Faculdade de Engenharia da Universidade do Porto

FEUP

Visualização de Dados para Redes de Veículos Autónomos

Carlos José Rangel Vieira Ribeiro

VERSÃO FINAL

Dissertação realizada no âmbito do
Mestrado Integrado em Engenharia Eletrotécnica e de Computadores
Major Automação

Orientador: Prof. João Tasso de Figueiredo Borges de Sousa

30 de julho de 2012

© Carlos Ribeiro, 2012

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES | 2011/2012

A Dissertação intitulada

"Visualização de Dados para Redes de Veículos Autónomos"

foi aprovada em provas realizadas em 10/12/2012

o júri

Presidente Professor Doutor Adriano da Silva Carvalho
Professor Associado do Departamento de Engenharia Electrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

Professor Doutor Jorge Miguel Nunes Santos Cabral
Professor Auxiliar do Departamento de Sistemas de Informação da Escola de Engenharia da Universidade do Minho

Mestre João Tasso de Figueiredo Borges de Sousa
Assistente Convidado do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.

Autor - Carlos José Rangel Vieira Ribeiro

Faculdade de Engenharia da Universidade do Porto

Resumo

Vários desenvolvimentos tecnológicos recentes levaram ao aparecimento e proliferação de sistemas baseados em redes de veículos. Estes sistemas são caracterizados por estabelecerem uma colaboração entre operadores humanos e nós móveis da rede (veículos robóticos), nós esses que estabelecem caminhos de comunicação dinâmicos e, em simultâneo, dependem dos caminhos de comunicação criados para desempenhar os seus objetivos.

Este trabalho visa estender o software Neptus, desenvolvido pelo Laboratório de Sistemas e Tecnologia Subaquática (LSTS) da Faculdade de Engenharia da Universidade do Porto (FEUP), dotando-o de novas visualizações que permitam compreender de que forma estão ligados vários sistemas e subsistemas numa rede de veículos e como evoluem essas ligações ao longo do tempo.

Numa primeira fase, foi necessário fundir dados provenientes de vários veículos, transformando registos de dados individuais num log multi-sistema que permite posteriormente perceber as interações da rede.

Na segunda fase do trabalho, procedeu-se à visualização das interações através da criação de grafos de comunicações. Os grafos são gerados usando a linguagem dot (GraphViz) que permite a utilização de várias ferramentas para layout, manuseamento e exportação dos grafos como o *yEd Graph Editor* ou o *Microsoft Automatic Graph Layout*.

Por fim, estendeu-se o software Neptus com plugins que permitem gerar rapidamente grafos de comunicações a partir de logs de missão. As ferramentas criadas foram testadas em laboratório com dados reais e foram já utilizadas no terreno, durante missões reais de veículos autónomos.

Página em branco

Abstract

Recent technological advances have led to the appearance and proliferation of networked vehicle systems. These systems are characterized by allowing collaboration between human operators and mobile nodes (robotic vehicles). These mobile nodes are used as routers of information creating dynamic communication networks.

This work aims to extend the Neptus software, developed at the Underwater Systems and Technology Laboratory from Faculty of Engineering of the University of Porto, by creating new visualizations that allow the user to understand how the various nodes are connected to each other in a networked vehicle system and how existing connections evolve through execution time.

In a first phase data generated from multiple vehicles was merged, transforming the individual log files into a multi-vehicle log that can be used to study the network interactions.

On a second phase new data visualizations were created to generate different communication graphs. The generated graphs use a dot language (GraphViz) and associated tools like *yEd Graph Editor* or the *Microsoft Automatic Graph Layout*, which allow automatic graph layout, user-interactions (zooming, pan, node movement) and exporting graphs to other formats.

Finally, plugins were added to allow rapid generation of communication graphs from mission log files in the Neptus Software. The created tools have been tested in the lab with real-world data acquired at the field while operating the LSTS autonomous vehicles.

Página em branco

Agradecimentos

Com os melhores agradecimentos para o meu orientador João Tasso de Figueiredo Borges de Sousa por me ter dado a oportunidade de desenvolver este trabalho sob a sua supervisão

Também gostaria de agradecer ao doutor José Carlos de Queirós Pinto pelo tempo despendido, por todas as discussões que tivemos e pelos conselhos fornecidos, que me ajudaram a realizar um trabalho melhor.

Também gostaria de agradecer ao Daniel Filipe de Azeredo Silva pela sua contribuição e principalmente por disponibilizar o laboratório I004 e I005 sempre que possível.

Por último, mas não menos importante, gostaria de agradecer aos meus pais, ao meu irmão e amigos por todo seu apoio.

Página em branco

Índice

Capítulo 1 .... **1**

Introdução 1
1.1 - Arquitetura 3
1.2 - DUNE 4
1.3 - Neptus 5
1.4 - IMC 8
1.5 - Objectivos do trabalho 9
1.6 - Âmbito do trabalho 10
1.7 - Estrutura do documento 10

Capítulo 2 .... **11**

Estado da Arte 11
2.1 - ROS 11
2.2 - R project 13
2.3 - OpenJAUS 16
2.4 - IMC 19

Capítulo 3 .... **31**

Descrição do Problema 31
3.1 - Intercalar dados 31
3.2 - Criar grafo de comunicações 32
3.3 - Criar ferramentas de visualização 33

Capítulo 4 .... **35**

Abordagem do Problema 35
4.1 - Intercalar dados 35
4.2 - Criar grafo de comunicações 37
4.3 - Criar ferramentas de visualização 38

Capítulo 5 .... **41**

Resultados 41
5.1 - Intercalar dados 41
5.2 - Grafo de comunicações 43
5.3 - Ferramentas de visualização 45

Capítulo 6 .... **57**

Conclusão 57
6.1 – Visualização de dados numa rede de veículos 57

6.2 – Trabalho futuro 58

Referências 59

Lista de figuras

Figura 1 - Conceito da rede do LSTS para sistemas de veículos [1]. 2

Figura 2 - Diferentes camadas da arquitetura do controlo dos veículos [12]. 3

Figura 3 - Implementação da arquitetura e possível troca entre controladores ativos/inativos [1]. 4

Figura 4 - API do neptus. 6

Figura 5 - Consola operacional do NEPTUS [19]. 7

Figura 6 - Neptus MRA. 8

Figura 7 - Exemplo da estrutura de uma mensagem IMC [1]. 9

Figura 8 – *rxgraph* [20]. 13

Figura 9 - *rxconsole* [20]. 13

Figura 10 - *Screenshot* do R *project* num *MacOS X RAqua desktop* [32]. 14

Figura 11 - Sistema JAUS feito de subsistemas ligados em rede [43]. 17

Figura 12 - Alguns veículos que usam IMC [11]. 20

Figura 13 - Mensagens IMC no AUV *Seascout Light*[11]. 21

Figura 14 - Conceito de transferência de mensagens e implementação de *tasks* do DUNE [1]. 21

Figura 15 - Exemplo de *serialization* IMC-XML [12]. 23

Figura 16 - Formato do pacote [53]. 23

Figura 17 - Diagrama de classes. 39

Figura 18 - Grafo de comunicações para as mensagens *Announce*. 43

Figura 19 - Grafo de comunicações de todas as mensagens do log. 44

Figura 20 - Resultado obtido no Neptus, anteriormente à implementação dos highlights. 45

Figura 21 - Resultado obtido no Neptus, com highlights. 46

Figura 22 - Grafo de comunicações editado no *gephi*. 47

Figura 23 - Recorte da figura 21. 48

Figura 24 - Grafo de comunicações para todas as mensagens, após implementação dos *highlights*. 49

Figura 25 - Grafo de comunicações para as mensagens *Voltage*. 50

Figura 26 - Grafo de comunicações para as mensagens *Heartbeat*. 51

Figura 27 - Grafo de comunicações *EntityList*, com legenda. 51

Figura 28 - Grafo de comunicações *Current*, com legenda. 52

Figura 29 - Análise no MRA dos *TransportBindings*. 53

Figura 30 - Grafo de comunicações EntityList final. 53

Figura 31 - Grafo de comunicações da rede no Neptus MRA. 54

Figura 32 - Grafo de comunicações da rede a partir da classe *IMCgraph*. 55

Lista de tabelas

Tabela 1 - Alguns serviços do JSS Core [43].. 19

Tabela 2 - Lista de tipos de campos válidos [53]... 22

Tabela 3 - Tipo de *serialization* [53].. 22

Tabela 4 - Header do pacote [53]. 24

Tabela 5 - Footer do pacote [53]. 24

Tabela 6 - Grupo lógico de mensagens [53]... 25

Tabela 7 - Mensagem convertida de xml para pdf [53]. 26

Tabela 8 - Lista completa de mensagens [53]. 26

Abreviaturas

API	*Application Programming Interface*
ASV	*Autonomous Surface Vehicle*
AUV	*Autonomous Underwater Vehicle*
BSD	*Berkeley Software Distribution*
CCL	*Compact Control Language*
CCU	*Command and Control Unit*
CORBA	*Common Object Request Broker Architecture*
CPU	*Central Processing Unit*
CRAN	*Comprehensive R Archive Network*
DCOM	*Distributed Component Object Model*
DEEC	Departamento de Engenharia Eletrotécnica e de Computadores
DUNE	*Unified Navigational Environment*
ETH	*Swiss Federal Institute of Technology Zurich*
FEUP	Faculdade de Engenharia da Universidade do Porto
GAM	*Generalized additive model*
GPL	*General Public License*
GUI	*Graphical User Interface*
HIL	*Hardware-in-the-loop*
IDE	*Integrated Development Environment*
IDL	*Interface Definition Language*
IEEE	*Institute of Electrical and Electronics Engineers*
IMC	*Inter-Module Communication Protocol*
JAUS	*Joint Architecture for Unmanned Systems*
JSIDL	*JAUS Service Interface Definition Language*
JSON	*JavaScript Object Notation*
JSS Core	*JAUS Core Service Set*
JVM	*Java Virtual Machine*
KML	*Keyhole Markup Language*
LLF	*LSTS Log Format*

LSTS	*Laboratório de Sistemas e Tecnologia Subaquática*
LVQ	*Learning Vector Quantization*
MRA	*Mission Review and Analysis*
NATO	*North Atlantic Treaty Organization*
netCDF	*network Common Data Form*
PDF	*Portable Document Format*
ROS	*Robot Operating System*
ROV	*Remotely Operated Vehicle*
RPC	*Remote Procedure Call*
SAE	*Society of Automotive Engineers*
SDK	*Software Development Kit*
SNAME	*Society of Naval Architects and Marine Engineers*
SOA	*Service Oriented Architecture*
STAIR	*STanford AI Robot*
SVN	*Subversion*
TCP	*Transmission Control Protocol*
UAV	*Unmanned Air Vehicle*
UDT	*User Datagram Protocol*
URL	*Uniform Resource Locator*
USTL	*Underwater Systems and Technology Laboratory*
XML	*Extensible Markup Language*
XSD	*XML schema*
XSLT	*Extensible Stylesheet Language Transformations*

Capítulo 1

Introdução

Este trabalho desenvolve uma aplicação a integrar no *software* de controlo neptus, o qual foi desenvolvido pelo LSTS da FEUP. Com esta aplicação torna-se possível visualizar, através de grafos, todas as comunicações realizadas por uma rede de veículos ou, em alternativa, apenas aquelas pretendidas pelo operador. Como tal, permite ao leitor uma melhor compreensão dos problemas e das soluções encontradas para esses veículos, bem como fornece alguns conhecimentos acerca da implementação e do controlo da rede de veículos desenvolvida pelo LSTS. Por esta razão, nesta introdução seguirei de perto o artigo "*Implementation of a Control Architecture for Networked Vehicle Systems*" [1].

O LSTS visa a criação e o desenvolvimento de veículos autónomos e assistidos por operadores humanos e de redes de sensores [2] (figura 1). As redes formadas por estes sistemas são dinâmicas e como os veículos têm alcance de comunicação limitado e estão constantemente a mover-se, são criados ou eliminados *links* de comunicação e de controlo em tempo de execução. Os nós da rede podem funcionar como sensores, dispositivos móveis de comunicação ou mesmo trabalhar como mulas de informação (transportando dados) e recuperação de redes de comunicação [1].

O LSTS já construiu vários veículos autónomos diferentes, nomeadamente: Veículos Operados Remotamente (ROV) em 2005 [3], veículos autónomos de superfície (ASV) em 2007 [4], veículos submarinos autónomos [5] desde 2005 e vários veículos aéreos não tripulados [6] desde 2007. Para controlar estes veículos, a equipa do LSTS criou *software* e protocolos de comunicação que podem ser reutilizados para vários tipos de veículos, de acordo com necessidades específicas a cada veículo, e que podem realizar comportamentos simples ou cooperativos.

Para suportar redes deste tipo, foi necessário gerir sensores e atuadores a bordo dos veículos e a sua utilização para controlo e navegação autónoma e para comunicação entre veículos, *gateways* e consolas de operadores. Foram desenvolvidas normas de comunicação para planeamento, seguimento e análise das missões, bem como interfaces adaptativas. Todo o *software* foi desenvolvido tendo em vista a sua extensibilidade de forma a oferecer flexibilidade para adicionar novos dispositivos, controladores, visualizações e eventuais soluções de problemas futuros [1]. Numa tentativa de satisfazer essas necessidades foram criadas varias ferramentas de *software* que juntas são denominadas como a *LSTS Toolchain*

(ferramentas que, unidas, implementam uma arquitetura de controlo para múltiplos veículos). A LSTS Toolchain compreende o *DUNE Unified Navigational Environment* (DUNE), que é o *software* de bordo, o Neptus, que é o *software* de comando e controlo, e o protocolo de comunicações *Inter-Module Communication* (IMC) [2]. Todas estas ferramentas vão permitir a criação de *logs* que são posteriormente analisados.

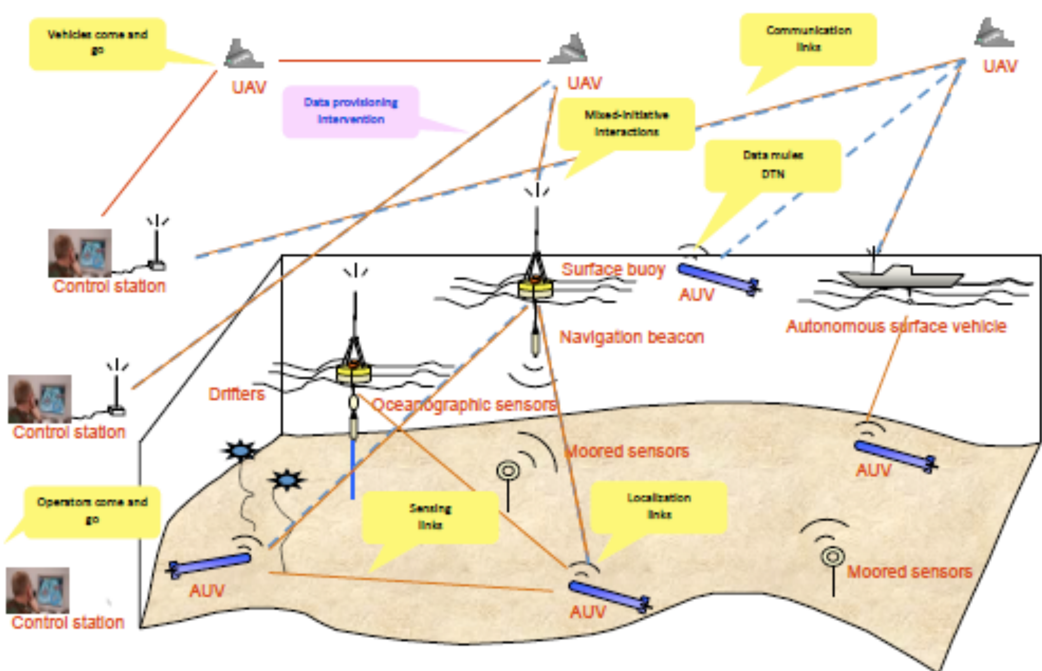


Figura 1 - Conceito da rede do LSTS para sistemas de veículos [1].

Existem vários projetos de protocolos de comando e controlo similares ao IMC que permitem controlo e comandos de baixo nível, como é o caso do JAUS/SAE AS-4 [7] e da *Compact Control Language* (CCL) [8] que utilizam mensagens para controlar o veículo. O IMC possui um conjunto de mensagens de alto nível que fornecem características idênticas às do NATO STANAG 4586 [9] e *Common Control Language* (*common* CL), os quais permitem o envio de mensagens genéricas e a monitorização do veículo em tempo real. O CCL e *common* CL [10] são projetados para submarinos autónomos (*Autonomous Underwater Vehicles* ou, simplesmente AUVs) tendo em consideração as comunicações subaquáticas acústicas, pelo que neste protocolo todas as mensagens são projetadas para caber em pacotes altamente compactos e assim otimizar as comunicações. O IMC, por sua vez, fornece vários formatos de serialização, o que viabiliza a comunicação para larguras de banda reduzidas, sendo comum usar um formato binário eficiente. O NATO STANAG 4586 é utilizado para definir padrões que podem ser usados para controlar veículos aéreos não tripulados (*Unmanned Air Vehicles* ou, simplesmente UAVs) com características distintas e, dada a possibilidade de operarem em conjunto, foram desenvolvidos protocolos para comandar qualquer UAV. O IMC também foi desenvolvido com o objetivo de obter interoperabilidade e abstração de *hardware*. O *Joint Architecture for Unmanned Systems* (JAUS) é semelhante ao IMC, permitindo a existência de uma vasta gama de veículos em que é utilizada uma vista hierárquica de veículos, como sistemas com subsistemas, nódulos, componentes e instâncias de componentes [11]. Quando

interrogados sobre o porquê de desenvolver um novo protocolo, os investigadores do LSTS indicam que o seu objetivo é desenvolver um novo protocolo não para um tipo de veículo específico mas sim para redes de veículos heterogéneos, tendo também interesse em adaptar o protocolo de acordo com as necessidades que vão sendo encontradas.

1.1 - Arquitetura

Na arquitetura de controlo do sistema dos veículos em rede é utilizada uma abordagem de controlo por camadas, uma vez que existem vários nós diferentes na rede. Estes são compostos por vários componentes, como por exemplo veículos (figura 2), sensores, controladores (figura 3), operadores humanos, consolas de operação, dispositivos de comunicação, etc., de modo a estabelecer interfaces comuns para a comunicação e coordenação entre os componentes. Cada camada encapsula detalhes do nível inferior e fornece interfaces para leitura de estado e receção de comandos.

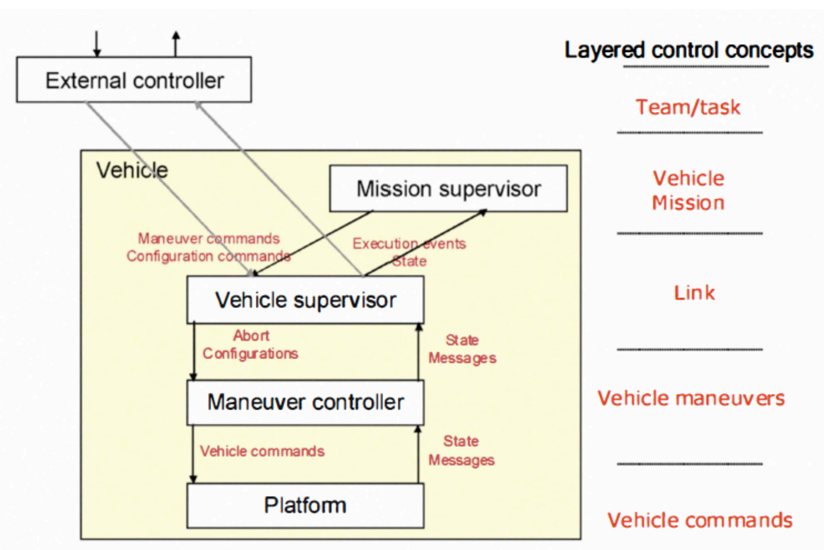


Figura 2 - Diferentes camadas da arquitetura do controlo dos veículos [12].

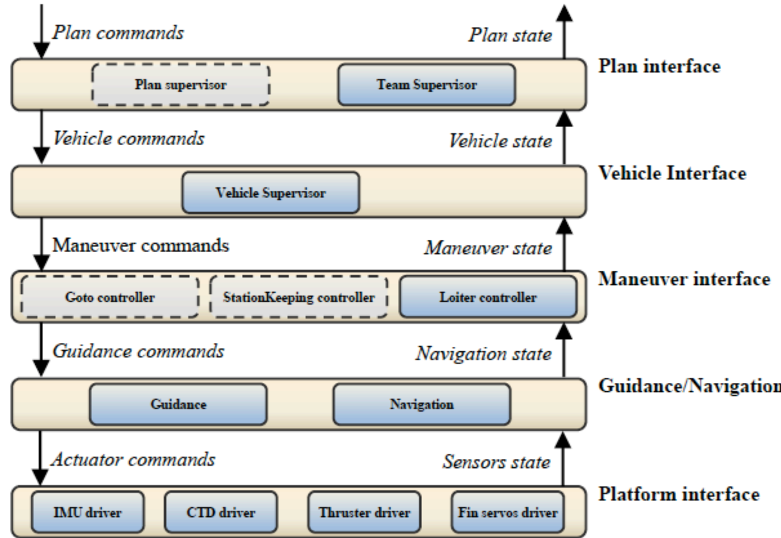


Figura 3 - Implementação da arquitetura e possível troca entre controladores ativos/inativos [1].

Todos os veículos têm uma plataforma de baixo nível com sensores e atuadores que são utilizados pela camada *Guidance/Navigation*, à qual compete a leitura e o comando dos atuadores. Na camada seguinte temos a *Maneuver interface*, que vai ler a informação do estado do veículo e vai enviar comandos para a camada de *Guidance/Navigation*. Em cima da camada *Guidance/Navigation* temos a *Vehicle Interface* que vai supervisionar o veículo, isto é, vai verificar se o sistema está a funcionar corretamente e inicia ou termina os controladores de manobra de acordo com as especificações solicitadas pela camada superior. O *Vehicle Supervisor* (integrado na camada *Vehicle Interface*), por sua vez, verifica se a manobra é possível fisicamente e se é seguro executá-la, de acordo com a informação recebida das camadas inferiores (níveis de bateria, falhas de *hardware*, etc.) e também pode terminar a execução da manobra em caso de falha de *hardware* ou qualquer violação de segurança.

A camada superior envia comandos de especificação de manobras para o *Vehicle Supervisor*. Estes comandos podem provir do *Team Supervisor*, o qual comanda a execução de manobras em vários veículos, ou do *Plan Supervisor*, o qual se encontra a bordo dos veículos e, de acordo com uma especificação da missão, dispara a execução de manobras no veículo. Exceto para a camada da plataforma de *hardware*, todas as outras camadas seguem interfaces comuns; este fato permite ter várias instâncias de controladores na camada superior, o que por sua vez proporciona uma maior flexibilidade.

As camadas *vehicle*, *maneuver*, *guidance*, *navigation* e *Platform Interface* foram implementadas pela *framework* do DUNE.

1.2 - DUNE

O DUNE é o *software* de bordo usado nos veículos e é responsável pelas comunicações, supervisão de navegação, controlo, sensores e atuadores. É escrito em C++ e tem uma arquitetura multiplataforma que permite diferentes arquiteturas de CPUs (Intel x86 ou compatível, Sun SPARC, ARM, PowerPC e MIPS) e sistemas operativos (Linux, Solaris, Mac OS X, FreeBSD, NetBSD, OpenBSD, eCos, RTEM, Microsoft Windows ou superior e QNX Neutrino).

Este fato permite a utilização do DUNE em barcos autónomos (*Autonomous Surface Vehicles* ou, simplesmente *ASVs*), ROVs, AUVs e UAVs e nas *gateways* de comunicação [13]. A sua adaptação a diferentes sistemas é feita através da alteração de perfis de execução e configurações, podendo existir tarefas comuns aos vários veículos, mudando apenas as suas configurações.

Os conjuntos de parâmetros de configuração de uma tarefa são determinados pelo esquema de configuração que permite a ativação e desativação de tarefas sem haver necessidade de recompilar o *software*. O DUNE também pode ser configurado com perfis diferentes, que permitem ativar e desativar conjuntos de tarefas predefinidos no perfil.

A versatilidade e modularidade do DUNE vai permitir um nível de abstração em que, para instalar um novo sensor ou um novo controlador, apenas é necessário habilitar ou desabilitar algumas tarefas, o que permite criar facilmente novo *software* para um módulo.

O DUNE pode ser executado num perfil de simulação, o qual simula todos os sensores e atuadores, ou no perfil *Hardware-in-the-loop (HIL) Simulation*, o que permite ativar alguns sensores e atuadores bem como a simulação de tarefas, possibilitando testar e validar novas tarefas e funcionalidades.

O DUNE funciona como um mecanismo de passagem de mensagens onde tarefas independentes são executadas em diferentes *threads* e onde as tarefas são ligadas a um canal de mensagens (*bus*) que permite publicar e assinar mensagens e através da qual podem ser consumidas ou publicadas por outras tarefas.

As mensagens passadas para o *bus* são especificadas no protocolo de comunicações do LSTS, o IMC [11], sendo que o DUNE procede à criação de *logs* de todas as mensagens recebidas por uma tarefa especial de registo de dados (*Transports.Logging)* que concatena todas as mensagens recebidas num ficheiro em disco.

1.3 - Neptus

Neptus é o *software* de comando e controlo usado por operadores humanos para interagir com sistemas de redes de veículos e suporta diferentes fases do ciclo de vida de uma missão: planeamento, simulação, execução, revisão e disseminação [14–18]. Este fornece uma interface que permite o controlo individual ou em simultâneo de vários veículos, como AUVs, UAVs, ASVs e ROVs.

No Neptus, uma missão é visualizada graficamente e especificada a partir de um conjunto de manobras e transições entre manobras. Uma manobra pode ser realizada por um veículo específico ou uma classe de veículos resultando na instanciação de um controlador a bordo do veículo que potencialmente altera o seu estado físico. As transições usam uma expressão booleana que pode ser desencadeada por eventos assíncronos ou avaliando o estado (interno) do veículo.

O Neptus tem uma *Application Programming Interface* (API) que fornece um mecanismo de modelos (*templates*) que gera planos de missão com base em parâmetros introduzidos pelo utilizador, sendo possível adicionar novos *templates* através de plugins em *javascript* (figura 4). Este *software* também permite efetuar o planeamento da missão visualmente, possuindo para o efeito um editor gráfico que fornece uma visualização do mapa do local da missão, e as manobras podem ser adicionadas e editadas a este mapa, também permite editar as ligações entre manobras e os parâmetros das manobras. Para verificar um plano de missão, geralmente os operadores pré-visualizam a sua execução usando um simulador.

O neptus oferece três níveis diferentes de simulação:

- Previsão rude do comportamento.
- Simulação de *software*.
- Simulação de HIL

Para efetuar a simulação é necessário simular um ou mais veículos no DUNE em modo de simulação para que seja possível comunicar com o Neptus, não nos podemos esquecer também podem ser executados no interior dos veículos reais para testar sensores e atuadores.

A simulação de Software HIL é utilizada para testar as especificações da missão, e para treinar os elementos da equipa antes das missões reais; a simulação HIL também pode ser usada para testar *hardware* em *dry-run*. O Neptus também fornece simulações de comportamento numa missão, quando os veículos perdem a ligação à base, para prever o estado dos veículos e os locais onde estes se possam encontrar, de forma a facilitar a vida ao operador em missões com múltiplos veículos.

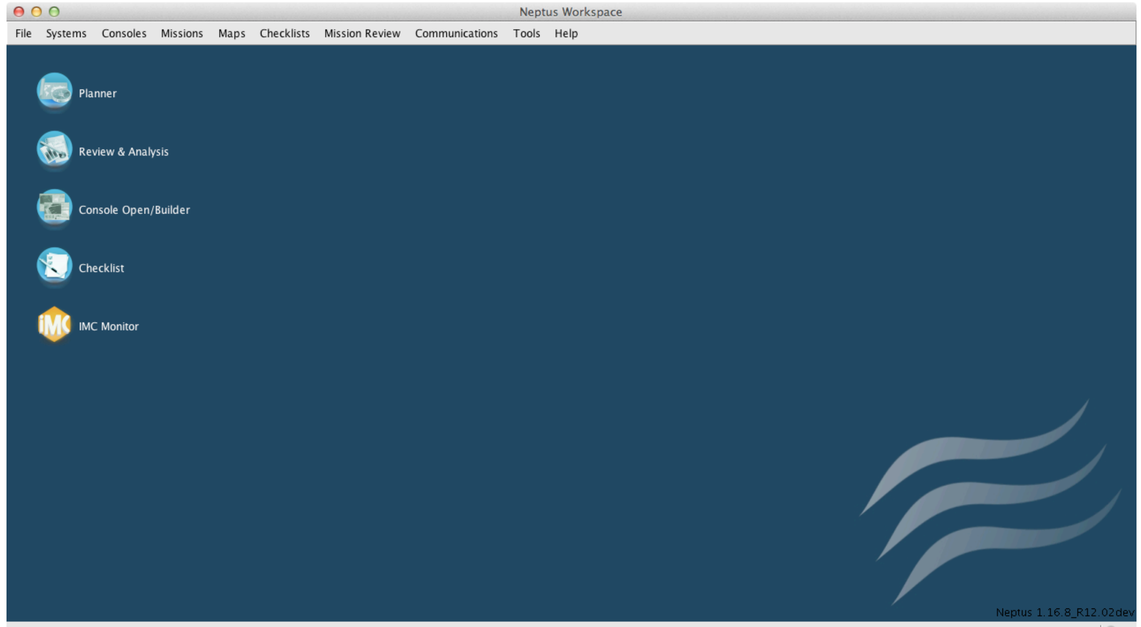


Figura 4 - Ambiente de trabalho do neptus.

Em missões são utilizadas as consolas operacionais do Neptus (figura 5), as quais permitem monitorizar a execução do veículo, alterar ou criar novos planos, enviar planos de execução, operar remotamente os veículos, etc. Estas consolas operacionais suportam o controlo de múltiplos veículos em simultâneo, mostrando dados recebidos de todos os veículos, e permitem ao utilizador alternar entre veículos controlados. Como os veículos estão maioritariamente num modo de funcionamento autónomo, torna-se possível planear missões futuras enquanto outras estão a ser executadas. Para proporcionar uma operação mais segura, o Neptus fornece uma *framework* de alarmes composta por vários *daemons* que monitorizam os dados a serem recebidos pelos veículos, notificando o utilizador de eventos de interesse como avarias, o início e a conclusão das missões.

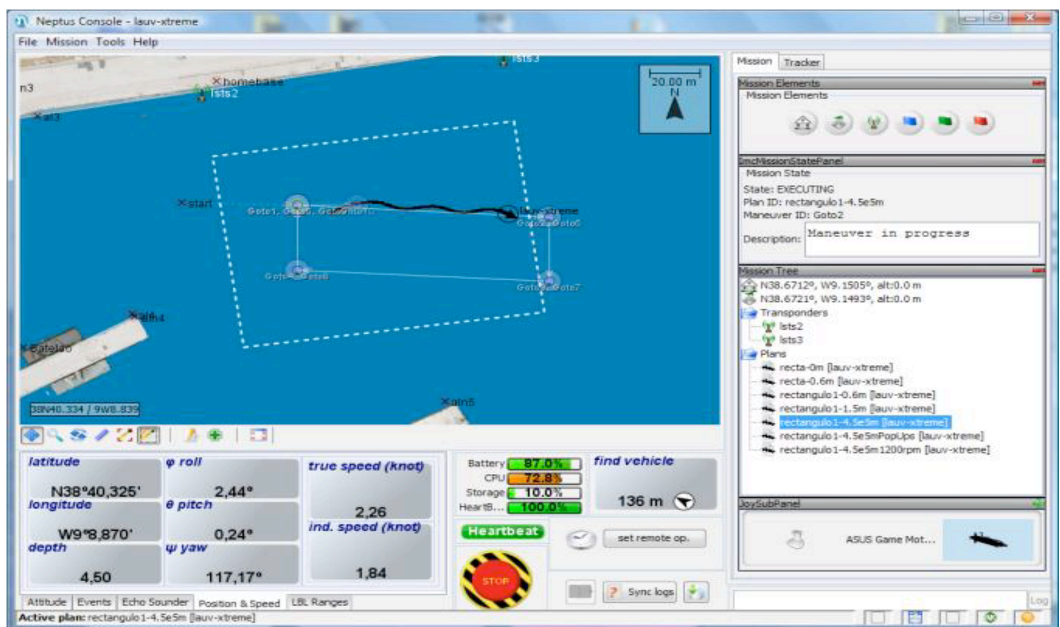


Figura 5 - Consola operacional do NEPTUS [19].

Os veículos do LSTS armazenam os dados das mensagens geradas e recebidas nas missões em *serialized streams*. Para que seja possível inspecionar e analisar esses dados posteriormente, o Neptus fornece uma aplicação que descompacta os dados em arquivos de texto, bem como várias visualizações e utilitários para processar os dados e *plugins* de revisão *logs* e mapas de cores; os arquivos de texto podem ser exportados para ferramentas como *Excel* e o *Matlab*. Todas as mensagens têm uma *tag* com o tempo em que foram geradas pelo sistema. O neptus *Mission Review and Analysis* (MRA) permite a reprodução dos dados da missão (figura 6), recriando uma visualização da execução dos veículos em missões passadas e enviando os dados para as consolas operacionais. Os *plugins* do Neptus MRA vão viabilizar a exportação dos dados das missões para vários formatos, como PDF, NetCDF e KML.

O Neptus tem algumas semelhanças com o conjunto de ferramentas do ROS [20]. Enquanto tentam satisfazer requisitos semelhantes, cada um segue a sua filosofia, o Neptus fornece interfaces configuráveis que podem ser adaptadas para cada tipo de veículo autónomo, enquanto que o ROS tem uma única interface. O Neptus foi testado várias vezes no campo, tendo *feedback* da comunidade académica, industrial e militar. O DUNE é executado num espaço muito pequeno (16 MB) e foi desenvolvido para processadores embutidos, de capacidade limitada, podendo ser executado em sistemas operacionais que não possuam processos, tais como o RTEMS ou ECOS. Por outro lado, o ROS é *open-source* e tem uma comunidade que contribui, ajudando a expandir o conjunto de ferramentas.

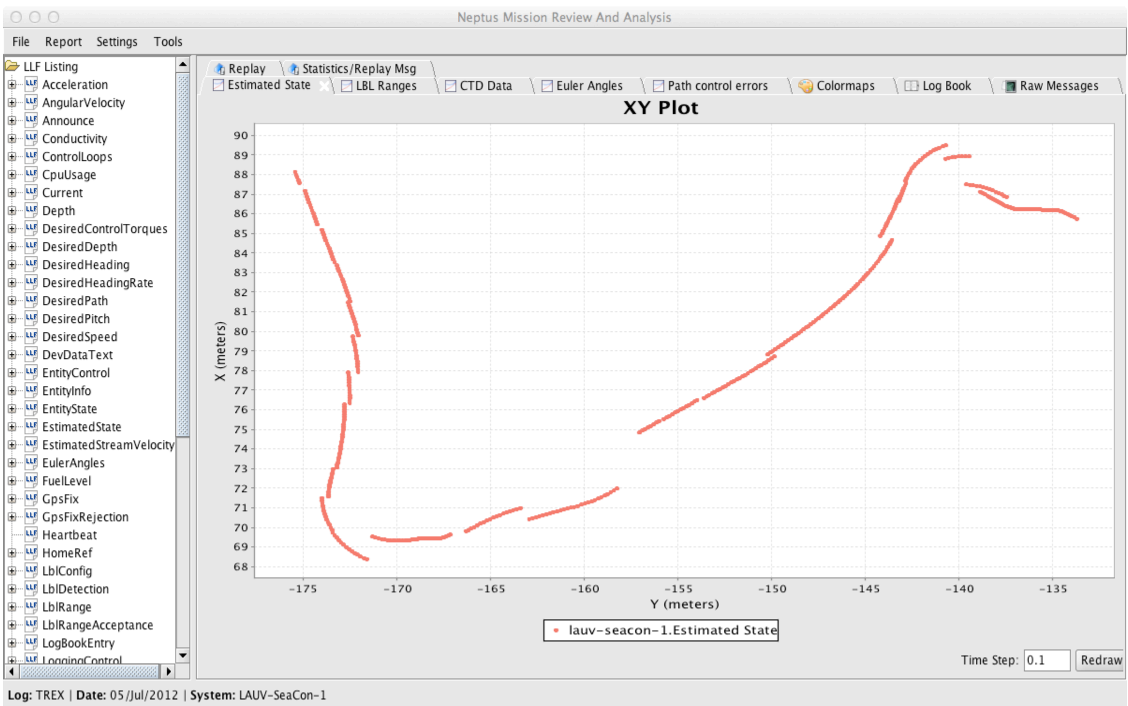


Figura 6 - Neptus MRA.

1.4 - IMC

O IMC é um protocolo orientado a mensagens, tendo sido projetado e implementado para facilitar as comunicações entre os veículos, sensores e operadores. O IMC define um conjunto de mensagens comuns, sendo compreendido por todos os sistemas, e é usado para comunicação entre os nós da rede, desde as *tasks* do DUNE aos *plugins* do *neptus*. O IMC está totalmente definido e documentado num único arquivo XML, podendo ser traduzido noutras linguagens utilizando a tecnologia Extensible Stylesheet Language Transformations (XSLT). O IMC também estabelece normas para a *serialização de dados* nos *formatos* JavaScript Object Notation (JSON) e XML, que permite a sua utilização por todos os dispositivos habilitados para a web. Todas as mensagens IMC são divididas em *header*, *payload* e *footer* (figura 7). O *header* contém o número de sincronização, o qual permite detetar diferentes *byte-orders (endianess)* das *serializações* e diferentes versões do protocolo, um identificador da mensagem, uma origem e um destino. O *payload* da mensagem varia de acordo com o identificador de mensagem. O *footer* possui o *checksum*, utilizado para detetar erros acidentais que podem ter sido introduzidos durante o seu transporte ou armazenamento e para verificar a integridade dos dados a qualquer momento.

O controlo é um controlo tipicamente utilizado em veículos autónomos [21] [22], o qual torna possível o desenvolvimento de aplicações modulares. O IMC permite que o *software* seja executado em paralelo com outros módulos somente através da troca de mensagens IMC, em que as interfaces dos diferentes tipos de componentes estão bem definidas. As redes de veículos e consolas são baseadas em mecanismos de comunicação IP, como *sockets* TCP e UDP, protocolo *Real-Time Publish-Subscribe, GSM/HDSPA* ou *modems* acústicos subaquáticos.

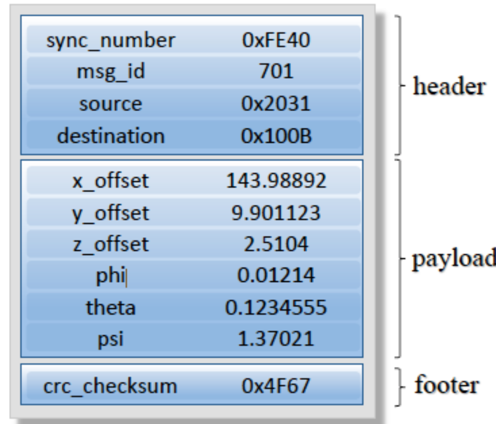


Figura 7 - Exemplo da estrutura de uma mensagem IMC [1].

O DUNE, o neptus e os veículos vão utilizar o protocolo IMC para comunicarem entre si, e vão criar registos das mensagens enviadas que podem ser analisados em tempo real ou posteriormente.

Esta análise em muito resultou do trabalho que me foi proposto: criar um *plugin* em Java para adicionar ao Neptus, de forma a concatenar todos os *logs* num único, mostrar as mensagens de todos eles, identificar quantos nós existem na rede e identificar o seu Id. Assim, este estudo resultou da análise de artigos e documentação mas também do desenvolvimento e extensão das mesmas ferramentas.

1.5 - Objetivos do trabalho

Neste trabalho é necessário desenvolver uma aplicação para integrar no software de controlo Neptus desenvolvido pelo LSTS da FEUP que permita visualizar todas as comunicações realizadas por uma rede de veículos quer entre tarefas que correm dentro de um mesmo veículo, quer entre os vários veículos e consolas. Os objetivos dividem-se em três grupos que são:

- intercalar dados provenientes de vários veículos:
    - Perceber o protocolo IMC. O Protocolo IMC é um protocolo orientado a mensagens que é utilizado por todos os veículos e consolas e que por isso é necessário compreender bem, de forma a perceber o que se vai projetar;
    - Identificar quantos nós existem na rede e identificar o seu id;
    - Concatenar todos os logs num único e mostrar as mensagens de todos eles, o que vai permitir visualizar o que aconteceu na rede de veículos em apenas um log.
- Criação de grafo de comunicações usando a ferramenta *graphviz*, ou semelhante como o *yEd Graph Editor* e o *Microsoft Automatic Graph Layout*, ferramentas estas que permitem uma fácil análise das comunicações da rede. A linguagem utilizada pelo *graphviz* é a linguagem DOT, e é compatível com os outros programas.
- Criar ferramentas de visualização:

- Visualização das comunicações com *highlight* das ligações, o que vai permitir visualizar as varias ligações efetuadas durante uma missão;
- Visualização de informações provenientes dos *logs* processados, tornando possível identificar as tarefas ou os sistemas que enviaram mensagens.

1.6 - Âmbito do trabalho

Este trabalho insere-se na disciplina de Dissertação do Mestrado Integrado em Engenharia Electrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto, no 2º semestre do ano lectivo de 2011/2012, e foi efetuado nos Laboratório de Sistemas e Tecnologia Subaquática.

1.7 - Estrutura do documento

O documento está estruturado em 6 capítulos. O presente capítulo consiste numa introdução ao tema em questão, a qual inclui uma descrição alargada da arquitetura e das ferramentas utilizadas na rede de veículos do LSTS, e numa apresentação dos objectivos deste trabalho.

O capítulo 2 é dedicado ao estado da arte de trabalhos existentes, focando temas que vão desde o IMC a outros projetos com uma arquitetura semelhante à do LSTS (como o ROS, e OpenJAUS) e referindo também ferramentas visualização como o R project, e o ROS.

No capítulo 3 é apresentada a descrição do problema, bem como as ferramentas utilizadas neste projeto.

No capítulo 4 encontra-se descrita a abordagem seguida na resolução de cada problema. Neste tema é projetada a arquitetura de código a seguir para intercalar dados, criar grafos de comunicações e ferramentas de visualização.

No capítulo 5 é realizada uma análise dos resultados obtidos, bem como a demonstração dos testes efetuados a partir de informação recolhida durante as missões.

Por último, no capítulo 6 encontram-se explícitas as conclusões a que se chegou com este trabalho, assim como os objectivos que se atingiram e varias sugestões para desenvolvimento futuro.

Capítulo 2

Estado da Arte

Escrever *software* para robôs é difícil, principalmente porque diferentes tipos de robôs podem ter *hardwares* extremamente diferentes, tais como os AUVs, ASVs, UAVs e os ROVs. Isto leva a que a reutilização de código muitas vezes não seja fácil, podendo resultar num código com tamanho assustador.

Para enfrentar estes desafios, muitos investigadores já criaram uma grande variedade de *software* para diminuir a complexidade e facilitar a prototipagem dos robôs usados no mundo académico, na indústria e a nível militar.

2.1 - ROS

O ROS foi concebido para satisfazer um conjunto específico de desafios encontrados no desenvolvimento do projeto STAIR [23], da Universidade de *Stanford*, e no programa de robôs pessoais [24], da *Willow Garage*. Os objetivos da filosofia ROS podem ser resumidos em:

- *Peer-to-peer*
- *Tools-based*
- *Multi-lingual*
- *Thin*
- *Free and Open-Source*

O sistema *Peer-to-Peer* usado no ROS consiste numa série de processos (*hosts*) ligados em *runtime* numa topologia *peer-to-peer*. Cenários com base num servidor central [25] podem beneficiar do *design multi-process* e *multi-host*, de forma a evitar o congestionamento do tráfego nos links *ethernet*, uma vez que a rede é maioritariamente constituída por links de redes sem fios que podem ser lentos e com a utilização das ligações *peer-to-peer* combinadas com módulos de software buffer ou "*fanout*". É importante não esquecer que com a topologia *peer-to-peer* é necessário um mecanismo de pesquisa que permita encontrar os processos em *runtime,* mecanismo esse é conhecido por *name service*, ou *master* [22].

O ROS foi projetado para suportar várias linguagens de programação, tais como C++,*Python*, *Octave* e LISP; a especificação ROS é utilizada na camada de mensagens com a negociação *peer-to-peer* e a configuração da ligação ocorre em XML-RPC.

Para facilitar o desenvolvimento *multi-linguagem*, o ROS usa a *Interface Definition Language* (IDL) de forma a descrever as mensagens enviadas entre os módulos. O gerador de código para cada linguagem suportada gera implementações nativas, que são automaticamente *serializadas* e *desserializadas* pelo ROS, enquanto as mensagens são enviadas e recebidas. No momento da escrita, o *ROS-based codebases* contém mais de 400 tipos de mensagens, que vão de transporte de dados do sensor para a deteção de objetos e para os mapas.

Para gerir a complexidade do ROS foi implementado um sistema *Tools-based* com um *microkernel*, o que permitiu a criação de um grande número de pequenas ferramentas para criar e executar os vários componentes ROS. Essas ferramentas permitem ver o código fonte, obter e definir parâmetros de configuração, visualizar graficamente as mensagem, etc.

Para tornar o ROS pequeno, e devido a dificuldades em reutilizar o código desenhado, utiliza-se uma métodologia de bibliotecas independentes que não têm dependências sobre o ROS, e utiliza o *CMake* tornando relativamente fácil de seguir essa ideologia. Toda a complexidade é praticamente colocada em bibliotecas, o que permite uma fácil reutilização do código a partir da criação de pequenos executáveis que expõem a funcionalidade da biblioteca do ROS, e ainda permite uma forma mais fácil de testar código ou uma unidade. Devido a esta característica o ROS pode reutilizar código de outros projetos *open-source*, como os drivers, sistema de navegação, simuladores do *Player Project* [26], algoritmos de visão do *OpenCV*[27], algoritmos de planeamento do *OpenRAVE*[28], etc.

O ROS é livre e *Open-Source*, pelo que o código fonte está disponível publicamente na sua totalidade. Este fato é fundamental para facilitar a deteção e depuração dos erros do *software* e para permitir um desenvolvimento consoante as necessidades dos utilizadores, em contraste com ambientes proprietários, como por exemplo o *Microsoft Robotics Studio* [29] e *Webots* [30], que apesar de possuírem bons atributos é difícil encontrar substituto numa plataforma *Open-Source* quando se está a desenvolver hardware e software em simultâneo.

O ROS é distribuído sob os termos da licença BSD, o que permite o desenvolvimento de projetos não-comerciais e comerciais. O ROS passa dados entre os módulos usando os processos de intercomunicação e não tem necessidade de unir os módulos no mesmo executável, o que viabiliza o licenciamento dos seus módulos individuais sob licenças, que vão da GPL à BSD até à proprietária.

O ROS foi desenhado sob uma filosofia de *software* modular, com uma estrutura em aberto; o *software* permite a implementação de novo hardware, e a necessidade de novos requisitos, apenas alterando algumas configurações ou criando novas librarias.

Como se pode ver na *Cheat Sheet* do ROS disponível no *site* [20], este é constituído pelos seguintes conjuntos de ferramentas, o *Filesystem Command-line Tools*, o *Common Command-line Tools*, o *Logging Command-line Tools*, o *tf Command-line Tools*, e o *Graphical Tools*. Nas figuras 8 e 9 encontram-se ilustradas duas ferramentas relativas ao *Graphical Tools*: o *rxgraph* e o *rxconsole*, respetivamente. O *rxgraph* mostra um gráfico no qual podem ser visualizados os nós da rede ROS e também os tópicos que os ligam, ao passo que o *rxconsole* é utilizado para mostrar e filtrar mensagens publicadas pelo *rosout*.

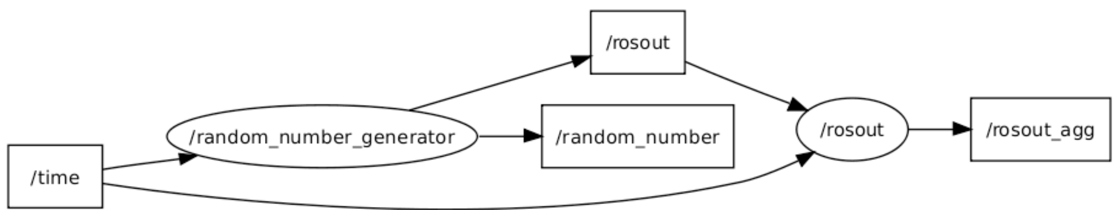


Figura 8 - *rxgraph* [20].

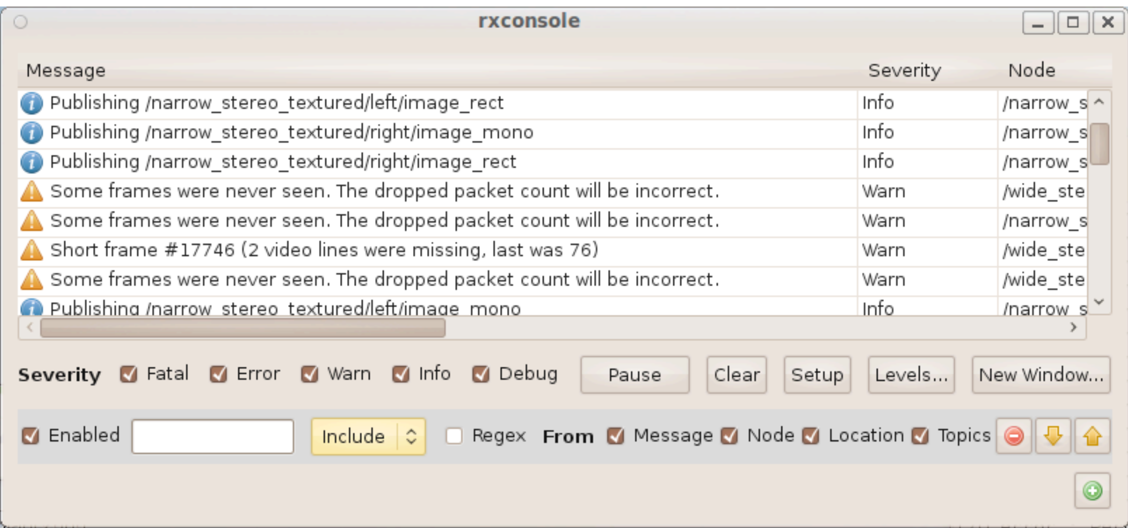


Figura 9 - *rxconsole* [20].

2.2 - R project

O R é uma linguagem e ambiente para computação estatística e gráficos que é muito parecido com a linguagem S [31], tendo sido desenvolvido nos Laboratórios *Bell* por *John Chambers* e colegas (figura 10). Na linguagem R existem algumas diferenças mas permite executar muito código escrito para S, e tem a capacidade de executar programas armazenados em arquivos de *script*. O R foi inicialmente escrito por *Ross Ihaka* e *Robert Gentleman* do Departamento de Estatística da Universidade de *Auckland*, Nova Zelândia, e como um *software* Livre o código fonte está disponível sob os termos da *Free Software Foundation's* GNU *General Public License* e tem uma vasta comunidade que contribui para o envio de código e relatórios de erros, e também permite ser compilado nas plataformas UNIX (FreeBSD, Linux, etc), Windows e MacOS [32].

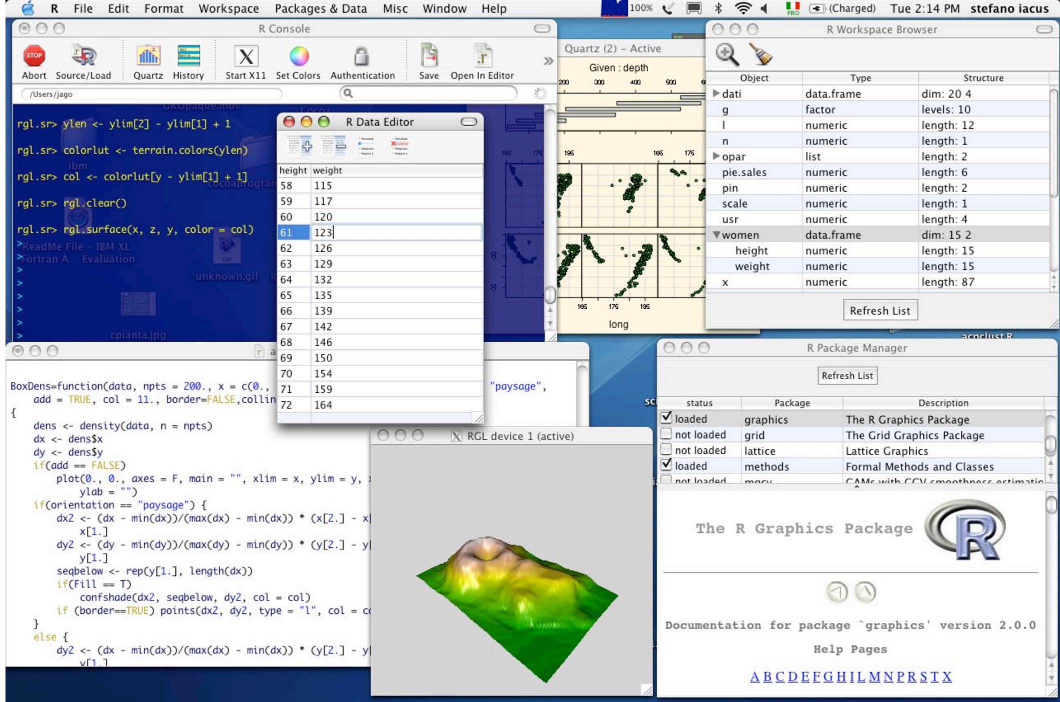


Figura 10 - *Screenshot* do R *project* num *MacOS X RAqua desktop* [32].

A linguagem R tem funções que permitem a programação modular, *branching* e *looping*. As funções em R são escritas em R e permitem efetuar a interface com os procedimentos escritos em C, C++ e *Fortran*, de forma a obter uma maior eficiência. A distribuição R contém várias funcionalidades, tais como modelos lineares generalizados, modelos de regressão não linear, análise de séries cronológicas, testes paramétricos clássicos e não paramétricos, *clustering* e *smoothing*. Não obstante, verifica-se ainda a existência de funções para o ambiente gráfico, o que o torna flexível, bem como de módulos adicionais (*add-on packages*). A linguagem R é fácil e para além de originar publicações de qualidade também cria gráficos bem concebidos, incluindo símbolos e fórmulas matemáticas, em que o utilizador tem controlo total.

O ambiente R tem um conjunto de *software* que inclui [32]:

- Um manuseio eficiente de dados e facilidade de armazenamento.
- Um conjunto de operadores para cálculos sobre cálculo*rrays*, em algumas matrizes.
- Uma grande coleção, coerente e integrada, de ferramentas para análise de dados.
- Facilidade gráfica para análise e *display* de dados.
- Uma linguagem de programação simples e eficaz que inclui condicionais, *loops*, funções recursivos definidas pelo utilizador, e *input* e *output facilities*.

O código fonte, as distribuições binárias e a documentação para R podem ser obtidos através da *Comprehensive R Archive Network* (CRAN). O código fonte também está disponível através do repositório SVN (*Subversion*) do *R-project* [33]; *Tarballs* com *snapshots* diários da *r-devel* e *r-patched* e versões de desenvolvimento do R podem ser encontrados no ftp do ETH [34].

A distribuição R trás os seguintes *Add-on packages*:

- base: Funções de base R.
- *compiler*: compilador código R.
- *datasets*: Base de dados para suporte R.

- *grDevices*: Dispositivos gráficos para gráficos base e *grid*.
- *graphics*: Funções R para gráficos de base.
- *grid*: A reescrita dos recursos de *layout* gráficos, além de algum apoio para interação.
- *methods*: Métodos formalmente definidos e classes para os objetos R, além de ferramentas de programação, como descrito no *Green Book*.
- *parallel*: Suporte para computação paralela, incluindo por bifurcação e soquetes, e geração de números aleatórios.
- *Splines*: Regressão *spline,* funções e classes.
- *stats*: Funções R estatísticas.
- *stats4*: Funções estatísticas usando classes S4.
- *tcltk*: Interface e ligações de linguagem a elementos Tcl/Tk do GUI.
- *tools*: Ferramentas para o desenvolvimento de pacotes e administração.
- *utils*: Utilitários para funções R.

Os *Add-on packages* para o R disponíveis no repositório do CRAN são:

- *KernSmooth*: Funções para a suavização do *kernel* correspondentes ao livro "*Smoothing Kernel*" [35].
- MASS: Funções e conjuntos de dados do pacote principal de *Venables* e *Ripley*, "*Modern Applied Statistics with* S" [36].
- *Matrix*: Um pacote de Matrizes.
- *boot*: Funções e conjuntos de dados de inicialização do livro "*Bootstrap Methods and Their Applications*" [37].
- *class*: Funções de classificação (k-vizinho mais próximo e LVQ).
- *cluster*: Funções para análise de cluster.
- *codetools*: Ferramentas de análise de código.
- *foreign*: Funções para ler e gravar dados armazenados pelo software estatístico como o *Minitab*, S, SAS, SPSS, *Stata*, *Systat*, etc.
- *lattice*: Gráficos *lattice*, uma implementação de funções gráficas *Trellis*.
- *mgcv*: Rotinas para GAMs e outros problemas generalizados de regressão de *ridge* com seleção múltipla, alisamento, e parâmetros por GCV ou UBRE.
- *nlme*: Ajusta e compara modelos lineares Gaussianos e não-lineares de efeitos mistos.
- *nnet*: *Software* para *feed-forward neural networks*.
- *rpart*: Particionamento recursivo e árvores de regressão.
- *spatial*: Funções para *kriging* e análise de padrões de pontos de W. *Venable*s e *Ripley* B. [36].
- *survivel*: Funções para análise de sobrevivência, incluindo a probabilidade penalizada.

As ligações mais utilizadas são ligações por ficheiros criadas pela função *file*. Estas ligações podem ser abertas para leitura, escrita ou anexar, em texto ou no modo binário; existe ainda a possibilidade de leitura e escrita simultânea. Por norma, uma ligação não se encontra aberta quando o ficheiro é criado, pelo que é necessário abrir uma e fechar a ligação após o uso; existem funções genéricas *open* e *close* com métodos para abrir e fechar as ligações. As ligações também podem ser efetuadas a partir de ficheiros comprimidos ou através do algoritmo *gzip* [38] (pela função *gzfile*), enquanto que os ficheiros compactados por *bzip2* [39] são através da função *bzfile* [32].

Também é possível efetuar ligações de texto, as quais permitem que os vetores de caracteres possam ser lidos como se as linhas fossem ficheiros de texto. Uma ligação de texto é aberta, designando-se por *textConnection*, e copia o conteúdo atual do vetor de caracteres para um *buffer* interno no momento da criação.

As *pipes* [40] são uma forma especial de arquivo que se ligam a outro processo, e são criadas pela função *pipe*. A abertura de uma ligação *pipe* para entrada executa um comando do sistema operativo que torna a saída padrão, disponível para a entrada da ligação R.

Também é possível utilizar URLs de tipos de "http://", "ftp://" e "file://" que podem ser lidas usando a função url; a função file também aceita as URLs como especificação de ficheiro e chamada a url.

Os *sockets* [41] também podem ser usados nas ligações *socketConnection,* podem ser escritas ou lidas a partir de ficheiros e podem ser usados *sockets* de cliente ou servidor. Uma das desvantagens dos *sockets* é que a instalação pode ser bloqueada por razões de segurança ou para forçar o uso de caches. O interface *sockets* de baixo nível é dado pelas funções *make.socket*, *read.socket*, *write.socket* e *close.socket*.

A linguagem XML é utilizada para fornecer estruturas de dados padrão e está a tornar-se o padrão de dados de marcação e troca. O XML fornece uma maneira de especificar a codificação dos ficheiros.

A função *download.file* é fornecida para ler um ficheiro Web via FTP ou HTTP e gravá-lo num ficheiro; Isto também pode ser feito com as funções *read.table* e scan, que podem ler diretamente a partir de uma URL, usando url para abrir a ligação.

O *Common Object Request Broker Architecture* (CORBA) é semelhante ao *Distributed Component Object Model* (DCOM), que permite os aplicativos chamarem métodos ou operações, em objetos de servidores que correm noutras aplicações, e podem ser programados em linguagens diferentes e executados em máquinas diferentes. O pacote CORBA disponível no *site* da *Omegahat* [42] permite que comandos R possam ser usados para localizar servidores CORBA, consultar métodos e invocar métodos dinamicamente nesses objetos. Valores de R dados como argumentos a essas chamadas são exportados na chamada e disponibilizados para a invocação da operação. Podem-se também criar servidores CORBA em R, que permitem outras aplicações chamar esses métodos e é ainda possível usar o pacote CORBA para obter computação distribuída, e paralela em R.

2.3 - OpenJAUS

O JAUS é um *standard* da SAE *International* para comunicação, comando e controlo de sistemas não-tripulados. Ele especifica como o *software* deve enviar e receber dados, a fim de interagir numa rede, o que permite que, quando o *software* está em conformidade com o JAUS, facilmente se pode integrar o sistema com outros sistemas baseados em JAUS (figura 11). Isto permitiu a criação do *Open*JAUS, que consiste numa biblioteca de *software* escrito em C++ e possui um Kit de desenvolvimento que fornece exemplos de projetos diversos e utilitários[43].

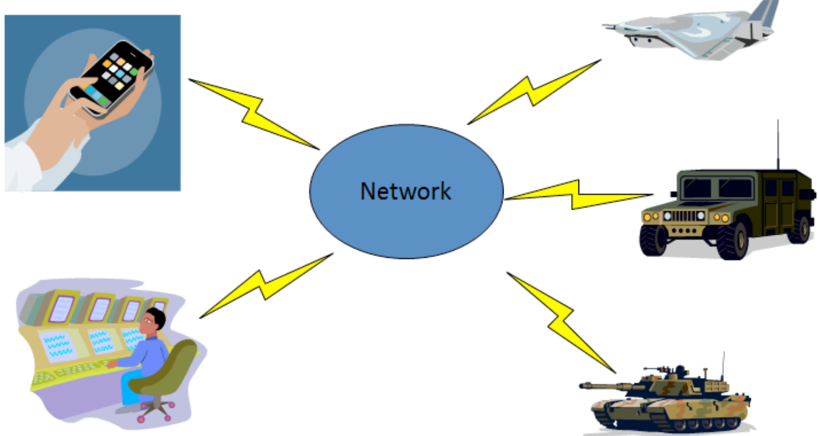


Figura 11 - Sistema JAUS feito de subsistemas ligados em rede [43].

O JAUS define o protocolo de comunicação para os sistemas de veículos não tripulados, alguns dos seus componentes internos e a sua interação com estações de controlo, o que possibilita a formatação de mensagens para o transporte entre os serviços do sistema, bem como conjuntos de serviços *standard*[44], que descrevem componentes específicos. Este emprega uma Arquitetura *Service Oriented Architecture* (SOA) [45] o que lhe permite um comando e controlo distribuído.

O *standard* JAUS foi desenvolvido pela *Society of Automotive Engineers* (SAE) no âmbito do *Aerospace Standards Unmanned Systems Steering Committee* (AS-4), e trata-se de um *standard* proprietário, em que os documentos que definem o JAUS podem ser comprados. O objetivo do JAUS é tornar os produtos de diferentes fabricantes uniformes e consistentes, a fim de permitir a interoperabilidade, a permutabilidade e modularidade o que, por sua vez, levará à redução do custo de aquisição e esforço humano em integração de sistemas.

O *Open*JAUS foi desenhado com o intuito de ter uma interface simples e robusta para o utilizador, clareza e integralidade da API, e o cumprimento das normas publicadas[46].

O principal objetivo do JAUS é a comunicação e a interação dos sistemas em rede. A rede é subdividida em camadas hierárquicas, sendo que a camada mais alta é constituída por subsistemas; um subsistema é uma entidade física, como um veículo ou uma unidade de controlo. No nível seguinte, os subsistemas são divididos em nós, que representam uma computação física no sistema; um nó pode ser um computador ou microcontrolador dentro de um subsistema e os nós podem conter vários componentes, os quais podem ser aplicações ou *threads* em execução no nó. Na última camada, os componentes são feitos de um ou mais serviços, sendo que um serviço é uma abstração lógica que fornece alguma função útil para o sistema e que tem uma interface de mensagens e protocolos bem definida.

A SAE publica a norma JAUS em conjuntos de documentos relacionados mas separados, tal como se encontra exemplificado no *site* do *OpenJAUS*[43]:

- JAUS *Transport Specification* (AS5669A) - Especificações para o UDP, TCP e da transmissão de dados baseada em série das mensagens JAUS, com o objetivo de padronizar o formato das mensagens JAUS que serão transportadas por comunicações *Ethernet* ou série.
- JAUS *Service Interface Definition Language* (AS5684) - Define as estruturas de dados de serviços, mensagens e protocolo, como um esquema XML, para eliminar a ambiguidade

em serviço e interpretação da mensagem, e faz a validação/geração automática de código para os possíveis serviços.

- JAUS *Core Service Set* (AS5710A) - Define com JSIDL os serviços de baixo nível de descoberta e transporte. Estes incluem definições de formato de mensagem e protocolo escrito em tabelas e diagramas, para especificar um conjunto base de serviços comuns que podem ser implementadas por todos os componentes JAUS.
- JAUS *Mobility Service Set* (AS6009) – Define os serviços comuns de mobilidade, tais como posicionamento global e a plataforma de controlo de veículos definida em JSIDL e também inclui definições de formato de mensagem e protocolo escrito em tabelas e diagramas, o que permite ativar o comando e controlo de muitas plataformas de veículos definindo serviços abstratos que são agnósticos aos tipos específicos de mobilidade de veículos.
- JAUS *Human Machine Interface Service Set* (AS6040) – Definição de serviços em JSIDL para interação HMI e inclui o desenho, entrada de teclado, entrada de dispositivo apontador, e controlos analógicos e digitais, para fornecer um meio *standard* para a interação humana com componentes duma rede JAUS.
- JAUS *Manipulation Service Set* (AS6057) - Definições de serviço em JSIDL para controlar manipuladores robóticos. As mensagens são definidas genericamente, de modo que possam ser aplicadas a muitos tipos diferentes de manipuladores, pelo que padroniza o comando e o controlo de manipuladores robóticos que podem ser parte de uma plataforma de sistemas não-tripulados.

O *JAUS Service Interface Definition Language* (JSIDL) do JAUS, para satisfazer a arquitetura SOA, fornece um esquema XML preciso, o qual é utilizado para conceber serviços JAUS. Um serviço é definido por um conjunto específico de entrada e saída de mensagens JAUS e por protocolo numa máquina de estado que controla como as mensagens devem ser recebidos, processadas e enviadas pelo serviço. O objetivo do JSIDL é reduzir erros humanos na interpretação de serviços *standard* do JAUS. O XML permite a especificação dum serviço para ser automaticamente validado e para gerar *software* de implementação de *stubs* de serviços e das mensagens, o que permite eliminar erros humanos. A sintaxe XML é usada para a especificação de serviços quando eles são concebidos e publicados na norma; o *Open*JAUS usa a sintaxe da máquina validada para o *design* e para gerar código automaticamente.

O JSIDL é utilizado para especificar como as mensagens devem ser estruturadas, pois define o formato dos campos de dados nas mensagens JAUS e também fornece a sintaxe de como designar esses campos.

A definição de protocolo de serviço sob a forma de máquinas de estado do JSIDL é um serviço que responde às mensagens de entrada de forma diferente, dependendo do estado em que ele se encontra. O JSIDL vai fornecer a sintaxe XML para descrever a estrutura de serviço de uma máquina de estado que respeita a documentação *standard* JAUS.

O *JAUS Core Service Set* (JSS Core) permite a interoperabilidade *on-line* de sistemas não-tripulados e dos seus componentes, através de um conjunto de serviços que permitem: trocas de informação entre os componentes, configuração de eventos baseados em mensagens, descoberta dos serviços *on-line*, etc. O JSS *Core* define os serviços básicos que são necessários pelos componentes de nível superior, o que o torna numa ferramenta crítica quando é necessário evitar que um serviço receba múltiplos sinais de controlo que entrem em conflito a partir de fontes diferentes. Os detalhes de implementação desses serviços em

*Open*JAUS são fornecidos no Guia de utilizador do *Open*JAUS. Alguns dos serviços definidos no JSS *Core* e para que são usados encontram-se definidos na tabela 1.

Tabela 1 - Alguns serviços do JSS Core [43].

Serviço	Propósito	Mensagens
Transport	Atua como uma *gateway* para todas as mensagens que entram e saem de um componente.	*None*
Events	Permite que outros componentes possam solicitar mensagens de serviços que herdam dos eventos de serviço de forma periódica fixa. Isso é o equivalente JAUS de um mecanismo de publicação e assinatura.	*CreateEvent, UpdateEvent, CancelEvent, QueryEvents, RejectEventRequest, Event*
Access Control	Permite que os serviços que herdam do serviço de controlo de acesso a ser exclusivamente controlado por uma única fonte. Por sua vez, os serviços controlados só aceita comandos a partir de seus componentes de controlo. Isto é como os componentes JAUS implementam exclusão mútua.	*RequestControl, ReleaseControl, QueryControl, QueryAuthority, ReportControl, ConfirmControl*
Management	Permite que os componentes do cliente para controlar e aceder a informação sobre o estado interno de outra função de componentes. Por exemplo, isso permite que os componentes do cliente para redefinir ou desligar um componente de servidor.	*Shutdown, Standby, Resume, Reset, SetEmergency, ClearEmergency, QueryStatus, ReportStatus*
Time	Fornece uma interface para reportar o tempo de um componente do sistema interno para outros componentes.	*QueryTime, ReportTime*
Liveness	Permite que os componentes do cliente para determinar se outra componente está *online* e responde há comunicação de mensagens. Similar ao conceito de um *ping* rede.	*QueryHeartbeatPulse, ReportHeartbeatPulse*
Discovery	Permite aos componentes descobrir a presença uns dos outros e também permite trocar informações sobre os serviços que implementa e apoia.	*RegisterServices, QueryIdentification, QueryConfiguration, ReportIdentification, ReportConfiguration, ReportServiceList*

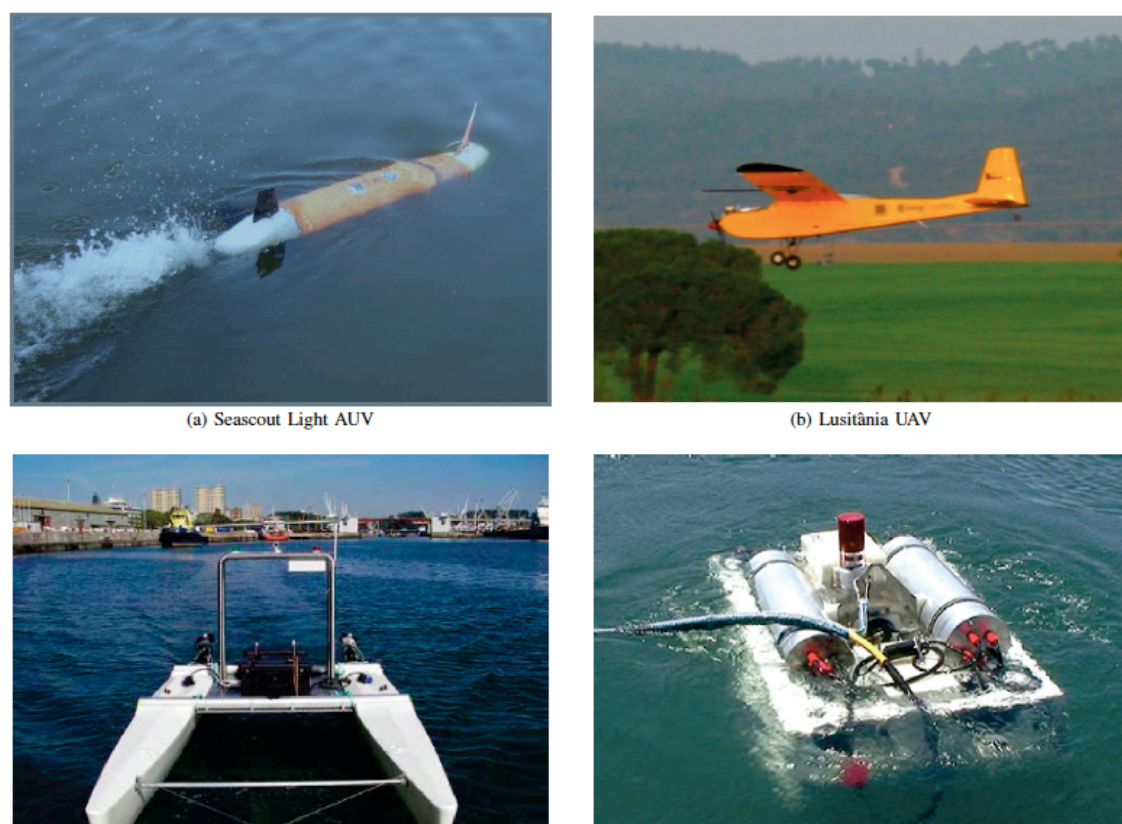
Podemos concluir que o *Open*JAUS ajudar a implementar as normas JAUS, de uma forma simplificada e bem planeada, e a sua implementação pode ser efetuada pelas ferramentas *Eclipse*, *Visual Studio* e linguagem de programação C, C++, e *Java* e oferece mecanismos simples para melhor *timing*, compressão de dados, configuração do sistema, etc.

2.4 - IMC

O IMC é um protocolo orientado a mensagens, projetado e implementado no Laboratório de sistemas e Tecnologia subaquática (LSTS), para construir sistemas interligados de veículos, sensores e operadores humanos que são capazes de cooperar entre si, abstraindo-se do *hardware* e heterogeneidade das comunicação e fornecendo um conjunto compartilhado de mensagens que podem ser serializadas [12] e transferidas através de diferentes meios.

O objetivo principal é construir sistemas interligados de veículos (figura 12), sensores e operadores humanos que são capazes de perseguir objetivos comuns de forma cooperativa, trocando informações em tempo real sobre o meio ambiente e objetivos atualizados, com o intuito de permitir aos operadores poderem responder a situações imprevistas onde a sua contribuição é fundamental, como amostragem adaptativa[47], patrulhamento da fronteira

ou guerra cooperativa[48], em que o IMC permite uma rápida integração de novo *hardware* e facilita o desenvolvimento de novos algoritmos de cooperação.



(a) Seascout Light AUV (b) Lusitânia UAV

(c) Swordfish ASV (d) KOS ROV

Figura 12 - Alguns veículos que usam IMC [11].

O protocolo IMC define uma infraestrutura que é modular e fornece camadas diferentes de controlo e de detecção nos ASVs, ROVs, AUVs e UAVs. O fluxo de mensagens que corresponde ao controlo e várias camadas de deteção dentro IMC[11]:

- Mensagens de *Mission control* definem a especificação de uma missão e do ciclo de vida, para a interface entre um comando CCU e um módulo supervisor da missão. A missão é uma sequência ou um gráfico de manobras.
- Mensagens de *Vehicle control* são utilizados para interligar o veículo a partir de uma fonte externa.
- Mensagens de *Maneuver* são usadas para definir manobras e associam-se a comandos e a estados de execução.
- Mensagens de *Guidance* estão relacionadas com a orientação usada nas manobras autónomas.
- Mensagens de *Navegation* definem a interface para a comunicação do estado de um veículo de navegação. As mensagens *Estimated State* definem o estado de navegação de um veículo pela convenção SNAME[49].
- Mensagens dos sensores são usadas para comunicar as leituras dos sensores pelos respetivos controladores de *hardware*.
- Mensagens dos atuadores vão especificar a interface com os controladores de *hardware*.

Os grupos de mensagem citados são apenas alguns exemplos, sendo que a lista completa pode ser consultada na tabela 6.

Esta estrutura (figura 13) vai-nos permitir o desenvolvimento modular de aplicações que podem ser executadas em isolamento lógico, interagindo com outros módulos através da troca de mensagens IMC.

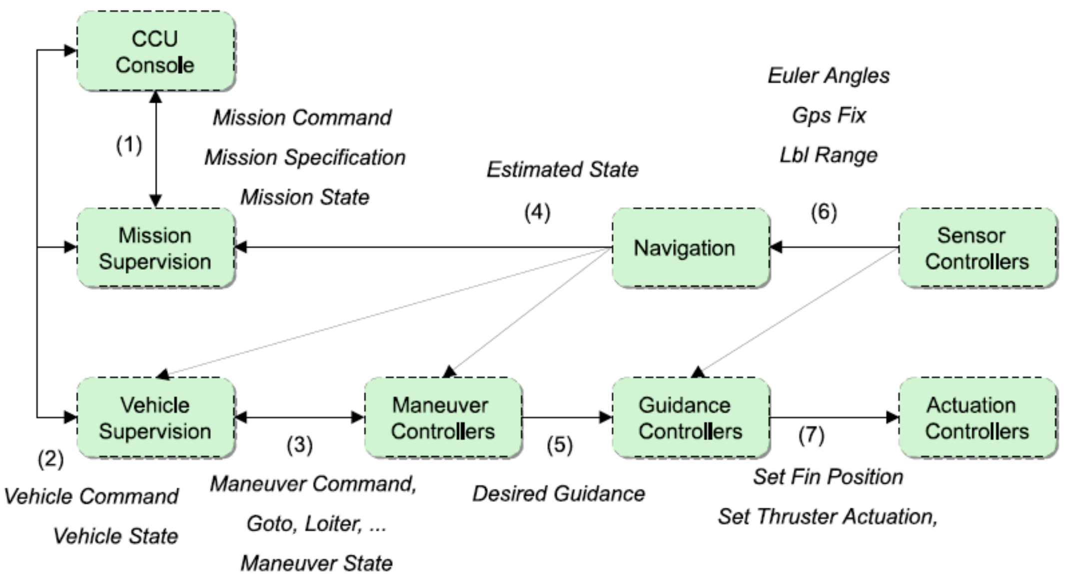


Figura 13 - Mensagens IMC no AUV *Seascout Light*[11].

O DUNE[49] é o *software* implementado nos veículos autónomos que permite a abstração de mensagens, e fornece mecanismos de transporte para comunicação externa (figura 14). Os veículos podem ser monitorizados e controlados externamente, através de consolas neptus [49], [50]. A Rede de veículos e consolas é baseada em mecanismos de comunicação IP, tais como *sockets* TCP e UDP, *o* protocolo *Real-Time Publish-Subscribe*[51] ou *modems* acústicos subaquáticos [52].

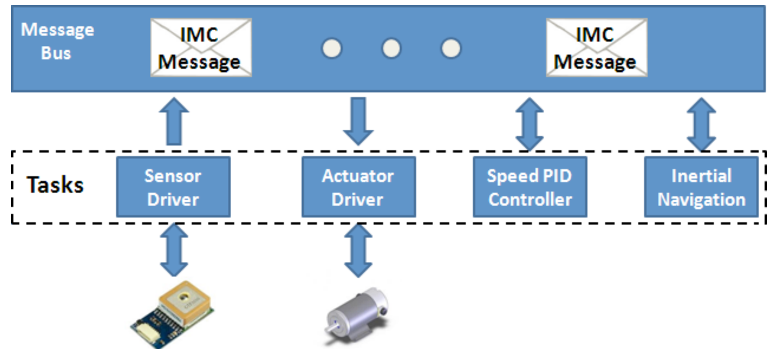


Figura 14 - Conceito de transferência de mensagens e implementação de *tasks* do DUNE [1].

As mensagens IMC definem entidades que contêm um número de identificação exclusivo e que consistem numa sequência de campos de dados capazes de representar inteiros de largura fixa, números de vírgula flutuante e sequências de bytes de comprimento variável. Os Inteiros podem ser atribuídos ou não atribuídos, com tamanhos que variam dos 8 aos 64 bits;

os números de vírgula flutuante têm dois tamanhos, que podem ser de 32 ou 64 bits, como podemos verificar na seguinte tabela 2.

Tabela 2 - Lista de tipos de campos válidos [53].

Tipo ID	Nome	Largura (*bytes*)	Descrição
0	int8_t	1	8 bit signed integer.
1	uint8_t	1	8 bit unsigned integer.
2	int16_t	2	16 bit signed integer.
3	uint16_t	2	16 bit unsigned integer.
4	int32_t	4	32 bit signed integer.
5	uint32_t	4	32 bit unsigned integer.
6	int64_t	8	A 64 bit signed integer
7	fp32_t	4	32 bit single precision floating point number in IEEE 754 format.
8	fp64_t	8	64 bit double precision floating point number in IEEE 754 format.
9	rawdata	-	Variable length byte stream.
10	plaintext	-	Variable length ASCII character stream.
11	message	-	An inline message. Useful for encapsulating other messages.

Para garantir o transporte ou armazenamento das mensagens na transmissão e receção é necessário encapsular as mensagens num pacote ou utilizar a *serialization*. A *serialization* é realizada a fim de traduzir os dados dos pacotes para um fluxo binário na mesma ordem em que foram definidos. Por inspeção do número de sincronização o recetor é capaz de deduzir a ordem de bytes dos restantes campos de dados e realizar as conversões necessárias para a interpretação correta, o que é denominado por *deserialization*. Isto permite que a comunicação entre nós com as mesmas ordens de bytes não sofra nenhuma sobrecarga na conversão. Não é necessário utilizar uma *deserialization* especial para os campos descritos na tabela 3.

Tabela 3 - Tipo de *serialization* [53].

Nome	*Serialization*
rawdata	*A sequence of type rawdata is serialized by prepending a value of type uint16_t, representing the length of the sequence, to the stream of bytes. On deserialization the prepended value is used to retrieve the correct size of data bytes. The rawdata type length is limited only by the communication protocol in use.*
plaintext	*A sequence of type plaintext is serialized by prepending a value of type uint16_t, representing the length of the sequence, to the stream of ASCII characters. On deserialization the prepended value is used to retrieve the correct ASCII character sequence size. The plaintext type length is limited only by the communication protocol in use.*
message	*A field of type message is serialized by prepending a value of type uint16_t, representing the identification number of the message, to the serialized message payload. The special identification number 65535 must be used when no message is present. On deserialization the prepended value is used to retrieve the correct message identification number. The message type length is limited only by the communication protocol in use.*

Existem três formatos de *serialization* diferentes. Um deles é o formato LSTS *Log Format* (LLF), tratando-se de um formato de texto utilizado para criar *logs* IMC; este formato é de fácil compreensão e pode ser analisado diretamente por muitos aplicativos, como o Excel e o Matlab, e *software* personalizado de análise e revisão das missões[54]. Como o protocolo IMC

está constantemente em evolução o formato LLF tinha que ser independente da versão do protocolo IMC para permitir a revisão de dados de missões passadas, pelo que se definiu que o formato do *log* usa tabulações. Para cada coluna há um cabeçalho descrevendo o tipo de dados, o nome e as unidades a ser usadas, quando representados os dados.

Um segundo formato é o LSTS Serialization Format (LSF) que consiste na concatenação das mensagens (binárias) num único ficheiro. Para a correcta interpretação desse ficheiro é necessário ter conhecimento da definição IMC correspondente e, por isso, os ficheiros LSF são normalmente acompanhados do ficheiro de definição IMC (IMC.xml).

O último formato é o IMC-XML (figura 15) que consiste na tradução da definição da mensagem e conteúdos para XML. Este formato permite a integração de componentes *web-based* e *web-enabled* e sensores *third-party* em aplicações de larga escala de disseminação de dados[55].


```
- <imc version="3.0.0">
  - <message name="EstimatedState" time="1.258308235E9" src="0" dst="0">
      <field name="ref" type="uint8_t">NED_ONLY</field>
      <field name="x" type="fp64_t">70.17007237213076</field>
      <field name="y" type="fp64_t">187.69280917289933</field>
      <field name="z" type="fp64_t">0.0</field>
      <field name="phi" type="fp64_t">0.0</field>
      <field name="theta" type="fp64_t">0.1</field>
      <field name="psi" type="fp64_t">0.2</field>
      <field name="lat" type="fp64_t">41.342343</field>
      <field name="lon" type="fp64_t">-8.124922</field>
      <field name="depth" type="fp64_t">23.34</field>
    </message>
  </imc>
```


Figura 15 - Exemplo de *serialization* IMC-XML [12].

Todas as mensagens IMC são divididas em *header*, *payload* e footer. O header e o footer têm a mesma estrutura para todos os pacotes e são definidos por uma sequência de campos de dados de tamanho fixo. O *payload* varia de mensagem para mensagem e tem um tamanho variável, podendo ser vazio, como mostra a figura 16.

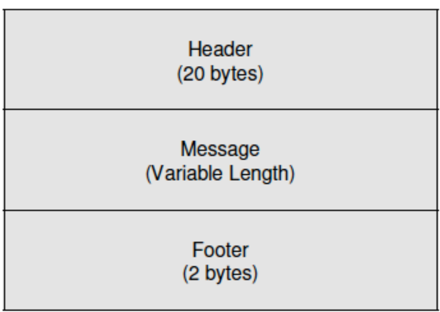


Figura 16 - Formato do pacote [53].

O *header* é sempre colocado no índice do pacote e contém o número de sincronização, que nos permite detetar diferentes ordens de bytes das *serializations* e diferentes versões de protocolo; possui ainda um identificador de mensagem, uma origem e um destino, como podemos ver na tabela 4

Tabela 4 - Header do pacote [53].

Campo	Nome	Tipo	Valor Fixo	Descrição
1	*Synchronization Number*	*uint16_t*	*0xFE46*	*The synchronization number marks the beginning of a packet. It denotes the packet API version and can be used to deduce the byte order of the sending host. It encodes value 0xFE where major equals the major version number of the protocol and minor equals the minor version of the protocol. The packet recipient is responsible for the correct interpretation of the synchronization number and byte order conversions.*
2	*Message Identification Number*	*uint16_t*	-	*The identification number of the message contained in the packet. This field is used for correct message interpretation and deserialization.*
3	*Message size*	*uint16_t*	-	*The size of the message data in the packet. The maximum allowed value is 65535.*
4	*Time stamp*	*fp64_t*	-	*The time when the packet was sent, as seen by the packet dispatcher. The number of seconds is represented in Universal Coordinated Time (UCT) in seconds since Jan 1, 1970 using IEEE double precision floating point numbers.*
5	*Source Address*	*uint16_t*	-	*The Source IMC system ID.*
6	*Source Entity*	*uint8_t*	-	*The entity generating this message at the source address*
7	*Destination Address*	*uint16_t*	-	*The Destination IMC system ID.*
8	*Destination Entity*	*uint8_t*	-	*The entity that should process this message at the destination address*

O *footer* tem informação de validação do pacote como o *checksum*, utilizado para detetar erros acidentais, os quais podem ter sido introduzidos durante o seu transporte ou armazenamento, e para verificar a integridade dos dados a qualquer momento, sendo sempre colocado na extremidade do pacote. Os campos do *footer* estão descritos na tabela 5.

Tabela 5 - Footer do pacote [53].

Campo	Nome	Tipo	Descrição
1	*Check Sum (CRC-16-IBM)*	*uint16_t*	*The check sum field is computed using the CRC-16-IBM with polynomial 0x8005 (x"16 + x"15 + x"2 + 1). The data contributing for the CRC includes all preceding header and message bytes.*

O *payload* da mensagem varia de acordo com o identificador de mensagem e as mensagens podem ser agrupadas por agrupamentos lógicos, como podemos ver na tabela 6. A lista completa de mensagens encontra-se na tabela 8.

Tabela 6 - Grupo lógico de mensagens [53].

Grupo	Min. msg. id	Max. msg. id
Core	0	49
Simulation	50	99
Storage	100	149
Networking	150	199
Acoustic Networking	200	249
Sensors	250	299
Actuation	300	349
Navigation	350	399
Guidance	400	449
Maneuvering	450	499
Vehicle Supervision	500	549
Plan Supervision	550	599
DUNE	600	649
CCU	650	699
Temporary and Debugging Messages	700	999
Custom	1000	65534

O protocolo IMC utiliza o documento *Extensible Markup Language* (XML)[56] que, quando alteradas, podem ser verificadas por um *XML schema* (XSD). O documento XML é organizado nas seguintes seções:

- Descrição do protocolo de IMC.
- Lista de tipos suportados com tamanho associado.
- Regras de *serialization* e de *deserialization* para os campos de comprimento variável.
- Lista de unidades com suporte para campos de dados.
- Definição do *header* do pacote.
- Definição do *footer* pacote.
- Lista de grupos de mensagens.
- Definição de mensagens e respetivos campos.

Cada campo de mensagem XML deve ter pelo menos um nome, uma abreviatura e um tipo, e também podem ser definidos unidades e gamas de valores permissíveis. Uma mensagem do tipo *Entity List* é definida utilizando o código XML o qual se encontra defenido na lista 1.

```
1   <message id="5" name="Entity List" abbrev="EntityList" source="vehicle">
2       <description>This message describes the names and identification numbers of all entities in the
    system</description>
3       <field name="operation" abbrev="op" type="uint8_t" unit="Enumerated" prefix="OP">
4         <enum abbrev="REPORT" name="Report" id="0"/>
5         <enum abbrev="QUERY" name="Query" id="1"/>
6       </field>
7       <field name="list" abbrev="list" type="plaintext" unit="TupleList">
8         <description>Example: "Battery=11;CTD=3"</description>
9       </field>
10    </message>
```

Lista 1 - Código XML de uma mensagem do tipo *Entity* List.

Esta é uma mensagem do tipo *Entity List:* tem como identificador o número cinco, um *payload* fixo de três *bytes* e é utilizada por veículos reais ou simulados; a sua função é descrever os nomes e números de identificação de todas as entidades do sistema. A tabela 7 mostra o excerto de xml convertido para pdf.

Tabela 7 - Mensagem convertida de xml para pdf [53].

Campo	Nome	Tipo	Unidade	Descrição	Variação
1	*op*	*uint8_t*	*Enumerated*	*Operation*	*0 - Report* *1 - Query*
2	*list*	*plaintext*	*TupleList*	*Example: "Battery=11;CTD=3"*	*Same as field type.*

A linguagem *XSL Transformations* (XSLT)[57] é usada para gerar automaticamente a documentação do protocolo IMC e implementações otimizadas em C++, C# e Java, o que proporciona flexibilidade e desempenho para efetuar a operação em tempo real.

O IMC está atualmente na versão 4.6.1, e ainda se encontra sob um desenvolvimento ativo para suportar o novo *hardware* e conceitos de operações.

Tabela 8 - Lista completa de mensagens [53].

ID	Abreviatura	Nome
0	*MessageList*	*Message List*
1	*EntityState*	*Entity State*
2	*QueryEntityState*	*Query Entity State*
3	*EntityInfo*	*Entity Information*
4	*QueryEntityInfo*	*Query Entity Information*
5	*EntityList*	*Entity List*
6	*EntityControl*	*Entity Control*
7	*CpuUsage*	*CPU Usage*
8	*TransportBindings*	*Transport Bindings*
50	*SimulatedState*	*Simulated State*
51	*LeakSimulation*	*Leak Simulation*

52	*UASimulation*	*Underwater Acoustics Simulation*
100	*StorageUsage*	*Storage Usage*
101	*CacheControl*	*Cache Control*
102	*LoggingControl*	*Logging Control*
103	*LogBookEntry*	*Log Book Entry*
104	*LogBookControl*	*Log Book Control*
105	*ReplayControl*	*Replay Control*
106	*ClockControl*	*Clock Control*
150	*Heartbeat*	*Heartbeat*
151	*Announce*	*Announce*
152	*AnnounceService*	*Announce Service*
153	*Acknowledgement*	*Acknowlegement*
154	*Envelope*	*Envelope*
155	*RSSI*	*Receive Signal Strength Information*
156	*VSWR*	*Voltage Standing Wave Ratio*
157	*LinkLevel*	*Link Level*
158	*Sms*	*SMS*
200	*LblRange*	*LBL Range*
201	*LblRangeAcceptance*	*LBL Range Acceptance*
202	*LblPingNext*	*LBL Ping Next*
203	*LblDetection*	*LBL Detection*
204	*LblBeacon*	*LBL Beacon Configuration*
205	*LblConfig*	*LBL Configuration*
206	*AcousticRange*	*Acoustic Range*
207	*AcousticRangeReply*	*Acoustic Range Reply*
208	*AcousticMessage*	*Acoustic Message*
209	*AcousticDiagnostic*	*Acoustic Diagnostic*
210	*AcousticNoise*	*Acoustic Noise*
211	*AcousticPing*	*Acoustic Ping*
212	*AcousticPingReply*	*Acoustic Ping Reply*
213	*AcousticOperation*	*Acoustic Operation*
250	*Rpm*	*Rotations Per Minute*
251	*Voltage*	*Voltage*
252	*Current*	*Current*
253	*GpsFix*	*GPS Fix*
254	*GpsFixRejection*	*GPS Fix Rejection*
255	*EulerAngles*	*Euler Angles*
256	*AngularVelocity*	*Angular Velocity*
257	*Acceleration*	*Acceleration*
258	*MagneticField*	*Magnetic Field*
259	*GroundVelocity*	*Ground Velocity*
260	*WaterVelocity*	*Water Velocity*
261	*BottomDistance*	*Bottom Distance*
262	*Temperature*	*Temperature*
263	*Pressure*	*Pressure*

264	*Depth*	*Depth*
265	*DepthOffset*	*Depth Offset*
266	*SoundSpeed*	*Sound Speed*
267	*WaterDensity*	*Water Density*
268	*Conductivity*	*Conductivity*
269	*Salinity*	*Salinity*
270	*WindSpeed*	*Wind Speed*
271	*RelativeHumidity*	*Relative Humidity*
272	*DevDataText*	*Device Data (Text)*
273	*DevDataBinary*	*Device Data (Binary)*
274	*SidescanConfig*	*Sidescan Configuration*
275	*SidescanPing*	*Sidescan Ping*
276	*DeviceControl*	*Device Control*
277	*Pulse*	*Pulse*
278	*PulseDetectionControl*	*Pulse Detection Control*
279	*EulerAnglesDelta*	*Euler Angles Delta*
280	*VelocityDelta*	*Velocity Delta*
281	*FuelLevel*	*Fuel Level*
300	*CameraZoom*	*Camera Zoom*
301	*SetThrusterActuation*	*Set Thruster Actuation*
302	*SetServoPosition*	*Set Servo Position*
303	*SetControlSurfaceDeflection*	*Set Control Surface Deflection*
304	*RemoteActionsRequest*	*Remote Actions Request*
305	*RemoteActions*	*Remote Actions*
306	*ButtonEvent*	*Button Event*
307	*LedPattern*	*LED Pattern*
308	*LcdControl*	*LCD Control*
309	*PowerOperation*	*Power Operation*
310	*PowerChannelControl*	*Power Channel Control*
311	*QueryPowerChannelState*	*Query Power Channel State*
312	*PowerChannelState*	*Power Channel State*
313	*LedControl*	*LED Control*
350	*EstimatedState*	*Estimated State*
351	*HomeRef*	*Home Reference Setting*
352	*NavigationStartupPoint*	*Navigation Startup Point*
353	*EstimatedStreamVelocity*	*Estimated Stream Velocity*
354	*IndicatedSpeed*	*Indicated Speed*
355	*TrueSpeed*	*True Speed*
356	*NavigationUncertainty*	*Navigation Uncertainty*
357	*NavigationReset*	*Navigation Reset*
400	*DesiredHeading*	*Desired Heading*
401	*DesiredDepth*	*Desired Depth*
402	*DesiredSpeed*	*Desired Speed*
403	*DesiredRoll*	*Desired Roll*
404	*DesiredPitch*	*Desired Pitch*

405	*DesiredVerticalRate*	*Desired Vertical Rate*
406	*DesiredPath*	*Desired Path*
407	*DesiredControlTorques*	*Desired Control Torques*
408	*DesiredHeadingRate*	*Desired Heading Rate*
409	*DesiredAltitude*	*Desired Altitude*
410	*PathControlState*	*Path Control State*
411	*AllocatedControlTorques*	*Allocated Control Torques*
452	*Goto*	*Goto Maneuver*
453	*Popup*	*Popup Maneuver*
454	*Teleoperation*	*Teleoperation Maneuver*
455	*Loiter*	*Loiter Maneuver*
456	*IdleManeuver*	*Idle Maneuver*
457	*LowLevelControl*	*Low Level Control Maneuver*
458	*Rows*	*Rows Maneuver*
459	*FollowPath*	*Follow Path Maneuver*
460	*PathPoint*	*Path Point*
461	*YoYo*	*Yo-Yo Maneuver*
462	*TeleoperationDone*	*Teleoperation Done*
463	*SubPlan*	*Sub Plan*
464	*StationKeeping*	*Station Keeping*
465	*Elevator*	*Elevator Maneuver*
466	*FollowTrajectory*	*Follow Trajectory*
467	*TrajectoryPoint*	*Trajectory Point*
468	*CustomManeuver*	*Custom Maneuver*
469	*VehicleFormation*	*Vehicle Formation*
470	*VehicleFormationParticipant*	*Vehicle Formation Participant*
472	*StopManeuver*	*Stop Maneuver*
473	*RegisterManeuver*	*Register Maneuver*
474	*ManeuverControlState*	*Maneuver Control State*
475	*FollowSystem*	*Follow System*
500	*WarningSignal*	*Warning Signal*
501	*VehicleState*	*Vehicle State*
502	*VehicleCommand*	*Vehicle Command*
503	*MonitorEntityState*	*Monitor Entity State*
504	*EntityMonitoringState*	*Entity Monitoring State*
505	*OperationalLimits*	*Operational Limits*
506	*GetOperationalLimits*	*Get Operational Limits*
507	*Calibration*	*Calibration*
508	*ControlLoops*	*Control Loops*
540	*VehicleLinks*	*Vehicle Links*
541	*TrexGoal*	*TREX Goal*
551	*Abort*	*Abort*
554	*PlanSpecification*	*Plan Specification*
555	*ManeuverSpecification*	*Maneuver Specification*
556	*ManeuverTransition*	*Maneuver Transition*

557	*EmergencyControl*	*Emergency Control*
558	*EmergencyControlState*	*Emergency Control State*
559	*PlanDB*	*Plan DB*
560	*PlanDBState*	*Plan DB State*
561	*PlanDBInformation*	*Plan DB Information*
562	*PlanControl*	*Plan Control*
563	*PlanControlState*	*Plan Control State*
650	*ReportedState*	*Reported State*
600	*RestartSystem*	*Restart System*
651	*RemoteSensorInfo*	*Remote Sensor Info*
652	*Map*	*Map*
653	*MapFeature*	*Map Feature*
654	*MapPoint*	*MapPoint*
655	*AcousticSystems*	*Acoustic Systems Listing*
700	*VideoData*	*Video Data*
701	*RawImage*	*Raw Image*
702	*CompressedImage*	*Compressed Image*
703	*ImageTxSettings*	*Image Transmission Settings*
704	*NavigationState*	*Navigation State*
705	*BrainsState*	*BRAINS State*
706	*GpsNavData*	*GPS Navigation Data*
707	*LblEstLog*	*LBL Estimator Log*
708	*LblEstimate*	*LBL Beacon Position Estimate*
709	*LblConfigEstimate*	*LBL Configuration Estimate*
710	*RemoteState*	*Remote State*
711	*PiccoloWaypoint*	*Piccolo Waypoint*
712	*ListPiccoloWaypoints*	*List Piccolo Waypoints*
713	*PiccoloWaypointDeleted*	*Piccolo Waypoint Deleted*
714	*PiccoloTrackingState*	*Piccolo Tracking State*
715	*PiccoloPacket*	*Piccolo Packet*
716	*GotoPiccoloWaypoint*	*Goto Piccolo Waypoint*
717	*PiccoloControlConfiguration*	*Piccolo Control Configuration*
718	*GetPiccoloControlConfiguration*	*Get Piccolo Control Configuration*

Capítulo 3

Descrição do Problema

O LSTS tem desenvolvido várias tecnologias no sentido de criar uma rede de veículos autónomos. Uma rede de veículos é uma rede constituída por nós de vários tipos: consolas de operação, sensores, *gateways* de comunicação e veículos autónomos.

Uma das questões mais importantes nestas redes é a comunicação efetiva de dados entre nós da rede, sendo que cada nó pode ser um sistema físico ou um sistema lógico (processo a correr dentro de um veículo).

Neste momento, apesar de muitas tecnologias que possibilitam a criação de redes de veículos autónomos terem já sido desenvolvidas pelo LSTS, o suporte para visualização de dados provenientes de uma rede deste tipo é ainda limitado:

- Cada nó da rede regista um conjunto de dados (ficheiro de log) e é necessário agrupar todos os logs da rede de forma correta *à posteriori*. A este problema chamamos "intercalar logs de vários veículos".
- Não existe forma simples de perceber que nó na rede originou informação e que outros nós receberam informação. Para isso devem ser criadas novas visualizações de dados multi-veículo que permitam perceber eficazmente as ligações que ocorreram na rede.

3.1 - Intercalar dados

A necessidade de intercalar dados ocorreu aquando da análise dos *logs*, na qual se pretendia verificar quem comunicou com quem e quais foram as mensagens trocadas entre os vários sistemas. Esta verificação torna-se muito difícil de efetuar analisando os *logs* individualmente, o que gerou a necessidade de intercalar os dados de vários *logs* num único log. Para intercalar os dados provenientes de vários veículos, é necessário perceber o funcionamento da rede, principalmente o protocolo IMC, e também compreender as classes existentes no projeto, a fim de ser possível desenvolver um programa que seja eficiente e que cumpra os objetivos propostos; o mesmo deve identificar quantos nós existem na rede, identificar o seu id e concatenar todos os *logs* num único para ser possível mostrar as

mensagens de todos eles. As ferramentas sugeridas pelo LSTS foram o Eclipse com acesso a SVN do LSTS[53] que contém o código fonte do IMC em java.

O *Java*[58–62] é uma linguagem de programação desenvolvida por *James Gosling* na *Sun Microsystems* em 1995. Esta é de uso geral, concorrente, baseada em classes, e orientada a objetos, foi projetado para ter o mínimo de dependências de execução possível e permitir criar programas capazes de serem executados em qualquer arquitetura que tenha instalada uma *Java Virtual Machine* (JVM). O Java foi criado com o objetivo de ser simples, orientado a objetos e familiar, robusto e seguro, de ter arquitetura neutra e portável, ser executável em alta performance, ser interpretado e ser dinâmico. Ele tem vindo a evoluir de uma forma que é totalmente compatível com as aplicações existentes, sendo que os compiladores e sistemas são capazes de suportar as várias versões simultaneamente, com compatibilidade completa.

O *Eclipse*[63] é um ambiente multilinguagem de desenvolvimento de software que inclui um IDE e um sistema plug-in extensível e é escrito em Java. Ele é utilizado para desenvolver aplicações em Java e tem plug-ins para incluir Ada, C, C++, COBOL, *Haskell, Perl, PHP, Python, R, Ruby, Scala, Clojure, Groovy* e *Scheme*, entre outros. O Eclipse é distribuído sob a Licença Pública *Eclipse*[64]. O Eclipse SDK é um software livre e *open source*, que permite escrever e contribuir com módulos plug-in para a plataforma *Eclipse*[65–67].

A *Subversion* (SVN)[68–70] é uma aplicação *open source* de controlo de versões, criada em 2000 pela CollabNet Inc. e atualmente faz parte do projeto Apache. A SVN gere as mudanças feitas em arquivos e diretórios ao longo do tempo, e permite recuperar versões antigas dos dados ou examinar o histórico de como os dados foram alterados. Esta pode funcionar em rede e apresenta a vantagem de permitir que várias pessoas sejam capazes de modificar e gerir o mesmo conjunto de dados a partir de vários locais, propiciando a colaboração entre os elementos de uma equipa.

3.2 - Criar grafo de comunicações

A criação de grafos de comunicações é feita utilizando a linguagem DOT e os campos src, src_ent, dst e dst_ent das mensagens. Para a visualização do grafo pode ser usado o *graphviz* ou ferramentas semelhantes, como o *yEd Graph Editor* e o *Microsoft Automatic Graph Layout*, as quais permitem visualizar e editar os mesmos. Esta tem por objetivo a fácil analise das comunicações ocorridas numa missão, recorrendo para o efeito a ferramentas gráficas, de preferência *open source*. Além disso, permite efetuar a sua inspeção a qualquer altura e em qualquer lugar, por investigadores ou engenheiros que não estejam ligados ao LSTS e elimina a necessidade de utilizar o NEPTUS MRA, tornando-se numa mais-valia.

O campo src corresponde ao sistema que cria a mensagem, ao passo que o src_ent é a entidade do sistema que cria a mensagem; do mesmo modo, o dst é o sistema de destino enquanto o dst_ent consiste na entidade de destino. Neste momento, as mensagens não são ainda endereçadas a tarefas específicas, sendo que o campo dst_ent está sempre preenchido com o valor 255, endereçando todas as tarefas que ouçam o tipo de mensagem enviado.

A linguagem DOT foi sugerida pelo LSTS, e consiste na descrição de um grafo num texto simples, convertendo-se numa forma simples de descrever um grafo e fácil de usar, na qual os ficheiros usam a extensão .gv ou .dot. Um grafo é um objeto abstrato, definido por dois tipos de entidades, nós (*nodes*) ou vértices e ramos ou arestas (*edges*), em que os vértices representam entidades como, cidades, números, etc, e as arestas representam, existência das ligações entre nós, o valor da ligação entre nós, etc.

Existem vários programas que leem ficheiros do tipo DOT e transformam-nos na forma gráfica, bem como alguns que realizam cálculos sobre o grafo representado e fornecem uma interface interativa. O Dot permite fazer layouts que distribuem os nós do grafo de forma automática e apelativa (nós espaçados corretamente e transições com nenhuns ou poucos cruzamentos). e fornece uma variedade de formas, estilos e cores adequadas para os diagramas.

Esta linguagem pode descrever grafos dirigidos e não dirigidos. As principais ferramentas de *layout* utilizadas pelo graphviz são o Dot, para grafos dirigidos, e o Neato, para grafos não dirigidos. Estas leem os grafos, calculam os *layouts* e escrevem-nos nos formatos Dot, Gv, *PostScript*, GIF, etc., com o objetivo de criar bons diagramas, com tamanho razoável e bem dimensionado, e isto com etiquetas legíveis em grafos até 100 nós; quando estas forem maiores, é necessário interações adicionais ou técnicas de *layout* perante uma complexidade visual elevada. O Neato é compatível com o Dot pois aceita os mesmos ficheiros de entrada e as opções da linha de comandos.

3.3 - Criar ferramentas de visualização

As ferramentas de visualização[71] a implementar no NEPTUS MRA permitirão o replay das comunicações com *highlight* das ligações e a visualização de informações provenientes dos *logs* processados. A visualização gráfica é uma forma útil e fácil de representar modelos, tendo encontrado muitas aplicações no projeto e análise de redes de comunicação, documentos, e permite estruturá-los de forma estática e dinâmica, levando à necessidade de ferramentas para exibir e manipular grafos.

O replay das comunicações deverá possibilitar a visualização de qual o tipo de mensagens trocadas pelos vários sistemas. Em contra partida o *highlight* deverá permitir diferenciar a frequência com que estas foram trocadas, de forma a facilitar a diferenciação do número de mensagens trocadas pelos vários sistemas.

Capítulo 4

Abordagem do Problema

Neste capítulo descreve-se a abordagem efetuada ao problema relatado no capítulo 3, na qual explicarei como foi efetuado o tratamento dos dados provenientes dos vários veículos, a criação de grafos de comunicações e a criação de ferramentas de visualização. Finalmente irei abordar os problemas individualmente e obter soluções para os mesmos, as quais irão ser validadas no capítulo seguinte.

As ferramentas utilizadas foram as sugeridas pelo LSTS, porque me davam uma resposta rápida e eficaz aos problemas e porque apesar de já saber trabalhar com elas teria uma equipa que me poderia prestar apoio caso se revelasse necessário.

4.1 - Intercalar dados

Os dados provenientes dos veículos são recolhidos no final das missões, usando wi-fi e posteriormente enviados para um servidor Web, onde existe um repositório SVN com todos os dados de missões executadas previamente pelos vários veículos do LSTS.

Para receção, envio e processamento de dados IMC, o Neptus usa uma biblioteca Java chamada IMCJava. Esta biblioteca permite, por exemplo, o carregamento de um log de mensagens através da classe LSFIndex. Essa classe permite ler as mensagens dos *logs*, e tem de utilizar um método *LsfIndex*, o qual procede à *deserialization*; este método converte os *logs* do formato *.lsf* para *.index* e usa como argumentos o caminho do ficheiro *lsfile* e o ficheiro xml com as definições de *serialization*, sendo utilizado da seguinte forma: "*LsfIndex(File lsfile, IMCDefinition defs*)".

Algumas das mensagens mais importantes do sistema utilizadas neste trabalho são: *Announce, TransportBindings, EntityInfo* e *Heartbeat*, de forma a perceber o funcionamento da rede vai ser necessário compreender que as mensagens de *Announce* permitem identificar os sistemas existentes, sendo enviada a descrição do veículo no modo *broadcast* para os outros sistemas; as mensagens *EntityInfo* descrevem as entidades locais existentes; as de *TransportBindings* são criadas quando as *tasks* se ligam às mensagens e permitem às entidades saberem quais as que devem ouvir; por fim, as mensagens *Heartbeat* permitem saber se os sistemas estão visíveis na rede e se estão a comunicar.

Para melhor perceber o funcionamento desta biblioteca, foram feitos vários testes iniciais de introdução à ferramenta. O primeiro desses testes (tarefa desempenhada) foi imprimir todas as mensagens de um certo tipo indicando a hora em que foram registadas.

O teste consistia na criação de um método para ler as mensagens do tipo *LogBookEntry*, para o qual se criou um vetor com todas as mensagens deste tipo e depois utilizou um ciclo para correr esse vetor, onde se leram os campos *htime* (*Timestamp*) com o método "*getTimestampMillis*()" que retorna um *long* e *text* (texto da mensagem) e com o método "*getString* (*String*)" que retorna uma *string*, obtendo o código java da lista 2.

```
1  Vector<Integer> msgs = index.getMessagesOfType("LogBookEntry");
2  for (int i = 0; i < msgs.size();i++) {
3     IMCMessage m = index.getMessage(msgs.get(i));
4     System.out.println(new Date(m.getTimestampMillis())+": "+m.getString("text"));
5  }
6  System.out.println(msgs);
```

Lista 2 - Método para mensagens do tipo *LogBookEntry*.

Nas mensagens *ControlLoops* utilizou-se o mesmo raciocínio mas leu-se o campo *mask* (*Control loop mask*) com o método "*getRawData*(*String*)" que retorna *byte[],* o resultando encontra-se na lista 3.

```
1  Vector<Integer> msgscl = index.getMessagesOfType("ControlLoops");
2  for (int i = 0; i < msgscl.size();i++) {
3     IMCMessage m = index.getMessage(msgscl.get(i));
4     System.out.println(m.getRawData("0xFFFFFFFF"));
5  }
6  System.out.println(msgscl);
7  int msgs3n = index.getNumberOfMessages();
8  System.out.println(msgs3n+": n. mensagens");
```

Lista 3 - Método para mensagens do tipo *ControlLoops*.

Para as mensagens *PlanControlState* utilizou-se um código idêntico ao do *LogBookEntry* com os métodos *getTypeOf*(*String*), *getString*(*String*), e *getTimestamp*().

Também se criou um método *LsfTest* para colocar o código que se tinha desenvolvido no método *main*, como podemos ver na lista 4.

```
1  public static void LsfTest(LsfIndex index) {
2  }
```

Lista 4 - Método *LsfTest*.

Para invocar este método no *main* do *LsfIndex* utilizou-se a linha de código descrita na lista 5.

```
1  LsfTest (index);
```

Lista 5 - método para invocar *LsfIndex* no *main*.

Depois de efetuar estes testes estava preparado para pensar num código que permitisse concatenar todos os *logs* num único e que posteriormente mostrasse todas as mensagens.

Antes de pensar na estrutura do código verificou-se que as mensagens criadas nos *logs* estavam com a ordem que foram geradas, o que implica que estas ficam organizadas no tempo. Isto permitiu pensar em duas abordagens. A primeira abordagem era colar um log ao outro, isto é, juntar um log no final do outro acrescentando os bits de um log no outro.

Contudo, como tínhamos interesse que as mensagens ficassem organizadas na sequência temporal em que foram geradas, esta solução tinha a desvantagem de consumir muita memória para organizar o log final, uma vez que teria de criar um vetor do tamanho do log, pelo que se optou pela seguinte abordagem descrita em pseudocódigo, na lista 6.

```
1   while ( i < logA.NumberOfMessages and j < logB.NumberOfMessages ) do
2   if (logA[i].timestamp > logB[j].timestamp and i < logA.NumberOfMessages) then
3       resultado[k++] = logB[j++];
4   else
5   if (logA[i].timestamp < logB[j].timestamp and j < logA.NumberOfMessages) then
6       resultado[k++] = logA[i++];
7     else
8   if (logA[i].timestamp > logB[j].timestamp and i = logA.NumberOfMessages and j < logA.NumberOfMessages) then
9     while (j < logA.NumberOfMessages) do
10      resultado[k++] = logB[j++];
11  else
12  if (logA[i].timestamp < logB[j].timestamp and j = logA.NumberOfMessages and i < logA.NumberOfMessages) then
13    while (i < logA.NumberOfMessages) do
14      resultado[k++] = logA[i++];
15  end
```

Lista 6 - Descrição do LsfLogMerge.

Como podemos verificar, enquanto os *logs* tiverem mensagens vamos verificar o tempo em que foram geradas e colocar no novo *log* as mensagens mais antigas. É de salientar que também se verifica se algum dos *logs* já não tem mais mensagens, pois se não fizéssemos essa verificação o programa poderia parar nessa altura.

Quando um dos logs chega ao fim, vamos escrever só as mensagens do outro *log* até este chegar igualmente ao final. Esse método denomina-se "*public static void writestream(String OutFile1, LsfIndex index1, LsfIndex index2)*", e pode ser consultado a versão completa do código efetuado na SVN do LSTS como *LsfLogMerge.java*. Esta estrutura pode ser verificada no diagrama de classes na figura 17.

Para identificar quantos nós existem na rede e qual o seu id criei o método "*public static void NString(LsfIndex index, String stList)*". Este método permite identificar os nós da rede e verificar a origem das mensagens. Para isso procede à leitura do campo src, que corresponde ao sistema origem e que deve ser definido ao invocar o método, pois este também aceita como entrada o src_ent, o qual corresponde ao módulo, dentro do sistema, que originou a mensagem o que nos irá permitir verificar quantos módulos enviaram mensagens. Este código também se encontra na SVN do LSTS como *LsfLogMerge.java*.

4.2 - Criar grafo de comunicações

Na criação de grafos de comunicações efetuei três abordagens diferentes, nas quais utilizei a classe *LinkedHashMap* para criar mapas de *entitiesToTasks*, *msgConsumers*, e *msgsGenerated*. O mapa das *entitiesToTasks*, como o próprio nome indica, vai criar um *link* de uma entidade para uma tarefa; por sua vez, as *msgConsumers* são as mensagens consumidas, isto é, vão indicar o destinatário das mensagens, em que um id vai ligar a um vetor de consumidores; por último, as *msgsGenerated* vão indicar as mensagens geradas, onde um sistema de origem corresponde a um vetor de módulos dentro desse sistema.

A primeira abordagem foi bastante simplista, pois foram criados mapas utilizando os campos src e *src_ent*, ligados aos campos *dst* e *dst_ent* das mensagens. Todavia, viemos a

constatar que esta abordagem era uma solução incompleta, pois os sistemas que compõem a rede enviam as mensagens no modo *broadcast*, em que o *dst* das mensagens é igual a 255; esta perceção permitiu-me ter uma melhor compreensão do funcionamento da rede e passar à abordagem seguinte.

Na segunda abordagem foi importante a ajuda de uma API em java criada por *Laszlo Szathmary*[72], em que é necessário ter o *graphviz* instalado; numa primeira fase foi preciso definir o caminho da instalação do dot do *graphviz* que, mais tarde, o Doutor José Pinto otimizou para a deteção do sistema operativo, removendo assim a necessidade de configuração. Esta API efetua uma otimização automática e aceitável dos grafos até 100 nós e também permite criar ou converter grafos da linguagem DOT para gv, gif, dot, fig, pdf, ps, svg, png, etc., o que me possibilitou criar os primeiros grafos com a linguagem DOT. O código da API encontra-se na SVN do LSTS como *GraphViz.java*.

Por último, alterou-se o código dos mapas do *entitiesToTasks* para as mensagens do tipo *EntityInfo* que descrevem as entidades; para *msgConsumers* utilizei as mensagens *TransportBindings* que são criadas aquando das tarefas e fornecem o nome do consumidor e o id das mensagens a serem ouvidas; o *msgsGenerated* associa a cada id de tarefa src_ent os tipos de mensagens que foram produzidas por essa tarefa. Os grafos gerados podem ser visualizados com as ferramentas *graphviz*, o *yEd Graph Editor* e o *Microsoft Automatic Graph Layout*.

4.3 - Criar ferramentas de visualização

Na criação de ferramentas de visualização, a API já existente em Java desempenhou um papel fundamental. Com ela foi possível desenvolver (com ajuda de investigadores do LSTS) o método *showGraph*(*String*) e adicionar uma nova opção "View communications graph" ao Neptus MRA. Esta opção permite, através de um clique do rato nas mensagens que se pretendem visualizar, o diagrama de comunicações correspondente. Além disso existe a opção de salvar esse diagrama clicando com o botão do lado direito do rato no grafo e escolhendo um formato e nome do ficheiro de saída. A visualização de informações provenientes dos logs processados pode ser efetuada de duas formas: a primeira é através do NEPTUS MRA, que nos permite ver o grafo do tipo mensagens selecionado e as mensagens transmitidas entre os vários sistemas, e a segunda é executando a classe *IMCGraph.java*, que permite a visualização das mensagens utilizando "*g.showGraph*(*String*)" e a criação de ficheiros "*g.writeGraphToFile*(*String, String*)" nos formatos descritos em 4.3. É de ter em consideração que quando se tenta visualizar um número muito elevado de nós em simultâneo, pode levar ao consumo de muitos recursos e a ter de cancelar a operação, sendo que o grafo final pode não ser legível. Por essa razão ficou também disponível a opção de criar ficheiros GraphViz (.gv), o que permite utilizar ferramentas como o *graphviz*, *Gephi*, *yEd Graph Editor*, etc. para otimizar e visualizar os grafos.

Também foi implementado o *highlight* das comunicações na qual se utilizou a classe *colormap* do NEPTUS, a qual vai criar uma cor em função do argumento de entrada. Este tem que ser um valor entre zero e um e corresponde ao resultado do quociente entre o número de vezes que a mensagem foi enviada e o número total de *edges* criados.

Para a visualização do tipo de mensagens trocadas entre os veículos e as várias consolas desenvolveu-se a classe “*generateSystemsGraph*” a qual mostra um grafo a indicar quais as mensagens trocadas. A implementação dos *highlight* neste grafo não tinha muito interesse pois o que se pretendia visualizar era quais as mensagens que são trocadas entre sistemas, e para ter informação mais detalhada deve-se recorrer á análise individual de cada mensagem.

Na figura 17 encontra-se o diagrama de classes para uma melhor compreensão da abordagem do problema.

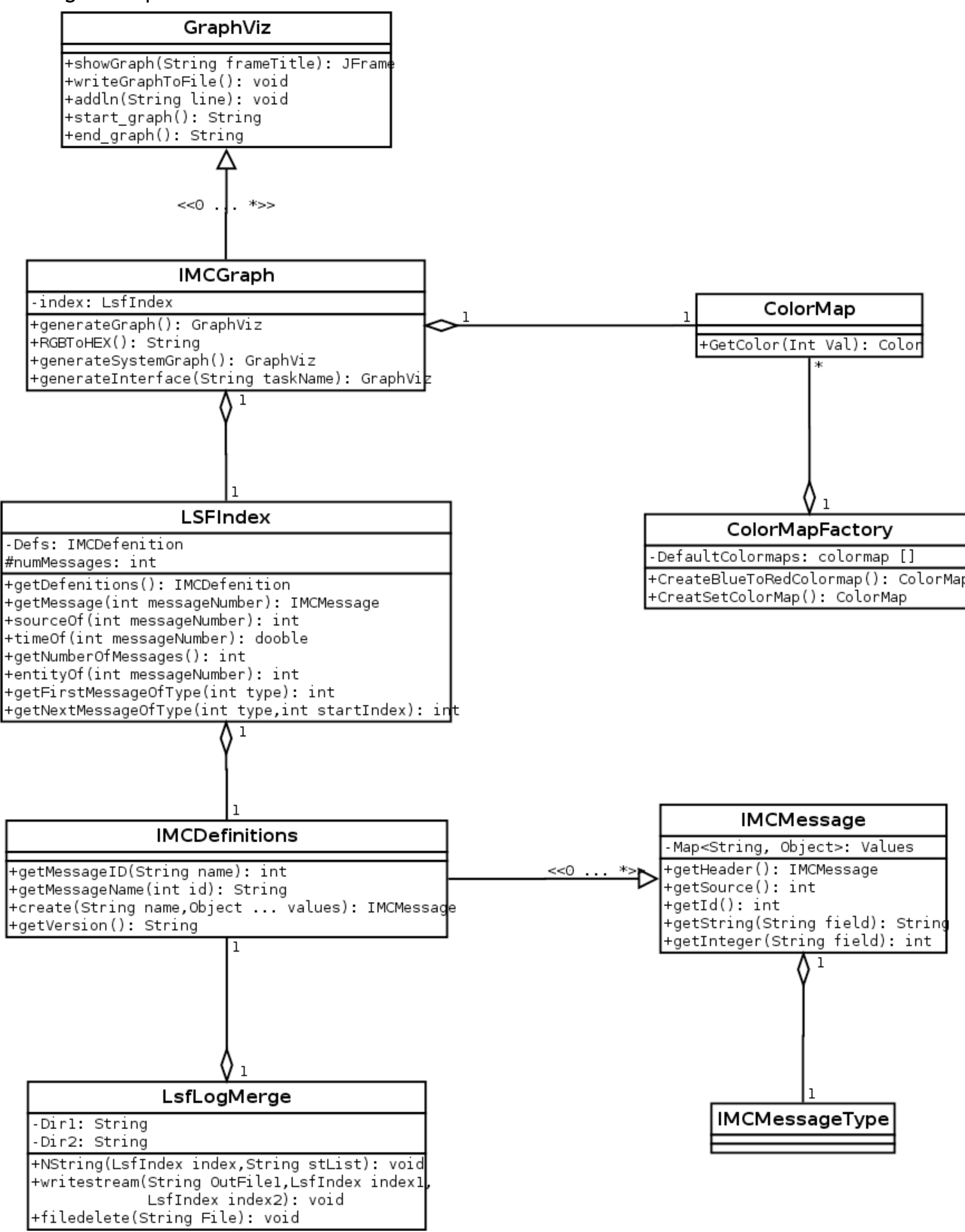


Figura 17 - Diagrama de classes.

Capítulo 5

Resultados

No presente capítulo serão apresentados os resultados obtidos para o problema descrito no capítulo anterior. No geral, os dados obtidos estabelecem uma relação entre a quantidade de nós existentes num grafo e a sua análise, evidenciando que quantos mais nós existirem num grafo mais difícil se torna a criação e análise dos mesmos.

A arquitetura implementada encontra-se descrita num modelo UML, o qual é constituído por um diagrama de classes (representado na figura 17) e indica: os principais métodos e campos públicos de cada classe, os relacionamentos entre classes e caracteriza as associações existentes. Este diagrama exclui as classes e métodos não utilizadas neste projeto mas pertencentes as librarias do LSTS, de forma e serem obtidos diagramas simples e fáceis de compreender.

5.1 - Intercalar dados

Da análise do algoritmo desenvolvido no contexto desta dissertação, a classe *LsfLogMerge*, depreendeu-se que tem uma complexidade O(n,m)=n+m. Por sua vez, o algoritmo com que vamos comparar os resultados obtidos é a classe *LSFMerger*, a qual se encontra na SVN do LSTS e possui complexidade O(n,m)=log(n+m). Dos dados apresentados podemos concluir que o algoritmo que eu desenvolvi é muito mais lento, como teremos oportunidade de comprovar no teste que se efetuou.

O teste mencionado foi efetuado no IDE Eclipse, com o desenvolvimento da classe *MergeBattle* em que se utilizou a função "*System.currentTimeMillis()*" antes e depois de chamar as classes correspondentes ao teste a efetuar, como podemos verificar no código na lista 7.

```
1   package pt.up.fe.dceg.neptus.imc.merge;
2   import java.io.File;
3   import pt.up.fe.dceg.neptus.imc.lsf.LsfIndex;
4   public class MergeBattle {
5
6       public static void main(String[] args) throws Exception {
7           String dir_input1, dir_input2, dir_output_ribcar, dir_output_rsilva;
8
9           dir_input1 = "/home/ribcar/teste1/";
10          dir_input2 = "/home/ribcar/teste2/";
11
12          dir_output_ribcar = "/home/ribcar/Desktop/logs/ribcar/";
13          dir_output_rsilva = "/home/ribcar/Desktop/logs/rsilva/";
14
15          long startMillis = System.currentTimeMillis();
16
17          System.out.println(startMillis);
18
19          LsfIndex index1 = new LsfIndex(new File(dir_input1+"Data.lsf"), null);
20          LsfIndex index2 = new LsfIndex(new File(dir_input1+"Data.lsf"), null);
21
22          new File(dir_output_ribcar).mkdirs();
23          new File(dir_output_rsilva).mkdirs();
24
25          LsfLogMerge.writestream(dir_output_ribcar+"Data.lsf", index1, index2);
26
27          System.out.println(System.currentTimeMillis()+", "+(System.currentTimeMillis() - startMillis));
28
29          startMillis = System.currentTimeMillis();
30
31          System.out.println(startMillis);
32
33          LSFMerger.main(new String[] {"-o", dir_output_rsilva, dir_input1, dir_input2});
34
35          System.out.println(System.currentTimeMillis()+", "+(System.currentTimeMillis() - startMillis));
36
37      }
38
39  }
```

Lista 7 - Classe MergeBatlle

O eclipse foi configurado para escrever o output num ficheiro .txt, de forma a viabilizar a posterior análise dos resultados. Nessa análise constatamos que o tempo necessário para intercalar os logs foi de 94625 milissegundos para a classe desenvolvida e de 12375 milissegundos para a classe LSFMerger; este resultado já era esperado em virtude da complexidade dos algoritmos.

As vantagens da classe *LsfLogMerge*, em comparação com a *LSFMerger*, são que esta utiliza uma quantidade muito reduzida de memória, mas em contrapartida utiliza o disco rígido, que são baratos e tem capacidades de armazenamento na ordem dos *terabyte*, em que as desvantagem são termos um sistema mais lento por não alocar os dados na memória RAM que permitem um acesso mais rápido do que a partir dos discos, mas fica limitado pela capacidade na ordem dos gigabyte e o preço dos módulos de memória RAM.

5.2 - Grafo de comunicações

O grafo de comunicações para as mensagens *Announce*, selecionado aleatoriamente, pode ser consultado na figura 18.

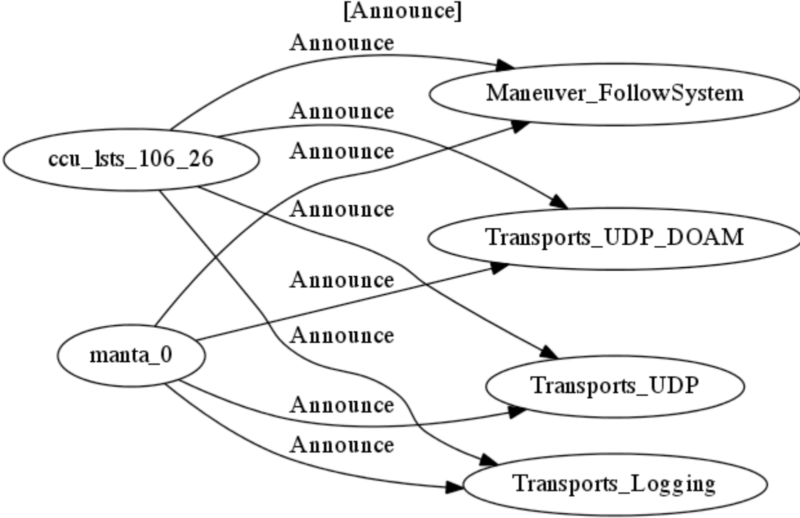


Figura 18 - Grafo de comunicações para as mensagens *Announce*.

Como podemos inferir pela análise da figura 18, este grafo é bastante legível e informativo, o que permite afirmar que a API em java do *GraphViz* promove um bom trabalho, sendo portanto, em última análise, uma boa escolha. É de notar que o espaçamento entre os nós é razoável (nós espaçados) e que as transições não se sobrepõem como no grafo gerado a partir de todas as comunicações, o qual pode ser consultado na figura 19.

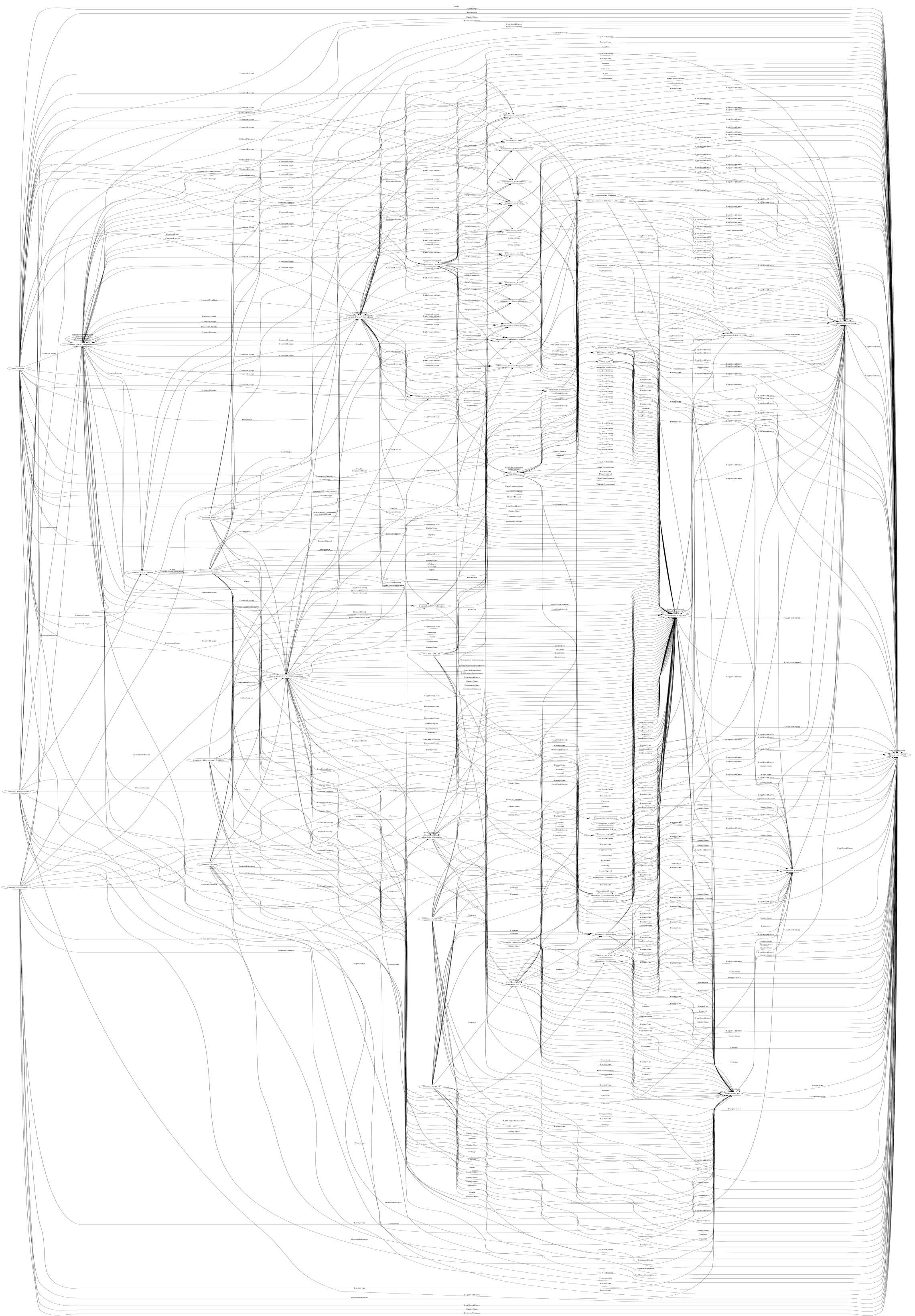


Figura 19 - Grafo de comunicações de todas as mensagens do log.

A análise da figura 19 sugere que, ao contrário do observado para o grafo de comunicações para as mensagens *Announce*, a análise do grafo de comunicações que engloba todas as mensagens enviadas não é muito legível e existe sobreposição das ligações, o que já era esperado.

Com estes resultados podemos concluir que, como era esperado, usando a API Java os grafos gerados só são legíveis para um baixo número de nós. Assim, para um número elevado de nós são necessárias ferramentas de análise, como vamos demonstrar mais á frente.

5.3 - Ferramentas de visualização

A visualização pode ser efetuada através do Neptus MRA(figura 20), executando a *class IMCGraph.java*, através da linha de comandos ou do eclipse, ou outra ferramenta capaz de compilar Java (figura 24), ou ainda através da criação de um ficheiro dot, gv, png, etc., utilizando ferramentas de visualização como o *graphviz, Gephi, yEd Graph Editor, dot2tex*, etc.(figuras 21).

Em relação às ferramentas de visualização, encontram-se expostos nas figuras 20 e 21, respetivamente, os resultados finais obtidos no NEPTUS, referentes aos períodos pré e pós implementação dos *highlight;* na figura 20 também se pode visualizar a opção de "*View communications graph*", utilizada para gerar os grafos.

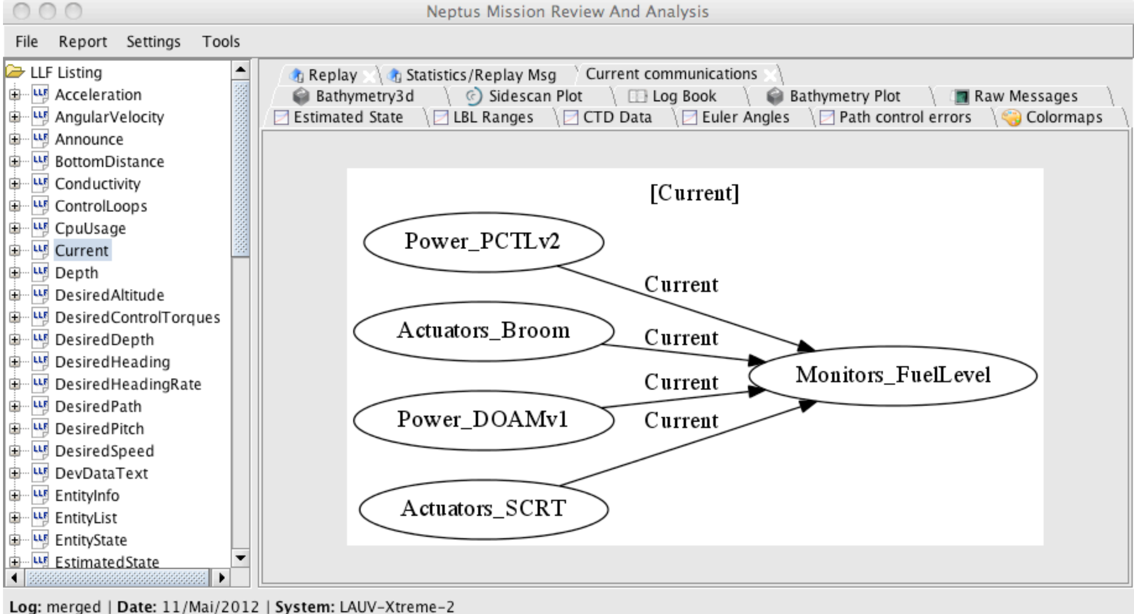


Figura 20 - Resultado obtido no Neptus, anteriormente à implementação dos highlights.

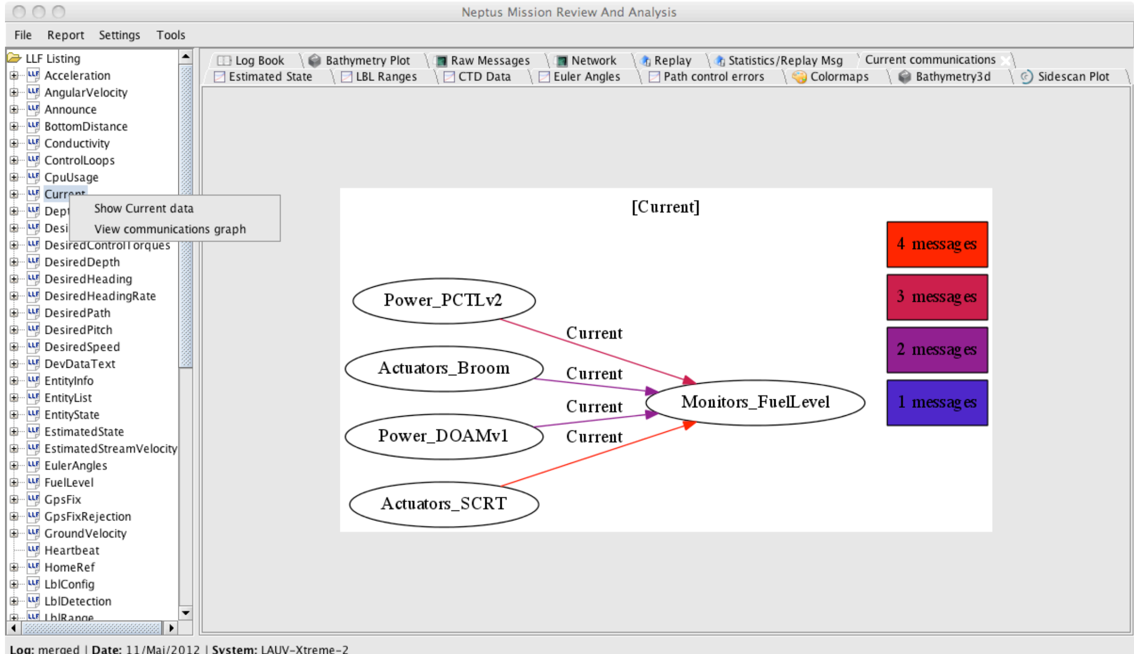


Figura 21 - Resultado obtido no Neptus, com highlights.

Na figura 20 e 21 podemos ver a evolução do projeto com a implementação dos highlights. Os grafos são criados através da classe *IMCGraph.java*, que se encontra na biblioteca IMCJava, a qual tem bibliotecas partilhadas com o Neptus; assim sendo, todas as vezes que se atualiza a classe *IMCGraph.java* é necessário atualizar a biblioteca no Neptus.

Na figura 22 encontra-se representada a visualização de todas as comunicações, obtido no *gephi* a partir do ficheiro .dot do grafo da figura 19, com um *template* predefinido, alterando apenas as cores das legendas.

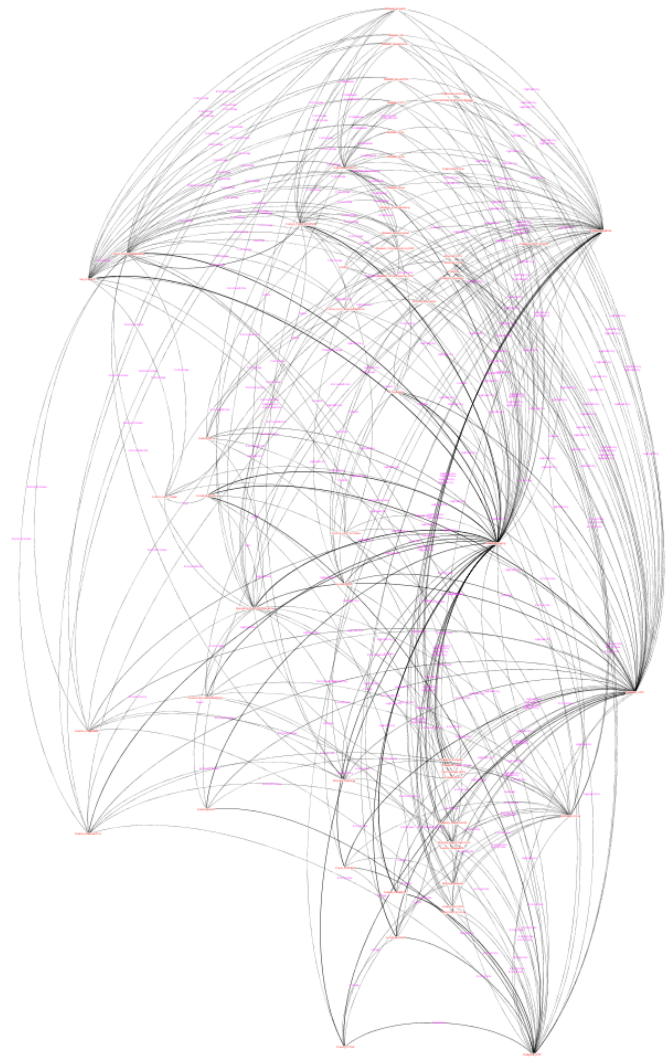


Figura 22 - Grafo de comunicações editado no *gephi*.

Se compararmos o grafo apresentado na figura 19 com este que é apresentado na figura 22, é possível constatar que este último proporciona uma maior facilidade no seguimento das ligações entre os nós, fato que demonstra que com este tipo de ferramentas é possível visualizar e otimizar os grafos.

A figura 23, foi recortada e ampliada da figura 22 e a sua observação atenta facilmente corrobora a ideia anteriormente expressa.

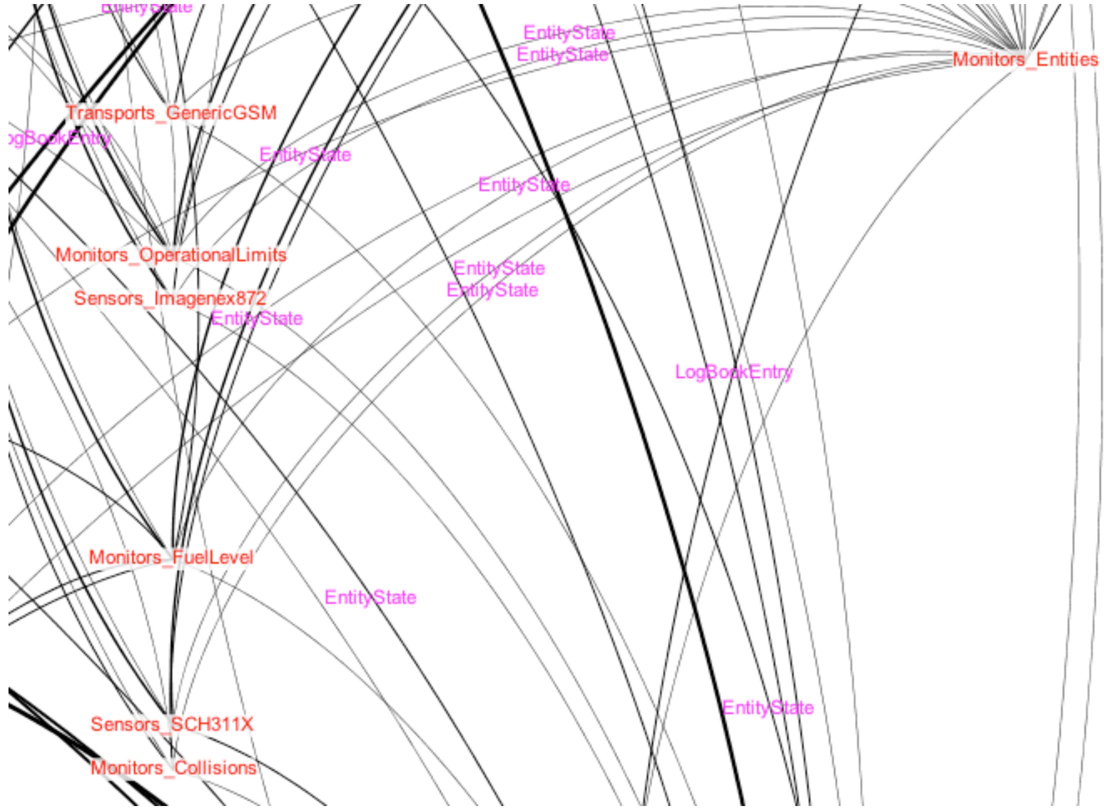


Figura 23 - Recorte da figura 21.

Os resultados obtidos após a implementação dos *highlights* para todas as comunicações encontram-se expressos na figura 24.

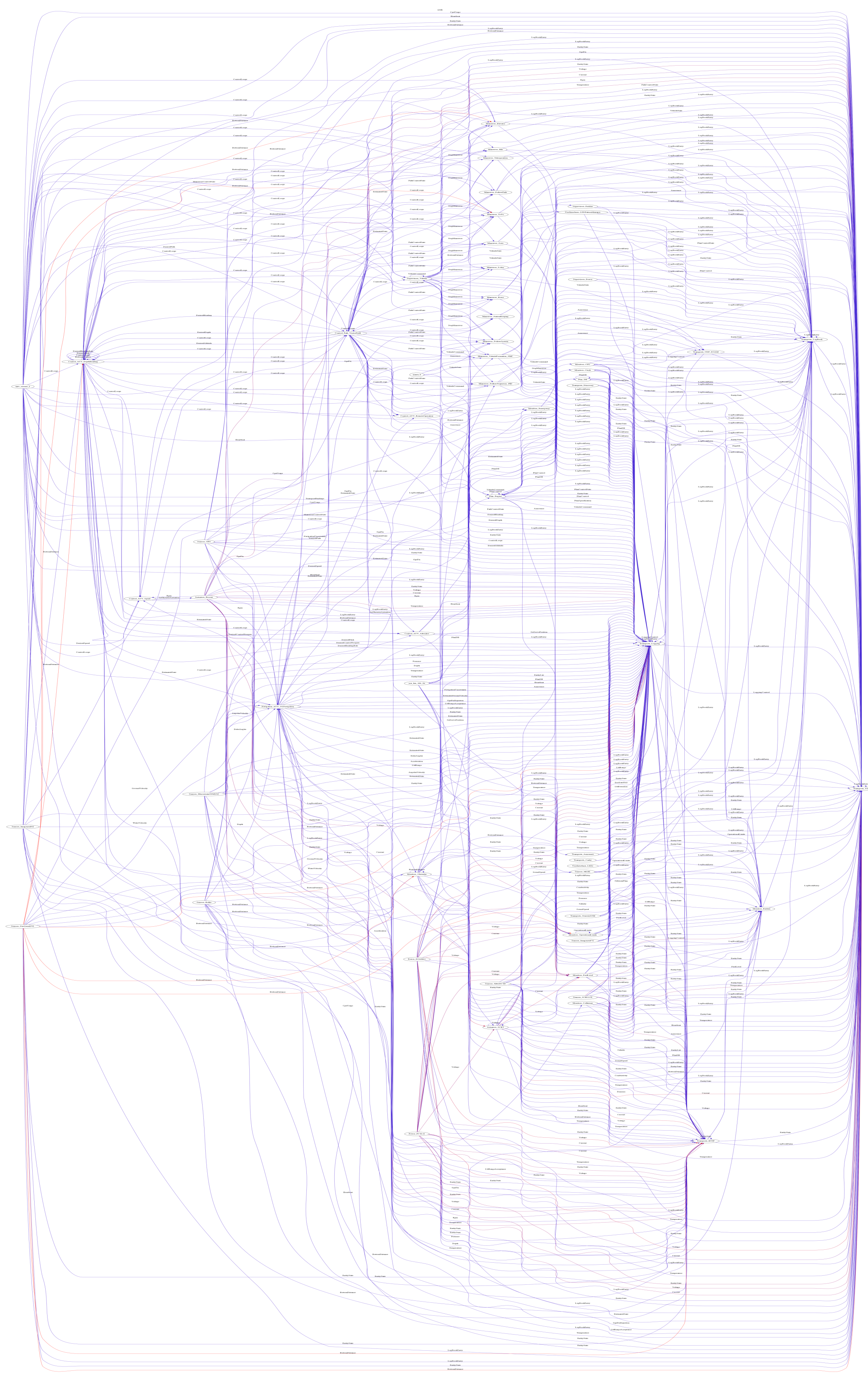


Figura 24 - Grafo de comunicações para todas as mensagens, após implementação dos *highlights*.

As mensagens com *highlights* estão definidas para serem criadas com um mapa de cores que vai do azul (código #0000FF em hexadecimal) para o vermelho (código #FF0000 em hexadecimal), em que o azul corresponde a ter zero ligações e o vermelho corresponde a ter o número máximo de ligações.

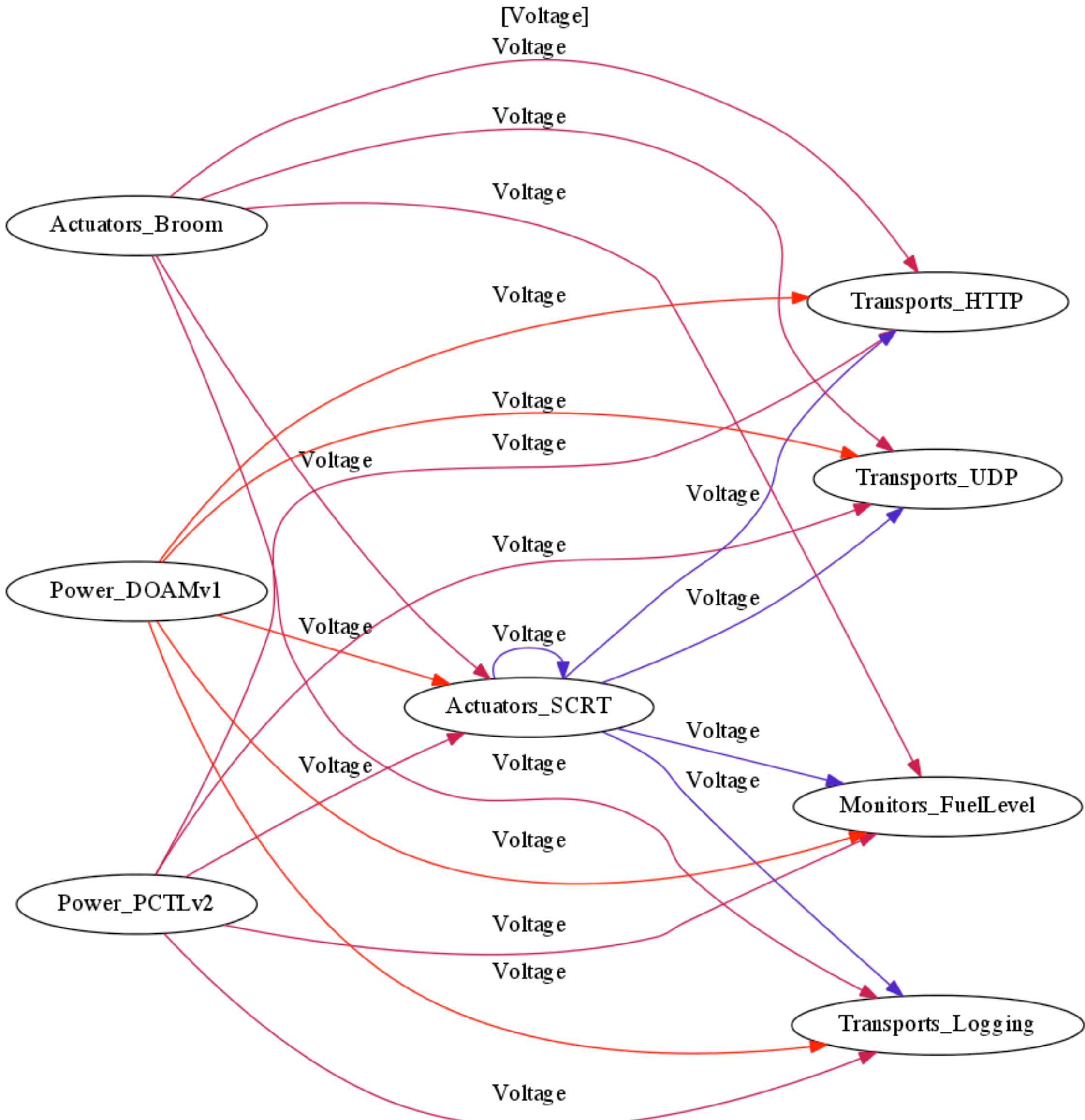


Figura 25 - Grafo de comunicações para as mensagens *Voltage*.

Da análise da figura 25 podemos concluir que o número de mensagens trocadas entre *Power_DOAMv1* e o *Actuators_SCRT* tem uma maior frequência, uma vez que aparecem a vermelho; por oposição, as mensagens trocadas entre *Actuators_SCRT* e *Transports_UDP* são as menos frequentes porque aparecem a uma cor muito próxima do azul. Para validar esta conclusão encontra-se seguidamente exposto para análise o *log* de *debugging* gerado pelo eclipse, ou o ficheiro .dot, no qual podemos constatar que:

Power_DOAMv1 -> Actuators_SCRT [label="Voltage" color="#FF0000"];

Actuators_SCRT -> Transports_UDP [label="Voltage" color="#3F00BF"];

Daqui facilmente verificamos que a cor *#FF0000* corresponde ao vermelho e que a cor *#3F00BF* corresponde a uma cor próxima do azul e através de uma análise mais aprofundada,

com a ajuda do eclipse, vamos poder verificar que o número máximo de mensagens trocadas foi quatro, o que gerou a cor *#FF0000,* e o número mínimo foi uma, com a cor *#3F00BF*.

Caso pretendamos visualizar quais foram os veículos e sistemas que estiveram ativos durante a missão, é necessário ver o grafo das mensagens *He*artbeat. Como podemos verificar no log da figura 26, só estiveram ativos o veículo lauv_xtreme_2 e o sistema ccu_lsts_106_26, como.

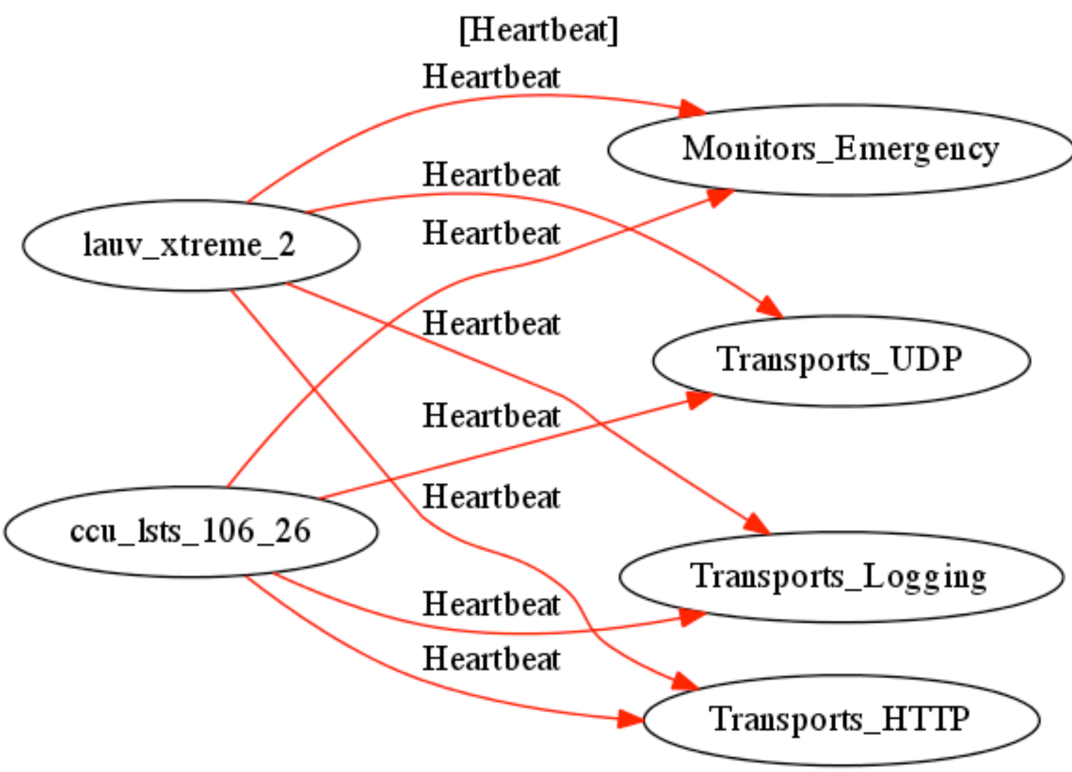


Figura 26 - Grafo de comunicações para as mensagens *Heartbeat*.

Com o decorrer do projeto verificou-se que os grafos necessitavam de uma legenda, para uma fácil perceção do que aconteceu como podemos ver na figura 27 e figura 28. Para criar esta legenda recorreu-se a utilização de um *subgraph* no canto inferior direito do grafo que pode conter no máximo cinco cores dependendo do numero de *edges*, se for superior a cinco é calculado dividindo o código de cores em cinco partes.

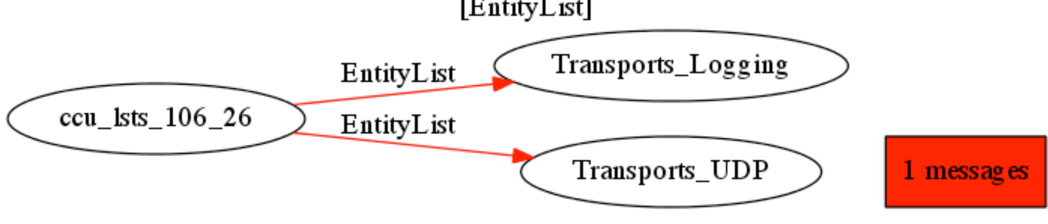


Figura 27 - Grafo de comunicações *EntityList*, com legenda.

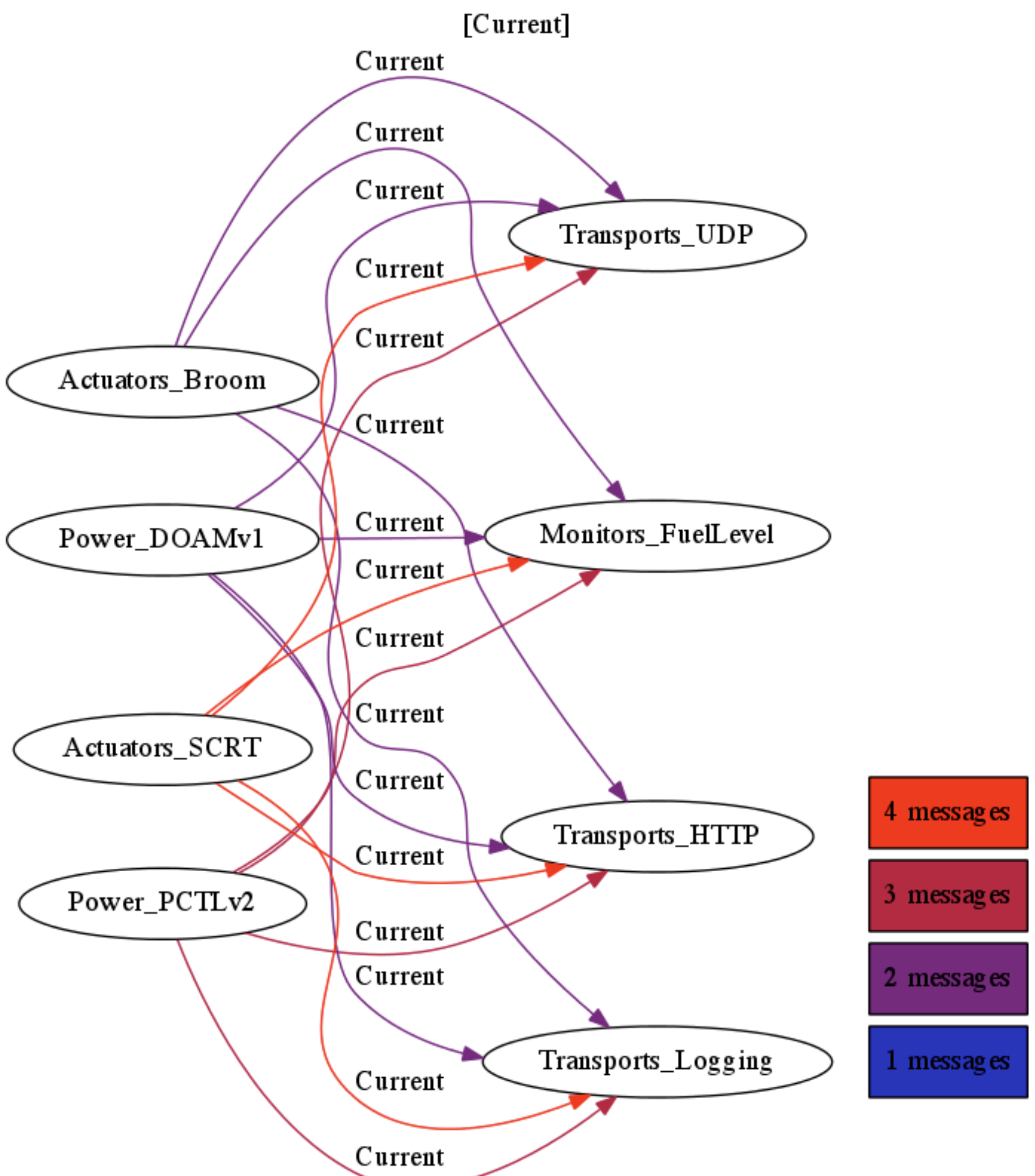


Figura 28 - Grafo de comunicações *Current*, com legenda.

Como podemos verificar a figura 27, as mensagens do tipo *EntityList* são transmitidas apenas uma vez enquanto as mensagens do tipo *Current* são transmitidas no máximo quatro vezes (figura 28).

Se compararmos a figura 27 com a figura 29 vamos verificar que em vez das três mensagens que deveríamos ter apenas aparecem duas. Isto deve-se ao *Daemon* não estar a gerar id, fato que teve de ser reportado, resultando na necessidade de criar uma rotina que verifica quando este tipo de mensagens aparece e que as vai colocar na *hashmap* para poderem ser criadas no grafo, pois estas estavam a ser descartadas.

- GroundVelocity
- Heartbeat
- HomeRef
- LblConfig
- LblDetection
- LblRange
- LblRangeAcceptance
- LogBookEntry
- LoggingControl
- ManeuverControlState
- NavigationStartupPoint
- NavigationUncertainty
- OperationalLimits
- PathControlState
- PlanControl
- PlanControlState
- PlanDB
- PlanSpecification
- Pressure
- Rpm
- Salinity
- SetServoPosition
- SetThrusterActuation
- SidescanPing
- SoundSpeed
- StopManeuver
- Temperature
- TransportBindings

time	src	src_ent	dst	dst_ent	consumer	message_id
09:21:45.312	lauv-xtreme-2	System	*	*	Maneuver.Loiter	472
09:21:45.312	lauv-xtreme-2	System	*	*	Maneuver.Rows	472
09:21:45.313	lauv-xtreme-2	System	*	*	Maneuver.StationKeeping	472
09:21:45.313	lauv-xtreme-2	System	*	*	Maneuver.Teleoperation	472
09:21:45.313	lauv-xtreme-2	System	*	*	Maneuver.VehicleFormation.SMC	472
09:21:45.315	lauv-xtreme-2	System	*	*	Maneuver.YoYo	472
09:21:45.383	lauv-xtreme-2	System	*	*	Transports.Logging	472
09:21:45.343	lauv-xtreme-2	System	*	*	Supervisors.Vehicle	473
09:21:45.343	lauv-xtreme-2	System	*	*	Supervisors.Vehicle	474
09:21:45.383	lauv-xtreme-2	System	*	*	Transports.Logging	474
09:21:45.310	lauv-xtreme-2	System	*	*	Maneuver.FollowSystem	475
09:21:45.298	lauv-xtreme-2	System	*	*	Daemon	5
09:21:45.383	lauv-xtreme-2	System	*	*	Transports.Logging	5
09:21:45.386	lauv-xtreme-2	System	*	*	Transports.UDP	5
09:21:45.383	lauv-xtreme-2	System	*	*	Transports.Logging	50
09:21:45.386	lauv-xtreme-2	System	*	*	Transports.UDP	50
09:21:45.327	lauv-xtreme-2	System	*	*	Plan.Engine	501
09:21:45.342	lauv-xtreme-2	System	*	*	Supervisors.Entities	501
09:21:45.383	lauv-xtreme-2	System	*	*	Transports.Logging	501
09:21:45.386	lauv-xtreme-2	System	*	*	Transports.UDP	501
09:21:45.388	lauv-xtreme-2	System	*	*	Transports.UDP/DOAM	501
09:21:45.388	lauv-xtreme-2	System	*	*	UserInterfaces.LEDPatternManager	501
09:21:45.327	lauv-xtreme-2	System	*	*	Plan.Engine	502
09:21:45.343	lauv-xtreme-2	System	*	*	Supervisors.Vehicle	502
09:21:45.383	lauv-xtreme-2	System	*	*	Transports.Logging	502
09:21:45.318	lauv-xtreme-2	System	*	*	Monitors.Entities	503
09:21:45.343	lauv-xtreme-2	System	*	*	Supervisors.Vehicle	504

Figura 29 - Análise no MRA dos *TransportBindings*.

Na figura 29 podemos confirmar que temos três mensagens com id 5, pelo que podemos concluir que foram enviadas 3 mensagens do tipo *EntityList*. Também houve indicação para substituir os *Transports.UDP* pelas *CCUs*, em que se criou uma rotina muito idêntica à do *Daemon*, o que facilitou muito o trabalho. Estas alterações podem ser visualizadas na figura 30.

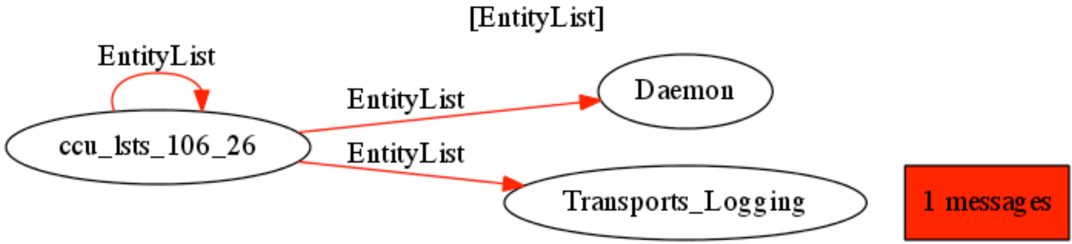


Figura 30 - Grafo de comunicações EntityList final.

Para uma análise ainda mais simples pensou-se em implementar um grafo que identifica quais as mensagens transmitidas entre os vários sistemas. Essa visualização pode ser efetuada através da janela *Network* do Neptus MRA (figura 31), se a imagem for maior do que a janela e possível utilizar as *scrollbars*, ou através da classe *IMCGraph* (figura 32).

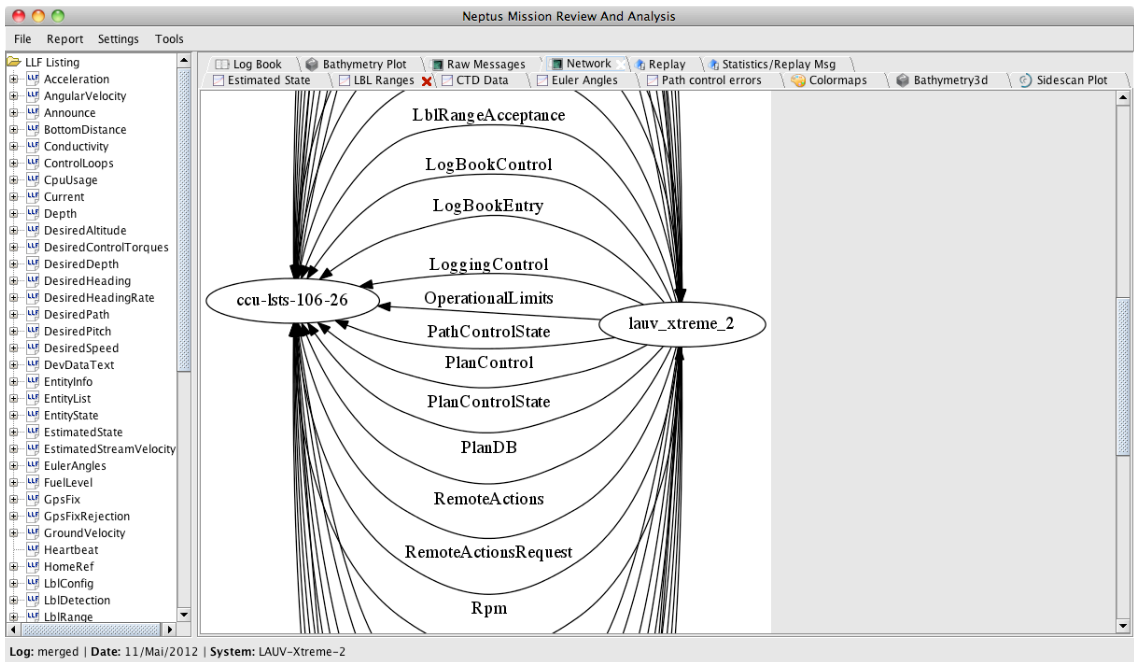


Figura 31 - Grafo de comunicações da rede no Neptus MRA.

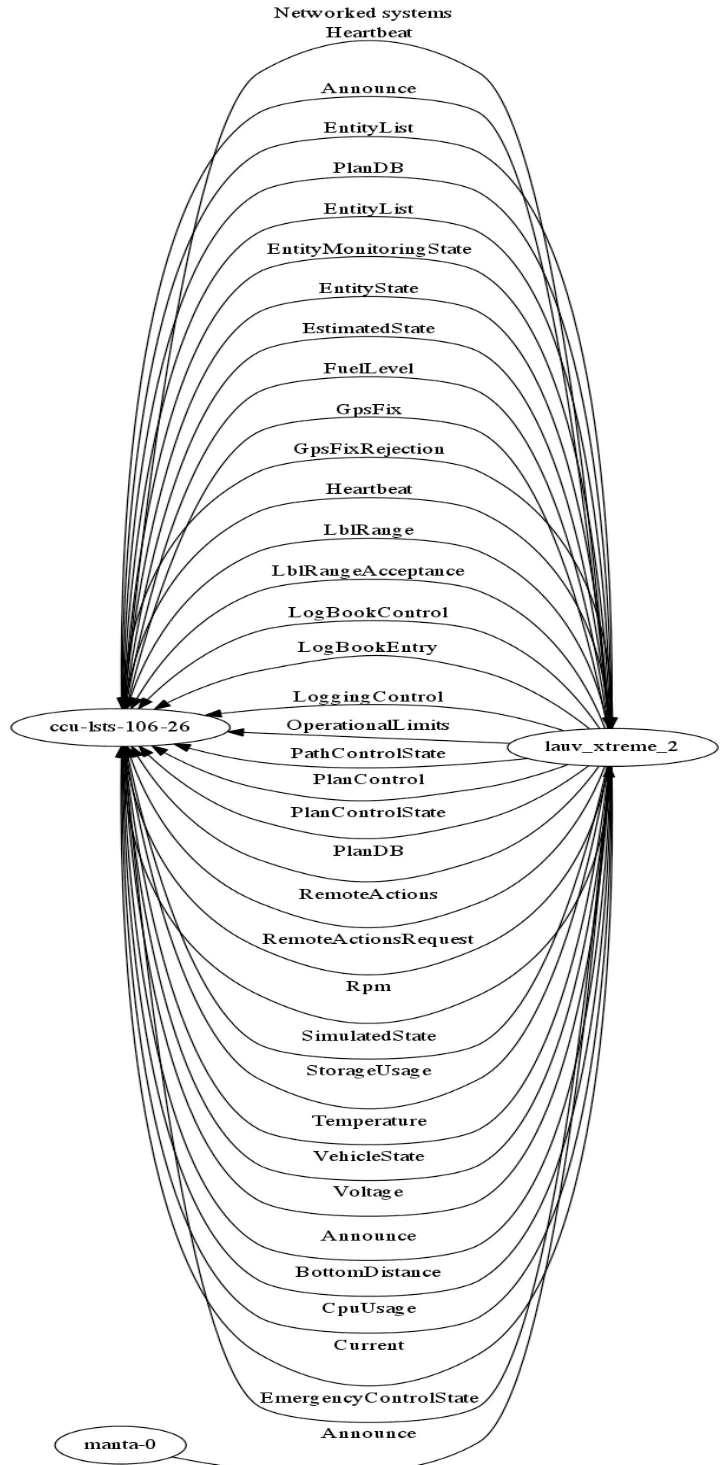


Figura 32 - Grafo de comunicações da rede a partir da classe *IMCgraph*.

Capítulo 6

Conclusão

6.1 – Visualização de dados numa rede de veículos

A utilização da linguagem DOT para efetuar a visualização de dados, através de grafos, numa rede de veículos revelou-se uma boa solução, uma vez que permite analisar os *logs* duma missão de uma forma gráfica e muito intuitiva. As visualizações desenvolvidas são modulares no sentido de poderem ser usadas para diferentes fins e de vários modos. Por exemplo, podem ser acedidas através do Neptus MRA, através da linha de comandos, eclipse, e de outras ferramentas como o *graphviz, Gephi, yEd Graph Editor, dot2tex*, etc.

A utilização das bibliotecas já existentes (Neptus MRA, IMCJava, JgraphViz) tornou-se uma mais-valia para o desenvolvimento deste projeto, permitindo um desenvolvimento muito rápido do mesmo. O único inconveniente foi o tempo de aprendizagem despendido para as compreender, durante o qual pude constatar, e reafirmo a ideia anterior, que a utilização de bibliotecas num projeto é uma mais-valia porque depois de as compreender e saber manipular conseguimos criar ferramentas a uma velocidade incrível, além de que facilitam o trabalho em equipa.

A implementação dos *highlights*, apesar de não ser um requisito inicial, proporcionou uma melhor compreensão e análise dos grafos, já que permitiu saber quais foram as mensagens enviadas com maior frequência, e com a implementação da legenda conseguimos saber o número de mensagens trocadas.

Esta aplicação tem a vantagem de permitir uma fácil compreensão das mensagens trocadas na rede mas como se trata de uma aplicação estática, tem a desvantagem, de na criação de visualizações dinâmicas ter um custo muito elevado em recursos, o que a torna a sua implementação inviável, isto acontece porque é necessário criar uma nova visualização e para esta é necessário calcular de novo todos os nós e ramos, e com o aumento destes aumenta o tempo necessário para efetuar o cálculo, e aparece a necessidade de controlar a taxa de refrescamento do grafo em que esta não pode ser inferior ao tempo de cálculo do mesmo.

As visualizações desenvolvidas neste trabalho foram já utilizadas inúmeras vezes pelos investigadores do LSTS para depurar as comunicações que ocorreram durante uma missão tendo permitido detectar e resolver falhas em tempo de missão (identificação dos subsistemas que estão a enviar uma mensagem de um certo tipo e para que subsistemas).

A ferramenta tem sido utilizada também para documentar as interações dos módulos do software de bordo do LSTS de uma forma automática e com resultados adequados para sua inclusão em artigos e/ou manuais técnicos.

6.2 - Trabalho futuro

O projeto tem muitas potencialidades para a sua continuação e desenvolvimento em trabalhos futuros. Uma possibilidade seria adaptar o projeto de forma a receber os dados em tempo real e representar os grafos de comunicações de uma forma dinâmica; neste caso, será necessário fazer um estudo da arquitetura do Neptus bem como dos requisitos computacionais necessários.

É possível implementar um sistema que altere a cor e o formato dos nós dependendo do sistema a que cada um corresponde, a atualização do grafo em tempo real, identificar mensagens perdidas e o agrupamento de mensagens (nas ligações do grafo).

O mecanismo de junção de logs (merge) deverá ser testado e eventualmente optimizado para logs provenientes de mais de dois sistemas, apesar de o algoritmo actual suportar a mesma funcionalidade através de uma execução iterativa.

Pode-se ainda estender o projeto, utilizando ferramentas que permitam uma inspeção e análise dos grafos de forma mais intuitiva. Como exemplo, poderia ser utilizada a biblioteca Ubigraph [73] que permite visualizar um grafo alterado dinamicamente em 3D.

Referências

[1] J. Pinto, P. S. Dias, E. R. B. Marques, P. Calado, J. Braga, J. B. Sousa, and R. Martins, "Implementation of a Control Architecture for Networked Vehicle Systems," in *IFAC Workshop on Navigation, Guidance and Control of Underwater Vehicles (NGCUV'2012)*, 2012.

[2] "LSTS - Laboratório de Sistemas e Tecnologia Subaquática | Underwater Systems and Technology Laboratory." [Online]. Available: http://whale.fe.up.pt/. [Accessed: 04-Jun-2012].

[3] R. M. F. Gomes, A. Martins, A. Sousa, J. B. Sousa, S. Fraga, and F. L. Pereira, "A new rov design: issues on low drag and mechanical symmetry," in *Oceans 2005-Europe*, 2005, vol. 2, pp. 957–962.

[4] H. Ferreira, R. Martins, E. Marques, J. Pinto, A. Martins, J. Almeida, J. Sousa, and E. P. Silva, "Swordfish: an autonomous surface vehicle for network centric operations," in *OCEANS 2007-Europe*, 2007, pp. 1–6.

[5] L. Madureira, A. Sousa, J. Sousa, and G. Gonçalves, "Low cost autonomous underwater vehicles for new concepts of coastal field studies," *CERF ICS*, 2009.

[6] E. Pereira, R. Bencatel, J. Correia, L. Félix, G. Gonçalves, J. Morgado, and J. Sousa, "Unmanned Air Vehicles for coastal and environmental research," *CERF ICS*, 2009.

[7] "JAUS - Joint Architecture for Ground Systems." [Online]. Available: http://www.jauswg.org/. [Accessed: 29-Apr-2012].

[8] R. P. Stokey, L. E. Freitag, and M. D. Grund, "A Compact Control Language for AUV acoustic communication," in *Oceans 2005-Europe*, 2005, vol. 2, pp. 1133–1137 Vol. 2.

[9] R. Franklin, "STANAG 4586, NATO Network Enabled Capability Service Oriented Architecture Working Group–Status Overview Presentation," *Oct 2010 STANAG*, vol. 4586.

[10] C. N. Duarte and B. B. Werger, "Defining a common control language for multiple autonomous vehicle operation," in *OCEANS 2000 MTS/IEEE Conference and Exhibition*, 2000, vol. 3, pp. 1861–1867 vol. 3.

[11] R. Martins, P. S. Dias, E. R. B. Marques, J. Pinto, J. B. Sousa, and F. L. Pereira, "IMC: A communication protocol for networked vehicles and sensors," in *OCEANS 2009 - EUROPE*, 2009, pp. 1 –6.

[12] J. Pinto, R. Martins, and J. B. Sousa, "Towards a REST-style architecture for networked vehicles and sensors," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, 2010, pp. 745–750.

[13] E. R. B. Marques, J. Pinto, S. Kragelund, P. S. Dias, L. Madureira, A. Sousa, M. Correia, H. Ferreira, R. G. calves, R. Martins, D. P. Horner, A. J. Healey, G. M. G. calves, and J. B. Sousa, “AUV Control and Communication using Underwater Acoustic Networks,” in *Oceans IEEE 2007*, Aberdeen, Scotland, 2007.

[14] J. Pinto, P. S. Dias, G. M. G. calves, R. G. calves, E. Marques, J. B. Sousa, and F. L. Pereira, “Neptus – A Framework to Support a Mission Life Cycle,” in *7th IFAC Conference on Manoeuvring and Control of Marine Craft*, Lisboa, 2006.

[15] P. S. Dias, J. B. Sousa, and F. L. Pereira, “Networked Operations (with Neptus),” in *MAST 2008, 3rd annual Maritime Systems and Technologies conference*, Cadiz, Spain, 2008.

[16] P. S. Dias, R. M. F. Gomes, J. Pinto, S. L. Fraga, G. M. G. calves, J. B. Sousa, and F. L. Pereira, “Neptus - A Framework to Support Multiple Vehicle Operation,” in *Oceans05Europe*, Brest, France, 2005.

[17] P. S. Dias, R. M. F. Gomes, J. Pinto, S. L. Fraga, G. M. G. calves, J. B. Sousa, and F. L. Pereira, “Mission Planning and Specification in the Neptus Framework,” in *IEEE International Conference on Robotics and Automation*, Orlando, Florida, USA, 2006.

[18] P. S. Dias, J. Pinto, G. M. G. calves, R. G. calves, J. B. Sousa, and F. L. Pereira, “Mission Review and Analysis,” in *Fusion2006 - 9th International Conference on Information Fusion*, Florence, Italy, 2006.

[19] P. S. Dias, J. Pinto, R. Goncalves, and J. B. Sousa, “Enabling a dialog – A C2I infrastructure for unmanned vehicles and sensors,” in *Autonomous and Intelligent Systems (AIS), 2010 International Conference on*, 2010, pp. 1–6.

[20]“Documentation - ROS Wiki.” [Online]. Available: http://www.ros.org/wiki/. [Accessed: 04-Jul-2012].

[21] B. Gerkey, R. T. Vaughan, and A. Howard, “The player/stage project: Tools for multi-robot and distributed sensor systems,” in *Proceedings of the 11th international conference on advanced robotics*, 2003, pp. 317–323.

[22] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “ROS: an open-source Robot Operating System,” in *ICRA Workshop on Open Source Software*, 2009, vol. 3.

[23] M. Quigley, E. Berger, and A. Y. Ng, “Stair: Hardware and software architecture,” in *AAAI 2007 Robotics Workshop, Vancouver, BC*, 2007.

[24] K. A. Wyrobek, E. H. Berger, H. F. M. Van der Loos, and J. K. Salisbury, “Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot,” in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, 2008, pp. 2165–2170.

[25] M. Montemerlo, N. Roy, and S. Thrun, “Perspectives on standardization in mobile robot programming: The Carnegie Mellon navigation (CARMEN) toolkit,” in *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, 2003, vol. 3, pp. 2436–2441 vol. 3.

[26] R. T. Vaughan and B. P. Gerkey, “Really reusable robot code and the player/stage project,” *Software Engineering for Experimental Robotics. Springer*, 2007.

[27] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. O’Reilly Media, 2008.

[28] S. Srinivasa, D. Ferguson, M. V. Weghe, R. Diankov, D. Berenson, C. Helfrich, and H. Strasdat, “The robotic busboy: Steps towards developing a mobile robotic home assistant,” in *International conference on intelligent autonomous systems*, 2008, pp. 651–656.

[29] J. Jackson, “Microsoft robotics studio: A technical introduction,” *Robotics & Automation Magazine, IEEE*, vol. 14, no. 4, pp. 82–87, 2007.

[30] O. Michel, “Webots: a powerful realistic mobile robots simulator,” in *Proceeding of the Second International Workshop on RoboCup*, 1998.

[31] R. A. Becker, J. M. Chambers, and A. R. Wilks, *The new S language: a programming environment for data analysis and graphics*. Wadsworth & Brooks/Cole Advanced Books & Software, 1988.

[32]“The R Project for Statistical Computing.” [Online]. Available: http://www.r-project.org/. [Accessed: 23-Apr-2012].

[33]“R Project’s Subversion.” [Online]. Available: https://svn.r-project.org/R/. [Accessed: 28-Jun-2012].

[34]“ETH ftp server.” [Online]. Available: ftp://ftp.stat.math.ethz.ch/Software/R/. [Accessed: 28-Jun-2012].

[35] P. Wand and C. Jones, *Kernel Smoothing*. Chapman & Hall, 1995.

[36] W. N. Venables and B. D. Ripley, *Modern Applied Statistics with S. Fourth Edition*. New York: Springer, 2002.

[37] A. C. Davison and D. V. Hinkley, *Bootstrap Methods and Their Application*. Cambridge University Press, 1997.

[38] L. P. Deutsch, “GZIP file format specification version 4.3,” 1996.

[39] J. Seward, “The bzip2 and libbzip2 official home page,” *Online]. http://www. bzip. org*, 2000.

[40] C. Morbidoni, D. Le Phuoc, A. Polleres, M. Samwald, and G. Tummarello, “Previewing semantic web pipes,” *The Semantic Web: Research and Applications*, pp. 843–848, 2008.

[41] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming: The Sockets Networking API*, vol. 1. Addison-Wesley Professional, 2004.

[42]“The Omega Project for Statistical Computing.” [Online]. Available: http://www.omegahat.org/. [Accessed: 28-Apr-2012].

[43]“OpenJAUS - JAUS Robotics Software Development Kit (SDK).” [Online]. Available: http://www.openjaus.com/. [Accessed: 29-Jun-2012].

[44] I. X. Part and M. Peschke, “Design and validation of computer protocols,” 2003.

[45] T. Erl, *Service-oriented architecture: concepts, technology, and design*. Prentice Hall PTR, 2005.

[46] T. Galluzzo and D. Kent, “The OpenJAUS Approach To Designing And Implementing The New Sae JAUS Standards,” in *AUVSI Unmanned Systems Conference*, 2010.

[47] N. E. Leonard, D. A. Paley, F. Lekien, R. Sepulchre, D. M. Fratantoni, and R. E. Davis, “Collective motion, sensor networks, and ocean sampling,” *Proceedings of the IEEE*, vol. 95, no. 1, pp. 48–74, 2007.

[48] J. Sousa, T. Simsek, and P. Varaiya, “Task planning and execution for UAV teams,” in *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, 2004, vol. 4, pp. 3804–3810 Vol. 4.

[49]“LSTS - Seascout Toolchain (Combined Releases).” [Online]. Available: http://whale.fe.up.pt/seascout/index.php?n=Private.Software. [Accessed: 04-Jun-2012].

[50]“LSTS - Information,” 09:45:47. [Online]. Available: http://whale.fe.up.pt/neptus/info.html. [Accessed: 04-Jun-2012].

[51] E. Marques, G. M. Gonçalves, and J. B. Sousa, “Seaware: the use of a publish/subscribe communications middleware for networked vehicle systems,” 2006.

[52] V. Potdar, A. Sharif, and E. Chang, “Wireless sensor networks: A survey,” in *Advanced Information Networking and Applications Workshops, 2009. WAINA’09. International Conference on*, 2009, pp. 636–641.

[53] “LSTS Development Server.” [Online]. Available: https://whale.fe.up.pt/svn/neptus/IMCJava/. [Accessed: 05-Jul-2012].

[54] J. Pinto, P. S. Dias, J. B. Sousa, and F. L. Pereira, “Large scale data collection using networks of heterogeneous vehicles and sensors,” in *OCEANS 2009 - EUROPE*, 2009, pp. 1–6.

[55] S. Rowe and C. R. Wagner, “An Introduction to the Joint Architecture for Unmanned Systems (JAUS),” *Ann Arbor*, vol. 1001, p. 48108, 2008.

[56] D. C. Fallside and P. Walmsley, “XML schema part 0: primer second edition,” *W3C recommendation*, 2004.

[57] A. Berglund, *Extensible Stylesheet Language (XSL) Version 1.1. W3C Recommendation 05 December 2006*. 2006.

[58] “Java (programming language) - Wikipedia, the free encyclopedia.” [Online]. Available: http://en.wikipedia.org/wiki/Java_(programming_language). [Accessed: 05-Jul-2012].

[59] “java.com: Java + You.” [Online]. Available: http://www.java.com. [Accessed: 05-Jul-2012].

[60] “Oracle Documentation.” [Online]. Available: http://docs.oracle.com/. [Accessed: 05-Jul-2012].

[61] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language*, vol. 2. Addison-wesley Reading, MA, 1996.

[62] K. Arnold, J. Gosling, and D. Holmes, *Java (TM) Programming Language, The*. Addison-Wesley Professional, 2005.

[63] “Eclipse - The Eclipse Foundation open source community website.” [Online]. Available: http://www.eclipse.org/. [Accessed: 05-Jul-2012].

[64] “Eclipse Public License - Version 1.0.” [Online]. Available: http://www.eclipse.org/legal/epl-v10.html. [Accessed: 04-Jul-2012].

[65] E. Clayberg and D. Rubel, *Eclipse: Building Commercial-Quality Plug-ins (Eclipse)*. Addison-Wesley Professional, 2006.

[66] J. D’Anjou, *The Java developer’s guide to Eclipse*. Addison-Wesley Professional, 2005.

[67] J. McAffer and J. M. Lemieux, *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java (TM) Applications*. Addison-Wesley Professional, 2005.

[68] “Version Control with Subversion.” [Online]. Available: http://svnbook.red-bean.com/. [Accessed: 05-Jul-2012].

[69] “Apache Subversion - Wikipedia, the free encyclopedia.” [Online]. Available: http://en.wikipedia.org/wiki/Apache_Subversion. [Accessed: 05-Jul-2012].

[70] “Apache Subversion.” [Online]. Available: http://subversion.apache.org/. [Accessed: 05-Jul-2012].

[71] E. R. Gansner and S. C. North, “An open graph visualization system and its applications to software engineering,” *Software Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, 2000.

[72] “GraphViz Java API.” [Online]. Available: http://www.loria.fr/~szathmar/off/projects/java/GraphVizAPI/index.php. [Accessed: 19-Jul-2012].

[73] “Ubigraph: Free dynamic graph visualization software.” [Online]. Available: http://ubietylab.net/ubigraph/. [Accessed: 10-Jul-2012].