UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Felipe Brum Mello

IMPLEMENTAÇÃO DE UM CLIENTE REMOTO PARA O SIMULADOR MASA SWORD

Santa Maria, RS
2016

Felipe Brum Mello

IMPLEMENTAÇÃO DE UM CLIENTE REMOTO PARA O SIMULADOR MASA SWORD

Trabalho apresentado ao Curso Ciência da Computação, Área de Sistemas Distribuídos, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Ciência da Computação.**

Orientador: Prof. Dr. Raul Ceretta Nunes

413
Santa Maria, RS
2016

Felipe Brum Mello

IMPLEMENTAÇÃO DE UM CLIENTE REMOTO PARA O SIMULADOR MASA SWORD

Trabalho apresentado ao Curso Ciência da Computação, Área de Sistemas Distribuídos, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Bacharel em Ciência da Computação.**

Aprovado em 14 de julho de 2016:

Raul Ceretta Nunes, Dr. (UFSM)
(Presidente/Orientador)

Luís Alvaro de Lima Silva, Dr. (UFSM)

Antonio Marcos de Oliveira Candia, Me. (UFSM)

Santa Maria, RS
2016

DEDICATÓRIA

À minha família, meus pais Roberto e Rosane, meu irmão Eduardo e minha namorada Carolina.

AGRADECIMENTOS

Este trabalho não teria sido possível se não fosse pela contribuição de diversas pessoas. Agradeço:

Ao meu orientador, pela oportunidade de realizar este trabalho e pelo seu suporte durante o desenvolvimento do mesmo, bem como por sua confiança e comprometimento.

Aos meus pais, por todo o apoio, amor e carinho que recebi durante minha vida. Tudo que tenho, é graças a eles.

À minha namorada, por todas as noites que passou comigo acordada até o amanhecer, me incentivando sempre a ser o melhor possível.

Aos meus colegas de graduação e amigos, por todas as memórias que compartilhamos, dentro e fora da universidade.

Por fim, aos meus colegas de trabalho, pois nesses últimos anos de graduação eles foram uma grande parte da minha vida e me ampararam em diversos momentos difíceis.

Ninguém pode ver nem compreender nos outros o que ele próprio não tiver vivido.

(Hermann Hesse)

RESUMO

IMPLEMENTAÇÃO DE UM CLIENTE REMOTO PARA O SIMULADOR MASA SWORD

AUTOR: Felipe Brum Mello
ORIENTADOR: Raul Ceretta Nunes

A área de modelagem e simulação assumiu um papel fundamental no treinamento das forças armadas de todo o mundo. Graças ao advento das simulações distribuídas, aliadas às simulações live – virtual – constructive(LVCs), os treinamentos envolvendo grandes contingentes deixam, cada vez mais, o mundo real e passam a ocorrer no mundo virtual. O simulador MASA SWORD oferece uma alternativa aos comandantes que buscam melhorar suas habilidades de tomada de decisão ao fornecer aos seus usuários o controle sobre unidades autônomas que podem ser agregadas a formações dos mais variados tipos, como pelotões, brigadas e divisões, facilitando a comunicação e organização dessas unidades dentro de um ambiente de simulação construtiva. Considerando a importância das simulações distribuídas e o alto grau de customização e integração oferecido pelo simulador construtivo SWORD, esse trabalho propõe o desenvolvimento de um cliente remoto que interaja com o simulador construtivo, através de troca de mensagens amparada na ferramenta protocol buffers, enviando e recebendo dados da simulação sobre suas formações, autômatos e unidades subordinadas, missões e informações gerais acerca da sessão.

Palavras-chave: Simulação construtiva. LVC. Simulação distribuída.

ABSTRACT

DEVELOPMENT OF A REMOTE CLIENT FOR THE MASA SWORD SIMULATOR

AUTHOR: Felipe Brum Mello
ADVISOR: Raul Ceretta Nunes

The modeling and simulation area has a fundamental role in military doctrines training. Due to the advent of distributed simulations and it's differents resolutions (live, virtual and constructive), the training of large numbers of personnel has increasingly moved from real environments to virtual ones. The MASA SWORD simulation system offers an alternative for training military strategies. Over it, commanders can practice their abilities of decision making. The simulator provides control over autonomous units that can be aggregated to formations of the most various types, such as platoons, brigades and divisions. These features facilitate the communication and arrangement of different units inside a simulation environment. Considering the importance of distributed simulations and the high degree of customization and integration offered by the SWORD constructive simulator, this work proposes the development of a SWORD remote client. This client interacts with the constructive simulator through message exchange supported by the google protocol buffers tool, sending and receiving data about the formations, automats, engaged units, missions and general information included in a simulation session.

Palavras-chave: Constructive simulation. LVC. Distributed simulation.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

DoD Department of Defense
DMSO Defense Modeling and Simulation Office
M&S CO Modeling and Simulation Coordination Office

SUMÁRIO

1 INTRODUÇÃO

O termo simulação, em seu significado mais primitivo, remete a fingimento ou imitação, carregando quase sempre uma conotação negativa. No direito civil, é considerado um ato de má-fé, onde se realiza uma declaração falsa com o intuito de aparentar um negócio diferente do desejado. Transportando esse termo para a área de tecnologia de informação, a simulação computacional é definida por Fujimoto (2000, p. 4, nossa tradução) como “uma computação que modela o comportamento de um sistema real ou imaginário por um período de tempo”. Simular, entrando no contexto tecnológico atual e abandonando seus sentidos negativos, pode ser descrito portanto como o ato de imitar o funcionamento de sistemas e operações reais através de modelos que traduzam suas características e comportamentos chave, uma descrição similar ao conceito de simulação computacional.

Seu uso pode se dar em diversos contextos, tendo como finalidade, entre outras, validação de modelos, otimização de desempenho, análise de resultados, lazer, ensino e treinamento. Um touro mecânico imita o comportamento de um touro com o intuito de proporcionar uma simulação da experiência real aos indivíduos que pagarem para montá-lo. Nos bancos, são executadas simulações para definir quais serão os rendimentos obtidos em uma aplicação financeira qualquer, dado um período de tempo e um montante inicial. Esses exemplos trazem algumas das diversas formas em que as simulações se encontram inseridas na vida contemporânea. Dentre seus inúmeros campos de atuação, se destacam principalmente a área bélica, a medicina, a engenharia, a tecnologia de informação, a educação, economia e finanças. Considerando essa pluralidade de aplicações, observa-se que as simulações são componentes essenciais quando busca-se entender a velocidade exponencial dos avanços tecnológicos no final do século XX e início do século XXI. As simulações computacionais são responsáveis por diminuir consideravelmente o custo – monetário e temporal – de desenvolvimento de sistemas. Graças a elas evita-se construir protótipos reais, cuja produção pode vir a ser muito cara ou gastar considerável tempo e recursos, e constrói-se um modelo computacional, cuja execução determinará sua eficácia e validade.

Surge então a Modelagem e Simulação (M&S), uma disciplina emergente –

pois seus méritos como disciplina por si só ainda estão sendo discutidos (Padilla et al, 2011) - cujas principais contribuições, em termos de suporte e de financiamento, provém da área bélica, mais precisamente, do Departamento de Defesa (DoD) dos Estados Unidos da América. A maior vantagem vista pelos militares foi a possibilidade de substituir exercícios de campo por uma alternativa mais segura e menos custosa: ambientes virtuais. Além disso, surgiu também a possibilidade de integrar esses ambientes virtuais a outro campo da computação, as simulações distribuídas. Ao aliar essas duas áreas podem ser realizados treinamentos e exercícios militares envolvendo de dezenas a centenas de jogadores, necessitando apenas que se tenha computadores suficientes para todos. As simulações distribuídas, como vieram a ser chamadas, também tem como vantagem a distribuição geográfica, o que permite que esses computadores (ou terminais) se encontrem distribuídos em um prédio, campus ou até mesmo cidade. Elas também permitem a integração entre diferentes simuladores, aumentando ainda mais o grau de imersão dos exercícios. Essa integração entre simulações – tanto iguais quanto distintas – impactou não só o setor bélico, mas também os jogos online, a computação de alta performance, a educação e as redes de telecomunicações.

1.1 OBJETIVO DO TRABALHO

O sistema de simulação construtiva MASA Sword é um simulador de uso militar que fornece aos seus usuários o controle sobre um campo de batalha. Seu objetivo é treinar as habilidades de tomada de decisão de comandantes militares, provendo os usuários com dezenas de unidades autônomas que operam a partir de comportamentos de inteligência artificial. Este simulador oferece a possibilidade de integração das simulações a aplicações externas, através de sua API de Rede. Por ser um simulador comercial e de uso militar, ele possui um certo grau de confidencialidade, o que restringe trabalhos e artigos que concentrem-se nessas aplicações. Como não se tem informação sobre outras aplicações remotas de uso acadêmico além dos materiais fornecidos pela própria MASA, a proposta deste trabalho é explorar o uso da API de Rede deste simulador de forma a servir de base para trabalhos futuros.

1.2 METODOLOGIA DE DESENVOLVIMENTO

A exploração do simulador se deu através da implementação de um cliente que se comunica com o sistema de simulação construtiva MASA SWORD. Após estabelecida a comunicação, ele busca e acompanha suas unidades subordinadas dentro do cenário. O cliente, portanto, assume o papel de um usuário, de forma remota, que visualiza suas unidades e recebe do simulador os efeitos que ordens e missões quaisquer terão sobre a execução de seu cenário.

Com o resultado – um cliente remoto que interage com o simulador construtivo MASA SWORD, com uma descrição detalhada de seu projeto e e de sua implementação – abre-se uma porta para futuros trabalhos que busquem utilizar a API do simulador MASA SWORD para desenvolvimento de sistemas e simulações distribuídas.

1.3 ORGANIZAÇÃO DO TEXTO

O trabalho está organizado da seguinte forma. O capítulo 2 traz a fundamentação teórica necessária para o entendimento do contexto em que o trabalho se insere, bem como realiza uma apresentação das ferramentas empregadas. O capítulo 3 descreve os diversos métodos empregados para realizar o projeto. O capítulo 4 faz uma descrição do cliente proposto e descreve o funcionamento de seus componentes. O capítulo 5 expõe a execução do cliente em conjunto a um exercício do simulador e as dificuldades encontradas em sua implementação. Por fim, o capítulo 6 conclui este trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta uma síntese teórica dos principais conceitos relacionados ao trabalho. A subseção 2.1 apresenta um histórico do uso de treinamento baseado em simulação e indica como esta técnica de treinamento ingressou no âmbito militar. Nas subseções 2.2 e 2.3, abordam-se, respectivamente, as simulações distribuídas e o conceito de simulações vivas, virtuais e construtivas. Na subseção 2.4 é exposto o simulador construtivo MASA SWORD e são demonstradas algumas de suas funcionalidades e características. Na subsseção 2.5, é apresentado o Google protocol buffers, que será utilizado para realizar a comunicação entre o cliente proposto e o simulador.

2.1 TREINAMENTO BASEADO EM SIMULAÇÃO

Para compreender a importância da simulação como mecanismo para ensino e treinamento, é preciso entender a principal diferença entre os dois conceitos. Enquanto o primeiro leva o indivíduo a adquirir competências, o segundo tem o intuito de capacitar indivíduos para executarem atividades práticas. Ou seja, o ensino tem um enfoque maior no âmbito teórico e na conversão da informação em conhecimento. O treinamento, por outro lado, se aplica à prática, focando na experiência própria como forma de aprendizado.

Simulações são muito utilizadas como ferramentas de treinamento, como em simuladores de voo (HOLMAN, 1979), exercícios de treinamento militares (SMITH, 2010), educação médica baseada em simulação (LATEEF, 2010) (FLATO; GUIMARÃES, 2011), treinamento para prevenção de catástrofes (ŠIMIĆ, 2012) entre diversas outras. Lateef (2010, p. 348, tradução nossa) descreve a simulação como “uma técnica que substitui e amplifica experiências reais por guiadas, geralmente imersivas, que evocam ou replicam aspectos substanciais do mundo real de uma maneira totalmente interativa”. É interessante destacar que, mesmo que o aprendizado baseado em simulação possa ser implementado de forma não-guiada, estudos como o de Issenberg et al (2005, p. 26) mostram que oferecer ajuda de forma instrutiva, através de ferramentas como feedback, controle do ambiente de

simulação e estipulação de objetivos bem definidos, entre outras, torna o aprendizado muito mais eficiente.

No âmbito militar, simulações são usadas para treinamento e instrução desde muito antes da invenção do computador. O exemplo mais antigo de um jogo de estratégia, como descrito por Smith (2010, p. 2), foi encontrado no Japão e é datado dos anos 3000 a.C., na forma de um jogo de tabuleiro onde os jogadores disputavam territórios. Mesmo assim, foi somente após o século XVII que estes “simuladores” se voltaram ao treinamento militar. No decorrer dos séculos seguintes, eles foram se tornando cada vez mais complexos.

Com o advento da computação, as dificuldades e complexidade de gerenciamento dos jogos foram deixadas a cargo dos computadores, permitindo que os usuários se focassem apenas no exercício. “Também se tornou prático distribuir a simulação entre diversas salas e apresentar uma visão personalizada da batalha para cada jogador”, de acordo com Allen (1987 apud SMITH, 2010, p. 5). Dessa forma, simulações distribuídas se tornaram indispensáveis ao treinamento militar, oferecendo uma alternativa menos custosa de treinamento e permitindo a sincronicidade entre diversos jogadores.

2.2 SIMULAÇÃO DISTRIBUÍDA

No decorrer dos anos 70, simulações computacionais voltadas para treinamento já vinham sendo empregadas no ambiente militar, como traz o artigo de Holman (1979), sobre a eficiência de treinamentos utilizando simuladores de voo. Na década de 80 ocorreu o primeiro esforço, por parte dos militares, no campo da simulação distribuída através do projeto SIMNET (SIMulation NETwork), que conectava diversos simuladores autônomos e distribuídos a determinado exercício de treinamento. As motivações militares são descritas no trecho a seguir:

> Um fator-chave conduzindo o desenvolvimento e adoção de simulações distribuídas para ambientes sintéticos têm sido a necessidade dos militares de desenvolver meios de treinamento de pessoal mais efetivos e econômicos antes de entrarem em combate (FUJIMOTO, 2000, p. 9, nossa tradução).

Na década de 90, o Departamento de Defesa dos Estados Unidos da América

(USA) foi encarregado, pelo congresso estado-unidense, de coordenar uma política de simulação, promover o uso de simulação dentro de instalações militares e estabelecer padrões e protocolos de interoperabilidade, entre outras atribuições.[1] Logo, foi criado o Gabinete de Defesa de Modelagem e Simulação (DMSO), que foi renomeado em 2007 para Gabinete de Coordenação de Modelagem e Simulação (M&S CO). O DoD criou, durante o período em que se tornou autoridade na área, a High Level Architecture (HLA), um padrão de arquitetura para sistemas de simulação distribuída que permite a interoperabilidade entre sistemas de simulação distintos. Segundo Dahmann et al (1997, p. 142, tradução nossa) “a intenção da HLA é fornecer uma estrutura que suporte o reuso das potencialidades disponíveis em diferentes simulações”. Todas as simulações do DoD devem seguir essa arquitetura.

2.3 SIMULAÇÃO LVC

De acordo com o Glossário de Modelagem e Simulação, publicado pelo DoD, as definições de LVC (Live – Virtual – Constructive) são:

> **Simulação Viva**: Simulação viva envolve pessoas reais operando sistemas reais. Exercícios de treinamento militar usando equipamentos reais são simulações vivas. Esses exercícios são considerados simulações por não serem conduzidos contra inimigos reais.
> **Simulação Virtual:** Simulação envolvendo pessoas reais operando sistemas simulados. Simulações virtuais injetam human-in-the-loop[2] em um papel central ao exercitar habilidades de controle motor (e.g. pilotar um avião), habilidade de tomada de decisão (e.g. aplicando recursos de controle de incêndios) ou habilidades de comunicação (membros de um esquadrão).
> **Simulação Construtiva:** Uma simulação construtiva inclui pessoas simuladas operando sistemas simulados. Pessoas reais estimulam (produzindo entradas) essas simulações, mas não estão envolvidas na determinação dos resultados. Uma simulação construtiva é um programa de computador. Por exemplo, um usuário militar pode inserir dados instruindo uma unidade a se mover e atacar com um alvo inimigo. A simulação construtiva determina a velocidade do movimento, o efeito do ataque ao inimigo e qualquer dano que a batalha possa causar (DOD, 2011, p. 85, 119, 159, tradução nossa).

Simulações podem ser classificadas dentro dessas três categorias – podendo

1 Trecho retirado da página https://en.wikipedia.org/wiki/Live,_virtual,_and_constructive#History (DoD Appropriations Bill, p. 154–155, 1990)

2 1. Modelo que requer interação humana em tempo de execução. 2. Simulação ou simulador que emprega um ou mais operadores humanos em controle direto sobre a simulação/simulador ou em alguma função-chave de apoio. (DoD, 2011, p. 109, tradução nossa)

pertencer a apenas uma classificação ou a combinações entre duas ou mais categorias – de acordo com o nível de interação humana presente. Pesquisas na área de simulações LVC focam na busca os possíveis benefícios na integração entre simulações vivas, virtuais e construtivas e em formas de implementá-la.

Buscando uma forma de integrar as diferentes arquiteturas desenvolvidas para simulações distribuídas ao longo dos anos, o DoD publicou (HENNINGER, 2008) o LVCAR (Live Virtual Constructive Architecture Roadmap), um roteiro determinando os desafios apresentados pela interoperabilidade entre diferentes arquiteturas e possíveis soluções. Nele, é descrita uma estratégia com o intuito de guiar o desenvolvimento de arquiteturas integradas para sistemas LVC pelos próximos dez anos.

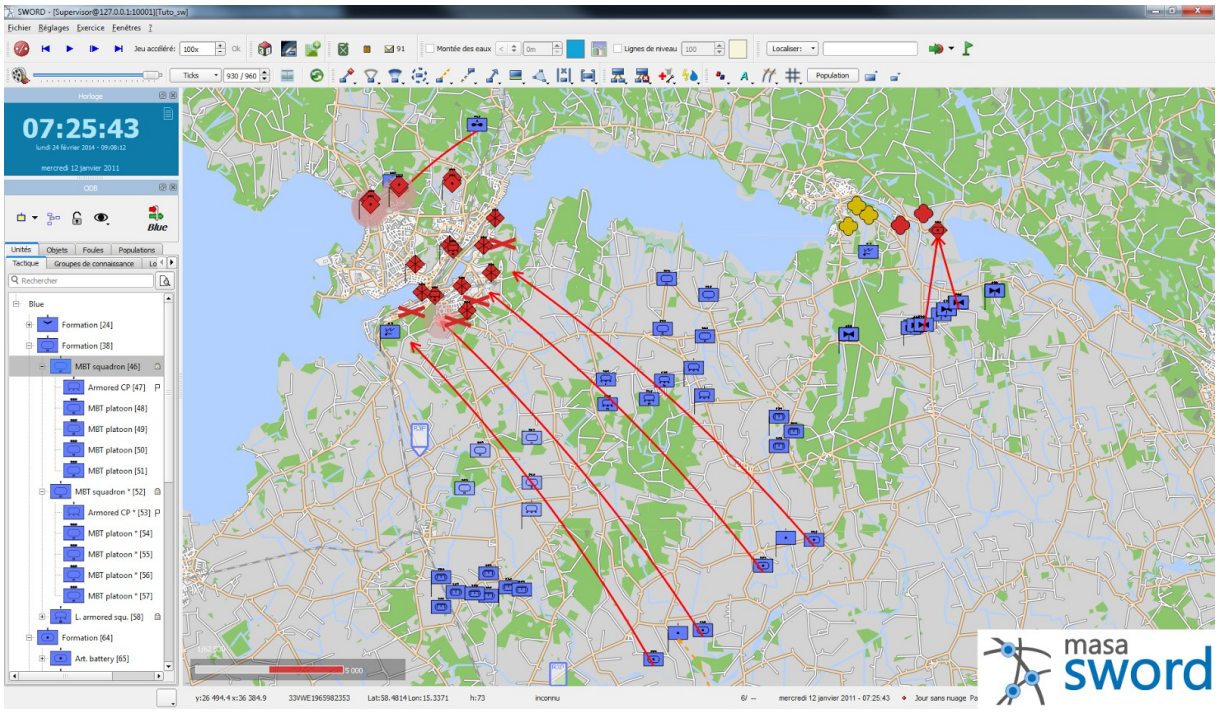
2.4 SIMULADOR CONSTRUTIVO MASA SWORD

O simulador MASA SWORD é um simulador construtivo, agregado e automatizado, desenvolvido para treinar as habilidades de tomada de decisão de equipes de comando de brigadas e divisões, permitindo a imersão em simulações de conflitos, operações militares e controle e prevenção de desastres. Ele é utilizado pelo exército brasileiro desde 2013, com o intuito de melhorar as habilidades de tomada de decisão de comandantes militares e da defesa civil. No total, ele se encontra presente em 15 países, e seu crescente mercado se dá, entre outros motivos, por permitir e facilitar a integração entre a simulação e os sistemas já utilizados pelas respectivas forças, se adaptando inclusive as doutrinas militares de seus clientes.

Simulações construtivas, como visto anteriormente, envolvem pessoas reais gerenciando tanto pessoas simuladas quanto sistemas simulados. No caso do SWORD, por lidar com formações militares que podem envolver de dezenas (pelotões) à milhares de tropas (brigadas e divisões), é adotada uma abordagem agregada, onde estas unidades individuais são agregadas a unidades maiores, como visto na Figura 1. Caldwell, et al (2000, p. 2) nota que essa combinação entre as unidades individuais é simples de ser implementada, pois a hierarquia militar já define os agrupamentos apropriados. Por outro lado, as unidades individuais e suas

respectivas características se perdem quando são agrupadas (p. 3).

Figura 1 – Simulação de conflito no simulador MASA SWORD



Fonte: http://www.miltechmag.com/2015/04/itec-2015-masa-group-enhances-core.html
Pelotões são as unidades básicas da hierarquia (lado esquerdo da Figura, não legível).

O comportamento dessas unidades é automatizado, isto é, elas recebem ordens de alto nível dos usuários e as interpretam, repassam (caso possuam unidades subordinadas) e executam, através de inteligência artificial (autômatos). Unidades autônomas que simulam diferentes comportamentos e possuem capacidades de comandar são extensivamente usadas dentro das simulações do SWORD, com o intuito de aumentar a eficiência e o realismo do treinamento seja através de forças inimigas, neutras ou entidades civis. Os comportamentos de inteligência artificial mencionados surgem dos modelos de doutrina que cada unidade implementa, determinando seu papel na simulação. Esses modelos permitem customização por parte do usuário para seguir as doutrinas e comportamentos mais adequados para o exercício em questão.

Os cenários oferecem dois tipos de usuários, que possuem permissões e papéis diferentes dentro de cada simulação. O primeiro tipo é o treinado, que

exercita suas habilidades de tomar decisões ao definir as ordens que são dadas às unidades e analisar as situações em que é colocado. Em diversas ocasiões, o treinado pode ter visão limitada do campo de batalha (enxergando apenas o que suas unidades veem, respeitando o prazo de validade estipulado da informação) para que a simulação seja mais real. O segundo tipo é o instrutor, responsável por preparar e comandar a simulação. É sua atribuição definir os jogadores que entrarão no exercício, parar e recomeçar o exercício a qualquer momento, alterar as propriedades de unidades e grupos de conhecimento, realizar alterações táticas no posicionamento das unidades durante o exercício e retransmitir o exercício, após o fim, para explicar em detalhes os acontecimentos e suas causas. Os critérios de performance e as permissões de cada usuário (quem pode controlar, quem pode ver, se são supervisores, se tem controle temporal, etc) são definidas durante a preparação do exercício.

Para que um usuário execute uma simulação, a aplicação denominada *Gaming* – uma espécie de cliente remoto nativo ao simulador – deve ser acessada a partir do menu principal do SWORD. A aplicação *Gaming* permite que o usuário comece um novo exercício ou conecte-se a um exercício que já se encontre em execução e essa conexão pode ocorrer de forma local ou remota.

O simulador MASA SWORD disponibiliza duas formas de comunicação para implementação de sistemas distribuídos. A primeira consiste em um módulo HLA que permite que o SWORD se conecte com outros simuladores e realize exercícios distribuídos. A segunda é a API de rede do SWORD, que implementa a comunicação entre cliente e servidor utilizando o protocolo TCP/IP e a tecnologia Protobuf para realizar codificação das mensagens. Essa forma é a utilizada para realizar a comunicação entre a aplicação *Gaming*, que é oferecida pelo próprio simulador, e a simulação.

2.5 GOOGLE PROTOCOL BUFFERS

A ferramenta protocol buffers (GOOGLE, 2016), também chamada de Protobuf, é um mecanismo de serialização de dados estruturados desenvolvido pela empresa Google, muito utilizado para armazenamento de dados e troca de

mensagens. É também extensível e de linguagem e plataforma neutras, ou seja, a serialização e desserialização dos dados podem ocorrer em plataformas e linguagens diferentes. O Protobuf é uma alternativa mais simples, menor e mais fácil de manipular dados quando comparado a outra ferramenta muito utilizada para o mesmo propósito, o XML (Linguagem de Marcação Extensível). Em alguns casos, porém, o XML se sai melhor, como quando ocorre o uso de linguagens de marcação, como o HTML.

A estrutura dos dados é especificada, em um primeiro momento, em arquivos *proto* (esquemas). Esses arquivos especificam diversos tipos de mensagem e seus respectivos campos. Cada mensagem é representada por uma série de pares nome-valor, como pode ser notado na Figura 2.

Estruturados os dados, basta utilizar o compilador de protocol buffers para compilar o arquivo proto para a linguagem desejada – disponível nas linguagens C++, Java e Python – para que seja gerada uma classe que permita acesso aos valores dos campos (leitura e escrita), além dos métodos de serialização e desserialização da estrutura.

Figura 2 – Estrutura de um arquivo `pessoa.proto`

```
message Pessoa {
        required string nome = 1;
        required int32 idade = 2;
        optional string email = 3;
        enum EstadoTipo {
                CASADO = 0;
                SOLTEIRO = 1;
                DIVORCIADO = 2;
                VIUVO = 3;
         }
        message EstadoCivil {
                required EstadoTipo tipo = 1;
        }
        required EstadoCivil estado_civil = 4;
}
```

Fonte: (GOOGLE, 2016 nossa tradução).
Baseado no exemplo de arquivo *.proto* apresentado na fonte.

3 METODOLOGIA

Foram realizados, em um primeiro momento, estudos sobre a ferramenta protocol buffers e o funcionamento do simulador MASA SWORD. Os resultados desses estudos estão descritos nos capítulos 2 e 3. Os estudos sobre protocol buffers focaram em sua documentação, que descreve todo o processo de estruturação das mensagens, bem como suas particularidades para cada linguagem e a geração automática de métodos que acessam seus atributos, com tutoriais disponibilizados pelos desenvolvedores nas linguagens C++, Java e Python. Essa pesquisa foi feita principalmente para que, posteriormente, pudesse ser identificado o funcionamento da tecnologia protobuf dentro do contexto da API de Rede do SWORD.

O estudo sobre o SWORD, por outro lado, começou com a análise e leitura de diversos documentos, tutoriais e manuais disponibilizados pela empresa MASA, buscando compreender como executar e preparar exercícios dentro do simulador. Após isso, foram realizados diversos experimentos utilizando cenários novos e preexistentes com o intuito de compreender o comportamento das diferentes unidades autômatas presentes no simulador perante certas ordens e situações de combate. Por fim, chegou-se a um cenário satisfatório que visa descrever uma missão de artilharia (vide capítulo 4). Também se deu o estudo dos logs gravados pelos cenários, mais precisamente os `Dispatcher.log` e `Protobuf.log`, que são gerados sempre que uma nova execução de exercício é iniciada. Através destes logs, foi possível identificar a troca de mensagens entre os componentes *SWORD Gaming* e *Dispatcher* – ou seja – entre o cliente e o servidor, traçando um paralelo com a comunicação que ocorre entre o cliente remoto proposto e o servidor.

A etapa de implementação do cliente remoto se baseou unicamente na documentação da API que foi fornecida pela MASA. Essa documentação é a única fonte de informação sobre a formatação das mensagens empregadas e características da conexão entre a aplicação remota e o servidor. Como essa API se ampara em arquivos *.proto*, as linguagens de programação são limitadas à Java, C++ e Python, as três linguagens para as quais o compilador do Google *protocol buffers* suporta a geração de classes. Dessa forma, a escolha pela linguagem Java

partiu dos conhecimentos prévios adquiridos durante este curso de graduação, pois foi a única linguagem orientada a objetos abordada a fundo nas disciplinas relacionadas a programação.

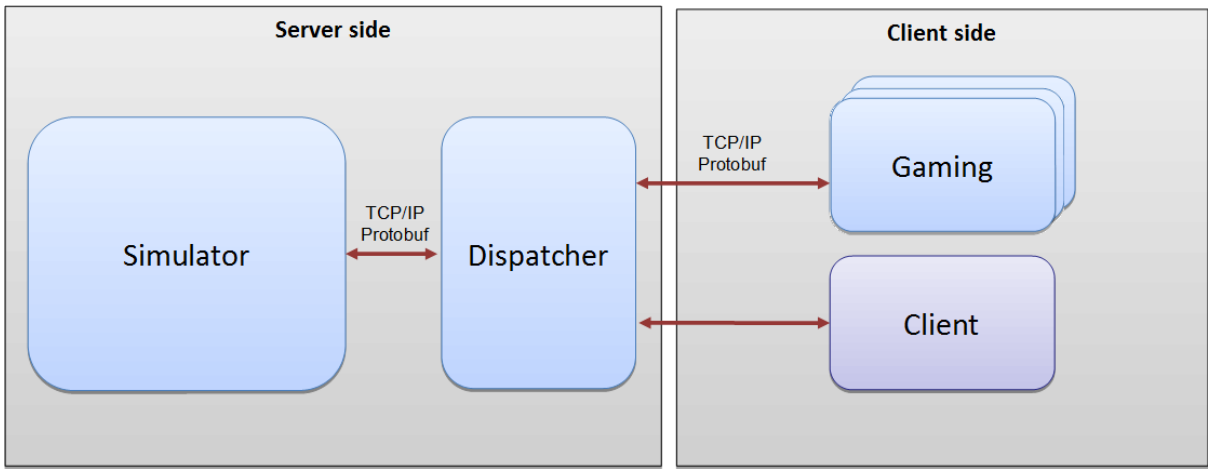
4 CLIENTE MASA SWORD

Este capítulo descreve o cliente MASA SWORD desenvolvido. A seção 4.1 descreve a arquitetura da solução, sendo primeiramente apresentado um exercício criado no simulador (seção 4.1.1), para servir de estudo de caso, e, posteriormente, o funcionamento de sua API de rede (seção 4.1.2). A seção 4.2 relata a usabilidade dos protocol buffers no cliente remoto, e a seção 4.3 explica a implementação do cliente e seu funcionamento.

4.1 ARQUITETURA DA SOLUÇÃO

O simulador construtivo MASA SWORD segue uma arquitetura cliente/servidor, como mostra a Figura 3. Entre a aplicação denominada *Gaming* (acessada pelos usuários quando estão em uma sessão) e o Simulador, existe um Dispatcher que controla a comunicação através do protocolo TCP/IP e da ferramenta protocol buffers. O cliente remoto proposto, denominado Client, também se encontra na Figura 3. O funcionamento do simulador é apresentado na seção 4.1.1 e a forma de comunicação é apresentada na seção 4.1.2.

Figura 3 – Arquitetura do Simulador MASA SWORD



Fonte: Sword API Training.

O servidor é composto pelo Simulador e pelo componente *Dispatcher*, que

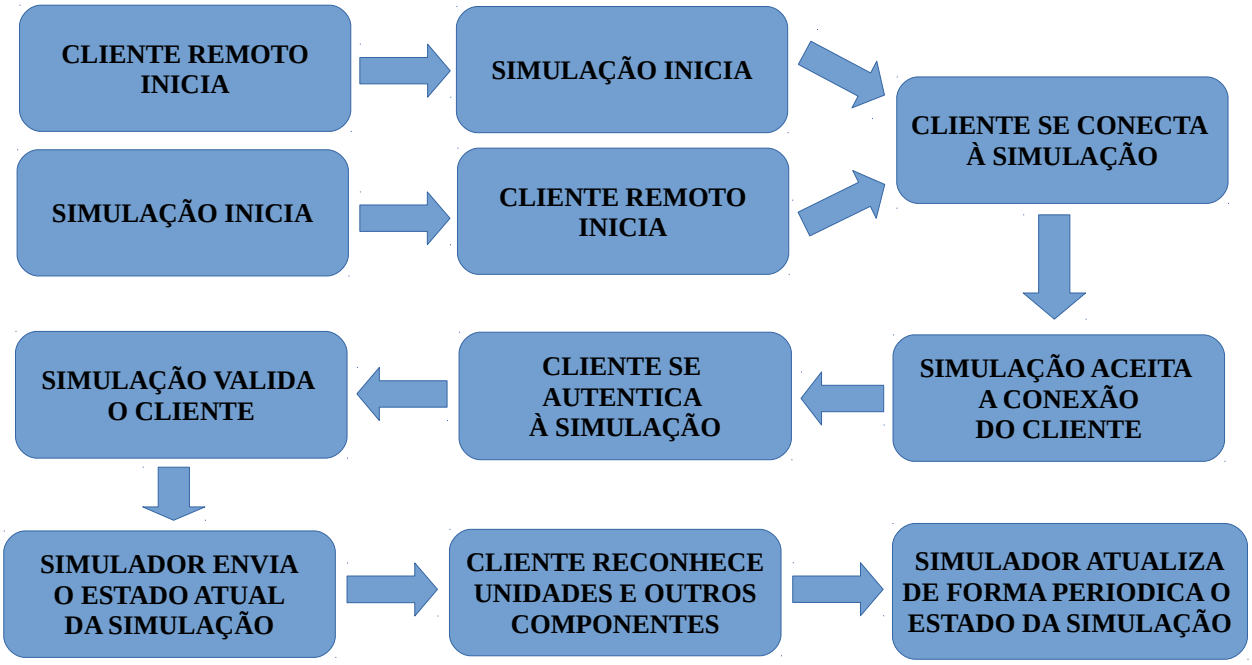
realiza o intermédio entre o simulador e as mensagens que chegam do lado do cliente. Essas mensagens são definidas através da ferramenta protocol buffers.

O lado do cliente é composto pela aplicação remota proposta e pelo componente *Gaming*. Como mencionado anteriormente, esse componente é a aplicação utilizada pelos usuários do simulador para executar os exercícios. Diversos usuários – remotos ou locais – podem se conectar ao mesmo exercício, ao mesmo tempo, utilizando o *SWORD Gaming*. Cada sessão define um limite máximo de usuários conectados, sendo esse limite passível de alteração. Portanto, pode ser dito que o componente *Gaming* também faz o papel de um cliente remoto dentro da arquitetura do simulador. Por esse motivo, o cliente remoto proposto tem como foco o monitoramento da simulação, incluindo também as interações que o usuário conectado à aplicação *Gaming* realizar com a simulação.

Para que o cliente proposto possa se conectar a uma simulação, é preciso antes que um usuário inicie o exercício através da aplicação *Gaming*. Isso ocorre porque a inicialização de um cenário de simulação só ocorre quando, na aplicação Gaming, ocorre a primeira conexão (*Start*).

A Figura 4 apresenta o fluxo dos passos executados pelo cliente remoto. Primeiro, o cliente e a simulação precisam iniciar. Depois, o cliente precisa se conectar e se autenticar à simulação. Assim que o cliente é validado (reconhecido como um perfil válido) pela simulação, ele tem o estado atual da simulação inicializado em seu ambiente próprio. O cliente então é mantido atualizado de possíveis mudanças, como outros clientes dando ordens e o *tick* atual da simulação, entre diversas outras.

Figura 4 – Diagrama de blocos descrevendo o funcionamento da solução.



Fonte: Próprio autor.

4.1.1 Exercício do estudo de caso

Como caso de uso, foi preparado um cenário que implementa uma missão de artilharia no simulador MASA SWORD. Este cenário visa demonstrar a realização de todos os passos necessários para permitir que uma unidade de artilharia abra fogo. São eles:

a) determinar a zona de reunião;

b) reconhecer a área de posição;

c) construir as posições de tiro;

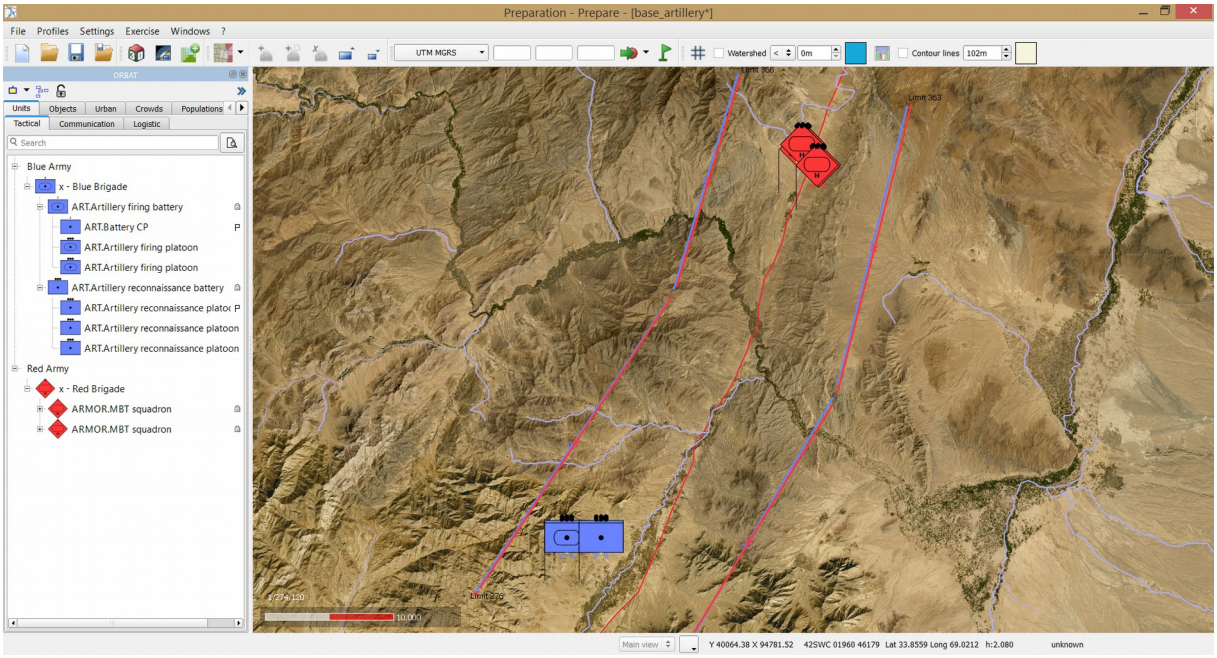
d) ocupar as posições de tiro e atirar.

Durante a execução do exercício, esses passos serão demonstrados a partir das ordens disponibilizadas pelo simulador.

O cenário, denominado “base_artillery”, possui dois lados – os exércitos azul e vermelho – que estão definidos como inimigos. Cada exército possui uma formação brigada, sendo a do exército azul composta por dois autômatos do tipo bateria de artilharia, um de reconhecimento e um de disparo, e a do exército

vermelho composta por dois autômatos do tipo esquadrão, conforme a Figura 5. Ambos os lados possuem linhas paralelas aos seus possíveis trajetos, denominadas limites. Eles são parâmetros quase sempre necessários no momento de enviar ordens às unidades.

Figura 5 – Captura de tela do exercício base_artillery no simulador MASA SWORD



Fonte: Simulador MASA SWORD – Preparation Module.

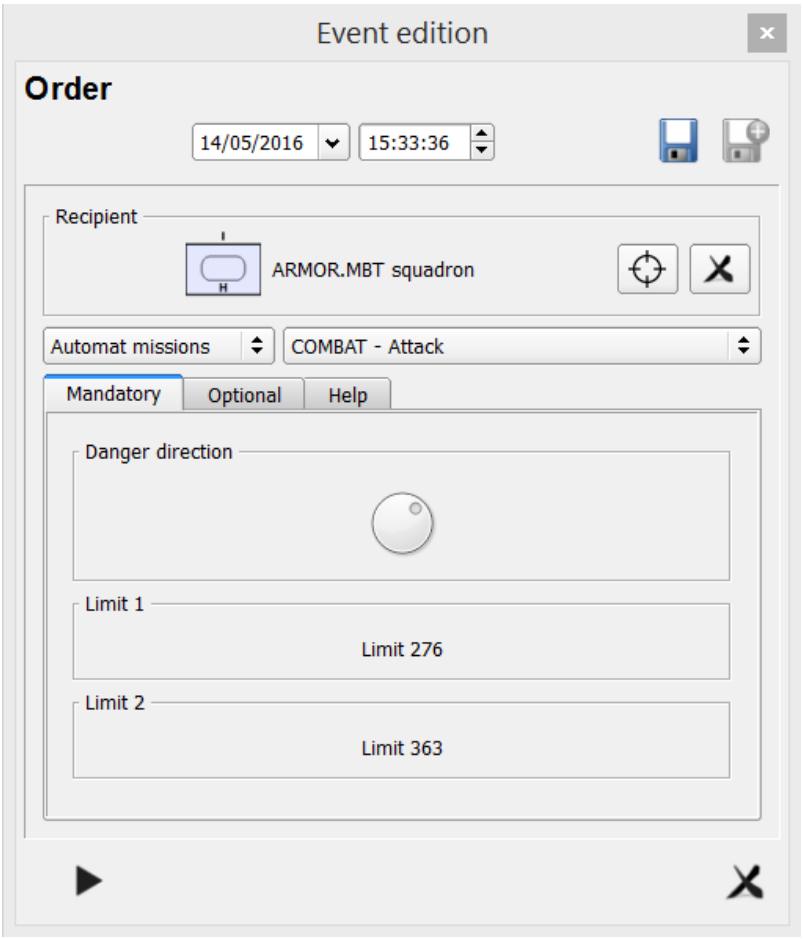
Para iniciar o exercício, é necessário primeiro entrar na aplicação *SWORD Gaming* conectar-se a um perfil de usuário, que é criado durante a preparação do cenário, consistindo em login e senha. Neste exercício, o perfil de usuário denominado Supervisor possui permissões de comando sobre ambos os exércitos, facilitando a gerência do combate. Além disso, ele também possui permissões de supervisor, o que lhe permite criar novos perfis e executar ordens mágicas, como teleporte e criação de unidades, ambos durante o exercício. Outros dois usuários, denominados Blue User e Red User, possuem permissão apenas para visualizar e comandar seus respectivos exércitos. Os três usuários também possuem permissão para controle temporal (podem parar ou acelerar a simulação).

Depois de conectado, o usuário tem a possibilidade de enviar ordens às suas

unidades subordinadas. Para executar a missão de artilharia, o perfil escolhido deverá ter permissões de comando sobre o exército azul. Além disso pode-se simplificar o exemplo utilizando um perfil que tenha – ao menos – permissões de visibilidade sobre o exército vermelho. Portanto, o perfil utilizado será o *Supervisor*.

Para abrir fogo, é necessário que sejam determinadas posições de tiro onde a artilharia se posicionará antes de atirar. No SWORD, essas posições são tratadas como objetos chamados de *Firing Posts*, que precisam ser construídos pela bateria de artilharia de reconhecimento. Antes disso, porém, a área onde esses objetos serão construídos deve ser reconhecida e a zona de reunião deve ser determinada. Para isso, uma única ordem denominada *Recon Deployment Sites*, que pode ser dada a bateria de reconhecimento, realiza todo a movimentação necessária e constrói as posições de tiro. Os parâmetros obrigatórios dessa ordem são as coordenadas do ponto de reunião, a direção do perigo, os limites de deslocamento e as posições onde serão construídos os *Firing Posts*. A Figura 6 traz o exemplo de uma janela de criação de ordens de um autômato qualquer.

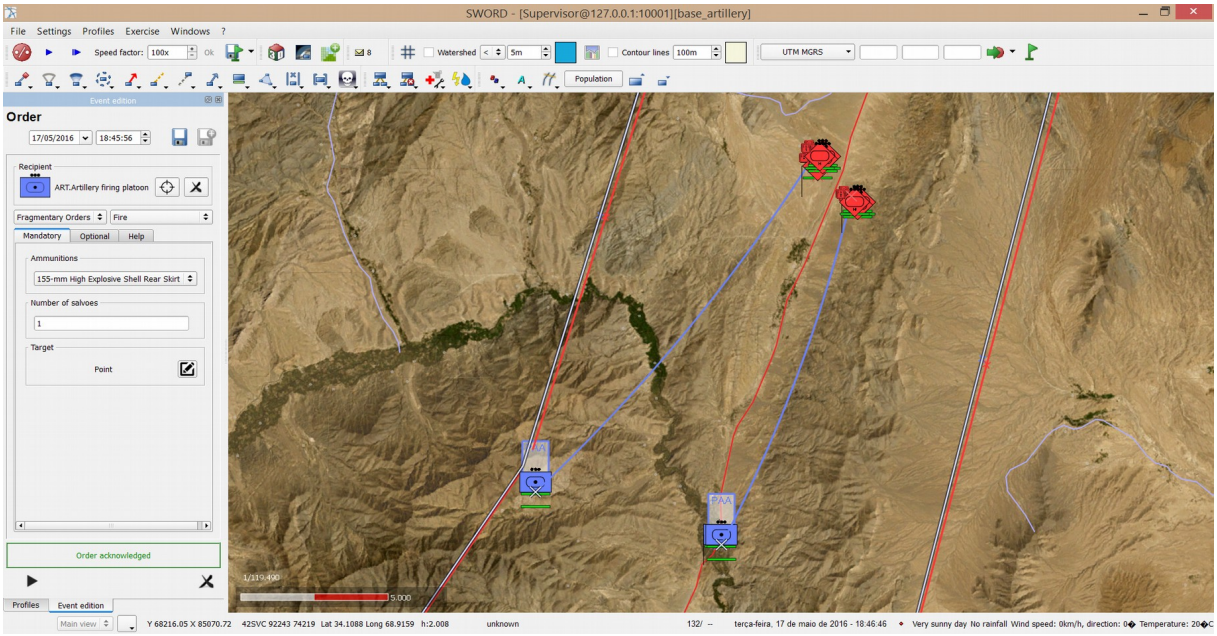
Figura 6 – Captura de tela da janela de criação de ordens



Fonte: Simulador MASA SWORD.

Quando a construção for completada, a bateria de disparo deve se posicionar junto aos *Firing Posts*, através da ordem *Deploy (On Deployment Sites)*. Se foram construídos mais de um *Firing Posts*, essa ordem faz com que os pelotões que compõe a bateria de disparo se posicionem em *Firing Posts* diferentes. Finalizada essa ordem, deve-se selecionar o autômato correspondente a bateria de artilharia de disparo e separar os pelotões que o compõe (através da opção disengage). Por fim, deve-se dar a cada unidade a ordem fragmentária de disparar, denominada *Fire*, determinado os parâmetros de posição do alvo, número de disparos e munição a ser utilizada. A Figura 7 mostra o exato momento em que a ordem *Fire* é executada no exercício, para duas unidades distintas. À esquerda da figura está a janela de criação de ordem.

Figura 7 – Captura de tela da execução da ordem fragmentária de disparar no exercício base_artillery



Fonte: Simulador MASA SWORD.

Após a execução do exercício, pode-se revisar o mesmo no módulo replay do MASA SWORD, bem como os logs gerados pela sessão, que mostram toda a troca de mensagens entre os diferentes componentes do simulador (Figuras 8 e 9).

Figura 8 – Recorte do arquivo `Dispatcher.log`

```
19 [2016-05-31 00:25:49] <Dispatcher> <info> 0 clients connected
20 [2016-05-31 00:25:50] <Dispatcher> <info> **** Time tick 3 - Memory: 314.398 MB / 772.77 MB (VM)
21 [2016-05-31 00:25:50] <Dispatcher> <info> Connection received from client '127.0.0.1:59079'
22 [2016-05-31 00:25:50] <Dispatcher> <info> 1 clients connected
23 [2016-05-31 00:25:50] <Dispatcher> <info> 0 clients authentified
24 [2016-05-31 00:25:51] <Dispatcher> <info> **** Time tick 4 - Memory: 315.445 MB / 772.77 MB (VM)
25 [2016-05-31 00:25:52] <Dispatcher> <info> **** Time tick 5 - Memory: 315.859 MB / 772.77 MB (VM)
26 [2016-05-31 00:25:53] <Dispatcher> <info> Connection received from client '127.0.0.1:59082'
27 [2016-05-31 00:25:53] <Dispatcher> <info> 2 clients connected
28 [2016-05-31 00:25:53] <Dispatcher> <info> Auth - Profile 'Blue User' authenticated (127.0.0.1:59082)
29 [2016-05-31 00:25:53] <Dispatcher> <info> 1 clients authentified
30 [2016-05-31 00:25:53] <Dispatcher> <info> **** Time tick 6 - Memory: 316.445 MB / 772.77 MB (VM)
31 [2016-05-31 00:25:54] <Dispatcher> <info> **** Time tick 7 - Memory: 316.754 MB / 772.77 MB (VM)
32 [2016-05-31 00:25:55] <Dispatcher> <info> /select
```

Fonte: Dispatcher.log.

Figura 9 – Recorte do arquivo `Protobuf.log`

```
1 Dispatcher sent : context: 0 message { control_information { current_tick: 2 initial_date_time { data:
  "20160517T152956" } date_time { data: "20160517T153006" } tick_duration: 10 time_factor: 10
  checkpoint_frequency: 360000000 status: running send_vision_cones: false profiling_enabled: false } }
2 Dispatcher sent : context: 0 message { control_send_current_state_begin { } }
3 Dispatcher sent : context: 0 message { party_creation { party { id: 215 } name: "Blue Army" type:
  friendly } }
4 Dispatcher sent : context: 0 message { knowledge_group_creation { knowledge_group { id: 216 } party {
  id: 215 } type: "Standard" name: "Knowledge group [216]" crowd: false } }
5 Dispatcher sent : context: 0 message { knowledge_group_creation { knowledge_group { id: 403 } party {
  id: 215 } type: "Standard" name: "knowledge group[403]" crowd: true } }
6 Dispatcher sent : context: 0 message { formation_creation { formation { id: 220 } party { id: 215 }
  level: x name: "Blue Brigade" app6symbol: "symbols/sfgpucfhe" logistic_level: none symbol:
  "symbols/sfgpucfhe" log_maintenance_manual: false log_supply_manual: false } }
7 Dispatcher sent : context: 0 message { automat_creation { automat { id: 377 } type { id: 162 } name:
  "ART.Artillery firing battery" parent { formation { id: 220 } } party { id: 215 } knowledge_group { id:
  216 } app6symbol: "symbols/sfgpucfhe" logistic_level: none symbol: "symbols/sfgpucfhe"
  log_maintenance_manual: false log_supply_manual: false } }
8 Dispatcher sent : context: 0 message { unit_creation { unit { id: 378 } type { id: 26 } name:
  "ART.Battery CP" automat { id: 377 } pc: true repartition { male: 0 female: 0 children: 0 } } }
9 Dispatcher sent : context: 0 message { unit_creation { unit { id: 379 } type { id: 79 } name:
  "ART.Artillery firing platoon" automat { id: 377 } pc: false repartition { male: 0 female: 0 children: 0
  } } }
```

Fonte: Protobuf.log.

4.1.2 API de rede

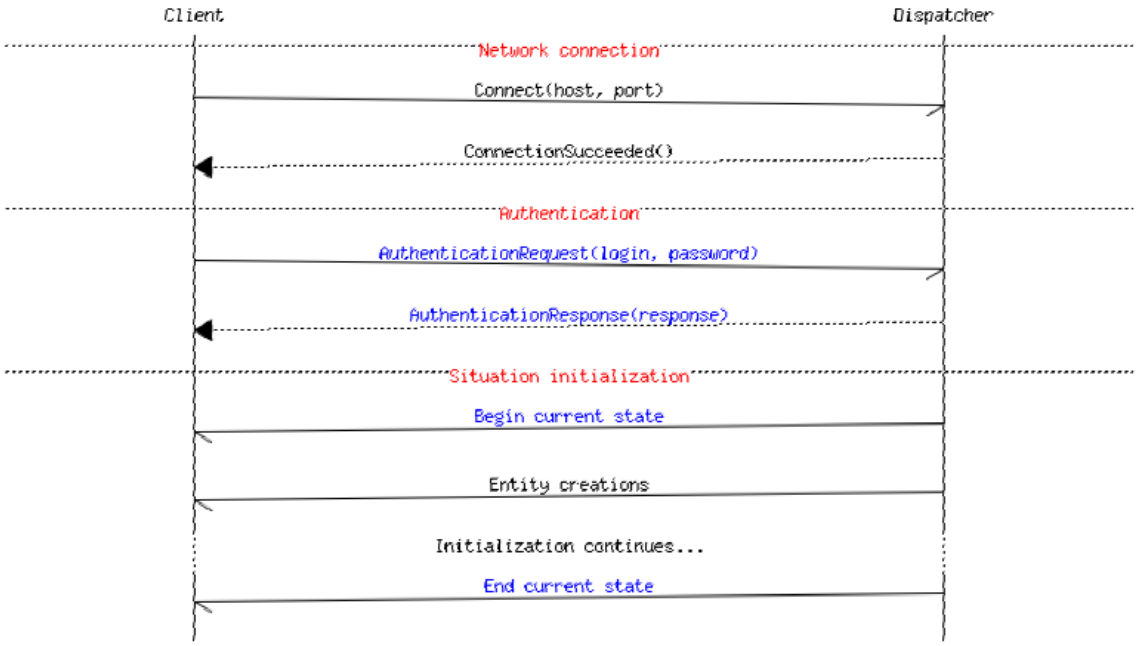
A API de rede do SWORD foi desenvolvida com o objetivo de permitir que aplicações remotas sejam integradas aos seus exercícios e simulações. Assim, os usuários finais podem customizar suas experiências da forma que considerarem mais adequada e efetiva. Ela implementa o protocolo TCP/IP, onde o simulador é o servidor e a aplicação remota o cliente. A conexão se dá, geralmente, através da porta 10001. A porta pode mudar em cada sessão e está definida no arquivo `session.xml`, dentro da pasta do exercício em questão.

Após estabelecida a conexão, a comunicação se inicia com a fase de autenticação. O cliente deve enviar ao servidor uma mensagem de requisição de autenticação, que deve conter um perfil de usuário e uma senha. A partir desta conexão inicial, o servidor responde enviando uma série de mensagens que inicializam os diversos perfis (conectados ou não), e suas permissões. Se o cliente tentar se conectar com um perfil inválido, ele recebe uma lista de perfis e se possuem ou não senha para que tente se conectar novamente.

Terminada com sucesso a fase de autenticação, o servidor começa a enviar mensagens ao cliente para inicializar o estado atual da simulação. Essa fase é

chamada de fase de inicialização. A Figura 10 demonstra com um diagrama as diferentes fases descritas anteriormente.

Figura 10 – Diagrama de comunicação da API de rede do simulador MASA SWORD, representando as fases de conexão, autenticação e inicialização do cliente com o exercício



Fonte: Sword API Training.

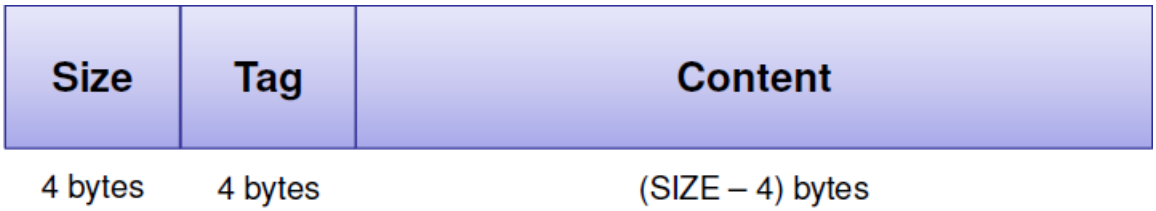
Durante a fase de inicialização, o cliente recebe o estado global da simulação e mensagens que visam criar as entidades existentes na simulação e, após sua criação, seus respectivos atributos. Essas primeiras mensagens são as chamadas mensagens de inicialização. Outra classificação, as mensagens de pedido, descrevem ordens enviadas pelo cliente para unidades individuais e agregadas (conjuntos de unidades subordinadas a outra unidade) e as respostas do sistema. Nessa situação, é enviada uma ordem pelo cliente ao servidor, que responde com uma mensagem de *acknowledge* e, depois, envia a ordem para todos os outros clientes. Se a ordem foi enviada a um autômato, o servidor envia a ordem para cada unidade ligada a ele, para todos os clientes.

A terceira e última classificação são as mensagens de conhecimento, que descrevem novas informações adquiridas pelas unidades e grupos de

conhecimentos (unidades que participam de um determinado canal de comunicação).

As mensagens trocadas na rede são codificadas em três campos: tamanho, tag e conteúdo. A Figura 11 apresenta um modelo do formato das mensagens. O campo *Size* ocupa 4 bytes e informa o tamanho do resto da mensagem. O campo *Tag* os próximos 4 bytes e determina o tipo de mensagem que está codificada. Por fim, o campo *Content* é o objeto mensagem serializado em um vetor de bytes. Para tornar mais claro: a codificação é a construção da mensagem que será enviada ao servidor e a serialização é a transformação do objeto em um vetor de bytes para compor o campo conteúdo da mensagem codificada. O objeto mensagem nada mais é, portanto, do que um trecho da mensagem codificada.

Figura 11 – Formato das mensagens



Fonte: Sword API Training.

4.2 FUNCIONALIDADES VIA PROTOBUF

A ferramenta utilizada pela API de Rede na codificação das mensagens trocadas entre cliente e servidor é o Protobuf. As mensagens estão definidas em arquivos de extensão .proto, que são necessários para que possa ser realizada a comunicação entre cliente e servidor. Neles, estão definidos os formatos das mensagens utilizadas na simulação e seus respectivos métodos de manipulação. São fornecidos, no total, 14 arquivos que implementam diferentes tipos de mensagem. Através do compilador protocol buffer, esses arquivos podem ser compilados em classes Java, C++ ou Python para serem utilizados pelo cliente. Essas classes oferecem construtores para todos os tipos de mensagens utilizados pela API, bem como interfaces e métodos para definição e resgate de valores (get e

set).

Cada um dos arquivos *.proto* possui uma classe *Message*, que o cliente utiliza para construir o objeto que será serializado e enviado para o servidor. O objeto em si é imutável, portanto, para atribuir valores aos campos pertencentes à mensagem, o cliente deve primeiro criar um construtor – em Java, *Message.Builder* – e atribuir os valores aos campos da classe. Depois que todos os campos receberem valores, deve-se construir o objeto através do respectivo método – em Java, *.build()* – e a mensagem estará pronta para ser serializada. Os métodos de serialização e desserialização também são fornecidos pela classe *Message* – em Java, *toByteArray()* e *mergeFrom()*.

4.3 DETALHAMENTO DA SOLUÇÃO

O cliente foi desenvolvido na linguagem Java, utilizando a NetBeans IDE. Ele possui uma janela de interface gráfica que serve para facilitar a visualização do fluxo de mensagens e das diferentes fases de comunicação. O cliente possui um conjunto de 14 classes Java geradas pelo compilador *protocol buffer* a partir dos arquivos *proto* mencionados anteriormente, que definem as classes – e seus respectivos métodos – dos diferentes tipos de mensagem que o cliente pode receber do *Dispatcher*. Os Anexos de A à K apresentam algumas dessas classes.

Para realizar o envio de uma mensagem, é necessário primeiro codificá-la. O tamanho das mensagens depois de codificadas é o primeiro item a compor a codificação das mensagens. Esse valor consiste na soma do tamanho do objeto mensagem serializado mais quatro bytes correspondentes ao tamanho do item *Tag*. O tamanho das mensagens serializadas é um valor retornado por um método – em Java, *getSerializedSize()* – presente em todos os tipos de mensagem. Esse item é guardado em um espaço de 4 bytes.

Para identificar o tipo de cada mensagem recebida e realizar sua codificação e desserialização de forma correta são utilizadas varíaveis tag que são guardadas no cliente. No momento da codificação, a tag é alocada também em um espaço de 4 bytes, logo depois dos 4 bytes que definem o tamanho do resto da mensagem. Já no momento da desserialização, elas são utilizadas para definir para qual tipo de

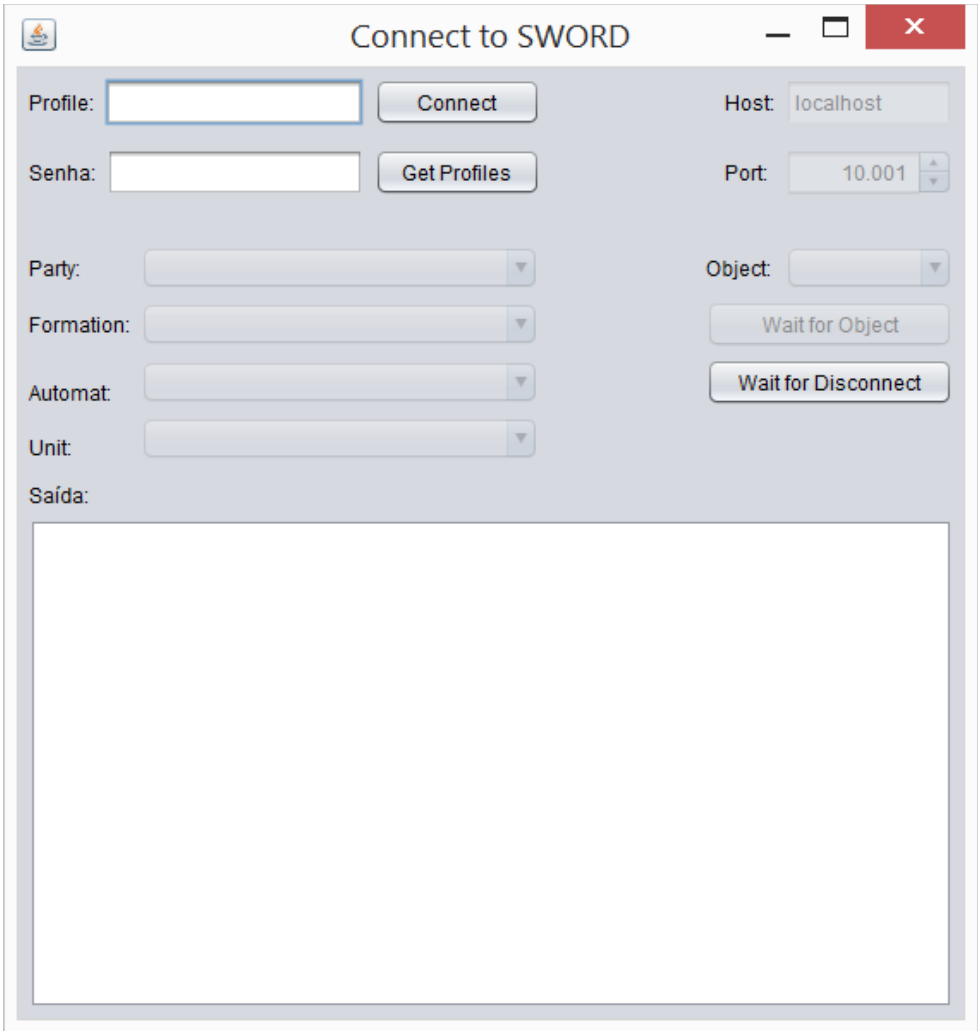
mensagem a desserialização deve ser aplicada. A consistência na transmissão das mensagens é de extrema importância, pois um mísero byte que seja lido a mais ou a menos irá embaralhar todo o fluxo de rede.

Por fim, a mensagem é serializada através de um método – em Java, *toByteArray()*, como mencionado anteriormente – que retorna um vetor de bytes. Então são transmitidos, na seguinte ordem, o tamanho da mensagem codificada, a tag que define o tipo de mensagem e o objeto serializado.

Na decodificação lê-se os primeiros 4 bytes que trazem o tamanho do resto da mensagem – em Java, que são convertidos para um tipo *long* – e então lê-se o resto dos bytes com base no tamanho. A partir do campo *Tag* é identificado o tipo da mensagem e então é realizada a desserialização.

A janela de interface gráfica é implementada na classe *ConnectGUI.java*, e seu estado inicial se encontra na Figura 12. O botão *Connect* tenta realizar a conexão através de sockets utilizando os valores de *Host* e *Port*. Caso tenha sucesso, ele tenta realizar a autenticação do perfil de usuário e senha informados nos campos *Profile* e *Senha*. O botão *Get Profiles* envia um perfil de usuário e senha nulos. Essa é a primeira mensagem trocada entre cliente e servidor na aplicação remota, e ela é criada a partir da definição de uma mensagem do tipo *AuthenticationRequest*, presente na classe *ClientAuthentication*.

Figura 12 – Interface gráfica, fase de autenticação



Fonte: ConnectGUI.java.

Após o envio da mensagem contendo os dados do perfil de usuário, o cliente receberá como resposta mensagens contendo os perfis cadastrados e os já conectados, ou mensagens que confirmam sua conexão à simulação, dependendo do sucesso da tentativa de autenticação. Quando a autenticação tem sucesso, o servidor começa a enviar as mensagens de inicialização da simulação ao cliente. As mensagens recebidas pelo cliente durante o restante da fase de autenticação são do tipo *AutenthicationClient* e as mensagens recebidas durante a inicialização são do tipo *SimulationClient*.

Depois da etapa de inicialização, os componentes *Party*, *Formation*, *Automat* e *Unit* da interface recebem seus itens. O campo *Party* é definido em função das

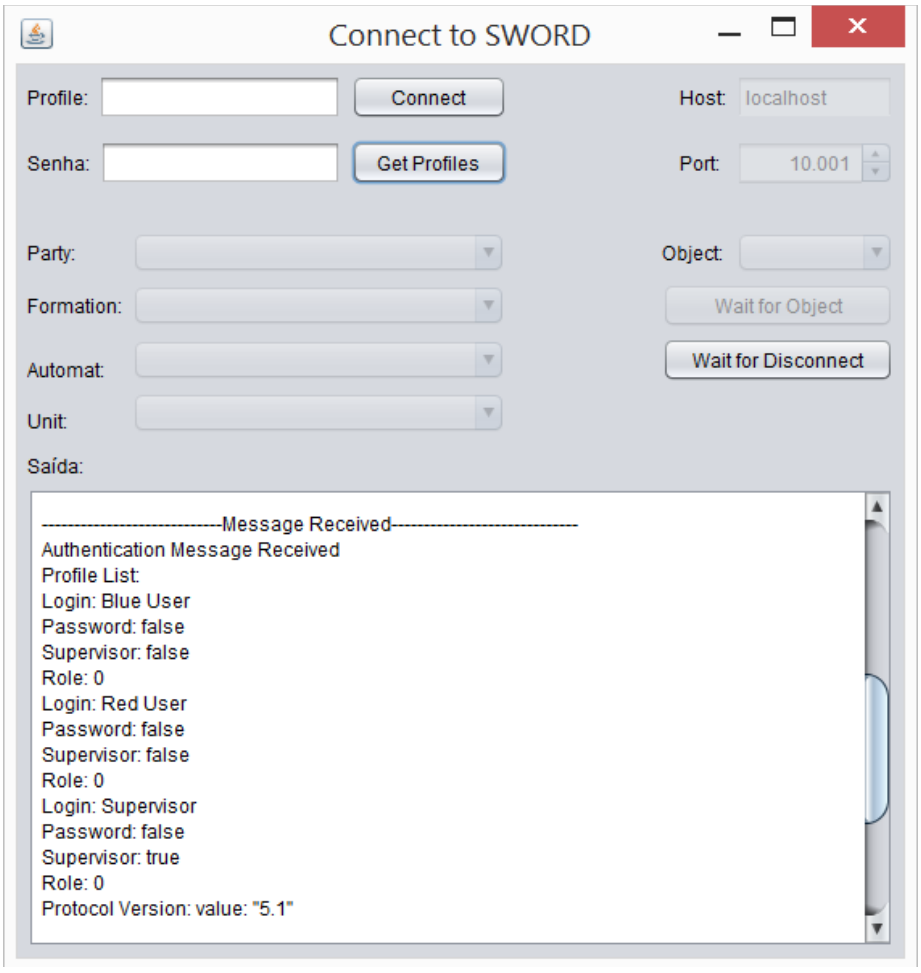
permissões do perfil do usuário que se conectou e, consequentemente, todos os outros campos dependem do campo imediatamente acima. Essa abordagem permite uma visualização mais intuitiva das unidades limitando, por exemplo, que o campo *Unit* mostre alguém que não pertence ao autômato selecionado em *Automat*. Em um primeiro momento, são transmitidas mensagens de criação – por exemplo, o tipo *PartyCreation*, *FormationCreation*, *AutomatCreation* e *UnitCreation* – e após são enviadas outras que trazem características – como *AutomatAttributes* e *UnitAttributes* – o que garante que os componentes já tenham sido criados antes de serem referenciados em mensagens posteriores.

Por fim, o cliente estará integrado ao exercício. A partir do momento em que a etapa de inicialização finaliza, todas as mensagens que o cliente receber serão para atualizar o estado da simulação, seja por alguma interferência dos outros usuários conectados, seja por mensagens de atualização do próprio simulador. Exemplos do primeiro caso são ordens ou uma pausa na simulação. No segundo caso, podemos incluir mensagens de tique-taque (*time tick*).

5 EXPERIMENTOS E TESTES

Como experimento, o exercício apresentado na seção 4.1.1 foi executado em conjunto com o cliente remoto. Depois de iniciados o exercício (*SWORD Gaming*) e o cliente, o próximo passo é solicitar, como cliente remoto, a lista de perfis de usuário com permissão de acesso ao exercício em questão. Essa informação pode ser recebida por duas formas: através do botão *Get Profiles* ou tentando se conectar com um perfil inválido. A mensagem de resposta pode ser vista na Figura 13.

Figura 13 – Interface gráfica, solicitação de perfis



Fonte: ConnectGUI.java.

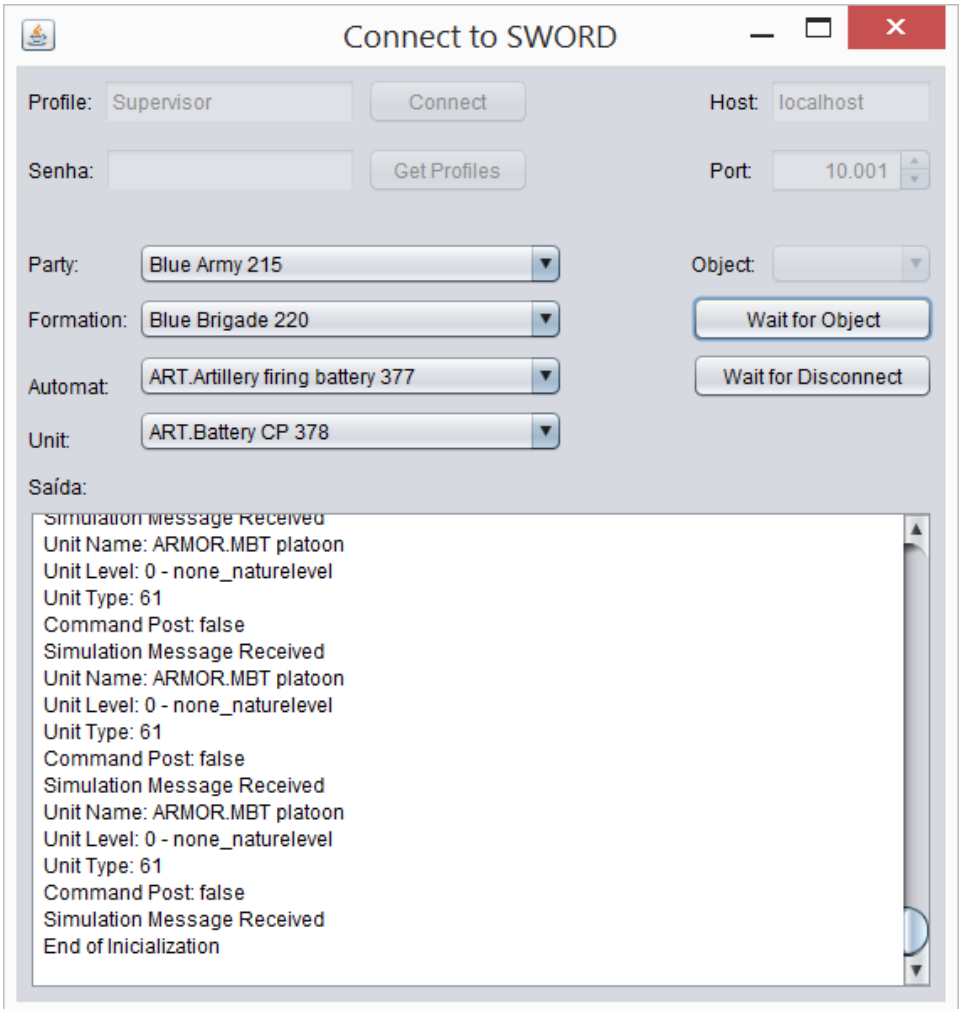
Depois de receber os perfis válidos para a simulação, o perfil desejado deve ser informado no campo *Profile*. Para facilitar o exemplo, a conexão foi feita no perfil

de *Supervisor*. Vale notar que, assim que a autenticação ocorre, a simulação reconhece a conexão do cliente remoto acendendo uma luz verde ao lado do perfil conectado, na aba *Profiles*.

Assim que é validada a conexão, o servidor começa a fase de inicialização. Essa fase começa com uma mensagem de inicialização do envio do estado atual da simulação (*ControlSendCurrentStateBegin*) e termina com uma mensagem de finalização do envio (*ControlSendCurrentStateEnd*). Ambas pertencem a classe *SimulationClient*. Durante o período de inicialização do estado atual da simulação, o cliente recebe informações acerca de todas as unidades que são suas subordinadas, além de inúmeras outras mensagens acerca da simulação, como mostra a Figura 14. Todas as mensagens enviadas entre a fase de inicialização e finalização do envio também pertencem a classe *SimulationClient*.

Depois da inicialização, o cliente pára de ler as mensagens da entrada até que sejam acionados ou o botão *Wait for Object* ou o botão *Wait for Disconnect*. Dessa forma, respectivamente, o cliente começa novamente a receber mensagens até ocorrer a criação de um objeto (no caso, um *Firing Post*) ou ele volta a receber mensagens até ser desconectado pela simulação (geralmente ocorre quando a simulação é finalizada). Dessa forma, o cliente registra todas as ordens que forem dadas. Primeiro aguardando a criação do objeto *Firing Post* e mostrando ele na interface e depois aguardando o resto das ordens que serão enviadas aos clientes.

Figura 14 – Interface gráfica, inicialização do estado atual da simulação



Fonte: ConnectGUI.java.

5.1 ENVIO DE ORDENS

O cliente remoto implementado apenas monitora a simulação. Porém, para torná-lo um participante ativo do exercício, seria interessante permitir que ele também enviasse ordens a suas entidades. Para realizar o experimento anterior, por exemplo, são necessárias no mínimo três ordens que são dadas às entidades para chegar ao objetivo final de atirar com uma unidade de artilharia. São elas as missões de autômatos *Recon Deployment Sites* e *Deploy (On Deployment Sites)* e a ordem fragmentária *Fire*. No momento em que uma ordem é dada, através da aplicação *Gaming*, é enviada uma mensagem para o componente *Dispatcher* contendo todos

os paramêtros necessários para que a simulação e os comportamentos de inteligência artificial inseridos nas unidades ajam conforme ordenado. O Dispatcher então reenvia essas mensagens, que são do tipo *SimulationClient*, para o resto dos clientes presentes no exercício. O cliente remoto imprime os parâmetros recebidos para que seja possível identificar como se dá a formação destas mensagens. Essas mensagens podem ser do tipo *AutomatOrder*, *UnitOrder* ou *FragOrder*. Cada uma delas exige, durante a construção de seu objeto, uma série de parâmetros que são encapsulados em outro objeto, *MissionParameters*.

De forma resumida, para que o cliente remoto possa enviar uma ordem para uma unidade ele precisaria saber o identificador da missão, conforme descrito no arquivo `Missions.xml` presente nas bases de dados do simulador e quais parâmetros são considerados obrigatórios para esta missão em específico (limites, direção dos inimigos, posição do alvo, posição do objetivo, tipo de objeto a ser criado, tipo de munição, entre dezenas de outros parâmetros). O meio mais fácil de realizar algo desse tipo seria mapear ordens específicas, como as três mencionadas no parágrafo anterior, e identificar a origem dos parâmetros que foram recebidos quando a ordem foi dada no *SWORD Gaming*. Infelizmente não foi possível, para este trabalho, aumentar o papel do cliente remoto. Apesar disso, deixa-se uma essa seção como uma introdução às ferramentas necessárias para realizar essa tarefa.

5.2 DIFICULDADES ENCONTRADAS

Uma das maiores dificuldades encontradas durante a fase de implementação deste cliente foi a realização do primeiro contato entre o cliente e o servidor. O grupo MASA forneceu um exemplo de cliente remoto, porém implementado em C++ e não funcional. O primeiro problema se deu na construção dos objetos mensagem. O exemplo implementado em C++ trazia métodos de construção de objetos completamente diferente da forma utilizada pelos construtores gerados, pelo protocol buffers, para Java. Assim, foi preciso compreender os métodos de construção utilizados nas duas linguagens para que, em um primeiro momento, fosse possível compreender o resultado do método em C++ e depois fosse descoberto qual era o método equivalente em Java. Esse contato inicial ocupou

grande parte do tempo de desenvolvimento do cliente.

A segunda situação ainda diz respeito ao primeiro contato ou mais especificamente, à troca de mensagens. Os tipos utilizados para guardar os componentes das mensagens (tamanho, tag e conteúdo) dependiam muito de variáveis sem sinal (*unsigned*) no exemplo. Além disso, os tipos int e long, na linguagem Java, possuem tamanhos diferentes dos seus respectivos tipos em C++. Até que as conversões entre tipos de variáveis e bytes e a montagem e leitura dos arrays de bytes das mensagens em fluxo na rede se deram de forma correta, outra parte do desenvolvimento se perdeu.

Por fim, o terceiro e último problema já foi abordado na seção 5.1. Permitir que o cliente ordenasse suas unidades foi um objetivo que, na fase final da implementação, foi perseguido. Como dito anteriormente, existem diversos parâmetros obrigatórios que devem ser enviados para a simulação junto a ordem. Dois em especial acabaram se tornando os mais difíceis de identificar. O primeiro é o parâmetro identificador do tipo do missão. Não foi possível encontrar um local que guardasse esses identificadores nas classes fornecidas pelo simulador. A única forma portanto de descobrir esses identificadores seria buscando esses valores nos logs e colocando-os de forma estática para cada ordem que fosse implementada no cliente. O segundo são os parâmetros que envolvem coordenadas, que também teriam que ser buscadas em logs ou na aplicação *Gaming* e informadas de forma estática, no código fonte. Esses dois problemas poderiam ser resolvidos de formas semelhantes, porém eles fazem parte de uma questão muito maior: a construção de um objeto de qualquer tipo ordem (mencionados anteriormente) ocuparia de 20 a 40 linhas de código. O tipo AutomatOrder, por exemplo, precisa que sejam setados diversos outros objetos que também precisam ser construídos, como *MissionParameters*, *Value*, *Point*, *Location* e *CoordLatLong*. Cada um desses objetos exige também outros, mais genéricos, e a complexidade aumenta para cada objeto que precisa ser construído. Por esse motivo as tentativas de envio de ordens acabaram não sendo frutíferas, e continuar tentando realiza-lo acabaria atrasando ainda mais o trabalho. Também tentou-se utilizar ordens mágicas (*UnitMagicAction*), que são um tipo de ordem reservadas a usuários com privilégios de supervisor, mas os parâmetros exigidos eram os mesmos.

6 CONCLUSÃO

O objetivo deste trabalho dentro do âmbito acadêmico era explorar as possibilidades de montar um cliente remoto que se integrasse a exercícios do simulador construtivo MASA SWORD, com foco na troca de mensagens utilizando a ferramenta *protocol buffers*. Este objetivo foi vencido.

O cliente projetado e implementado conversa com o simulador através de uma conexão TCP/IP, e eles trocam mensagens acerca do estado da simulação. O cliente se autentica com um perfil válido, identifica suas unidades subordinadas e seus respectivos atributos durante a fase de inicialização e por fim identifica/decodifica as mensagens relevantes para sua execução durante o resto da simulação.

Para trabalhos futuros, é extremamente interessante ter um cliente remoto que consiga ser mais ativo em sua participação no exercício. Isto é, ter um cliente capaz de enviar ordens mágicas, missões e até mesmo que consiga parar e recomeçar a simulação de forma remota.

Como o simulador MASA SWORD é um simulador construtivo, tem-se a utilização de diversos comportamentos de inteligência artificial, implementados nas unidades autônomas para que elas ajam conforme as ordens que receberem dos usuários. Outro desdobramento interessante seria utilizar estes comportamentos em outras aplicações, realizando o acesso ao SWORD apenas para capturar esses comportamentos para uma aplicação externa.

Esse trabalho se insere em um contexto maior na forma de projeto que visa explorar a HLA, que também permite essa integração, mas em mais alto nível por se tratar de uma arquitetura para interoperabilidade entre simulações. A execução de um cliente remoto representa, portanto, uma abordagem bem sucedida dentro de um esforço maior para criar sistemas e simulações que trabalham em conjunto com o simulador construtivo MASA SWORD.

REFERÊNCIAS

ALLEN, T. B. **War Games**. New York: Berkeley Books, 1987 apud SMITH, 2010, p. 5;

CALDWELL, B.; HARTMAN, J.; PARRY, S. **Aggregated Combat Models**. Monterey, California: Navy Postgraduate School, 2000. cap. 1, p. 1-4.

DAHMANN, J. S.; FUJIMOTO, R. M.; WEATHERLY, R. M. **The Department of Defense High Level Architecture**. Proceedings of the 1997 Winter Simulation Conference. p. 142-149, 1997.

DEPARTMENT OF DEFENSE. **Modeling and Simulation (M & S) Glossary**. 2011.

FLATO, U.; GUIMARÃES, H. **Educação baseada em simulação em medicina de urgência e emergência: a arte imita a vida**. Revista Brasileira de Clínica Médica, v. 9, n. 5, p. 5–9, 2011.

FUJIMOTO, R. M. **Parallel and distributed simulation systems**. New York: John Wiley & Sons, Inc., 2000. cap. 1 p. 3-25.

GOOGLE protocol buffers. In: DEVELOPERS GUIDE. California: Google, 2016. Disponível em: <https://developers.google.com/protocol-buffers/docs/overview>.

HENNINGER, A. E. et al. **Live Virtual Constructive Architecture Roadmap (LVCAR) Final Report**. n. 09, p. 73, 2008.

HOLMAN, G. **Training Effectiveness of the CH-47 Flight Simulator**, U.S. Army Research Institute for Behavioral and Social Sciences, 1979.

ISSENBERG, S. B. et al. **Features and uses of high-fidelity medical simulations that lead to effective learning: a BEME systematic review**. Medical teacher, v. 27, n. 1, p. 10–28, 2005.

LATEEF, F. **Simulation-Based Learning: Just like the real thing**. [S.l.]: Journal of Emergencies, Trauma and Shock 3.4, p. 348–352, 2010.

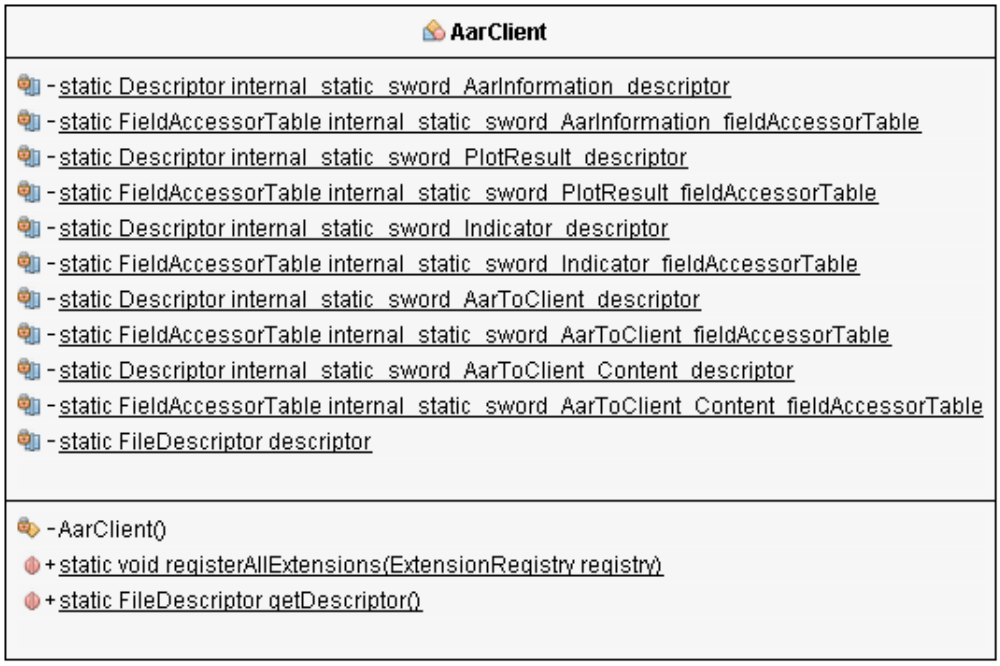
PADILLA, J. J.; DIALLO, S. Y.; TOLK, A. **Do We Need M & S Science?** SCS M&S Magazine, v. 4, p. 161–166, 2011.

ŠIMIĆ, G. Constructive simulation as a collaborative learning tool in education and training of crisis staff. **Interdisciplinary Journal of Information, Knowledge, and Management**, v. 7, p. 221–236, 2012.

SMITH, R. The Long History of Gaming in Military Training. **Simulation & Gaming**, v. 41, n. 1, p. 6–19, 2010.

ANEXO A – DIAGRAMA DA CLASSE AARCLIENT

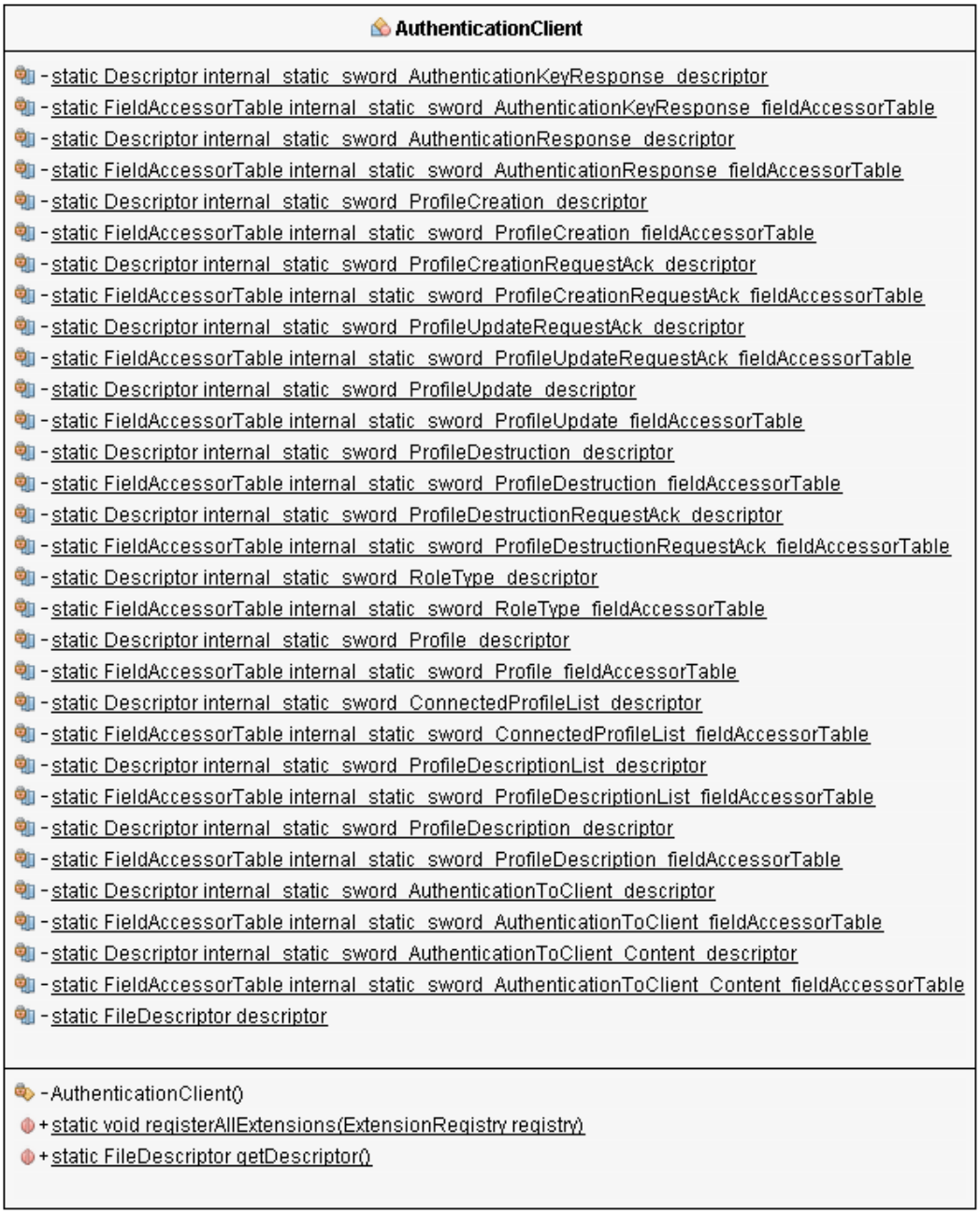
Figura 15 – Classe `AarClient.java`



Fonte: easyUML.

ANEXO B – DIAGRAMA DA CLASSE AUTHENTICATIONCLIENT

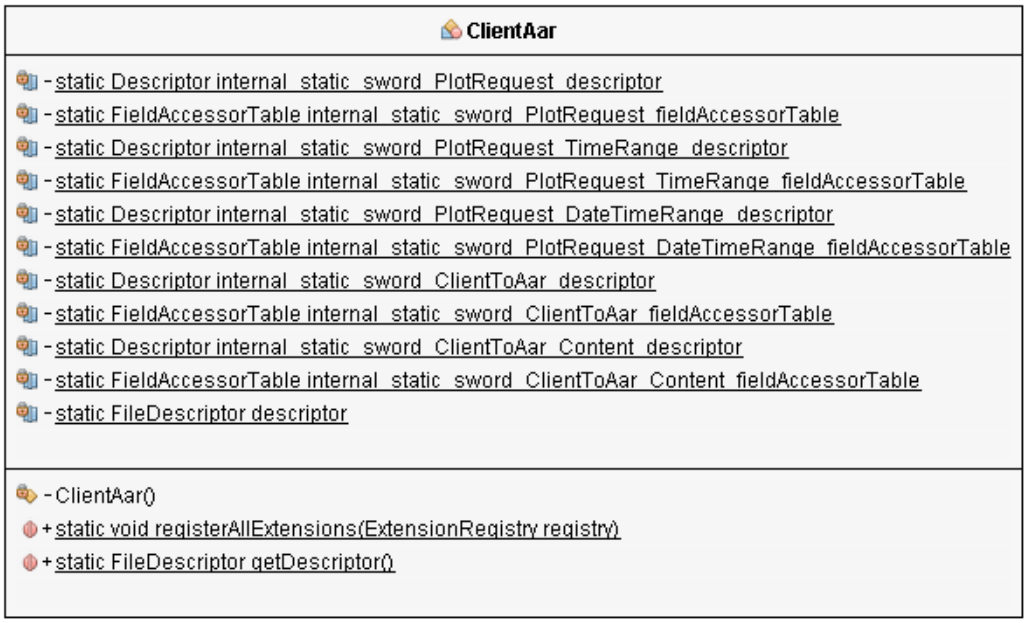
Figura 16 – Classe `AuthenticationClient.java`



Fonte: easyUML.

ANEXO C – DIAGRAMA DA CLASSE CLIENTAAR

Figura 17 – Classe `ClientAar.java`



Fonte: easyUML.

ANEXO D – DIAGRAMA DA CLASSE CLIENTAUTHENTICATION

Figura 18 – Classe `ClientAuthentication.java`

Fonte: easyUML.

ANEXO E – DIAGRAMA DA CLASSE CLIENTMESSENGER

Figura 19 – Classe `ClientMessenger.java`

Fonte: easyUML.

ANEXO F – DIAGRAMA DA CLASSE CLIENTREPLAY

Figura 20 – Classe `ClientReplay.java`

Fonte: easyUML.

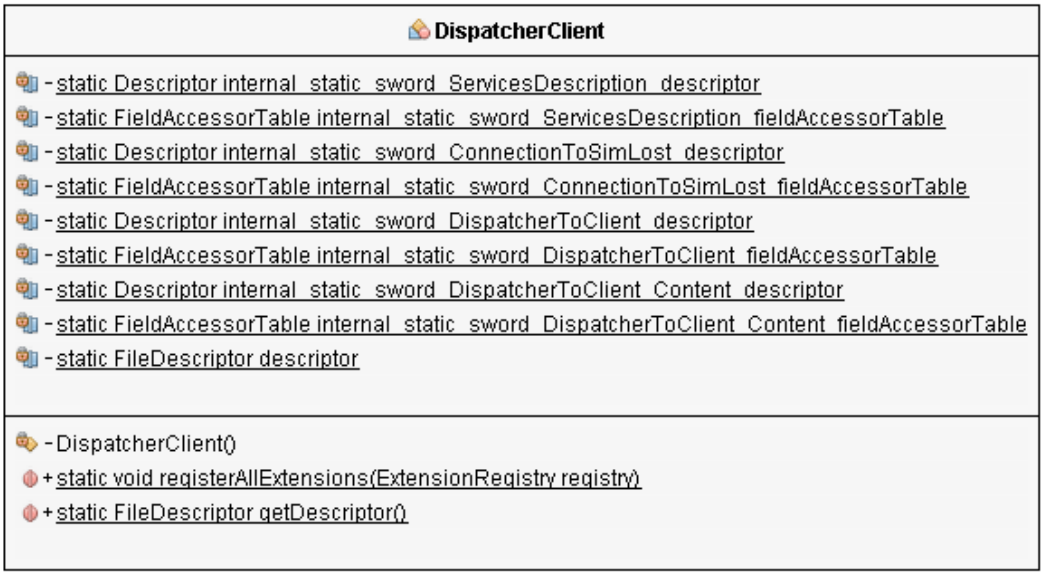
ANEXO G – DIAGRAMA DA CLASSE CLIENTSIMULATION

Figura 21 – Classe `ClientSimulation.java`

Fonte: easyUML.

ANEXO H – DIAGRAMA DA CLASSE DISPATCHERCLIENT

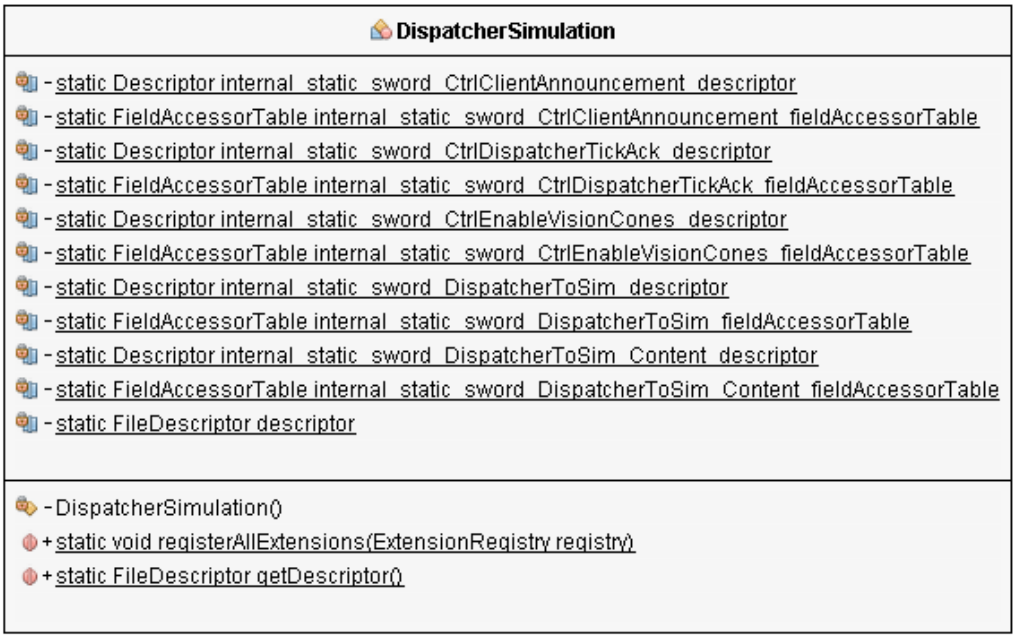
Figura 22 – Classe `DispatcherClient.java`



Fonte: easyUML.

ANEXO I – DIAGRAMA DA CLASSE DISPATCHERSIMULATION

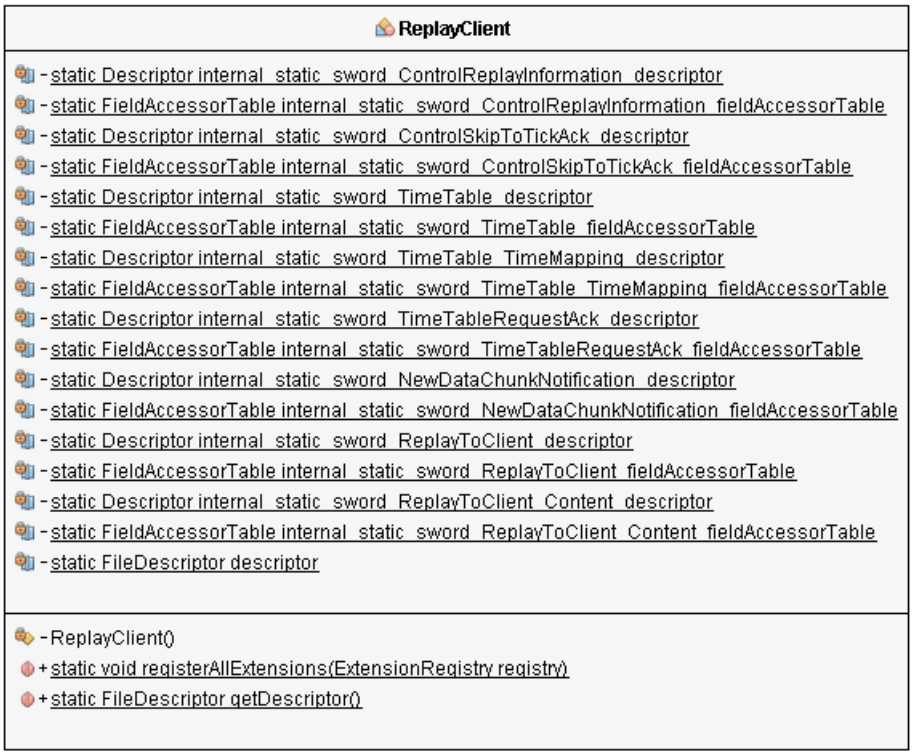
Figura 23 – Classe `DispatcherSimulation.java`



Fonte: easyUML.

ANEXO J – DIAGRAMA DA CLASSE REPLAYCLIENT

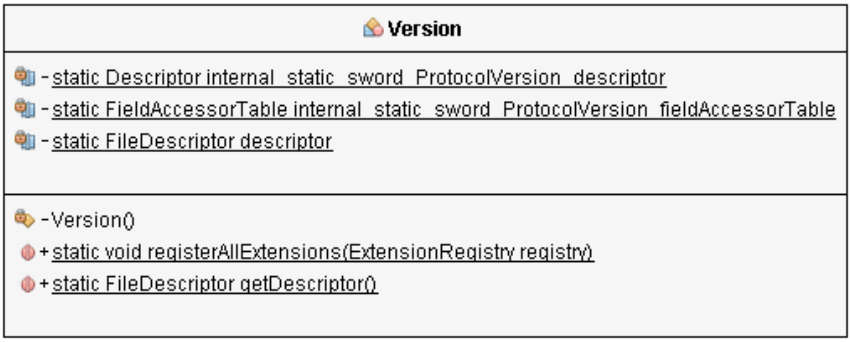
Figura 24 – Classe `ReplayClient.java`



Fonte: easyUML.

ANEXO K – DIAGRAMA DA CLASSE VERSION

Figura 25 – Classe `Version.java`



Fonte: easyUML.