

**Universidade do Minho**

Escola de Engenharia, Departamento de Informática

LEI 3º ano – Comunicações por Computador

“Music Game”

**Desenho e implementação de um jogo distribuído na Internet**

**Abstract.**

Este relatório visa a documentação de todo o processo e decisões implementadas durante a construção e implementação de um serviço de distribuição de conteúdos por transmissão de informação através do uso de *sockets* TCP e UDP. Os servidores terão acesso à informação através de comunicação por *sockets* TCP e, os clientes, receberão os dados enviados pelo servidor por *sockets* UDP. O desafio é, então, implementar um procedimento fiável, com controlo de fluxo e de erros, que possibilite a comunicação entre os diferentes participantes do jogo (servidores e clientes).

Bruno Pereira, 69303 João Mano 69854 Patrícia Rocha 67636

1. Introdução

O avanço tecnológico permitiu a introdução de aplicações únicas suportadas pela internet, que permitem o fornecimento de um dado serviço. Nestas aplicações, a distribuição de informação dos servidores para os clientes (habitualmente sempre em maior número), é das maiores preocupações a ter aquando o desenvolvimento dos programas.

Para que a comunicação entre as diversas entidades envolvidas seja possível e para que a sua eficácia e desempenho seja também, tão elevada quanto possível, é necessário projetar, instalar e operar redes otimizadas de comunicação que têm em conta paradigmas como a fiabilidade, disponibilidade, atrasos e débitos dos serviços fornecidos.

Assim sendo, como objetivo deste projeto, será construído uma aplicação que fornecerá um jogo sobre música a todos os seus clientes, implementando elementos como rankings, pontuações, desafios entre jogadores, etc. Esta aplicação, sendo suportada pela internet, exige a construção de um protocolo de comunicação que será então o objetivo principal, ao longo deste documento será explicitado o processo de construção do mesmo.

1. **Arquitetura da Aplicação**

A solução desenvolvida implementa a seguinte estrutura, existe um servidor denominado MusicServer que, quando é inicializado “parte-se” em duas threads de escuta, a thread AtendimentoCliente que estará sempre à escuta de pedidos UDP, direcionada, como o próprio nome indica, para atender clientes, e a thread AtendimentoServidor que estará à escuta para pedidos TCP, de outros servidores. Esta última thread ainda se encarrega de, caso o servidor não seja primário, registar-se perante o servidor primário.

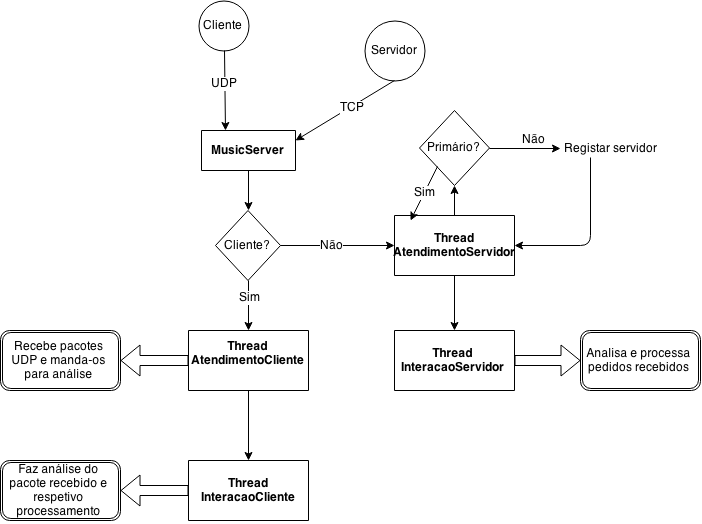


Figura 1 – Processamento de pedidos

Após a criação das threads, sempre que é recebido um pacote, visto que é preciso estar sempre “à escuta”, são criadas a thread de análise e processamento dos pedidos, denominadas InteracaoCliente para pedidos provenientes dos clientes e, InteracaoServidor para pedidos provenientes de outros servidores.

A estrutura utilizada para armazenar os dados, denominada BD é constituída por:

* Um HashMap de Utilizadores (chave é a alcunha);
* Um HashMap para o ranking local e outro para o ranking global (mapeiam a alcunha de um jogador para a sua pontuação local ou global respetivamente);
* Um HashMap de desafios locais e outro para os desafios globais (mapeiam a string com o nome do desafio para a data de início);
* Um HashMap listaDesafiosServidores para conseguir associar ao nome do desafio não local o InetAddress e a porta do servidor a quem é preciso pedir as perguntas;
* Um ArrayList de perguntas;
* Um HashMap de servidores (mapeia um InetAddress de um servidor à sua porta);
* Uma string com o nome da pasta de músicas e outra com o nome da pasta de imagens.

Esta estrutura é comum a todos os servidores e a todas as threads pertencentes ao servidor (threads de atendimento, processamento).

Note-se que o uso de threads é mais que necessário para a “libertação” dos servidores, de modo a que estes estejam constantemente a receber pedidos e a reencaminhá-los para outra thread, tal como um rececionista, caso assim não fosse o servidor teria uma capacidade muito menor de atender pedidos!

* 1. **PDU**

A classe PDU representa o pacote que permite a comunicação entre os diversos componentes.

Falta aqui!!!!!!!!!!!!!!!!!!!!

* 1. **Cliente**

Após a entrada de um pedido na thread AtendimentoCliente é criada então, tal como já foi referido, a thread InteracaoCliente. Esta thread analisa o tipo de pedido recebido (login, registo, aceitar jogo, etc.) e encarrega-se de tratar do seu processamento.

De todos os diversos pedidos abordaremos apenas aqueles que achamos de maior relevância, outros como Login, Registo, etc. são bastante simples e por isso não mencionados.

* + 1. **Jogar Desafio**

Tanto a aceitação como a criação de um desafio implicam a criação de uma nova thread, Jogo, para fazer os clientes aguardarem pelo jogo.

Ao criar um desafio é introduzido o desafio localmente (no HashMap de desafios locais e no de globais) e enviada a informação de um novo desafio para os restantes servidores, estes vão atualizar a estrutura de desafios globais. É ainda introduzido o jogador no desafio.

De seguida, no servidor, é criada a thread Jogo que, espera até à hora do desafio chegar e, posteriormente envia a primeira pergunta do desafio ao jogador respetivo se estiver tudo bem. Isto acontece também no pedido de aceitar desafio.

Assim quando o utilizador recebe a pergunta espera até receber o último pacote e realiza uma validação para verificar se todos os blocos foram recebidos, caso não tenham sido, é pedido um retransmit do bloco em falta.

Tanto os pedidos de retransmit como as respostas às questões são recebidas pela thread de escuta, AtendimentoCliente reencaminhadas à thread InteracaoCliente e, caso seja resposta à pergunta e não seja a última é enviada a nova pergunta e o processo repetido, caso seja a última é esperado que todos os utilizadores terminem e então enviadas as cotações do desafio para todos os utilizadores.

Utilizamos uma thread neste ponto para, de alguma forma, estruturar a aplicação, assim esta thread torna-se responsável por fazer a espera do utilizador e iniciar o jogo.

* + 1. **Controlo de perdas**

O controlo de perdas de pacote é feito na parte do cliente, sempre que existe um timeout, isto é quando uma resposta não chega entre um determinado período de tempo, é lançado uma TimeOutException que é apanhada e despoleta um retransmit do último pacote enviado. Desta forma é requerido que o último pacote enviado seja armazenado por parte do cliente.

Note-se que o retransmit não é infinito, ou seja, após um número estipulado de retransmit’s não se volta a pedir uma retransmissão e é lançada a exceção ServerUnreachable.

Ao retransmitir o pacote enviado será obtida uma nova resposta do servidor.

* + 1. **Gestão de Concorrência**

A gestão de concorrência é feita com a aplicação de synchronized aos métodos de alteração de conteúdo das estruturas partilhadas.

Apesar de não ser a solução mais eficaz é uma solução funcional e rápida na execução/implementação.

* 1. **Servidor**

Falta aqui!!!!!!!!!!!!!!!!!!!!

* 1. **Outras Estruturas**

1. **Mudanças em Relação ao Enunciado**

Foram aplicadas algumas alterações em relação ao enunciado, nomeadamente:

* As datas dos desafios só utilizam 5 bytes, 3 para o ano, 1 para o mês e 1 para o dia.
* As horas apenas usam 3 bytes, um para cada campo (hora, minuto, segundo).

As últimas duas mudanças foram executadas para poupar espaço.

Alterou-se ainda na transferência de informação de servidor para servidor (código 14) foram associados códigos às diferentes intenções nos pedidos:

* Registo de um novo servidor: 32
* Registo de novo desafio: 33
* Lista de desafios disponíveis localmente: 34
* Registo de aceitação do desafio: 35
* Resultados de um desafio: 36
* Ranking dos utilizadores locais: 37
* Registar um servidor sem resposta: 38

Esta mudança surgiu da necessidade de mais facilmente se conseguir processar os pacotes recebido entre as comunicações servidor – servidor.

1. **Implementação**
   1. **Cliente -> Servidor**

É importante mencionar que foi criada uma interface em JavaFX para tornar o programa mais apelativo ao uso. Assim sendo passaremos agora à explicação de como está implementada a classe que corresponde ao cliente, MusicClient.

Inicialmente é enviado com o **menuInit** um hello ao servidor e recebida a resposta, de seguida são apresentadas as opções de autenticação ou registo, para autenticação tem-se a função **menuLogin** que envia um pacote com a alcunha e a password e recebe uma resposta de erro ou de validação com os restantes parâmetros do utilizador (pontuação, nome, etc.). Para registo tem-se a função **menuRegista** que envia os parâmetros de registo (nome, alcunha, password) espera uma resposta e, retorna um valor inteiro igual a 0 se houver erro e 1 caso o registo seja bem-sucedido.

Por forma a explicar o que realizam as restantes funções serão, de seguida, enumeradas e explicadas:

1. **menuLogout:** função que envia um pacote de logout ao servidor e capta a resposta.
2. **receivePDU:** função que se encarrega de receber um pacote do servidor e, caso exista um timeout volta a enviar o pacote, para causar uma nova resposta do servidor.
3. **receivePDUNoException:** esta função é exatamente igual à anterior mas, sem pedido de retransmit caso não receba resposta, surgiu para resolver o seguinte caso: imagine-se que um jogador é muito rápido e termina o jogo e espera por uma resposta não deveria ser posto um tempo mínimo de resposta mas sim exigido que esperasse o tempo que fosse preciso, isto poderá ser prejudicial caso o pacote se perda, exigindo ao fecho da aplicação.
4. **sendPDU:** função que recebe um conjunto de dados, constrói um pacote e envia-o ao servidor.
5. **menuQuit:** esta função envia um pacote que identifica a desistência de um desafio.
6. **menuEnd:** envia um pacote que simboliza o término do desafio por parte deste utilizador e recebe as pontuações do desafio.
7. **menuDelete: Falta ver se funciona!!!!!!!!!**
8. **answer:** envia a resposta ao servidor, recebe a informação sobre os pontos conseguidos, qual a resposta certa, etc. e mostra-a ao utilizador.
9. **proximaPergunta:** pede ao servidor que envie a próxima pergunta e manda-a apresentar ao utilizador.
10. **menuListRankings:** envia o pedido de listar os rankings e vai construindo a lista com a ajuda da **getNextUserRanking** que capta a pontuação do próximo utilizador.
11. **menuListChallenge:** semelhante à anterior mas direcionada para os desafios e com a ajuda da **getNextDesafio**.
12. **acceptChallenge:** envia ao servidor a informação de que o cliente quer aceitar um dado desafio.
13. **menuMakeChallenge:** envia ao servidor a informação de que o utilizador quer criar um novo desafio e todos os dados necessários desse desafio (nome).
14. **Jogar:** encarrega-se de receber a informação de uma questão, a música e imagem são recebidas pelo uso da função **recebeBlocos** e, logo de seguida é chamada a função **checkBlocos** que verifica se todos os blocos de música e imagem foram bem recebidos e, caso não tenham sido é utilizada a função **askBlockRetransmit,** que envia ao servidor um pedido de retransmissão de um dado bloco de música ou imagem e recebe-o armazenando-o no devido lugar.
15. **constroiFicheiroAudio:** tal como o nome indica, esta função constrói o ficheiro áudio que pega nos blocos recebidos, anteriormente armazenados num treemap, e junta-os num único byte array e cria o ficheiro áudio numa pasta temporária.
16. **constroiFicheiroImagem:** idêntica à anterior mas para a imagem.
    1. **Servidor -> Cliente**

Nesta parte do relatório serão documentadas as funções que o servidor utiliza para responder ao cliente, ou seja, as funções implementadas na classe InteracaoCliente.

1. **analisaPacote:** função que analisa o pacote recebido, dependendo do tipo de pedido, o pacote é processado de diferentes maneiras.
2. **quitDesafio:** esta função recebe e processa pacotes de desistência de um desafio, ou seja, retira o dado utilizador do desafio e este não ganha nem perde pontos.
3. **fimDesafio:** responsável pela atualização das pontuações e envio das cotações do desafio aos utilizadores.
4. **processaLogin:** verifica se os dados de login estão corretos e envia ou um pacote de erro caso não esteja tudo bem ou então um pacote com o InetAdress e com a porta associada ao utilizador.
5. **processaRegisto:** recebe as informações de registo se não houverem conflitos regista o novo utilizador e envia um pacote de OK, senão envia um pacote de erro e o utilizador não é registado.
6. **listaDesafios:** envia um pacote por cada desafio existente ainda disponível para jogar, com a indicação de continua ou não caso não existam mais desafios a enviar. Se não existirem desafios a listar é enviado um pacote de erro correspondendo a não existem desafios.
7. **listaRanking:** funciona exatamente igual à anterior mas para os utilizadores existentes em vez dos desafios, com a exceção de nunca ser enviado pacote de erro porque existe sempre pelo menos um pacote a enviar (o do próprio utilizador).
8. **criaDesafio:** cria aqui um desafio com os dados recebidos, caso já exista é enviado um pacote de erro com a mensagem de desafio existente, caso seja bem-sucedida a criação, é enviado um pacote com os dados do desafio incluindo a data de início e, posteriormente, criada a thread jogo.
   1. **Servidor->Servidor**

**(detalhes, parâmetros, bibliotecas, funções…)**

1. **Testes e Resultados**
2. **Conclusões e Trabalho Futuro**