Mestrado em Engenharia Informática e Computação

Sistemas Distribuídos

2014-2015

Jogo Cliente-Servidor Multi-jogador
(Relatório do segundo projeto)

SDIS1415-T4G06-A2:
João Tiago Arriscado da Silva Cavaleiro | ei01099@fe.up.pt | 200004508
Luís Carlos Branco Amaro | up201306622@fe.up.pt | 201306622
Luís Eduardo de Magalhães Reis | ei12085@fe.up.pt | 201200688
Susana Isabel do Vale Ventura de Sousa | ei12009@fe.up.pt | 201207062

5 de Junho de 2015

Conteúdos

1. Introdução

O projeto descrito neste relatorio consistíu no desenvolvimento de uma aplicação distribuída, mais especificamente de um jogo multi-jogador, com uma arquitetura cliente-servidor, que permite aos utilizadores interagírem num ambiente sincronizado e altamente responsivo.

O jogo em si consiste na condução de pequenos tanques num ambiente 2D com alguns obstáculos, onde o objetivo é destruir os adversários disparando sobre eles. O jogo tem uma interface gráfica muito simples, pois o principal propósito do projeto foi estudar e desenvolver o back-end necessário para a aplicação distribuída.

Foi ainda desenvolvido um serviço externo para autenticação dos jogadores e listagem de servidores disponíveis.

A aplicação principal foi desenvolvida em Java e pode ser corrida em PCs e dispositivos móveis Android. O serviço externo foi desenvolvido em PHP e utiliza uma pequena base de dados criada com SQLite.

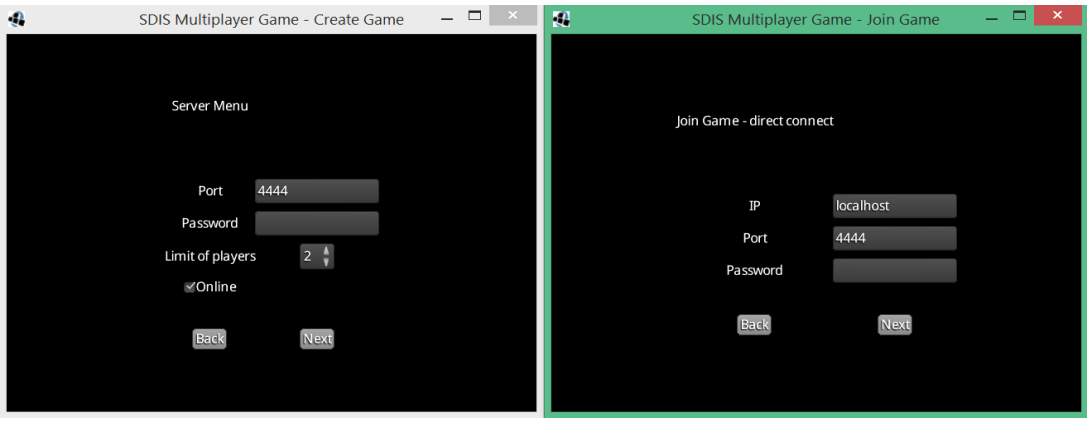

Figura 1: Interface da aplicação ao iniciar um jogo (servidor à esquerda e cliente à direita)

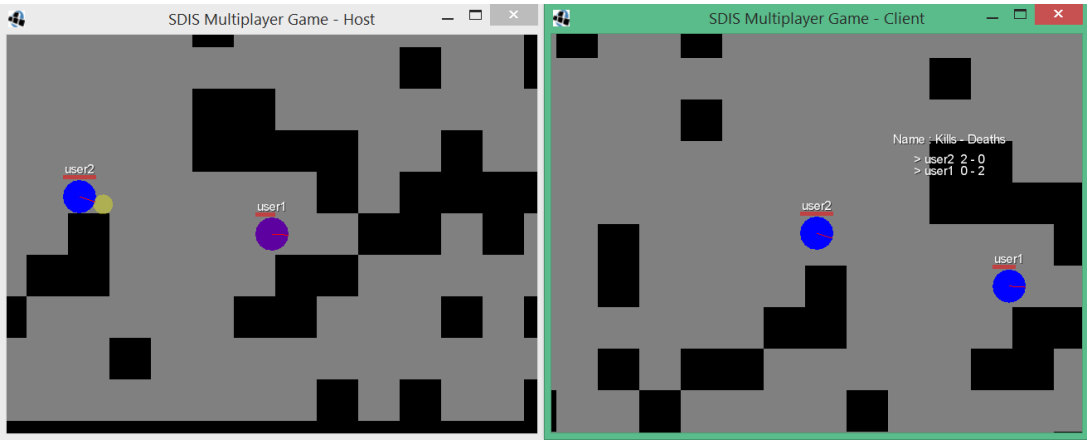

Figura 2: Interface da aplicação ao decorrer um jogo

Toda a estrutura e implementação da aplicação, bem como alguns pontos que consideramos relevantes, são explicados pormenorizadamente nos capítulos que se seguem. No capítulo relativo à arquitetura é descrita toda a estrutura de comunicação da aplicação distribuída. No capítulo que aborda a implementação são indicadas as bibliotecas utilizadas bem como algumas informações relativas aos módulos implementados. No capítulo seguinte são abordados alguns pontos relevantes relativos a questões de segurança, consistência e tolerância a falhas. Por último serão apresentadas as conclusões do grupo relativamente à realização do trabalho e também as referências consultadas.

2. Arquitetura

O sistema desenvolvido é constituido por 3 partes: Cliente, Servidor e Serviço Externo.
O módulo principal necessário ao funcionamento do jogo em si é constituido apenas pelo Cliente e pelo Servidor, podendo funcionar sem ligação ao Serviço de Autenticação se assim configurado (modo Offline). Este módulo funciona através de uma arquitetura Cliente-Servidor

típica simples, mas com a particularidade de todos os Servidores serem ao mesmo tempo Clientes.

Um Servidor está encarregue de receber amostras de input dos vários clientes, gerar estados de jogo iterativamente a um ritmo fixo e enviar regularmente amostras do estado de jogo a todos os clientes.
Um Cliente está encarregue de gerar e enviar amostras de input para um servidor ao mesmo ritmo que este gera estados, e de receber e mostrar ao utilizador os estados de jogo que recebe desse servidor.

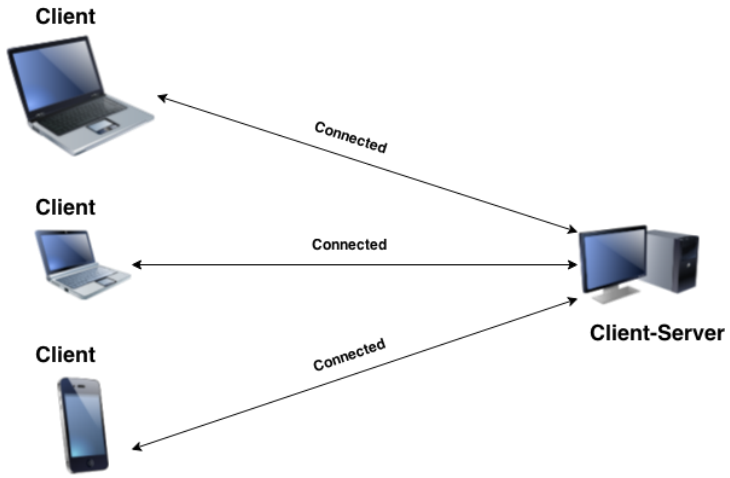

Figura 3: Arquitetura cliente-servidor da aplicação

A comunicação Cliente-Servidor é realizada através de um protocolo com as seguintes mensagens:

- **SnapshotMessage** - mensagem com um "*snapsho*t" do estado de jogo atual de um servidor, enviada regularmente por UDP a todos os clientes a um ritmo configurável. Inclui também o índice do último input recebido de cada jogador para permitir várias medidas de compensação de latência;
- **InputState** - mensagem com um "*snapshot*" do estado atual do input do jogador enviada regularmente por UDP ao servidor a um ritmo alto constante;
- **InputBundle** - mensagem semelhante à anterior mas que agrupa vários estados de forma a reduzir o número de pacotes enviados pelo cliente;
- **PlayerInfoBundle** - mensagem com as informações (nome, pontuação, etc.) sobre os jogadores conectados atualmente, enviada regularmente por UDP a todos os clientes a um ritmo baixo constante;
- **Level** - mensagem com o nível de jogo atual no servidor, enviada por UDP a pedido do cliente;
- **LoginMessage** - mensagem de *login* de um cliente num servidor, incluindo o nome que o cliente pretende usar e potenciais parâmetros de autenticação. Enviada por TCP pelo cliente ao servidor após a ligação inicial ter sido establecida;

- **LoginResponseMessage** - mensagem de resposta ao login de um cliente, enviada por TCP, incluindo informação sobre se o login foi bem sucedido, a razão de falha, e o id atribuido ao cliente;
- **LevelRequestMessage** - mensagem enviada por UDP, de pedido de envio de uma mensagem Level descrita anteriormente;
- **SnapshotRateMessage** - mensagem com o ritmo desejado de envio de mensagens *SnapshotMessage*, enviada por UDP pelo cliente;
- **PlayerConnectedMessage, PlayerDisconnectedMessage** e **PlayerKilledMessage** - mensagens enviadas por um servidor a todos os clientes por UDP para informar de eventos no jogo, respetivamente, quando um jogador entra no jogo, sai do jogo ou é destruido.

Quanto ao serviço externo implementado, este disponibiliza as funcionalidades de registo e autenticação de clientes, bem como a manutenção de uma lista de salas de jogo ativas. Para tal, tanto Clientes como Servidores comunicam com este serviço através de pedidos HTTP. Os pedidos disponíveis podem ser consultados na tabela que se segue.

Nome	Method	HTTPS required	Parametros	Descrição	Devolve
user/register	POST	Yes	username password	Regista um utilizador com o username e password dados	200 - Ok 400 - Pedido malformado 403 - Username já existe ou não está a ser usado HTTPS 500 - Erro a criar o utilizador
user/login	POST	Yes	username password	Efetua o login de um utilizador e inicia uma sessão	200 - Ok 400 - Pedido malformado 401 - Credenciais invalidas 403 - Username não existe ou não está a ser usado HTTPS
user/getToken	GET	Yes	-	Para a sessão atual, gera um novo token com um tempo de vida de 15 segundos e devolve-o.	200 - Ok + Token no corpo 401 - User não fez o login 403 - Não está a ser usado HTTPS
user/verifyToken	POST	Yes	username token	Verifica se, para o user especificado, o token dado é valido. Ser for, invalida o	200 - Ok 400 - Pedido malformado

				token para não poder ser reutilizado.	403 - Token invalido ou não está a ser usado HTTPS
server/getServerList	GET	No	-	Devolve uma lista de servidores ativos em formato JSON.	200 - Ok + Lista de Servidores em JSON
server/heartbeat	GET	No	name number_players max_players port	Guarda um registo de que existe um servidor ativo no endereço de onde originou o pedido e na porta especificada. Este registo tem um tempo de vida de 90 segundos.	200 - Ok 400 - Pedido malformado

Tabela 1: Lista de pedidos disponíveis no serviço externo

O modo como o serviço externo é utilizado pode ser observado na figura que se segue:

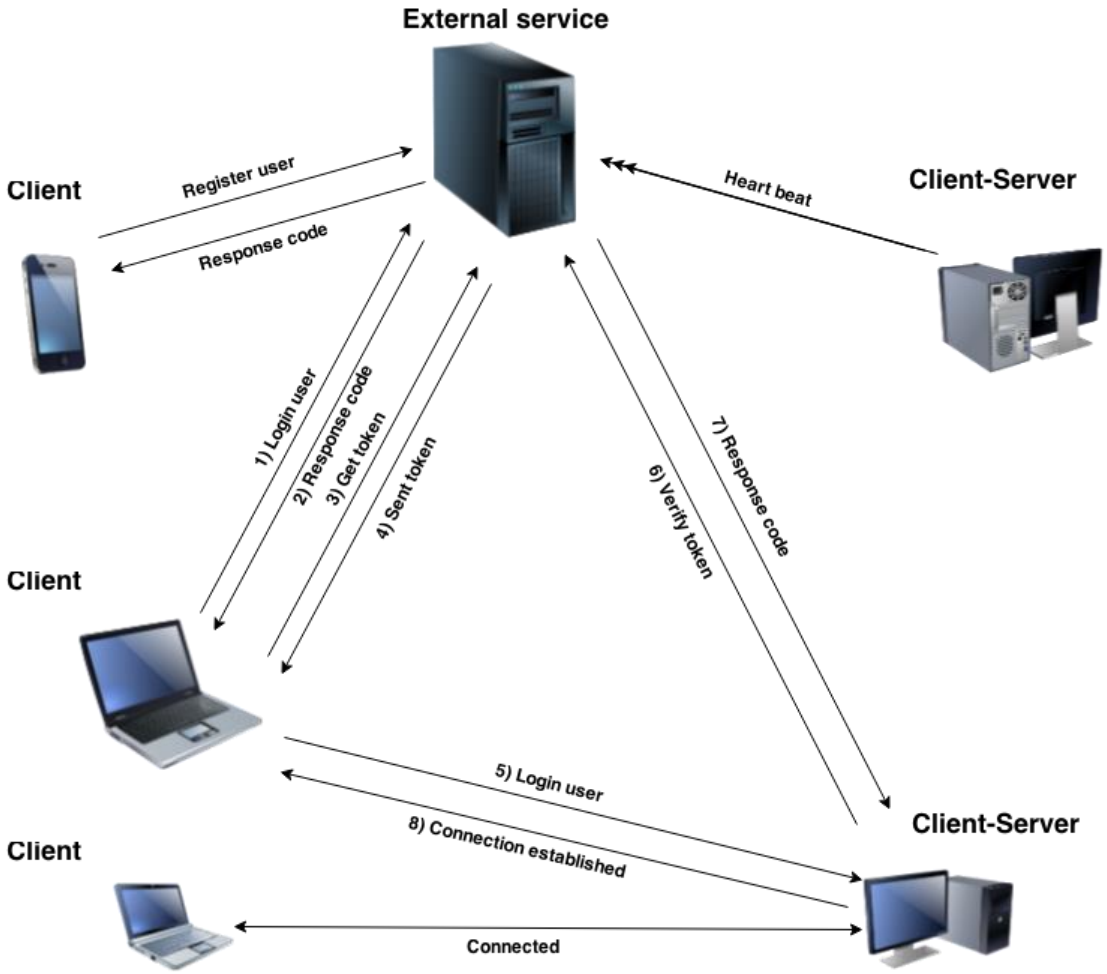

Figura 4: Arquitetura da aplicação com o serviço externo

O procedimento através do qual um cliente se autentica perante um servidor está contemplado na figura através dos passos 1 a 8:

1) O cliente envia um pedido para efetuar um login no serviço externo.
2) Se o pedido for bem sucedido, é criada uma sessão para o cliente no serviço externo e devolvido um codigo de sucesso.
3) Quando quiser ligar-se a um servidor, o cliente tem primeiro de enviar um pedido para obter um *token*.
4) Caso o cliente tenha efetuado o login corretamente é-lhe devolvido um *token* gerado aleatoriamente com uma validade muito curta de 15 segundos para proteção contra possível reutilização do mesmo.
5) O cliente envia uma mensagem de login com o seu username e o *token* que obteve.
6) Para confirmar se é um cliente autenticado, o servidor envia pedido ao serviço externo para verificar se o *token* é válido para o username do cliente que está a fazer login.
7) Em caso afirmativo, é enviada uma resposta condizente
8) É enviada uma mensagem de resposta ao login do cliente.

3. Implementação

O módulo principal Cliente-Servidor foi implementado em Java, com recurso a três bibliotecas: libGDX, KryoNet e org.json. Quanto ao serviço externo, este foi implementado em PHP e faz uso de uma pequena base de dados criada com SQLite.

3.1. Detalhes relevantes da implementação

Cliente:

De modo a ficar separado da *thread* principal da aplicação (criada pela libGdx), o cliente tem uma *thread* dedicada. A execução do cliente é *state-driven*, tendo este um estado atual que influência a visualização apresentada ao jogador e o tratamento de mensagens recebidas.

Para lidar com problemas de concorrência, para além das medidas que serão descritas no capítulo 5, foi necessário tornar os métodos de acesso à logica de jogo sincronizados. Foi também utilizada a classe java.util.concurrent.CountDownLatch para implementar uma espera com timeout por respostas do servidor.

As classes principais onde pode ser analisada a implementação do cliente são pt.feup.sdis.game.network.GameClient e pt.feup.sdis.game.logic.GameClientLogic.

Servidor:
Da mesma forma que um Cliente, um Servidor tem uma *thread* dedicada. A implementação do servidor é relativamente mais simples do que a do cliente, uma vez que, no geral, o servidor não tem tantas preocupações com compensação de latência como o cliente.
Para lidar com a concorrência, para além do que será descrito no capitulo 5, foram utilizados buffers para sincronizar eventos desencadeados por pedidos dos clientes com a lógica do servidor; e são criadas *threads* dedicadas para o tratamento de pedidos dos clientes cuja execução seja mais morosa como, por exemplo, o processo de login de um cliente.
As classes principais onde pode ser analisada a implementação do servidor são pt.feup.sdis.game.network.GameServer e pt.feup.sdis.game.logic.GameServerLogic.

3.2. Pedidos HTTP

O código relativo aos pedidos HTTP que a aplicação precisa de realizar estão agrupados na classe pt.feup.sdis.authserver.HttpRequests.

3.3. Bibliotecas

Foram utilizadas 3 bibliotecas para o desenvolvimento da aplicação:

libGDX:
É uma biblioteca para desenvolvimento de jogos cross-platform. A sua utilização facilitou o desenvolvimento das interfaces gráficas do jogo e permitiu fazer o *deployment* da aplicação para diversas plataformas de modo quase transparente.

KryoNet:
Esta biblioteca fornece uma API simples para comunicação Cliente/Servidor utilizando *Non-blocking I/O,* bem como uma ferramenta para serialização de mensagens. A biblioteca fornece duas classes Client e Server que mantêm uma *thread* que trata do envio e receção de mensagens e permitem a instalação de *listeners* para os eventos *connected*, *disconnected*, *received* e *idle*.
A classe Client mantem apenas uma ligação ao servidor, e tem os métodos *sendUDP* e *sendTCP* para enviar mensagens ao servidor.
A classe Server mantem um conjunto de ligações a cada cliente, e tem os mesmos métodos *sendUDP* e *sendTCP*, com a diferença de que estes recebem uma ligação específica para onde devem enviar a mensagem.
As mensagens enviadas e recebidas são objetos normais java e são serializados e deserializados automaticamente pela biblioteca. É necessário que os objetos a enviar sejam registados igualmente em clientes e servidores. Isto é feito implementação da classe pt.feup.sdis.game.network.messages.Register.

org.json:
Esta biblioteca permite o *parse* de objetos em formato JSON. Foi utilizada na comunicação com o serviço externo implementado, para fazer *parse* das respostas enviadas em JSON, mais especificamente, da lista de servidores disponiveis.

4. Questões relevantes

Segurança:
Uma das questões de segurança mais importantes no âmbito dos jogos multi-jogador modernos é a verificação da autenticidade dos clientes que se ligam aos servidores de jogo. Isto é desejavel por duas razões: para evitar que clientes que não tenham uma conta legítima, pela qual potêncialmente têm que pagar, utilizem os serviços disponibilizados; e para evitar que seja possível a um dado cliente fazer-se passar por outros.
Para solucionar este problema criou-se o serviço de autenticação externo.

Consistência:
Devido à latência que existe entre cada Cliente e Servidor, o Cliente vê um estado de jogo em que o Servidor estava no passado.
Se apenas fosse apresentado ao cliente os estados que chegam do servidor, este teria de esperar o tempo de *roundtrip* sempre que desse input até ver a sua personagem mexer!
Para resolver esse problema, o utilizador tem de prever a sua posição futura, em relação aos estados que recebe do servidor, para ver o estado da sua personagem em tempo real. Isto faz com que efetivamente todos os jogadores vejam um estado de jogo diferente: vêem-se a si mesmos no presente, e a todos os outros no passado.
Isto leva a dois problemas:

- O que acontece quando a posição em que o jogador prevê estar difere da posição onde o servidor diz que ele está (que chega como uma correção no passado)?
- Para um jogador acertar noutro precisa de apontar para onde o outro estava.

As soluções implementadas foram as seguintes:

- Ao nível do cliente, este mantém um buffer circular com o seu input, e marca cada input que envia ao servidor com um id. Quando o servidor envia ao jogador o estado de jogo, envia o id do ultimo input que recebeu. Para cada estado que recebe, o jogador aplica todo o input que tem guardado mas que ainda não foi recebido para chegar ao seu estado atual. Assim o jogador consegue ver o seu estado atual, mas continua a estar dependente do estado que o servidor envia.
- Ao nível do servidor, este mantém um buffer circular com os estados de jogo mais recentes. Quando tem de efetuar uma ação sensível a *timing*, como calcular colisões para

tiros do jogador, o servidor consegue reconstruir o estado de jogo que o cliente estava a ver no momento em que disparou, com base na latência observada.

Tolerância a falhas:
De forma a sincronizar o estado de jogo entre os vários clientes é necessária uma troca de mensagens a um ritmo elevado. Uma vez que é usado o protocolo UDP para o envio destas mensagens, a aplicação está sujeita às incertezas do serviço providênciado pelo protocolo. No caso da aplicação desenvolvida os mais relevantes são: possibilidade de perda de mensagens, possibilidade de receção de mensagens fora de ordem e variação no atraso de entrega das mensagens (*jitter*). Todos estes problemas poderão traduzir-se em erros na simulação ou visualização do jogo. Foram implementadas as seguintes soluções:

- Interpolação de estados de jogo: todos os estados de jogo que o cliente recebe do servidor são guardados num buffer circular. Em vez de mostrar simplesmente ao utilizador o estado mais recente, é mostrada uma interpolação entre dois estados guardados no buffer, com um atraso igual ao tempo necessário para que cheguem dois estados (2 * 1 / cadência de envio de estados). Estados que chegem fora de ordem são simplesmente descartados. Assim, há proteção contra a perda ocasional de um estado, incluindo casos em que chegam estados fora de ordem. Esta implementação permite ainda mascarar o facto de se receber estados a um ritmo menor ao que estes seriam gerados caso o jogo estivesse a correr localmente. (classe pt.feup.sdis.game.logic.GameClientLogic)
- Buffer de Input do jogador: do lado do servidor é necessário fornecer um "*stream*" constante de input ao motor de jogo. Porém, as mensagens de input chegam ao servidor com atrasos imprevisíveis, podendo ainda chegar "aos molhos" como foi referido no capítulo Arquitetura. Para resolver isto foi implementada uma solução normalmente usada em sistemas de *streaming* de média, como no Youtube ou em protocolos VoIP, conhecida por *Playout Delay Buffer*. (Ver classe pt.feup.sdis.game.network.InputBuffer)

5. Conclusão

O grupo implementou todos os pontos indicados na proposta do trabalho, incluindo o serviço externo de autenticação, que era dado como ponto "extra". Como melhorias a efetuar à aplicação, seria necessário em primeiro lugar aumentar o conteúdo do jogo (como criar a existência de inimigos, por exemplo), e a sua interface gráfica, de modo a torná-lo mais apelativo aos seus possíveis clientes. . No entanto, tendo em conta os objetivos do projeto, essa tarefa não foi de todo uma prioridade. Para além do serviço externo de autenticação criado, seria também interessante utilizar um serviço já existente como, por exemplo, o de uma rede social.

Distribuição do trabalho pelos membros do grupo:

- Luis Reis - 35%: Relatório, implementação da lógica de jogo, e arquitetura cliente-servidor.

- Luís Amaro - 18%: Relatório, testes à aplicação e serviço.
- Tiago Cavaleiro - 22%: Relatório, implementação do serviço externo e testes à aplicação.
- Susana Ventura - 25%: Relatório, implementação do serviço externo, interface gráfica e menus.

6. Referências

Esoteric Software. “kryonet - TCP/UDP client/server library for Java, based on Kryo”. Disponível em: https://github.com/EsotericSoftware/kryonet/.

Gabriel Gambetta. “Fast-Paced Multiplayer (Part IV): Headshot! (AKA Lag Compensation)”. Disponível em: http://www.gabrielgambetta.com/fpm4.html.

Gaffer on games. “What every programmer needs to know about game networking”. Disponível em: http://gafferongames.com/networking-for-game-programmers/what-every-programmer-needs-to-know-about-game-networking/.

JSON: JavaScript Object Notation. Disponível em: http://json.org/. [Acedido a 30 de Abril de 2015].

Mario Zechner. “libGDX - Desktop/Android/BlackBerry/iOS/HTML5 Java game development framework”. Disponível em: http://libgdx.badlogicgames.com/.

Tanenbaum, Andrew S, and Steen, Maarten Van. "Distributed systems: principles and paradigms", 2nd ed, Pearson/Prentice Hall ed., 2007.

Valve Developer Community. “Source Multiplayer Networking”. Disponível em: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking/.

Sean Middleditch. “How should multiplayer games handle authentication?”. Disponível em: http://gamedev.stackexchange.com/a/46595