

**UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA ELÉTRICA**

**rqt_mrta: Um Pacote ROS para
Configuração e Supervisão de Arquiteturas
MRTA**

Adriano Henrique Rossette Leite

Itajubá, 27 de novembro de 2017

**UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA ELÉTRICA**

Adriano Henrique Rossette Leite

**rqt_mrta: Um Pacote ROS para
Configuração e Supervisão de Arquiteturas
MRTA**

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica como parte dos requisitos para obtenção do Título de Mestre em Ciências em Engenharia Elétrica.

**Área de Concentração: Automação e Sistemas Elé-
tricos Industriais**

Orientador: Prof. Dr. Guilherme Sousa Bastos

**27 de novembro de 2017
Itajubá**

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA ELÉTRICA

rqt_mrta: Um Pacote ROS para
Configuração e Supervisão de Arquiteturas
MRTA

Adriano Henrique Rossette Leite

Dissertação aprovada por banca examinadora em
15 de Dezembro de 2017, conferindo ao autor o
título de **Mestre em Ciências em Engenharia
Elétrica.**

Banca Examinadora:

Prof. Dr. Guilherme Sousa Bastos (Orientador)

Prof. Dr. Edson Prestes

Prof. Dr. Laércio Augusto Baldochi Júnior

**Itajubá
2017**

Adriano Henrique Rossette Leite

rqt_mrta: Um Pacote ROS para Configuração e Supervisão de Arquiteturas MRTA

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica como parte dos requisitos para obtenção do Título de Mestre em Ciências em Engenharia Elétrica.

Trabalho aprovado. Itajubá, 15 de Dezembro de 2017:

Prof. Dr. Guilherme Sousa Bastos
Orientador

Prof. Dr. Edson Prestes

Prof. Dr. Laércio Augusto Baldochi
Júnior

Itajubá
27 de novembro de 2017

Agradecimentos

À Deus ...

À meus familiares, namorada e amigos ...

Ao meu orientador ...

À banca examinadora, ...

Aos amigos e colegas do LRO, ...

À Capes pelo apoio financeiro durante estes 2 anos.

*“I may never find all the answers.
I may never understand why.
I may never prove what I know to be true.
But I know that I still have to try.”
(Dream Theater)*

Resumo

Este trabalho apresenta o desenvolvimento do pacote baseado em ROS *rqt_mrta*, o qual fornece um *plugin* de interface gráfica de usuário para a parametrização amigável de arquiteturas para a resolução de problemas de alocação de tarefa em sistema multirrobô. Além disso, em tempo de execução, o *plugin* dispõe elementos gráficos para a supervisão e monitoramento da arquitetura e do sistema multirrobô. Utilizando uma aplicação de testes com a arquitetura ALLIANCE, pode-se constatar que o pacote reduziu a complexidade da tarefa de parametrização de arquiteturas de alocação de tarefa.

Palavras-chaves: ALLIANCE. Alocação de Tarefa. Arquitetura. ROS. Sistema Multirrobô.

Abstract

This work presents the *rqt_mrta*, an ROS-based project that provides a Graphical User Interface (GUI) plugin for configuring multirobot task allocation (MRTA) architectures. Moreover, it arranges graphical elements for system supervision and monitoration them during runtime. A generic approach of the ALLIANCE architecture is developed in order to test and validate this graphical tool by creating a multirobot patrol application. The results shows that the usage of the *rqt_mrta* plugin facilitates the parametrization of MRTA architectures.

Key-words: ALLIANCE. Architecture. Multirobot System. ROS. Task Allocation.

Lista de figuras

Figura 1 – Representação visual da taxonomia de três eixos	22
Figura 2 – Conceitos básicos de comunicação do ROS.	27
Figura 3 – Exemplo de ferramentas gráficas existentes no ROS.	31
Figura 4 – Diagrama de classes da camada do modelo.	45
Figura 5 – Janela principal do <i>plugin rqt_mrta</i>	46
Figura 6 – <i>Wizard</i> para a criação de uma nova aplicação.	47
Figura 7 – Pastas e arquivos gerados após a criação de uma aplicação.	49
Figura 8 – Carregando um arquivo de configuração	49
Figura 9 – Diagrama de classes do camada de controle.	50
Figura 10 – Detalhamento da motivação <i>/robot2/alliance/wander</i> ao longo do tempo.	61
Figura 11 – Motivações das configurações de comportamento do robô <i>/robot3</i>	62
Figura 12 – Ferramenta <i>rospack</i> encontra a arquitetura <i>alliance</i>	69
Figura 13 – Robô Pioneer 3 DX da Adept MobileRobots.	70
Figura 14 – Criação da aplicação <i>patrulha</i>	72
Figura 15 – Pastas e arquivos gerados após a criação de uma aplicação.	73
Figura 16 – Ferramenta <i>rospack</i> encontra a aplicação <i>patrulha</i>	74
Figura 17 – Inicialização dos nós da arquitetura para três robôs.	75
Figura 18 – Grafo da arquitetura <i>alliance</i> no ROS para três robôs.	76
Figura 19 – Grafo da aplicação <i>patrulha</i> no ROS.	77
Figura 20 – Motivação da configuração de comportamento <i>/robot1/wander</i>	78
Figura 21 – Motivação da configuração de comportamento <i>/robot2/wander</i>	79
Figura 22 – Motivação da configuração de comportamento <i>/robot2/border_protection</i>	80

Lista de tabelas

Tabela 1 – Comparação de três variações do CNP.	24
Tabela 2 – Exemplos de resolução de nomes no ROS	30

Lista de abreviaturas e siglas

ABNT	Associação Brasileira de Normas Técnicas
API	<i>Application Programming Interface</i>
CBR	Competição Brasileira de Robótica
CNP	<i>Contract Net Protocol</i>
GPU	<i>Graphics Processing Unit</i>
GUI	<i>Graphical User Interface</i>
LRO	Laboratório de Robótica
MAS	<i>Multi-Agent System</i>
MRS	<i>Multi-Robot System</i>
MRTA	<i>Multi-Robot Task Allocation</i>
MVC	<i>Model-View-Controller</i>
OAP	<i>Optimal Assignment Problem</i>
P3DX	<i>Adept MobileRobots Pioneer 3 DX</i>
ROS	<i>Robot Operating System</i>
UI	<i>User Interface</i>
UML	<i>Unified Modeling Language</i>
UNIFEI	Universidade Federal de Itajubá
XML	<i>Extensible Markup Language</i>
YAML	<i>YAML Ain't Markup Language</i>

Sumário

1	INTRODUÇÃO	15
1.1	Motivação	15
1.2	Objetivos	16
1.3	Contribuições	17
1.4	Estrutura do Trabalho	17
2	REVISÃO TEÓRICA	18
2.1	Sistema Multirrobo	18
2.1.1	Composição: homogêneo <i>versus</i> heterogêneo	19
2.1.2	Cooperação: cooperativo <i>versus</i> competitivo	19
2.1.3	Coordenação: deliberativa <i>versus</i> reativa	19
2.1.4	Comunicação: implícita <i>versus</i> explícita	19
2.1.5	Organização: centralizada <i>versus</i> distribuída	20
2.2	Alocação de Tarefa em Sistema Multirrobo	20
2.2.1	Definição formal	20
2.2.2	Taxonomia	21
2.2.3	Arquitetura MRTA	22
2.2.3.1	Arquiteturas baseadas em comportamento	22
2.2.3.2	Arquiteturas baseadas em negociação	23
2.3	ROS - Robot Operating System	24
2.3.1	Conceitos básicos	25
2.3.1.1	Sistema de arquivos do ROS	25
2.3.1.2	Grafo de computação do ROS	26
2.3.1.3	Comunidade do ROS	28
2.3.1.4	Nome de recurso de grafo	28
2.3.1.5	Nome de recurso de pacote	30
2.3.2	Interface gráfica de usuário do ROS	30
2.4	Trabalhos Relacionados	32
3	DESENVOLVIMENTO	33
3.1	Arquivo de configuração de arquitetura	34
3.1.1	Formatação	34
3.1.2	Registro da arquitetura	39
3.2	Arquivo de configuração de aplicação	40
3.2.1	Formatação	40
3.2.2	Registro da aplicação	43

3.3	Camada do modelo	43
3.4	Camada de visualização	44
3.4.1	Widget principal do <i>rqt_mrta</i>	44
3.4.2	Wizard para criação de nova aplicação	46
3.4.3	Dialogs para a seleção de arquiteturas e aplicações	48
3.5	Camada de controle	48
4	EXPERIMENTOS E RESULTADOS	53
4.1	<i>alliance</i>	54
4.1.1	Plugin de sensor	55
4.1.2	Plugin de avaliação sensorial	56
4.1.3	Plugin de camada	58
4.1.4	<i>alliance_msgs</i>	60
4.1.5	<i>rqt_alliance</i>	60
4.1.6	Arquivos de parâmetro do <i>alliance</i>	63
4.1.7	Arquivo de inicialização do <i>alliance</i>	64
4.1.8	Cadastro da arquitetura do pacote <i>alliance</i>	65
4.2	<i>patrulha</i>	69
4.2.1	Criação da aplicação através do <i>plugin rqt_mrta</i>	71
4.2.2	Análise da aplicação gerada	73
4.2.3	Simulação a partir dos arquivos gerados	74
5	CONCLUSÃO E TRABALHOS FUTUROS	81
5.1	Conclusão	81
5.2	Trabalhos Futuros	81
	REFERÊNCIAS	82
	APÊNDICES	85
	APÊNDICE A – EXEMPLO DE UM ARQUIVO DE CONFIGURAÇÃO DE ARQUITETURA PARA O <i>RQT_MRTA</i>	86
	APÊNDICE B – EXEMPLO DE UM ARQUIVO DE CONFIGURAÇÃO DE APLICAÇÃO PARA O <i>RQT_MRTA</i>	94
	APÊNDICE C – EXEMPLO DE MANIFESTO DE PACOTE PARA ARQUITETURA	95

APÊNDICE D – EXEMPLO DE MANIFESTO DE PACOTE PARA
APLICAÇÃO 96

APÊNDICE E – ALLIANCE 97

1 Introdução

1.1 Motivação

Aplicações de robótica onde vários robôs interagem entre si e também com o ambiente em que estão inseridos são chamadas de sistemas multirrobo, do inglês *Multi-robot systems* (MRS). Um sistema multirrobo possui diversas vantagens sobre sistemas com apenas um robô. Entre elas se encontram o ganho de flexibilidade, a simplificação de tarefas complexas e o aumento da eficiência no uso de recursos, de desempenho do sistema como um todo e da robustez através de redundâncias (CAO; FUKUNAGA; KAHNG, 1997; DUDEK et al., 1996; ZLOT et al., 2002). Entretanto, aplicações dessa natureza demandam arquiteturas complexas para o controle da coordenação dos robôs envolvidos e, intrinsecamente, possuem problema de escalabilidade nos processos computacionais e na rede de comunicação.

Um dos problemas mais desafiadores em aplicações com vários robôs é denominado *alocação de tarefa* (MRTA, acrônimo para *Multi-Robot Task Allocation*), que busca atribuir a execução de um conjunto de tarefas para um grupo de robôs sujeitos à limitações de forma que o desempenho geral do sistema seja otimizado. Esse tipo de problema pode ser resolvido por arquiteturas que se baseiam em modelos de organização encontrados no cotidiano. Cada arquitetura possui um conjunto de premissas que delimita os tipos de problema que podem ser resolvidos por ela. Com o intuito de classificar os tipos de problemas MRTA existentes, Gerkey e Mataric (2004) sugeriu uma taxonomia independente do domínio, mostrando algumas arquiteturas que atende cada tipo (PARKER, 1998; GERKEY; MATARIC, 2002; BOTELHO; ALAMI, 1999; WERGER; MATARIĆ, 2000; FRANK, 2005; STENTZ; DIAS, 1999; CHAIMOWICZ; CAMPOS; KUMAR, 2002).

Com o advento do ROS (do inglês *Robot Operating System*) (QUIGLEY et al., 2009), vários sistemas inteligentes puderam ser reutilizados em diversas aplicações de robótica, tais como: localização (LI; BASTOS, 2017), navegação robótica, gerenciamento de largura de banda (JULIO; BASTOS, 2015), planejamento e escalonamento de ações e tarefas (FOX; LONG, 2003; MANNE, 1960), algoritmos de inteligência artificial (SCHNEIDER et al., 2015; WATKINS; DAYAN, 1992), entre outros. Sendo um *middleware* dedicado para aplicações robóticas, ele possibilitou a integração de trabalhos desenvolvidos por equipes distintas de pesquisa em robótica, pois ele simplifica o desenvolvimento de processos e dá suporte à comunicação e interoperabilidade deles. Desta forma, pesquisadores de robótica podem ater-se ao desenvolvimento de projetos dentro da sua especialização, necessitando apenas configurar os demais pacotes para a execução da aplicação. Problemas que anteriormente possuíam difícil solução em termos de *software*, foram simplificados a

partir da modularidade proporcionada pelo ROS.

Apesar da vasta existência de arquiteturas de alocação de tarefa para sistema multirrobo, houveram poucas tentativas de aproximação genérica delas em projetos baseados em ROS. Li et al. (2016) elaboraram um pacote ROS¹ que abstrai a arquitetura ALLIANCE, fornecendo um *framework* que encapsula vários elementos do ALLIANCE e a comunicação entre os robôs. Assim, seus usuários podem focar na lógica da aplicação. Reis e Bastos (2015) mostraram as facilidades que o ROS oferece na implementação da arquitetura ALLIANCE, proposta por Parker (1998). Contudo, essa aproximação atende apenas o problema aplicado nesse trabalho. O conjunto de pacotes *auction_methods_stack*² é um projeto que foi desenvolvido em uma versão antiga do ROS e, além de não estar documentado, nunca mais foi atualizado.

Com isso, verifica-se a necessidade de ferramentas que facilitem a utilização de arquiteturas de alocação de tarefa em sistemas multirrobo para o *framework* ROS para incentivar o desenvolvimento de abordagens genéricas dessas arquiteturas.

1.2 Objetivos

Esse trabalho propõe desenvolver um pacote ROS, denominado *rqt_mрта*, que facilite a utilização de arquiteturas de alocação de tarefa no ROS para sistemas multirrobo. Esse pacote fornece uma interface gráfica que foi desenvolvida com o intuito de disponibilizar serviços para dois tipos de clientes: (1) desenvolvedor e (2) usuário de arquitetura MRTA. Seus serviços são:

- Cadastro de novas arquiteturas no ROS para seu uso na solução de problemas de alocação de tarefa em sistemas multirrobo;
- Criação de novos projetos contendo a definição de um problema de alocação de tarefa;
- Configuração da arquitetura escolhida para resolver o problema MRTA;
- Armazenamento dos dados de configuração no projeto criado;
- Monitoramento da comunicação dos robôs do sistema no ROS em tempo de execução;
- Monitoramento das atividades dos robôs no sistema em tempo de execução.

¹ <http://wiki.ros.org/micros_mars_task_alloc>

² <https://github.com/joaquintas/auction_methods_stack>

1.3 Contribuições

A partir da elaboração deste trabalho, os seguintes pacotes baseados em ROS foram obtidos e disponibilizados para a comunidade ROS:

- ***rqt_mrta***³: uma interface gráfica de usuário que facilita a utilização de arquiteturas que resolvem o problema de alocação de tarefa em um sistema multirrobo no ROS.
- ***alliance***⁴: uma aproximação genérica da arquitetura ALLIANCE para a utilização em aplicações baseadas em ROS.

1.4 Estrutura do Trabalho

No Capítulo 2, é feita uma revisão bibliográfica sobre as características de um sistema multirrobo, o problema de alocação de tarefa e a classificação desses problemas segundo uma taxonomia independente do domínio. Ainda neste capítulo, é feita uma descrição do ROS, plataforma sobre a qual este trabalho foi desenvolvido, abordando seus conceitos básicos e o desenvolvimento de interfaces gráficas para seus usuários. O Capítulo 3 trata sobre o desenvolvimento da interface gráfica *rqt_mrta*, detalhando a estrutura e a geração dos arquivos de configuração. Em seguida, no Capítulo 4, é explicado a construção do pacote *alliance* para o ROS, mostrando suas características e suas configurações. Esta arquitetura é utilizada na criação e execução do aplicação *patruha*. Finalmente, no Capítulo 5, são discutidas as conclusões e possibilidades de melhorias futuras para a interface gráfica *rqt_mrta* e também para o pacote *alliance*.

³ <http://wiki.ros.org/rqt_mrta>

⁴ <<http://wiki.ros.org/alliance>>

2 Revisão teórica

Primeiramente, são revisados o conceito de sistema multirrobo, suas vantagens e características. Em seguida é apresentado o problema de atribuição de tarefas em sistemas com vários robôs, revisando uma taxonomia independente do domínio e algumas abordagens de arquiteturas que visam resolver este problema. Enfim, são tratadas as contribuições do ROS para a comunidade de pesquisa em robótica, dos seus conceitos básicos e do desenvolvimento de interfaces gráficas integradas com o ROS.

2.1 Sistema Multirrobo

Os avanços em *hardware*, *software* e comunicação influenciaram diretamente no crescimento da pesquisa sobre aplicações envolvendo vários robôs. Tais como, redes de sensores autônomos, vigilância e patrulha de construções, transporte de objetos grandes, monitoramento aéreo e subaquático de poluição, detecção de incêndios florestais, sistemas de transporte e busca e resgate em áreas afetadas por grandes desastres (LIMA; CUSTODIO, 2005). Um sistema multirrobo, do inglês *Multirobot System* (MRS), é caracterizado por aplicações que envolvem vários robôs que se interagem em um mesmo ambiente. Yan, Jouandeau e Cherif (2013) apontam diversas vantagens que um sistema multirrobo (MRS, *Multi-Robot System*) possui perante um sistema com apenas um robô:

- possui melhor distribuição espacial;
- alcança um melhor desempenho geral do sistema;
- adiciona robustez ao sistema através da fusão de dados e troca de informações entre os robôs;
- pode ter custos menores: usando um número de robôs simples pode ser mais fácil para programar e mais barato para construir do que usando um único robô poderoso que é complexo e caro para realizar uma tarefa;
- além de exibir maior confiabilidade, flexibilidade, escalabilidade e versatilidade.

Para melhor organizar e compreender um sistema multirrobo, diversas características podem ser utilizadas para classificá-lo. Serão mostradas a seguir as características que um sistema multirrobo apresenta.

2.1.1 Composição: homogêneo *versus* heterogêneo

Um sistema multirrobô pode ser formado por um conjunto de robôs homogêneos ou heterogêneos. As capacidades individuais dos robôs em um sistema homogêneo são idênticas, mesmo que suas estruturas físicas não são iguais. Já em um time de robôs heterogêneos, as capacidades dos robôs são diferentes. Caso em que os robôs podem se especializar na realização de algumas tarefas.

2.1.2 Cooperação: cooperativo *versus* competitivo

Os robôs do sistema podem responder a estímulos externos cooperativamente ou competitivamente. Quando há cooperação entre os robôs, eles se interagem conjuntamente de modo à completar uma tarefa para o aumento da utilidade total do sistema. Por outro lado, em um sistema competitivo, cada robô visa aumentar a própria utilidade, não importando com os demais robôs do sistema. Sistemas que apresentam escassez em recursos possuem maior desempenho quando seus agentes se interagem competitivamente.

2.1.3 Coordenação: deliberativa *versus* reativa

Um sistema de vários robôs necessita coordenação, a qual pode ser de dois tipos: (1) deliberativa, também conhecida como estática ou *offline*; e (2) reativa, também conhecida como dinâmica ou *online*. Na coordenação deliberativa, é adotado um conjunto de convenções antes do início da execução da tarefa. Como por exemplo, algumas regras de trânsito podem evitar acidentes: manter a direita, parar em interseções, manter uma distância de segurança do robô da frente, etc. No entanto, a coordenação reativa ocorre durante a execução de uma tarefa e é geralmente baseada em análise e síntese de informação. Este último tipo ainda pode ser distinguido entre coordenação explícita e implícita. Quando é aplicado uma técnica em que se emprega uma comunicação intencional e métodos colaborativos, esta é dita coordenação explícita. Enfim, a coordenação implícita (ou emergente) se dá quando é aplicada uma técnica que atinge o desempenho coletivo desejado através da interação dinâmica entre os robôs e o ambiente, isto é, os robôs se coordenam por meio dos reflexos que possuem.

2.1.4 Comunicação: implícita *versus* explícita

A troca de informação em um sistema multirrobô é extremamente importante, pois ela permite a cooperação e coordenação entre seus robôs. Quando a comunicação do sistema é explícita, os robôs trocam mensagens intencionalmente na forma um-para-um (*unicast*) ou um-para-muitos (*broadcast*). Porém, quando sua comunicação é do tipo implícita, os robôs do sistema obtêm informação do ambiente e dos demais robôs através dos seus sensores. Especificando ainda mais este último tipo, comunicação implícita ativa

diz respeito à robôs que se comunicam através da coleta de resto de informação deixada pelos demais robôs no sistema, como por exemplo quando um robô segue os rastros de outro robô; e comunicação implícita passiva se refere à robôs que se comunicam ao observar mudanças no ambiente através dos seus sensores.

2.1.5 Organização: centralizada *versus* distribuída

Sistemas organizados em uma forma centralizada possuem um líder que observa todo o sistema como um todo e, a partir dessa observação, delega tarefas para os demais robôs. Assim, enquanto o líder toma decisões, os demais robôs agem conforme o seu comando. Em sistemas distribuídos, cada robô é capaz de tomar sua própria decisão autonomamente com respeito aos outros robôs.

2.2 Alocação de Tarefa em Sistema Multirrobô

Um dos problemas mais desafiadores em aplicações multirrobô leva o nome *alocação de tarefa*, na língua inglesa, *Multi-Robot Task Allocation* (MRTA). Problemas dessa natureza buscam como solução atribuir de forma ótima um conjunto de robôs para um conjunto de tarefas de maneira que o desempenho geral de um sistema sujeito a um conjunto de limitações seja otimizado.

É dada a seguir uma definição formal do problema de alocação de tarefa em sistema multirrobô. Na sequência, é mostrado uma taxonomia para a classificação de problemas MRTA. Finalmente, é apresentado o papel de uma arquitetura MRTA e também as principais abordagens existentes.

2.2.1 Definição formal

Zlot e Stentz (2006) define o problema de alocação de tarefa em um sistema multirrobô conforme o seguinte problema de atribuição ótima (OAP - *Optimal Assignment Problem*) estático.

Definição 2.1. (*Alocação de Tarefa em um Sistema Multirrobô*) Sejam dados um conjunto T , um conjunto R e uma função de custo para cada subconjunto de robots $r \in R$ que especifique o custo de performance para cada subconjunto de tarefas, $c_r : 2^T \rightarrow \mathbb{R}_+ \cup \{\infty\}$: procure a alocação $A^* \in R^T$ que minimiza a função objetivo global $C : R^T \rightarrow \mathbb{R}_+ \cup \{\infty\}$. —

Para que um algoritmo consiga encontrar uma solução ótima para este problema, é necessário levar em consideração todo o espaço de alocação R^T , cujo tamanho aumenta exponencialmente em função do número de tarefas e robôs no sistema. No entanto, como

um problema MRTA possui natureza dinâmica, que varia no tempo com mudanças do ambiente, a solução de um OAP estático não é mais aplicável.

2.2.2 Taxonomia

Gerkey e Mataric (2004) sugeriram uma taxonomia de três eixos independente do domínio para a classificação de problemas de alocação de tarefas em sistemas multirrobo.

O primeiro eixo determina o *tipo dos robôs* que compõem o problema. Os tipos de robôs possíveis são: *ST* (acrônimo para *Single-Task*) ou *MT* (acrônimo para *Multi-Task*). Problemas que envolvem robôs que só podem executar uma tarefa por vez são compostos por robôs do tipo *ST*. Entretanto, se houver pelo menos um robô capaz de executar mais de uma tarefa simultaneamente, então esse problema é composto por robôs do tipo *MT*.

O segundo eixo da taxonomia determina o *tipo das tarefas* que compõem o problema. Nesse caso, são possíveis os tipos: *ST* (acrônimo para *Single-Robot*) ou *MR* (acrônimo para *Multi-Robot*). Em problemas cujo tipo das tarefas é *SR*, todas as tarefas envolvidas só podem ser executadas por um único robô. Porém, quando o tipo das tarefas envolvidas é *MR*, estas podem ser executadas por mais de um robô.

O terceiro eixo, por sua vez, determina o *tipo da alocação* do problema, o qual pode assumir os valores: *IA* (acrônimo para *Instantaneous Assignment*) ou *TA* (acrônimo para *Time-extended Assignment*). O primeiro caso, *IA*, diz respeito à problemas MRTA onde as alocações das tarefas para os robôs são realizadas instantaneamente, sem levar em consideração o estado futuro do sistema. Por outro lado, em problemas cujo tipo de alocação é *TA*, além de conhecido o estado atual de cada robô e do ambiente, também é conhecido o conjunto de tarefas que precisarão ser alocadas no futuro. Neste último caso, diversas tarefas são alocadas para um robô, o qual deve executar cada alocação conforme seu agendamento. De acordo com Bastos, Ribeiro e Souza (2008), quando o tipo de alocação do problema MRTA é *IA*, o número de robôs é superior ao número de tarefas alocadas e quando *TA*, o oposto acontece. Isso se deve ao fato de que, em problemas MRTA cujo tipo de alocação é *IA*, o número de robôs no sistema é capaz de suprir a taxa de tarefas a serem atribuídas, de modo que é muito provável que haverão robôs ociosos no sistema; enquanto, naqueles cujo tipo de alocação é *TA*, o número de robôs que compõem o sistema não é suficiente para atender a taxa de tarefas a serem alocadas no sistema.

A Figura 1 exibe uma representação gráfica da taxonomia de Gerkey e Mataric (2004) para a classificação de problemas MRTA (*Multi-Robot Task Allocation*), onde pode-se notar que existem oito classes de problemas MRTA bem definidos.

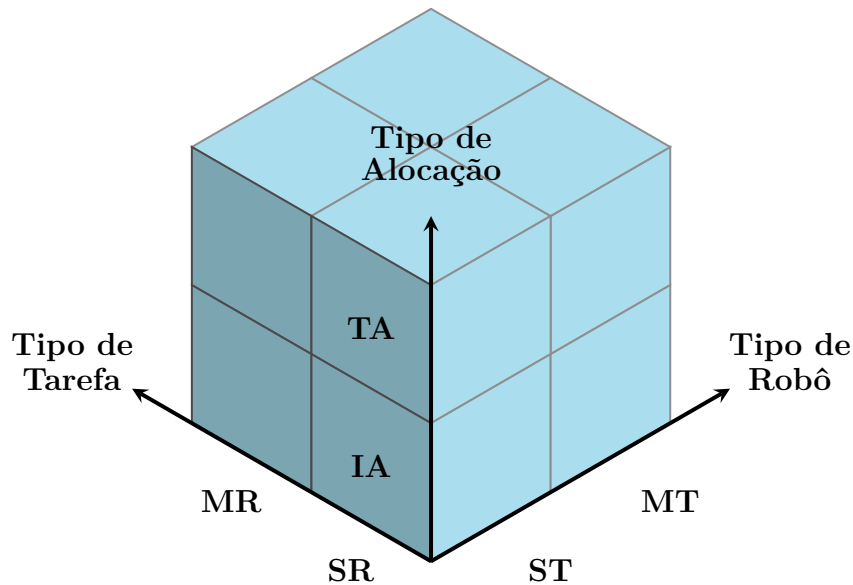


Figura 1 – Representação visual da taxonomia de três eixos sugerida por Gerkey e Mataric (2004).

2.2.3 Arquitetura MRTA

As arquiteturas MRTA possuem a função de solucionar o problema de alocação de tarefas em um dado sistema multirrobô.

Basicamente existem duas variantes para a atribuição de tarefas: iterativa e instantânea. As abordagens iterativas apresentam uma dinâmica progressiva mediante ao estado do sistema para que ocorra uma alocação. Nesse caso, as tarefas existentes são conhecidas *a priori*. Por outro lado, a atribuição de tarefas instantânea acontece em sistemas em que o conjunto de tarefas não é revelado de uma só vez, mas as tarefas são introduzidas uma a uma (GERKEY; MATARIĆ, 2004).

Com o objetivo de classificar e facilitar o entendimento, as arquiteturas podem ser divididas em arquiteturas baseadas em comportamento e em negociação.

2.2.3.1 Arquiteturas baseadas em comportamento

São modelos inspirados em sistemas biológicos. Normalmente, são baseadas na arquitetura de subsunção de Brooks (1986), a qual fornece uma abordagem em camadas para a criação de regras reativas para sistemas de controle completo de baixo para cima.

- **ALLIANCE**: é uma arquitetura distribuída que aloca tarefas mediante os requerimentos da missão, das atividades dos outros robôs, as atuais condições do ambiente e o próprio estado interno do robô (PARKER, 1998);
- **L-ALLIANCE**: é uma variação da arquitetura ALLIANCE capaz de estimar e atualizar os parâmetros de controle das configurações de comportamento a partir

de um conhecimento adquirido (PARKER, 1996);

- **BLE**: procura, entre os robôs disponíveis e as tarefas a serem alocadas, o par robô-tarefa (i, j) que possui maior utilidade e aloca a tarefa j para o robô i até que exista robôs disponíveis (WERGER; MATARIĆ, 2000).

2.2.3.2 Arquiteturas baseadas em negociação

Muitas arquiteturas baseadas em regras de negociação são variações do Protocolo de Rede de Contrato, do inglês *Contract Net Protocol* (CNP), sugerido por Smith (1980). Este mecanismo estabelece um protocolo de comunicação para que os agentes de um sistema possam negociar a atribuição de tarefa com controle distribuído. Existem três abordagens clássicas do CNP para negociação: baseadas em regras de mercado, de leilão e de comércio. A primeira delas, abordagens baseada em regras de mercado, é composta por indivíduos competitivos cujo objetivo é se beneficiar maximizando o seu lucro e minimizando seus custos mesmo quando se trata de seus colegas de trabalho (ZLOT; STENTZ, 2006). Na sequência, abordagens baseadas em regras de leilão, uma excelente escolha para a alocação de recursos escassos (GERKEY; MATARIC, 2002). E, por fim, abordagens baseadas em regras de comércio, que são compostas por compradores e vendedores, cuja relação consiste em trocas: o comprador usa dinheiro para adquirir bens e serviços dos vendedores, enquanto os vendedores recebem o dinheiro para a entrega dos bens ou serviços (YAN; JOUANDEAU; CHERIF, 2011). A Tabela 1 mostra uma comparação realizada por Yan, Jouandeau e Cherif (2013) dessas abordagens.

Segue abaixo, exemplos de arquiteturas de alocação de tarefa em sistema multirrobô baseadas em modelos de negociação.

- **M+**: foi a primeira variação do CNP para a alocação e realização de tarefas em sistemas multirrobô. É composta por várias camadas, cada uma contendo um planejador e um supervisor. O planejador é responsável por gerar uma sequência de ações com o intuito de atingir um objetivo, enquanto o supervisor é responsável por executar e interagir com a próxima camada (BOTELHO; ALAMI, 1999);
- **FMS**: do inglês *The Free Market System*, é uma abordagem baseada em regras de mercado (DIAS; STENTZ, 2000);
- **Murdoch**: cada atribuição de tarefa é tratada como um processo de leilão, em que o robô vencedor é aquele que oferece o maior lance (GERKEY; MATARIC, 2002);

Tabela 1 – Comparação de três variações do CNP (YAN; JOUANDEAU; CHERIF, 2013).

	Abordagens baseadas em mercado	Abordagens baseadas em leilão	Abordagens baseadas em comércio
Modelo de comunicação na negociação	<i>publish / subscribe</i>	<i>publish / subscribe</i>	<i>apply / allocate</i>
Algoritmo de alocação de tarefa	algoritmo guloso	algoritmo guloso	algoritmo guloso
Abilidade de alocação de tarefa por iteração	uma tarefa	uma tarefa	várias tarefas
Determinação do papel dos robôs	voluntária	voluntária	negociação
Consideração de utilidade	custo e benefício	custo	custo
Reatribuição de tarefa	permitida	não permitida	permitida
Complexidade de comunicação	$O(1)$ /licitante, $O(n)$ /leiloeiro	$O(1)$ /licitante, $O(n)$ /leiloeiro	$O(1)$ /comprador, $O(n)$ /vendedor
Complexidade de computação	$O(n)$	$O(n)$	$O(n)$

2.3 ROS - Robot Operating System

Acrônimo para *Robot Operating System* (QUIGLEY et al., 2009), o ROS é um *framework* para robótica que tem incentivado a comunidade de pesquisadores desta área do conhecimento a trabalhar conjuntamente desde seu lançamento. Ao observar o grande avanço desta ferramenta de comunicação, muitos fabricantes de manipuladores industriais começaram a investir em pesquisas para integrar seus robôs com o ROS.

Uma lacuna que antes existia na nova geração de aplicações robóticas foi preenchida com o lançamento do ROS. Como um fornecedor de serviços de *middleware*, ele (1) simplifica o desenvolvimento de processos, (2) suporta comunicação e interoperabilidade, (3) oferece e facilita serviços frequentemente utilizados em robótica e, ainda, oferece (4) utilização eficiente dos seus recursos disponíveis, (5) abstrações heterogênicas e (6) descoberta e configuração automática de recursos (QUIGLEY et al., 2009). No intuito de cobrir todas exigências de um *middleware*, ROS 2.0 tenta dar suporte à sistemas embarcados e dispositivos de baixo recurso.

No ROS, projetos atômicos são chamados *pacotes* e podem ser desenvolvido em diversas linguagens de programação. Isso mostra que o ROS é flexível, pois seus usuá-

rios podem tirar proveito das vantagens que cada linguagem suportada tem, sejam elas eficiência em tempo de execução, confiabilidade, recursos, sintaxe, semântica, suporte ou documentação existente. Atualmente, as linguagens de programação suportadas são C++, Python e Lisp. As linguagens Java e Lua ainda estão em fase de desenvolvimento.

Projetos de robótica possuem rotinas que poderiam ser reutilizadas em outros projetos. Por esta razão, ROS é também modular, pois pacotes configuráveis existentes podem ser combinados para realizar uma aplicação específica de robótica. Várias bibliotecas externas já foram adaptadas para serem usadas no ROS: *aruco*¹, *gmapping*², interfaces de programação para aplicações de robôs³, sensores⁴ e simuladores⁵, planejadores⁶, reconhecimento de voz⁷, entre outros. Isso evidencia que os usuários de ROS podem focar no desenvolvimento de pesquisa de sua área e contribuir da melhor forma com essa comunidade.

Enfim, ROS disponibiliza diversas ferramentas para auxiliar no desenvolvimento de projetos e, também, verificar o funcionamento de aplicação. Suas ferramentas típicas são: *get* e *set* de parâmetros de configuração, visualização da topologia de conexão *peer-to-peer*, medição de utilização de banda, gráficos dos dados de mensagem e outras mais. É altamente recomendado o uso dessas ferramentas para garantir a estabilidade e confiança dos pacotes desenvolvidos, que normalmente têm alta complexidade.

Esta seção apresenta conceitos básicos para entender o funcionamento deste *framework*. Em seguida, são expostas as regras de nomenclatura dos recursos do ROS. Ao final, é discutido o uso de aplicações gráficas integradas com o ROS.

2.3.1 Conceitos básicos

Sua concepção foi fundada sobre conceitos divididos em três níveis: (1) sistema de arquivos do ROS, (2) grafo de computação do ROS e (3) comunidade do ROS. Serão explicados a seguir os três níveis, cada um com seu respectivo conjunto de conceitos. Além disso, também serão detalhados os dois tipos de nomes definidos no ROS: nomes de recursos de pacote e nomes de recursos de grafo.

2.3.1.1 Sistema de arquivos do ROS

Os conceitos envolvidos no nível do *sistema de arquivos do ROS* se referem aos arquivos armazenados em disco. São eles:

¹ <http://wiki.ros.org/ar_sys>

² <<http://wiki.ros.org/gmapping>>

³ <<http://wiki.ros.org/Robots>>

⁴ <<http://wiki.ros.org/Sensors>>

⁵ <<http://wiki.ros.org/gazebo>>

⁶ <<http://kcl-planning.github.io/ROSPlan/>>

⁷ <http://wiki.ros.org/Sensors#Audio_.2BAC8_Speech_Recognition>

- **Pacotes:** em inglês *Packages*, é uma forma atômica de organização de criação e lançamento de *software* no ROS. Um pacote contém definições de processos (nós), de dependência de bibliotecas, de tipos de mensagens, ações e serviços, de estruturas de dados e, por fim, de configuração.
- **Metapacotes:** em inglês *Metapackages*, é um tipo especial de pacote que tem por objetivo agrupar pacotes relacionados.
- **Manifestos de Pacote:** em inglês *Package Manifests*, arquivo nomeado *package.xml* contido na raiz de cada pacote. Seu papel é fornecer metainformações sobre seu pacote: nome, versão, descrição, informações de licença, dependências, entre outras.
- **Tipos de Mensagem:** em inglês *Message Types*, arquivos de extensão *.msg*, localizados dentro da pasta *msg* de um dado pacote. Seu conteúdo define a estrutura de dados de uma mensagem que poderá ser enviado pelo ROS.
- **Tipos de Serviço:** em inglês *Service Types*, arquivos de extensão *.srv*, localizados dentro da pasta *srv* de um dado pacote. Seu conteúdo define a estrutura de dados das mensagens de requisito e resposta de um serviço, as quais poderão ser enviadas pelo ROS.

2.3.1.2 Grafo de computação do ROS

O *grafo de computação do ROS* é uma rede ponto-a-ponto de processos que processam dados conjuntamente. Os conceitos presentes neste nível são:

- **Nós:** em inglês *Nodes*, são processos computacionais que são executados para desempenhar o controle de atuadores, realizar leitura e filtragem de sinais sensoriais ou implementar algoritmos avançados de planejamento e tomada de decisão. É desejável que os nós sejam desenvolvidos da forma mais genérica possível, para sua reutilização em outros projetos. Cada linguagem de programação suportada encapsula as funcionalidades do ROS em uma biblioteca. Para a escrita de um nó na linguagem C++, é utilizada a biblioteca do pacote *roscpp*⁸ e, para escrever um nó em Python, é utilizada a biblioteca contida no pacote *rospy*⁹;
- **Nó Mestre:** em inglês *Master*, fornece cadastro e pesquisa de nome no Grafo de Computação do ROS, ou seja, este nó é responsável por garantir a comunicação entre os nós. Sem a sua execução, não existe comunicação entre os nós.
- **Servidor de Parâmetros:** em inglês *Parameter Server*, parte do Nó Mestre que centraliza a consulta e o armazenamento de dados indexados por uma cadeia de caracteres.

⁸ <<http://wiki.ros.org/roscpp>>

⁹ <<http://wiki.ros.org/rospy>>

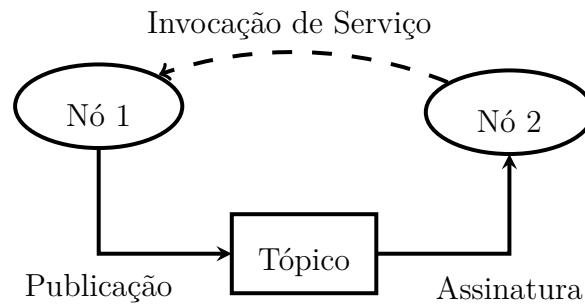


Figura 2 – Conceitos básicos de comunicação do ROS.

- **Mensagens:** em inglês *Messages*, a comunicação entre os nós no ROS consiste no transporte de mensagens, as quais são estruturas de dados que possuem campos tipados. Os campos de uma mensagem podem ser do tipo primitivo (booleano, inteiro, ponto flutuante, caracter, enumerado, cadeia de caracteres), aninhar outras mensagens ou vetores desses tipos.
- **Tópicos:** em inglês *Topics*, são canais que ligam os nós para o transporte de mensagens utilizando a semântica de comunicação *publish/subscribe*. Assim, nós que enviam mensagens para o sistema, as publica no tópico e nós recebem as mensagens ao assinar o tópico. Cada tópico possui um tipo, o que lhe permite transportar apenas este tipo de mensagem. Como característica da sua semântica, vários nós podem publicar e se inscrever no mesmo tópico. E um nó pode publicar e se inscrever em vários tópicos;
- **Serviços:** em inglês *Services*, é um sistema de comunicação no ROS que obedece a semântica *request/reply*. Neste caso, um nó cliente solicita um serviço através de um pedido para um nó servidor que, por sua vez, retorna uma resposta ao nó cliente ao finalizar o serviço prestado.
- **Bolsas:** do inglês *Bags*, são arquivos de extensão *.bag* que contêm dados de mensagens do ROS.

A Figura 2 ilustra os tipos básicos de comunicação entre nós no ROS. Nessa figura, nós são representados por elipses, tópicos por retângulos, conexões entre nó e tópico por setas de linha contínua e invocações de serviço por setas de linha tracejada. Verifica-se assim que o *Nó 1* publica no *Tópico* e o *Nó 2* o subscreve. Além disso, o *Nó 1* é servidor do *Serviço* e o *Nó 2* é seu cliente.

Nós que publicam mensagens em um tópico só estão interessados em disponibilizar a informação, não importando com quem irá utilizá-lo. Da mesma forma, um nó que assina um tópico está apenas interessado em receber a informação disponível no tópico sem se importar com sua fonte. Deste modo, é aconselhado utilizar esse tipo de comunicação na

troca de dados de fluxo contínuo, por exemplo, dados de sensores e sinais de atuação e controle.

Uma invocação de serviço é equivalente a chamada remota de um procedimento. Quando um cliente de serviço solicita um pedido ao seu servidor, ambos ficam aguardando o procedimento finalizar. Com isso, é recomendado o uso desse tipo de comunicação em casos onde o serviço prestado é rápido, como alterações do estado de alguma variável interna.

Vale salientar que muitas mensagens e serviços já foram padronizadas em pacotes do ROS¹⁰.

2.3.1.3 Comunidade do ROS

De modo que comunidades separadas possam trocar código fonte e conhecimento, vários recursos foram criados na *comunidade do ROS*. Tais como:

- **Distribuições:** agrupa coleções de pacotes versionados para facilitar a instalação do ROS. Além disso, é mantido uma versão consistente de cada conjunto de pacotes relacionados.
- **Repositórios:** uma rede federada de repositórios de código permite que instituições diferentes possam desenvolver e lançar componentes de *software* para seus próprios robôs.
- **ROS Wiki**¹¹: é o principal fórum para informações de documentação sobre o ROS. Qualquer pessoa pode solicitar uma conta para contribuir com sua própria documentação, ou ainda fornecer correções e atualizações, bem como, escrever tutoriais.
- **Listas de endereços eletrônicos:** é o meio de comunicação primário entre os usuários de ROS para perguntar sobre questões de *software* do ROS e para receber notificações de novas atualizações.
- **ROS Answers**¹²: é uma página *web* de perguntas e respostas diretamente relacionada ao ROS.
- **Blog**¹³: providencia notícias regularmente com fotos e vídeos.

2.3.1.4 Nome de recurso de grafo

Os recursos de grafo presentes no ROS são: nós, parâmetros, tópicos e serviços. Com o uso adequado da sintaxe de nomes, é possível obter encapsulamento desses recursos

¹⁰ <http://wiki.ros.org/common_msgs>

¹¹ <<http://wiki.ros.org>>

¹² <<https://answers.ros.org/questions/>>

¹³ <<http://www.ros.org/news/>>

através do mecanismo que os nomeia, pois ele gera uma estrutura hierárquica de nomes. Em outras palavras, cada recurso no ROS possui um *namespace* que pode ser compartilhado com vários outros recursos. Normalmente, recursos podem criar outros recursos dentro do seu próprio *namespace* e acessar recursos que estão dentro ou acima dele. Contudo, recursos em camadas inferiores podem ser acessados através da integração de código em *namespaces* superiores. Abaixo, seguem exemplos de nomes de recurso no ROS.

- /
- /rqt_mrta
- /lro/p3dx/pose
- /lro/amigobot/pose
- /lro/alliance

O primeiro exemplo mostra o *namespace* global (/). Todos os recursos com seu respectivo *namespace* estão sob ele. O exemplo seguinte mostra um recurso denominado *rqt_mrta* cujo *namespace* se encontra no nível mais alto. Em seguida, verifica-se três exemplos de recursos que estão sob o *namespace* *lro*. Entretanto, os recursos mostrados nos terceiro e quarto exemplos ainda estão sob um outro *namespace*, *p3dx* e *amigobot*, respectivamente. Note que neste caso, o nome de ambos recursos são iguais (*pose*), porém eles são diferenciados pelos seus *namespaces* (/lro/p3dx e /lro/amigobot, respectivamente). Por último, é dado o recurso cujo nome é *alliance* que se encontra sob o *namespace* /lro.

Existem quatro tipos de resolução de nomes de recursos no ROS: *base*, *relativa*, *global* e *privada*.

- base
- relativa/nome
- /global/nome
- ~privada/nome

Nomes são resolvidos relativamente, então recursos não necessitam estar cientes de qual *namespace* eles se encontram. Isso simplifica a programação como nós que trabalham em conjunto podem ser escritos como se eles estivessem todos no nível de *namespace* mais alto.

A tabela 2 mostra três exemplos de resolução de nomes de recurso de grafo no ROS, cada um nas três variações: relativa, global e privada. À esquerda da seta, encontra-se o nome do recurso e, à sua direita, encontra-se a resolução do seu nome.

Tabela 2 – Exemplos de resolução de nomes no ROS

Nó	Relativa	Global	Privada
/no	img→/no/img	/img→/img	~img→/no/img
/no	img/raw→/no/img/raw	/img/raw→/img/raw	~img/raw→/no/img/raw
/ns/no	img→/ns/no/img	/img→/img	~img→/ns/no/img

Esses conceitos possuem extrema importância em sistemas multirrobo, principalmente naqueles cuja frota de robôs é homogênea. Neste último caso, a partir de replicação das configurações de um robô, todo o sistema pode ser iniciado, variando apenas o *namespace* de cada robô do sistema.

2.3.1.5 Nome de recurso de pacote

O outro tipo de recurso no ROS é encontrado no nível de arquivos do sistema. Seus nomes facilitam a referência de arquivos e tipos de dados em disco. Eles são nomeados com o nome do pacote em que eles estão localizados seguido do seu nome. Por exemplo, o nome *alliance_msgs/Motivation* se refere ao tipo de mensagem *Motivation* do pacote *alliance_msgs*.

2.3.2 Interface gráfica de usuário do ROS

Além de ferramentas disponíveis em terminal via comando de linha, o ROS também disponibiliza ferramentas gráficas cujas funcionalidades são controladas por um *plugin*. Estes são desenvolvidos através do *rqt*¹⁴ que disponibiliza uma interface de programação de aplicação (do inglês, *Application Programming Interface* - API) em C++ e Python para a criação de interface gráfica de usuário (GUI, acrônimo para *Graphical User Interface*) integrada com o ROS. Por sua vez, esta API utiliza o Qt (lê-se *cute*) como seu *kit* de desenvolvimento de *software* (SDK - *Software Development Kit*). A Figura 3 foi extraída da página do metapacote *rqt* e mostra a aplicação de vários *plugins* que foram acoplados em uma mesma janela através do *rqt_gui*¹⁵.

Essas ferramentas são agrupadas em categorias. Entre elas estão:

- **Configuração:** reúne ferramentas relacionadas a execução e configuração de nós, os *plugins* *rqt_launch*¹⁶ e *rqt_reconfigure*¹⁷ são exemplos disso;

¹⁴ <<http://wiki.ros.org/rqt>>

¹⁵ <http://wiki.ros.org/rqt_gui>

¹⁶ <http://wiki.ros.org/rqt_launch>

¹⁷ <http://wiki.ros.org/rqt_reconfigure>

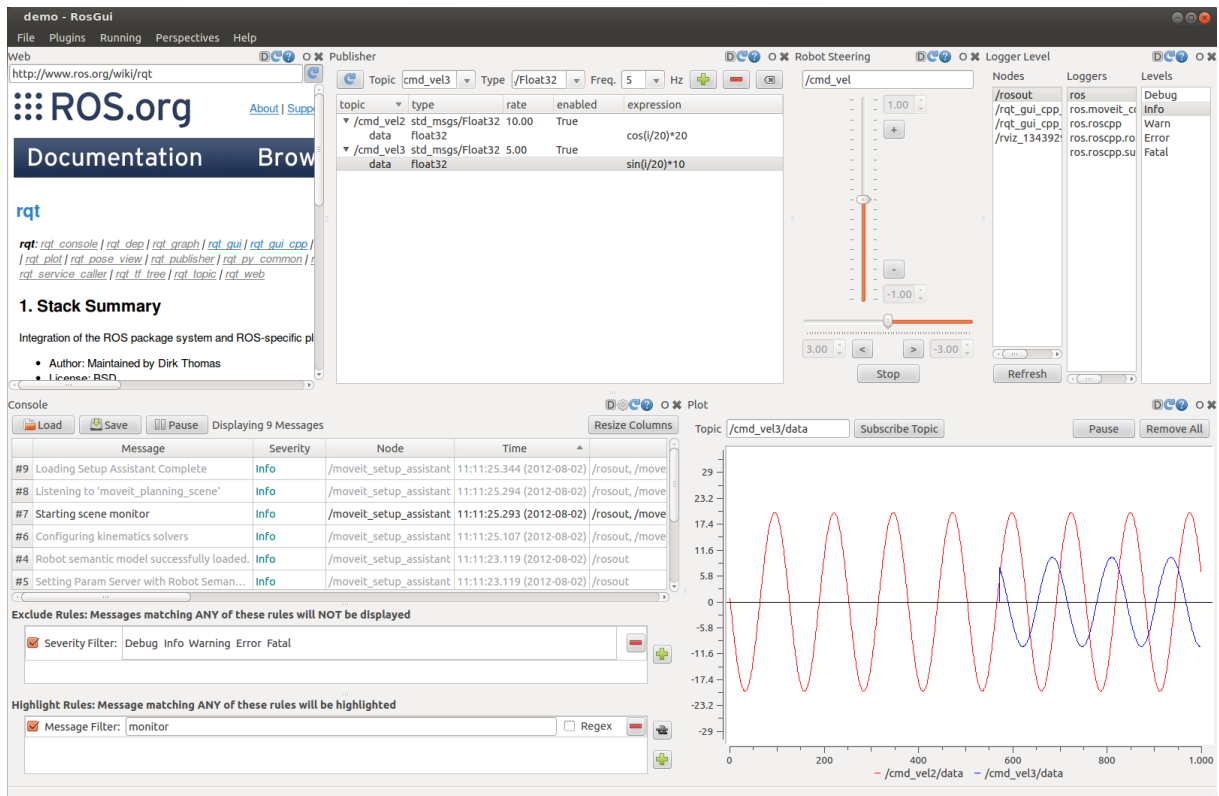


Figura 3 – Exemplo de ferramentas gráficas existentes no ROS.

- **Introspecção:** junta *plugins* para a análise do Grafo de Computação e das dependências entre pacotes;
- **Logging:** agrupa ferramentas para alternar o nível de *log* nos nós e para filtrar *logs*;
- **Tópicos:** são reunidas ferramentas diretamente relacionadas tópicos no ROS, como a publicação de mensagens, monitor de tópico e navegador para definições de mensagem;
- **Serviços:** cliente de serviços e navegador para definições de serviços, são exemplo de ferramentas relacionadas com serviços;
- **Visualização:** agrupa ferramentas que traçam gráficos de dados numéricos no tempo, mostram de imagens publicadas em tópicos e sistemas supervisórios, são exemplos de *plugins* que pertencem a esta categoria *rqt_image_view*¹⁸, *rqt_multiplot*¹⁹ e *rqt_rviz*²⁰;
- e muitas outras.

¹⁸ <http://wiki.ros.org/rqt_image_view>

¹⁹ <http://wiki.ros.org/rqt_multiplot>

²⁰ <http://wiki.ros.org/rqt_rviz>

2.4 Trabalhos Relacionados

Reis e Bastos (2015) elaboraram uma aproximação da arquitetura ALLIANCE dedicada para uma aplicação de patrulhamento simulada. O enfoque desse trabalho foi mostrar os benefícios que o ROS oferece ao tratar aplicações de robôs heterogêneos: fácil comunicação entre robôs heterogêneos, a aplicação pode crescer adicionando novos robôs com poucas modificações e a reutilização de projetos desenvolvidos por outros pesquisadores. A arquitetura desenvolvida neste trabalho não pode ser reutilizada em outros domínios.

A fim de enfrentar os dois grandes desafios na criação de *software* para sistema multirrobô: cooperação distribuída entre robôs e reusabilidade de aplicações em robótica, Li et al. (2016) desenvolveram uma arquitetura tolerante a falhas para a cooperação em sistema multirrobô: ALLIANCE-ROS²¹. Foram encapsulados os mecanismos do ROS e bibliotecas em Python para a criação das funções básicas da arquitetura ALLIANCE (PARKER, 1998). Deste modo, foram combinadas as vantagens da arquitetura ALLIANCE e do *framework* ROS, pois forneceram os seguintes benefícios para os seus usuários: *templates* para a descrição de comportamento e coordenação, métodos tolerantes a falhas para a alocação de tarefas em sistema multirrobô, vários módulos para o ROS e uma interface para programação em Python.

Guidotti et al. (2017) criaram um pacote ROS²² para a arquitetura Murdoch, sugerida por Gerkey e Mataric (2002), para ser utilizada em problemas de alocação de tarefa em sistema multirrobô. Foi disponibilizado uma interface de programação em C++ para a integração de cada robô licitante com o leilão gerenciado pelo leiloeiro da arquitetura. Para verificar a validade do pacote criado, foi simulado um sistema com 4 robôs e 3 tarefas distintas para serem realizadas. Esta aproximação não é totalmente modular.

²¹ <http://wiki.ros.org/micros_mars_task_alloc>

²² <<https://github.com/caueguidotti/Murdoch>>

3 Desenvolvimento

O trabalho de muitos desenvolvedores de arquiteturas de alocação de tarefa para sistemas de vários robôs pode passar despercebido ou ser ignorado por pessoas que buscam resolver esse problema. A falta de uma documentação mínima leva a maioria dos seus usuários a desistir de compreender tais trabalhos. Além disso, a configuração dessas arquiteturas pode deixar seus usuários confusos devido a imensidão de parâmetros existentes. Levando esses fatos em consideração, foi desenvolvido o pacote *rqt_mrta* que implementa uma aplicação gráfica para o auxílio na utilização de arquiteturas MRTA do ROS.

Este capítulo apresenta detalhes pertinentes sobre o desenvolvimento dos pacotes *rqt_mrta* para o *framework* ROS.

O pacote desenvolvido atende dois tipos de usuários: desenvolvedores de arquiteturas MRTA e usuários de arquiteturas MRTA. Desenvolvedores de arquiteturas MRTA podem utilizar este *software* para o registro e definição da arquitetura desenvolvida. Ao fazê-los, sua arquitetura estará disponível para o uso de usuários de arquitetura MRTA através do *rqt_mrta*. Por sua vez, os usuários de arquitetura MRTA podem utilizar o *rqt_mrta* para a definição do seu problema MRTA, para escolher uma arquitetura disponível para uso e também para a configuração da arquitetura escolhida conforme a definição do seu problema. Ao final desse procedimento, a aplicação salva os arquivos necessários para o uso da arquitetura escolhida em um novo pacote ROS.

Para isso, existem dois arquivos de configuração, um para a configuração da arquitetura e outro para a configuração do problema. Ambos arquivos possuem a extensão XML (*Extensible Markup Language*). Ao serem carregados pela aplicação, esta se adapta para tratar o problema MRTA definido, utilizando a arquitetura escolhida.

Esse pacote está em conformidade com a convenção¹ de nomes de pacote do ROS. O nome de pacotes que proveem ferramentas GUI baseada em Qt para o ROS deve ter o prefixo “*rqt_*”. Esta interface gráfica utiliza a API em C++ do *framework* *rqt* e pode ser utilizada como uma aplicação *standalone*² ou, então, como um *plugin* que pode ser acoplado em uma janela do *rqt_gui*³ juntamente com outras ferramentas gráficas (vide Seção 2.3.2) do ROS.

Este projeto foi concebido utilizando o padrão arquitetural MVC (*Model-View-Controller*), sendo assim dividido em três camadas: dados (*model*), apresentação (*view*)

¹ <<http://wiki.ros.org/ROS/Patterns/Conventions>>

² Tipo de programa que não precisa de *software* auxiliar para a sua execução.

³ <http://wiki.ros.org/rqt_gui>

e regra de negócio (*controller*).

Serão detalhados a seguir os arquivos de configuração de arquitetura e aplicação, bem como, as camadas do modelo, de visualização e controle.

3.1 Arquivo de configuração de arquitetura

O arquivo de configuração de arquitetura tem como propósito:

- classificar a arquitetura segundo a taxonomia sugerida por [Gerkey e Mataric \(2004\)](#);
- identificar os parâmetros necessários para sua configuração inicial;
- identificar os arquivos necessários para a sua inicialização;
- associar os arquivos de parâmetros com os arquivos de inicialização;
- identificar as ferramentas gráficas para a análise do desempenho da arquitetura em tempo de execução.

3.1.1 Formatação

Esse arquivo de extensão XML conta com quatro *tags* principais: *architecture*, *configs*, *launches* e *widgets*. É mostrada a seguir uma simplificação de um arquivo de configuração de arquitetura. No entanto, o Apêndice A apresenta um exemplo completo, contendo as configurações da arquitetura implementada no pacote ROS *alliance*, o qual será discutido na Seção 4.

```
<rqt_mrta>
  <architecture>
    <name>ALLIANCE</name>
    <robots>
      <type>ST</type>
      <busy_robots>
        <topic>
          ...
        </topic>
      </busy_robots>
      <config_id>alliance_params</config_id>
      <idle_robots>
        <topic>
          ...
        </topic>
      </idle_robots>
    </robots>
  </architecture>
</rqt_mrta>
```

```

    </idle_robots>
    <launch_id>alliance</launch_id>
</robots>
<tasks>
  <type>SR</type>
  <incoming_tasks>
    <topic>
      ...
    </topic>
  </incoming_tasks>
</tasks>
<allocations>
  <type>IA</type>
  <allocated_tasks>
    <topic>
      ...
    </topic>
  </allocated_tasks>
</allocations>
</architecture>
<configs>
  <config_0>
    <id>alliance_params</id>
    ...
  <config_0>
    ...
</configs>
<launches>
  <launch_0>
    <id>alliance</id>
    ...
  </launch_0>
  ...
</launches>
<widgets>
  <widget_0>
    <plugin_name></plugin_name>
  </widget_0>
  ...

```

```

</widgets>
</rqt_mрта>

```

A *tag architecture* possui as *tags name*, *robots*, *tasks* e *allocations*, as quais contêm, respectivamente, informações sobre o nome da arquitetura e considerações sobre os robôs, tarefas e alocações do problema MRTA. A *tag robots* agrupa informações sobre o tipo dos robôs do problema MRTA que ela resolve e, através da *tag topic*, dá detalhes pertinentes para a verificação da alteração de estado dos robôs. A *tag topic* será explicada a seguir. A *tag tasks*, por sua vez, traz dados sobre o tipo de tarefa que a arquitetura considera, bem como, informações para a identificação de novas tarefas que surgem no sistema. A *tag allocations* classifica o tipo de alocação tratada pela arquitetura, assim como, traz informações para a verificação de alteração de estado da alocação. Essas informações são úteis para ajudar o usuário a escolher uma arquitetura válida para o seu problema MRTA. Além disso, elas permitirão a identificação dos estados dos robôs, tarefas e alocações em tempo de execução.

```

<topic>
  <name>/allocations</name>
  <type>mрта_msgs/Allocation</type>
  <field>header/frame_id</field>
  <timeout>2.5</timeout>
  <queue_size>10</queue_size>
</topic>

```

As *tags busy_robots*, *idle_robots*, *incoming_tasks* e *allocated_tasks* têm a *tag topic* dentro do seu escopo. Esta *tag*, por sua vez, possui cinco *tags*: *name*, *type*, *field*, *timeout* e *queue_size*. Elas informam o nome do tópico, o tipo da mensagem transportada pelo tópico, o campo da mensagem, o tempo máximo considerado sem receber uma nova mensagem e o tamanho do *buffer* de mensagens. Através dessas informações, a aplicação é capaz de identificar o estado dos robôs, tarefas e alocações do sistema.

A *tag configs* reúne um grupo de *templates* para a geração de arquivos de parâmetros que serão utilizados na inicialização dos nós da arquitetura. Esses arquivos de parâmetros possuem extensão YAML (*YAML Ain't Markup Language*). Cada *template* é definida dentro de uma *tag config*, a qual possui um identificador que pode ser associado a um robô. Nessa situação, será pedido ao usuário durante a criação de um novo problema o preenchimento dos parâmetros para cada robô do sistema. Aqueles que não são associados aos robôs necessitam ser preenchidos apenas uma vez. As seguintes *tags* são utilizadas para a criação de uma *template*:

- *id*: identifica o arquivo de parâmetros, seu valor pode ser usado para associar essa

template com os robôs;

- *param*: traz informações importantes sobre o preenchimento de dados de um parâmetro. Esta *tag* contém as *tags* *name*, *type*, *value*, *default* e *tool_tip*. Quando a *tag* *default* não é dada, o parâmetro em questão é considerado obrigatório;
- *params*: armazena uma coleção de parâmetros, seus filhos podem ser *tags* do tipo *param*, *params* e *array*;
- *array*: armazena uma coleção de parâmetros, seus filhos podem ser *tags* do tipo *param*, *params* e *array*; além disso, deve ser filha de uma *tag* *params* e ter uma *tag* *param* irmã cujo valor da *tag* *name* deve ser *size*. Através deste parâmetro, é possível identificar o número de vezes que a coleção de parâmetros armazenados pela *tag* *array* irá se repetir;
- *name*: informa o nome do parâmetro;
- *type*: informa o tipo do valor do parâmetro, assim seus valores podem ser *bool*, *int*, *double* ou *string*;
- *value*: armazena o valor do parâmetro;
- *tool_tip*: traz uma instrução para contextualizar o usuário durante o preenchimento do parâmetro.

É mostrado a seguir um trecho da *tag* *configs* extraído do Apêndice A.

```
<configs>
  <config_0>
    <id>alliance_params</id>
    <param_0>
      <name>name</name>
      <type>string</type>
      <tool_tip>Enter the robot name.</tool_tip>
    </param_0>
    ...
  <params_6>
    <name>behaviour_sets</name>
    <param_0>
      <name>size</name>
      <type>int</type>
      <value>@array_size@</value>
      <tool_tip>Enter the number of behaviour sets that the
        robot have.</tool_tip>
```

```

    </param_0>
    <array_1>
      <name>behaviour_set@index@</name>
      <param_0>
        <name>task_id</name>
        <type>string</type>
        <tool_tip>Enter the task id in which this behaviour
          set is related to.</tool_tip>
      </param_0>
      ...
    </array_1>
  </params_6>
  ...
</config_0>
...
</configs>

```

A *tag launches* é uma coleção de *templates* para a geração de arquivos de inicialização de nós do ROS. A extensão desses arquivos é do tipo LAUNCH, os quais podem ser utilizados pela ferramenta *roslaunch*⁴ para a inicialização do sistema no ROS. Cada *template* de arquivo de inicialização é definida por uma *tag launch*. Para incluir um arquivo de inicialização dentro da *template* deve-se utilizar uma *tag include*, indicando seus argumentos através da *tag arg* e a localização do arquivo desejado. As demais *tags* de um arquivo de inicialização do ROS não são suportadas, pois todos os demais recursos podem ser utilizados dentro do arquivo incluído pela *template*. É mostrado a seguir um trecho da *tag launches* extraído do Apêndice A.

```

<launches>
  <launch_0>
    <id>alliance</id>
    <includes>
      <include_0>
        <file>$(find alliance)/launch/alliance.launch</file>
        <args>
          <arg_0>
            <name>robot_id</name>
            <value>@robot_id@</value>
          </arg_0>
          <arg_1>

```

⁴ <http://wiki.ros.org/roslaunch>

```

        <name>robot_params</name>
        <value>$(find @package@)/config/
            @robot_id@_alliance_params.yaml</value>
    </arg_1>
    ...
</args>
</include_0>
...
</includes>
</launch_0>
...
</launches>

```

Finalmente, pode ser informado ao *plugin* uma coleção de outros *plugin* desenvolvidos para a análise da arquitetura em tempo de execução. Para isso, deve-se usar a *tag widgets* informando o nome de cada *plugin*.

3.1.2 Registro da arquitetura

No entanto, torna-se necessário um mecanismo de busca capaz de identificar pacotes ROS que contenham arquivos de configuração de arquitetura; pois, assim, será possível auxiliar o usuário final a escolher uma arquitetura que resolva o seu problema MRTA.

A biblioteca *rospack*⁵ possui diversas ferramentas de busca de informação de pacotes existentes no sistema de arquivos do ROS. Dentre elas, a ferramenta *rospack plugin* examina os pacotes ROS procurando por aqueles que dependem diretamente do pacote dado, extraíndo de cada um deles seu nome acompanhado pelo valor do atributo requisitado. Logo, é possível utilizar esse mecanismo em benefício deste projeto ao adicionar uma dependência do *rqt_mrt*a ao pacote da arquitetura, assim como, exportar a localização relativa do seu arquivo de configuração.

Portanto, para que o *rqt_mrt*a tenha visibilidade das arquiteturas configuradas, são necessárias algumas alterações no arquivo manifesto do pacote que implementa cada uma delas. Primeiramente, é necessário adicionar a dependência em tempo de execução do pacote *rqt_mrt*a. Para isso, é necessário adicionar ao arquivo *package.xml* do pacote desejado a seguinte linha:

```
<run_depend>rqt_mrt</run_depend>
```

A segunda alteração necessária exportará a localização do arquivo de configuração dentro do pacote em questão. Para isso, deve-se adicionar a seguinte linha dentro do

⁵ <<http://wiki.ros.org/rospack>>

escopo da *tag export* do arquivo *package.xml*:

```
<rqt_mrta architecture="${prefix}/rqt_mrta.xml"/>
```

Note que, neste caso, deve ser requisitado o atributo *architecture*. Será visto adiante que este atributo tem outro nome quando se trata de arquivos de configuração de aplicação.

Dessa forma, o arquivo manifesto do pacote (*package.xml*) deve conter, obrigatoriamente, as linhas abaixo. Deve-se respeitar a hierarquia das *tags*.

```
<package>
  ...
  <run_depend>rqt_mrta</run_depend>
  ...
  <export>
    ...
    <rqt_mrta architecture="${prefix}/plugin.xml"/>
    ...
  </export>
  ...
</package>
```

O Apêndice C mostra um exemplo de arquivo manifesto de um pacote configurado para ser uma arquitetura MRTA do ponto de vista do *rqt_mrta*. Note a dependência em tempo de execução do pacote *rqt_mrta*, bem como, a exportação da sua *tag* com o atributo *architecture*.

3.2 Arquivo de configuração de aplicação

Um arquivo de configuração de aplicação informações gerais da aplicação, identifica os robôs do sistema e armazena as escolhas feitas pelo usuários durante a parametrização da arquitetura. Será apresentado o formato deste arquivo e também como este arquivo é reconhecido pelo *rqt_mrta*.

3.2.1 Formatação

Esse arquivo de extensão XML conta com três *tags* principais: *application* e *configs*, *launches*. É mostrada a seguir uma simplificação de um arquivo de configuração de aplicação. No entanto, o Apêndice B apresenta um exemplo completo, contendo as configurações da aplicação *patrulha*, a qual é discutida na Seção 4.

```
<rqt_mrta>
```



```
<application>
  <architecture>alliance</architecture>
  <name>Patrulha na UNIFEI</name>
  <robots>
    <robot_0>
      <id>/robot1</id>
      <tasks>
        <task_0>
          <id>wander</id>
        </task_0>
        ...
      </tasks>
    </robot_0>
    ...
  </robots>
</application>
<configs>
  ...
</configs>
<launches>
  ...
</launches>
</rqt_mrta>
```

A tag *application* armazena metadados sobre o problema MRTA e composta pelas seguintes *tags*:

- *architecture*: armazena o nome do pacote que implementa a arquitetura escolhida;
- *name*: armazena o nome da aplicação;
- *robots*: agrupa uma coleção de *tags* do tipo *robot*;
- *robot*: possui uma *tag id* e uma *tag tasks*;
- *tasks*: reúne um conjunto de *tags* do tipo *task*;
- *task*: possui uma *tag id*;
- *id*: identifica robôs e tarefas no sistema.

A tag *application* agrupa metadados sobre o problema, como o nome da aplicação e o nome do pacote que implementa a arquitetura MRTA escolhida. Além disso, a tag *robots*

identifica os robôs do sistema. Este arquivo também armazena os parâmetros adotados durante a criação do problema MRTA. Isso permite que os seus valores possam ser atualizados. Por fim, este arquivo também guarda a estrutura dos arquivos de inicialização do ROS que foram gerados após sua criação.

As *tags configs* e *launches* permitem a reconfiguração da arquitetura e da aplicação após sua geração. Elas possuem a mesma estrutura descrita em 3.1. Porém, este arquivo não possui *tags* do tipo *array*, pois cada uma delas foi substituída por uma coleção de *tags* do tipo *params*, conforme o exemplo extraído do Apêndice B que se segue.

```
<configs>
  <config_0>
    ...
    <params_5>
      <name>sensors</name>
      <param_0>
        <default></default>
        <name>size</name>
        <tool_tip>Enter the number of sensors that the robot
          have.</tool_tip>
        <type>int</type>
        <value>2</value>
      </param_0>
      <params_1>
        <name>sensor0</name>
        <param_0>
          <default></default>
          <name>plugin_name</name>
          <tool_tip>Enter the sensor plugin name.</tool_tip>
          <type>string</type>
          <value>alliance_test/odometry</value>
        </param_0>
        ...
      </params_1>
      <params_2>
        <name>sensor1</name>
        <param_0>
          ...
        </param_0>
        ...
      </params_2>
```

```
    </params_5>
    ...
  </config_0>
  ...
</configs>
```

3.2.2 Registro da aplicação

Semelhante às configurações de arquitetura, é desejável que o *plugin rqt_mrta* tenha visibilidade dos pacotes que contêm os dados e as configurações de um problema MRTA. Novamente, são necessárias algumas alterações no arquivo *package.xml* do pacote desejado, semelhante àsquelas realizadas para um pacote que implementa uma arquitetura MRTA. Entretanto, a única diferença está no nome do atributo da *tag* de exportação do *rqt_mrta*, a qual passa a ser *application* ao invés de *architecture*, conforme visto abaixo.

```
<package>
  ...
  <run_depend>rqt_mrta</run_depend>
  ...
  <export>
    ...
    <rqt_mrta application="${prefix}/plugin.xml"/>
    ...
  </export>
  ...
</package>
```

O Apêndice D mostra um exemplo de arquivo manifesto de um pacote configurado para ser um problema MRTA do ponto de vista do *rqt_mrta*. Note a dependência em tempo de execução do pacote *rqt_mrta*, bem como, a exportação da sua *tag* com o atributo *application*.

3.3 Camada do modelo

A camada de modelo facilita a importação e exportação dos dados dos arquivos de configuração detalhados em 3.1 e 3.2. Conforme mostra a Figura 4, a relação entre as classes do modelo se apresenta como um grafo, semelhantemente à estrutura dos arquivos de configuração de arquitetura e aplicação. Ao utilizar o padrão de projeto de *software* cadeia de responsabilidade (GAMMA et al., 1993) nesta organização de clas-

ses, simplificou-se a geração dos arquivos de inicialização, de parâmetros, configuração de arquitetura e aplicação, assim como, a leitura desses dois últimos tipos de arquivo.

Cada classe mostrada na Figura 4 representa uma *tag* não-folha. E as *tags* folhas são membros de alguma classe do modelo. Algumas dessas classes possuem métodos utilitários que ajudam na geração de arquivos. São elas:

- *RqtMrtaArchitecture*: possui métodos que facilitam a leitura e escrita de arquivos de configuração de arquitetura;
- *RqtMrtaApplication*: possui métodos que facilitam a leitura e escrita de arquivos de configuração de aplicação;
- *Launches*: possui método que auxiliam na chamada pela geração e organização dos arquivos de inicialização de uma configuração de aplicação;
- *Launch*: possui um método que facilita a escrita de arquivos de inicialização (na extensão *.launch*);
- *Configs*: possui métodos que auxiliam na chamada pela geração e organização dos arquivos de parâmetros de uma configuração de aplicação;
- *Config*: possui um método que facilita a escrita de arquivos de parâmetros (na extensão *.yaml*).

3.4 Camada de visualização

O Qt (YAFEI, 2012) possui uma ferramenta chamada *Qt Designer* que permitiu o desenvolvimento da camada de apresentação (*view*) deste projeto, pois ela representa graficamente a disposição dos componentes Qt para a customização de janelas (*widgets*, *dialogs* e *wizards*). A partir dessas representações foram gerados arquivos de extensão UI (*User Interface*) que armazenam a árvore de relação entre os componentes Qt e formata os dados em XML. Finalmente, em tempo de compilação, os arquivos UI foram convertidos para classes codificadas em C++.

3.4.1 Widget principal do *rqt_mrta*

A Figura 5 mostra o *widget* principal do *plugin* *rqt_mrta*. No parte superior da janela se localiza a barra de ferramentas, a qual possui os botões com as seguintes função:

- **Nova aplicação:** abre um novo *wizard* para a criação de uma nova aplicação. Se este *wizard* é finalizado com sucesso, o *widget* principal do *plugin* se adapta conforme as configurações da nova aplicação criada;

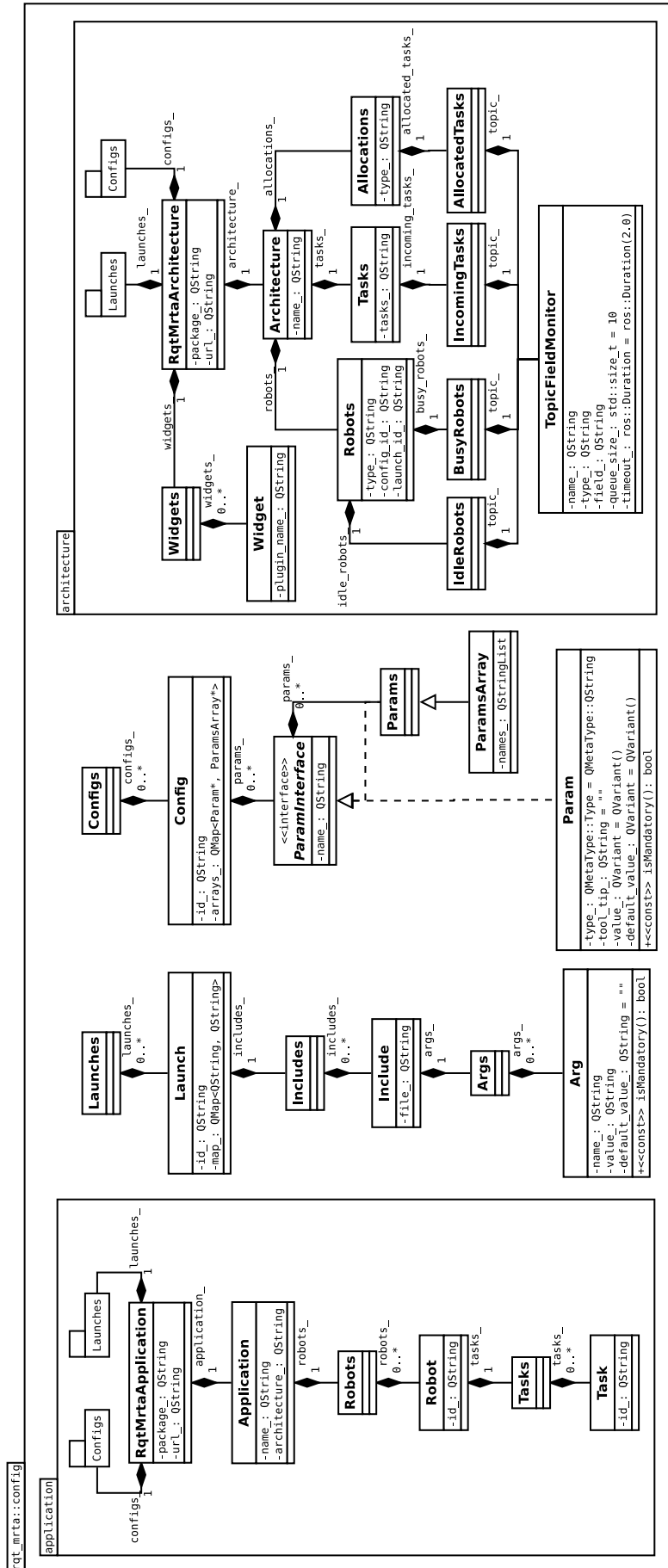


Figura 4 – Diagrama de classes da camada do modelo.

- **Carregar aplicação:** abre um *dialog* para a seleção de uma aplicação já criada anteriormente. Se uma das aplicações listadas é selecionada, então o *widget* principal do *plugin* se adapta conforme as configurações desta aplicação;
- **Executar aplicação:** executa os arquivos de inicialização da aplicação carregada. Esta função não está habilitada ainda;
- **Nova arquitetura:** abre um novo *wizard* para a configuração e cadastro de uma nova arquitetura. Se este *wizard* é finalizado com sucesso, o *widget* principal do *plugin* se adapta conforme as configurações da nova arquitetura cadastrada. Esta função não está habilitada ainda;
- **Carregar arquitetura:** abre um *dialog* para a seleção de uma arquitetura já cadastrada. Se uma das arquiteturas listadas é selecionada, então o *widget* principal do *plugin* se adapta conforme as configurações desta arquitetura, isto é, serão carregados os *widgets* fornecidos pela arquitetura.

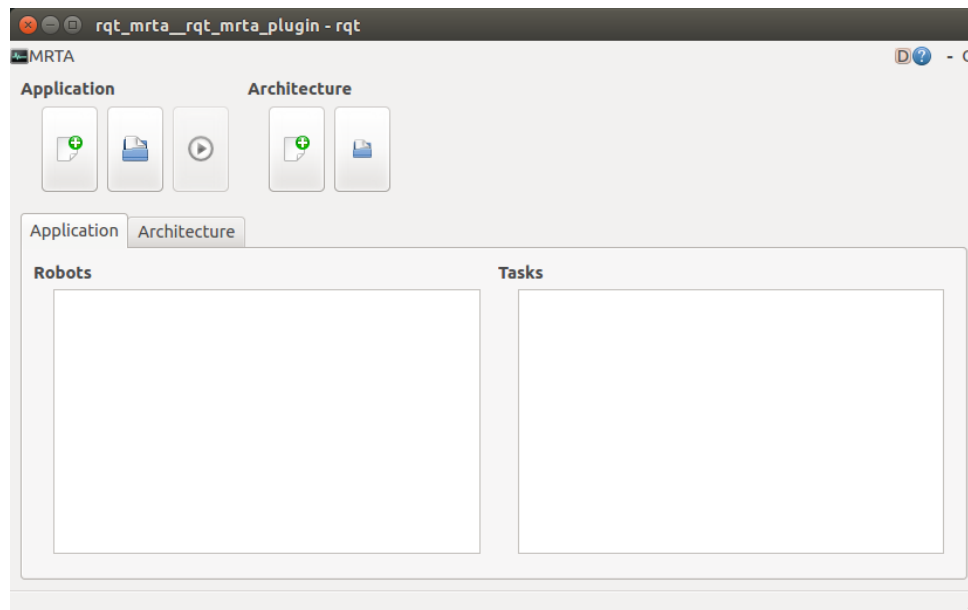


Figura 5 – Janela principal do *plugin rqt_mrta*.

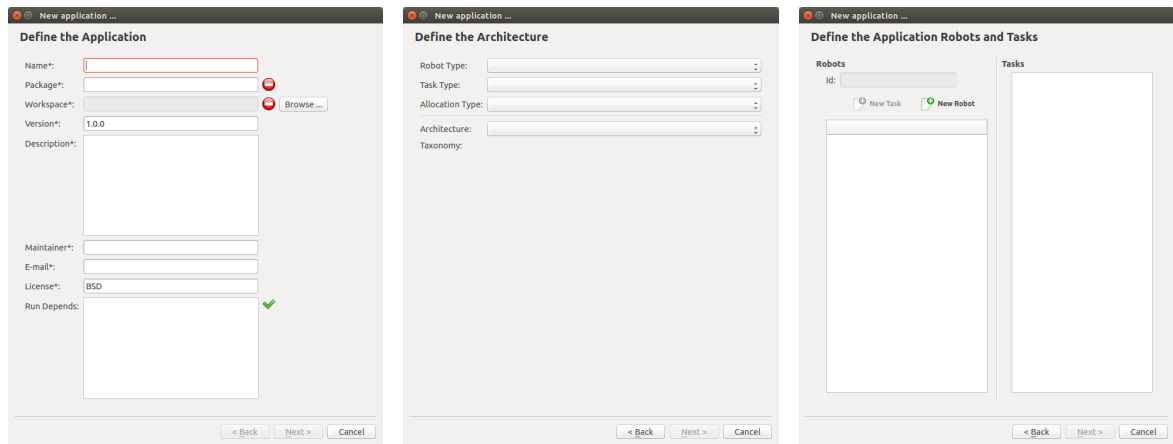
A parte central da janela mostra dois painéis, o painel da esquerda mostra o estado de cada robô do sistema e o painel da direita mostra o estado das tarefas do sistema.

3.4.2 Wizard para criação de nova aplicação

A Figura 6 mostra as telas do *wizard* para a criação de uma nova aplicação, onde são definidos seus dados gerais, é selecionado uma das arquiteturas registradas, são definidos os robôs do sistema, bem como, as tarefas que cada um é capaz de realizar. Em seguida, cada arquivo necessário para parametrizar a arquitetura escolhida é preenchido

e, finalmente, é mostrado quais arquivos e pastas serão criados a partir das informações dadas.

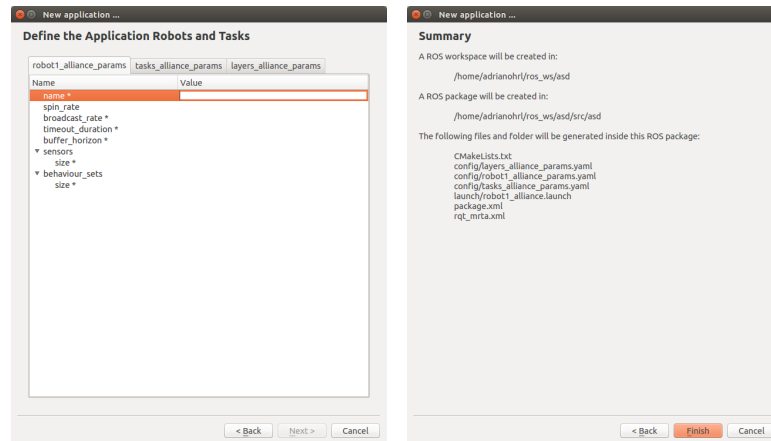
Ao final deste procedimento, será criado um pacote contendo o arquivos manifesto (configurado conforme 3.2.2), CMakeLists.txt e de configuração de aplicação. Em seguida, são gerados os arquivos de parâmetro e de inicialização a partir dos dados inseridos pelo usuário. Os arquivos de parâmetro serão agrupados na pasta *config* do pacote gerado e os arquivos de inicialização serão agrupados na pasta *launch* do pacote gerado.



(a) Dados gerais da aplicação.

(b) Escolha da arquitetura.

(c) Definição dos robôs do sistema.



(d) Parametrização da arquitetura.

(e) Sumário.

Figura 6 – *Wizard* para a criação de uma nova aplicação.

A primeira tela do *wizard* de novas aplicações, mostrado na Figura 6a, coleta os dados gerais da aplicação. Estes dados serão utilizados na geração do arquivo manifesto do pacote. Caso o *workspace* dado não seja um *workspace* do ROS, o usuário é alertado de que será criado um novo *workspace* no diretório especificado. Deve-se atentar para o preenchimento correto do endereço eletrônico do mantenedor do pacote. Caso contrário, o pacote não poderá ser compilado. Após o preenchimento de todos os dados obrigatórios, o usuário pode prosseguir para a próxima tela.

Em seguida, pede-se para o usuário escolher uma arquitetura, conforme Figura 6b. A escolha pode ser realizada com o auxílio de três filtros: pelo tipo dos robôs (ST *versus* MT), pelo tipo das tarefas (SR *versus* MR) e pelo tipo das alocações (IA *versus* TA), conforme a taxonomia revisada na Seção 2.2.2. Assim que o usuário passa para a próxima tela, o *rqt_mrta* carrega o arquivo de configuração da arquitetura selecionada. Logo, os *templates* dos arquivos de parâmetros e de inicialização estarão disponíveis na memória para a geração dos arquivos necessários para a aplicação.

Após a seleção da arquitetura, pede-se para identificar os robôs do sistema. A interface para a entrada desses dados é mostrada na Figura 6c. A partir dos robôs e dos *templates* de arquivos de parâmetros do arquivo de configuração de arquitetura carregado, pede-se para o usuário preencher os arquivos de parâmetros para cada robô e os demais arquivos que não são associados diretamente aos robôs, conforme mostra a Figura 6d.

Os campos que aparecem o símbolo asterisco (*), identificam para o usuário quais entradas devem ser preenchidas obrigatoriamente. O usuário fica impossibilitado de avançar para a próxima tela enquanto ele não tenha preenchido todas entradas obrigatórias. Isso garante que ao final do procedimento, a arquitetura estará devidamente configurada e será possível gerar o pacote da aplicação com sucesso.

Na última tela do *wizard*, após o preenchimento de todos parâmetros obrigatórios, o usuário pode ver um resume das ações que serão realizadas pelo *plugin* ao concluir o procedimento. Conforme mostra a Figura 6e, esta tela informa se o *plugin* iniciará um novo *workspace* do ROS (conforme necessidade), mostra o diretório em que o pacote da aplicação será criado, bem como, os arquivos e pastas que serão gerados neste pacote.

3.4.3 Dialogs para a seleção de arquiteturas e aplicações

As Figuras 8a e 8b mostram, respectivamente, as janelas para a seleção de uma arquitetura e de uma aplicação para o carregamento do seu arquivo de configuração. Ao invés dos usuários navegarem pelos diretórios do sistema operacional procurando o arquivo desejado, são listados para ele os pacotes que foram devidamente configurados, conforme descrito nas Seções 3.1 e 3.2, respectivamente.

3.5 Camada de controle

A camada de controle auxilia na supervisão do sistema em tempo de execução. Esta camada encapsula classes que monitoram os tópicos do ROS para a verificação de alteração do estado dos robôs, tarefas e alocações do sistema. Deste modo, a camada de visualização só se preocupa em exibir graficamente os estados dessas entidades no *widget* principal do *rqt_mrta*.



Figura 7 – Pastas e arquivos gerados após a criação de uma aplicação.



Figura 8 – Carregando um arquivo de configuração

A Figura 9 mostra o diagrama UML (*Unified Modeling Language*) que relaciona as classes do modelo utilizado na camada de controle do *rqt_mrta*. Será detalhada a seguir cada classe contida neste diagrama.

As classes principais deste modelo são: *System*, *Problem*, *Robot*, *Task*, *Allocation* e *Architecture*, cuja relação se dá mediante a definição de um problema de atribuição de tarefa em sistema multirrobo. Logo, como o sistema possui vários robôs e um problema de alocação de tarefa para ser resolvido, um objeto do tipo *System* é composto por vários objetos do tipo *Robot* e também de uma instância de objeto da classe *Problem*. A classe *Problem* tem como responsabilidade relacionar cada tarefa a ser executada com um robô (caso o tipo das tarefas do sistema seja SR) ou um grupo de robôs (caso o tipo das tarefas do sistema sejam MR) através de uma alocação. Logo, um objeto do tipo *Problem* é composto por vários objetos do tipo *Task*, vários do tipo *Allocation* e uma instância de *Architecture*. A classe *Architecture* apenas armazena a classe de problema que pode ser

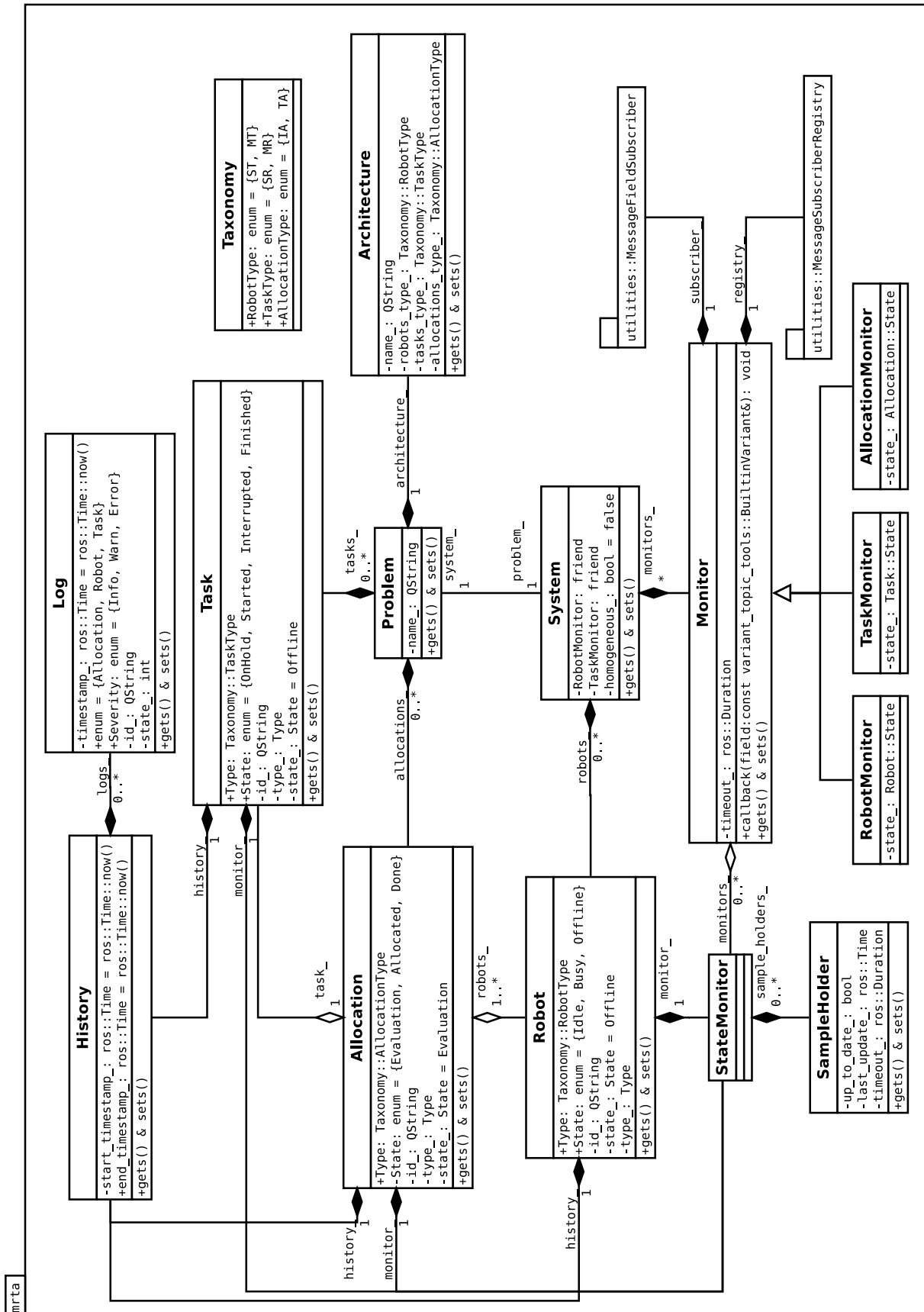


Figura 9 – Diagrama de classes do camada de controle.

resolvido pela arquitetura MRTA escolhida.

As classes *Robot*, *Task* e *Allocation* são muito parecidas. Elas mantêm uma identificação única para cada robô, tarefa e alocação identificados no sistema, respectivamente. Contudo, objetos da classe *Allocation* têm uma instância de *Task* e pode ter um ou vários objetos *Robot*, dependendo do tipo das tarefas que a arquitetura MRTA escolhida resolve. Cada uma dessas três classes ainda possuem um objeto do tipo *History* que armazena *logs* das alterações de estado, o qual é monitorado pelo objeto *StateMonitor* que elas possuem. Objetos *StateMonitor*, por sua vez, são compostos por vários objetos *SampleHolder*, um para cada estado sendo monitorado. Esta classe funciona como se fosse um demultiplexador, direcionando o evento na sua entrada para a saída apropriada. Isto é, se o objeto *StateMonitor* de um dado robô recebe a informação que ele se encontra no estado *Busy* (ocupado), esse encaminha esta informação para o objeto *SampleHolder* que mantém o histórico de notificações desse estado. Portanto, a classe *SampleHolder* é responsável por identificar as rampas de subida e de descida de um dado estado do objeto em monitoramento. Este objeto leva em consideração um parâmetro (*timeout*) que especifica o tempo máximo considerado para manter o dado estado em nível lógico alto.

Voltando a classe *System*, objetos desse tipo ainda são compostos por vários objetos do tipo *Monitor* que pode ser especificado para os tipos *RobotMonitor*, *TaskMonitor* e *AllocationMonitor*. Cada um tem a função de monitorar um campo específico de uma dada mensagem provida de um tópico específico, conforme os parâmetros do arquivo de configuração da arquitetura MRTA escolhida. Como objeto *Monitor* analisa um estado específico de um conjunto de entidades da mesma natureza. Por exemplo, objetos *RobotMonitor* analisam um dado estado dos robôs do sistema; objetos do tipo *TaskMonitor* analisam um estado específico para as tarefas do sistema; e, por fim, um estado específico das alocações do sistema é analisado em objetos do tipo *AllocationMonitor*. Assim, de forma semelhante à classe *StateMonitor*, as classes do tipo *Monitor* têm um papel similar a um demultiplexador, entregando a notificação recebida ao objeto *StateMonitor* apropriado. Exemplificando, seja um objeto *RobotMonitor* que observa o tópico */busy_robots* onde os robôs do sistema publicam quais atividades eles estão desempenhando. Ao ser notificado da chegada de uma mensagem cujo o remetente é o robô *robot1*, este monitor encaminhará uma notificação para o objeto *StateMonitor* do *robot1* dizendo que ele se encontra no estado *Busy*. Por sua vez, o objeto *StateMonitor* de *robot1* direciona esta notificação para o *SampleHolder* que mantém o histórico do estado *Busy* de *robot1*. Finalmente, a cada rampa de subida ou descida de um dos seus estados, *robot1* atualiza seu estado atual.

Perceba que a classe *Monitor* faz uso de duas classes utilitárias:

- *utilities::MessageSubscriberRegistry*: faz uso da biblioteca *variant_topic_tools*⁶ para a subscrição em tópicos cujo tipo é desconhecido em tempo de compilação. Através do mecanismo *signal*⁷ - *slot*⁸ do Qt (YAFEI, 2012), sempre quando uma nova mensagem é recebida, essa classe notifica todas as outras que têm interesse no recebimento de mensagens de um dado tópico;
- *utilities::MessageFieldSubscriber*: utiliza a classe *utilities::MessageSubscriberRegistry* para a notificação do recebimento de mensagens de um tópico particular. Ao ser notificado, essa classe avalia se o valor do campo especificado confere com o valor desejado. Sempre que esta condição é satisfeita, é emitido um *signal* deste evento para as classes interessadas.

O uso dessas classes utilitárias pelo *Monitor* garante flexibilidade no cadastro das arquiteturas, pois não é necessário fazer adaptações para que a arquitetura se adéque a um suposto padrão estipulado pelo *rqt_mрта*.

⁶ <<https://github.com/ethz-asl/variant>>

⁷ Um *signal* é emitido quando um evento particular ocorre em um *QObject*.

⁸ Um *slot* é uma função de um *QObject* que é chamada em resposta a um *signal* específico.

4 Experimentos e Resultados

Para testar o pacote *rqt_mrta*, será necessário uma arquitetura desenvolvida sobre o *framework* ROS. Primeiramente, será necessário compreendê-la e entender como parametrizá-la. A partir disso, será possível estruturar os *templates* de arquivos de configuração e de inicialização, isto é, criar seu arquivo de configuração, conforme descrito em 3.1.1. Tendo este arquivo em mãos, é preciso alterar o arquivo manifesto do seu pacote para que a configuração possa ser vista pelo *rqt_mrta*. Este processo foi descrito em 3.1.2. Finalmente, a arquitetura estará cadastrada e pronta para ser configurada por usuários na criação de aplicações.

Além de uma arquitetura, será necessário a elaboração de uma aplicação que utilize a arquitetura cadastrada. Pois, assim, será validado o processo de criação da aplicação. Isto é, deverá ser verificado se o pacote da nova aplicação foi gerado com sucesso. Para isso, deve se verificar se ele é visível pelas ferramentas de busca do ROS e se a formatação dos arquivos de parâmetro, de inicialização e de configuração de aplicação está correta. Se tudo estiver em conformidade, então a aplicação estará pronta para a execução. É preciso, agora, executar a aplicação utilizando os arquivos de inicialização gerados pelo *rqt_mrta*. Para isso, será utilizada a ferramenta *roslaunch* para a inicialização dos nós da aplicação.

Finalmente, será verificado o sistema supervisorio do *rqt_mrta*. Para isso, a aplicação deverá ser carregada pelo *rqt_mrta*, o qual ficará aguardando a aplicação ser executada. Nesta etapa será verificada a alteração do estado dos robôs no sistema e, se fornecido, a inicialização do *plugin* para o monitoramento da arquitetura.

Pelo fato de haver poucas aproximações genéricas de arquitetura MRTA para aplicações baseadas em ROS, foi desenvolvido o pacote *alliance*. Esse pacote faz uma aproximação independente do domínio da arquitetura tolerante a falhas ALLIANCE (PARKER, 1998) para atribuição de tarefa em sistema multirrobo. Esta arquitetura será utilizada para testar o funcionamento do pacote *rqt_mrta*.

Será criado, através do *rqt_mrta*, uma aplicação que utilize a arquitetura desenvolvida no pacote *alliance* para a atribuição de tarefa no sistema. Será simulado a patrulha realizada por múltiplos robôs.

Será descrito a seguir o desenvolvimento da arquitetura ALLIANCE no pacote *alliance*. Em seguida, esta arquitetura será cadastrada para sua utilização por aplicações no *rqt_mrta*. Depois, a aplicação será explicada e criada através do *rqt_mrta*. Por fim, os arquivos de inicialização da arquitetura serão executados em conjunto com o simulador e os demais nós para a execução da aplicação.

4.1 *alliance*

O *alliance* é um projeto baseado em ROS que contém nós que fazem o controle distribuído da alocação de tarefa em um sistema com múltiplos robôs segundo o modelo sugerido por Parker (1998), a arquitetura ALLIANCE. Esta arquitetura resolve problema do tipo *ST-SR-IA*, ou seja, cada robô do sistema só pode executar uma tarefa por vez, as tarefas requisitadas só podem ser executadas por um único robô e atribuição das tarefas ocorre instantaneamente, não sendo considerado o estado do sistema no futuro.

Esta abordagem é baseada em comportamento. Cada robô do sistema possui diversos comportamentos. A ativação de um dado comportamento faz com que o robô passe a executar uma tarefa específica. Assim, uma nova alocação acontece sempre que um robô muda de comportamento.

Cada configuração de comportamento nos robôs possui um mecanismo para o cálculo de motivação que cresce em função de várias variáveis. Quando o nível de motivação de um dado comportamento atinge seu limite, este é ativado. As duas variáveis principais são impaciência e aquiescência. A impaciência do robô aumenta o nível de motivação em função das atividades dos demais robôs do sistema. Porém, a aquiescência leva o nível de motivação para zero, quando o robô verifica que ele deve desistir da tentativa de executar a tarefa especificada pelo comportamento. O Apêndice E dá mais detalhes sobre o funcionamento da arquitetura ALLIANCE.

Este pacote possui dois nós: *high_level* e *low_level*. Cada robô do sistema deve rodar esses dois nós para o bom funcionamento da arquitetura. Não são necessários nós adicionais, no que diz respeito à alocação de tarefa. O nó *high_level*, possui um nível de abstração maior, pois ele controla a ativação do comportamento a partir da análise do estado do sistema. Este nó possui diversos parâmetros, os quais afetam diretamente no desempenho do sistema. Pois estes parâmetros ditam a dinâmica da motivação de comportamento de um dado robô do sistema. Enquanto isso, o nó *low_level*, possui uma abstração mais baixa, pois este interage diretamente com o nível de controle de execução de tarefa. Este nó desempenha o papel de cuidar da análise sensorial dos comportamentos do robô e, ainda, direcionar para o nível de controle da execução da tarefa qual tarefa deve ser executada.

O nó *low_level* demanda maior detalhamento, pois ele é altamente dependente de implementações realizadas pelos usuários do pacote *alliance*. Para que fosse possível uma aproximação genérica do ALLIANCE, isto é, que pudesse ser utilizada em qualquer aplicação que atende as premissas desta arquitetura, utilizou-se o pacote *pluginlib*¹. Este pacote contém uma biblioteca C++ para carregar e descarregar *plugins* de pacotes do ROS. *Plugins* são classes dinamicamente carregáveis que são carregadas de uma biblioteca externa

¹ <<http://wiki.ros.org/pluginlib>>

em tempo de execução. Desta forma, o usuário pode fazer a análise sensorial e desenvolver a camada de controle de execução de tarefa de forma a atender a sua aplicação. Para isso foram criadas as seguintes classes base: *alliance::Sensor*, *alliance::SensoryEvaluator* e *alliance::Layer*. Será explicado a seguir sobre o desenvolvimento de *plugins* de sensor, avaliação e camada, mostrando o cabeçalho em C++ da classe base de cada tipo de *plugin*.

4.1.1 *Plugin* de sensor

A classe *alliance::Sensor* permite que o usuário do *alliance* desenvolva *plugins* de sensores reais ou virtuais. O usuário tem a flexibilidade de se inscrever em qualquer tópico do ROS para fazer leitura dos sensores cujos sinais são publicados por outros nós. Ou ainda pode ser utilizadas técnicas de fusão sensorial, filtragem e conversões para melhorar a análise sensorial. Mas, também, é possível criar sensores virtuais que utilizam temporizadores ou avaliam o estado abstrato das entidades do sistema. Os *plugins* dos sensores são estipulados na inicialização do nó *low_level* por meio de parâmetros do ROS. Se o *plugin* desenvolvido pelo usuário estiver devidamente cadastrado (conforme descrito na página do pacote *pluginlib*), o nó *low_level* não haverá problemas para carregá-lo. Por fim, é importante salientar que o usuário pode especificar diversos *plugins* de sensor.

```

1  #ifndef _ALLIANCE_SENSOR_H_
2  #define _ALLIANCE_SENSOR_H_
3
4  #include <ros/time.h>
5
6  namespace alliance
7  {
8  class Sensor
9  {
10 public:
11     Sensor();
12     virtual ~Sensor();
13     virtual void initialize(const std::string& ns, const std::
14                           string& name,
15                           const std::string& id);
16     virtual void readParameters();
17     std::string getNamespace() const;
18     std::string getName() const;
19     std::string getId() const;
20     virtual bool isUpToDate() const;
21     virtual bool operator==(const Sensor& sensor) const;

```

```

21     virtual bool operator!=(const Sensor& sensor) const;
22
23 protected:
24     std::string ns_;
25     std::string name_;
26     std::string id_;
27 };
28
29 typedef boost::shared_ptr<Sensor> SensorPtr;
30 typedef boost::shared_ptr<Sensor const> SensorConstPtr;
31 }
32
33 #endif // _ALLIANCE_SENSOR_H_

```

O *plugin* deve sobre-escrever o método *initialize* para que ele possa ser inicializado corretamente. Será disponibilizado à ele o *namespace* do nó *low_level*, o nome e o *id* do sensor. Porém, o *plugin* pode ser configurado através da leitura de parâmetros do ROS. Para uma melhor organização, é aconselhável sobre-escrever o método *readParameters* para fazer isso. Pois, a classe *alliance::Sensor* já chama este método logo após a inicialização do *plugin*. Enfim, o *plugin* pode sobre-escrever o método *isUpToDate* para informar se os dados providos do sensor real estão sendo recebidos.

Foi criada uma classe utilitária genérica, denominada *nodes::ROSSensorMessage*, para simplificar o trabalho do usuário do *alliance*. Esta classe herda os métodos da classe *alliance::Sensor* e, assim, também pode ser usada como classe base para criação de *plugins* de sensor. Diferentemente da classe *alliance::Sensor*, a classe *nodes::ROSSensorMessage* se inscreve no tópico que transporta mensagens contendo o sinal do sensor desejado. Neste caso, o *id* do sensor é considerado como o nome deste tópico. Esta classe sobre-escreve o método *readParameters* para coletar parâmetros que são utilizados no método *isUpToDate*, o qual é sobre-escrito para identificar se a última mensagem foi recebida dentro do tempo máximo especificado. Assim, os *plugins* que se baseiam nesta classe terão a última mensagem recebida disponível para a sobre-escrita do método *isApplicable*.

4.1.2 Plugin de avaliação sensorial

A classe base *alliance::SensoryEvaluator* tem como responsabilidade analisar os sensores estipulados em uma dada configuração de comportamento. Através dessa análise este avaliador enviará uma mensagem ao nó *high_level* dizendo se ativação deste comportamento específico é aplicável ou não. Como esta análise sensorial varia de uma aplicação para outra, essa classe foi projetada para a criação de *plugins* de avaliação. Através do re-

curso de herança do C++, o método *isApplicable* deve ser implementado pelo *plugin*, onde deverá ser realizada a análise sensorial. Para isso, este objeto terá disponível a coleção de *plugins* de sensor devidamente carregados para a análise. O *plugin* é inicializado através da sobre-escrita do método *initialize*, onde são dados: (1) um objeto *ros::NodeHandlePtr* para a interação com o ROS, (2) o objeto robô que contém a instância de todos os sensores, (3) a tarefa sobre o qual é feita a análise e, também, (4) uma lista com os *ids* dos sensores que devem ser considerados durante a análise.

```

1  #ifndef _ALLIANCE_SENSOR_EVALUATOR_H_
2  #define _ALLIANCE_SENSOR_EVALUATOR_H_
3
4  #include "alliance/sensor.h"
5  #include "alliance/task.h"
6  #include <alliance_msgs/SensoryFeedback.h>
7  #include <list>
8  #include <ros/publisher.h>
9
10 namespace alliance
11 {
12     class BehavedRobot;
13     typedef boost::shared_ptr<BehavedRobot> BehavedRobotPtr;
14     typedef boost::shared_ptr<BehavedRobot const>
15         BehavedRobotConstPtr;
16
17     class SensoryEvaluator
18     {
19     public:
20         SensoryEvaluator();
21         virtual ~SensoryEvaluator();
22         virtual void initialize(const ros::NodeHandlePtr& nh,
23                                 const BehavedRobotPtr& robot, const
24                                     Task& task,
25                                     const std::list<std::string> &
26                                         sensors);
27
28         void process();
29         virtual bool isApplicable() = 0;
30         virtual SensorPtr getSensor(const std::string& sensor_id)
31             const;
32
33     protected:

```

```

29     typedef std::list<SensorPtr>::iterator iterator;
30     typedef std::list<SensorPtr>::const_iterator const_iterator
        ;
31     std::list<SensorPtr> sensors_;
32
33 private:
34     ros::NodeHandlePtr nh_;
35     ros::Publisher sensory_feedback_pub_;
36     alliance_msgs::SensoryFeedback sensory_feedback_msg_;
37     bool contains(const std::string &sensor_id) const;
38 };
39
40 typedef boost::shared_ptr<SensoryEvaluator>
        SensoryEvaluatorPtr;
41 typedef boost::shared_ptr<SensoryEvaluator const>
        SensoryEvaluatorConstPtr;
42 }
43
44 #endif // _ALLIANCE_SENSORY_EVALUATOR_H_

```

4.1.3 *Plugin* de camada

Os *plugins* criados a partir da classe base *alliance::Layer* têm como responsabilidade controlar a execução da tarefa pelo robô. O usuário deve criar um *plugin* de camada para cada tarefa. Entretanto, se houver robôs que executam uma mesma tarefa de modos diferentes, cada modo terá seu próprio *plugin*. Como o nome da classe base sugere, o usuário tem a flexibilidade de implementar esses *plugins* em camadas, conforme sugerido por (PARKER, 1998).

```

1  #ifndef _ALLIANCE_LAYER_H_
2  #define _ALLIANCE_LAYER_H_
3
4  #include <boost/shared_ptr.hpp>
5  #include <string>
6  #include "alliance/sensory_evaluator.h"
7
8  namespace alliance
9  {
10     class Layer
11     {

```

```

12 public:
13     Layer();
14     virtual ~Layer();
15     virtual void initialize(const std::string& ns, const std::
        string& name);
16     virtual void setEvaluator(const SensoryEvaluatorPtr&
        evaluator);
17     virtual void readParameters();
18     virtual void process() = 0;
19     std::string getName() const;
20     bool operator==(const Layer& layer) const;
21     bool operator!=(const Layer& layer) const;
22
23 private:
24     std::string name_;
25     SensoryEvaluatorPtr evaluator_;
26 };
27
28 typedef boost::shared_ptr<Layer> LayerPtr;
29 typedef boost::shared_ptr<Layer const> LayerConstPtr;
30 }
31
32 #endif // _ALLIANCE_LAYER_H_

```

O *plugin* de camada é inicializado através da chamada do método *initialize*. Logo, este método deve ser sobre-escrito na classe do *plugin*, onde será disponibilizado o *namespace* do seu nó, bem como, o nome da camada. Porém, o *plugin* pode ser configurado a partir da leitura de parâmetros do ROS. Para uma melhor organização, é recomendado que esta leitura seja realizada dentro da sobre-escrita do método *readParameters*, pois a classe *alliance::Layer* faz sua chamada logo após a inicialização do *plugin*. Finalmente, o controle da execução da tarefa é realizado periodicamente através da sobre-escrita do método *process*. Este método deve ser utilizado para atualizar os sinais de comando dos atuadores do robô.

Verifica-se que o desenvolvimento desta aproximação do ALLIANCE é genérica e possui uma API simplificada. Deste modo, o usuário pode focar no desempenho dos robôs na execução das tarefas.

4.1.4 *alliance_msgs*

O pacote *alliance_msgs* foi criado em conjunto com o pacote *alliance* para separar as definições dos tipos de mensagens utilizadas por ele. Essas mensagens são utilizadas na comunicação entre os nós do pacote *alliance*.

Este pacote define as seguintes mensagens:

- *alliance_msgs/InterRobotCommunication*: armazena informação sobre a atividade de um robô específico em um dado instante. Possui o cabeçalho padrão do ROS que identifica o robô que enviou a mensagem e instante que ela foi enviada. Além disso, ela possui um campo que identifica a tarefa que o robô está executando;
- *alliance_msgs/Motivation*: utilizada para o monitoramento da arquitetura. Esta mensagem possui o cabeçalho padrão do ROS para identificar o robô que a enviou e também o instante em que ela foi enviada. Além disso, ela possui um campo que identifica a tarefa que o cálculo de motivação se referencia e, ainda, o valor das variáveis que influenciam no cálculo da motivação;
- *alliance_msgs/SensoryFeedback*: utilizada na comunicação entre os nós de baixo e alto nível de abstração de um mesmo robô. Esta mensagem informa se um dado comportamento é aplicável em um dado instante segundo uma análise sensorial realizada no nível de baixa abstração do *alliance*. Logo, esta mensagem possui o cabeçalho padrão do ROS que identifica o robô que a enviou e o instante em que ela foi enviada. Além disso, ela possui um campo que identifica a tarefa sobre a qual a análise é referenciada e, ainda, um campo informando se a ativação do comportamento que leva esse robô à execução dessa tarefa é aplicável.

4.1.5 *rqt_alliance*

O pacote *rqt_alliance* foi desenvolvido para auxiliar no monitoramento da arquitetura implementada no pacote *alliance*. Ele fornece uma ferramenta gráfica que detalha as variáveis que influenciam no cálculo de motivação de uma dada configurações de comportamento de um robô específico. Este *plugin* também fornece gráficos que mostram o nível de motivação para a ativação de cada comportamento de um dado robô.

A Figura 10 detalha graficamente o cálculo da motivação da configuração de comportamento que leva o robô */robot2* executar a tarefa *wander*. De cima para baixo estão os seguintes gráficos: (1) nível de motivação, (2) taxa de impaciência, (3) aquiescente, (4) suprimido, (5) reiniciada, (6) aplicável e (7) ativa. Perceba que o comportamento se mantém ativo (gráfico mais abaixo) enquanto o nível de motivação (linha contínua azul do gráfico mais acima) é igual ou superior ao *threshold* (linha tracejada vermelha do gráfico mais acima). Note que no instante em que o robô se tornou aquiescente (impulso visto

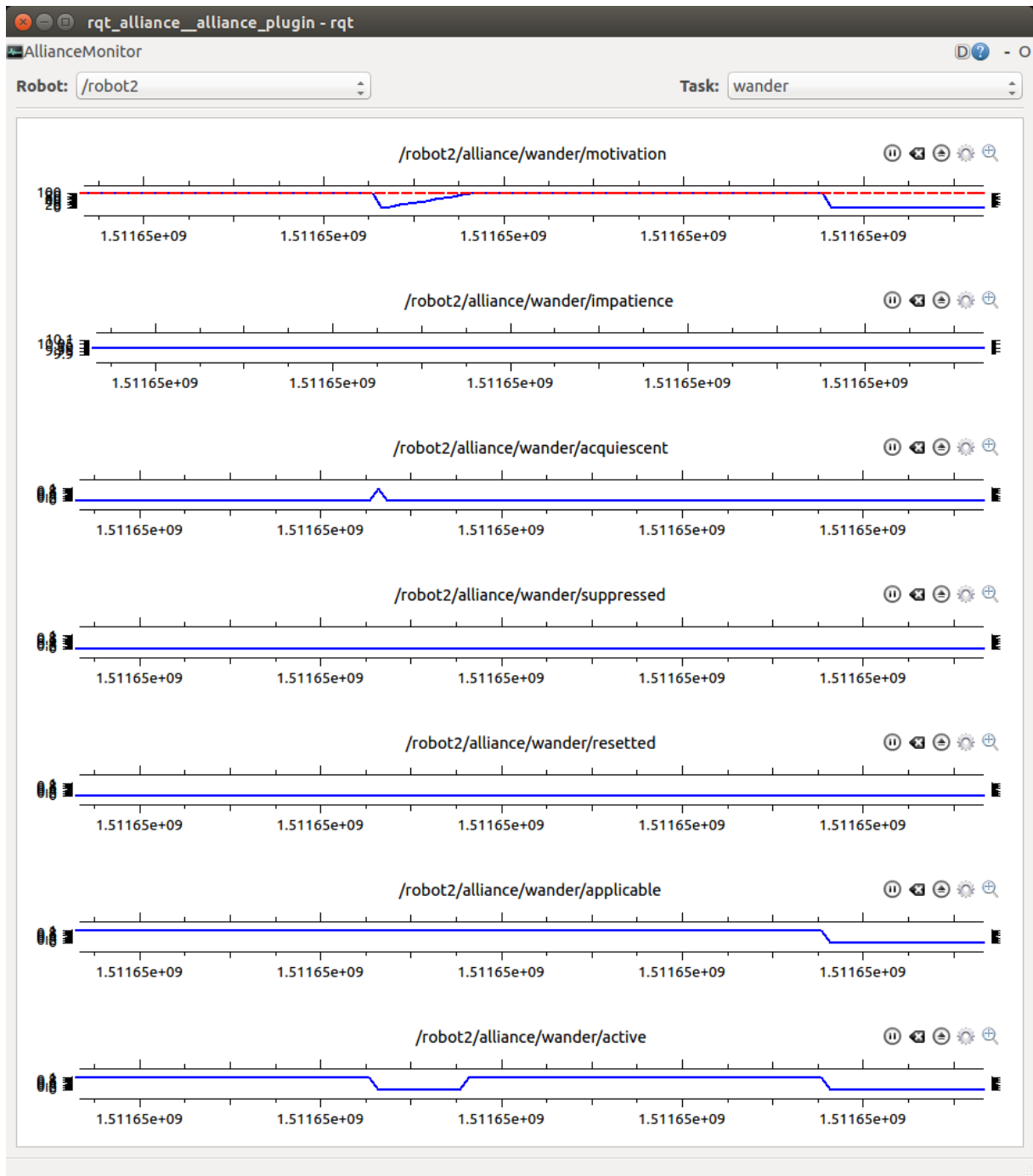


Figura 10 – Detalhamento da motivação `/robot2/alliance/wander` ao longo do tempo.

no terceiro gráfico de cima para baixo), o nível da sua motivação para ativar o comportamento *wander* é zerado. O nível de motivação é zerado também, ao final, quando esse comportamento se torna inaplicável. As demais variáveis permanecem constante durante todo o intervalo.

A Figura 11, por exemplo, mostra que o robô `/robot3` possui três configurações de comportamento: (1) *wander*, (2) *border_protection* e (3) *report*. Nesta figura, é mostrado o nível de motivação do robô `/robot3` para cada um dos comportamentos ao longo

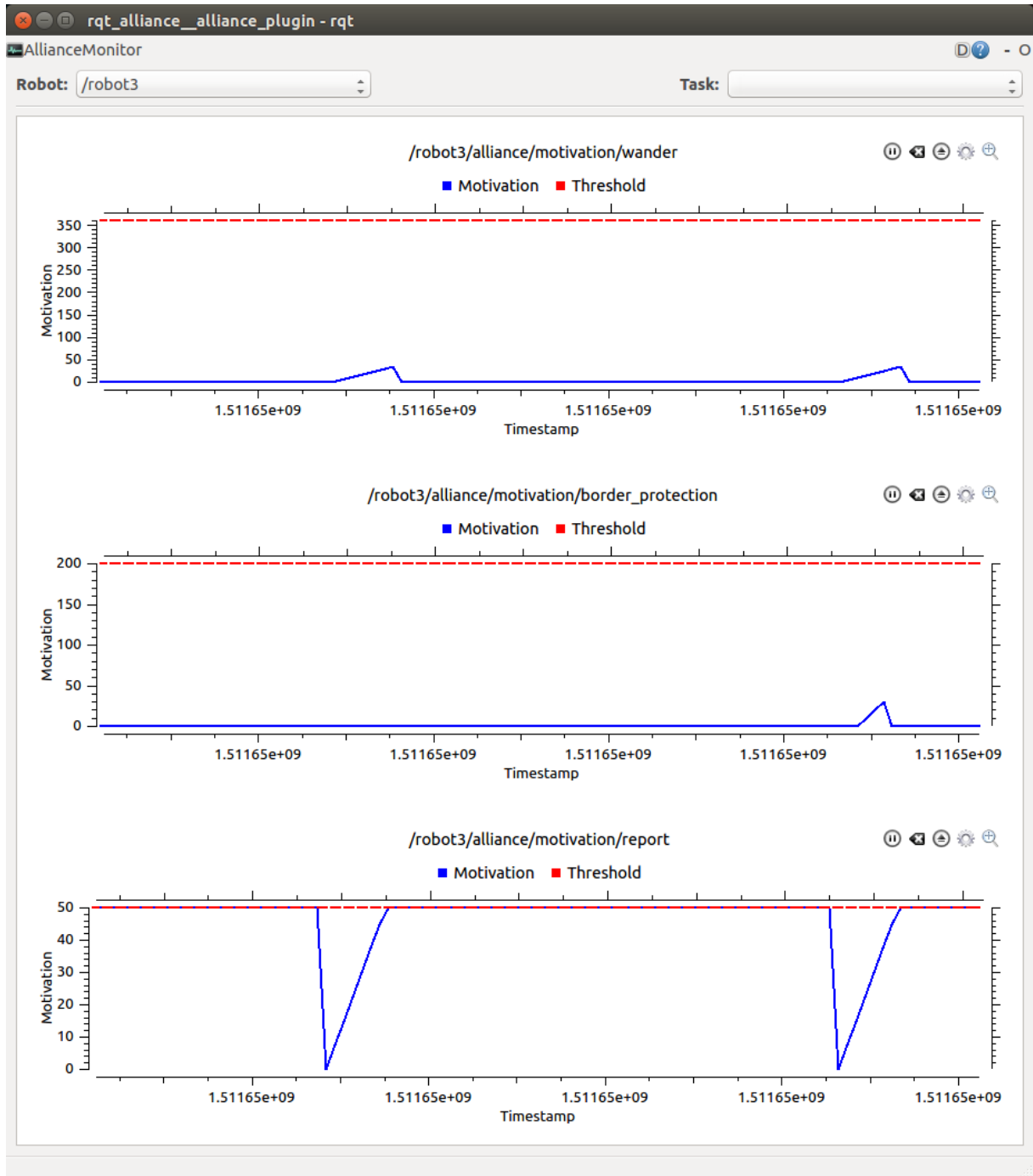


Figura 11 – Motivações das configurações de comportamento do robô */robot3*.

do tempo. Uma dada configuração de comportamento é ativada quando sua motivação (linha contínua azul) atinge o *threshold* (linha tracejada vermelha). Note que, enquanto o comportamento *report* do robô */robot3* não está ativo, há um aumento na motivação de todos os seus comportamentos. Porém, em ambos casos, a motivação de *report* atinge o *threshold* antes das outras. Isso se deve pelo fato dessa motivação de comportamento apresentar uma dinâmica mais rápida que as demais. Isso não significa que o robô */robot3* executará somente a tarefa *report*. Dependendo do estado do sistema, outro robô pode passar a realizá-la em seu lugar.

4.1.6 Arquivos de parâmetro do *alliance*

Os nós *high_level* e *low_level* do pacote *alliance* são configurados a partir da leitura de parâmetros do ROS durante a inicialização.

Existem três tipos de arquivos de parâmetro no *alliance*: de camada, de tarefa e de robô. O arquivo de parâmetro de camada traz configurações pertinentes para a execução das tarefas. Por este fato, este arquivo não é definido pelo pacote *alliance*. O arquivo de parâmetros de tarefa especifica quais são as tarefas existentes no sistema. Um exemplo deste arquivo é exibido a seguir.

```
1 tasks:
2   size: 3
3   task0:
4     id: wander
5     name: Wander
6     layers:
7       size: 1
8       layer0:
9         plugin_name: alliance_test/wander
10  task1:
11    id: border_protection
12    name: Border Protection
13    layers:
14      size: 1
15      layer0:
16        plugin_name: alliance_test/border_protection
17  task2:
18    id: report
19    name: Report
20    layers:
21      size: 1
22      layer0:
23        plugin_name: alliance_test/report
```

Cada robô deve informar quais sensores e configurações de comportamento ele possui. Cada configuração é parametrizada de acordo com a dinâmica de ativação desejada.

```
1 name: Robot 1
2 spin_rate: 2.0           # in Hertz
3 broadcast_rate: 0.5      # in Hertz
4 timeout_duration: 10.0   # in seconds
```

```

5  buffer_horizon: 10.0          # in seconds
6  sensors:
7    size: 1
8    sensor0:
9      plugin_name: alliance_test/point_cloud
10     topic_name: sonar
11     timeout_duration: 5.0
12     buffer_horizon: 5.0
13 behaviour_sets:
14   size: 1
15   behaviour_set0:
16     task_id: wander
17     task_expected_duration: 100.0 # in seconds
18     motivational_behaviour:
19       threshold: 150.0
20       acquiescence:
21         yielding_delay: 75.0      # in seconds
22         giving_up_delay: 110.0    # in seconds
23       impatience:
24         fast_rate: 5.0
25       sensory_feedback:
26         plugin_name: alliance_test/updated_sensory
27         sensors:
28           size: 1
29           sensor0:
30             topic_name: sonar

```

4.1.7 Arquivo de inicialização do *alliance*

O pacote *alliance* possui apenas um arquivo de inicialização. Este arquivo carrega os arquivos de parâmetros de tarefa e de robô para o nó *high_level* e os de tarefa, de robô e de camada para o nó *low_level*. Este arquivo deve ser utilizado para cada robô do sistema, cada um especificando os seus arquivos de parâmetros. Ele é mostrado a seguir.

```

1  <?xml version="1.0"?>
2  <launch>
3    <arg name="robot_id" />
4    <arg name="robot_params" />
5    <arg name="tasks_params" />
6    <arg name="layers_params" />

```



```

7   <group ns="$(arg_robot_id)">
8     <group ns="alliance">
9       <node name="high_level" pkg="alliance" type="high_level
10         " output="screen">
11         <param name="id" type="string" value="$(arg_robot_id)
12           " />
13         <rosparam file="$(arg_robot_params)" command="load" /
14           >
15         <rosparam file="$(arg_tasks_params)" command="load" /
16           >
17       </node>
18       <node name="low_level" pkg="alliance" type="low_level"
19         output="screen">
20         <param name="id" type="string" value="$(arg_robot_id)
21           " />
22         <rosparam file="$(arg_robot_params)" command="load" /
23           >
24         <rosparam file="$(arg_tasks_params)" command="load" /
25           >
26         <rosparam file="$(arg_layers_params)" command="load"
27           />
28       </node>
29     </group>
30 </group>
31 </launch>

```

4.1.8 Cadastro da arquitetura do pacote *alliance*

Primeiramente, é preciso criar o arquivo de configuração de arquitetura do pacote *alliance*. Para o preenchimento completo da *tag architecture*, as seguintes informações são pertinentes:

- a arquitetura ALLIANCE resolve problemas MRTA em que os robôs, as tarefas e as alocações são do tipo *ST*, *SR* e *IA*, respectivamente;
- esta aproximação foi desenvolvida de modo que os robôs publicam suas atividades no tópico */alliance/inter_robot_communication*. O tipo de mensagem transportada por este tópico é *alliance_msgs/InterRobotCommunication*. Seus campos *header/frame_id* e *task_id* identificam, respectivamente, o robô que enviou a mensagem e a tarefa que ele estava executando no instante em que ela foi enviada;

- e, além disso, devem ser gerados para cada robô do sistema um arquivo de parâmetros e um arquivo de inicialização.

Com isso, tem-se:

```

1 <architecture>
2   <name>ALLIANCE</name>
3   <robots>
4     <type>ST</type>
5     <busy_robots>
6       <topic>
7         <name>/alliance/inter_robot_communication</name>
8         <type>alliance_msgs/InterRobotCommunication</type>
9         <field>header/frame_id</field>
10        <queue_size>10</queue_size>
11        <timeout>2.0</timeout>
12      </topic>
13    </busy_robots>
14    <config_id>alliance_params</config_id>
15    <launch_id>alliance</launch_id>
16  </robots>
17  <tasks>
18    <type>SR</type>
19  </tasks>
20  <allocations>
21    <type>IA</type>
22    <allocated_tasks>
23      <topic>
24        <name>/alliance/inter_robot_communication</name>
25        <type>alliance_msgs/InterRobotCommunication</type>
26        <field>task_id</field>
27        <queue_size>10</queue_size>
28        <timeout>0.5</timeout>
29      </topic>
30    </allocated_tasks>
31  </allocations>
32 </architecture>

```

Para que os arquivos de parâmetros sejam gerados corretamente, é importante representar os padrões existentes em cada um deles através das *tags param*, *params* e

array. A cada avanço de *namespace* foi utilizada a *tag params* para agrupar um conjunto de parâmetros e a *tag array* foi utilizada sempre que identificado um padrão de repetição. Nisso, foram criados os seguintes *templates* de arquivos de parâmetros:

- *alliance_params*: *template* de robô, a partir deste modelo serão gerados arquivos de parâmetros para cada robô do sistema;
- *tasks_params*: *template* de tarefas, a partir deste modelo será gerado apenas um arquivo de parâmetro referente aos dados das tarefas existentes no sistema;
- *layers*: será solicitado ao usuário a inserção deste arquivo de parâmetros, já que ele não possui um modelo.

Para representar o *template* do arquivo de inicialização do *alliance*, foram associados todos os arquivos de parâmetros com o nome do argumento apropriado. De modo que, se obteve a seguinte representação:

```

1 <launches>
2   <launch_0>
3     <id>alliance</id>
4     <includes>
5       <include_0>
6         <file>$(find alliance)/launch/alliance.launch</file>
7         <args>
8           <arg_0>
9             <name>robot_id</name>
10            <value>@robot_id@</value>
11          </arg_0>
12          <arg_1>
13            <name>robot_params</name>
14            <value>$(find @package@)/config/
              @robot_id@_alliance_params.yaml</value>
15          </arg_1>
16          <arg_2>
17            <name>tasks_params</name>
18            <value>$(find @package@)/config/
              tasks_alliance_params.yaml</value>
19          </arg_2>
20          <arg_3>
21            <name>layers_params</name>
22            <value>$(find @package@)/config/
              layers_alliance_params.yaml</value>

```

```

23         </arg_3>
24     </args>
25 </include_0>
26 </includes>
27 </launch_0>
28 </launches>

```

O recurso de substituição da identificação do robô (pela palavra-chave *@robot_id@*) e o nome do pacote da aplicação (pela palavra-chave *@package@*) fornecido pelo *rqt_mrta*, aumenta as possibilidades de generalidade na construção dos arquivos de inicialização.

Finalmente, é identificado o nome do *plugin rqt_alliance* fornecido pela arquitetura para o monitoramento do ALLIANCE em tempo de execução. A seguinte representação foi utilizada:

```

1 <widgets>
2   <widget_0>
3     <plugin_name>rqt_alliance/alliance_plugin</plugin_name>
4   </widget_0>
5 </widgets>

```

Assim, foi obtido o arquivo de configuração da arquitetura MRTA do pacote *alliance*. O arquivo completo criado pode ser visto no Apêndice A.

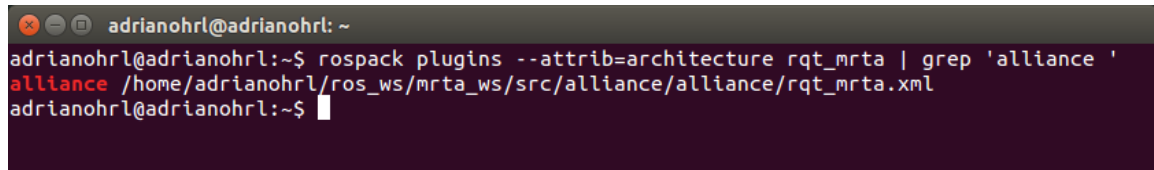
Para finalizar o cadastro da arquitetura do pacote *alliance* é necessário realizar algumas inserções no seu arquivo *package.xml*, as quais são:

- adicionar a dependência em tempo de execução do pacote *rqt_mrta*;
- e adicionar uma *tag* de exportação para o pacote *rqt_mrta* com o atributo *architecture*, informando a localização do arquivo de configuração da arquitetura.

O arquivo *package.xml* do pacote *alliance* é mostrado no Apêndice C.

A fim de verificar se o cadastro foi realizado com sucesso, foi utilizada a ferramenta *rospack plugin*. Um comando de linha que irá procurar dentre os pacotes do ROS aqueles que possuem dependência em tempo de execução do pacote *rqt_mrta* e que, ainda, exportam para ele o atributo *architecture*. Esta mesma ferramenta é utilizada pelo *rqt_mrta* para procurar as aplicações e arquiteturas no sistema de arquivos do ROS. O resultado deste comando pode ser visualizado na Figura 12.

Portanto, verifica-se que o pacote *alliance* está devidamente cadastrado como uma arquitetura disponível para o uso em aplicações através do *plugin rqt_mrta*.



```

adrianoahl@adrianoahl: ~
adrianoahl@adrianoahl:~$ rospack plugins --attrib=architecture rqt_mrta | grep 'alliance '
alliance /home/adrianoahl/ros_ws/mrta_ws/src/alliance/alliance/rqt_mrta.xml
adrianoahl@adrianoahl:~$

```

Figura 12 – Ferramenta *rospack* encontra a arquitetura *alliance*.

4.2 patrulha

Esta aplicação consiste no patrulhamento realizado por três robôs homogêneos em um ambiente fechado. Foram definidos três comportamentos para cada robô: *wander*, *border_protection* e *report*. O comportamento *wander* é uma implementação do algoritmo evitador de colisões. O robô permanece avançando para frente até que ele se depara com um anteparo. Sua direção aponta gradativamente para uma direção paralela ao obstáculo com o intuito de evitar uma colisão entre o robô e a parede. O comportamento *border_protection* implementa um algoritmo seguidor de parede. Através de um controlador P, são comparadas as distâncias aferidas pelos sensores ultrassônicos laterais do robô para mantê-lo dentro de um certo perímetro distante da parede. Enfim, o comportamento *report* é desempenhado pelos robôs de modo a informar ao solicitante da missão sobre o seu progresso.

Com o auxílio do simulador Stage MobileSim² da Adept MobileRobots foram utilizados três robôs diferenciais móveis Pioneer 3 DX (MOBILEROBOTS, 2011), mostrado na Figura 13, para simular a dinâmica dos robôs em um ambiente fechado para a simulação de uma aplicação de patrulhamento. Este robô está disponível para uso no Laboratório de Robótica (LRO) da Universidade Federal de Itajubá (UNIFEI). Dado que o foco deste trabalho está na avaliação do pacote *rqt_mrta*, a aplicação foi simplificada ao máximo, de modo que foram apenas utilizados: (1) o sensor *encoder* para ter uma estimativa do deslocamento do robô, (2) os sensores ultrassônicos para uma estimativa da distância entre o robô e as paredes do ambiente simulado, assim como, (3) seus motores para deslocar o robô pelo ambiente.

Também foi utilizado o pacote *RosAria*³ do ROS para ter acesso aos sensores e atuadores dos robôs no ambiente simulado com o sistema de comunicação do ROS através de uma rede de computadores.

Como é mostrado a seguir, o arquivo de descrição dos *plugin*⁴ desenvolvidos no pacote *alliance_test* para o pacote *alliance* foram criados para a interação entre a arqui-

² <<http://robots.mobilerobots.com/wiki/MobileSim>>

³ <<http://wiki.ros.org/ROSARIA>>

⁴ O arquivo de descrição de *plugin* é um arquivo XML que serve para armazenar toda informação importante sobre um *plugin* em um formato legível à máquina. Ele contém informação sobre a biblioteca onde o *plugin* está localizado, seu nome, seu tipo, etc. Para mais informações, vide <<http://wiki.ros.org/pluginlib>>.



Figura 13 – Robô Pioneer 3 DX da Adept MobileRobots ([MOBILEROBOTS, 2011](#)).

tetura ALLIANCE e a aplicação de patrulhamento.

```

1 <class_libraries>
2   <library path="lib/liballiance_test">
3     <class name="alliance_test/border_protection"
4       type="alliance_test::BorderProtection"
5       base_class_type="alliance::Layer">
6       <description>
7         Border Protection layer for ALLIANCE MRTA
8         architecture.
9       </description>
10    </class>
11    <class name="alliance_test/report"
12      type="alliance_test::Report"
13      base_class_type="alliance::Layer">
14      <description>
15        Report layer for ALLIANCE MRTA architecture.
16      </description>
17    </class>
18    <class name="alliance_test/wander"
19      type="alliance_test::Wander"
20      base_class_type="alliance::Layer">
21      <description>
22        Wander layer for ALLIANCE MRTA architecture.
23      </description>
24    </class>
25    <class name="alliance_test/odometry"

```

```

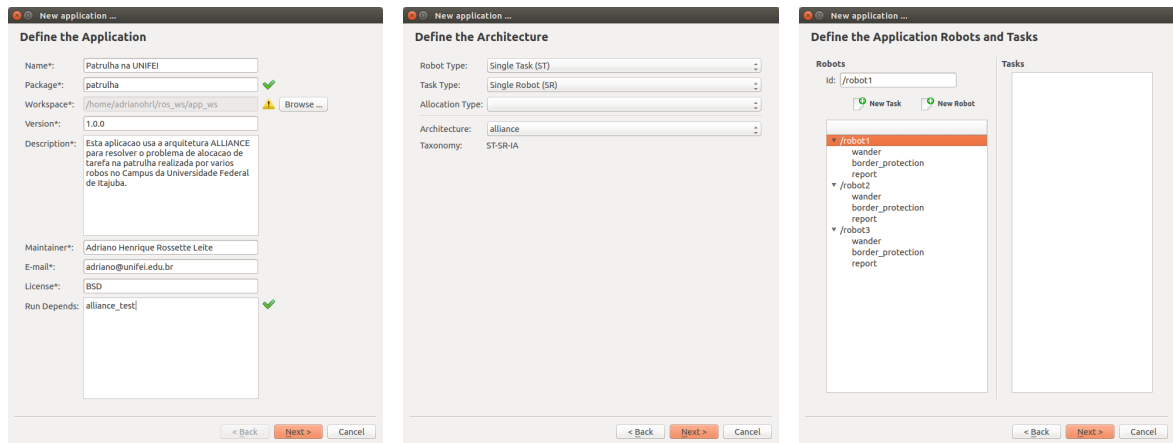
25         type="alliance_test::Odometry"
26         base_class_type="alliance::Sensor">
27     <description>
28         nav_msgs/Odometry message wrapper for sensor data
           receiving.
29     </description>
30 </class>
31 <class name="alliance_test/point_cloud"
32         type="alliance_test::PointCloud"
33         base_class_type="alliance::Sensor">
34     <description>
35         sensor_msgs/PointCloud message wrapper for sensor
           data receiving.
36     </description>
37 </class>
38 <class name="alliance_test/updated_sensory"
39         type="alliance_test::UpdatedSensory"
40         base_class_type="alliance::SensoryEvaluator">
41     <description>
42         This evaluator verifies if all sensors are up-to-date
           .
43     </description>
44 </class>
45 </library>
46 </class_libraries>

```

4.2.1 Criação da aplicação através do *plugin rqt_mrta*

Para a criação da aplicação *patrulha*, foi utilizado o *wizard* de criação de aplicações do *plugin rqt_mrta* descrito em 3.4. A Figura 14 exibe as telas do *wizard* de criação de aplicação do *rqt_mrta* devidamente preenchidas.

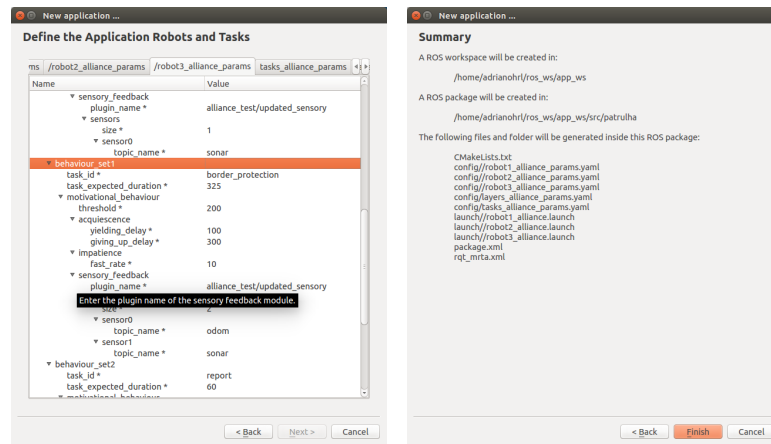
Primeiramente, foram preenchidos os dados referentes a criação do pacote da aplicação (vide Figura 14a). Nesta etapa, foram inseridos: (1) o nome da aplicação, (2) o nome do pacote da aplicação, (3) o diretório do *workspace* em que o pacote foi criado, (4) a versão do pacote, (5) a descrição da aplicação, (6) o nome do mantenedor do pacote da aplicação e (7) seu *e-mail*, (8) o tipo de licença do pacote e (9) a dependência em tempo de execução do pacote *alliance_test*, que implementa os *plugins* necessários para o controle da execução das tarefas da aplicação *patrulha*. Como o diretório do *workspace* fornecido não era um *workspace* do ROS, um alerta foi gerado, informando que este diretório seria



(a) Dados gerais da aplicação.

(b) Escolha da arquitetura.

(c) Definição dos robôs do sistema.



(d) Parametrização da arquitetura.

(e) Sumário.

Figura 14 – Criação da aplicação *patrulha*.

transformado em um *workspace* do ROS.

Em seguida, foi selecionada a arquitetura *alliance* (vide Figura 14b). Lembrando que, se o usuário não souber qual arquitetura ele irá usar, este pode utilizar os filtros de acordo com a classificação da taxonomia de (GERKEY; MATARIĆ, 2004) para o seu problema MRTA.

Na sequência, foi solicitado o preenchimento dos robôs do sistema (vide Figura 14c). Os três robôs foram inseridos: (1) */robot1*, (2) */robot2* e (3) */robot3*. Como este é um sistema homogêneo, todos os robôs podem executar as três tarefas do problema: (1) *wander*, (2) *border_protection* e (3) *report*.

Foram preenchidas, na próxima tela (vide Figura 14d), os parâmetros para cinco arquivos de parâmetros: um para cada robô, uma para as tarefas e um para as camadas. Assim que todos parâmetros obrigatórios foram preenchidos, o botão *Next* se tornou habilitado.

Na última tela do *wizard* (vide Figura 14e), foi exibida uma mensagem resumindo

o que aconteceria ao pressionar o botão *Finish*. Foi dito que:

- seria iniciado um *workspace* do ROS no diretório dado;
- seria criado um pacote ROS no diretório *src* do *workspace*;
- seriam gerados os arquivos *CMakeLists.txt*, *package.xml* e *rqt_mrta.xml* na raiz do pacote;
- seriam gerados os cinco arquivos de parâmetros (de extensão *.yaml*) dentro da pasta *config* do pacote;
- e seriam gerados os três arquivos de inicialização (de extensão *.launch*) dentro da pasta *launch* do pacote.

4.2.2 Análise da aplicação gerada

Ao pressionar o botão *Finish* do *wizard* de criação de aplicação, todas as pastas e arquivos citados acima foram criados e gerados, respectivamente. A Figura 15 mostra a estrutura do pacote *patrulha* criado.

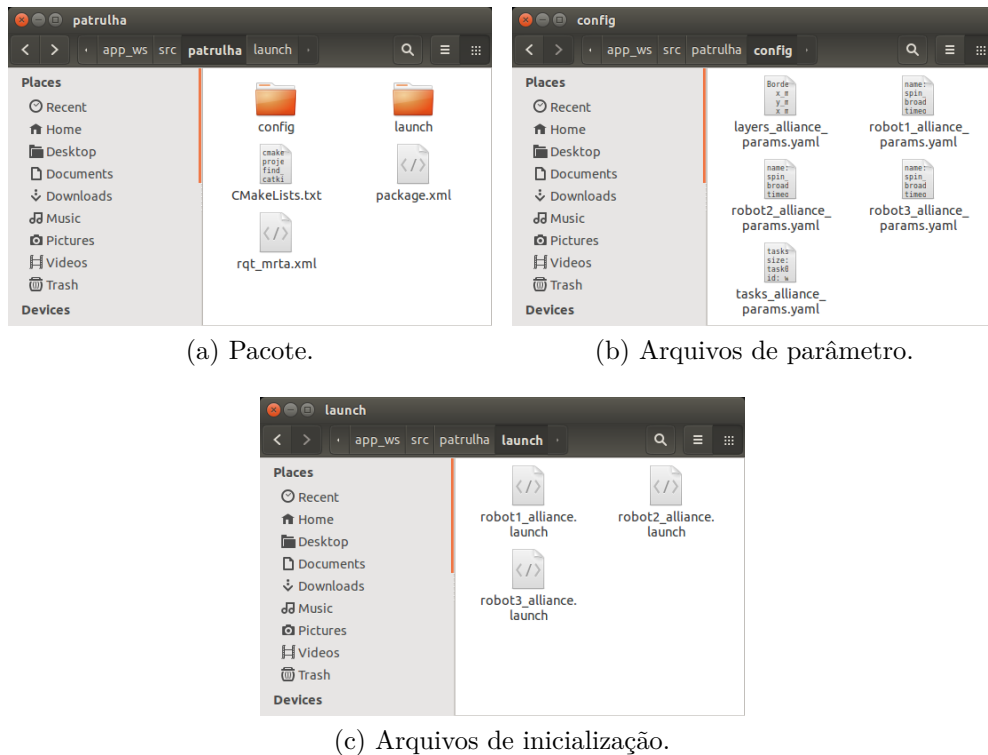
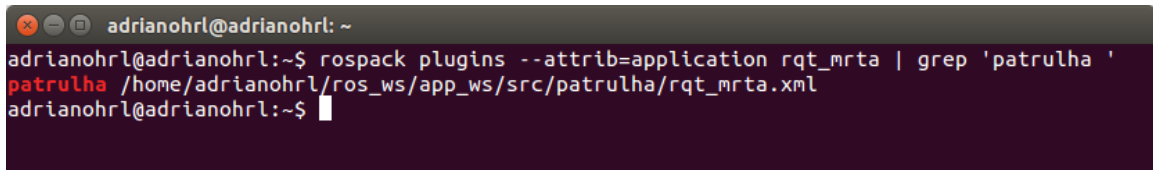


Figura 15 – Pastas e arquivos gerados após a criação de uma aplicação.

Observa-se que o pacote está devidamente organizado, agrupando os arquivos de parâmetros na pasta *config* e os arquivos de inicialização na pasta *launch*. Além disso, verificou-se que os arquivos gerados, possuem a formatação esperada. Os Apêndices B e

D mostram os arquivos de configuração de aplicação e de manifesto do pacote *patrulha* gerados.

Para verificar se o pacote *patrulha* foi configurado como uma aplicação do ponto de vista do *plugin rqt_mrta*, foi usada a ferramenta *rospack plugin*, só que desta vez, passando como atributo *application*. Como foi dito anteriormente, esta mesma ferramenta é utilizada pelo *rqt_mrta* para procurar as aplicações e arquiteturas no sistema de arquivos do ROS. O resultado deste comando pode ser visualizado na Figura 16.



```

adrianoahl@adrianoahl: ~
adrianoahl@adrianoahl:~$ rospack plugins --attrib=application rqt_mrta | grep 'patrulha '
patrulha /home/adrianoahl/ros_ws/app_ws/src/patrulha/rqt_mrta.xml
adrianoahl@adrianoahl:~$

```

Figura 16 – Ferramenta *rospack* encontra a aplicação *patrulha*.

Portanto, verifica-se que o pacote *patrulha* está devidamente cadastrado como uma aplicação no ponto de vista do *plugin rqt_mrta*.

4.2.3 Simulação a partir dos arquivos gerados

Cabe agora a validação dos arquivos gerados através da execução dos arquivos de inicialização gerados através da ferramenta *roslaunch*.

A Figura 17 exibe cinco terminais enumerados de 1 à 5. O Terminal 1 mostra as saídas do comando de linha *rostopic list* que mostra os nós em execução no ROS. Este comando foi executado antes e após a inicialização dos nós da arquitetura de cada robô. Logo, verifica-se que inicialmente apenas o nó mestre do ROS que está em execução. Após a inicialização dos nós do robô */robot1* (vide Terminal 3), os nós */robot1/alliance/low_level* e */robot1/alliance/high_level* passaram a executar juntamente com o nó mestre do ROS. No Terminal 4, ao inicializar os nós do robô */robot2*, os nós *low_level* e *high_level* com o *namespace /robot2/alliance* passaram a executar juntamente com os demais. Por fim, no Terminal 5, ao executar o comando *roslaunch* para o arquivo de inicialização do robô */robot3*, os nós *low_level* e *high_level* com o *namespace /robot3/alliance* passaram a executar juntamente com os demais. O Terminal 2 lista os tópicos existentes no ROS após a inicialização de todos os nós citados acima.

Outra forma de visualizar a execução correta dos nós da arquitetura do pacote *alliance* para a aplicação *patrulha* pode ser vista com mais detalhes na Figura 19. Esta figura foi extraída da ferramenta *rqt_graph*⁵ disponível no ROS, onde nós são representados por elipses e tópicos são representados por retângulos.

⁵ <http://wiki.ros.org/rqt_graph>

The figure displays three terminal windows side-by-side, each showing the output of ROS commands for three different robots (robot1, robot2, and robot3). The windows are labeled 1, 2, 3, 4, and 5 at the bottom.

Window 1 (Left): Shows the output of `rosnode list` for robot1. The output lists nodes for robot1/alliance/high_level, robot1/alliance/low_level, and robot1/alliance/low_level/tasks/ta. The ROS_MASTER_URI is http://localhost:11311.

Window 2 (Right): Shows the output of `rostopic list` for robot1. The output lists topics for robot1/alliance/inter_robot_communication, robot1/alliance/motivation/border_protection, robot1/alliance/motivation/report, robot1/alliance/motivation/wander, robot1/alliance/sensory_feedback, robot1/cmd_vel, robot1/odom, robot1/sonar, robot2/alliance/motivation/border_protection, robot2/alliance/motivation/report, robot2/alliance/motivation/wander, robot2/alliance/sensory_feedback, robot2/cmd_vel, robot2/odom, robot2/sonar, robot3/alliance/motivation/border_protection, robot3/alliance/motivation/report, robot3/alliance/motivation/wander, robot3/alliance/sensory_feedback, robot3/cmd_vel, robot3/odom, robot3/sonar, rosout, and rosout_agg.

Windows 3, 4, and 5 (Bottom): Show the output of `rostopic list` and `rosnode list` for robot1, robot2, and robot3, respectively. The output for each robot is identical to the output in Window 1 and Window 2, respectively.

Figura 17 – Inicialização dos nós da arquitetura para três robôs.

Entretanto, até o presente momento, apenas a arquitetura está sendo executada juntamente com o camada de controle de execução das tarefas. Torna-se, ainda, necessário adicionar os robôs ao sistema. Pois os sinais providos dos seus sensores são inexistentes. Além disso, não existem atuadores para serem comandados.

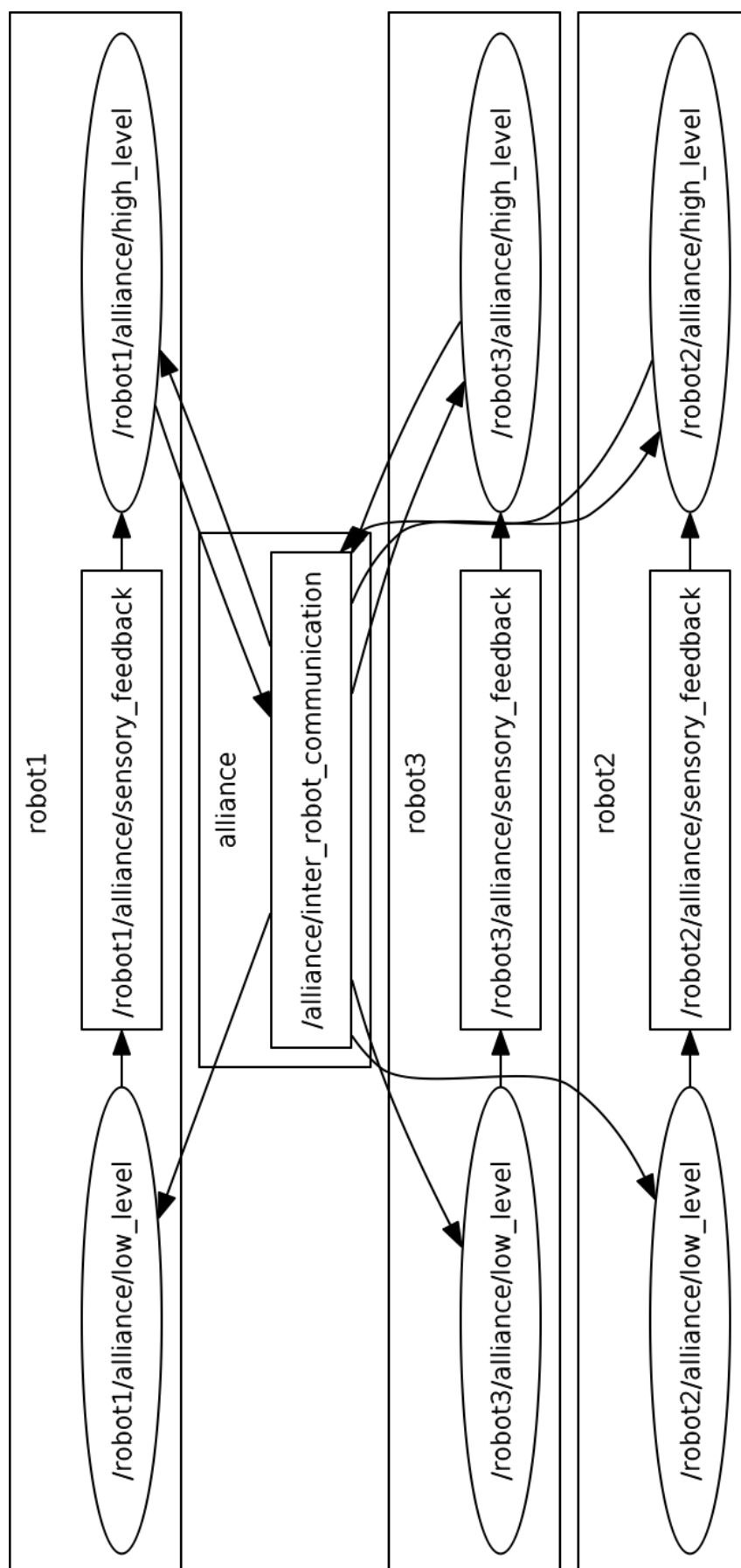
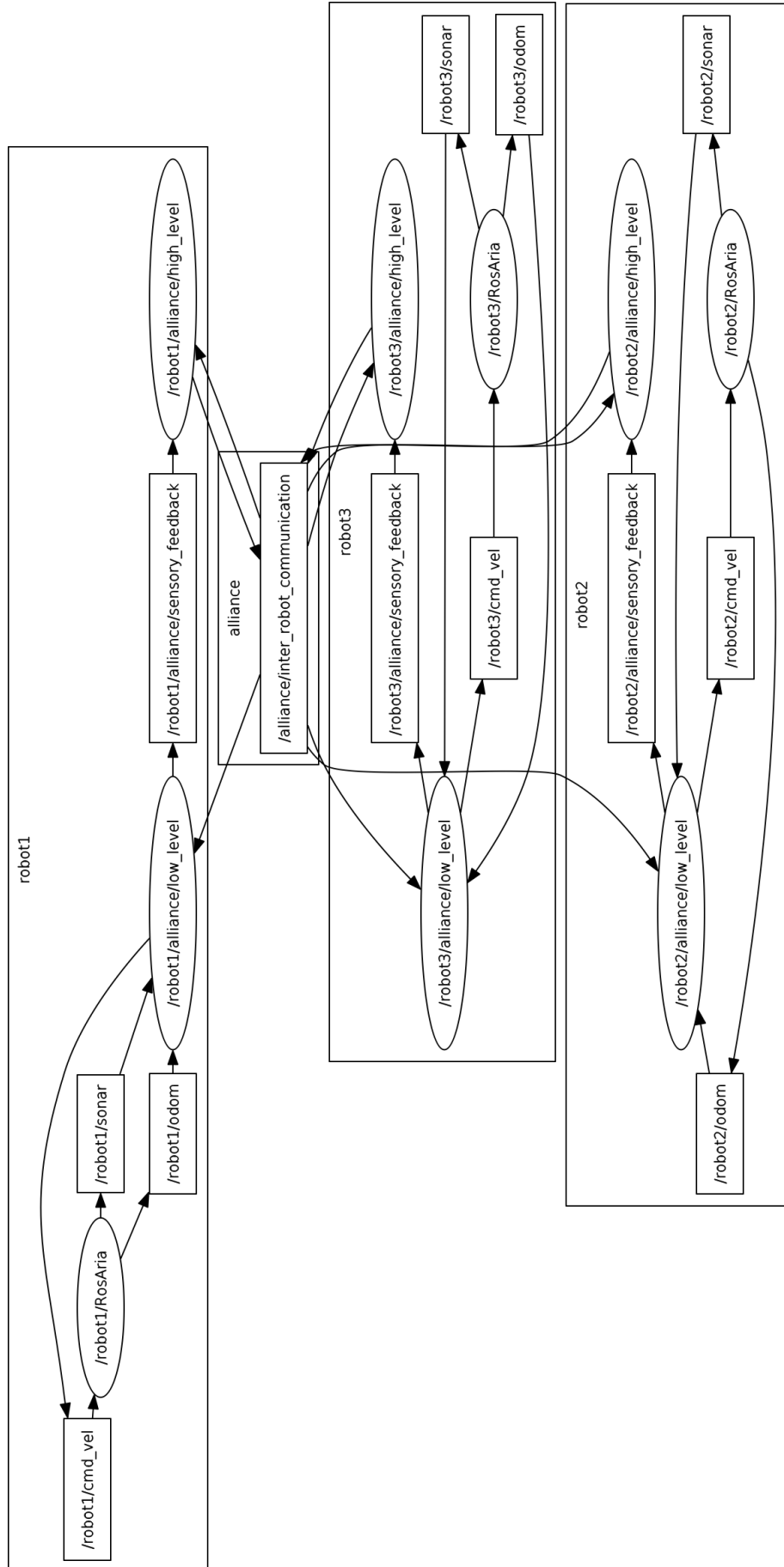


Figura 18 – Grafo da arquitetura *alliance* no ROS para três robôs.

Figura 19 – Grafo da aplicação *patrulha* no ROS.

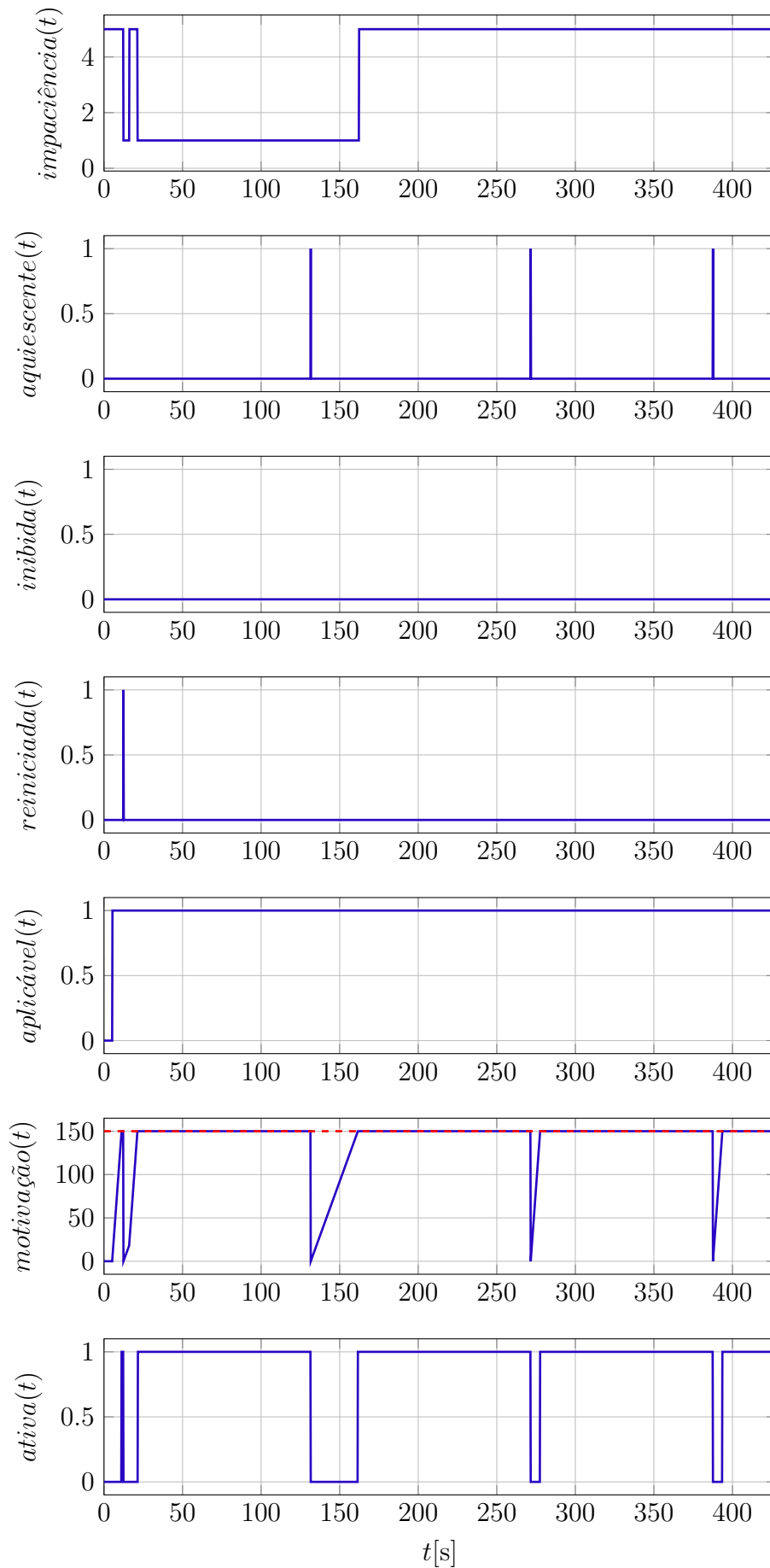


Figura 20 – Motivação da configuração de comportamento /robot1/wander.

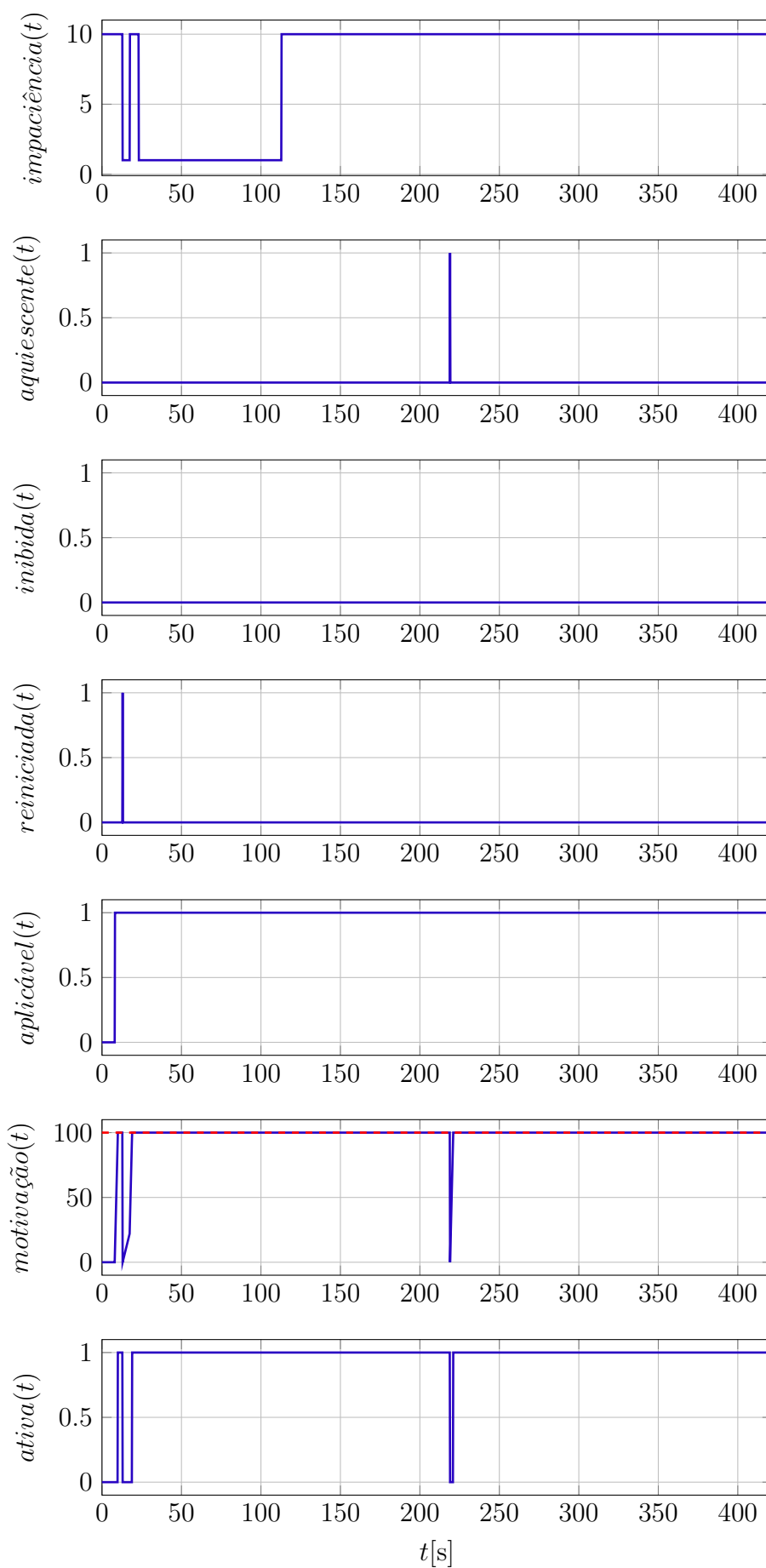


Figura 21 – Motivação da configuração de comportamento /robot2/wander.

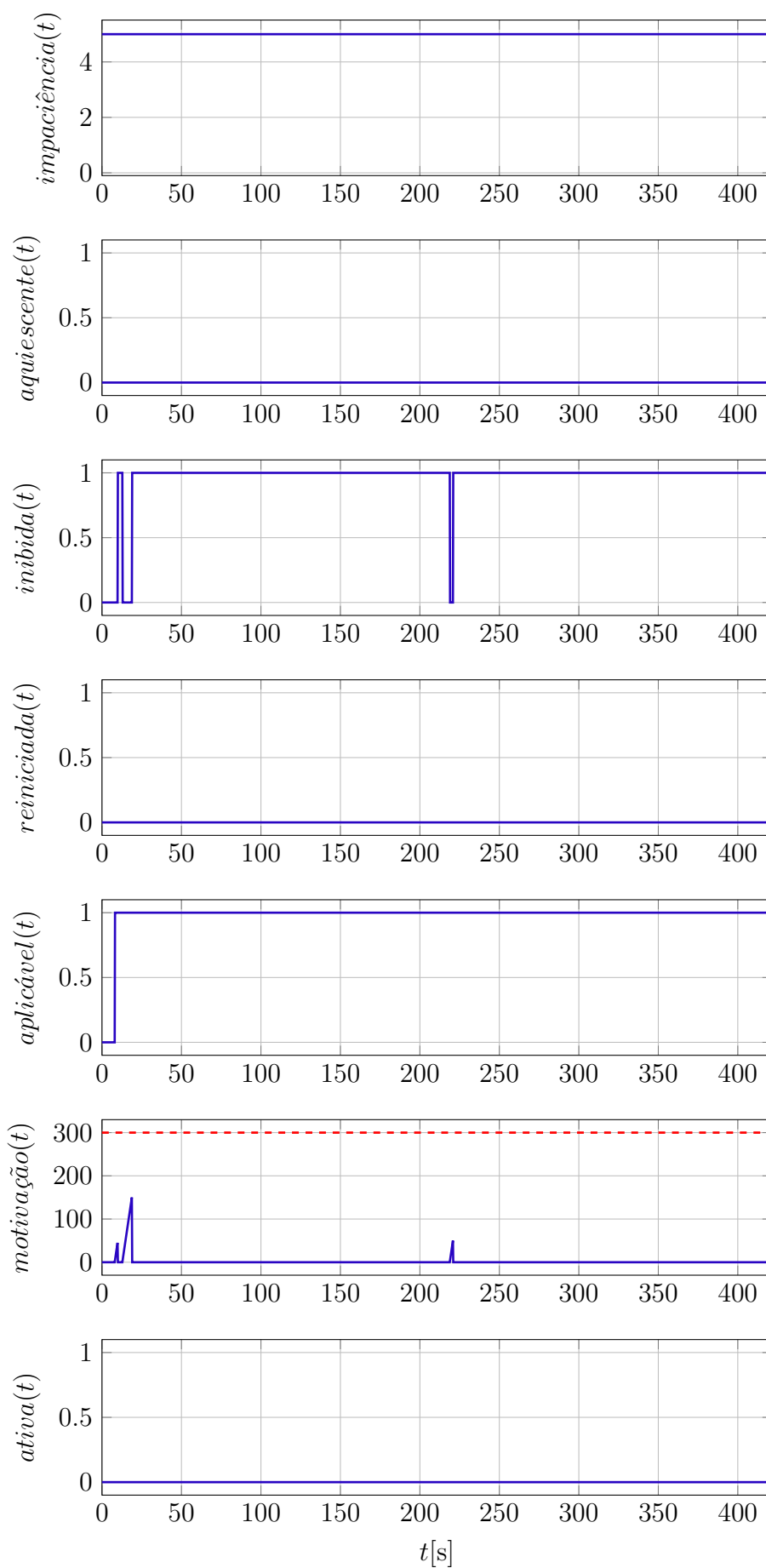


Figura 22 – Motivação da configuração de comportamento `/robot2/border_protection`.

5 Conclusão e Trabalhos Futuros

5.1 Conclusão

5.2 Trabalhos Futuros

Uma melhoria significativa desta aplicação seria possibilitar a inscrição de novas arquiteturas devidamente configuradas no repositório deste projeto. De modo que, sempre que uma nova inscrição ocorresse no *rqt_mrta*, seria solicitada uma nova atualização a todos usuários do pacote *rqt_mrta* para baixar e instalar a nova arquitetura inscrita. Isso eliminaria a necessidade de procurar novas abordagens na comunidade ROS.

Adição de mais recursos gráficos para a supervisão e monitoramento do sistema e da arquitetura em tempo de execução. Como alarmes identificando a falha na comunicação de algum robô, gráficos temporais de agenda ...

Referências

- BASTOS, G. S.; RIBEIRO, C. H. C.; SOUZA, L. E. de. Variable utility in multi-robot task allocation systems. In: IEEE. *Robotic Symposium, 2008. LARS'08. IEEE Latin American*. [S.l.], 2008. p. 179–183. [21](#)
- BOTELHO, S. C.; ALAMI, R. M+: a scheme for multi-robot cooperation through negotiated task allocation and achievement. In: IEEE. *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*. [S.l.], 1999. v. 2, p. 1234–1239. [15](#), [23](#)
- BROOKS, R. A robust layered control system for a mobile robot. *IEEE journal on robotics and automation*, IEEE, v. 2, n. 1, p. 14–23, 1986. [22](#), [97](#)
- CAO, Y. U.; FUKUNAGA, A. S.; KAHNG, A. Cooperative mobile robotics: Antecedents and directions. *Autonomous robots*, Kluwer Academic Publishers, v. 4, n. 1, p. 7–27, 1997. [15](#)
- CHAIMOWICZ, L.; CAMPOS, M. F.; KUMAR, V. Dynamic role assignment for cooperative robots. In: IEEE. *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*. [S.l.], 2002. v. 1, p. 293–298. [15](#)
- DIAS, M. B.; STENTZ, A. A free market architecture for distributed control of a multirobot system. In: *6th International Conference on Intelligent Autonomous Systems (IAS-6)*. [S.l.: s.n.], 2000. p. 115–122. [23](#)
- DUDEK, G. et al. A taxonomy for multi-agent robotics. *Autonomous Robots*, Springer, v. 3, n. 4, p. 375–397, 1996. [15](#)
- FOX, M.; LONG, D. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 2003. [15](#)
- FRANK, A. On kuhn's hungarian method—a tribute from hungary. *Naval Research Logistics (NRL)*, Wiley Online Library, v. 52, n. 1, p. 2–5, 2005. [15](#)
- GAMMA, E. et al. Design patterns: Abstraction and reuse of object-oriented design. In: SPRINGER. *European Conference on Object-Oriented Programming*. [S.l.], 1993. p. 406–431. [43](#)
- GERKEY, B. P.; MATARIC, M. J. Sold!: Auction methods for multirobot coordination. *IEEE transactions on robotics and automation*, IEEE, v. 18, n. 5, p. 758–768, 2002. [15](#), [23](#), [32](#)
- GERKEY, B. P.; MATARIĆ, M. J. A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research*, SAGE Publications, v. 23, n. 9, p. 939–954, 2004. [15](#), [21](#), [22](#), [34](#), [72](#)
- GUIDOTTI, C. F. et al. A murdoch-based package on ros for multi-robot task allocation. In: . [S.l.]: Universidade Federal de Itajubá, 2017. [32](#)

- JULIO, R. E.; BASTOS, G. S. Dynamic bandwidth management library for multi-robot systems. In: IEEE. *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*. [S.l.], 2015. p. 2585–2590. 15
- LI, A. W.; BASTOS, G. S. A hybrid self-adaptive particle filter through kld-sampling and samcl. In: IEEE. *Advanced Robotics (ICAR), 2017 18th International Conference on*. [S.l.], 2017. p. 106–111. 15
- LI, M. et al. Alliance-ros: A software architecture on ros for fault-tolerant cooperative multi-robot systems. In: SPRINGER. *Pacific Rim International Conference on Artificial Intelligence*. [S.l.], 2016. p. 233–242. 16, 32
- LIMA, P. U.; CUSTODIO, L. M. Multi-robot systems. In: *Innovations in robot mobility and control*. [S.l.]: Springer, 2005. p. 1–64. 18
- MANNE, A. S. On the job-shop scheduling problem. *Operations Research*, INFORMS, v. 8, n. 2, p. 219–223, 1960. 15
- MOBILEROBOTS, A. *Pioneer 3DX*. 2011. Disponível em: <<http://www.mobilerobots.com/ResearchRobots/PioneerP3DX.aspx>>. Acesso em: 13 outubro 2015. 69, 70
- PARKER, L. E. L-alliance: Task-oriented multi-robot learning in behavior-based systems. *Advanced Robotics*, Taylor & Francis, v. 11, n. 4, p. 305–322, 1996. 23, 101
- PARKER, L. E. Alliance: An architecture for fault tolerant multirobot cooperation. *IEEE transactions on robotics and automation*, IEEE, v. 14, n. 2, p. 220–240, 1998. 15, 16, 22, 32, 53, 54, 58, 97
- QUIGLEY, M. et al. Ros: an open-source robot operating system. In: KOBE. *ICRA workshop on open source software*. [S.l.], 2009. v. 3, p. 5. 15, 24
- REIS, W. P. N. dos; BASTOS, G. S. Multi-robot task allocation approach using ros. In: IEEE. *Robotics Symposium (LARS) and 2015 3rd Brazilian Symposium on Robotics (LARS-SBR), 2015 12th Latin American*. [S.l.], 2015. p. 163–168. 16, 32
- SCHNEIDER, D. G. et al. Robot navigation by gesture recognition with ros and kinect. In: IEEE. *Robotics Symposium (LARS) and 2015 3rd Brazilian Symposium on Robotics (LARS-SBR), 2015 12th Latin American*. [S.l.], 2015. p. 145–150. 15
- SMITH, R. G. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on computers*, IEEE, n. 12, p. 1104–1113, 1980. 23
- STENTZ, A.; DIAS, M. B. *A free market architecture for coordinating multiple robots*. [S.l.], 1999. 15
- WATKINS, C. J.; DAYAN, P. Q-learning. *Machine learning*, Springer, v. 8, n. 3-4, p. 279–292, 1992. 15
- WERGER, B. B.; MATARIĆ, M. J. Broadcast of local eligibility for multi-target observation. In: *Distributed autonomous robotic systems 4*. [S.l.]: Springer, 2000. p. 347–356. 15, 23
- YAFEI, H. *Qt creator quick start*. [S.l.]: Beijing: Beihang University Press, 2012. 44, 52

- YAN, Z.; JOUANDEAU, N.; CHERIF, A. A. Multi-robot decentralized exploration using a trade-based approach. In: *ICINCO (2)*. [S.l.: s.n.], 2011. p. 99–105. [23](#)
- YAN, Z.; JOUANDEAU, N.; CHERIF, A. A. A survey and analysis of multi-robot coordination. *International Journal of Advanced Robotic Systems*, SAGE Publications Sage UK: London, England, v. 10, n. 12, p. 399, 2013. [18](#), [23](#), [24](#)
- ZLOT, R. et al. Multi-robot exploration controlled by a market economy. In: IEEE. *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*. [S.l.], 2002. v. 3, p. 3016–3023. [15](#)
- ZLOT, R. M.; STENTZ, A. An auction-based approach to complex task allocation for multirobot teams. Carnegie Mellon University, The Robotics Institute, 2006. [20](#), [23](#)

Apêndices

APÊNDICE A – Exemplo de um arquivo de configuração de arquitetura para o *rqt_mrta*

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <rqt_mrta>
3   <architecture>
4     <name>ALLIANCE</name>
5     <robots>
6       <type>ST</type>
7       <busy_robots>
8         <topic>
9           <name>/alliance/inter_robot_communication</name>
10          <type>alliance_msgs/InterRobotCommunication</type>
11          <field>header/frame_id</field>
12          <queue_size>10</queue_size>
13          <timeout>2.0</timeout>
14        </topic>
15      </busy_robots>
16      <config_id>alliance_params</config_id>
17      <launch_id>alliance</launch_id>
18    </robots>
19    <tasks>
20      <type>SR</type>
21    </tasks>
22    <allocations>
23      <type>IA</type>
24      <allocated_tasks>
25        <topic>
26          <name>/alliance/inter_robot_communication</name>
27          <type>alliance_msgs/InterRobotCommunication</type>
28          <field>task_id</field>
29          <queue_size>10</queue_size>
30          <timeout>0.5</timeout>
31        </topic>
32      </allocated_tasks>
33    </allocations>

```

```
34 </architecture>
35 <configs>
36   <config_0>
37     <id>alliance_params</id>
38     <param_0>
39       <name>name</name>
40       <type>string</type>
41       <tool_tip>Enter the robot name.</tool_tip>
42     </param_0>
43     <param_1>
44       <name>spin_rate</name>
45       <type>double</type>
46       <default>2.0</default>
47       <tool_tip>Enter the spin rate of the node (in [Hz]).<
         /tool_tip>
48     </param_1>
49     <param_2>
50       <name>broadcast_rate</name>
51       <type>double</type>
52       <tool_tip>Enter the robot broadcast rate (in [Hz]).</
         tool_tip>
53     </param_2>
54     <param_3>
55       <name>timeout_duration</name>
56       <type>double</type>
57       <tool_tip>Enter the robot timeout duration (in [s]).<
         /tool_tip>
58     </param_3>
59     <param_4>
60       <name>buffer_horizon</name>
61       <type>double</type>
62       <tool_tip>Enter the robot buffer horizon (in [s]).</
         tool_tip>
63     </param_4>
64     <params_5>
65       <name>sensors</name>
66       <param_0>
67         <name>size</name>
68         <type>int</type>
```

```

69         <value>@array_size@</value>
70         <tool_tip>Enter the number of sensors that the
           robot have.</tool_tip>
71     </param_0>
72     <array_1>
73         <name>sensor@index@</name>
74         <param_0>
75             <name>plugin_name</name>
76             <type>string</type>
77             <tool_tip>Enter the sensor plugin name.</tool_tip>
78         </param_0>
79         <param_1>
80             <name>topic_name</name>
81             <type>string</type>
82             <tool_tip>Enter the sensor topic name.</tool_tip>
83         </param_1>
84         <param_2>
85             <name>timeout_duration</name>
86             <type>double</type>
87             <default>5.0</default>
88             <tool_tip>Enter the duration in which the sensor
           is considered unreachable (in [s]).</tool_tip>
89         </param_2>
90         <param_3>
91             <name>buffer_horizon</name>
92             <type>double</type>
93             <default>5.0</default>
94             <tool_tip>Enter the sensor buffer horizon length
           (in [s]).</tool_tip>
95         </param_3>
96     </array_1>
97 </params_5>
98 <params_6>
99     <name>behaviour_sets</name>
100    <param_0>
101        <name>size</name>
102        <type>int</type>
103        <value>@array_size@</value>

```



```

104         <tool_tip>Enter the number of behaviour sets that
           the robot have.</tool_tip>
105     </param_0>
106     <array_1>
107         <name>behaviour_set@index@</name>
108         <param_0>
109             <name>task_id</name>
110             <value>@task_id@</value>
111             <type>string</type>
112             <tool_tip>Enter the task id in which this
                       behaviour set is related to.</tool_tip>
113         </param_0>
114         <param_1>
115             <name>task_expected_duration</name>
116             <type>double</type>
117             <tool_tip>Enter the expected duration that this
                       robot takes to accomplish this task (in [s]).<
                       /tool_tip>
118         </param_1>
119         <params_2>
120             <name>motivational_behaviour</name>
121             <param_0>
122                 <name>threshold</name>
123                 <type>double</type>
124                 <tool_tip>Enter this motivational behaviour
                           threshold for activation.</tool_tip>
125             </param_0>
126             <params_1>
127                 <name>acquiescence</name>
128                 <param_0>
129                     <name>yielding_delay</name>
130                     <type>double</type>
131                     <tool_tip>Enter the yielding delay of the
                           acquiescence module (in [s]).</tool_tip>
132                 </param_0>
133                 <param_1>
134                     <name>giving_up_delay</name>
135                     <type>double</type>

```

```

136         <tool_tip>Enter the giving up delay of the
           acquiescence module (in [s]).</tool_tip>
137     </param_1>
138 </params_1>
139 <params_2>
140     <name>impatience</name>
141     <param_0>
142         <name>fast_rate</name>
143     <type>double</type>
144     <tool_tip>Enter the motivation fast rate of the
           impatience module.</tool_tip>
145     </param_0>
146 </params_2>
147 <params_3>
148     <name>sensory_feedback</name>
149     <param_0>
150         <name>plugin_name</name>
151         <type>string</type>
152         <tool_tip>Enter the plugin name of the
           sensory feedback module.</tool_tip>
153     </param_0>
154     <params_1>
155         <name>sensors</name>
156         <param_0>
157             <name>size</name>
158             <type>int</type>
159             <value>@array_size@</value>
160             <tool_tip>Enter the number of sensors used
           in this sensory feedback module.</
           tool_tip>
161         </param_0>
162         <array_1>
163             <name>sensor@index@</name>
164             <param_0>
165                 <name>topic_name</name>
166                 <type>string</type>
167                 <tool_tip>Enter the sensor topic name.</
           tool_tip>
168             </param_0>

```

```

169         </array_1>
170     </params_1>
171 </params_3>
172 </params_2>
173 </array_1>
174 </params_6>
175 </config_0>
176 <config_1>
177     <id>tasks_alliance_params</id>
178     <params_0>
179         <name>tasks</name>
180         <param_0>
181             <name>size</name>
182             <type>int</type>
183             <value>@array_size@</value>
184             <tool_tip>Enter the number of tasks in the system.<
                /tool_tip>
185         </param_0>
186         <array_1>
187             <name>task@index@</name>
188             <param_0>
189                 <name>id</name>
190                 <type>string</type>
191                 <tool_tip>Enter the task id.</tool_tip>
192             </param_0>
193             <param_1>
194                 <name>name</name>
195                 <type>string</type>
196                 <tool_tip>Enter the task name.</tool_tip>
197             </param_1>
198         </params_2>
199         <name>layers</name>
200         <param_0>
201             <name>size</name>
202             <type>int</type>
203             <value>@array_size@</value>
204             <tool_tip>Enter the number of layers in the
                task.</tool_tip>
205         </param_0>

```

```

206         <array_1>
207             <name>layer@index@</name>
208             <param_0>
209                 <name>plugin_name</name>
210                 <type>string</type>
211                 <tool_tip>Enter the layer plugin name.</
                    tool_tip>
212             </param_0>
213         </array_1>
214     </params_2>
215 </array_1>
216 </params_0>
217 </config_1>
218 <config_2>
219     <id>layers_alliance_params</id>
220 </config_2>
221 </configs>
222 <launches>
223     <launch_0>
224         <id>alliance</id>
225         <includes>
226             <include_0>
227                 <file>$(find alliance)/launch/alliance.launch</file>
228                 <args>
229                     <arg_0>
230                         <name>robot_id</name>
231                         <value>@robot_id@</value>
232                     </arg_0>
233                     <arg_1>
234                         <name>robot_params</name>
235                         <value>$(find @package@)/config/
                            @robot_id@_alliance_params.yaml</value>
236                     </arg_1>
237                     <arg_2>
238                         <name>tasks_params</name>
239                         <value>$(find @package@)/config/
                            tasks_alliance_params.yaml</value>
240                     </arg_2>

```

```
241         <arg_3>
242             <name>layers_params</name>
243             <value>$(find @package@)/config/
                layers_alliance_params.yaml</value>
244         </arg_3>
245     </args>
246 </include_0>
247 </includes>
248 </launch_0>
249 </launches>
250 <widgets>
251     <widget_0>
252         <plugin_name>rqt_alliance/alliance_plugin</plugin_name>
253     </widget_0>
254 </widgets>
255 </rqt_mrta>
```

APÊNDICE B – Exemplo de um arquivo de configuração de aplicação para o *rqt_mrta*

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <rqt_mrta format="application">
3   <application>
4     <name>Alliance Test</name>
5     <url></url>
6     <robots>
7       <robot_0>
8         <id>/robot1</id>
9       </robot_0>
10      <robot_1>
11        <id>/robot2</id>
12      </robot_1>
13      <robot_2>
14        <id>/robot3</id>
15      </robot_2>
16      <robot_3>
17        <id>/robot4</id>
18      </robot_3>
19      <robot_4>
20        <id>/robot5</id>
21      </robot_4>
22    </robots>
23  </application>
24 </rqt_mrta>
```

APÊNDICE C – Exemplo de manifesto de pacote para arquitetura

```
1 <?xml version="1.0"?>
2 <package>
3   <name>alliance</name>
4   <version>1.0.5</version>
5   <description>This package implements the ALLIANCE multi-
      robot task allocation architecture.</description>
6
7   <maintainer email="adrianohrl@unifei.edu.br">Adriano
      Henrique Rossette Leite</maintainer>
8   <license>Apache 2.0</license>
9   <url type="website">https://github.com/adrianohrl/alliance<
      /url>
10  <author email="adrianohrl@unifei.edu.br">Adriano Henrique
      Rossette Leite</author>
11
12  <buildtool_depend>catkin</buildtool_depend>
13
14  <build_depend>alliance_msgs</build_depend>
15  <build_depend>pluginlib</build_depend>
16  <build_depend>roscpp</build_depend>
17
18  <run_depend>alliance_msgs</run_depend>
19  <run_depend>pluginlib</run_depend>
20  <run_depend>roscpp</run_depend>
21  <run_depend>rqt_mrta</run_depend>
22
23  <export>
24    <rqt_mrta architecture="${prefix}/rqt_mrta.xml" />
25  </export>
26 </package>
```

APÊNDICE D – Exemplo de manifesto de pacote para aplicação

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <package>
3   <author email="adrianohtml@gmail.com">Adriano Henrique
      Rossette Leite</author>
4   <buildtool_depend>catkin</buildtool_depend>
5   <description>Esta aplicacao usa a arquitetura ALLIANCE
      para resolver o problema de alocao de tarefa na
      patrulha realizada por varios robos no Campus da
      Universidade Federal de Itajuba.</description>
6   <export>
7     <rqt_mrta application="${prefix}/rqt_mrta.xml"/>
8   </export>
9   <license>BSD</license>
10  <maintainer email="adrianohtml@unifei.edu.br">Adriano
      Henrique Rossette Leite</maintainer>
11  <name>patrulha</name>
12  <run_depend>alliance</run_depend>
13  <run_depend>alliance_test</run_depend>
14  <run_depend>rqt_mrta</run_depend>
15  <version>1.0.0</version>
16 </package>
```


APÊNDICE E – ALLIANCE

Esta é uma arquitetura totalmente distribuída, tolerante a falhas, que visa atingir controle cooperativo e atender os requisitos de uma missão à ser desempenhada por um grupo de robôs heterogêneos (PARKER, 1998). Cada robô é modelado usando uma aproximação baseada em comportamentos. A partir do estado do ambiente e dos outros robôs cooperadores, uma configuração de comportamento é selecionada conforme sua respectiva função de realização de tarefa na camada de alto nível de abstração. Cada configuração de comportamento permite controlar os atuadores do robô em questão de um modo diferente.

Sejam $R = \{r_1, r_2, \dots, r_n\}$, o conjunto de n robôs heterogêneos, e $A = \{a_1, a_2, \dots, a_m\}$, o conjunto de m sub-tarefas independentes que compõem uma dada missão. Na arquitetura ALLIANCE, cada robô r_i possui um conjunto de p configurações de comportamento, dado por $C_i = \{c_{i1}, c_{i2}, \dots, c_{ip}\}$. Cada configuração de comportamento fornece ao seu robô uma função de realização de tarefa em alto nível, conforme definido em (BROOKS, 1986). Por fim, é possível saber qual tarefa em A é executada por r_i quando sua configuração de ativação c_{ik} é ativa. Tal informação é obtida através da função $h_i(c_{ik})$, a qual pertence ao conjunto de n funções $H : C_i \rightarrow A$, $H = \{h_1(c_{1k}), h_2(c_{2k}), \dots, h_n(c_{nk})\}$.

A ativação de uma dada configuração de comportamento c_{ij} do robô r_i para a execução da tarefa $h_i(c_{ij})$ em um dado instante, é dada pelo cálculo de motivação do seu comportamento motivacional. Por sua vez, cada comportamento motivacional possui um conjunto de módulos que tem a responsabilidade de monitorar alguma informação relevante sobre o sistema. Será detalhado a seguir o papel de cada um desses módulos e suas contribuições para o cálculo de motivação.

A primeira função, definida pela Equação E.1, tem como responsabilidade identificar quando a configuração de comportamento c_{ij} é aplicável. Esta função lógica é implementada no módulo de *feedback* sensorial, o qual observa constantemente as condições do ambiente por meio de sensores e, então, verifica se o sistema é favorecido se c_{ij} estiver ativada.

$$aplicável_{ij}(t) = \begin{cases} 1, & \text{se o módulo de } feedback \text{ sensorial da configuração de comportamento } c_{ij} \text{ do robô } r_i \text{ indicar que esta configuração é aplicável mediante ao estado atual do ambiente no instante } t; \\ 0, & \text{caso contrário.} \end{cases} \quad (E.1)$$

A Equação E.2 mostra uma das funções lógicas que também compõe o cálculo para ativação de c_{ij} . Seu papel, neste cálculo, é garantir que o robô r_i só tenha uma configuração de comportamento ativa por vez. Essa função é implementada pelo módulo de supressão, o qual observa a ativação das demais configurações de comportamento de r_i .

$$inibida_{ij}(t) = \begin{cases} 1, & \text{se outra configuração de comportamento } c_{ik} \\ & \text{(com } k \neq j) \text{ está ativa no robô } r_i \text{ no instante} \\ & t; \\ 0, & \text{caso contrário.} \end{cases} \quad (\text{E.2})$$

Cada configuração de comportamento c_{ij} possui um módulo de comunicação que auxilia vários outros módulos de c_{ij} por meio do monitoramento da comunicação entre os robôs do sistema. Este módulo mantém o histórico das atividades dos demais robôs do sistema no que diz respeito à execução da tarefa $h_i(c_{ij})$. Deste modo, os demais módulos de c_{ij} podem consultar se os outros robôs estavam executando a tarefa $h_i(c_{ij})$ em um dado intervalo de tempo $[t_1; t_2]$, conforme mostra a Equação E.3. Existem dois parâmetros no ALLIANCE que influenciam diretamente no módulo de comunicação de cada comportamento motivacional. O primeiro parâmetro, ρ_i , define a frequência com que r_i atualiza suas configurações de comportamento e publica seu estado atual, no que diz respeito à arquitetura. O segundo parâmetro, τ_i , indica a duração de tempo máxima que o robô r_i permite ficar sem receber mensagens do estado de qualquer outro robô do sistema. Quando esta duração é excedida para um dado robô r_k , o robô r_i passa considerar que r_k cessou sua atividade. A utilização deste parâmetro visa prever falhas de comunicação e de mal funcionamento.

$$recebida_{ij}(k, t_1, t_2) = \begin{cases} 1, & \text{se o robô } r_i \text{ recebeu mensagem do robô } r_k \\ & \text{referente à tarefa } h_i(c_{ij}) \text{ dentro do intervalo} \\ & \text{de tempo } [t_1; t_2], \text{ em que } t_1 < t_2; \\ 0, & \text{caso contrário.} \end{cases} \quad (\text{E.3})$$

A próxima função tem a incumbência de reiniciar o cálculo para a ativação da configuração de comportamento c_{ij} . Essa função lógica é impulsionada apenas uma vez para cada robô que tenta executar a tarefa $h_i(c_{ij})$. Isto é, no instante em que acontece a primeira rampa de subida na Equação E.3 para cada robô r_k , esta função retorna um nível lógico alto. Essa condição evita que problemas de falhas persistentes não comprometam a completude da missão.

$$reiniciada_{ij}(t) = \exists x, (recebida_{ij}(x, t - dt, t) \wedge \neg recebe_{ij}(x, 0, t - dt)) \quad (\text{E.4})$$

onde dt é o tempo decorrido desde a última verificação de comunicação.

A Equação E.5 auxilia o módulo de aquiescência no cálculo de desistência para a desativação de c_{ij} . Baseando-se no histórico de ativação de c_{ij} , o módulo de comportamento motivacional disponibiliza essa função lógica que verifica se c_{ij} ficou mantida ativa por um dado período de tempo até o instante desejado.

$$ativa_{ij}(\Delta t, t) = \begin{cases} 1, & \text{se a configuração de comportamento } c_{ij} \text{ do} \\ & \text{robô } r_i \text{ estiver ativa por mais de } \Delta t \text{ unidades} \\ & \text{de tempo no instante } t; \\ 0, & \text{caso contrário.} \end{cases} \quad (\text{E.5})$$

O módulo de aquiescência monitora o tempo decorrido após a ativação da configuração de comportamento c_{ij} do robô r_i com o auxílio da Equação E.5. São duas as suas preocupações: (1) verificar se c_{ij} permaneceu ativa por mais tempo que o esperado e (2) verificar se o tempo decorrido após um outro robô r_k ter iniciado a execução da tarefa $h_i(c_{ij})$, enquanto c_{ij} estava ativa, tenha excedido o tempo configurado para r_i passar sua vez para esse outro robô. A Equação E.6 define as condições em que r_i está aquiescente à desativação de c_{ij} .

$$\begin{aligned} aquiescente_{ij}(t) = & (ativa_{ij}(\psi_{ij}(t), t) \wedge \exists x, recebida_{ij}(x, t - \tau_i, t)) \\ & \vee ativa_{ij}(\lambda_{ij}(t), t) \end{aligned} \quad (\text{E.6})$$

onde $\psi_{ij}(t)$ é a duração de tempo que r_i deseja manter a configuração de comportamento c_{ij} ativa antes de dar preferência para outro robô executar a tarefa $h_i(c_{ij})$; e $\lambda_{ij}(t)$ é a duração de tempo que r_i deseja manter c_{ij} ativa antes de desistir para possivelmente tentar outra configuração de comportamento.

A impaciência de r_i para a ativação de c_{ij} cresce linearmente mediante a taxa de impaciência instantânea. Assim, o módulo de impaciência de c_{ij} é responsável por identificar falhas de execução da tarefa $h_i(c_{ij})$ por outros robôs do sistema e quantificar a insatisfação de r_i concernente à essa tarefa, conforme visto na Equação E.7. Para isso, três parâmetros são utilizados: (1) $\phi_{ij}(k, t)$, o qual estabelece o tempo máximo que r_i permite a um outro robô r_k executar a tarefa $h_i(c_{ij})$ antes dele próprio iniciar sua tentativa; (2) $\delta_{slow_{ij}}(k, t)$, que determina a taxa de impaciência do robô r_i com respeito à configuração de comportamento c_{ij} enquanto o robô r_k está executando a tarefa correspondente à c_{ij} ; e (3) $\delta_{fast_{ij}}(t)$, que determina a taxa de impaciência de r_i com relação à c_{ij} quando nenhum outro robô está executando a tarefa $h_i(c_{ij})$.

$$impaciência_{ij}(t) = \begin{cases} \min_x \delta_{slow_{ij}}(x, t), & \text{se } recebida_{ij}(x, t - \tau_i, t) \wedge \neg recebida_{ij}(x, 0, t - \\ & \phi_{ij}(x, t); \\ \delta_{fast_{ij}}(t), & \text{caso contrário.} \end{cases} \quad (\text{E.7})$$

Note que o método usado incrementa a motivação à uma taxa que permita que o robô mais lento r_k continue sua tentativa de execução de $h(c_{ij})$, desde que seja respeitada a duração máxima estipulada pelo parâmetro $\phi_{ij}(k, t)$.

A Equação E.8 mostra a função de motivação, a qual combina todas as funções mencionadas anteriormente para a ativação da configuração de comportamento c_{ij} . Seu valor inicial é nulo e aumenta mediante a taxa de impaciência instantânea de r_i para ativar c_{ij} quando satisfeitas as seguintes condições: (1) c_{ij} seja aplicável, (2) mas não tenha sido inibida, (3) nem reiniciada; (4) e, ainda, r_i não seja aquiescente em desistir de manter c_{ij} ativa. Quando uma das condições citadas não é satisfeita, seu valor volta a ser nulo.

$$\begin{aligned} \text{motivação}_{ij}(0) &= 0 \\ \text{motivação}_{ij}(t) &= (\text{motivação}_{ij}(t - dt) + \text{impaciência}_{ij}(t)) \\ &\quad \times \text{aplicável}_{ij}(t) \times \text{inibida}_{ij}(t) \\ &\quad \times \text{reiniciada}_{ij}(t) \times \text{aquiescente}_{ij}(t). \end{aligned} \quad (\text{E.8})$$

Assim que a motivação de r_i para ativar c_{ij} ultrapassa o limite de ativação, essa configuração de comportamento é ativada, conforme a Equação E.9:

$$\text{ativa}_{ij}(t) = \text{motivação}_{ij}(t) \geq \theta \quad (\text{E.9})$$

onde θ é o limite de ativação.

Fazendo uma análise das equações acima, verifica-se que, enquanto sua motivação cresce, é possível estimar quanto tempo resta para que a configuração de comportamento c_{ij} se torne ativa.

$$\overline{\Delta t}_{\text{ativação}_{ij}} = \frac{\theta - \text{motivação}_{ij}(t)}{\text{impaciência}_{ij}(t)\rho_i} \quad (\text{E.10})$$

onde ρ_i é a frequência aproximada, em [Hz], com que r_i atualiza as motivação das configurações de comportamento em C_i e, ainda, publica seu estado comportamental. Como a taxa de impaciência não é constante, a Equação E.10 é apenas uma estimativa, dada em [s].

Em conformidade com o que foi exposto, pode-se observar que é possível normalizar todas as funções de motivação, de modo que a imagem de cada uma delas pertença ao intervalo $[0; 1] \subset \mathbb{R}_+$. Para isso, é necessário: (1) parametrizar o módulo de impaciência de cada configuração de comportamento c_{ij} , de maneira que a imagem da sua função de taxa de impaciência instantânea pertença ao intervalo $(0; 1) \subset \mathbb{R}_+^*$; além disso, (2) atribuir o valor unitário ao limite de ativação; bem como, (3) saturar a função de motivação no limite de ativação. Como resultado, as Equações E.9 e E.10 podem ser rescritas como as Equações E.11 e E.12, respectivamente.

$$\text{ativa}_{ij}(t) = \text{motivação}_{ij}(t) == 1 \quad (\text{E.11})$$

$$\overline{\Delta t_{ativa\tilde{c}\tilde{a}o_{ij}}} = \frac{1 - motiva\tilde{c}\tilde{a}o_{ij}(t)}{impaci\tilde{e}ncia_{ij}(t)\rho_i} \quad (\text{E.12})$$

[Parker \(1996\)](#) desenvolveu também uma variação do ALLIANCE, chamada L-ALLIANCE, capaz de estimar alguns parâmetros do ALLIANCE durante a fase de aprendizado.