_quoted_string(string, &new_string_len);
1075        if (new_string == NULL) {
1076            return NULL;
1077        }
1078        value = json_value_init_string_no_copy(new_string, new_string_len);
1079        if (value == NULL) {
1080            parson_free(new_string);
1081            return NULL;
1082        }
1083        return value;
1084    }
```

```
1085
1086 static JSON_Value * parse_boolean_value(const char **string) {
1087     size_t true_token_size = SIZEOF_TOKEN("true");
1088     size_t false_token_size = SIZEOF_TOKEN("false");
1089     if (strncmp("true", *string, true_token_size) == 0) {
1090         *string += true_token_size;
1091         return json_value_init_boolean(1);
1092     } else if (strncmp("false", *string, false_token_size) == 0) {
1093         *string += false_token_size;
1094         return json_value_init_boolean(0);
1095     }
1096     return NULL;
1097 }
1098
1099 static JSON_Value * parse_number_value(const char **string) {
1100     char *end;
1101     double number = 0;
1102     errno = 0;
1103     number = strtod(*string, &end);
1104     if (errno == ERANGE && (number <= -HUGE_VAL || number >= HUGE_VAL))
        {
1105         return NULL;
1106     }
1107     if ((errno && errno != ERANGE) || !is_decimal(*string, end - *
        string)) {
1108         return NULL;
1109     }
1110     *string = end;
1111     return json_value_init_number(number);
1112 }
1113
1114 static JSON_Value * parse_null_value(const char **string) {
1115     size_t token_size = SIZEOF_TOKEN("null");
1116     if (strncmp("null", *string, token_size) == 0) {
1117         *string += token_size;
1118         return json_value_init_null();
1119     }
1120     return NULL;
1121 }
```

```
1122
1123 /* Serialization */
1124
1125 /*  APPEND_STRING() is only called on string literals.
1126     It's a bit hacky because it makes plenty of assumptions about the
         external state
1127     and should eventually be tidied up into a function (same goes for
         APPEND_INDENT)
1128  */
1129 #define APPEND_STRING(str) do {\
1130                                written = SIZEOF_TOKEN((str));\
1131                                if (buf != NULL) {\
1132                                    memcpy(buf, (str), written);\
1133                                    buf[written] = '\0';\
1134                                    buf += written;\
1135                                }\
1136                                written_total += written;\
1137                            } while (0)
1138
1139 #define APPEND_INDENT(level) do {\
1140                                int level_i = 0;\
1141                                for (level_i = 0; level_i < (level);
                                       level_i++) {\
1142                                    APPEND_STRING(PARSON_INDENT_STR);\
1143                                }\
1144                            } while (0)
1145
1146 static int json_serialize_to_buffer_r(const JSON_Value *value, char *
       buf, int level, parson_bool_t is_pretty, char *num_buf)
1147 {
1148     const char *key = NULL, *string = NULL;
1149     JSON_Value *temp_value = NULL;
1150     JSON_Array *array = NULL;
1151     JSON_Object *object = NULL;
1152     size_t i = 0, count = 0;
1153     double num = 0.0;
1154     int written = -1, written_total = 0;
1155     size_t len = 0;
1156
```

```
1157    switch (json_value_get_type(value)) {
1158        case JSONArray:
1159            array = json_value_get_array(value);
1160            count = json_array_get_count(array);
1161            APPEND_STRING("[");
1162            if (count > 0 && is_pretty) {
1163                APPEND_STRING("\n");
1164            }
1165            for (i = 0; i < count; i++) {
1166                if (is_pretty) {
1167                    APPEND_INDENT(level+1);
1168                }
1169                temp_value = json_array_get_value(array, i);
1170                written = json_serialize_to_buffer_r(temp_value, buf,
                        level+1, is_pretty, num_buf);
1171                if (written < 0) {
1172                    return -1;
1173                }
1174                if (buf != NULL) {
1175                    buf += written;
1176                }
1177                written_total += written;
1178                if (i < (count - 1)) {
1179                    APPEND_STRING(",");
1180                }
1181                if (is_pretty) {
1182                    APPEND_STRING("\n");
1183                }
1184            }
1185            if (count > 0 && is_pretty) {
1186                APPEND_INDENT(level);
1187            }
1188            APPEND_STRING("]");
1189            return written_total;
1190        case JSONObject:
1191            object = json_value_get_object(value);
1192            count  = json_object_get_count(object);
1193            APPEND_STRING("{");
1194            if (count > 0 && is_pretty) {
```

```
1195                APPEND_STRING("\n");
1196            }
1197            for (i = 0; i < count; i++) {
1198                key = json_object_get_name(object, i);
1199                if (key == NULL) {
1200                    return -1;
1201                }
1202                if (is_pretty) {
1203                    APPEND_INDENT(level+1);
1204                }
1205                /* We do not support key names with embedded \0 chars
                       */
1206                written = json_serialize_string(key, strlen(key), buf);
1207                if (written < 0) {
1208                    return -1;
1209                }
1210                if (buf != NULL) {
1211                    buf += written;
1212                }
1213                written_total += written;
1214                APPEND_STRING(":");
1215                if (is_pretty) {
1216                    APPEND_STRING(" ");
1217                }
1218                temp_value = json_object_get_value_at(object, i);
1219                written = json_serialize_to_buffer_r(temp_value, buf,
                       level+1, is_pretty, num_buf);
1220                if (written < 0) {
1221                    return -1;
1222                }
1223                if (buf != NULL) {
1224                    buf += written;
1225                }
1226                written_total += written;
1227                if (i < (count - 1)) {
1228                    APPEND_STRING(",");
1229                }
1230                if (is_pretty) {
1231                    APPEND_STRING("\n");
```

```
1232                }
1233            }
1234            if (count > 0 && is_pretty) {
1235                APPEND_INDENT(level);
1236            }
1237            APPEND_STRING("}");
1238            return written_total;
1239        case JSONString:
1240            string = json_value_get_string(value);
1241            if (string == NULL) {
1242                return -1;
1243            }
1244            len = json_value_get_string_len(value);
1245            written = json_serialize_string(string, len, buf);
1246            if (written < 0) {
1247                return -1;
1248            }
1249            if (buf != NULL) {
1250                buf += written;
1251            }
1252            written_total += written;
1253            return written_total;
1254        case JSONBoolean:
1255            if (json_value_get_boolean(value)) {
1256                APPEND_STRING("true");
1257            } else {
1258                APPEND_STRING("false");
1259            }
1260            return written_total;
1261        case JSONNumber:
1262            num = json_value_get_number(value);
1263            if (buf != NULL) {
1264                num_buf = buf;
1265            }
1266            if (parson_number_serialization_function) {
1267                written = parson_number_serialization_function(num,
                        num_buf);
1268            } else {
```

```
1269                const char *float_format = parson_float_format ?
                        parson_float_format : PARSON_DEFAULT_FLOAT_FORMAT;
1270                written = parson_sprintf(num_buf, float_format, num);
1271            }
1272            if (written < 0) {
1273                return -1;
1274            }
1275            if (buf != NULL) {
1276                buf += written;
1277            }
1278            written_total += written;
1279            return written_total;
1280        case JSONNull:
1281            APPEND_STRING("null");
1282            return written_total;
1283        case JSONError:
1284            return -1;
1285        default:
1286            return -1;
1287    }
1288 }
1289
1290 static int json_serialize_string(const char *string, size_t len, char *
        buf) {
1291     size_t i = 0;
1292     char c = '\0';
1293     int written = -1, written_total = 0;
1294     APPEND_STRING("\"");
1295     for (i = 0; i < len; i++) {
1296         c = string[i];
1297         switch (c) {
1298             case '\"': APPEND_STRING("\\\""); break;
1299             case '\\': APPEND_STRING("\\\\"); break;
1300             case '\b': APPEND_STRING("\\b"); break;
1301             case '\f': APPEND_STRING("\\f"); break;
1302             case '\n': APPEND_STRING("\\n"); break;
1303             case '\r': APPEND_STRING("\\r"); break;
1304             case '\t': APPEND_STRING("\\t"); break;
1305             case '\x00': APPEND_STRING("\\u0000"); break;
```

```
1306                case '\x01': APPEND_STRING("\\u0001"); break;
1307                case '\x02': APPEND_STRING("\\u0002"); break;
1308                case '\x03': APPEND_STRING("\\u0003"); break;
1309                case '\x04': APPEND_STRING("\\u0004"); break;
1310                case '\x05': APPEND_STRING("\\u0005"); break;
1311                case '\x06': APPEND_STRING("\\u0006"); break;
1312                case '\x07': APPEND_STRING("\\u0007"); break;
1313                /* '\x08' duplicate: '\b' */
1314                /* '\x09' duplicate: '\t' */
1315                /* '\x0a' duplicate: '\n' */
1316                case '\x0b': APPEND_STRING("\\u000b"); break;
1317                /* '\x0c' duplicate: '\f' */
1318                /* '\x0d' duplicate: '\r' */
1319                case '\x0e': APPEND_STRING("\\u000e"); break;
1320                case '\x0f': APPEND_STRING("\\u000f"); break;
1321                case '\x10': APPEND_STRING("\\u0010"); break;
1322                case '\x11': APPEND_STRING("\\u0011"); break;
1323                case '\x12': APPEND_STRING("\\u0012"); break;
1324                case '\x13': APPEND_STRING("\\u0013"); break;
1325                case '\x14': APPEND_STRING("\\u0014"); break;
1326                case '\x15': APPEND_STRING("\\u0015"); break;
1327                case '\x16': APPEND_STRING("\\u0016"); break;
1328                case '\x17': APPEND_STRING("\\u0017"); break;
1329                case '\x18': APPEND_STRING("\\u0018"); break;
1330                case '\x19': APPEND_STRING("\\u0019"); break;
1331                case '\x1a': APPEND_STRING("\\u001a"); break;
1332                case '\x1b': APPEND_STRING("\\u001b"); break;
1333                case '\x1c': APPEND_STRING("\\u001c"); break;
1334                case '\x1d': APPEND_STRING("\\u001d"); break;
1335                case '\x1e': APPEND_STRING("\\u001e"); break;
1336                case '\x1f': APPEND_STRING("\\u001f"); break;
1337                case '/':
1338                    if (parson_escape_slashes) {
1339                        APPEND_STRING("\\/");  /* to make json embeddable
                            in xml\/html */
1340                    } else {
1341                        APPEND_STRING("/");
1342                    }
1343                    break;
```

```
1344                default:
1345                    if (buf != NULL) {
1346                        buf[0] = c;
1347                        buf += 1;
1348                    }
1349                    written_total += 1;
1350                    break;
1351            }
1352        }
1353        APPEND_STRING("\"");
1354        return written_total;
1355    }
1356
1357    #undef APPEND_STRING
1358    #undef APPEND_INDENT
1359
1360    /* Parser API */
1361    JSON_Value * json_parse_file(const char *filename) {
1362        char *file_contents = read_file(filename);
1363        JSON_Value *output_value = NULL;
1364        if (file_contents == NULL) {
1365            return NULL;
1366        }
1367        output_value = json_parse_string(file_contents);
1368        parson_free(file_contents);
1369        return output_value;
1370    }
1371
1372    JSON_Value * json_parse_file_with_comments(const char *filename) {
1373        char *file_contents = read_file(filename);
1374        JSON_Value *output_value = NULL;
1375        if (file_contents == NULL) {
1376            return NULL;
1377        }
1378        output_value = json_parse_string_with_comments(file_contents);
1379        parson_free(file_contents);
1380        return output_value;
1381    }
1382
```

```
1383 JSON_Value * json_parse_string(const char *string) {
1384     if (string == NULL) {
1385         return NULL;
1386     }
1387     if (string[0] == '\xEF' && string[1] == '\xBB' && string[2] == '\
         xBF') {
1388         string = string + 3; /* Support for UTF-8 BOM */
1389     }
1390     return parse_value((const char**)&string, 0);
1391 }
1392 
1393 JSON_Value * json_parse_string_with_comments(const char *string) {
1394     JSON_Value *result = NULL;
1395     char *string_mutable_copy = NULL, *string_mutable_copy_ptr = NULL;
1396     string_mutable_copy = parson_strdup(string);
1397     if (string_mutable_copy == NULL) {
1398         return NULL;
1399     }
1400     remove_comments(string_mutable_copy, "/*", "*/");
1401     remove_comments(string_mutable_copy, "//", "\n");
1402     string_mutable_copy_ptr = string_mutable_copy;
1403     result = parse_value((const char**)&string_mutable_copy_ptr, 0);
1404     parson_free(string_mutable_copy);
1405     return result;
1406 }
1407 
1408 /* JSON Object API */
1409 
1410 JSON_Value * json_object_get_value(const JSON_Object *object, const
     char *name) {
1411     if (object == NULL || name == NULL) {
1412         return NULL;
1413     }
1414     return json_object_getn_value(object, name, strlen(name));
1415 }
1416 
1417 const char * json_object_get_string(const JSON_Object *object, const
     char *name) {
1418     return json_value_get_string(json_object_get_value(object, name));
```

```
1419 }
1420
1421 size_t json_object_get_string_len(const JSON_Object *object, const char
         *name) {
1422     return json_value_get_string_len(json_object_get_value(object, name
            ));
1423 }
1424
1425 double json_object_get_number(const JSON_Object *object, const char *
        name) {
1426     return json_value_get_number(json_object_get_value(object, name));
1427 }
1428
1429 JSON_Object * json_object_get_object(const JSON_Object *object, const
        char *name) {
1430     return json_value_get_object(json_object_get_value(object, name));
1431 }
1432
1433 JSON_Array * json_object_get_array(const JSON_Object *object, const
        char *name) {
1434     return json_value_get_array(json_object_get_value(object, name));
1435 }
1436
1437 int json_object_get_boolean(const JSON_Object *object, const char *name
        ) {
1438     return json_value_get_boolean(json_object_get_value(object, name));
1439 }
1440
1441 JSON_Value * json_object_dotget_value(const JSON_Object *object, const
        char *name) {
1442     const char *dot_position = strchr(name, '.');
1443     if (!dot_position) {
1444         return json_object_get_value(object, name);
1445     }
1446     object = json_value_get_object(json_object_getn_value(object, name,
            dot_position - name));
1447     return json_object_dotget_value(object, dot_position + 1);
1448 }
1449
```

```
1450 const char * json_object_dotget_string(const JSON_Object *object, const
         char *name) {
1451     return json_value_get_string(json_object_dotget_value(object, name)
             );
1452 }
1453
1454 size_t json_object_dotget_string_len(const JSON_Object *object, const
        char *name) {
1455     return json_value_get_string_len(json_object_dotget_value(object,
             name));
1456 }
1457
1458 double json_object_dotget_number(const JSON_Object *object, const char
        *name) {
1459     return json_value_get_number(json_object_dotget_value(object, name)
             );
1460 }
1461
1462 JSON_Object * json_object_dotget_object(const JSON_Object *object,
        const char *name) {
1463     return json_value_get_object(json_object_dotget_value(object, name)
             );
1464 }
1465
1466 JSON_Array * json_object_dotget_array(const JSON_Object *object, const
        char *name) {
1467     return json_value_get_array(json_object_dotget_value(object, name))
             ;
1468 }
1469
1470 int json_object_dotget_boolean(const JSON_Object *object, const char *
        name) {
1471     return json_value_get_boolean(json_object_dotget_value(object, name
             ));
1472 }
1473
1474 size_t json_object_get_count(const JSON_Object *object) {
1475     return object ? object->count : 0;
1476 }
```

```
1477
1478 const char * json_object_get_name(const JSON_Object *object, size_t
     index) {
1479     if (object == NULL || index >= json_object_get_count(object)) {
1480         return NULL;
1481     }
1482     return object->names[index];
1483 }
1484
1485 JSON_Value * json_object_get_value_at(const JSON_Object *object, size_t
     index) {
1486     if (object == NULL || index >= json_object_get_count(object)) {
1487         return NULL;
1488     }
1489     return object->values[index];
1490 }
1491
1492 JSON_Value *json_object_get_wrapping_value(const JSON_Object *object) {
1493     if (!object) {
1494         return NULL;
1495     }
1496     return object->wrapping_value;
1497 }
1498
1499 int json_object_has_value (const JSON_Object *object, const char *name)
     {
1500     return json_object_get_value(object, name) != NULL;
1501 }
1502
1503 int json_object_has_value_of_type(const JSON_Object *object, const char
     *name, JSON_Value_Type type) {
1504     JSON_Value *val = json_object_get_value(object, name);
1505     return val != NULL && json_value_get_type(val) == type;
1506 }
1507
1508 int json_object_dothas_value (const JSON_Object *object, const char *
     name) {
1509     return json_object_dotget_value(object, name) != NULL;
1510 }
```

```
1511
1512 int json_object_dothas_value_of_type(const JSON_Object *object, const
     char *name, JSON_Value_Type type) {
1513     JSON_Value *val = json_object_dotget_value(object, name);
1514     return val != NULL && json_value_get_type(val) == type;
1515 }
1516
1517 /* JSON Array API */
1518 JSON_Value * json_array_get_value(const JSON_Array *array, size_t index
     ) {
1519     if (array == NULL || index >= json_array_get_count(array)) {
1520         return NULL;
1521     }
1522     return array->items[index];
1523 }
1524
1525 const char * json_array_get_string(const JSON_Array *array, size_t
     index) {
1526     return json_value_get_string(json_array_get_value(array, index));
1527 }
1528
1529 size_t json_array_get_string_len(const JSON_Array *array, size_t index)
     {
1530     return json_value_get_string_len(json_array_get_value(array, index)
        );
1531 }
1532
1533 double json_array_get_number(const JSON_Array *array, size_t index) {
1534     return json_value_get_number(json_array_get_value(array, index));
1535 }
1536
1537 JSON_Object * json_array_get_object(const JSON_Array *array, size_t
     index) {
1538     return json_value_get_object(json_array_get_value(array, index));
1539 }
1540
1541 JSON_Array * json_array_get_array(const JSON_Array *array, size_t index
     ) {
1542     return json_value_get_array(json_array_get_value(array, index));
```

```
1543 }
1544
1545 int json_array_get_boolean(const JSON_Array *array, size_t index) {
1546     return json_value_get_boolean(json_array_get_value(array, index));
1547 }
1548
1549 size_t json_array_get_count(const JSON_Array *array) {
1550     return array ? array->count : 0;
1551 }
1552
1553 JSON_Value * json_array_get_wrapping_value(const JSON_Array *array) {
1554     if (!array) {
1555         return NULL;
1556     }
1557     return array->wrapping_value;
1558 }
1559
1560 /* JSON Value API */
1561 JSON_Value_Type json_value_get_type(const JSON_Value *value) {
1562     return value ? value->type : JSONError;
1563 }
1564
1565 JSON_Object * json_value_get_object(const JSON_Value *value) {
1566     return json_value_get_type(value) == JSONObject ? value->value.
        object : NULL;
1567 }
1568
1569 JSON_Array * json_value_get_array(const JSON_Value *value) {
1570     return json_value_get_type(value) == JSONArray ? value->value.array
         : NULL;
1571 }
1572
1573 static const JSON_String * json_value_get_string_desc(const JSON_Value
    *value) {
1574     return json_value_get_type(value) == JSONString ? &value->value.
        string : NULL;
1575 }
1576
1577 const char * json_value_get_string(const JSON_Value *value) {
```

```
1578      const JSON_String *str = json_value_get_string_desc(value);
1579      return str ? str->chars : NULL;
1580  }
1581
1582  size_t json_value_get_string_len(const JSON_Value *value) {
1583      const JSON_String *str = json_value_get_string_desc(value);
1584      return str ? str->length : 0;
1585  }
1586
1587  double json_value_get_number(const JSON_Value *value) {
1588      return json_value_get_type(value) == JSONNumber ? value->value.
            number : 0;
1589  }
1590
1591  int json_value_get_boolean(const JSON_Value *value) {
1592      return json_value_get_type(value) == JSONBoolean ? value->value.
            boolean : -1;
1593  }
1594
1595  JSON_Value * json_value_get_parent (const JSON_Value *value) {
1596      return value ? value->parent : NULL;
1597  }
1598
1599  void json_value_free(JSON_Value *value) {
1600      switch (json_value_get_type(value)) {
1601          case JSONObject:
1602              json_object_free(value->value.object);
1603              break;
1604          case JSONString:
1605              parson_free(value->value.string.chars);
1606              break;
1607          case JSONArray:
1608              json_array_free(value->value.array);
1609              break;
1610          default:
1611              break;
1612      }
1613      parson_free(value);
1614  }
```

```
1615
1616 JSON_Value * json_value_init_object(void) {
1617     JSON_Value *new_value = (JSON_Value*)parson_malloc(sizeof(
         JSON_Value));
1618     if (!new_value) {
1619         return NULL;
1620     }
1621     new_value->parent = NULL;
1622     new_value->type = JSONObject;
1623     new_value->value.object = json_object_make(new_value);
1624     if (!new_value->value.object) {
1625         parson_free(new_value);
1626         return NULL;
1627     }
1628     return new_value;
1629 }
1630
1631 JSON_Value * json_value_init_array(void) {
1632     JSON_Value *new_value = (JSON_Value*)parson_malloc(sizeof(
         JSON_Value));
1633     if (!new_value) {
1634         return NULL;
1635     }
1636     new_value->parent = NULL;
1637     new_value->type = JSONArray;
1638     new_value->value.array = json_array_make(new_value);
1639     if (!new_value->value.array) {
1640         parson_free(new_value);
1641         return NULL;
1642     }
1643     return new_value;
1644 }
1645
1646 JSON_Value * json_value_init_string(const char *string) {
1647     if (string == NULL) {
1648         return NULL;
1649     }
1650     return json_value_init_string_with_len(string, strlen(string));
1651 }
```

```
1652
1653 JSON_Value * json_value_init_string_with_len(const char *string, size_t
         length) {
1654     char *copy = NULL;
1655     JSON_Value *value;
1656     if (string == NULL) {
1657         return NULL;
1658     }
1659     if (!is_valid_utf8(string, length)) {
1660         return NULL;
1661     }
1662     copy = parson_strndup(string, length);
1663     if (copy == NULL) {
1664         return NULL;
1665     }
1666     value = json_value_init_string_no_copy(copy, length);
1667     if (value == NULL) {
1668         parson_free(copy);
1669     }
1670     return value;
1671 }
1672
1673 JSON_Value * json_value_init_number(double number) {
1674     JSON_Value *new_value = NULL;
1675     if (IS_NUMBER_INVALID(number)) {
1676         return NULL;
1677     }
1678     new_value = (JSON_Value*)parson_malloc(sizeof(JSON_Value));
1679     if (new_value == NULL) {
1680         return NULL;
1681     }
1682     new_value->parent = NULL;
1683     new_value->type = JSONNumber;
1684     new_value->value.number = number;
1685     return new_value;
1686 }
1687
1688 JSON_Value * json_value_init_boolean(int boolean) {
```

```
1689     JSON_Value *new_value = (JSON_Value*)parson_malloc(sizeof(
             JSON_Value));
1690     if (!new_value) {
1691         return NULL;
1692     }
1693     new_value->parent = NULL;
1694     new_value->type = JSONBoolean;
1695     new_value->value.boolean = boolean ? 1 : 0;
1696     return new_value;
1697 }
1698 
1699 JSON_Value * json_value_init_null(void) {
1700     JSON_Value *new_value = (JSON_Value*)parson_malloc(sizeof(
             JSON_Value));
1701     if (!new_value) {
1702         return NULL;
1703     }
1704     new_value->parent = NULL;
1705     new_value->type = JSONNull;
1706     return new_value;
1707 }
1708 
1709 JSON_Value * json_value_deep_copy(const JSON_Value *value) {
1710     size_t i = 0;
1711     JSON_Value *return_value = NULL, *temp_value_copy = NULL, *
             temp_value = NULL;
1712     const JSON_String *temp_string = NULL;
1713     const char *temp_key = NULL;
1714     char *temp_string_copy = NULL;
1715     JSON_Array *temp_array = NULL, *temp_array_copy = NULL;
1716     JSON_Object *temp_object = NULL, *temp_object_copy = NULL;
1717     JSON_Status res = JSONFailure;
1718     char *key_copy = NULL;
1719 
1720     switch (json_value_get_type(value)) {
1721         case JSONArray:
1722             temp_array = json_value_get_array(value);
1723             return_value = json_value_init_array();
1724             if (return_value == NULL) {
```

```
1725                return NULL;
1726            }
1727            temp_array_copy = json_value_get_array(return_value);
1728            for (i = 0; i < json_array_get_count(temp_array); i++) {
1729                temp_value = json_array_get_value(temp_array, i);
1730                temp_value_copy = json_value_deep_copy(temp_value);
1731                if (temp_value_copy == NULL) {
1732                    json_value_free(return_value);
1733                    return NULL;
1734                }
1735                if (json_array_add(temp_array_copy, temp_value_copy) !=
                        JSONSuccess) {
1736                    json_value_free(return_value);
1737                    json_value_free(temp_value_copy);
1738                    return NULL;
1739                }
1740            }
1741            return return_value;
1742        case JSONObject:
1743            temp_object = json_value_get_object(value);
1744            return_value = json_value_init_object();
1745            if (!return_value) {
1746                return NULL;
1747            }
1748            temp_object_copy = json_value_get_object(return_value);
1749            for (i = 0; i < json_object_get_count(temp_object); i++) {
1750                temp_key = json_object_get_name(temp_object, i);
1751                temp_value = json_object_get_value(temp_object,
                       temp_key);
1752                temp_value_copy = json_value_deep_copy(temp_value);
1753                if (!temp_value_copy) {
1754                    json_value_free(return_value);
1755                    return NULL;
1756                }
1757                key_copy = parson_strdup(temp_key);
1758                if (!key_copy) {
1759                    json_value_free(temp_value_copy);
1760                    json_value_free(return_value);
1761                    return NULL;
```

```
1762                }
1763                res = json_object_add(temp_object_copy, key_copy,
                        temp_value_copy);
1764                if (res != JSONSuccess) {
1765                    parson_free(key_copy);
1766                    json_value_free(temp_value_copy);
1767                    json_value_free(return_value);
1768                    return NULL;
1769                }
1770            }
1771            return return_value;
1772        case JSONBoolean:
1773            return json_value_init_boolean(json_value_get_boolean(value
                    ));
1774        case JSONNumber:
1775            return json_value_init_number(json_value_get_number(value))
                    ;
1776        case JSONString:
1777            temp_string = json_value_get_string_desc(value);
1778            if (temp_string == NULL) {
1779                return NULL;
1780            }
1781            temp_string_copy = parson_strndup(temp_string->chars,
                    temp_string->length);
1782            if (temp_string_copy == NULL) {
1783                return NULL;
1784            }
1785            return_value = json_value_init_string_no_copy(
                    temp_string_copy, temp_string->length);
1786            if (return_value == NULL) {
1787                parson_free(temp_string_copy);
1788            }
1789            return return_value;
1790        case JSONNull:
1791            return json_value_init_null();
1792        case JSONError:
1793            return NULL;
1794        default:
1795            return NULL;
```

```
1796     }
1797 }
1798
1799 size_t json_serialization_size(const JSON_Value *value) {
1800     char num_buf[PARSON_NUM_BUF_SIZE]; /* recursively allocating buffer
         on stack is a bad idea, so let's do it only once */
1801     int res = json_serialize_to_buffer_r(value, NULL, 0, PARSON_FALSE,
         num_buf);
1802     return res < 0 ? 0 : (size_t)(res) + 1;
1803 }
1804
1805 JSON_Status json_serialize_to_buffer(const JSON_Value *value, char *buf
    , size_t buf_size_in_bytes) {
1806     int written = -1;
1807     size_t needed_size_in_bytes = json_serialization_size(value);
1808     if (needed_size_in_bytes == 0 || buf_size_in_bytes <
         needed_size_in_bytes) {
1809         return JSONFailure;
1810     }
1811     written = json_serialize_to_buffer_r(value, buf, 0, PARSON_FALSE,
         NULL);
1812     if (written < 0) {
1813         return JSONFailure;
1814     }
1815     return JSONSuccess;
1816 }
1817
1818 JSON_Status json_serialize_to_file(const JSON_Value *value, const char
    *filename) {
1819     JSON_Status return_code = JSONSuccess;
1820     FILE *fp = NULL;
1821     char *serialized_string = json_serialize_to_string(value);
1822     if (serialized_string == NULL) {
1823         return JSONFailure;
1824     }
1825     fp = fopen(filename, "w");
1826     if (fp == NULL) {
1827         json_free_serialized_string(serialized_string);
1828         return JSONFailure;
```

```
1829        }
1830        if (fputs(serialized_string, fp) == EOF) {
1831            return_code = JSONFailure;
1832        }
1833        if (fclose(fp) == EOF) {
1834            return_code = JSONFailure;
1835        }
1836        json_free_serialized_string(serialized_string);
1837        return return_code;
1838    }
1839
1840    char * json_serialize_to_string(const JSON_Value *value) {
1841        JSON_Status serialization_result = JSONFailure;
1842        size_t buf_size_bytes = json_serialization_size(value);
1843        char *buf = NULL;
1844        if (buf_size_bytes == 0) {
1845            return NULL;
1846        }
1847        buf = (char*)parson_malloc(buf_size_bytes);
1848        if (buf == NULL) {
1849            return NULL;
1850        }
1851        serialization_result = json_serialize_to_buffer(value, buf,
               buf_size_bytes);
1852        if (serialization_result != JSONSuccess) {
1853            json_free_serialized_string(buf);
1854            return NULL;
1855        }
1856        return buf;
1857    }
1858
1859    size_t json_serialization_size_pretty(const JSON_Value *value) {
1860        char num_buf[PARSON_NUM_BUF_SIZE]; /* recursively allocating buffer
               on stack is a bad idea, so let's do it only once */
1861        int res = json_serialize_to_buffer_r(value, NULL, 0, PARSON_TRUE,
               num_buf);
1862        return res < 0 ? 0 : (size_t)(res) + 1;
1863    }
1864
```

```
1865 JSON_Status json_serialize_to_buffer_pretty(const JSON_Value *value,
         char *buf, size_t buf_size_in_bytes) {
1866     int written = -1;
1867     size_t needed_size_in_bytes = json_serialization_size_pretty(value)
             ;
1868     if (needed_size_in_bytes == 0 || buf_size_in_bytes <
             needed_size_in_bytes) {
1869         return JSONFailure;
1870     }
1871     written = json_serialize_to_buffer_r(value, buf, 0, PARSON_TRUE,
             NULL);
1872     if (written < 0) {
1873         return JSONFailure;
1874     }
1875     return JSONSuccess;
1876 }
1877
1878 JSON_Status json_serialize_to_file_pretty(const JSON_Value *value,
         const char *filename) {
1879     JSON_Status return_code = JSONSuccess;
1880     FILE *fp = NULL;
1881     char *serialized_string = json_serialize_to_string_pretty(value);
1882     if (serialized_string == NULL) {
1883         return JSONFailure;
1884     }
1885     fp = fopen(filename, "w");
1886     if (fp == NULL) {
1887         json_free_serialized_string(serialized_string);
1888         return JSONFailure;
1889     }
1890     if (fputs(serialized_string, fp) == EOF) {
1891         return_code = JSONFailure;
1892     }
1893     if (fclose(fp) == EOF) {
1894         return_code = JSONFailure;
1895     }
1896     json_free_serialized_string(serialized_string);
1897     return return_code;
1898 }
```

```
1899
1900 char * json_serialize_to_string_pretty(const JSON_Value *value) {
1901     JSON_Status serialization_result = JSONFailure;
1902     size_t buf_size_bytes = json_serialization_size_pretty(value);
1903     char *buf = NULL;
1904     if (buf_size_bytes == 0) {
1905         return NULL;
1906     }
1907     buf = (char*)parson_malloc(buf_size_bytes);
1908     if (buf == NULL) {
1909         return NULL;
1910     }
1911     serialization_result = json_serialize_to_buffer_pretty(value, buf,
        buf_size_bytes);
1912     if (serialization_result != JSONSuccess) {
1913         json_free_serialized_string(buf);
1914         return NULL;
1915     }
1916     return buf;
1917 }
1918
1919 void json_free_serialized_string(char *string) {
1920     parson_free(string);
1921 }
1922
1923 JSON_Status json_array_remove(JSON_Array *array, size_t ix) {
1924     size_t to_move_bytes = 0;
1925     if (array == NULL || ix >= json_array_get_count(array)) {
1926         return JSONFailure;
1927     }
1928     json_value_free(json_array_get_value(array, ix));
1929     to_move_bytes = (json_array_get_count(array) - 1 - ix) * sizeof(
        JSON_Value*);
1930     memmove(array->items + ix, array->items + ix + 1, to_move_bytes);
1931     array->count -= 1;
1932     return JSONSuccess;
1933 }
1934
```

```
1935 JSON_Status json_array_replace_value(JSON_Array *array, size_t ix,
         JSON_Value *value) {
1936     if (array == NULL || value == NULL || value->parent != NULL || ix
             >= json_array_get_count(array)) {
1937         return JSONFailure;
1938     }
1939     json_value_free(json_array_get_value(array, ix));
1940     value->parent = json_array_get_wrapping_value(array);
1941     array->items[ix] = value;
1942     return JSONSuccess;
1943 }
1944
1945 JSON_Status json_array_replace_string(JSON_Array *array, size_t i,
         const char* string) {
1946     JSON_Value *value = json_value_init_string(string);
1947     if (value == NULL) {
1948         return JSONFailure;
1949     }
1950     if (json_array_replace_value(array, i, value) != JSONSuccess) {
1951         json_value_free(value);
1952         return JSONFailure;
1953     }
1954     return JSONSuccess;
1955 }
1956
1957 JSON_Status json_array_replace_string_with_len(JSON_Array *array,
         size_t i, const char *string, size_t len) {
1958     JSON_Value *value = json_value_init_string_with_len(string, len);
1959     if (value == NULL) {
1960         return JSONFailure;
1961     }
1962     if (json_array_replace_value(array, i, value) != JSONSuccess) {
1963         json_value_free(value);
1964         return JSONFailure;
1965     }
1966     return JSONSuccess;
1967 }
1968
```

```
JSON_Status json_array_replace_number(JSON_Array *array, size_t i,
    double number) {
    JSON_Value *value = json_value_init_number(number);
    if (value == NULL) {
        return JSONFailure;
    }
    if (json_array_replace_value(array, i, value) != JSONSuccess) {
        json_value_free(value);
        return JSONFailure;
    }
    return JSONSuccess;
}

JSON_Status json_array_replace_boolean(JSON_Array *array, size_t i, int
    boolean) {
    JSON_Value *value = json_value_init_boolean(boolean);
    if (value == NULL) {
        return JSONFailure;
    }
    if (json_array_replace_value(array, i, value) != JSONSuccess) {
        json_value_free(value);
        return JSONFailure;
    }
    return JSONSuccess;
}

JSON_Status json_array_replace_null(JSON_Array *array, size_t i) {
    JSON_Value *value = json_value_init_null();
    if (value == NULL) {
        return JSONFailure;
    }
    if (json_array_replace_value(array, i, value) != JSONSuccess) {
        json_value_free(value);
        return JSONFailure;
    }
    return JSONSuccess;
}

JSON_Status json_array_clear(JSON_Array *array) {
```

```
    size_t i = 0;
    if (array == NULL) {
        return JSONFailure;
    }
    for (i = 0; i < json_array_get_count(array); i++) {
        json_value_free(json_array_get_value(array, i));
    }
    array->count = 0;
    return JSONSuccess;
}

JSON_Status json_array_append_value(JSON_Array *array, JSON_Value *
    value) {
    if (array == NULL || value == NULL || value->parent != NULL) {
        return JSONFailure;
    }
    return json_array_add(array, value);
}

JSON_Status json_array_append_string(JSON_Array *array, const char *
    string) {
    JSON_Value *value = json_value_init_string(string);
    if (value == NULL) {
        return JSONFailure;
    }
    if (json_array_append_value(array, value) != JSONSuccess) {
        json_value_free(value);
        return JSONFailure;
    }
    return JSONSuccess;
}

JSON_Status json_array_append_string_with_len(JSON_Array *array, const
    char *string, size_t len) {
    JSON_Value *value = json_value_init_string_with_len(string, len);
    if (value == NULL) {
        return JSONFailure;
    }
    if (json_array_append_value(array, value) != JSONSuccess) {
```

```
2042            json_value_free(value);
2043            return JSONFailure;
2044        }
2045        return JSONSuccess;
2046    }
2047
2048    JSON_Status json_array_append_number(JSON_Array *array, double number)
        {
2049        JSON_Value *value = json_value_init_number(number);
2050        if (value == NULL) {
2051            return JSONFailure;
2052        }
2053        if (json_array_append_value(array, value) != JSONSuccess) {
2054            json_value_free(value);
2055            return JSONFailure;
2056        }
2057        return JSONSuccess;
2058    }
2059
2060    JSON_Status json_array_append_boolean(JSON_Array *array, int boolean) {
2061        JSON_Value *value = json_value_init_boolean(boolean);
2062        if (value == NULL) {
2063            return JSONFailure;
2064        }
2065        if (json_array_append_value(array, value) != JSONSuccess) {
2066            json_value_free(value);
2067            return JSONFailure;
2068        }
2069        return JSONSuccess;
2070    }
2071
2072    JSON_Status json_array_append_null(JSON_Array *array) {
2073        JSON_Value *value = json_value_init_null();
2074        if (value == NULL) {
2075            return JSONFailure;
2076        }
2077        if (json_array_append_value(array, value) != JSONSuccess) {
2078            json_value_free(value);
2079            return JSONFailure;
```

```
2080        }
2081        return JSONSuccess;
2082    }
2083
2084    JSON_Status json_object_set_value(JSON_Object *object, const char *name
        , JSON_Value *value) {
2085        unsigned long hash = 0;
2086        parson_bool_t found = PARSON_FALSE;
2087        size_t cell_ix = 0;
2088        size_t item_ix = 0;
2089        JSON_Value *old_value = NULL;
2090        char *key_copy = NULL;
2091
2092        if (!object || !name || !value || value->parent) {
2093            return JSONFailure;
2094        }
2095        hash = hash_string(name, strlen(name));
2096        found = PARSON_FALSE;
2097        cell_ix = json_object_get_cell_ix(object, name, strlen(name), hash,
            &found);
2098        if (found) {
2099            item_ix = object->cells[cell_ix];
2100            old_value = object->values[item_ix];
2101            json_value_free(old_value);
2102            object->values[item_ix] = value;
2103            value->parent = json_object_get_wrapping_value(object);
2104            return JSONSuccess;
2105        }
2106        if (object->count >= object->item_capacity) {
2107            JSON_Status res = json_object_grow_and_rehash(object);
2108            if (res != JSONSuccess) {
2109                return JSONFailure;
2110            }
2111            cell_ix = json_object_get_cell_ix(object, name, strlen(name),
                hash, &found);
2112        }
2113        key_copy = parson_strdup(name);
2114        if (!key_copy) {
2115            return JSONFailure;
```

```
2116        }
2117        object->names[object->count] = key_copy;
2118        object->cells[cell_ix] = object->count;
2119        object->values[object->count] = value;
2120        object->cell_ixs[object->count] = cell_ix;
2121        object->hashes[object->count] = hash;
2122        object->count++;
2123        value->parent = json_object_get_wrapping_value(object);
2124        return JSONSuccess;
2125    }
2126
2127    JSON_Status json_object_set_string(JSON_Object *object, const char *
        name, const char *string) {
2128        JSON_Value *value = json_value_init_string(string);
2129        JSON_Status status = json_object_set_value(object, name, value);
2130        if (status != JSONSuccess) {
2131            json_value_free(value);
2132        }
2133        return status;
2134    }
2135
2136    JSON_Status json_object_set_string_with_len(JSON_Object *object, const
        char *name, const char *string, size_t len) {
2137        JSON_Value *value = json_value_init_string_with_len(string, len);
2138        JSON_Status status = json_object_set_value(object, name, value);
2139        if (status != JSONSuccess) {
2140            json_value_free(value);
2141        }
2142        return status;
2143    }
2144
2145    JSON_Status json_object_set_number(JSON_Object *object, const char *
        name, double number) {
2146        JSON_Value *value = json_value_init_number(number);
2147        JSON_Status status = json_object_set_value(object, name, value);
2148        if (status != JSONSuccess) {
2149            json_value_free(value);
2150        }
2151        return status;
```

```
2152 }
2153
2154 JSON_Status json_object_set_boolean(JSON_Object *object, const char *
         name, int boolean) {
2155     JSON_Value *value = json_value_init_boolean(boolean);
2156     JSON_Status status = json_object_set_value(object, name, value);
2157     if (status != JSONSuccess) {
2158         json_value_free(value);
2159     }
2160     return status;
2161 }
2162
2163 JSON_Status json_object_set_null(JSON_Object *object, const char *name)
         {
2164     JSON_Value *value = json_value_init_null();
2165     JSON_Status status = json_object_set_value(object, name, value);
2166     if (status != JSONSuccess) {
2167         json_value_free(value);
2168     }
2169     return status;
2170 }
2171
2172 JSON_Status json_object_dotset_value(JSON_Object *object, const char *
         name, JSON_Value *value) {
2173     const char *dot_pos = NULL;
2174     JSON_Value *temp_value = NULL, *new_value = NULL;
2175     JSON_Object *temp_object = NULL, *new_object = NULL;
2176     JSON_Status status = JSONFailure;
2177     size_t name_len = 0;
2178     char *name_copy = NULL;
2179
2180     if (object == NULL || name == NULL || value == NULL) {
2181         return JSONFailure;
2182     }
2183     dot_pos = strchr(name, '.');
2184     if (dot_pos == NULL) {
2185         return json_object_set_value(object, name, value);
2186     }
2187     name_len = dot_pos - name;
```

```
2188        temp_value = json_object_getn_value(object, name, name_len);
2189        if (temp_value) {
2190            /* Don't overwrite existing non-object (unlike
                   json_object_set_value, but it shouldn't be changed at this
                   point) */
2191            if (json_value_get_type(temp_value) != JSONObject) {
2192                return JSONFailure;
2193            }
2194            temp_object = json_value_get_object(temp_value);
2195            return json_object_dotset_value(temp_object, dot_pos + 1, value
                   );
2196        }
2197        new_value = json_value_init_object();
2198        if (new_value == NULL) {
2199            return JSONFailure;
2200        }
2201        new_object = json_value_get_object(new_value);
2202        status = json_object_dotset_value(new_object, dot_pos + 1, value);
2203        if (status != JSONSuccess) {
2204            json_value_free(new_value);
2205            return JSONFailure;
2206        }
2207        name_copy = parson_strndup(name, name_len);
2208        if (!name_copy) {
2209            json_object_dotremove_internal(new_object, dot_pos + 1, 0);
2210            json_value_free(new_value);
2211            return JSONFailure;
2212        }
2213        status = json_object_add(object, name_copy, new_value);
2214        if (status != JSONSuccess) {
2215            parson_free(name_copy);
2216            json_object_dotremove_internal(new_object, dot_pos + 1, 0);
2217            json_value_free(new_value);
2218            return JSONFailure;
2219        }
2220        return JSONSuccess;
2221    }
2222
```

```
2223 JSON_Status json_object_dotset_string(JSON_Object *object, const char *
        name, const char *string) {
2224     JSON_Value *value = json_value_init_string(string);
2225     if (value == NULL) {
2226         return JSONFailure;
2227     }
2228     if (json_object_dotset_value(object, name, value) != JSONSuccess) {
2229         json_value_free(value);
2230         return JSONFailure;
2231     }
2232     return JSONSuccess;
2233 }
2234
2235 JSON_Status json_object_dotset_string_with_len(JSON_Object *object,
        const char *name, const char *string, size_t len) {
2236     JSON_Value *value = json_value_init_string_with_len(string, len);
2237     if (value == NULL) {
2238         return JSONFailure;
2239     }
2240     if (json_object_dotset_value(object, name, value) != JSONSuccess) {
2241         json_value_free(value);
2242         return JSONFailure;
2243     }
2244     return JSONSuccess;
2245 }
2246
2247 JSON_Status json_object_dotset_number(JSON_Object *object, const char *
        name, double number) {
2248     JSON_Value *value = json_value_init_number(number);
2249     if (value == NULL) {
2250         return JSONFailure;
2251     }
2252     if (json_object_dotset_value(object, name, value) != JSONSuccess) {
2253         json_value_free(value);
2254         return JSONFailure;
2255     }
2256     return JSONSuccess;
2257 }
2258
```

```
2259 JSON_Status json_object_dotset_boolean(JSON_Object *object, const char
        *name, int boolean) {
2260     JSON_Value *value = json_value_init_boolean(boolean);
2261     if (value == NULL) {
2262         return JSONFailure;
2263     }
2264     if (json_object_dotset_value(object, name, value) != JSONSuccess) {
2265         json_value_free(value);
2266         return JSONFailure;
2267     }
2268     return JSONSuccess;
2269 }
2270
2271 JSON_Status json_object_dotset_null(JSON_Object *object, const char *
        name) {
2272     JSON_Value *value = json_value_init_null();
2273     if (value == NULL) {
2274         return JSONFailure;
2275     }
2276     if (json_object_dotset_value(object, name, value) != JSONSuccess) {
2277         json_value_free(value);
2278         return JSONFailure;
2279     }
2280     return JSONSuccess;
2281 }
2282
2283 JSON_Status json_object_remove(JSON_Object *object, const char *name) {
2284     return json_object_remove_internal(object, name, PARSON_TRUE);
2285 }
2286
2287 JSON_Status json_object_dotremove(JSON_Object *object, const char *name
        ) {
2288     return json_object_dotremove_internal(object, name, PARSON_TRUE);
2289 }
2290
2291 JSON_Status json_object_clear(JSON_Object *object) {
2292     size_t i = 0;
2293     if (object == NULL) {
2294         return JSONFailure;
```

```
2295        }
2296        for (i = 0; i < json_object_get_count(object); i++) {
2297            parson_free(object->names[i]);
2298            object->names[i] = NULL;
2299
2300            json_value_free(object->values[i]);
2301            object->values[i] = NULL;
2302        }
2303        object->count = 0;
2304        for (i = 0; i < object->cell_capacity; i++) {
2305            object->cells[i] = OBJECT_INVALID_IX;
2306        }
2307        return JSONSuccess;
2308    }
2309
2310    JSON_Status json_validate(const JSON_Value *schema, const JSON_Value *
        value) {
2311        JSON_Value *temp_schema_value = NULL, *temp_value = NULL;
2312        JSON_Array *schema_array = NULL, *value_array = NULL;
2313        JSON_Object *schema_object = NULL, *value_object = NULL;
2314        JSON_Value_Type schema_type = JSONError, value_type = JSONError;
2315        const char *key = NULL;
2316        size_t i = 0, count = 0;
2317        if (schema == NULL || value == NULL) {
2318            return JSONFailure;
2319        }
2320        schema_type = json_value_get_type(schema);
2321        value_type = json_value_get_type(value);
2322        if (schema_type != value_type && schema_type != JSONNull) { /* null
            represents all values */
2323            return JSONFailure;
2324        }
2325        switch (schema_type) {
2326            case JSONArray:
2327                schema_array = json_value_get_array(schema);
2328                value_array = json_value_get_array(value);
2329                count = json_array_get_count(schema_array);
2330                if (count == 0) {
2331                    return JSONSuccess; /* Empty array allows all types */
```

```
2332            }
2333            /* Get first value from array, rest is ignored */
2334            temp_schema_value = json_array_get_value(schema_array, 0);
2335            for (i = 0; i < json_array_get_count(value_array); i++) {
2336                temp_value = json_array_get_value(value_array, i);
2337                if (json_validate(temp_schema_value, temp_value) !=
                        JSONSuccess) {
2338                    return JSONFailure;
2339                }
2340            }
2341            return JSONSuccess;
2342        case JSONObject:
2343            schema_object = json_value_get_object(schema);
2344            value_object = json_value_get_object(value);
2345            count = json_object_get_count(schema_object);
2346            if (count == 0) {
2347                return JSONSuccess; /* Empty object allows all objects
                        */
2348            } else if (json_object_get_count(value_object) < count) {
2349                return JSONFailure; /* Tested object mustn't have less
                        name-value pairs than schema */
2350            }
2351            for (i = 0; i < count; i++) {
2352                key = json_object_get_name(schema_object, i);
2353                temp_schema_value = json_object_get_value(schema_object
                        , key);
2354                temp_value = json_object_get_value(value_object, key);
2355                if (temp_value == NULL) {
2356                    return JSONFailure;
2357                }
2358                if (json_validate(temp_schema_value, temp_value) !=
                        JSONSuccess) {
2359                    return JSONFailure;
2360                }
2361            }
2362            return JSONSuccess;
2363        case JSONString: case JSONNumber: case JSONBoolean: case
               JSONNull:
```

```
2364                return JSONSuccess; /* equality already tested before
                        switch */
2365            case JSONError: default:
2366                return JSONFailure;
2367        }
2368    }
2369
2370    int json_value_equals(const JSON_Value *a, const JSON_Value *b) {
2371        JSON_Object *a_object = NULL, *b_object = NULL;
2372        JSON_Array *a_array = NULL, *b_array = NULL;
2373        const JSON_String *a_string = NULL, *b_string = NULL;
2374        const char *key = NULL;
2375        size_t a_count = 0, b_count = 0, i = 0;
2376        JSON_Value_Type a_type, b_type;
2377        a_type = json_value_get_type(a);
2378        b_type = json_value_get_type(b);
2379        if (a_type != b_type) {
2380            return PARSON_FALSE;
2381        }
2382        switch (a_type) {
2383            case JSONArray:
2384                a_array = json_value_get_array(a);
2385                b_array = json_value_get_array(b);
2386                a_count = json_array_get_count(a_array);
2387                b_count = json_array_get_count(b_array);
2388                if (a_count != b_count) {
2389                    return PARSON_FALSE;
2390                }
2391                for (i = 0; i < a_count; i++) {
2392                    if (!json_value_equals(json_array_get_value(a_array, i)
                            ,
2393                                           json_array_get_value(b_array, i)
                                               )) {
2394                        return PARSON_FALSE;
2395                    }
2396                }
2397                return PARSON_TRUE;
2398            case JSONObject:
2399                a_object = json_value_get_object(a);
```

```
2400            b_object = json_value_get_object(b);
2401            a_count = json_object_get_count(a_object);
2402            b_count = json_object_get_count(b_object);
2403            if (a_count != b_count) {
2404                return PARSON_FALSE;
2405            }
2406            for (i = 0; i < a_count; i++) {
2407                key = json_object_get_name(a_object, i);
2408                if (!json_value_equals(json_object_get_value(a_object,
                       key),
2409                                       json_object_get_value(b_object,
                                          key))) {
2410                    return PARSON_FALSE;
2411                }
2412            }
2413            return PARSON_TRUE;
2414        case JSONString:
2415            a_string = json_value_get_string_desc(a);
2416            b_string = json_value_get_string_desc(b);
2417            if (a_string == NULL || b_string == NULL) {
2418                return PARSON_FALSE; /* shouldn't happen */
2419            }
2420            return a_string->length == b_string->length &&
2421                   memcmp(a_string->chars, b_string->chars, a_string->
                          length) == 0;
2422        case JSONBoolean:
2423            return json_value_get_boolean(a) == json_value_get_boolean(
                   b);
2424        case JSONNumber:
2425            return fabs(json_value_get_number(a) -
                   json_value_get_number(b)) < 0.000001; /* EPSILON */
2426        case JSONError:
2427            return PARSON_TRUE;
2428        case JSONNull:
2429            return PARSON_TRUE;
2430        default:
2431            return PARSON_TRUE;
2432    }
2433 }
```

```
2434
2435 JSON_Value_Type json_type(const JSON_Value *value) {
2436     return json_value_get_type(value);
2437 }
2438
2439 JSON_Object * json_object (const JSON_Value *value) {
2440     return json_value_get_object(value);
2441 }
2442
2443 JSON_Array * json_array(const JSON_Value *value) {
2444     return json_value_get_array(value);
2445 }
2446
2447 const char * json_string(const JSON_Value *value) {
2448     return json_value_get_string(value);
2449 }
2450
2451 size_t json_string_len(const JSON_Value *value) {
2452     return json_value_get_string_len(value);
2453 }
2454
2455 double json_number(const JSON_Value *value) {
2456     return json_value_get_number(value);
2457 }
2458
2459 int json_boolean(const JSON_Value *value) {
2460     return json_value_get_boolean(value);
2461 }
2462
2463 void json_set_allocation_functions(JSON_Malloc_Function malloc_fun,
        JSON_Free_Function free_fun) {
2464     parson_malloc = malloc_fun;
2465     parson_free = free_fun;
2466 }
2467
2468 void json_set_escape_slashes(int escape_slashes) {
2469     parson_escape_slashes = escape_slashes;
2470 }
2471
```

```
2472 void json_set_float_serialization_format(const char *format) {
2473     if (parson_float_format) {
2474         parson_free(parson_float_format);
2475         parson_float_format = NULL;
2476     }
2477     if (!format) {
2478         parson_float_format = NULL;
2479         return;
2480     }
2481     parson_float_format = parson_strdup(format);
2482 }
2483 
2484 void json_set_number_serialization_function(
        JSON_Number_Serialization_Function func) {
2485     parson_number_serialization_function = func;
2486 }
```