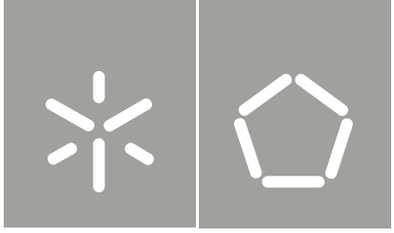
Universidade do Minho
Escola de Engenharia

Laurent Pereira Miranda

Monitorização e Registo de Movimento em Simulação de Ambientes Urbanos

Dezembro de 2012

Monitorização e Registo de Movimento em Simulação de Ambientes Urbanos
Laurent Pereira Miranda
UMinho | 2012



Universidade do Minho
Escola de Engenharia

Laurent Pereira Miranda

Monitorização e Registo de Movimento em Simulação de Ambientes Urbanos

Tese de Mestrado
Ciclo de Estudos Integrados Conducentes ao
Grau de Mestre em Engenharia de Comunicações

Trabalho efetuado sob a orientação de
Professor Doutor Adriano Moreira
Professora Doutora Maria João Nicolau

Dezembro de 2012

Agradecimentos

A conclusão deste estudo não teria sido possível sem a contribuição de algumas pessoas, às quais quero apresentar aqui a meu agradecimento.

Quero agradecer ao Professor Doutor Adriano Moreira e à Professora Doutora Maria João Nicolau, pela disponibilidade, empenho, dedicação e orientações durante a realização deste trabalho.

À minha mãe, ao meu irmão e ao companheiro da minha mãe, um agradecimento muito especial pelo apoio e auxílio que prestaram no meu dia a dia durante todos estes anos da minha formação académica.

Ao meu pai, que apesar de não estar presente fisicamente, está sempre presente no meu coração.

À minha namorada Cristiana Costa por toda a paciência, compreensão e carinho que teve comigo, por ter sido o meu suporte nos dias em que o desânimo era mais forte, e por estar ao meu lado em todos os momentos.

A todos os amigos que se cruzaram comigo no meu percurso académico e me fizeram crescer como estudante, com especial destaque para o meu colega de quarto Gui Andrade e amigos Luís Ferreira, Freddy Gonçalves, Diogo Mendes e Hélder Ribeiro pelo companheirismo e apoio.

Por fim, mas não menos importante, quero agradecer aos meus colegas que formaram equipa que trabalhou comigo neste projeto, o Rui Pinheiro e Francisco Silva, pela entreajuda e companheirismo.

A todos, muito obrigado!

Resumo

Nos dias que correm, os dispositivos móveis com capacidades de comunicação fazem parte da vida urbana, estando presentes em todo o lado num centro urbano. Este aglomerado de dispositivos indica-nos a possibilidade de se interligarem entre si formando um cenário de "todos ligados". Com este cenário surge a necessidade de criar novos serviços e aplicações que disponibilizam formas de interações diretas entre os dispositivos de maneira a atrair consumidores e operadoras.

Atualmente existem diversos modelos de mobilidade que tentam simular ambientes urbanos, mas estes simuladores não apresentam ainda conclusões satisfatórias, pois ainda não operam em grande escala, nem possuem grande detalhe de comportamento a nível dos atores. Estas lacunas servem de ponto de partida para o objetivo deste trabalho. Assim, pretende-se criar um simulador de mobilidade em ambientes urbanos que, para além de simular as movimentações de diversos tipos de atores neste ambiente, permita também simular as possíveis interações entre os diversos intervenientes. O simulador a desenvolver terá de ser capaz de simular um ambiente urbano com todos os tipos de intervenientes que é possível encontrar, como pessoas, carros, autocarros, bicicletas, etc. Esses atores, serão representados neste simulador, com o objetivo de agirem de forma o mais realista possível no decorrer da simulação.

Nesta dissertação é abordado o desenvolvimento do componente de visualização do simulador, registo da simulação e integração de mapas OSM com o mesmo. De forma a certificar o correto funcionamento das funcionalidades desenvolvidas, foram efetuados testes de desempenho e robustez que incluíram testes de funcionalidade, estabilidade, carga crítica, entre outros, sendo feita uma análise dos resultados obtidos em cada um deles.

Após a realização destes testes foi possível concluir que as componentes desenvolvidas nesta dissertação, para inclusão no simulador desenvolvido, alcançaram os objetivos pretendidos.

Abstract

In the present times, mobile devices with communication properties are a big part of urban life, being present everywhere in an urban center. This cluster of devices gives us the ability to interconnect with each other forming a scenario of “all connected”. With this scenario there is a need to create new services and applications that provide forms of direct interactions between devices in to order to attract consumers and operators.

Currently there are several mobility models that attempt to simulate urban environments, but these simulators have not yet satisfactory conclusions, they still do not operate on a large scale, nor do they have great detail at the behavior of actors. These gaps act as a starting point to the purpose of this work. Thus, it is intended to create a mobility simulator in urban environments, in addition to simulate the movement of different types of actors in this environment also allows simulating the possible interactions between different actors. The simulator developed must be able to simulate an urban environment with all types of players, such as people, cars, buses, bicycles, etc. the actor will be represented in this simulator, in order to act as realistically as possible.

These dissertation addresses the development of the visualization component of the simulator, registration and integration of OSM maps within the simulator. In order to ensure the correct operation of features developed, tests were performed that included performance and strength, including tests of functionality, stability, critical load, among others, and an analysis of the results in each.

After completion of these tests it was concluded that the components developed in this dissertation for inclusion in the simulator developed, achieved the intended objectives.

Conteúdo

Lista de figuras

Lista de tabelas

Lista de Acrónimos

API	Application Programming Interface
BartUM	Bartolomeu_Urban Mobility Simulator
IP	Internet Protocol
LAN	Local Area Network
ONE	Opportunistic Network Environment Simulator
OSM	Open Street Map
PDA	Personal Digital Assistant
SUMO	Simulation of Urban Mobility
TCP	Transmission Control Protocol
XML	Extensible Markup Language

1 Introdução

Para uma melhor introdução ao tema, este capítulo é composto por três partes. Na primeira parte é enquadrado o tema desta dissertação com a atualidade, bem como as suas funcionalidades práticas. Seguidamente, são expostos os objetivos pretendidos com este projeto. Por fim, é executada uma descrição da organização estrutural desta dissertação.

1.1 Enquadramento

Com a crescente utilização dos dispositivos móveis, é cada vez mais importante o estudo do desempenho dos sistemas de telecomunicações móveis, nomeadamente, em ambientes urbanos. Este estudo deve recorrer a modelos de mobilidade específicos (por exemplo, para pedestres e veículos automóveis), em vez de modelos genéricos de mobilidade. No entanto, os simuladores de mobilidade existentes continuam a ser muito genéricos, pois apesar de alguns simuladores possuírem características como capacidade de simular cenários urbanos em particular, outros serem capazes de simular um grande número de atores ou, ainda, realismo na movimentação dos atores, nenhum possui todas essas características simultaneamente.

Visto que os dispositivos móveis já estão profundamente inseridos nos centros urbanos, é importante dispor de um simulador que simule o movimento das entidades móveis (pedestres, veículos, entre outros), nestes cenários. Nos ambientes urbanos existem vários tipos de condicionantes que não permitem que os veículos e pedestres se desloquem livremente, por exemplo semáforos, sentidos obrigatórios das ruas, o facto dos veículos apenas poderem circular nas estradas, entre outros. Por outro lado, os passeios têm características próprias que influenciam o comportamento dos pedestres, por exemplo paragens de autocarros, lojas, aglomerados de pessoas, etc.

Os simuladores existentes não têm em conta todas as características referidas anteriormente, tratando todos os atores de igual forma. Para além disso, esses simuladores não reproduzem os aglomerados que os atores geram, por exemplo, em *shoppings* ou estações.

O simulador que se pretende desenvolver vai destacar-se em relação aos outros simuladores, já que deverá permitir a definição de cenários com base em mapas das estradas e a geração automática de tráfego, e tratará os vários tipos de atores existentes numa cidade, desde humanos, motos, carros, autocarros, até comboios e elétricos. Para além disso, os atores possuirão um comportamento que permitirá demonstrar de forma mais real que outros simuladores a mobilidade em ambientes urbanos.

Este simulador a apresentar deverá ser uma solução *open source* e multiplataforma. Para isso, será desenvolvida em *Java*. De forma a suportar um grande número de nós, este simulador funcionará num sistema distribuído, com distribuição de carga por vários computadores interligados numa rede LAN dedicada ao efeito. Este simulador permitirá ainda a interação entre diversos nós que estejam relativamente próximos, a visualização do decorrer da simulação e o registo da simulação.

1.2 Objetivos

Este trabalho integra-se num projeto de grupo, composto por 3 elementos, cada um responsável por diferentes partes do projeto, mas com um objetivo em comum: a construção de um simulador de mobilidade para ambientes urbanos em que o movimento das entidades móveis, pessoas e veículos, esteja condicionado às próprias características do espaço e que permita a definição de cenários com base em mapas das estradas e a geração automática de tráfego pedestre e automóvel.

As componentes que vão constituir o foco deste trabalho em concreto são a visualização e o *reporting*. A visualização tem como intuito criar uma interface gráfica que apresenta ao utilizador o estado da simulação. Pretende-se que esta componente seja independente da plataforma e possa ser executada a partir de qualquer sistema, desde que este tenha acesso à Internet.

O *reporting* é uma componente que armazena a evolução de toda a simulação para que seja possível visualizá-la e analisá-la posteriormente.

Também será desenvolvida uma funcionalidade para o simulador que permite, que este faça uso de mapas OSM (Open Street Maps).

1.3 Estrutura da dissertação

Esta dissertação está dividida em 7 capítulos principais. O primeiro capítulo aborda a introdução ao tema da dissertação desenvolvida. A introdução possui o enquadramento do tema, apontando as motivações que levaram ao desenvolvimento do simulador e os objetivos pretendidos com o desenvolvimento deste simulador.

No segundo capítulo, intitulado de Simulação de Sistemas Móveis, é abordada a análise feita a simuladores já existentes no enquadramento deste simulador, e modelos de visualização analisados para desenvolvimento desde simulador.

O capítulo Simulação de Mobilidade em Ambiente Urbano, que é o terceiro capítulo desta dissertação, expõe detalhadamente os objetivos pretendidos com este simulador, bem como uma descrição geral da arquitetura do simulador e dos componentes do sistema.

No quarto capítulo, Desenho das Componentes de Visualização e *Reporting*, são apresentadas as decisões tomadas no desenho das componentes de visualização e *Reporting* do simulador, as arquiteturas discutidas, as suas principais características e as decisões tomadas.

A implementação do trabalho desenvolvido no âmbito desta dissertação, isto é, a implementação das componentes *Visualization*, *Reporting* e a integração dos mapas OSM é apresentada no capítulo 5, denominado de Implementação. Neste capítulo são apresentadas as classes desenvolvidas, com a descrição das suas estruturas e funcionamento de cada uma. Para além da estrutura e funcionamento, também é realizada a explicação das principais funções desenvolvidas e algoritmos.

No capítulo Testes e Análise de Resultados, o capítulo 6 da dissertação, são apresentados os resultados dos testes efetuados ao sistema para garantir o funcionamento pretendido. Também é feita a análise e discussão dos resultados obtidos.

No capitulo 7, a Conclusão, estão expostas as conclusões retiradas com o desenvolvimento deste projeto, bem como o possível trabalho futuro. É ainda referido como este projeto contribuiu para a minha aprendizagem e desenvolvimento profissional.

2 Simulação de Sistemas Móveis

Neste capítulo é apresentado o estado da arte, bem como o trabalho já existente nesta área, com mais ênfase na parte de visualização.

Por isso, seguidamente será feito uma introdução sobre os simuladores analisados, seguida do estudo efetuado sobre cada simulador.

2.1 Simuladores Existentes

Os simuladores que se pretende estudar devem ter requisitos semelhantes ao que se pretende implementar no simulador a desenvolver. Desta forma, pode-se explorar o que já existe ao nível dos simuladores e também tentar melhorar o já existente nessa vertente. Um fator decisivo na escolha dos simuladores a estudar seria o facto de serem, tal como o presente simulador a desenvolver, *open source*.

Assim, os simuladores escolhidos foram:

- Mobilidade Urbana – *The ONE* (The ONE 2010);
- Modelos de Mobilidade – *BonnMotion* (Informatik 4:BonnMotion 2010);
- Mobilidade Veicular – *SUMO* (SUMO - Simulation of Urban Mobility 2010);

2.1.1 The ONE

O The ONE (*Opportunistic Network Environment*) é, como o nome indica, um simulador de redes oportunistas. Este simulador está a ser desenvolvido pelos projetos SINDTN (*Security Infrastructure for Delay Tolerant Network*) (SINDTN 2010) e CATDTN (*Connectivity, Applications, and Trials of Delay Tolerant Networking*) e conta com o apoio da *Nokia Research Center* (*Finland*) (Nokia Research Center 2010). O simulador está a ser desenvolvido em *Java* e é um projeto *open source*, estando o seu código disponível para

download no site do projeto[1]. Ao nível da portabilidade, sendo o simulador uma aplicação em *Java*, o The ONE é bastante flexível uma vez que funciona sem problemas nos principais sistemas operativos da atualidade: *Windows, Mac OSX* e *Linux*.

Este simulador é capaz de suportar nós móveis com diferentes modelos de mobilidade, troca de mensagens entre os nós, vários algoritmos de *routing* entre emissores e receptores e possui ainda uma interface gráfica que permite, entre outras coisas, a visualização da movimentação dos nós e da troca de mensagens entre os mesmos em tempo-real (Keränen, Ott e Kärkkäinen 2009).

2.1.1.1 Ambiente gráfico

Quando iniciado, este simulador mostra o seu ambiente gráfico. Este ambiente gráfico ou interface gráfica, é composto por quatro zonas: uma zona que mostra a área simulada, com o mapa das ruas e os nós em movimento, uma zona onde está a lista de nós ativos (*Nodes*), uma zona onde existe um menu que permite definir os tipos de mensagens que os nós podem trocar entre si (*Event log controls*), e por fim uma zona que demonstra as mensagens trocadas entre os nós (*Event log*). Na Figura 1, pode ver-se o ambiente gráfico deste simulador.

Na zona que contém a movimentação dos nós no mapa há ainda outras funcionalidades disponíveis numa barra acima da área de visualização do mapa que permite, por exemplo, captar imagens de um determinado momento, o que é possível através do botão *screen shot*, efetuar pausas na simulação, fazer a simulação correr mais depressa ou mais devagar, ou ainda, a possibilidade de se fazer *zoom* ao mapa apresentado no ecrã.

[1] http://www.netlab.tkk.fi/tutkimus/dtn/theone/, acedido a Outubro de 2010

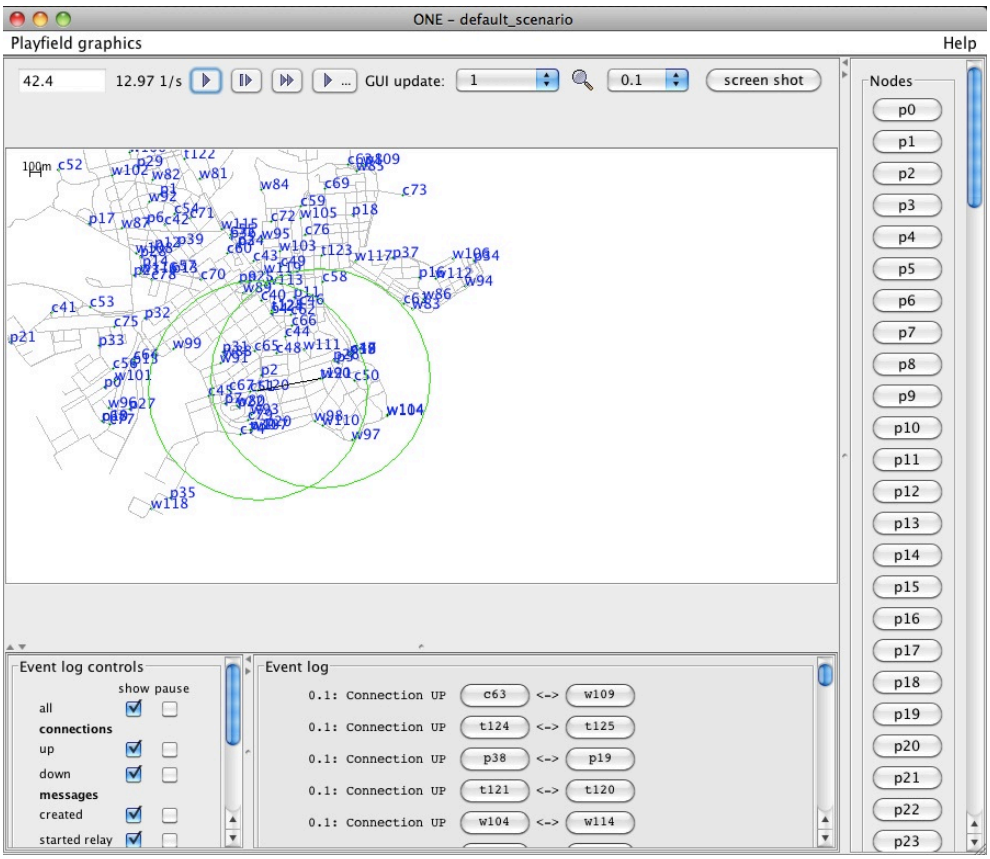


Figura 1 - Ambiente Gráfico do The ONE

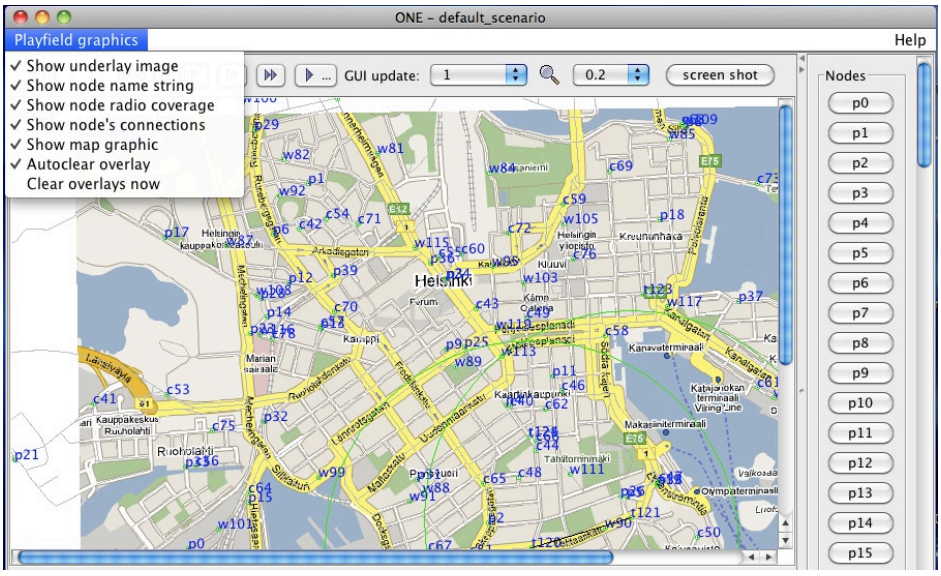


Figura 2 - Ambiente Gráfico do The ONE com mapa como fundo

Como se pode ver na figura 2, no canto superior esquerdo do ecrã aparece um menu com o nome de *Playfield graphics*. Se se abrir esse menu, aparecem várias opções de visua-

lização na janela que contém os nós e os seus caminhos, como é o caso do mapa base que foi usado para construir o percurso apresentado, neste caso, um mapa do centro de Helsínquia.

2.1.1.2 Cenários de simulação

No friso superior da janela do simulador é possível ver-se o nome da simulação que está a decorrer. Tanto na Figura 1 como na Figura 2,o cenário que está a ser simulado tem o nome de *default_scenario*. Trata-se do cenário que é aberto por defeito, quando não é escolhido qualquer tipo de cenário específico, quando se arranca o *software*. Para se iniciar um cenário diferente, basta colocar o nome do cenário que se pretende arrancar após o comando para iniciar o simulador.

Caso se pretenda simular um cenário específico, este tem que ser configurado através de um conjunto de parâmetros descritos num ficheiro de texto, e tem que ser escrito segundo as regras dos ficheiros *properties* do Java.

Se se pretender explorar diferentes tipos de cenários para além dos que o The ONE fornece, é aconselhado consultar o cenário *default_settings.txt* e seguir as instruções que estão lá disponibilizadas sobre cada uma das primitivas usadas e sobre os tipos de primitivas que se podem acrescentar. Outro ficheiro que pode ajudar a entender a utilidade de algumas das linhas dos ficheiros de cenário é o *WDM_conf_help.txt*.

2.1.1.3 Vantagens

O *The ONE* é um simulador que permite simular o movimento de diversos tipos de nós assim como as suas interações. Este simulador tem um ambiente gráfico bastante intuitivo, mostrando ao utilizador, à medida que a simulação decorre, quando é que os nós se conectam e desconectam. Além disso, permite que o utilizador controle que tipo de *logs* pretende ver durante a simulação, sendo que os pode alterar a qualquer momento. Uma outra característica que é muito relevante no *The ONE* é o facto de se poder acelerar ou abrandar a velocidade da simulação. Para que seja visualmente mais atrativo, o *The ONE* dispõe da possibilidade de se colocar uma imagem da cidade que está a simular como fundo no visualizador (Figura 2). Por fim, pode-se ainda referir que este simulador é multiplataforma e permite que sejam carregados diferentes mapas.

2.1.1.4 Limitações

Embora o *The ONE* seja um simulador com bons atributos, possui algumas limitações que estão constantemente a ser melhoradas, visto ser um projeto ainda em desenvolvimento. As principais limitações encontradas foram:

- O número de atores por simulação está limitado a 500;
- A quantidade de atores é sempre a mesma durante toda a simulação;
- Os atores não são dotados de dinamismo ao nível do seu movimento, ou seja, fazem sempre o mesmo circuito com a mesma velocidade;

2.1.2 BonnMotion

O *BonnMotin* é uma aplicação *Java* utilizada para criar e analisar cenários de mobilidade. Desenvolvido pelo grupo de Sistemas de Comunicação da Universidade de Bonn, na Alemanha, serve de ferramenta na investigação de dispositivos móveis. Como software open-source, o BonnMotion encontra-se atualmente na versão 2.0 lançada a 07/11/2011 e pode ser feito o download no site do projeto[2].

Este simulador permite criar vários tipos de cenários de mobilidade que seguem as regras dos modelos de mobilidade a eles associados. Os modelos de mobilidade suportados incluem, entre outros, os seguintes:

- *Randon Waypoint Mobility;*
- *Manhttan Grid Mobility;*
- *Gauss-Markov Mobility (original);*
- *Gauss-Markov Mobility;*
- *Reference Point Group Mobility;*
- *Randon Direction Mobility;*
- *Randon Walk Mobility;*
- *Column Mobility;*
- *Nomadic CommunityMobility;*

O formato nativo em que o *BonnMotion* guarda as trajetórias do movimento dos nós é uma linha num ficheiro, ou seja, cada linha do ficheiro corresponde a todas as movimentações de um determinado nó, o que significa que existe uma linha para cada nó. Esta linha

[2] http://net.cs.uni- bonn.de/start/, acedido a Outubro de 2012.

contém todos os pontos por onde o nó passou. Os cenários criados por este simulador podem ser exportados para outros simuladores de redes para que sejam usados por esses mesmos simuladores, nomeadamente pelos seguintes:

- NS-2 (The Network Simulator - ns-2 2010);
- GloMoSim/QualNet (GloMoSim 2010);
- COOJA (An Introduction to Cooja 2010);
- MiXiM (MiXiM 2010);
- The ONE (The ONE 2010);

Além deste formato, o *BonnMotion* permite ainda que se guarde a informação em XML.

Esta aplicação não possui qualquer tipo de ambiente gráfico para acompanhar ou analisar a simulação. No entanto, existe a possibilidade de se exportar um ficheiro que permite visualizar a movimentação de um dos nós usando o *gnuplot*(figura 3).

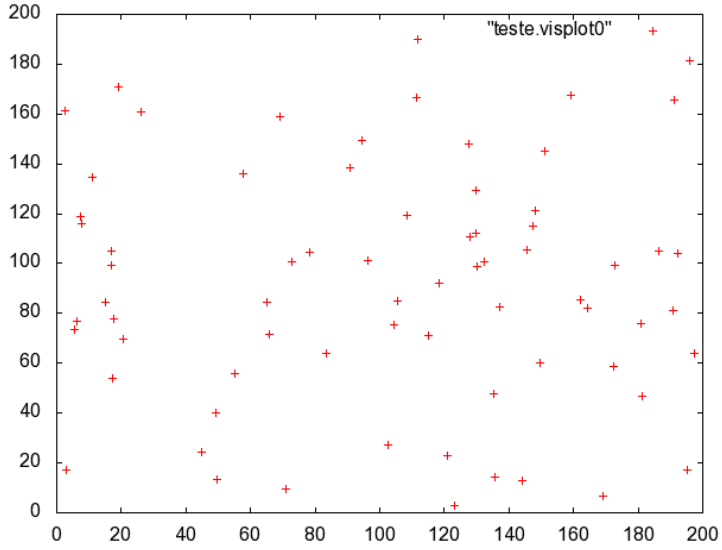


Figura 3 - Exemplo de um cenario de teste visto no *gnuplot*

Sendo um *software opensource*, o *BonnMotion* encontra-se disponível para *download* no *site* do projeto[3].

A análise deste simulador foi feita numa fase inicial do desenvolvimento do simulador em conjunto com os outros elementos grupo, tendo um valor para o objetivo global fi-

[3] http://www.gnuplot.info/, acedido a Outubro 2012.

nal, mas no âmbito desta dissertação e seus objetivos, possui uma grande limitação sendo que não possui componente gráfica.

2.1.3 SUMO

O SUMO (*Simulation of Urban Mobility*) é uma aplicação *open source*[4], de simulação de tráfego multi-modal, desenvolvido em linguagem C++. Este projeto, iniciado em 2000, conta com um vasto grupo de participantes composto maioritariamente por universidades alemãs:

- Zaik – University of Cologne (Universität zu Köln 2010);
- DLR (Deutsches fur Luft- und Raumfahrt 2010);
- University of Lubeck (Universität zu Lubeck 2010);
- Humboldt University of Berlin (Humboldt Universität zu Berlin 2010);
- University of Innsbruck (Universität Innsbruck 2010);
- Technical University of Munich (Technisch Universität München 2010);
- Indian Institute of Technology Bombay (Indian Institute of Technology Bombay 2010);
- Polytechnic of Torino (Politecnico di Torino 2010);
- University of Wroclaw (Uniwersytet Wrocławski 2010).

Esta aplicação tem como objetivo simular trânsito urbano, dando a possibilidade de criar mapas e viaturas de vários tipos, sendo que também se pode controlar o fluxo de trânsito a gerar.

Para se criar uma simulação com esta aplicação é necessário criar quatro ficheiros XML, um para definir pontos de estrada (extensão *.nod.xml*), outro para definir os sentidos de circulação (extensão *.edg.xml*), um outro que define os percursos pretendidos, assim como as viaturas (extensão *.rou.xml*) e por fim um ficheiro gerado pelo comando "NETCONVERT" (extensão *.net.xml*). Este último, importa os ficheiros onde estão as estradas e os sentidos de circulação e retorna um outro ficheiro onde se pode controlar os limites de velocidade e o tipo de veículos que irão circular durante a simulação.

[4] http://sumo.sourceforge.net/, acedido em Outubro 2010.

Para efetivamente correr um exemplo de simulação, é necessário criar um outro ficheiro, com a extensão *.cfg*, onde indicamos os ficheiros *.net* e *.rou* para que o SUMO tenha indicação de o que executar. É também neste ficheiro que se define a duração da simulação.

Após a criação dos ficheiros é possível executar o programa carregando o ficheiro criado, de forma a dar início à simulação. Quando é iniciada essa simulação, surge um ecrã semelhante ao que se pode ver na figura 4.

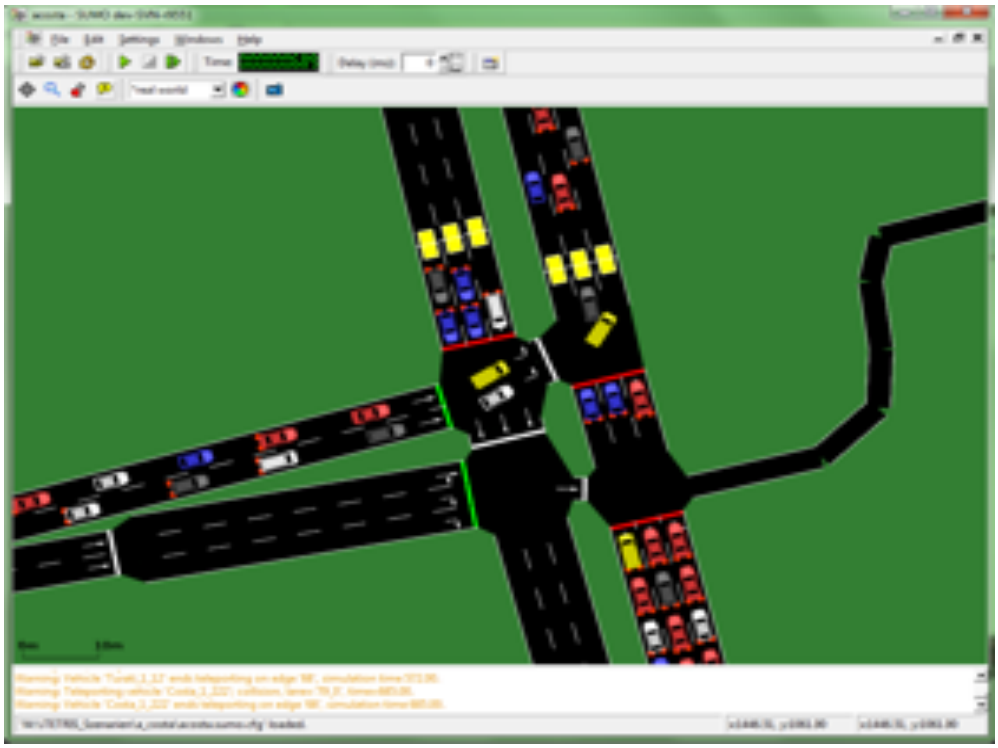


Figura 4 - Ambiente Gráfico do SUMO

Esta interface tem a particularidade de possibilitar várias janelas de observação da simulação, sendo estas no máximo quatro. A possibilidade de ter várias janelas é pertinente caso o cenário de simulação seja extenso, sendo assim possível focar vários pontos diferentes de simulação.

Para além disso, esta interface possui botões de play, stop e pause, possui um regulador de velocidade de simulação, um relógio com o tempo de simulação e também oferece a

possibilidade de fazer *zoom* e tirar uma fotografia da simulação clicando no ícone em formato de máquina fotográfica.

Após a análise dos simuladores explicitada acima, foi possível detetar importantes características nas interfaces de utilizador que será importante sintetizar. Em ambos os simuladores, existe a possibilidade de obter uma foto da simulação quando o utilizador o entender, fazer zoom e regular a simulação através de botões (play, stop, pause) e aumentar a velocidade de simulação.

No The ONE é possível ver a lista de nós ativos (Nodes), para além disso, também é possível ver e definir o tipo de mensagens que os nós podem trocar entre si.

Na interface do SUMO é possível ter várias janelas de observação da simulação, sendo esta a característica mais importante desta interface.

3 Simulador de Movimento em Ambiente Urbano

Após o estudo dos simuladores já existentes, neste capítulo serão expostas as decisões tomadas relativas ao simulador desenvolvido. Por isso, este capítulo possui os objetivos pretendidos com este simulador, a análise dos requisitos, bem como uma descrição do simulador a nível da arquitetura geral e dos componentes do sistema.

3.1 Objetivos

O desafio proposto com este trabalho foi a elaboração de um simulador capaz de, como o próprio nome indica, simular a mobilidade em ambiente urbanos. Os objetivos associados ao desenvolvimento deste simulador são:

- Simular mobilidade em ambientes urbanos em larga escala (grande áreas, muitos nós, etc.);
- Simular redes móveis inseridas nesses mesmos ambientes.
- Simular o comportamento dos atores de forma a tornar mais realista a simulação.

Estes objetivos têm como grande propósito fazer deste simulador melhor que os simuladores já existentes.

3.2 Análise dos requisitos

Dados os objetivos, irão ser agora abordados os requisitos necessários para elaborar a arquitetura do simulador. Para uma abordagem mais clara e objetiva, foi definida uma lista de princípios fundamentais para servirem de orientação na elaboração da sua arquitetura, sendo eles:

- Cada entidade será um processo autónomo;
- A mobilidade dos atores será baseada em modelos comportamentais (andarão sobre estradas, considerarão o comportamento dos outros atores no seu movimento futuro, etc.);
- O espaço de simulação será baseado em mapas;
- As entidades usufruirão de um espaço de memória partilhado para facilitar as interações entre estas;
- Distribuir esta solução por várias máquinas sincronizando a memória partilhada ;
- Distribuir a carga pelas diversas máquinas, tendo para isso de existir um coordenador;

3.3 Arquitetura do sistema

Após a análise dos requisitos pretendidos para o desenvolvimento do *BartUM* (nome atribuído pelo grupo ao simulador), seguiu-se a elaboração da arquitetura do sistema para suportar os princípios fundamentais pretendidos.

No desenvolvimento da arquitetura foram identificadas três entidades nucleares para o funcionamento do sistema, são elas o *GlobalCoordinator*, o *LocalCoordinator* e a *Visualization* (figura 5). Estas entidades possuem um funcionamento autónomo, embora necessitem de estar em permanente contacto.

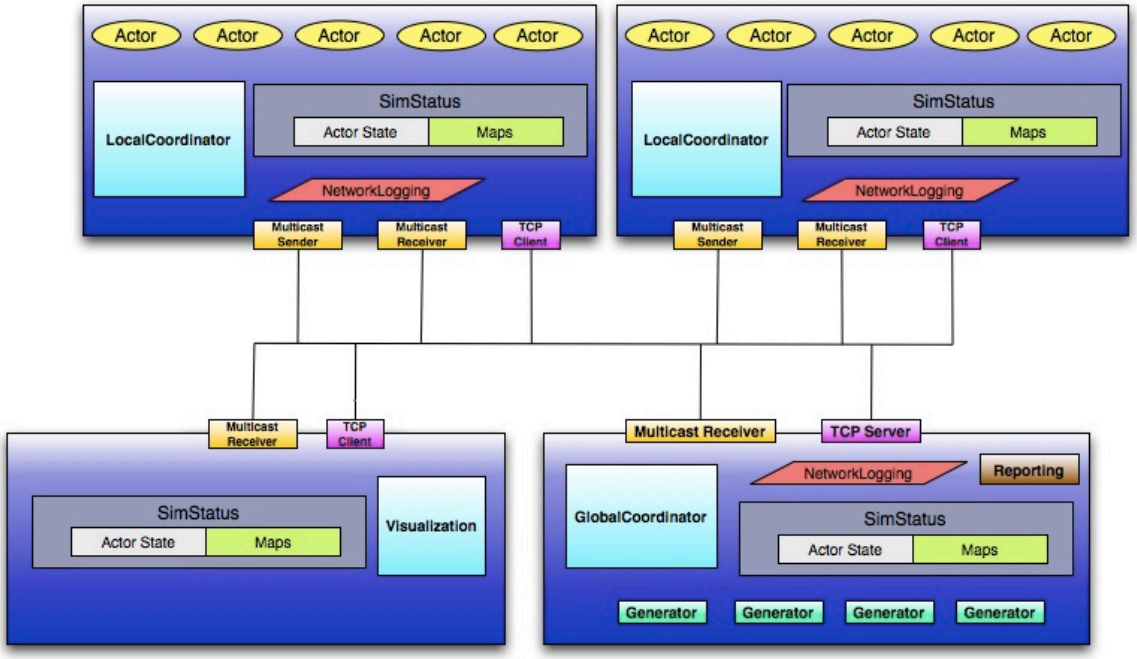


Figura 5 - Arquitetura Global do Sistema

A entidade central do sistema é o *GlobalCoordinator*. Esta entidade tem como função coordenar o desenrolar da simulação. Dado que esta é a entidade central, só existirá um *GlobalCoordinator* por simulação, e sem a existência desta entidade, a simulação não pode ser realizada. O *GlobalCoodinator* é composto por outras seis entidades: *Reporting*, *NetworkLogging, SimStatus*, *TCPServer*, *MulticastReceiver*, e *Generator*, sendo que existem várias entidades diferentes do *Generator* no *GlobalCoordinator*.

O *LocalCoordinator* tem como função coordenar e monitorizar os atores que estão associados a si. Esta entidade é um coordenador local, por isso, numa simulação existe a possibilidade de existirem vários *LocalCoordinator*s. Esta entidade será composta por outras cinco entidades: *NetworkLogging*, *TCPClient*, *MulticastReceiver*, *MulticastSender,* e os *Actors*, sendo que em cada *LocalCoordinator* existirão vários *Actors* de vários tipos diferentes.

Por último, a *Visualization*. Esta entidade não é necessária para o funcionamento da simulação, podendo ser iniciada a qualquer momento da simulação ou num momento onde não esteja a decorrer nenhuma simulação. A *Visualization* funciona com base na informação disponibilizada pelos *LocalCoordinator*s à rede ou através de informação carregada através

dos ficheiros de *reporting*. Esta entidade é responsável pela apresentação gráfica do decorrer da simulação e também é possível rever as simulações através desta entidade.

3.4 Componentes do sistema

Após traçar os objetivos, analisar os requisitos e definir a arquitetura, concluiu-se que eram necessários doze entidades para alcançar dos objetivos propostos. Essas entidades que se relacionam entre si, são as seguintes: *GlobalCoordinator*, *LocalCoordinator*, *SimStatus, Reporting, Visualization*, *Generator*, *NetworkLogging*, *TCPClient*, *MulticastReceiver*, *MulticastSender*, *TCPServer* e *Actor*.

3.4.1 GlobalCoordinator

O *GlobalCoordinator* terá o papel de coordenador no desenrolar da simulação. Esta entidade é quem decidirá quando criar um novo ator, que tipo de ator é criado e a que *LocalCoordinator* ficará associado, de forma a que a carga fique distribuída por todas as máquinas. Para além disso, esta entidade é notificada com todas as atualizações a circular na rede.

3.4.2 LocalCoordinator

Como foi referido anteriormente, o sistema a desenvolver irá ser distribuído, isso implica a necessidade de criar entidades que mantenham o sistema em constante contacto entre todos os computadores envolvidos, isto é, que mantenha a memória partilhada sincronizada. Para esse efeito, surgiu o conceito de *LocalCoordinator*. O *LocalCoordinator* será um coordenador do estado local, que periodicamente terá de enviar o estado dos seus atores para que a memória partilhada esteja sempre atual.

3.4.3 SimStatus

Esta entidade guardará toda informação relativa à simulação, ou seja, esta entidade foi projetada para ser genérica e utilizada por todas as entidades do simulador, para um acesso mais rápido e eficiente de todas entidades. Esta guarda a informação relativa aos mapas e ao estado atual da simulação.

3.4.4 Reporting

A entidade *Reporting* vai trabalhar em conjunto com o *GlobalCoordinator*, de forma a utilizar a informação disponível no *GlobalCoordinator* para criar uma gravação da simulação que pode servir para análise da simulação ou simplesmente para rever a simulação.

3.4.5 Visualization

A entidade *Visualization* é responsável por criar uma apresentação gráfica da forma como a simulação está a decorrer, ou seja, representa o movimento que os atores estão a ter.

Para que seja uma representação mais percetível, a *Visualization* necessitará de ter todos os mapas carregados para que os possa representar. Além disso, terá que ser constantemente informada das movimentações dos atores para que consiga representar o seu movimento.

Esta entidade não é obrigatória, isto é, não tem que ser criada no início da simulação, assim como não precisa de estar sempre ativa, podendo ser ativada apenas quando se achar oportuno monitorizar o estado de uma simulação, ou até mesmo nunca ser ligada.

A *Visualization* será responsável por disponibilizar ao utilizador uma representação gráfica da simulação. Para além disso, a entidade *Visualization* também pode ser usada para ver simulações anteriores através das gravações feitas pelo *Reporting*.

3.4.6 Actors

Os *Actors* serão as entidades mais importantes e mais elaboradas do sistema a desenvolver, pois eles terão que estar em constante movimento. Esta entidade poderá ser independente, fixa ou móvel, sendo que é um interveniente no curso da simulação.

Os atores podem representar um *smartphone* na posse de uma pessoa, um veículo, um semáforo, ou seja, um qualquer dispositivo que seja passível de se conectar a outro. Assim, o movimento não será feito de forma totalmente aleatória, sendo um dos objetivos que os atores sejam o mais próximos possível de um ator real. Logo, eles terão de ser suficientemente desenvolvidos ao ponto de levarem em consideração que existe alguém à sua frente, que uma determinada estrada tem apenas um sentido, que existem cruzamentos e semáforos, que há pontos de encontro onde eles poderão parar, etc.

Cada *Actor* que for criado tem de ficar associado a um dos *LocalCoordinator* existentes e terá que lhe comunicar qualquer tipo de alteração que surja no seu estado, para que

a informação guardada no *SimStatus* nunca fique obsoleta. De forma a que os vários nós se possam movimentar ao mesmo tempo sem que haja problemas, cada um correrá numa *thread* diferente. Assim, será possível ter vários *Actors* a movimentarem-se ao mesmo tempo.

Nesta fase de implementação, foram pensados 6 tipos diferentes de atores: *tram*, *pedestrian*, *car*, *train*, *cycle* e *bus*.

3.4.7 Generators

As entidades *Generators* serão responsáveis pela criação dos novos *Actors*. Cada *Generator* será especializado em apenas um tipo de *Actor* que serão colocados num determinado ponto fixo do mapa para começarem o seu movimento a partir daí. A geração de *Actors* não será feita com uma cadência igual para todos os tipos de nós, pois é normal que num ambiente urbano exista maior oscilação do número de pedestres do que em relação ao número de comboios ou autocarros, ou seja, serão usados diferentes modelos para cada tipo de ator. Esse intervalo de geração de novos atores irá ocorrer segundo um determinado padrão temporal e probabilístico.

3.4.8 TCPServer

Esta entidade serve para o *GlobalCoordinator* receber conexões TCP dos *LocalCoordinators* e do *Visualization*. Essas conexões vão servir para *GlobalCoordinator* disponibilizar a informação necessária para o funcionamento das outras entidades. No caso do *Visualization,* são enviados os dados relativos ao mapa da simulação, em relação aos *LocalCoordinatoors* e para além da informação relativa ao mapa também são enviadas notificações para criar novos atores.

3.4.9 TCPClient

O *TCPClient* serve para as entidades *LocalCoordinator* e *Visualization* se conectarem ao *GlobalCoordinator* e receberem a informação necessária para o seu funcionamento.

3.4.10 MulticastReceiver

A entidade *MulticastReceiver* é responsável por fazer com que a entidade que está a utilizá-la se junte ao grupo *multicast*, e capte os pacotes *multicast* que são enviados para este endereço.

3.4.11 MulticastSender

Esta entidade vai permitir que as entidades que a utilizem possam enviar mensagens sobre o estado de cada actor para o endereço *multicast*.

3.4.12 NetworkLogging

A entidade *NetworkLogging* será a responsável por fazer o relatório de tudo o que se vai passar na rede LAN. Para que seja possível fazer uma correta análise e diagnósticos sobre o que se passará na rede durante a simulação, esta classe terá que estar presente tanto no *GlobalCoordinator* como no *LocalCoordinator*. Esta classe terá 3 tipos de funcionamento que deverão ser definidos no início de cada simulação:

Tipo 1 - Só serão guardados em ficheiro informações do tipo *warning* e *config,* que serão alertas de erros nas interfaces e registos de configuração da rede, respetivamente;

Tipo 2 - Serão guardadas todas as informações da rede sendo registadas como *info*, *warning* e *config* que corresponderão, respetivamente, aos pacotes que são trocados na rede, aos alertas de erros nas interfaces e aos registos de configuração da rede;

Tipo 3 - Não será guardada qualquer informação da rede em ficheiro.

4 Desenho das Componentes de Visualização e Reporting

Neste capítulo são apresentadas as decisões tomadas nas componentes de Visualização e *Reporting* do simulador. Para esse fim, são apresentadas todas as arquiteturas discutidas, com os seus prós e contras, e as decisões tomadas.

4.1 API Visualização

Para definir a forma de desenvolver a aplicação *Visualization*, investigou-se quais seriam as melhores formas de criar uma representação gráfica em *Java*. Após uma análise das *APIs* existentes, foram encontradas as classes *Graphics* e *Applet*. Estas classes são próprias para o desenvolvimento de interfaces gráficas, existindo apenas algumas diferenças entre elas. Seguidamente, será efetuada uma apresentação destas classes, avaliando os seus pontos fortes e pontos fracos, tendo em conta os requisitos do simulador.

4.1.1 Objeto Graphics

A classe *Graphics* é uma classe abstrata usada para implementar interfaces gráficas. Esta classe permite desenhar figuras geométricas, selecionar as cores das figuras, posicioná-las segundo um sistema de coordenadas e inserir imagens na janela de visualização.

A quantidade de coordenadas disponíveis com esta classe é equivalente ao número de pixéis disponíveis na janela de desenho. Ou seja, caso seja criada uma janela para desenho de 800x500, existirá 400000 coordenadas diferentes. O eixo de coordenadas disponível nesta classe é diferente do eixo de coordenadas matemático, porque não possui valores negativos, sendo que o eixo das abcissas cresce de forma igual, mas o eixo das ordenadas cresce de forma contrária ao eixo de coordenadas tradicional.

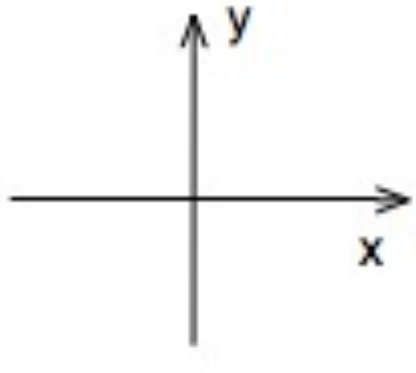


Figura 6 - Eixo de coordenadas matemático

Figura 7 - Eixo de coordenadas da classe *Graphics*

Uma classe que derive da classe *Graphics*, necessita reescrever o método *paint*(*graphics* g). O método *paint* não é chamado diretamente pelo programador, é um método chamado automaticamente quando o computador exibe a componente sobre o ecrã. Também é invocado se acontecer um evento que exige que a componente seja redesenhada. Caso seja desejado invocar este método para voltar a imprimir a imagem, existem os métodos *repaint*() e *update*(*graphics* g). O método *repaint* invoca o método *paint*, imprimindo o objeto gráfico já existente. Por outro lado, o método *update* recebe como argumento o objeto gráfico e chama o método *paint*, passando a este o objeto gráfico.

Através da utilização da classe *Graphic* é possível desenhar várias figuras geométricas com a utilização dos seguintes métodos:

- *draw3DRect(int x, int y, int width, int height, Boolean raised)* - Desenha o contorno do retângulo em 3-D.

- *drawLine(int x1, int y1, int x2, int y2)* - Desenha uma linha.
- *drawOval(int x, int y, int width, int height)* - Desenha o contorno de uma figura circular.
- *drawPolygon(int[] xPoints, int[] yPoints, int nPoints)* - Desenha o contorno de um polígono.
- *drawRect(int x, int y, int width, int height)* - Desenha o contorno de um retângulo.
- *drawString(String str, int x, int y)* - Desenha uma string.
- *fill3DRect(int x, int y, int width, int height, Boolean raised)* - Desenha um retângulo em 3-D inteiramente pintado.
- *fillOval(int x, int y, int width, int height)* - Desenha uma figura circular inteiramente pintada.
- *fillPolygon(int[] xPoints, int[] yPoints, int nPoints)* - Desenha um polígono inteiramente pintado.
- *fillRect(int x, int y, int width, int height)* - Desenha um retângulo inteiramente pintado.

Após análise desta classe e tendo em conta que este simulador não procura uma visualização com grandes grafismos, considerou-se que este objeto tem as características necessárias para ser a API deste simulador.

4.1.2 Objeto Applet

A classe *applet* também é uma classe abstrata destinada a fins gráficos, usando os métodos do objeto *graphic* para construir as suas representações gráficas. Por isso, consegue reproduzir as mesmas figuras que o objeto *graphic* usando os mesmos métodos.

Mas as aplicações *applets* possuem características fora do comum, características essas que podem ser úteis para este simulador. Os *applets* são programas em Java que são executados dentro de um *browser*. Os *applets* são considerados dinâmicos, independentes da plataforma e funcionam em ambiente de rede. As aplicações *applet* são ótimas para criar interfaces e representações gráficas. Porém, os *applets* apresentam algumas limitações, nomeadamente, o facto de não serem capazes de ler ou escrever ficheiros no computador local, existem questões de compatibilidade entre diferentes sistemas de ficheiros e existem

limitações em comunicar com servidores. Um *applet* possui 4 métodos que estão sempre presentes na sua constituição, sendo estes:

- *init()* - executado quando a *applet* é carregada pela primeira vez;
- *start()* - executado quando o navegador carrega ou volta à página com o *applet*.
- *stop()* - executado quando o navegador deixa a página com o *applet*.
- *destroy()* - executado quando o navegador é fechado.

Após análise do que o *applet* tem para oferecer, é possível verificar que possui características adequadas para ser o modelo de visualização deste simulador.

4.2 Visualização

Após algum tempo de estudo e pesquisa foram surgindo ideias sobre arquiteturas possíveis para a visualização, sendo possível delinear três arquiteturas diferentes. De seguida, será exposta uma proposta de arquitetura, onde a componente de visualização é integrada numa página Web através de um *Java Applet*. Com esta abordagem, ficamos com uma visualização disponibilizada por uma página *Web*, o que permite que a visualização seja executada no *browser* de um computador que esteja conectado na rede local.

A aplicação *Applet* obterá os dados relativos aos mapas através de uma ligação TCP com o *GlobalCoordinator*, pelo que terá de possuir uma conexão TCP cliente. Para além disso, também vai possuir *multicastReceiver* que estará à escuta no endereço *multicast* para captar os pacotes enviados pelos *LocalCoordinators*. Nesses pacotes a informação que é possível obter é a posição atual do atores. Este tipo de pacote só é gerado quando o ator alterar a sua posição, também sendo possível encontrar-se nesses pacotes a informação relativa à carga dos *LocalCoordinators*. Com toda esta informação a aplicação *applet* cria a reprodução visual da simulação.

Essa arquitetura tem ao seu favor o facto de dispensar instalação de aplicações adicionais, bastando possuir um dispositivo, podendo ser qualquer computador, *tablet* ou *smartphone,* que se possa conectar à rede local, possuindo um *browser*. No entanto, esta solução não funciona fora da rede local.

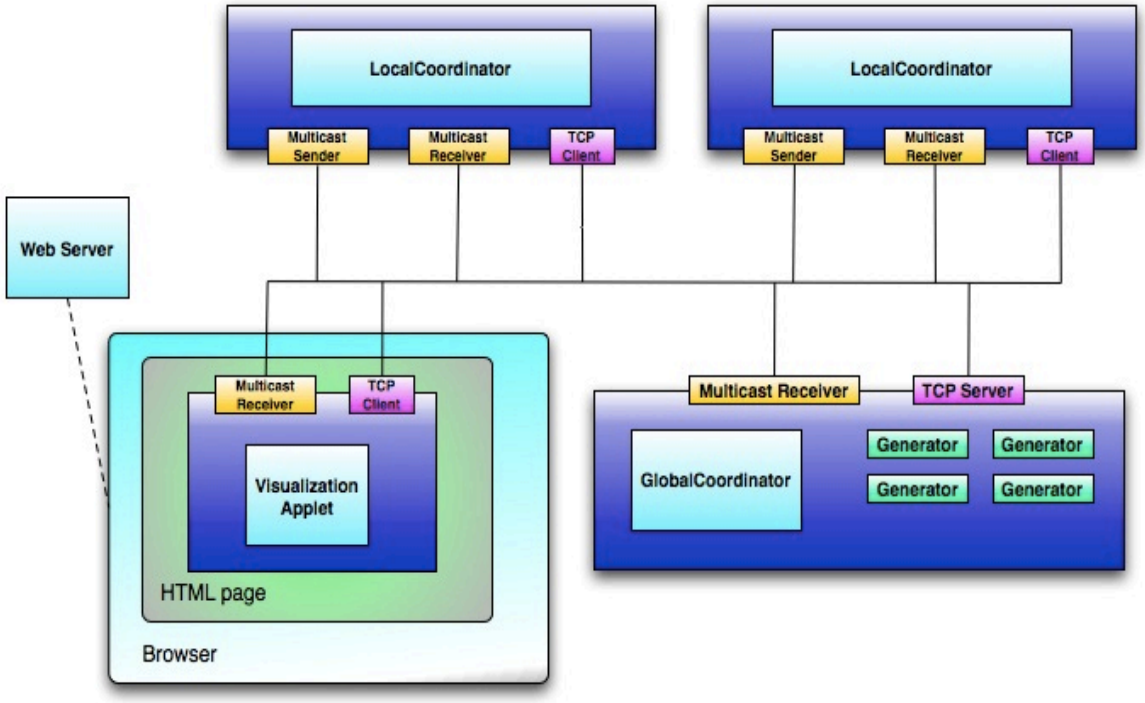


Figura 8 - Arquitetura do Sistema com a *Visualization* em *Java applet*

Na segunda arquitetura planeada, a visualização é uma aplicação *Java grafic* que será executada numa máquina ligada à rede local.

Com esta arquitetura a aplicação *Visualization* irá necessitar uma conexão TCP cliente e um *multicastReceiver* para troca de informações com as outras entidades. Assim, o *Visualization* conecta-se à aplicação *GlobalCoordinator*, por via de uma conexão TCP cliente, através da qual vai receber os mapas. Através do *multicastReceiver,* que estará à escuta no endereço *multicast,* os pacotes das alterações de posições dos atores enviados pelos *LocalCoordinators* serão captados. Com a recepção destes dados, a aplicação *Visualization* vai projetar o estado gráfico da simulação em tempo real. Com este desenho, é garantida uma troca de dados eficiente e segura, mas, por outro lado, esta arquitetura faz com que a utilização da aplicação *Visualization* esteja limitada à rede local, e que seja necessário instalar uma aplicação no computador a usar para efeitos de monitorização da simulação.

Esta foi a arquitetura escolhida para a componente *Visualization*. Apesar de as duas arquiteturas apresentarem características semelhantes, pois ambas apenas funcionam em

rede local, considerou-se que a última seria mais simples de aplicar já que não seria necessário criar a página HTML nem seria necessário o servidor Web.

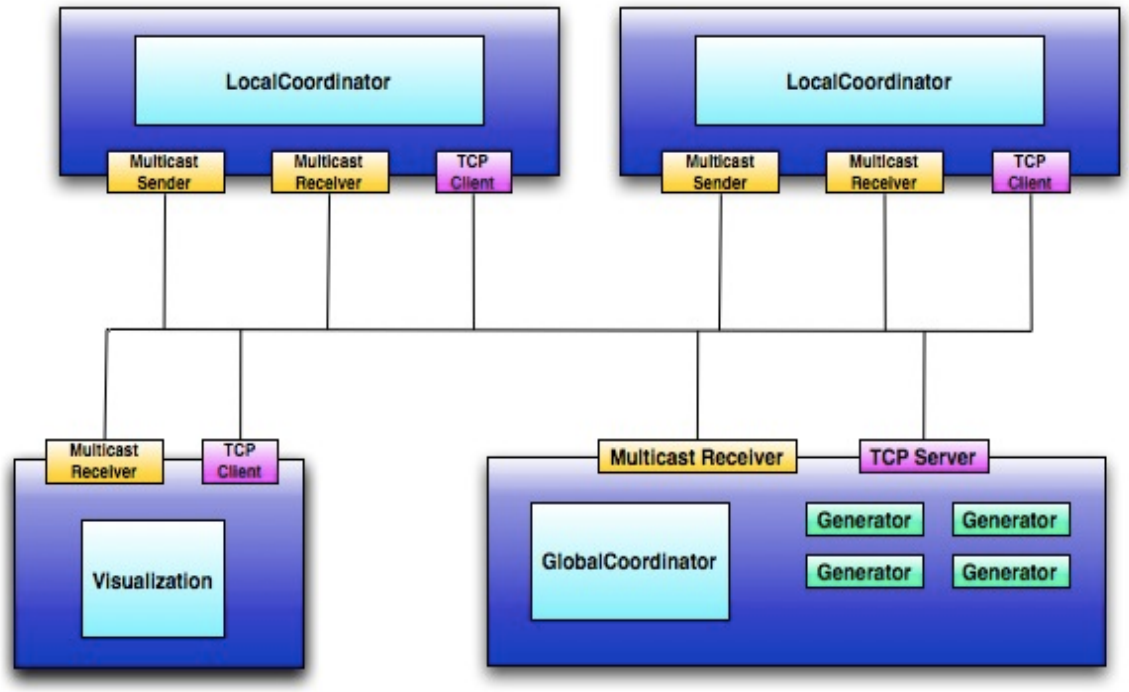


Figura 9 - Arquitetura do Sistema com a *Visualization* em *Java Graphic*

A próxima abordagem, surgiu com o intuito de conseguir ultrapassar a barreira da rede local, ou seja, conseguir que a *Visualization* funcionasse fora da rede local. Esta limitação deve-se ao facto dos pacotes *Multicast* dificilmente serem entregues fora da rede local. Para esse fim, foi pensado num *Gateway* dentro da rede local, que tem como propósito encaminhar os dados para a aplicação *Visualization*. Assim, o *Gateway* serviria de ponte de comunicação entre o *Visualization* e os *Coordinators* (*GlobalCoordinator* e *LocalCoordinators*).

Para isso, o *Gateway* seria uma aplicação sediada em qualquer computador conectado à rede local. Assim, o *Gateway* teria um cliente TCP, para a obtenção dos mapas vindos do *GlobalCoordinator*. Possuiria também um *multicastReceiver,* para captar os pacotes *multicast*. Toda a informação que o *Gateway* recebesse seria encaminhada por TCP para a

aplicação *Visualization*. Para isso, o *Gateway* também teria de possuir um servidor TCP para receber a conexão do *Visualization* e lhe enviar os dados.

Quanto à aplicação *Visualization,* possuiria um cliente TCP onde receberia todos os dados necessários para projetar o estado gráfico da simulação em tempo real.

Esta abordagem tem como grande vantagem o facto da aplicação poder funcionar fora da rede local.

Esta arquitetura não chegou a ser implementada devido ao curto espaço de tempo, mas pode ser uma opção válida para trabalho futuro, evoluindo assim a aplicação *Visualization* para fora da rede local.

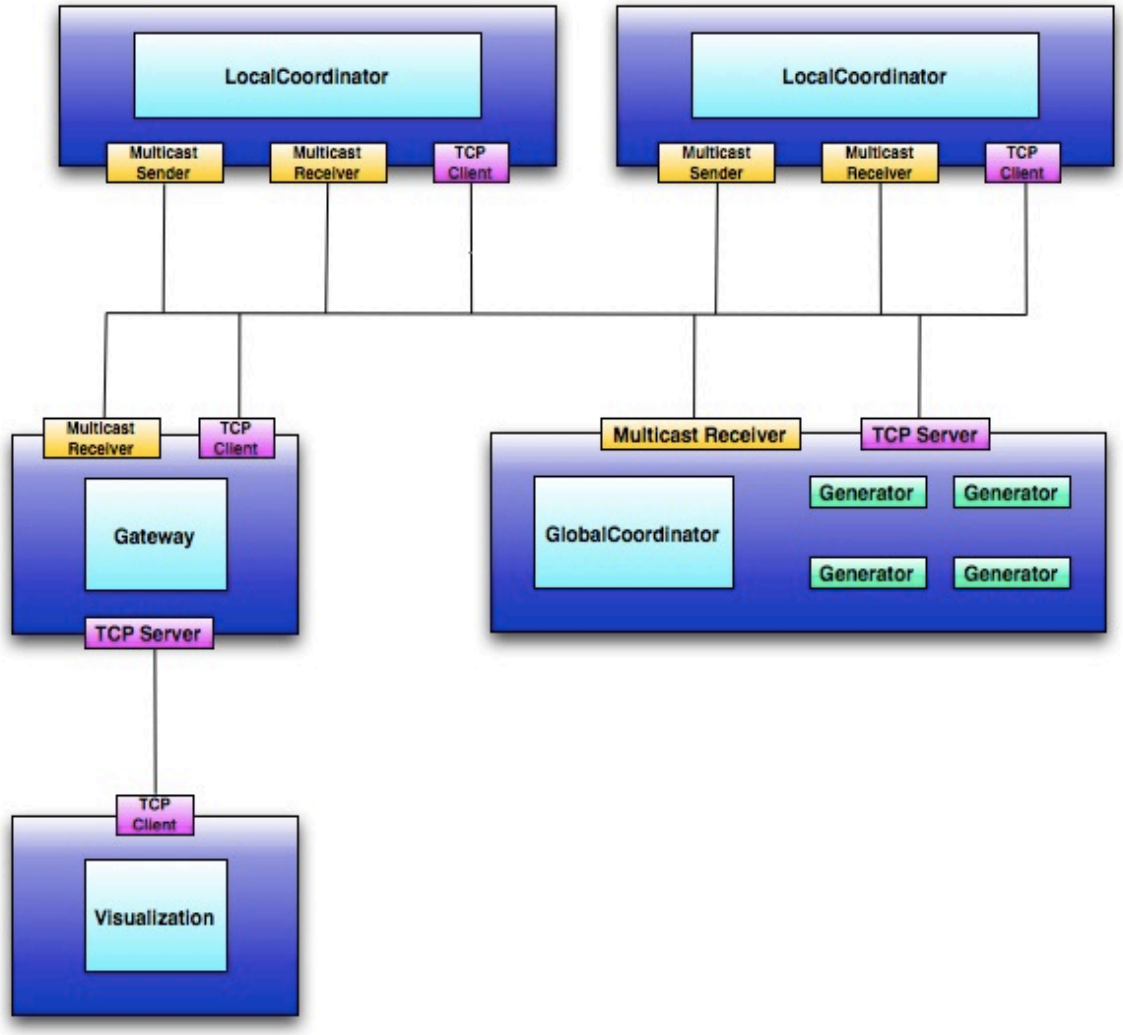


Figura 10 - Arquitetura do Sistema com *Gateway*

4.3 Reporting

A componente *reporting* tem como função criar uma gravação da simulação para quando o utilizador pretender ver ou rever a simulação. Para criar a gravação é necessário registar todas as movimentações dos atores, guardar os mapas onde decorre a simulação, e guardar a carga dos *LocalCoordinators* que participaram na simulação. O serviço de *reporting* é opcional, isto é, o utilizador pode incluir ou excluir o *reporting* da simulação. Para este fim, existe um campo no ficheiro de configuração do *GlobalCoordinator* que determina se a simulação vai possuir *reporting* ou não. Os registos vão ser guardados em três ficheiros de texto, um para os dados relativos ao mapa, outro para os dados das cargas dos *LocalCoordinators* e um outro ficheiro com os dados relativos às movimentações dos atores. Para o desenvolvimento desta componente foram equacionados dois cenários de funcionamento. Num dos cenários, esta componente pode funcionar juntamente com a aplicação do *GlobalCoordinator*, e no outro cenário pode funcionar de uma forma independente, numa máquina à parte.

Inicialmente será exposto o cenário em que o *reporting* funcionará numa máquina independente, ou seja, neste caso é uma aplicação independente.

Assim, neste caso, esta componente possuirá um *multicastReceiver* para receber os movimentos dos atores e a carga de cada *LocalCoordinator*, bem como um cliente TCP para comunicar com o *GlobalCoordinator* e obter os mapas relativos à simulação. Esses dados serão guardados nos ficheiros de *reporting*.

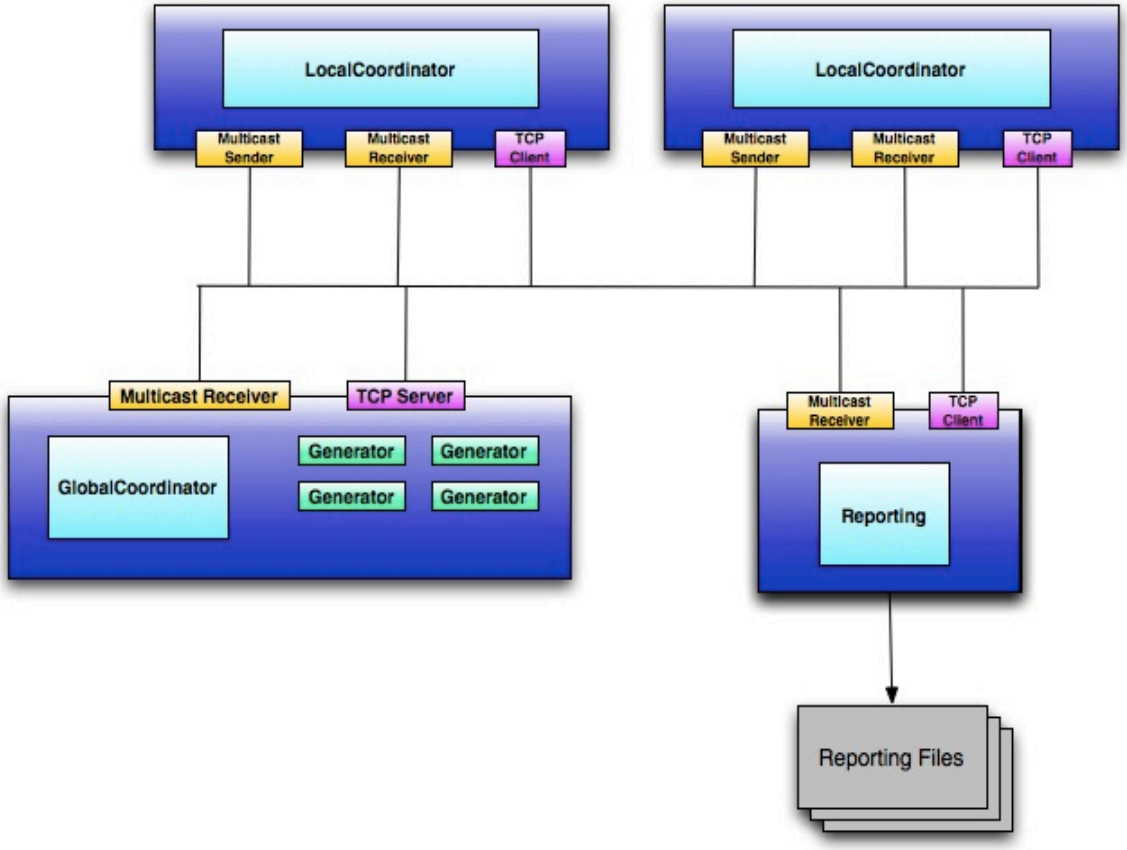


Figura 11 - Arquitetura com reporting numa maquina separada

Seguidamente será apresentado um cenário onde o *reporting* é anexado à aplicação *GlobalCoordinator*.

Desta forma, o *reporting* deixa de ser uma aplicação e passa a ser um processo do *GlobalCoordinator*, assim não será necessário implementar nem o *multicastReceiver*, nem o cliente TCP, pois a informação que esta componente necessita será obtida através do acesso direto à memória do *GlobalCoordinator*.

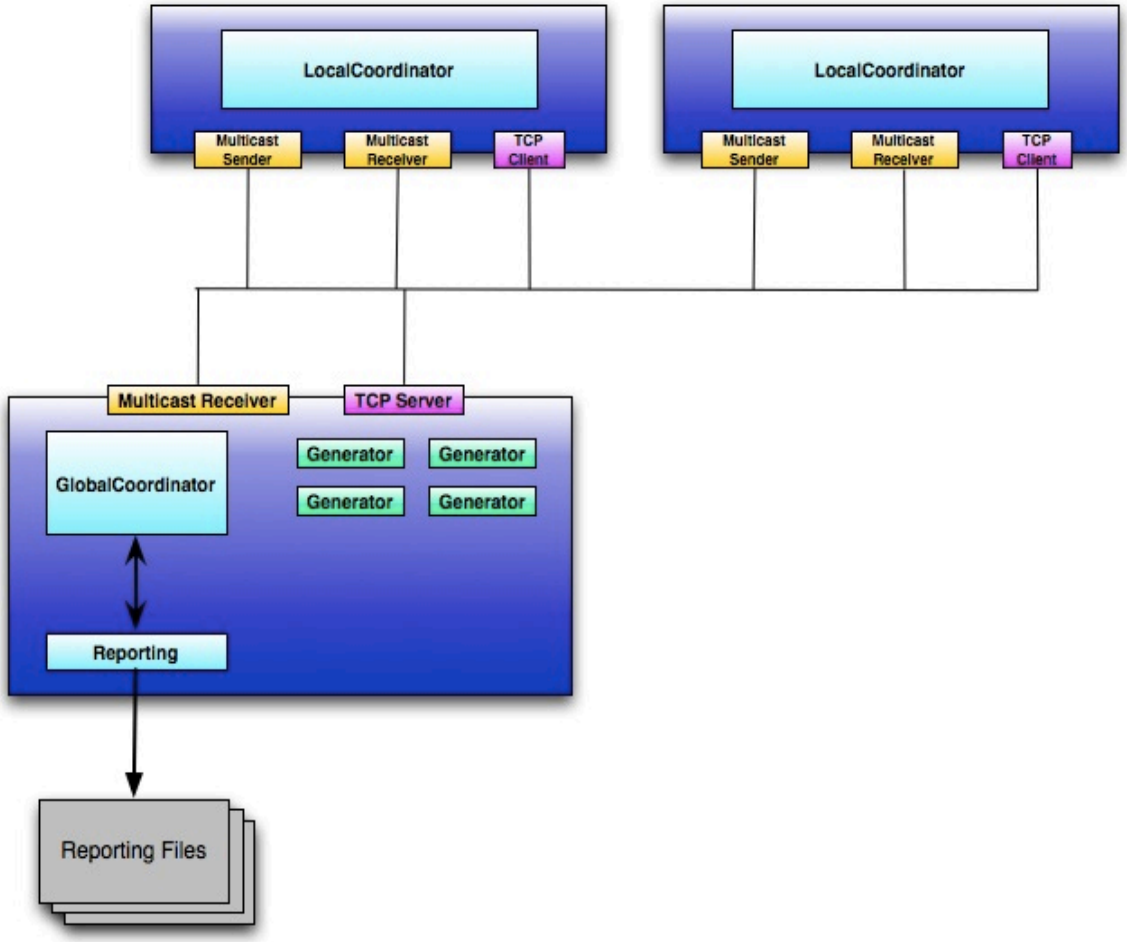


Figura 12 - Arquitetura do *reporting* como processo do *GlobalCoordinator*

Após comparar as duas arquiteturas, foi possível observar que a arquitetura onde o *reporting* é um processo do *GlobalCoordinator* possui mais vantagens que a outra arquitetura apresentada. O facto de não necessitar de comunicações, torna a sua implementação mais simples, e como é um processo do *GlobalCoordinator,* traz a garantia de que sempre que existir uma simulação, esta será completamente guardada, já que o *GlobalCoordinator* é o coração do simulador e sem a sua presença não existe simulação. Dados estes argumentos, a arquitetura onde o *reporting* é um processo do *GlobalCoordinator* é a escolha mais indicada para arquitetura do *reporting*.

De seguida estão expostos o nome e a estrutura dos ficheiros que o *reporting* vai criar para guardar todos os dados necessários para rever a simulação. Estes dados também podem ser usados posteriormente para análise estatística da simulação.

O nome dos ficheiros do *reporting* é construído do identificador do tipo de ficheiro, que é o início do nome, seguido da data da simulação. Os nomes dos ficheiros apresentam a seguinte forma:

- Map_ano-mês-dia hora/minutos/segundos/milisegundos .txt – estrutura para o nome do ficheiro onde são guardados os dados dos mapas.
- Mov_ano-mês-dia hora/minutos/segundos/milisegundos .txt - estrutura para o nome do ficheiro onde são guardados os dados das movimentações dos atores.
- Load_ano-mês-dia hora/minutos/segundos/milisegundos .txt - estrutura para o nome do ficheiro onde são guardados os dados das cargas dos *LocalCoordinators*.

A estrutura dos ficheiros de *reporting* é feita a pensar numa compreensão fácil para o utilizador, caso este queira analisar os dados guardados. Apenas o ficheiro que contém os dados do mapa não segue esta ideologia. O ficheiro que contém a informação do mapa é constituído pelo objeto serializado do *Global_map* gerado pelo *GlobalCoordinator*.

O ficheiro *reporting* que contém a informação relativa às movimentações dos atores possui os seguintes elementos:

- samplingTimestamp – data e hora do registo;
- id – identificador do ator;
- timestamp – data e hora da última movimentação do ator;
- X – posição em X;
- Y – posição em Y;
- , – separador dos dados;
- \n – indicador de mudança de linha.

Estes elementos são inseridos da seguinte forma:

- samplingTimestamp,id,timestamp,X,Y\n

Por último, o ficheiro que guarda a informação sobre a carga de processamento dos *LocalCoordinators*, possui os seguintes elementos:

- samplingTimestamp – data e hora do registo;
- IP – IP do PC ;
- timestamp – data e hora da ultima alteração da carga de processamento;

- carga – carga de processamento que o PC esta sujeito;
- , – separador dos dados;
- \n – indicador de mudança de linha.

Estes elementos são inseridos da seguinte forma:

- samplingTimestamp,IP,timestamp,load\n

5 Implementação

Após a exposição da análise do sistema a desenvolver e decisões, este novo capítulo explicita a implementação do que foi planeado. A implementação, como já foi referido, foi realizada na linguagem de programação *Java*.

A implementação desenvolvida nesta dissertação é relativa à componente de *Visualization*, à componente de *Reporting* e às alterações efetuadas ao código de uma versão anterior do simulador, nomeadamente a inclusão da leitura de um novo tipo de mapas (mapas OSM), com o intuito de melhorar funcionamento do simulador.

O desenvolvimento do simulador foi feito por *packages* de forma a ser mais organizada e mais fácil o trabalho em grupo. Desta forma, o *software* foi dividido em 6 *packages*, todos eles com o prefixo *um.simulator.*: *core*, *map*, *communications*, *actor*, *visualization e reporting*. Os *packages* que foram desenvolvidos nesta dissertação foram os packages *um.simulator.map, um.simulator.reporting e um.simulator.visualization*. No package *um.simulator.map* foram acrescentadas algumas classes e feitas alterações na classe *Global_map*. Seguidamente, serão apresentadas as classes que constituem os *packages* abordados nesta dissertação, que são as seguintes:

- *um.simulator.map.*:
    - *Global_map*;
    - *Map_Line*;
    - *Map_Point*;
    - *Point_Line*;
    - *MapaPaser*;
    - *Nodes*;
    - *Way*;

- *um.simulator.visualization.*:
  - *Visualization*;
  - *InterfaceBart*;
  - *InterfaceBartLoad*;
  - *SimMonitor*;
- *um.simulator.reporting.*:
  - *Reporting;*
  - *PlayReporting.*

Para uma compreensão mais objetiva da utilidade de cada uma das classes, assim como conhecimento de quais os seus métodos e variáveis, são apresentados nas figuras que se seguem os diagramas de classes do *BartUM* Simulator, frisando as partes desenvolvidas nesta dissertação.

Seguidamente será descrita a implementação dos *packages reporting*, *visualization* e das classes *MapaPaser*, *Nodes*, *Way*, e as alterações feitas na classe *Global_map*.

5.1 Diagramas de classes

Esta secção irá possuir os diagramas de classes dos processos constituintes deste simulador de forma a melhor identificar as classes desenvolvidas nesta dissertação que fazem parte do trabalho realizado em conjunto com outros elementos deste projeto. Por outro lado, também será possível observar o diagrama de classe do *Visualization.* A exposição destes diagramas de classes visa a realização de uma implementação estável e sem redundâncias.

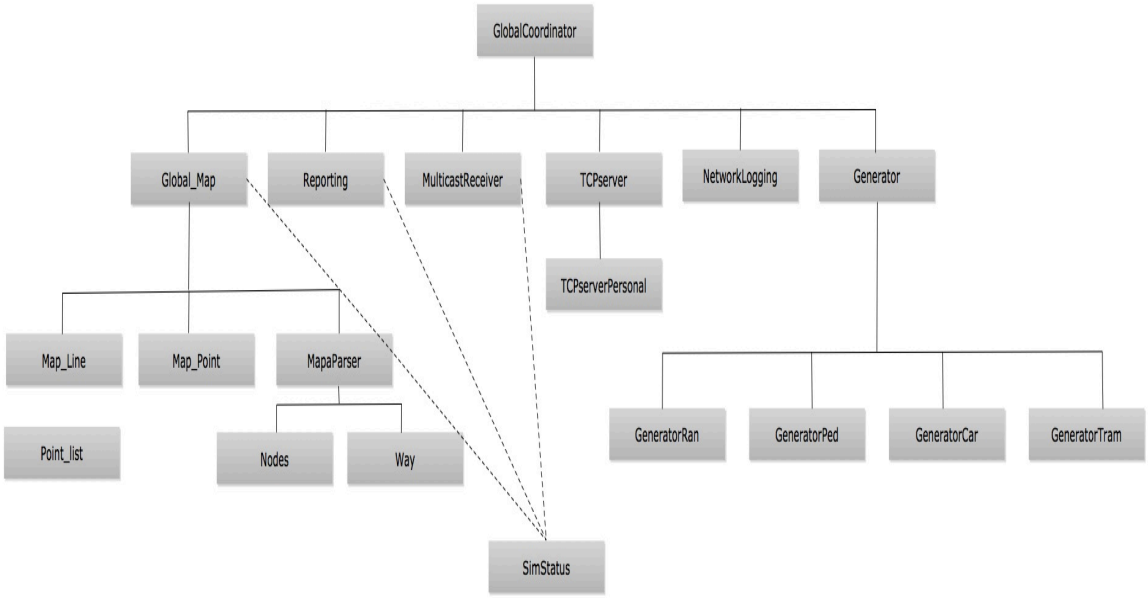


Figura 13 -Diagrama de classes do *GlobalCoordiantor*

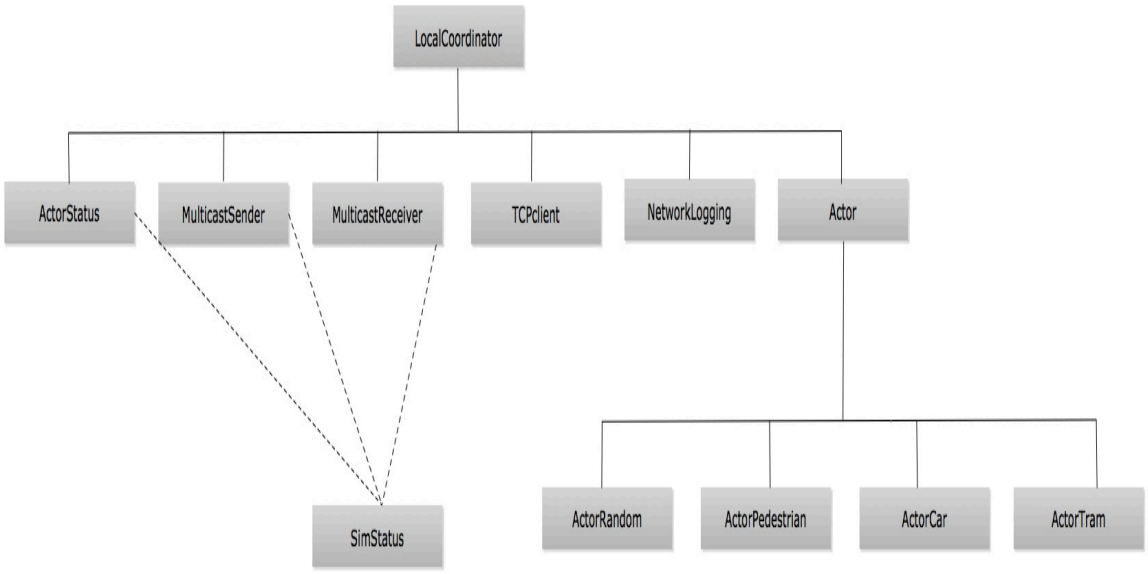


Figura 14 - Diagrama de classes do *LocalCoordinator*

Como é possível observar nestas imagens, o *GlobalCoordinator* contém na sua constituição as classes *Global_map*, *MapaPaser*, *Nodes*, *Way e Reporting.* As classes *Global_map*, *MapaPaser*, *Nodes* e *Way* são responsáveis pela integração de mapas OSM, pois até o desenvolvimento destas classes, o simulador só era capaz de ler mapas wkt. Assim, com a criação destas classes, o simulador é capaz de carregar este tipo de mapas para a

memória sem alterar a antiga estrutura de dados. A classe *Reporting* vem acrescentar ao simulador a capacidade de gravação no decorrer da simulação.

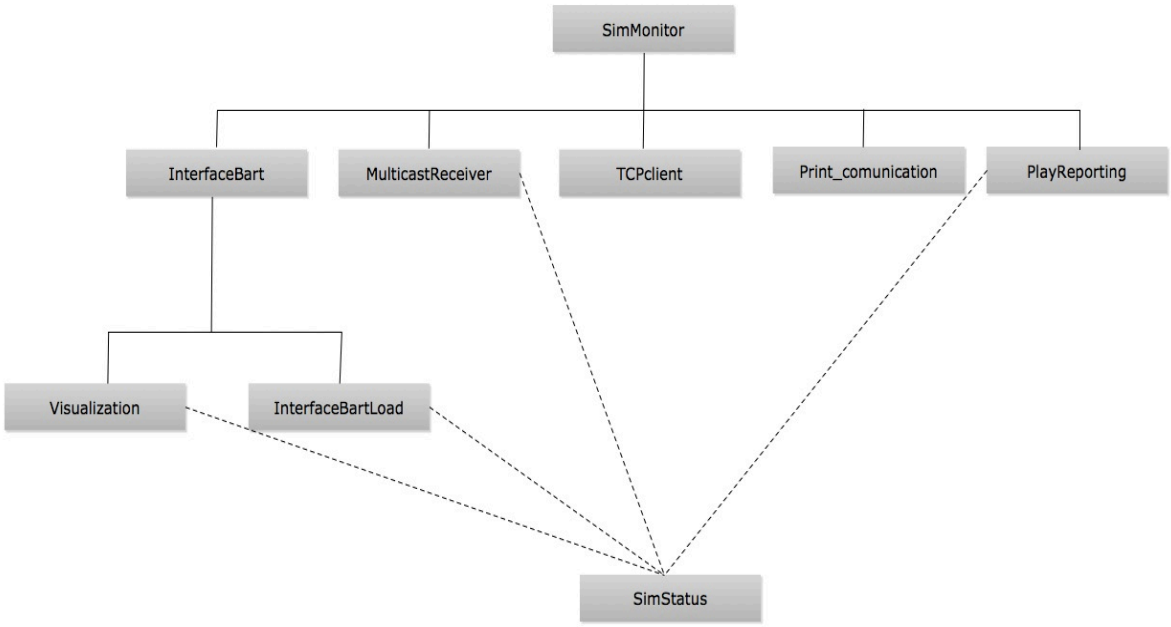


Figura 15 - Diagrama de classes do Visualization

5.2 Visualization

O *packageum.simulator.visualization*, é aquele que é composto por todas as classes que pertencem à representação gráfica do simulador.

5.2.1 SimMonitor

Esta é a classe central do *package visualization*, pois esta classe tem como propósito iniciar todos os serviços necessários para a visualização. Quando esta classe é iniciada recebe o tipo de visualização que o utilizador pretende ter, podendo ser visualização de uma simulação em tempo real.

Esta classe ao ser iniciada, inicia o cliente TCP e espera por uma resposta do *GlobalCoordinator* para seguidamente dar início ao *MulticastReceiver*. Se não receber nenhuma resposta do *GlobalCoodinator*, este imprime uma mensagem indicando ao utilizador que o *GlobalCoordinator* não responde. Após iniciar corretamente todas as comunicações,

inicia então a interface visual destinada à reprodução da simulação em tempo real. Os endereços IP e portas de comunicação são carregados através do ficheiro de configuração.

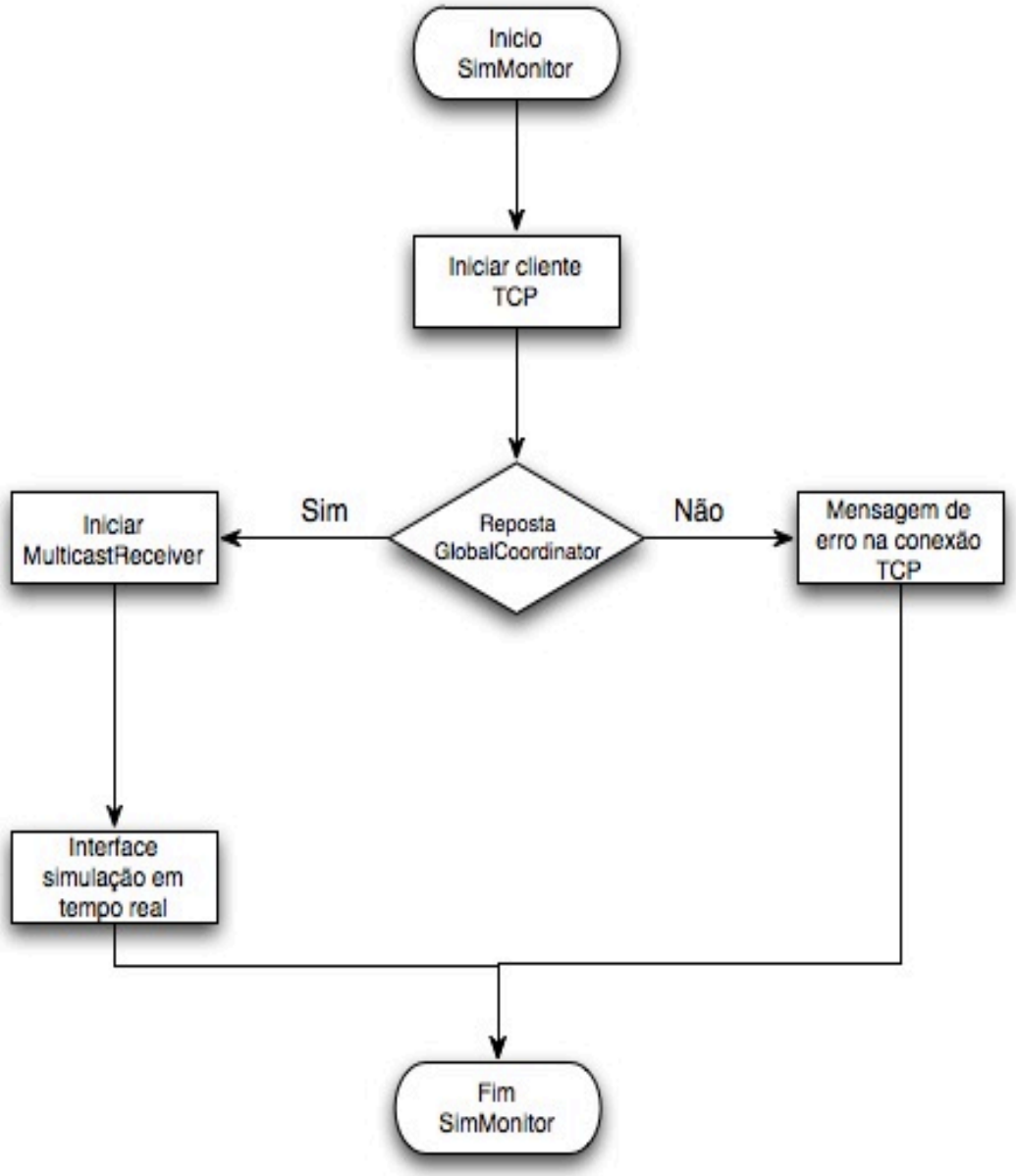


Figura 16 – Fluxograma de funcionamento da classe *SimMonitor*

5.2.1.1 Funções

Esta é uma classe bastante simples a nível de concepção. Possuindo, para além do construtor, apenas dois métodos: *initCom()* e *initSimMonitor()*.

A primeira função a ser invocada é a *initCom()*, esta função estancia as classes necessárias para as comunicações. Para estanciar estas classes, esta função carrega do ficheiro

de configuração o endereço do *GlobalCoordinator*, bem como a porta de conexão, de forma a criar a classe *TcpClient*. Para além disso, também carrega o endereço *multicast* para criar a conexão *MulticastReceiver*.

Seguidamente, é invocada a função *initSimMonitor*(). Este método vai inicializar as comunicações e, no caso de o *GlobalCoordinator* não responder, imprime uma mensagem que indica a inexistência de conexão, no entanto, se existir resposta, este vai proceder à criação da classe *InterfaceBart* para que seja possível a visualização.

5.2.2 InterfaceBart

Nesta classe são criadas as interfaces da visualização, sendo que esta pode reproduzir três tipos diferentes de interface.

Na figura apresentada a seguir (figura 17) está representada a interface que é apresentada quando é iniciada a *Visualization*. Esta é uma interface bastante simples, contendo apenas um menu na barra superior e uma caixa de texto.

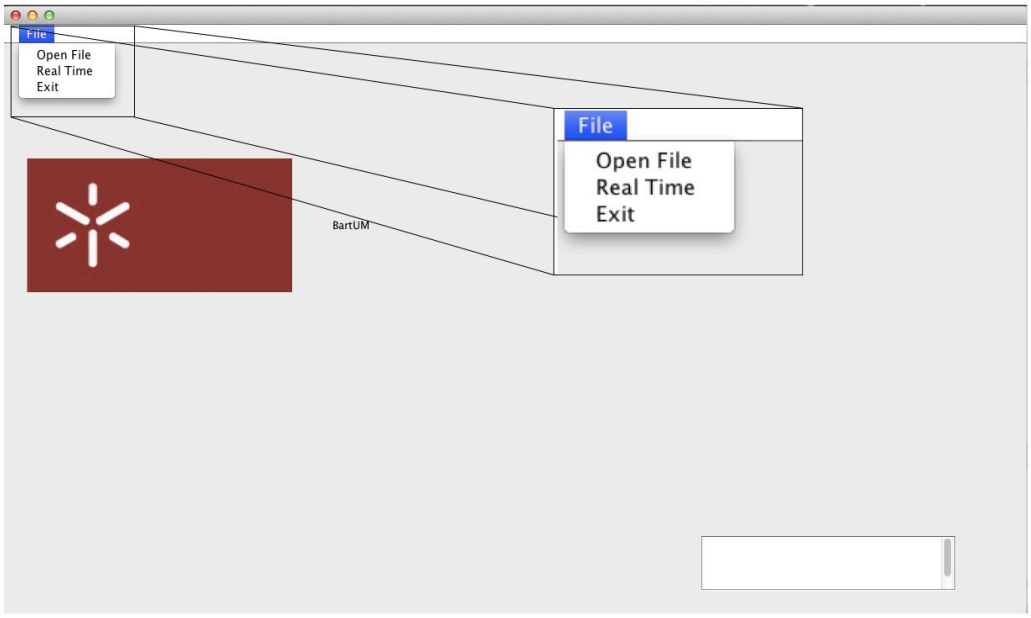


Figura 17 - Interface inicial

O menu presente na barra superior da interface oferece três opções ao utilizador. A opção Open File, que direciona para a *Visualization* em modo *reporting*, a opção Real Ti-

me, que direciona para a *Visualization* em tempo real e a opção Exit que termina a aplicação.

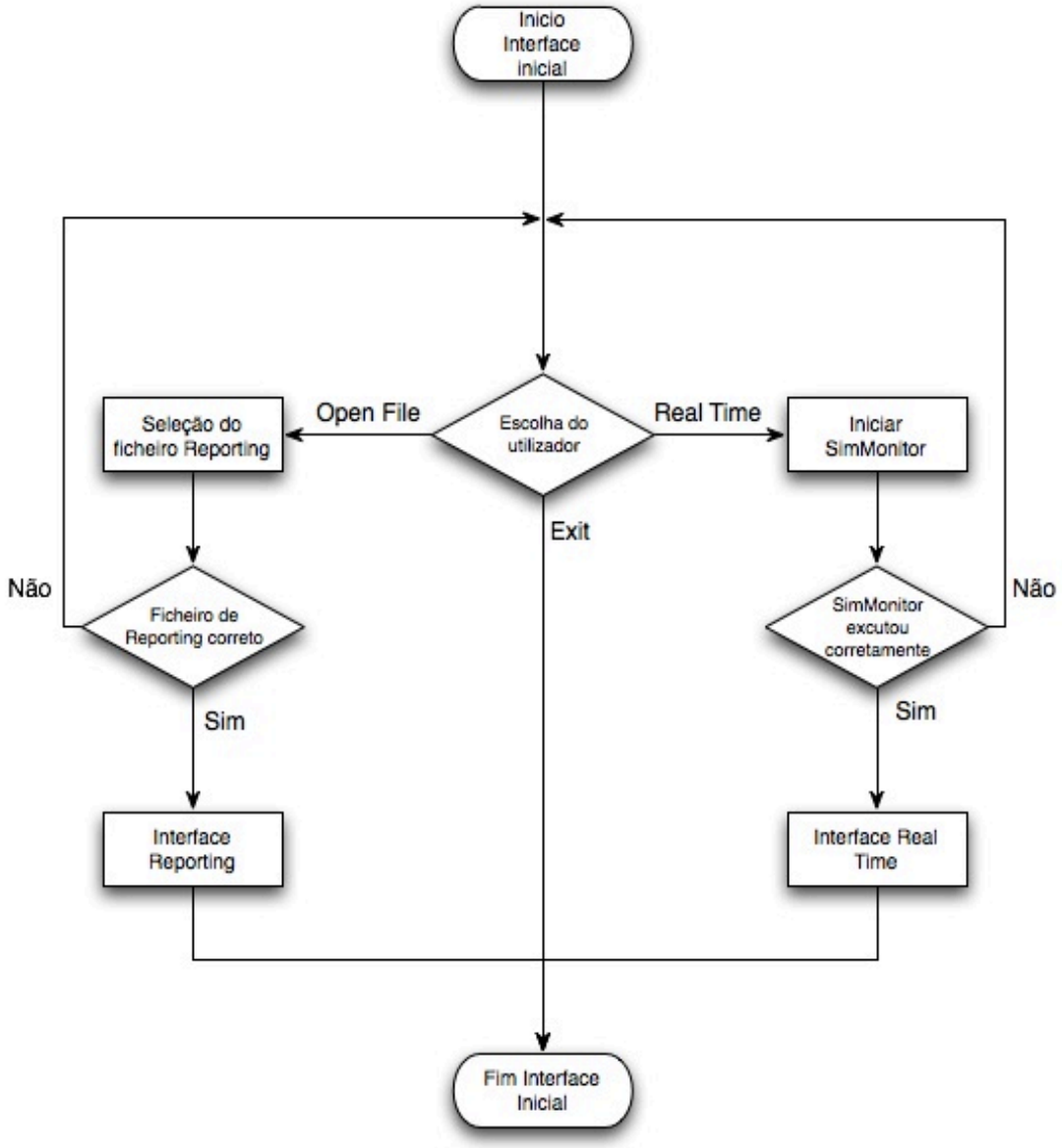


Figura 18 - Fluxograma de funcionamento da interface Inicial

Caso o utilizador selecione *Real Time*, isto é, a simulação em tempo real, a aplicação testa primeiro a conexão com o *GlobalCoordinator*, e caso exista algum problema com esta conexão será impressa uma mensagem de erro na caixa de texto da interface, indicando o erro de conexão que ocorreu. No caso em que a conexão é bem sucedida, inicia-se a interface para a simulação em tempo real (figura 19).

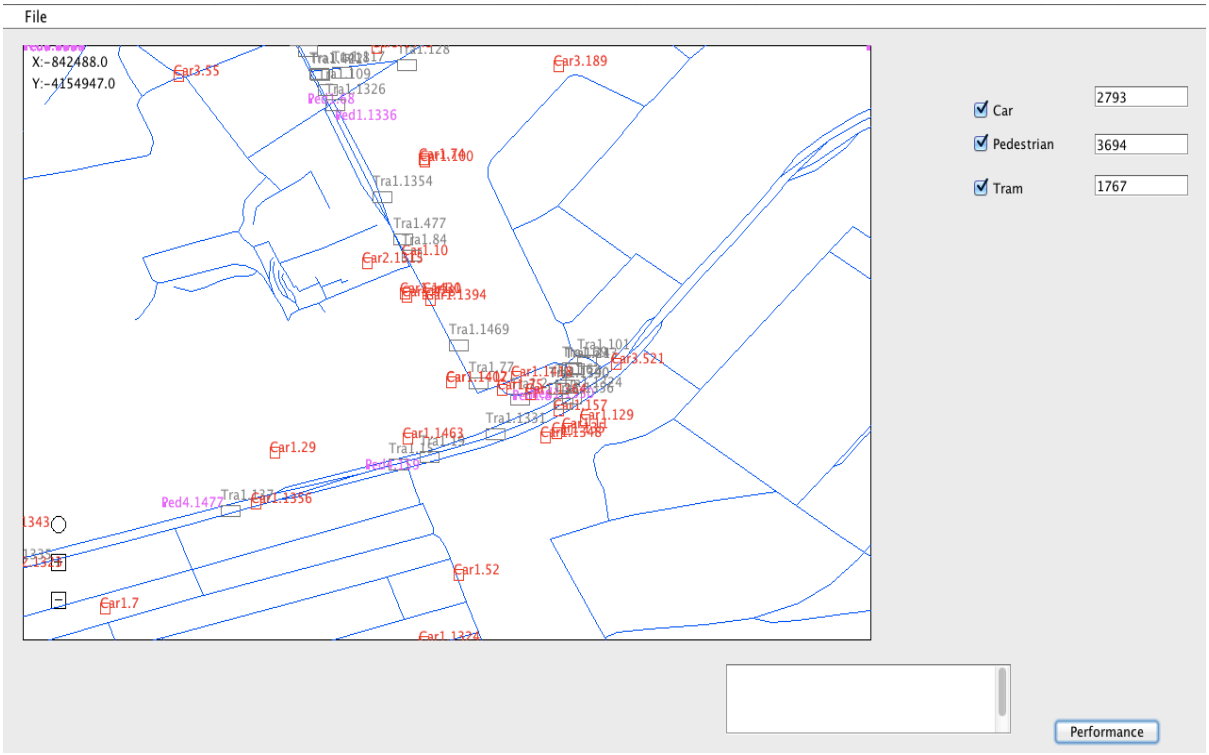


Figura 19 - Interface da Simulação em tempo real

A interface possui um *jpanel* (Classe *Visualization*) com a simulação que está a decorrer, sendo este uma caixa de mensagem que imprime informação relativa à simulação. À direita do *jpanel, existe* uma zona onde é possível ver o tipo de atores da simulação, bem como o número de cada tipo de ator. Associado a isso também tem uma *check box* por cada tipo de ator, permitindo ativar ou desativar a visualização de cada ator. Para além disso, existe um botão (*Performance*) que encaminha para outra janela onde é possível ver os *LocalCoodinators* ativos e a carga de cada um destes (figura 20).

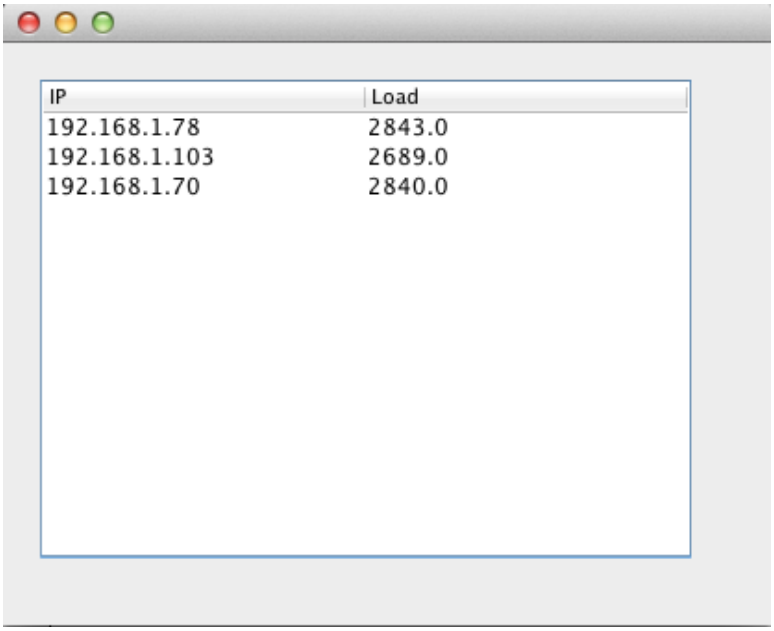


Figura 20 - Tabela de cargas

No caso em que é selecionado a opção Open File, surge uma janela que permite procurar os ficheiros de reporting (figura 21), para serem carregados pela aplicação. O utilizador dos três ficheiros de reporting (ficheiro com a informação do mapa, ficheiro com a informação das movimentações dos atores e ficheiro com a informação da variação das cargas dos *LocalCoordinators*) só necessita de carregar um deles. Podendo selecionar qualquer um dos três ficheiros, pois a aplicação, após selecionado um ficheiro, detecta quais os outros dois ficheiros relativos à mesma simulação. Caso o ficheiro carregado não seja um ficheiro de *reporting*, ou seja criado pelo *reporting*, é impressa uma mensagem de erro na caixa de texto da interface.

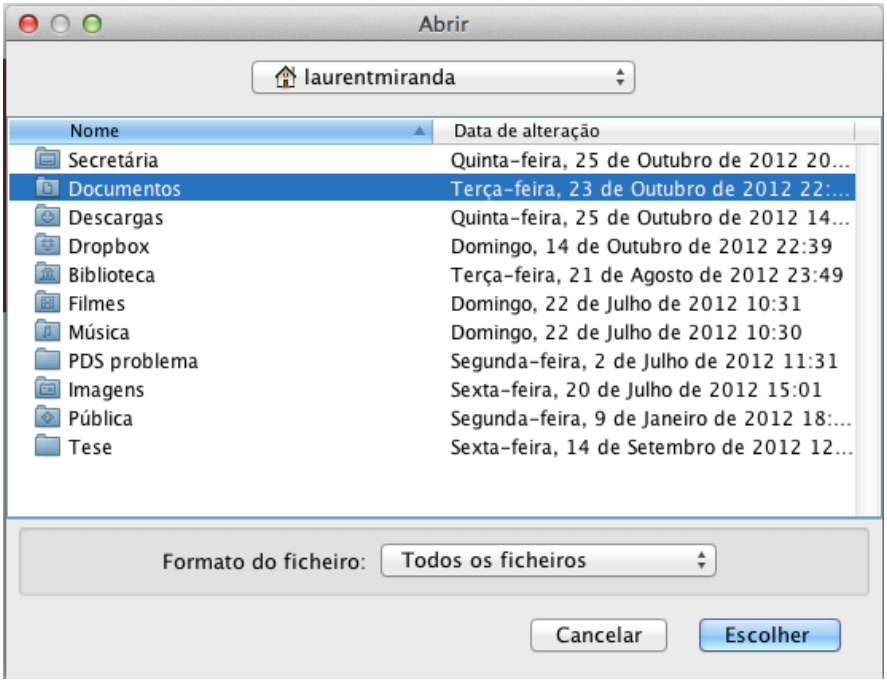


Figura 21 - Janela de procura de ficheiros

Após carregar os ficheiros de *reporting,* é iniciada a interface para o *reporting* (figura 22). Esta interface, como a interface anterior, contém um *jpanel* com a simulação que está a decorrer, uma caixa de mensagem que imprime informação relativa à simulação. À direita do *jpanel,* também se encontra uma zona onde é possível ver o tipo de atores da simulação, bem como o número de cada tipo de ator, associado a isso também tem uma *check box* por cada tipo de ator que permite ativar ou desativar a visualização de cada ator.

Para além disso, abaixo do *jpanel* existem botões, nomeadamente botão *Play*/*Pause, Stop*, *Fast* e *Slow*. Este conjunto de botões permite controlar a visualização do *reporting*. Com este botões é possível iniciar o playback da simulação, fazer pausa, retomar (botão *Play*/*Pause*), parar (botão *Stop*), fazer com que a simulação gravada decorra mais rápido (botão *Fast*, permite até 16 vezes mais rápido) ou mais lentamente (botão *Slow*, permite até 16 vezes mais lento). Abaixo do *jpanel* também existe uma *slider bar* onde é possível puxar a simulação para a frente ou para trás, e dois relógios, um com o tempo total da simulação e outro com o tempo atual onde a simulação se encontra. Por fim, e tal como na interface em tempo real, existe um botão (*Performance*) que encaminha para outra janela onde é possível ver os *LocalCoodinator*s ativos e a carga de cada um destes. É importante voltar a frisar que

a informação que gera o *reporting* é relativa a uma simulação já decorrida guardada em ficheiro.

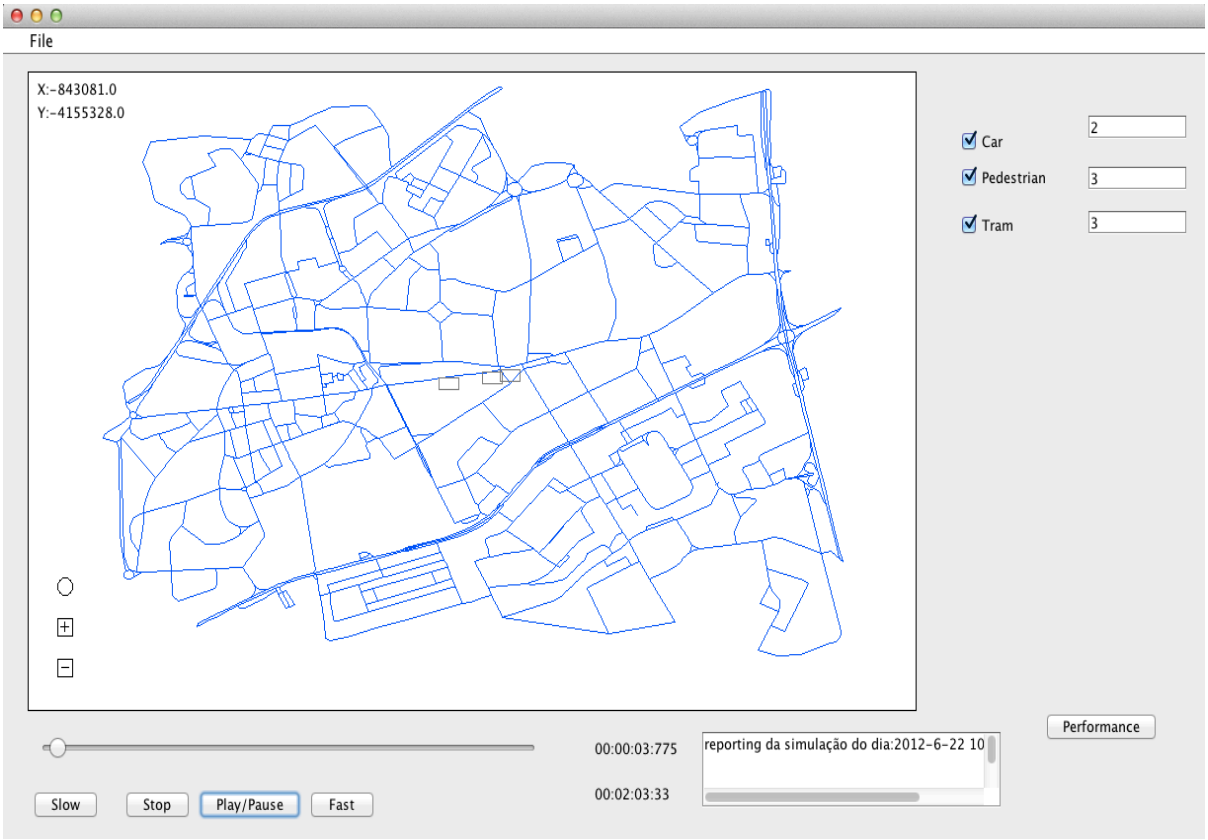


Figura 22 - Interface Reporting

5.2.2.1 Funções

Esta classe é uma classe *jframe* que é destinada à construção de interfaces gráficas. Com o uso do *netbeans*, o desenvolvimento de classes do tipo *jframe* é bastante facilitado. Pois o *netbeans* possui um grande número de elementos gráficos já construídos, como botões, caixas de texto, *Check Box*, etc. Nesta dissertação, utilizou-se essa funcionalidade do *netbeans* para conceber os interfaces gráficos. Esta classe possui um grande número de funções, por isso optou-se por explicar o funcionamento das funções mais importantes.

Uma das funções mais importantes desta classe é a função construtor. A função construtor recebe como parâmetro (*String type*) o tipo de interface que tem de criar. No caso em que a variável recebida por parâmetro tenha conteúdo vazio (“”), é criado o interface inicial (figura 17), caso o conteúdo da variável seja *realtime* é criado o *jpanel* (Classe *Visualization*) onde a simulação irá decorrer e as componentes que formam interface de tempo

real (figura 19). Por último, se a variável contiver o endereço físico do ficheiro, é iniciada a classe *PlayReporting*, criando o *jpanel* (Classe *Visualization*) onde é visível a simulação e as componentes da interface para o *reporting*.

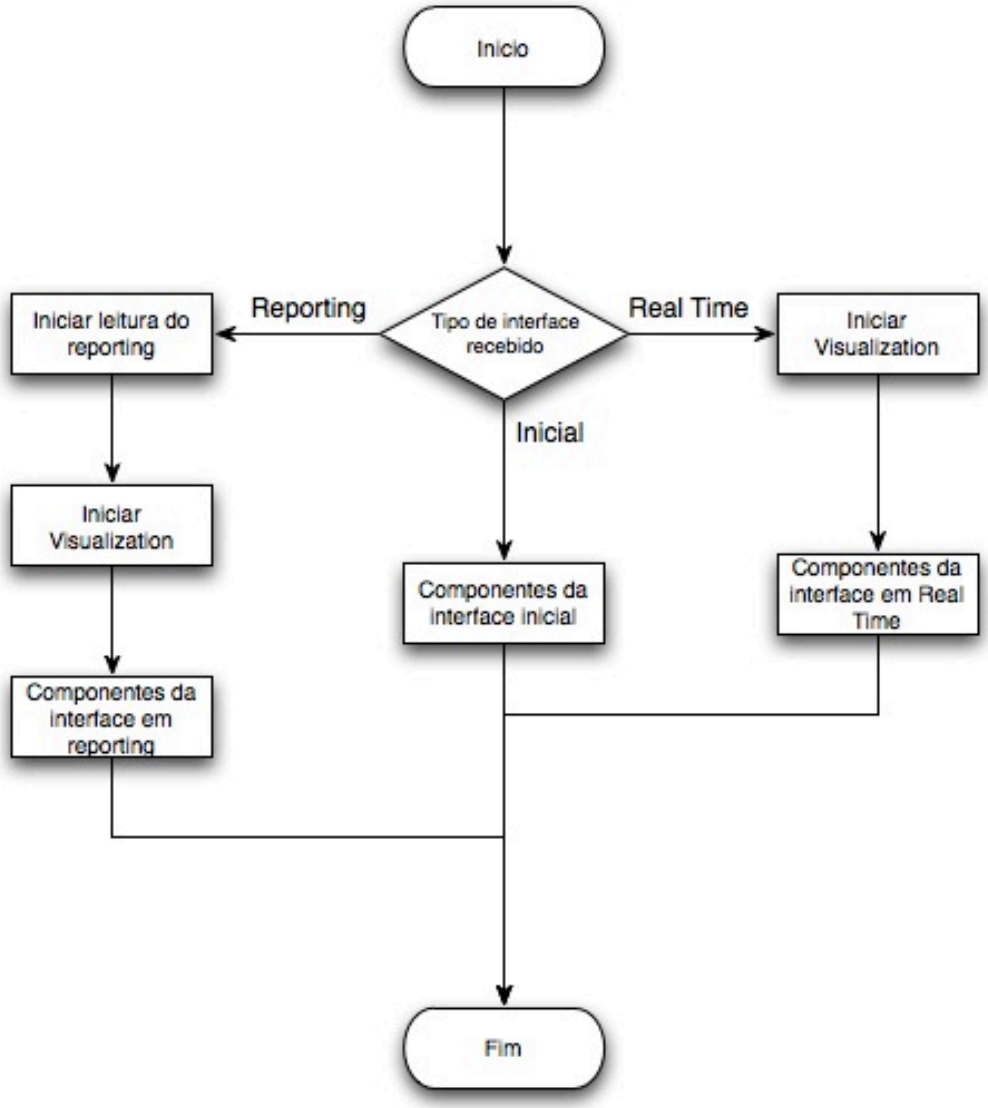


Figura 23 - Fluxograma da função construtor da classe InterfaceBart

No menu file (figura 17) como referido anteriormente, existem três opções: *open file, real time* e *exit.* Quando pressionado *open file* é invocada a função *jReportingActionPerformed* (java.awt.event.ActionEvent evt). Esta função cria a janela de procura de ficheiro (figura 21) com o auxilio do *netbeans* criando um *JFileChooser*(). Após a escolha do fichei-

ro esta função recebe o endereço do ficheiro e verifica se esse ficheiro corresponde a um ficheiro de *reporting*, com a invocação da função *checkfile(fname)*. Caso o ficheiro não seja de *reporting* é enviada uma mensagem de erro na caixa de texto da interface. Caso seja um ficheiro de *reporting* é feito *clear* da memória do visualizador, para garantir que não tenha informação, é criado uma interface em modo *reporting* e é terminada a interface que está a ser executada.

No caso em que é escolhido *real time,* é enviada a função *jRealtimeActionPerformed (java.awt.event.ActionEvent evt)*, que faz o *clear* a memória do visualizador e instancia a classe *SimMonitor* e executa-a. Caso a execução do *SimMonitor* decorra como o previsto, é terminada a interface que está a ser executada, caso isso não aconteça é enviada uma mensagem de erro na caixa de texto da interface.

Quando é selecionada a opção *exit,* é invocada a função jExitActionPerformed(java.awt.event.ActionEvent evt) que termina o visualizador.

A construção do menu que permite ver o tipo de ator, o número de cada tipo de atores e colocá-los visíveis ou invisíveis, tem uma particularidade na sua construção (Figura 24). Pois é construído conforme a simulação decorre, isto é, o primeiro tipo de ator a surgir na simulação (ou no *reporting*) ocupará a posição mais acima do menu. Conforme vão surgindo novos tipos de atores, vão ocupando as posições seguintes. Este menu é feito com auxílio da função *menuAtor(String typeAt)*. Esta função recebe o tipo de ator que é pretendido adicionar ao menu, e com o auxílio de uma variável global (*Yposimenu*) constrói o menu. Inicialmente, a variável global possui o valor do eixo y correspondente da primeira posição do menu, conforme vão surgindo os diferentes tipos de atores vai sendo incrementada fazendo com que os outros parâmetros do menu vão ficando uns abaixo dos outros.

Figure 24 - Menu de atores

Outra função importante, é a função que é invocada quando é pressionado o botão play/pause (jButtonPlayPause(java.awt.event.ActionEvent evt)). Esta função tem um algoritmo que verifica se o botão é pressionado pela primeira vez ou não. Caso seja a primeira vez, inicia automaticamente a reprodução da simulação gravada. Caso não seja, verifica se a simulação se encontra em reprodução ou parada. Caso esteja em reprodução, pára essa reprodução, caso esteja no estado parado volta a colocá-la em reprodução.

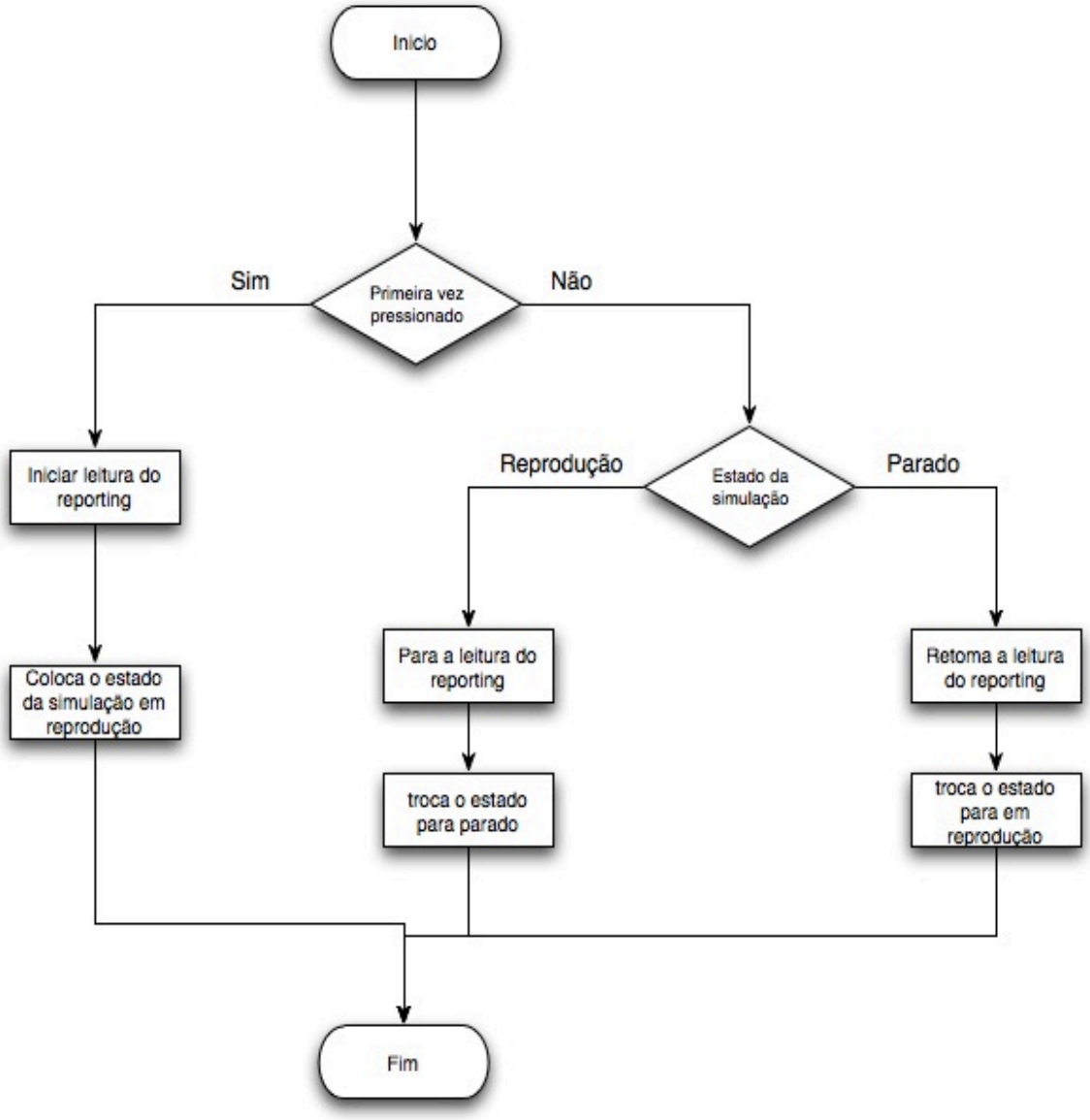


Figura 25 - Fluxograma função Play/pause

Esta classe também possui uma função criada automaticamente pelo *netbeans*, função *initComponents*(), esta função é responsável pelas componentes constituintes da interface criadas com o auxílio do *netbeans*. Nesta função encontra-se o código de criação desses componentes. Esta função não pode ser alterada.

5.2.3 InterfaceBartLoad

Esta classe é um *jframe*, sendo uma classe bastante simples, sendo que esta interface apenas possui uma tabela com a lista de *LocalCoordinator* e as suas cargas. A tabela é dinâmica, pois consoante vão surgindo novos *LocalCoordinator,* vão sendo acrescentados à tabela, e a carga também aumenta consoante as notificações recebidas (figura 20).

5.2.3.1 Funções

Como a classe *InterfaceBart,* esta classe é criada com o auxílio do *netbeans*, sendo também uma classe *jframe,* facilitando assim a sua concepção. Esta classe, para além do construtor, possui apenas três outras funções. Esta, como foi criada com o auxílio do netbeans, também possui a função *initComponents*(). As outras duas funções são as funções *run*() e *table*().

A função *run*(), de 100 em 100 milissegundos, vai atualizar a tabela, através da invocação da função *tabel*().

A função *tabel*() carrega os dados relativamente à carga dos *LocalCoordinator*s, através da memória partilhada do simulador, e imprime estes dados na tabela. Para isso percorre a lista de LocalCoordinators ativos e retira a carga de processamento de cada um.

5.2.4 Visualization

Esta é a classe mais importante do *package Visualization*, pois esta classe tem como tarefa criar a representação gráfica da simulação. Usando a biblioteca *Java Graphics e Java Graphics2D*, esta classe primariamente carrega os dados do mapa que são pretendidos projetar, alocados em memória (*SimStatus*), e cria uma projeção do mapa. Seguidamente, lê os dados relativos aos atores, de 35 em 35 milissegundos, sendo que também estes são carre-

gados do *SimStatus* e projetados numa representação gráfica de cada ator. O simulador, atualmente, só possui três tipos de atores (car, pedestrian e tram). Os atores do tipo Car são representados graficamente por um quadrado só com os contornos pintados de vermelho, os atores Tram são representados por um retângulo também só com os contornos pintados de cinzento e o atores do tipo Pedestrian são representados por um circulo totalmente preenchido de cor de rosa.

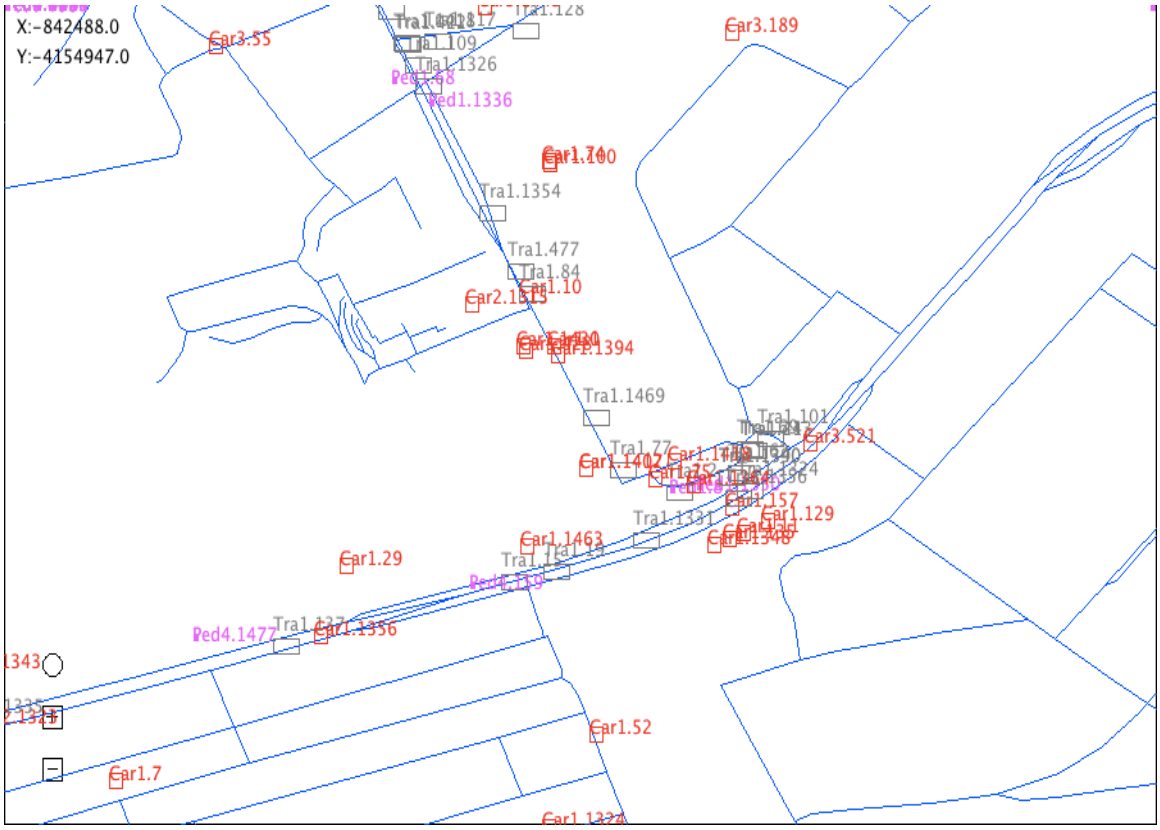


Figura 26 – Janela de reprodução da simulação

Esta classe funciona de igual modo para *reporting* ou visualização em tempo real, pois o que difere destes dois modos de reprodução é somente a maneira como os dados são lidos. Sendo que, em ambos os casos, os dados são guardados na memória, por isso, não interfere com o funcionamento desta classe.

Esta classe tem como particularidade o facto de possuir uma classe primária (*visualization*) e uma classe secundária (*drawActor*). Este tipo de programação é usada para que haja uma maior organização e para simplificar a reprodução da simulação. Deste modo, o

funcionamento é o seguinte: a classe primária, após a sua criação, invoca o método *paint* (*Graphics g*) - este método é um método característico das classes que usam as bibliotecas *Java Graphics* – onde, neste momento, só é impresso o mapa. Após este passo, a classe primária executa a sua *thread* que está, como já foi referido acima, a efetuar a leitura dos dados dos atores de 35 em 35 milissegundos, verificando ator a ator presente na simulação. Caso receba um ator novo, é criada uma classe secundária (classe *drawActor*), onde fica guardado o posicionamento do ator. Caso o ator já exista, é simplesmente atualizada a sua posição na classe secundária. A classe secundária é que irá criar a representação gráfica do ator, mas não a imprime. A impressão dos atores é feita pela classe primária que tem um *buffer* com todos os atores à espera de serem impressos.

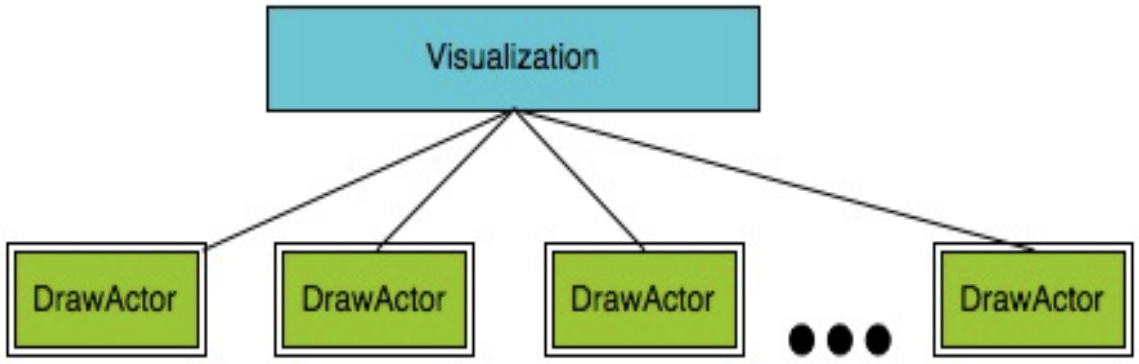


Figura 27 - Relação entre classes

5.2.4.1 Visualization classe primária

Essa classe, como já foi referido, é responsável por reproduzir a simulação. Mas para além disso, é responsável por outras funcionalidades relativas à visualização da simulação, funcionalidades estas como o *zoom* da simulação, indicar a coordenada no mapa relativa à posição onde o cursor do rato se encontra e percorrer o mapa com o arrastamento do rato. Esta classe também comunica com a classe *InterfaceBart* para transmitir os dados relativos ao número de cada tipo de ator e quando é criado um novo tipo de ator. Para além disso, recebe os alertas do *InterfaceBart* para notificar que o tipo de ator é visível ou invisível, e transmite essa informação à sua classe secundária.

5.2.4.1.1 Funções

Esta é uma classe com muitas funções, por isso optou-se por explicar as partes do código consideradas mais importantes.

Uma das partes mais importantes do código desta classe, senão a mais importante, é a parte responsável pelo desenho das componentes. Quando o mapa é carregado, é de certa forma redimensionado para ficar com o tamanho da janela e colocar o mapa no centro da janela. Esta operação é feita no método paint(), sendo efetuada da seguinte forma: primeiro são encontrados os pontos máximos (x máximo e y máximo) e os pontos mínimos (x mínimo e y mínimo) através do método FindM(Global_map map). Com esses máximos e mínimos é calculada a escala (calcEscal()) em que o mapa tem que se cingir para respeitar dimensões da janela, e um valor mínimo de cada eixo é subtraído ao valor correspondente ao mesmo eixo de cada ponto, de forma a respeitar o sistema de coordenadas da janela, que como foi explicado anteriormente, não possui valores negativos. Para uma melhor compreensão segue-se um exemplo:

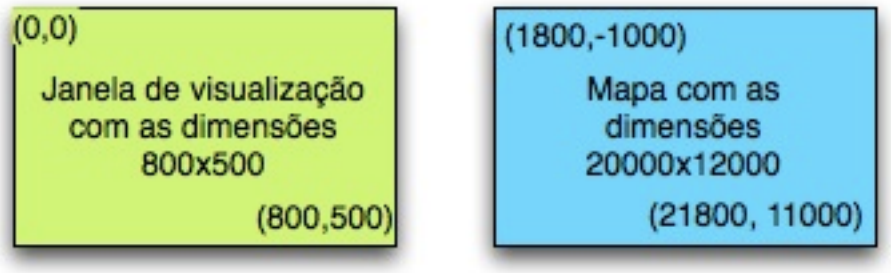


Figura 28 – Exemplo para calculo de escala

Neste caso temos um mapa com as dimensões 20000x12000 em que as coordenadas mínimas são (1800,-1000) e máximas (21800,11000). Para dimensionar este mapa de forma a ficar dentro da janela de visualização é necessário as seguintes operações:

Escala x = (x máximo do mapa - x mínimo do mapa) / x máximo da janela de visualização
Escala y = (y máximo do mapa - y mínimo do mapa) / y máximo da janela de visualização

Figura 29 -formulas de cálculo da escala

Neste caso a escala x é igual a 25 e a escala y é 24, com o x mínimo de 1800 e o y mínimo de -1000. Agora, para dimensionar um ponto do mapa para a janela de visualização, é necessário subtrair a coordenada x desse ponto com o x mínimo e dividir tudo pela escala de x, fazendo a mesma coisa para a coordenada y (exemplo (21800-1800)/25=800 (1100-(-1000))/24= 500). Após esses cálculos, pode-se proceder ao desenho do mapa. O mapa está dividido por estradas, essas estadas são constituídas por pontos, o desenho do mapa é feito através do desenho de linhas que unem os pontos das estradas. Por fim, quando terminado o desenho do mapa, é percorrido o *buffer* de atores a ser desenhado. O cálculo da escala só é efetuado na primeira vez que a função paint() é invocada.

Nesta classe também são feitas as operações de *zoom* da simulação. Na simulação é possível encontrar os seguintes botões de *zoom*:

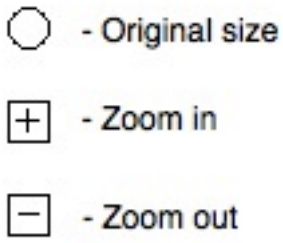


Figura 30 - Botões Zoom

O *zoom* é feito alterando os valores da variável escala. Se for pretendido fazer *zoom in* a variável escala diminui, sendo que quanto maior for a variável escala, mais ela diminui de forma a que o *zoom* seja mais rápido. Acontece a mesma coisa no caso contrário (*zoom out*), só que neste caso a variável cresce, e quanto maior for mais rapidamente cresce. O botão *original size* coloca o mapa do tamanho da janela, como se encontrava no início da simulação.

5.2.4.2 DrawActor Classe secundária

Esta classe serve para guardar o posicionamento do ator, criar uma reprodução gráfica do ator de acordo com o tipo de ator e as especificações do utilizador para cada ator. Para além disso, também é esta classe que coloca o ator visível ou invisível.

5.2.4.2.1 Funções

Esta classe possui cinco funções. O construtor (*public DrawActor*(double x, double y, String ID, String label)) que recebe como parâmetros as coordenadas (a posição em X e a posição em Y), o identificador do ator e a *label* com informação extra sobre o ator. Possui uma função para alterar as coordenadas do ator *(move(*double x1, double y1)). A função *paint(),* mas no caso desta classe essa função é invocada pela função *paint*() da classe primária. É nesta função que é testado o tipo de ator e verificado se este tipo de ator está no estado visível ou invisível, isso é determinado através da invocação da função *visible()*. Em seguida, cria a imagem gráfica do tipo de ator que ela carrega (forma e cor). Por fim, a função *private AlphaComposite makeComposite(float alpha)* serve para criar o modo transparente para quando for necessário colocar um tipo de ator invisível.

5.3 *Reporting*

O *package um.simulator.reporting* inclui duas classes. A classe *Reporting* que serve para gravar a simulação e é usada na aplicação *GlobalCoordinator* e a classe *PlayReporting* serve para a leitura e interpretação da simulação gravada pela classe *Reporting*. Esta classe é usada pela aplicação visualização quando é pretendido ver uma simulação gravada.

5.3.1. Reporting

Esta classe é responsável por gravar as simulações. Esta classe encontra-se anexada ao *GlobalCoordinator*, sendo obrigatória a presença do *GlobalCoordinator* para acontecer uma simulação. Assim, é garantido que todas a simulações são gravadas. Como esta classe está associada ao *GlobalCoordinator,* beneficia do facto do *GlobalCoordinator* possuir todas as informações necessárias para o *reporting*, como os mapas, informações relativas aos atores e a informação da carga de processamento dos *LocalCoordinators*.

De seguida, irá ser explicado o funcionamento do *reporting*. Após o globalCoordinator ser iniciado e estar pronto para iniciar a simulação, cria o *reporting*. Mal é criado o *reporting*, grava o mapa em ficheiro. Esse ficheiro contém o objeto serializado do *Global_map* onde se encontram todas as informações relativas ao mapa. Este é um ficheiro do tipo txt e contém como nome a data e hora da simulação com um prefixo *map*. Imediatamente após o surgimento do primeiro *LocalCoordinator* na simulação, inicia-se a fase de gravar os posicionamentos dos atores e a carga de cada *LocalCoordinator* presente na simulação. Para fazer a gravação, o *reporting* acede os dados que necessita através da memória *Globalcoordinator (SimStatus)*. Esses dados são carregados de 35 em 35 milissegundos. Para garantir que são exatamente 35 milissegundos entre cada gravação, retira-se o tempo antes da leitura e gravação de todos os dados, e é retirado o tempo imediatamente depois. Seguidamente, retira-se aos 35 milissegundos o tempo demorado, e obtêm-se o tempo necessário de espera para concluir os 35 milissegundos.

5.3.1.1 Funções

Esta classe, quando é criada, gera os três ficheiros onde irá ser guardada cada parte da simulação: ficheiro da carga de processamento dos *LocalCoordinators*, ficheiro com os movimentos dos atores, e o ficheiro com os dados do mapa. Para obter parte do nome dos ficheiros, invoca a função *gettimenow*(). Esta função retorna à data e hora com o seguinte formato ano-mês-dia hora:minuto:segundo:milissegundo. Após criar os ficheiros, copia o objeto serializado *Global_map* para o ficheiro de *reporting* dos dados do mapa.

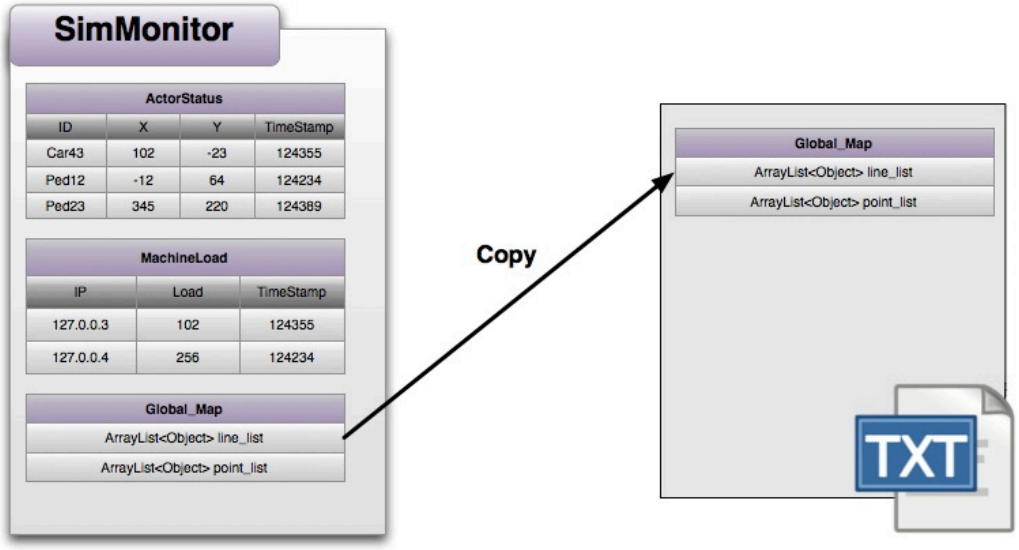


Figura 31 – Método de criação dos ficheiros *reporting* do mapa

A função *run*() desta classe é iniciada quando o primeiro LocalCoordinator se conecta ao *GlobalCoordinator*, ou seja, quando se inicia a simulação. Esta função vai operar durante o tempo total da simulação. Esta função lê os dados relativos aos atores e a carga de processamento dos *LocalCoordinator*, presentes na memória do *GlobalCoordinator* de 35 em 35 milissegundos. Esses dados são os ficheiros de carga dos *LocalCoordinators* e ficheiro com os movimentos dos atores.

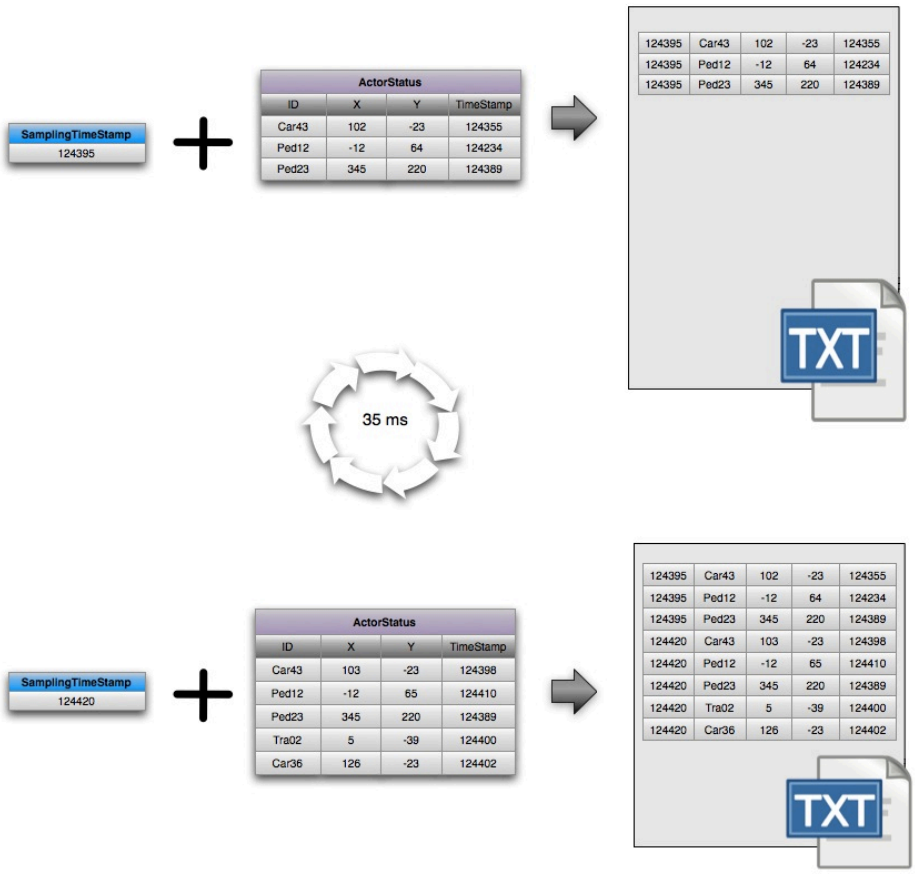


Figura 32 - Método de criação dos ficheiros de dados dos atores

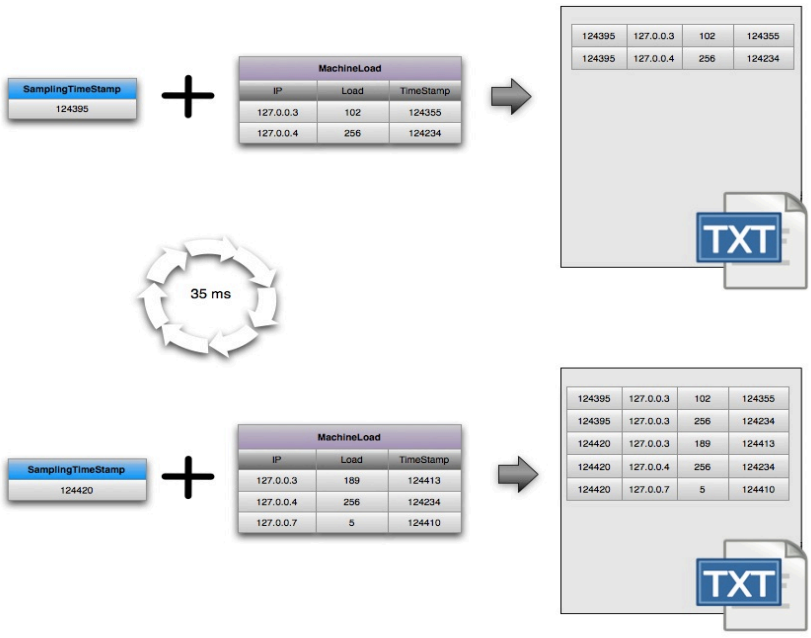


Figura 33 - Método de criação dos ficheiros de carga dos LocalCoordinatores

5.3.2 PlayReporting

Esta classe trabalha conjuntamente com a aplicação *visualization*. Quando a *visualization* pretende reproduzir uma simulação previamente gravada, é invocada esta classe para ler os dados guardados nos ficheiros de *reporting*. Esta classe é invocada e carrega imediatamente do ficheiro do mapa os seus dados para memória. Para além disso, cria os apontadores necessários para a leitura dos outros ficheiros. Como já foi referido, o utilizador seleciona apenas um dos três ficheiros de *reporting* (ficheiro dos dados do mapa, ficheiro dos dados dos atores e ficheiro dos dados das cargas dos *LocalCoordinators*). Esta classe é responsável por carregar os restantes ficheiros. Esse processo é feito através do manuseamento do nome dos ficheiros, pois esses só diferem no prefixo, sendo que o restante do nome é idêntico. Então, esta classe identifica o prefixo, retira-o do nome e acrescenta os outros prefixos para criar a leitura de todos os ficheiros, para isso todos os ficheiros têm de estar na mesma diretoria.

Quando é solicitado o *play* da gravação, esta classe começa a ler os dados da simulação, lendo os dados de 35 em 35 milissegundos, e atualiza a memória do *Visualization* (*SimStatus*) com esses dados. Para garantir que este tempo é comprido, é utilizado o tempo que está presente na gravação. Pois, por cada gravação existe a hora em que foi gravado, ou seja, os dados dos atores são lidos até o tempo de gravação mudar, quando esse tempo muda, a classe subtrai o tempo inicial ao tempo final (que se correto, irá dar 35), o tempo de leitura desses dados. O resultado desta operação ainda é dividido pela variável *speedR*. Esta variável inicialmente tem o valor 1, mas se for pretendido acelerar ou diminuir a velocidade de visualização, é só diminuir ou aumentar o valor desta variável, o que fará aumentar ou diminuir o tempo de leitura dos dados. Esta classe possui sempre um apontador para o início dos ficheiros dos dados de movimentação dos atores e da carga dos *LocalCoordinator*. Quando o utilizador procurar um tempo na barra do tempo presente na interface, este apontador procura pelo tempo desejado e coloca este apontador como o apontador de leitura. A partir desse momento, a memória do *Visualization* (*SimStatus*) é atualizada com os dados relativos ao tempo selecionado. É importante referir que a estrutura memória *SimStatus* não guarda os dados dos movimentos ou cargas anteriores, apenas os atuais, por isso é sistematicamente atualizada.

5.3.2.1 Funções

Esta classe quando é instanciada, portanto na função construtora, cria apontadores de leitura para os ficheiros *reporting* e carrega os dados do mapa. Para criar esses apontadores necessita o endereço físico desses ficheiros. Para isso, a função construtora invoca a função *returnAddress(String address)*, passando com argumento o endereço do ficheiro selecionado. A função *returnAddress(String address)* retorna um *array* de *Strings* com o endereço físico dos três ficheiros. Esses endereços são obtidos através da identificação do prefixo do nome do ficheiro recebido, através disso é possível retirar esse prefixo e colocar os prefixos dos outros ficheiros, obtendo assim o endereço dos ficheiros relativos à simulação carregada.

O carregamento dos dados para a memória do *Visualization*(*SimStatus*) é feito através da função *run()* desta classe, esta função é invocada quando o utilizador prime pela primeira vez o botão Play/Pause da interface. Nesta função são lidos de 35 em 35 milissegundos os dados dos ficheiros de carga dos *LocalCoordinators* e das movimentações dos atores, seguidamente, são carregados para a memória *SimStatus*.

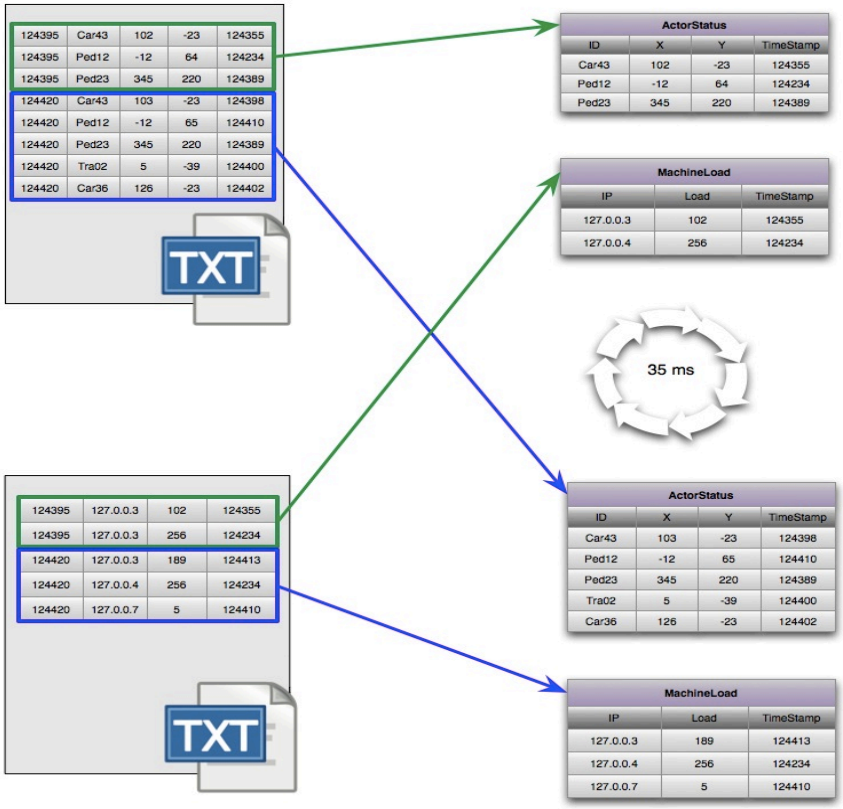


Figura 34 - Inserção dos dados dos ficheiros para a memoria SimStatus

Quando na interface é premido o botão Play/Pause com o intuito de parar a reprodução é invocada a função *wait*(). Esta função congela a execução do método *run()*. Quando é pretendido retomar a reprodução, ou seja, quando é premido novamente o botão *Play/Pause*, é invocado a função *notify*() que faz com que o método *run()* retome a sua execução.

A interface *Reporting* também interage com esta classe quando são premidos os botões *Stop*, *Fast*, *Slow* e a *scroll bar*. O botão Stop faz com que seja invocado a função *stop*(), o que faz parar definitivamente a execução do método run(). Quando é premido um dos botões de *Fast* ou *Slow* é invocada a função *setSpeedReporting(double speed).* Esta função recebe como parâmetro um valor que é multiplicado pela variável *speedR,* o que vai fazer com que esta variável aumente ou diminua, dependendo do botão que invocou esta função. Caso seja o botão *Fast* a variável é multiplicada por 2 o que faz com que aumento a velocidade de leitura (supondo que a variável *speedR* tenha o valor 1 o calculo do tempo de leitura é o seguinte 35/(1*2)= 12,5). No caso em que o botão premido seja o *Slow*, a variável *speedR* é multiplicada por 0,5 diminuindo a velocidade de leitura.

Quando é selecionado um momento da simulação através da *scroll bar* da interface é invocada a função *scorllTime(long newtime*). Esta função recebe como parâmetro o tempo escolhido pelo utilizador, seguidamente procura esse tempo, ou o tempo mais próximo desse no ficheiro onde estão os dados das cargas dos *LocalCoordinators*. É importante referir que ambos os ficheiros que estão a ser percorridos neste período de execução possuem os mesmos tempos, e foi escolhido o ficheiro de cargas porque é menor, ou seja, assim a procura é mais rápida. Quando encontrado esse tempo, o apontador que tem do ficheiro, substitui o apontador que estava a correr o ficheiro no método *run*().

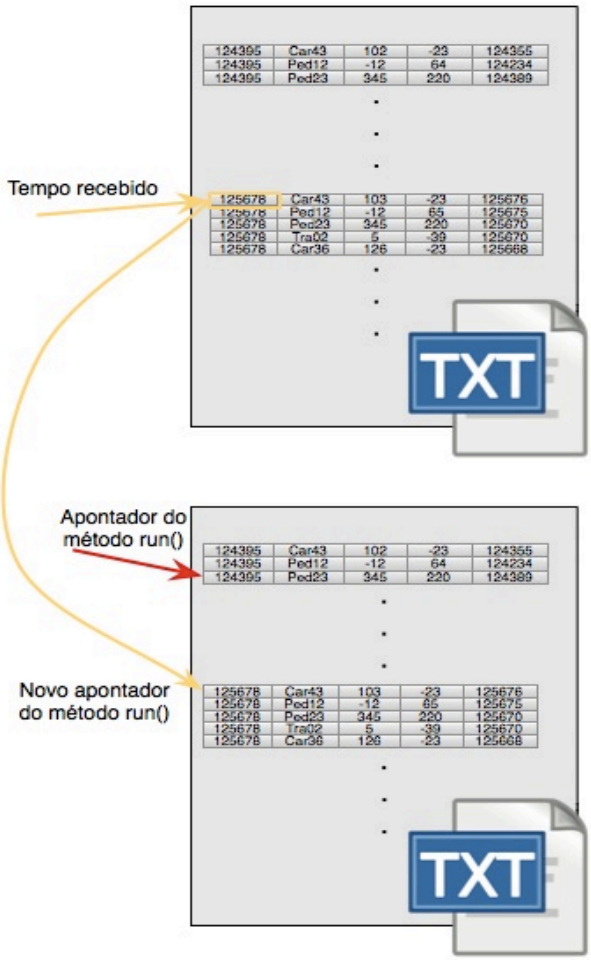


Figura 35 - Mudança de apontadores

5.4 Implementação da leitura dos mapas OSM

O simulador desenvolvido inicialmente só tinha a capacidade de ler mapas *wkt*, mas com o evoluir do simulador surgiu a necessidade de possuir mais informação sobre o mapa como, saber qual é o tipo de estrada, um caminho de ferro ou um passeio, quais os limites de velocidade de uma dada estrada, se é uma estrada principal ou secundária, se existe um ou dois sentidos numa dada via, etc.

A alternativa escolhida foi o OpenStreetMaps (OSM), pois é um mapa mundial livre, desenvolvido e atualizado pela comunidade do OSM em todo o mundo.

Para implementar a leitura dos mapas OSM no nosso simulador foi criado um *parser*, apesar de existir uma biblioteca já implementada, foi decidido criar uma mais vocacionada para os dados que necessitamos, logo mais simples.

Os ficheiros OSM possuem uma estrutura de dados em formato XML. Então, para criar uma leitura para mapas OSM, foi necessário fazer umas pequenas alterações no *Global_map*, modificar o construtor e criar uma função que carrega os dados do ficheiro OSM para a memória. Também foi criado um *parser* e uma estrutura de dados para ajudar a conversão dos dados do ficheiro para a memória do *Global_map*.

5.4.1 Mapas OSM

Para a integração da leitura de mapas OSM no nosso simulador, foi necessária uma análise pormenorizada da estrutura e conteúdos dos ficheiros OSM.

Os mapas *OpenStreetMap* são o resultado de um projeto que tem com objetivo fornecer a todos, de uma forma livre e gratuita, dados geográficos como mapas de estradas.

Quanto à estrutura, os *OpenStreetMap* seguem a linguagem XML, tendo assim as seguintes hierarquias (ou *tags*) na sua estrutura:

- osm
- bounds
- tag
- node
- way
- nd
- relation
- member

Para uma melhor compreensão do formato deste ficheiro segue-se um exemplo básico com distribuição do posicionamento das hierarquias entre elas:

```
 <?xml?>
<osm>
  <bounds/>
  <node>
    <tag/>
  </node>
```

```
  <way>
    <nd/>
    <tag/>
  </way>
  <relation>
    <menber/>
    <tag/>
  </relation>
</osm>
```

Cada hierarquia possui informação adicional. Seguidamente, irá ser especificada a informação adicional de cada um.

5.4.1.1 Hierarquia OSM

A hierarquia OSM é a hierarquia principal, sendo que as outras estão todas contidas dentro desta. Esta hierarquia possui como informação adicional apenas dois campos. O campo *version* onde está a versão do mapa e o campo *generator* que tem o programa que gerou a versão.

5.4.1.2 Hierarquia bounds

Esta hierarquia só pode existir uma vez, ao contrário das outras que podem existir várias vezes, não possuindo nenhuma hierarquia dentro dela. Contém como informação os limites do mapa, por isso, contém a latitude máxima do mapa, latitude mínima, longitude máxima e longitude mínima.

5.4.1.3 Hierarquia node

Esta é a hierarquia que contém a informação relativa aos nós do mapa. Por cada nó existe uma hierarquia *node,* sendo que dentro desta hierarquia podem existir hierarquias *tag* com informação adicional sobre o nó. A informação que está sempre na hierarquia *node* é:

- *Id* –identificador do nó.
- *Timestamp* – data de criação ou da ultima alteração do nó.
- *Uid* – identificador do utilizador que criou o nó.
- *User* – nome do utilizador que criou o nó.
- *Visible* – determina se é um nó visível ou não.
- *Lat* –latitude do nó.

- *Lon* – longitude do nó.

A hierarquia *node* contém a hierarquia *tag* na sua constituição, sendo que esta possui informação adicional do *node* e o seu formato de informação um pouco complexo. A hierarquia *tag* pode estar dentro de várias hierarquias, por isso tem uma forma diferente de apresentar dados. Possui o campo de informação k e *v*. O campo *k* tem o tipo de informação que é apresentada nessa *tag*, enquanto o *v* tem a resposta. Como é possível observar neste enxerto:

```
<tag k='is_in:continent' v='Europe' />
<tag k='name' v='Braga' />
<tag k='place' v='city' />
```

Esta hierarquia também possui informação que permite saber se o nó é um ponto de interesse. Neste campo é possível encontrar os seguintes tipos de pontos de interesse: *power, man made, leisure, amenity, shop, tourism, historic, landuse, military* e *natural.* Dentro de cada um destes tipos, existem vários subtipos que permitem distinguir todos os tipos de pontos de interesse existentes.

Nesta hierarquia a única informação que é necessária, de momento, para os mapas do simulador são o *id*, *lat* e *lon*.

5.4.1.4 Hierarquia way

Esta hierarquia possui a informação relativa aos nós que formam a estrada, bem como o tipo de estrada. Dentro desta hierarquia é possível observar outras duas: a hierarquia *nd* e *tag*. As hierarquias do tipo *nd* presentes, contêm os identificadores dos nós presentes nesta estrada. A hierarquia *tag* possui informação adicional sobre a estrada. A informação que está sempre na hierarquia *way* é:

- *Id* –identificador do estrada.
- *Timestamp* – data de criação ou da última alteração do estrada.
- *Uid* – identificador do utilizador que criou o estrada.
- *User* – nome do utilizador que criou o estrada.
- *Visible* – determina se é um estrada visível ou não.

A informação adicional presente na hierarquia *tag* tem o mesmo formato de apresentar os dados, referido acima. Esta hierarquia possui informação como o nome da estrada, se a estrada possui dois sentidos ou só um, para além disso possui um campo que define o tipo de estrada. Neste caso, existem vários tipos como *highway, cycleway, waterway, railway, aeroway* e *aerialway.* Dentro de cada um destes tipos, existem vários subtipos que permitem distinguir todos os tipos de caminhos existentes.

Nesta hierarquia, neste ponto de desenvolvimento do simulador, a informação que é necessária e retirada para construir os mapas do simulador, é o identificador da estrada e a informação das hierarquias *nd*, contidas dentro da hierarquia *way*, pois é a hierarquia *nd* que contém os identificadores dos nós que constituem a estrada.

5.4.1.5 Hierarquia relation

Nesta hierarquia são definidas relações entre estradas, como por exemplo, estradas que estão ligadas por rotundas, ou ligações de estradas onde não é permitido, por lei, virar. Dentro desta hierarquia também é possível observar outras duas, a hierarquia *member* e *tag*. As hierarquias do tipo *member* contêm os identificadores das estradas presentes na relação. A informação que está sempre na hierarquia *relation* é a mesma que foi observada nas outras duas hierarquias. A hierarquia *tag* possui informação adicional sobre a relação, essa informação é o tipo de relação existente. Desta hierarquia, neste momento, o simulador não necessita nenhuma informação, mas futuramente é provável que a informação nela presente seja utilizada.

5.4.2 Alterações no *Global_map*

A classe *Global_map* sofreu umas pequenas alterações para integrar os mapas OSM. Primeiro, alterou-se o construtor, este recebe o nome do ficheiro que é pretendido carregar e verifica se a sua extensão é .osm ou .wkt. Caso seja .wkt, é invocada a função que trata de carregar esse tipos de mapas, caso seja .osm, é invocada a função que permite carregar esses mapas *(OpenMapOSM(String nameFile*)). A função *OpenMapOSM(String nameFile*) recebe como parâmetro o endereço físico do ficheiro, instancia a classe *mapaPaser* e passa o endereço para esta classe. Esta classe vai guardar os dados nas classes *way* e *nodes*. A classe *way* possui as estradas com os identificadores dos nós que a estrada possui. A classe *no-*

des possui todos os nós do mapa com as suas coordenadas e os seus identificadores. Após carregar os dados todos do ficheiro, os dados são guardados no *Global_map*.

5.4.3 MapaPaser

Esta classe irá ler o ficheiro do mapa OSM e guardar os dados contidos nas hierarquias *way* e *nodes,* pois nestes estão contidas as informações necessárias para o simulador. Como já foi referido, a hierarquia *way* possui as estradas com os identificadores dos nós que a estrada possui. E a hierarquia *nodes* possui todos os nós do mapa com as suas coordenadas e os seus identificadores.

5.4.3.1 Funções

Quando o GlobalCoordinator pretende carregar o mapa OSM para a sua memória, invoca a função *makeMapaParser(String name).* Esta função lê todo o ficheiro OSM, recebendo o endereço do ficheiro. Esta função percorre todas as hierarquias do ficheiro e invoca a função de cada hierarquia para tratar os dados. As funções que tratam os dados são as seguintes:

- *Public void take_node(String rl)*– Esta função recebe a hierarquia *node*, separa os seus dados e guarda-os com o auxílio da classe *nodes*, seguidamente, guarda esta classe no *hashmap* de *nodes* para serem guardados pela classe *Global_map*.
- *Public void take_tag(String rl)* – Esta função recebe a hierarquia *bonds,* separa os seus dados, mas não os guarda. Realiza essa operação, caso posteriormente, na evolução deste simulador, esses dados sejam necessários, sendo assim mais fácil guardá-los em memória.
- *Public String take_nd(String rl)* – Esta função recebe a hierarquia *nd*, separa os seus dados, mas não os guarda. Realiza essa operação, caso posteriormente, na evolução deste simulador, esses dados sejam necessários, sendo assim muito fácil guardá-los em memória.
- *Public void take_way(String rl, ArrayList<String> node)*– Esta função recebe a hierarquia *way*, separa os seus dados e guarda-os com o auxílio da classe *way*, seguidamente, guarda esta classe no *hashmap* de *way* para serem guardados pela classe *Global_map*.

- *Public void take_member(String rl)* – Esta função recebe a hierarquia *member*, separa os seus dados, mas não os guarda. Realiza essa operação, caso posteriormente, na evolução deste simulador, esses dados sejam necessários, sendo assim muito fácil guardá-los em memória.

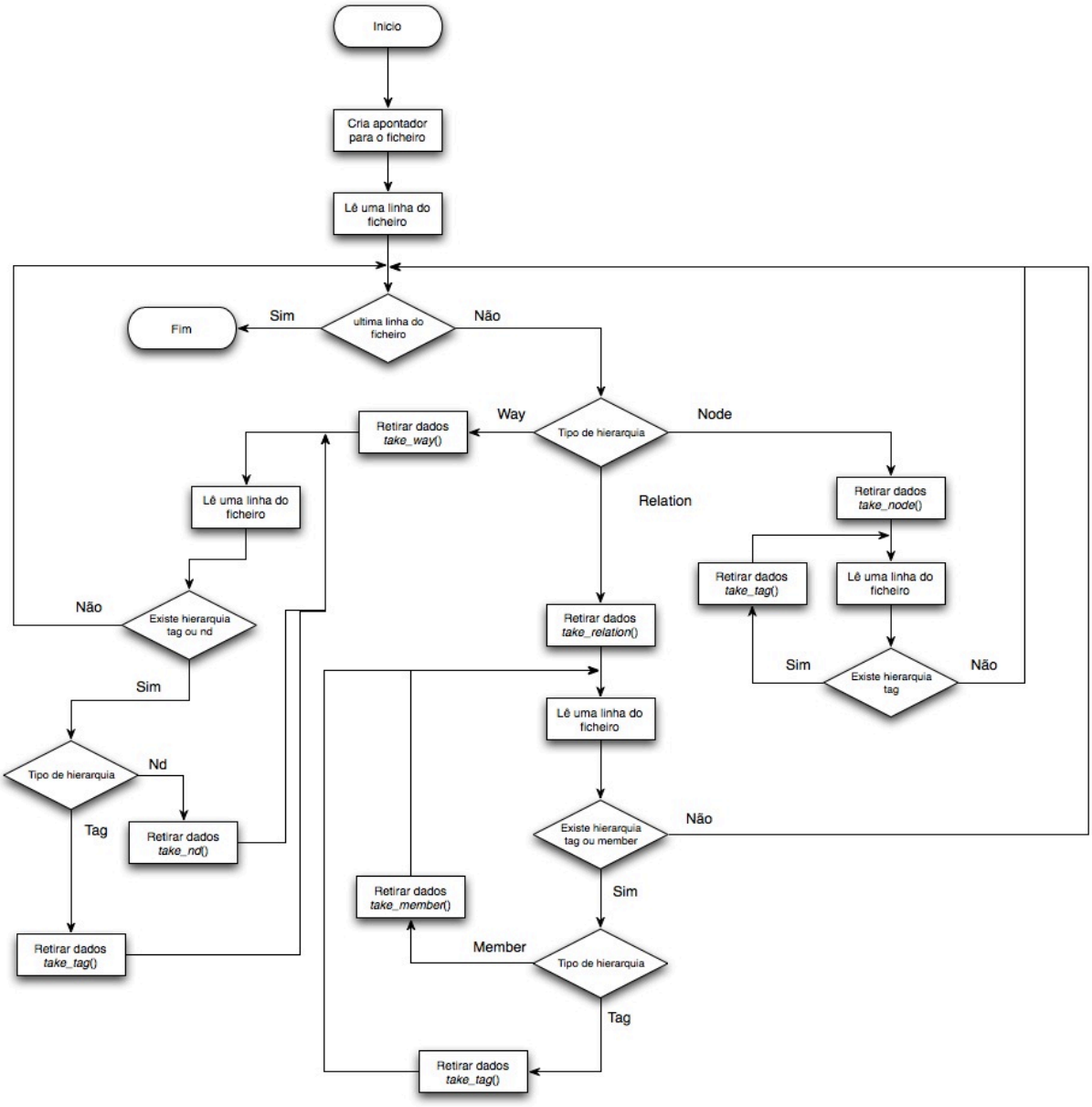


Figura 36 -Fluxograma do funcionamento da função parser

Nesta classe também existe funções *getWay()* e *getNodes()*. A função *getWay()* retorna um *hashmap* onde estão guardadas todas a estradas que o mapa contém, bem como a

informação a elas associada. A função *getNodes()* retorna um *hashmap* onde está guardado todos os nós que o mapa contém, bem como a informação a ela associada.

5.4.4 Nodes

Esta classe serve para guardar os nós presentes no mapa OSM. Esta classe tem guardadas as coordenadas do nó que contém o identificador do nó.

5.4.4.1 Funções

Esta classe, para além do construtor, só possui métodos de *get*, mais precisamente *getlat()* e *getlon*(). Esses métodos retornam a latitude e a longitude do *node*.

5.4.5 Way

Esta é uma função bastante simples, serve para guardar os nós associados a uma estrada. Por este motivo, contém a lista de nós que a estrada tem associada.

5.4.5.1 Funções

Nesta classe, para além da função construtora, só existe a função *getNodes()*. Esta função retorna a lista de nós que esta estrada possui associada.

6 Testes e Análise de resultados

Neste capítulo são apresentados os testes efetuados de forma a garantir o correto funcionamento das componentes desenvolvidas nesta dissertação para este simulador, garantindo assim o cumprimento dos objetivos inicialmente propostos.

No desenvolvimento de *software* não se pode garantir a 100% o funcionamento correto, sem a presença de erros. Por isso, no trabalho desenvolvido nesta dissertação também é impossível garantir isso, pois no desenvolvimento deste *software* existem um grande número de estados, atividades e algoritmos(Bach, 1999).

Desta forma, foram efetuados testes com o intuito de averiguar se o funcionamento das funcionalidades desenvolvidas é o mais correto. Durante a implementação desta dissertação foram feitos testes de unidade e integração, e numa fase final, foram feitos testes de sistema e de carga. Ficaram ainda por fazer testes de aceitação. Os testes de aceitação são um tipo de testes realizados por um grupo de utilizadores ao sistema final. Este grupo de utilizadores simula operações de rotina do sistema, com o objetivo de determinar se o comportamento da aplicação desenvolvida cumpre os objetivos pretendidos. Este teste não foi efetuado, pois para executar este simulador é necessário um grande número de computadores, e como já foi referido este simulador funciona num ambiente distribuído, o que não está ao alcance de todos os utilizadores.

6.1 Testes de unidade

Os testes de unidade, também denominados como testes unitários ou testes de módulos, são testes de funcionamento efetuados com unidades menores de *software* desenvolvidas. Estes testes têm o intuito de encontrar falhas de funcionamento dentro dessas unidades menores do sistema. No âmbito desta teste, inicialmente foi desenvolvida uma versão mais simples da *Visualization*, só com janela de visualização, em que nesse ponto a *Visualization*

era um processo da aplicação *GlobalCoordinator* (figura 37). Assim, nesta fase a *Visualization* usava os dados contidos na memória do *GlobalCoordinator* para criar a visualização da simulação.

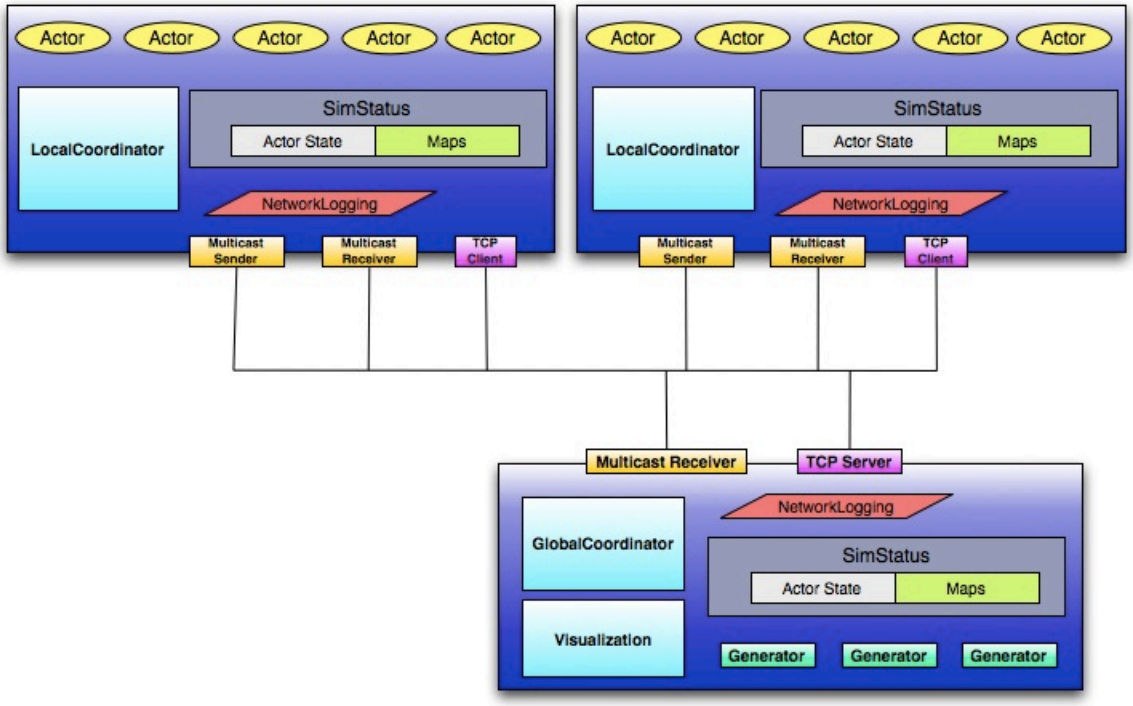


Figura 37 - Arquitetura inicial

Após esta versão estar a funcionar dentro do pretendido, foi desenvolvido outra versão, nesta versão a Visualization trabalhava como uma aplicação independente, mas ainda sem interface, só com janela de visualização. Nessa fase a *Visualization* passou a possuir comunicações e memória própria. A próxima fase foi introduzir a janela de visualização dentro da interface, neste ponto a aplicação só reproduzia simulações em tempo real. Por fim, foi adicionada a funcionalidade de leitura do *Reporting*, concluindo assim a aplicação *Visualization* apresentada nesta dissertação. Esta funcionalidade de leitura de *Reporting*, que possibilita ler simulações gravadas, inicialmente funcionava numa interface independente da reprodução da simulação em tempo real, para testar se o funcionamento era o pretendido.

Na implementação da leitura dos mapas OSM, inicialmente, este método simplesmente imprimia no ecrã os dados que era pretendido retirar do ficheiro OSM. Após garantir que os dados imprimidos eram os desejados, através da comparação dos dados impressos

com os dados contidos no ficheiro OSM, procedeu-se à passagem desses dados desejados para a memória do *GlobalCoordinator*.

6.2 Teste de integração

A fase de testes de integração, tem como objetivo detetar falhas que derivam da integração de novos componentes no sistema. Ou seja, sempre que era integrada uma nova funcionalidade, eram testadas as funcionalidades já existentes para determinar se a nova funcionalidade interferia com o funcionamento dessas funcionalidades. Estes testes foram feitos nas diferentes fases da aplicação *Visualization*, implementação do *reporting* e integração dos mapas OSM

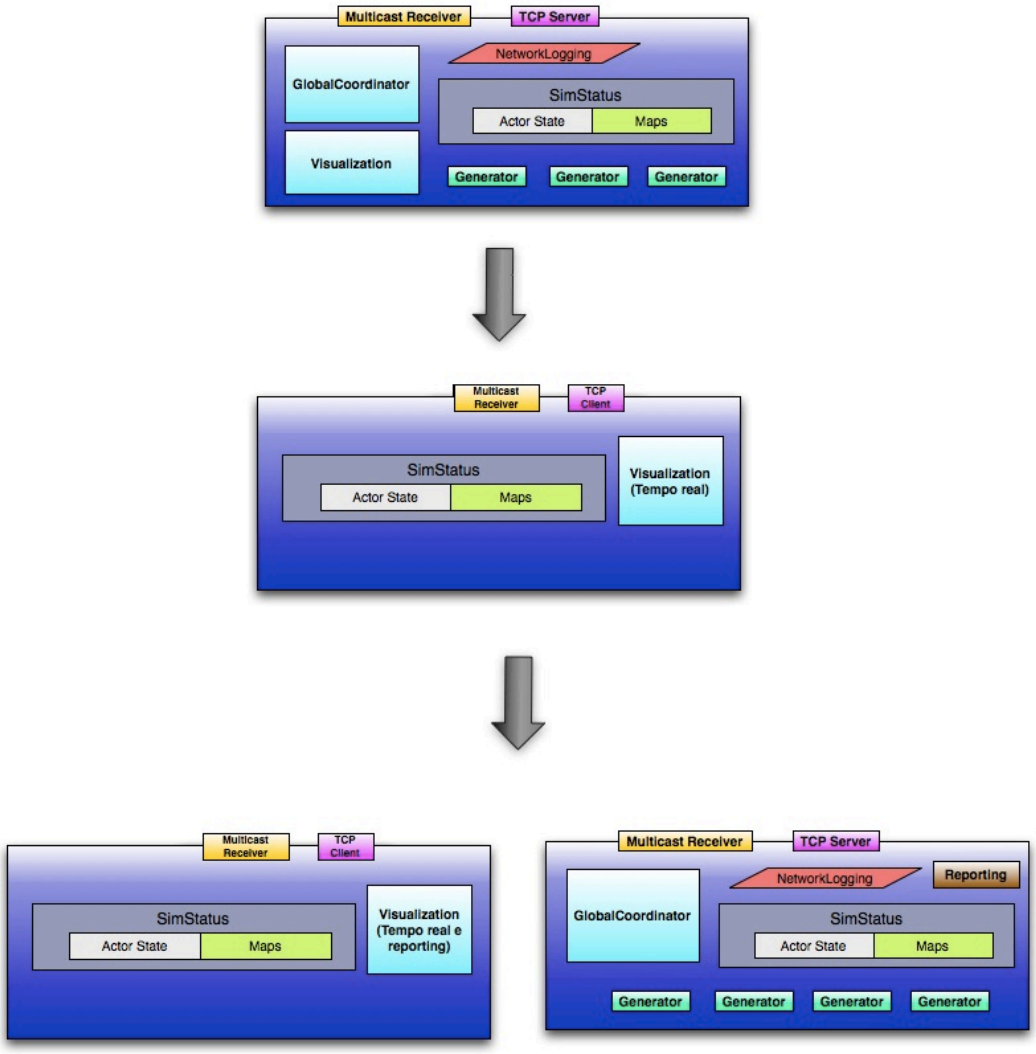


Figura 38 - Diferentes fase da aplicação Visualization

6.3 Teste de sistema

O teste de sistema foi realizado na fase final do desenvolvimento do *software* desta dissertação. Este teste tem como objetivo determinar falhas nas funcionalidades implementadas. Para isso, o sistema é executado sob o ponto de vista de seu utilizador final, testando-se todas as funcionalidades existentes.

No âmbito desta dissertação, foram testadas todas as funcionalidades das interfaces, foi comparadas simulações com o seu *reporting*, a simulação foi gravada através da captação da imagem do ecrã e comparando com a gravação feita pelo *reporting*. Também foram comparadas as projeções dos mapas OSM do simulador com projeções dos mesmos mapas feitas por um editor de mapas OSM já existente (JOSM).

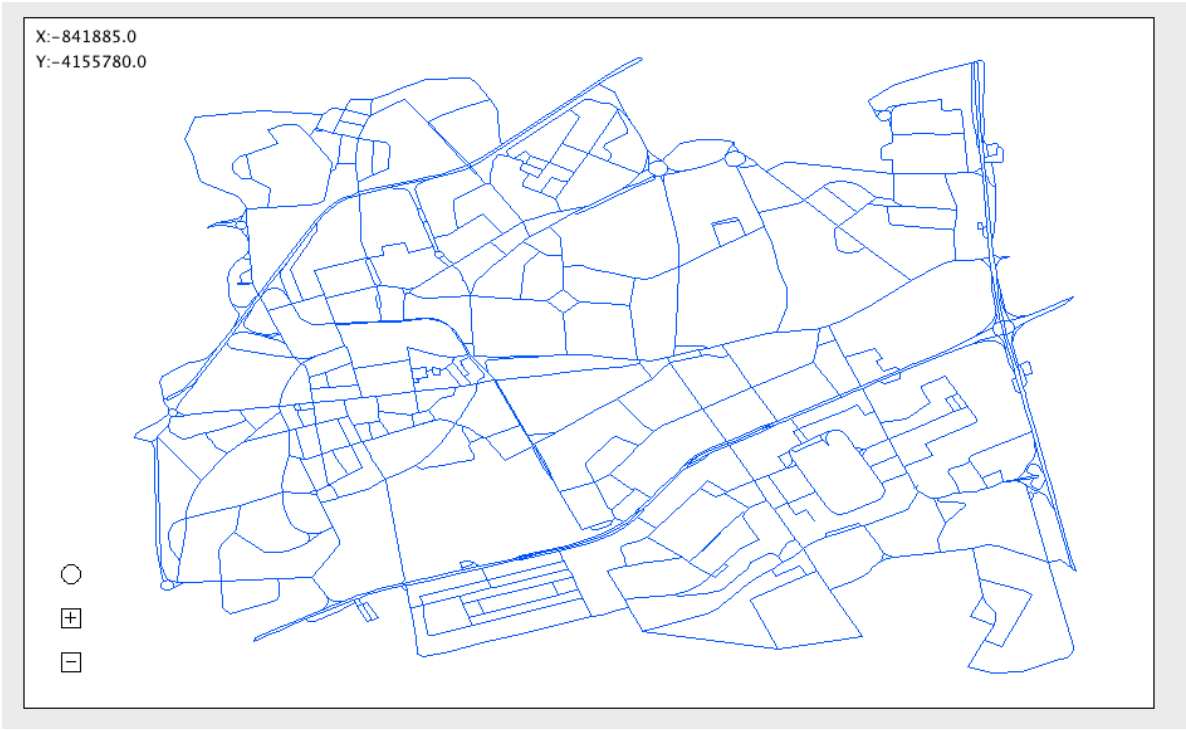


Figura 39 - Mapa OSM reproduzido pela Visualization

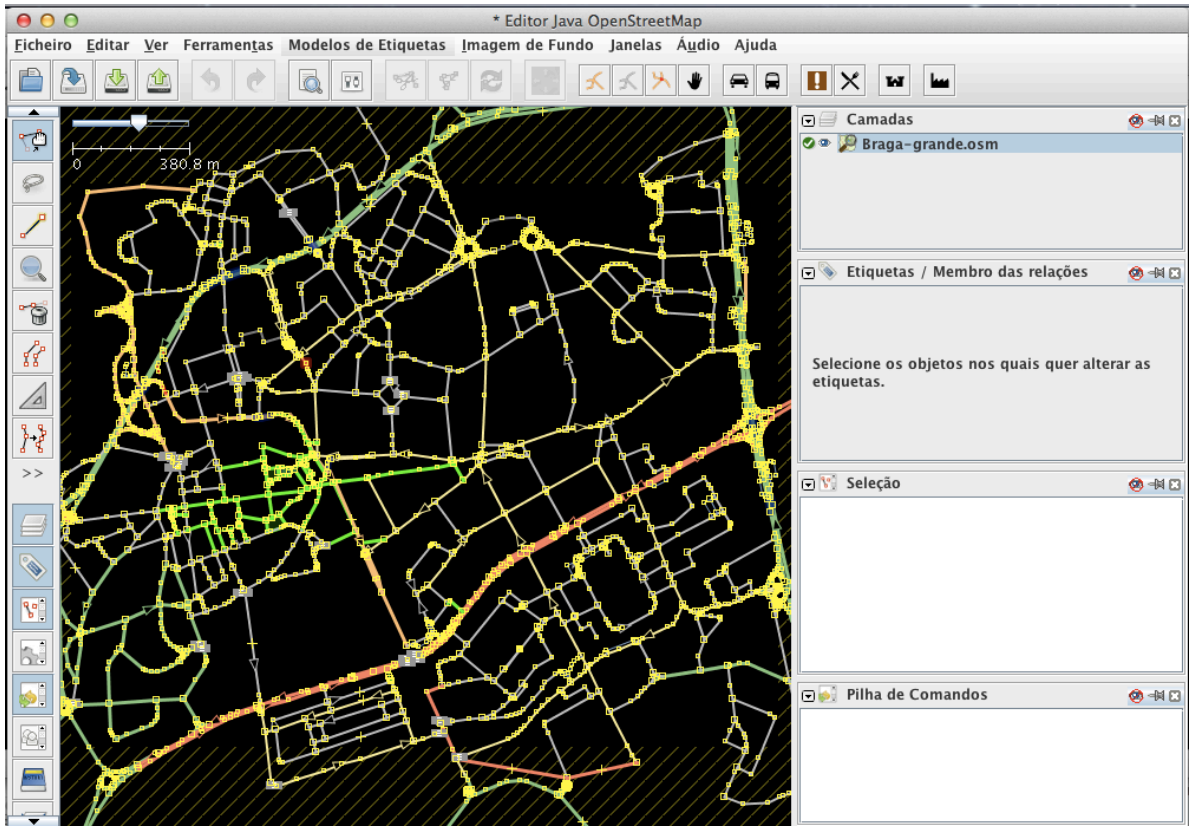


Figura 40 - Reprodução de mapa OSM pela aplicação JOSM

6.4 Testes de carga

Os testes de carga têm como objetivo determinar a carga de processamento que o computador está sujeito com as componentes desenvolvidas nesta dissertação. Para isso, foram feitos testes com o intuito de determinar a carga que a aplicação *Visualization* sujeita o computador, e foram feitos testes para determinar de que forma a funcionalidade *Reporting* aumenta a carga que a aplicação *GlobalCoordinator* produz.

Após expor os testes efetuados, é feita uma análise dos resultados obtidos.

Para a realização dos testes, foram utilizados quatro computadores diferentes, logo com componentes também eles diferentes, apresentados na tabela 1.

	Computador 1	Computador 2	Computador 3	Computador 4
Sistema Operativo	Windows 7 Home Premiun (64 bits)	Windows 7 Profes-sional (32 bits)	Windows 7 Profes-sional (32 bits)	Mac OS X versão 10.7.4 (32 bits)
Processador	Intel(R) Core(TM) i3 CPU 2367M @1,40 GHz 1,40 GHz	Intel(R) Core(TM) 2 Duo CPU T7500 @2,00 GHz 2,00 GHz	Genuine Intel(R) CPU T2130 @1,86 GHz 1,86 GHz	Intel Core 2 Duo @2,40 GHz 2,40 GHz
Memoria RAM	4 GB	1GB	1GB	2GB

Table 1 – Características principais dos computadores utilizados nos testes

6.4.1 Carga do Reporting

Foram feitos testes para determinar a carga extra que o *reporting* introduz à aplicação *GlobalCoordinator*. Para determinar essa carga foram realizadas várias simulações sem *reporting*, e várias com *reporting*. Foram comparados os resultados das cargas, relativas às simulações com diferente número de atores.Nestes teste o computador usado para executar o *GlobalCoordinator* foi o computador 1.

6.4.1.1 Resultados

Foram realizadas simulações de cada tipo (com *reporting* e sem *reporting*), nas quais foram atingidos cerca de 8000 atores por simulação, sempre com uma duração de cerca de 2h por cada simulação. Foi medida, periodicamente, a carga da máquina onde foi executada a aplicação. Seguidamente, é apresentado o gráfico de carga obtido:

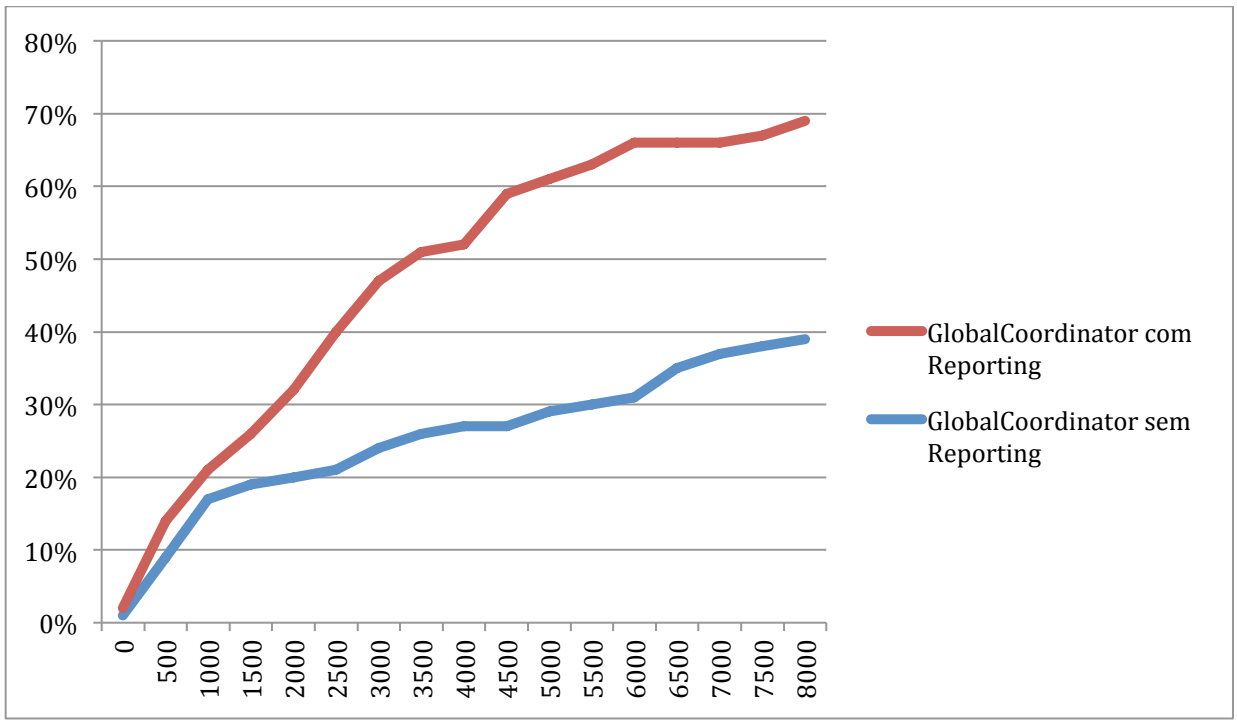


Figura 41 - Gráfico de comparação de carga com reporting ou sem reporting

6.4.1.2 Análise de resultados

Como é possível verificar, a aplicação *GlobalCoordinator* fica com um processamento mais elevado quando é executado com o *reporting* da simulação, isso deve-se ao facto do *reporting* estar ciclicamente a verificar o estado dos atores e a guardar o seu estado no ficheiro. Para além disso, o ficheiro de reporting fica com tamanho de 36GB, o que é um tamanho muito grande.

6.4.2 Carga da Visualization

Nesta secção vai ser testada a carga que a aplicação *visualization* traz ao processador de forma a definir qual é o número máximo de atores que o *visualization* consegue suportar. Para isso foram feitas simulações, algumas com todos os tipos de atores, outras só com um tipo de ator, e foi registada a carga do processador periodicamente. Desta forma, observou-se como cresce a carga que a *visualization* sujeita o processador, conforme o número de ato-

res que está a processar. Também foram feitas visualizações das gravações, para determinar a carga que este modo de visualização implica. Este teste foi realizado com o computador 1, a executar a aplicação *visualization*.

6.4.2.1 Resultados

Nas simulações feitas foi possível gerar, no máximo cerca de 8000 de atores por simulação, sempre com uma duração de cerca de 2h (figura 31). A visualização funcionou de forma satisfatória, ficando ligeiramente mais lento, a partir dos 7500 atores, quando é pretendido arrastar a imagem para visualizar outras partes do mapa.

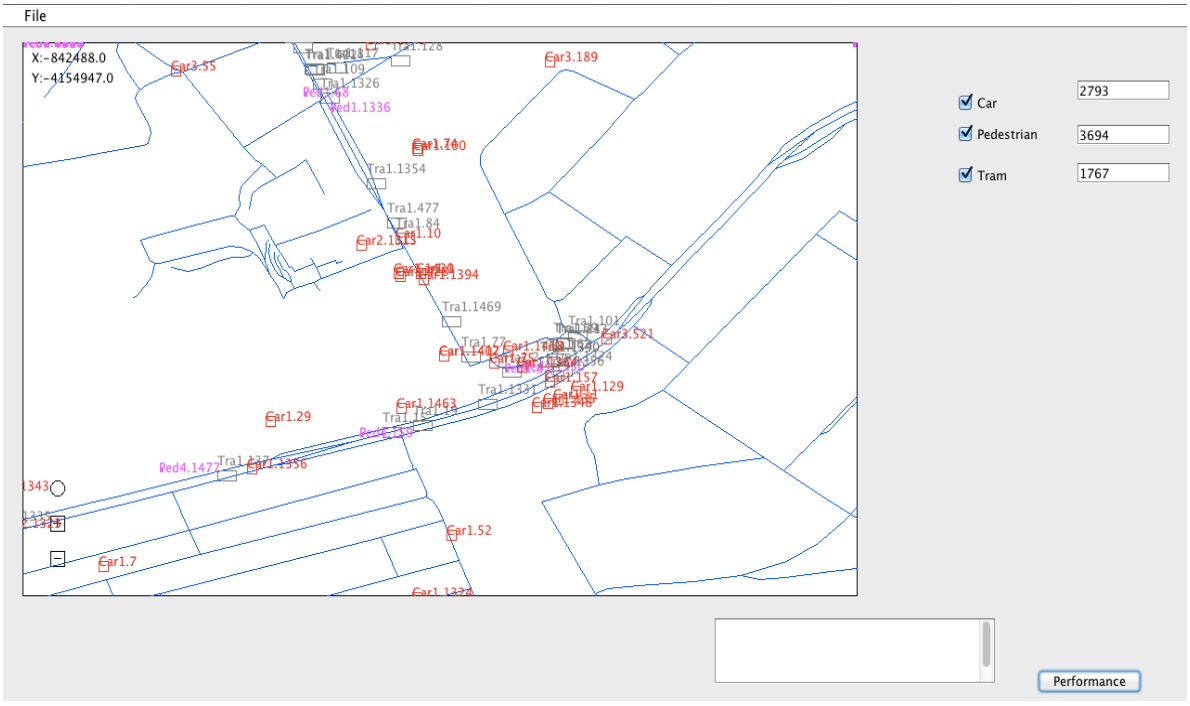


Figura 42 - PrintScreen da visualização durante a simulação

Seguidamente, será apresentado um gráfico que relaciona a carga com o número de atores com a aplicação *Visualization* a executar em modo tempo real.

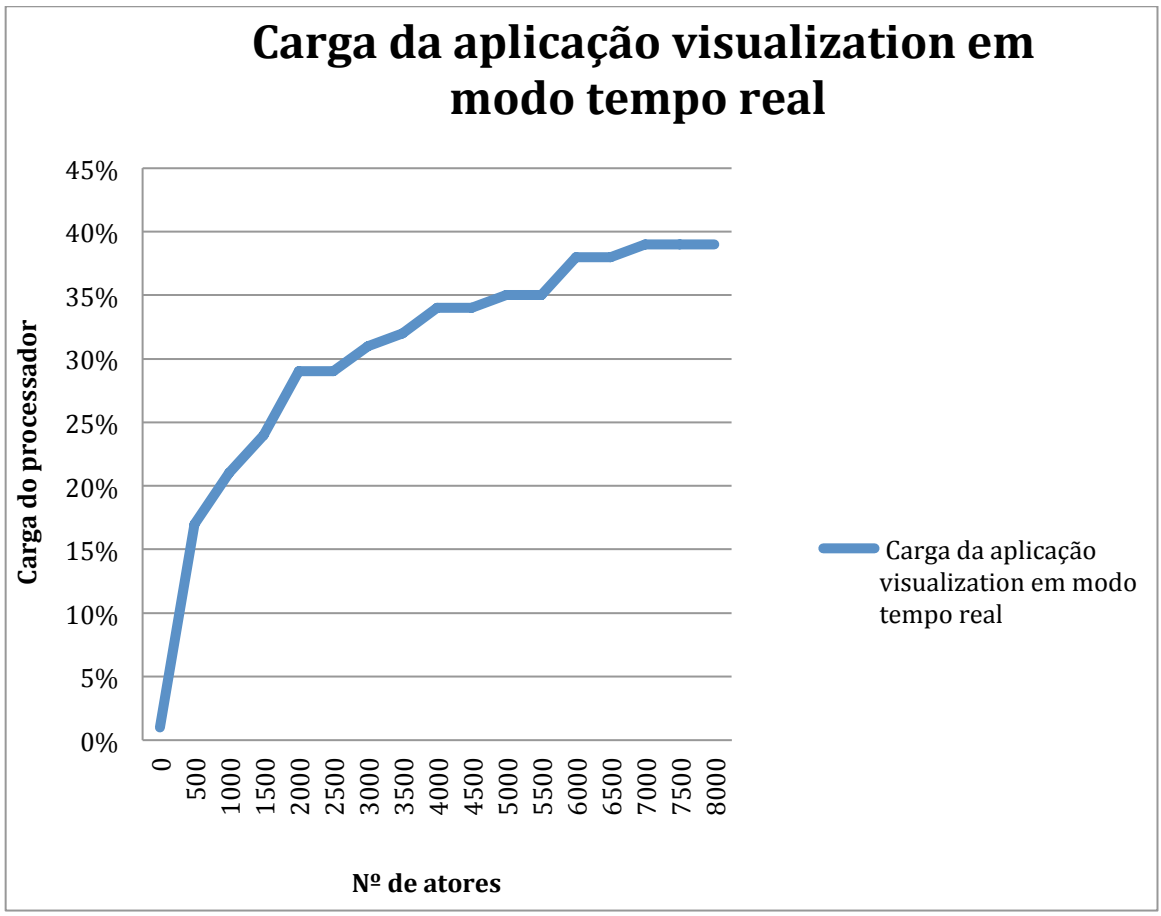


Figura 43 - Gráfico de carga da aplicação visualization em modo tempo real

Neste teste foram usados todos os tipos de atores, foram realizados outros testes com só um tipo de ator e os resultados obtidos foram sempre muito semelhantes a este.

Seguidamente, apresenta-se um gráfico que relaciona a carga com o número de atores, com a aplicação *Visualization* a executar em modo *reporting*.

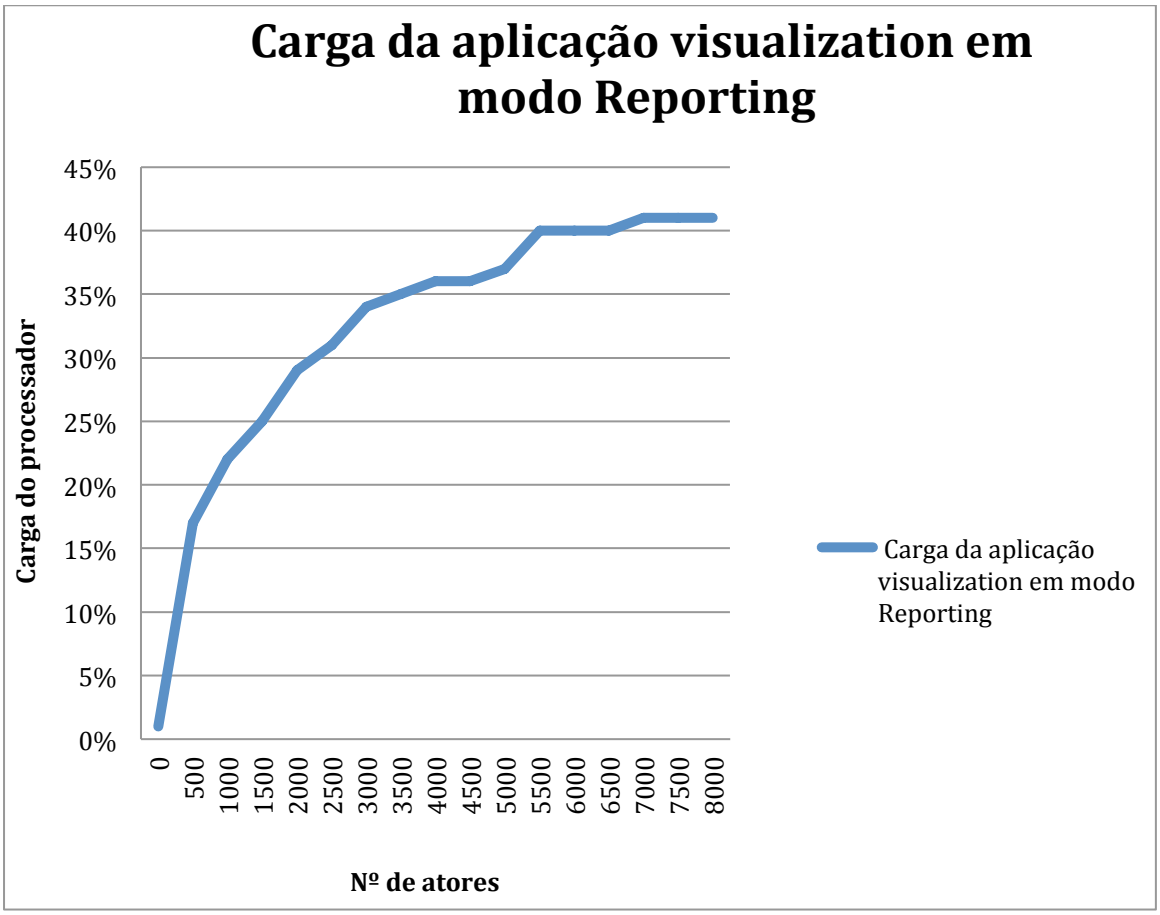


Figura 44 - Gráfico de carga da aplicação visualization em modo Reporting

6.4.2.2 Análise de resultados

Os resultados obtidos foram bastantes satisfatórios. O facto de serem necessários muitos computadores para aumentar o número de atores, impediu determinar o número máximo de atores que a aplicação *visualization* consegue suportar. Apesar desta limitação, este teste foi positivo pois foi possível verificar que a aplicação funciona com 8000 atores sem problemas. Por outro lado, os resultados também foram bastante satisfatórios, na medida em que todos os testes têm valores positivos, sendo que os valores obtidos em modo real time e reporting não apresentam diferenças significativas.

7 Conclusão

Com o trabalho realizado era pretendido desenvolver um simulador capaz de simular o ambiente vivido em cidades. O trabalho realizado é a base da visualização de um simulador que estará em desenvolvimento contínuo, com a possível inclusão de novas funcionalidades. Assim, apesar das funcionalidades desenvolvidas nesta dissertação estarem concluídas, podem ser alteradas conforme a evolução do simulador.

Como é possível verificar através do trabalho apresentado, a componente visualização, *reporting* e integração do simulador com os mapas OSM foi implementada como pretendido. Logo o objetivo deste projeto foi atingido com sucesso, tendo no entanto, como maiores imperfeições o facto dos ficheiros de *reporting* possuírem um tamanho demasiado extenso e a carga que o *Reporting* acrescenta ao GlobalCoordinator.

Como trabalho futuro, propõe-se que uma solução seja encontrada de forma a que a componente visualização seja executada fora da rede local. Esta solução poderá passar pela arquitectura proposta nesta dissertação, ou seja, a criação de um gateway que reencaminhe o tráfico para a aplicação visualization.

Na componente *reporting*, será necessário no futuro criar soluções para diminuir o tamanho dos ficheiros de gravação, o que poderá passar pelo registo da posição dos atores só quando existir uma alteração na sua posição. Assim, ocorrerá menos inserções no ficheiro, logo o tamanho do mesmo diminuirá. Por outro lado, também é possível utilizar técnicas de compressão aos ficheiros de modo a diminuir o seu tamanho, esta solução também ajudará a redução da carga imposta pelo reporting.

Como este trabalho também é importante salientar as competências que me foram possíveis desenvolver, pois com o desenvolvimento deste projeto adquiri competências ao nível do trabalho de equipa, bem como programação por objetos.

Referências

The ONE. Outubro 2010. http://www.netlab.tkk.fi/tutkimus/dtn/theone/ (accessed Outubro 2010).

Keränen, Ari, Jörg Ott, and Teemu Kärkkäinen. "The ONE Simulator for DTN Protocol Evaluation, Department of Communications and Networking." Article, Department of Communications and Networking, Helsinki University of Technology, Helsinki, 2009.

Informatik 4:BonnMotion. University of Bonn. Outubro 2012. http://net.cs.uni-bonn.de/wg/cs/applications/bonnmotion/ (accessed Outubro 2012).

Gnuplot. (Março 2012). Acedido a Outubro 2012, em http://www.gnuplot.info/

SUMO - Simulation of Urban Mobility. Outubro 2010. http://sumo.sourceforge.net/ (accessed Outubro 2010).

Open Street Map. Outubro 2011. http://www.openstreetmap.org/ (accessed Outubro 2011).
Java Graphic. Outubro 2011.

http://docs.oracle.com/javase/1.4.2/docs/api/java/awt/Graphics.html(accessed Outubro 2011).

Java Applet. Outubro 2010.
http://docs.oracle.com/javase/1.4.2/docs/api/java/applet/Applet.html(accessed Outubro 2010).

James Bach. " Risk and RequirementsBased Testing." *Proceedings of IEEE Information Communications Conference (INFOCOM 1999)*. 1999.