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

Um *framework* para o Desenvolvimento de Arquiteturas para Alocação de Tarefas em Sistema Multirrobô

Adriano Henrique Rossette Leite

Itajubá, 16 de abril de 2018

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

Adriano Henrique Rossette Leite

Um *framework* para o Desenvolvimento de Arquiteturas para Alocação de Tarefas em Sistema Multirrobô

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 Elétricos Industriais

Orientador: Prof. Dr. Guilherme Sousa Bastos

16 de abril de 2018
Itajubá

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

Um *framework* para o Desenvolvimento de Arquiteturas para Alocação de Tarefas em Sistema Multirrobô

Adriano Henrique Rossette Leite

Dissertação aprovada por banca examinadora em 16 de Março de 2018, 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
2018

Adriano Henrique Rossette Leite

Um *framework* para o Desenvolvimento de Arquiteturas para Alocação de Tarefas em Sistema Multirrobô

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, 16 de Março de 2018:

Prof. Dr. Guilherme Sousa Bastos
Orientador

Prof. Dr. Edson Prestes

Prof. Dr. Laércio Augusto Baldochi Júnior

Itajubá
16 de abril de 2018

Agradecimentos

À Deus por permitir meu ingresso no mestrado em uma universidade de excelência trabalhando com um tema que eu gosto tanto. Além disso, me deu a oportunidade de estar acompanhado de pessoas que me auxiliaram durante este árduo trabalho. Devo a Ti tudo o que eu tenho e o que sou. Dedico mais este trabalho à Ti.

Aos meus pais, Marcos e Eunice, que me deram bons conselhos e me aturaram meus momentos de estresse. À minha namorada, Marina, quem eu tanto amo, quem me ajuda a ser um homem mais sereno, mais sábio e menos ansioso. Às minhas queridas irmãs, Jaqueline e Poliane, as quais eu quero muito bem. Aos meus amigos, em especial ao pessoal da Aliança Bíblica Universitária (ABU) e das Repúblicas Lodu Altu e Matu Altu (Família Altu). Jamais esquecerei das conversas e risadas que trocamos aqui em Itajubá. À tia Olmira e sua filha Délia, genro e netos por me acolherem incondicionalmente em Itajubá. Não consigo imaginar como seria esta fase em minha vida sem cada um de vocês. Todos foram fundamentais para a conclusão do meu título de mestre. Que Deus abençoe a cada um assim como vocês têm abençoado tanto a minha vida.

Ao meu orientador com quem trabalho a 7 anos pela dedicação e ajuda. Ao MSc. Ricardo Emerson Júlio e ao BSc. Luiz Arthur Silva Moura por me ajudarem na revisão final deste trabalho. Quanta paciência vocês tiveram ao tirar um tempo para me ajudar neste processo. Também aos meus colegas da equipe Expertinos e do laboratório LRO. Que vocês continuem avançando e conquistando o espaço de vocês dentro da área acadêmica.

Por fim, à Capes pelo apoio financeiro durante estes 2 anos de pesquisa acadêmica.

"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 um *framework* cujo propósito é facilitar o desenvolvimento de arquiteturas de alocação de tarefas em sistema multirrobô. O *framework* proposto é denominado *TAlMech*, acrônimo para *Task Allocation Mechanisms*, pois este encapsula mecanismos que são utilizados por arquiteturas baseadas em comportamento e negociação para alocar tarefas aos robôs de um sistema. Para isso, é feito um estudo sobre os *frameworks* utilizados em aplicações de robótica, as características presentes em um sistema multirrobô (MRS) e o problema de alocação de tarefas em sistemas de múltiplos robôs (MRTA). Além disso, as arquiteturas que resolvem problemas desta natureza são revisadas abordando as características de arquiteturas baseadas em comportamento, em negociação e em métodos de otimização. Enfim, é apresentado o funcionamento de algumas arquiteturas de comportamento e de negociação. A formulação do *framework TAlMech* foi realizada a partir de uma comparação entre as arquiteturas revisadas. Cinco ensaios foram executados para testar os mecanismos do *framework TAlMech*. Em um dos ensaios, o *framework* foi configurado para funcionar como uma aproximação da arquitetura Murdoch. A arquitetura ALLIANCE foi desenvolvida a partir do *TAlMech* e testada em outro ensaio. Os resultados mostraram que o *framework TAlMech* pode ser configurado e estendido para simplificar o desenvolvimento de arquiteturas de alocação de tarefa em sistemas com múltiplos robôs.

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

Abstract

This work presents a framework that aims to facilitate the development of multi-robot task allocation architectures. The proposed framework, named as *TAlMech*, stands for Task Allocation Mechanisms because it encapsulates mechanisms that are used by behavior-based and negotiation-based architectures for assigning a set of tasks to a bundle of robots in a system. For that, the problem of this paper conducts a study about robotics frameworks, multi-robot system (MRS) features and multi-robot task allocation (MRTA). Moreover, it reviews characteristics of architectures that solve problems of this nature. After all, it presents the functioning of some behavior and negotiation based approaches. The *TAlMech* framework formulation was based on a comparison between the reviewed architectures. Five experiments were performed to test the *TAlMech* framework mechanisms. In one of them, the framework was configured to work as a Murdoch approach. In another one, the ALLIANCE architecture was developed using some mechanisms of *TAlMech* and tested, as well. The results showed the *TAlMech* framework can be configured and extended to simplify the development of multi-robot task allocation architectures.

Key-words: Framework. Multi-Robot System. Task Allocation. Architecture. ROS.

Lista de figuras

Lista de tabelas

Lista de abreviaturas e siglas

ABNT	Associação Brasileira de Normas Técnicas
API	*Application Programming Interface*
CBR	Competição Brasileira de Robótica
CNP	*Contract Net Protocol*
CSV	*Comma-Separated Values*
GPU	*Graphics Processing Unit*
GUI	*Graphical User Interface*
IoT	*Internet of Things*
k-SAAP	*k-Hop Simple Aggregation Auction Protocol*
k-SAP	*k-Hop Simple Auction Protocol*
LRO	Laboratório de Robótica
MAS	*Multi-Agent System*
MRS	*Multi-Robot System*
MRTA	*Multi-Robot Task Allocation*
MVC	*Model-View-Controller*
OAP	*Optimal Assignment Problem*
P3DX	*Adept MobileRobots Pioneer 3 DX*
RDE	*Robotic Development Evironment*
ROS	*Robot Operating System*
SAAP	*Simple Aggregation Auction Protocol*
SAP	*Simple Auction Protocol*
UI	*User Interface*
UML	*Unified Modeling Language*

UNIFEI	Universidade Federal de Itajubá
XML	*Extensible Markup Language*
YAML	*YAML Ain't Markup Language*

Sumário

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 multirrobô, do inglês *Multi-Robot Systems* (MRS). Um sistema multirrobô 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 a 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 Matarić (2004) sugeriram uma taxonomia independente do domínio, mostrando algumas arquiteturas que atendem 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, o ROS possibilitou a integração de trabalhos desenvolvidos por equipes distintas de pesquisa em robótica, pois este simplifica o desenvolvimento de processos e dá suporte à comunicação e interoperabilidade desses processos. 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 multirrobô, houve poucas tentativas de aproximação genérica delas em projetos baseados em ROS. Li *et al.* (2016) elaboraram um pacote ROS[1] 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 que desenvolveu quatro protocolos diferentes de leilão: *Simple Auction Protocol* (SAP), *k-Hop Simple Auction Protocol* (k-SAP), *Simple Aggregation Auction Protocol* (SAAP) e *k-Hop Simple Aggregation Auction Protocol* (k-SAAP).

Existem poucos projetos de arquiteturas MRTA para serem utilizados em aplicações de robótica no ROS, quando comparado com a quantidade de arquiteturas propostas na literatura. Este fato se dá pela dificuldade de codificar essas arquiteturas. Geralmente os trabalhos que propõem novas arquiteturas MRTA não apresentam todos detalhes necessários para implementá-las e, muito menos, disponibilizam o código de programação desenvolvido. Com isso, é proposto o *framework TAlMech* que visa auxiliar no desenvolvimento de arquiteturas MRTA com a ajuda do ROS.

1.2 Objetivos

Este trabalho tem como principal objetivo desenvolver um *framework* que encapsule mecanismos que auxiliem o desenvolvimento de arquiteturas de alocação de tarefas para sistema multirrobô. O *framework* e seus mecanismos devem ser desenvolvidos buscando obter as seguintes características:

- Permitir a extensão de novos mecanismos e novas estratégias de controle;
- Possibilitar representações independentes do domínio das aplicações em robótica;
- Fornecer ferramentas métricas para comparação de arquiteturas desenvolvidas a partir do *framework*;
- Ser um projeto de código aberto bem documentado para a contribuição, a manutenção e a utilização dos mecanismos do *framework* pela comunidade de robótica.

[1] <http://wiki.ros.org/micros_mars_task_alloc>
[2] <http://wiki.ros.org/auction_methods_stack>

1.3 Contribuições

A principal contribuição deste trabalho é o desenvolvimento do *framework TAlMech* que encapsula quatro mecanismos comumente utilizados em arquiteturas de alocação de tarefas em sistema multirrobô, os quais são:

- **Cálculo de Utilidade**: representação modular que permite uma formulação genérica do cálculo de utilidade dos agentes de um sistema multirrobô heterogêneo;
- **Negociação por Leilão**: encapsula o protocolo de comunicação entre leiloeiros e licitantes para a atribuição de tarefas em um sistema multirrobô;
- **Monitoramento de Comportamento**: disponibiliza um histórico de comportamento dos agentes de um sistema multirrobô para consulta;
- **Controle de Ativação de Comportamento**: representação genérica da função de ativação dos comportamentos e chamada do comportamento ativado.

Outras contribuições deste trabalho foram o desenvolvimento de um gerador randômico de tarefas que pode ser configurado para alimentar o sistema periodicamente com tarefas aleatórias com a finalidade de testar arquiteturas que utilizam o *framework TAlMech*, bem como, o desenvolvimento de uma ferramenta métrica para análise estatística e probabilística do mecanismo de negociação por leilão.

1.4 Estrutura do Trabalho

No Capítulo 2, é feita uma revisão bibliográfica sobre as características de um sistema multirrobô, o problema de alocação de tarefa e a classificação desses problemas segundo uma taxonomia independente do domínio. O Capítulo 3 trata sobre o desenvolvimento do *framework TAlMech*, detalhando cada um de seus mecanismos. Em seguida, no Capítulo 4, um conjunto de tarefas é gerado, observado e armazenado para ser utilizado para testar os mecanismo de cálculo de utilidade e negociação por leilão do *framework TAlMech*. Ainda neste capítulo, os mecanismo de monitoramento e controle de ativação de comportamento do *TAlMech* são usados no desenvolvimento da arquitetura ALLIANCE. Esta arquitetura é utilizada para alocar tarefas em uma aplicação de patrulhamento. Finalmente, no Capítulo 5, são discutidas as conclusões e possibilidades de melhorias futuras para o *framework TAlMech*.

2 Revisão teórica

Primeiramente, os conceitos de *frameworks*, *middlewares* e *arquiteturas* para aplicações robóticas são diferenciados. Posteriormente, é apresentado o conceito de sistema multirrobô, suas vantagens e características. Em seguida, o problema de atribuição de tarefas em sistemas com vários robôs é apresentado juntamente com suas taxonomias. Enfim, são apresentadas algumas abordagens de arquiteturas que visam resolver este problema.

2.1 *Frameworks*, *Middlewares* e Arquiteturas para Robótica

Tsardoulias e Mitkas (2017) definem três conceitos fundamentais para o desenvolvimento de soluções em *software* para aplicações em robótica: *framework*, *middleware* e *arquitetura*.

Um *framework* para robótica consiste em uma coleção de ferramentas de *software*, bibliotecas e convenções com o intuito de simplificar a tarefa de desenvolvimento de um sistema complexo de robótica. Alguns *frameworks* possuem um RDE (*Robotic Development Environment*), isto é, uma interface gráfica de usuário que possibilita o desenvolvimento do projeto através de ferramentas gráficas. De acordo com Colon, Sahli e Baudoin (2006), componentes de *framework* são divididos em duas categorias: *componentes estruturais* e *de aplicação*. Os componentes estruturais oferecem serviços básicos de utilidade para outros componentes, enquanto componentes da aplicação são blocos de criação que estão diretamente relacionado com a aplicação. Iñigo-Blasco *et al.* (2012) e Tsardoulias e Mitkas (2017) comparam diversos *frameworks* desenvolvidos para sistemas robóticos, entre eles se encontram:

- **ARIA**: uma biblioteca em C++ que fornece várias ferramentas. Estas ferramentas incluem integração via *software* de entradas e saídas (I/O) com o *hardware* de robôs da MobileRobots/ActivMedia (MOBILEROBOTS, 2017a; ESUBALEW *et al.*, 2013);
- **ASEBA**: fornece um conjunto de ferramentas que permite iniciantes a programar robôs de forma fácil e eficiente (MAGNENAT *et al.*, 2011);
- **Carmen**: é uma coleção de *software open-source* para o controle de robôs móveis que foi construída de uma forma modular e fornece funções básicas de navegação como: (1) controle de base e sensores, (2) *logging*, (3) evasão de obstáculo, (4) localização,

(5) mapeamento e (6) planejamento de rota (MONTEMERLO; ROY; THRUN, 2003);

- **CLARAty**: integra novos algoritmos de robótica escritos em C++ como: controle de movimento, visão computacional, manipulação de objetos, navegação, localização, planejamento e execução de tarefas (VOLPE *et al.*, 2001);
- **OpenRDK**: é um *framework* leve que tem a intenção de acelerar a criação de sistemas robóticos de pequenos projetos de pesquisa (CALISI *et al.*, 2008).
- **Orca**: é um *framework* para o desenvolvimento de sistemas robóticos através de diagramas de bloco (BROOKS *et al.*, 2007);
- **OROCOS**: possui uma coleção de bibliotecas portáveis em C++ para fazer o controle de robôs. Este *framework* contem bibliotecas que implementam equações de cinemática e dinâmica, filtros Bayesianos, máquinas de estado, processos distribuídos, entre outros (BRUYNINCKX, 2001);
- **YARP**: incorpora um conjunto de bibliotecas, protocolos e ferramentas visando módulos claramente desacoplados para o desenvolvimento de sistemas de robótica (METTA; FITZPATRICK; NATALE, 2006);
- entre outros.

Um *middleware* deve providenciar a infraestrutura de comunicação entre os nós de processos computacionais durante a execução de um sistema robótico. Mohamed, Al-Jaroodi e Jawhar (2008) e Tsardoulias e Mitkas (2017) apontam diversos *middlewares* dedicados para aplicações robóticas, mostrando suas diferenças, vantagens e desvantagens. Entre eles, se destacam os *middlewares*:

- **HOP**: foi desenvolvido com o intuito de atender a demanda de aplicações relacionadas à Internet das Coisas (IoT). Ele é capaz de orquestrar transferência de comandos e dados entre os objetos da rede de/para serviços *web* e componentes de interfaces de usuário (SERRANO; GALLESIO; LOITSCH, 2006);
- **Miro**: é um *middleware* orientado a objetos para robôs, cuja principal motivação é melhorar o desenvolvimento de *software* para robôs móveis e sistemas de informação empresarial (UTZ *et al.*, 2002);
- e **ROS**: além de encapsular os serviços de um *middleware*, este *framework* reúne ferramentas, bibliotecas e convenções com o objetivo de reduzir a complexidade concernente à produção de *software* para aplicações em robótica (QUIGLEY *et al.*, 2009). O Apêndice A faz uma revisão dos seus principais conceitos.

Os conceitos de *framework* e *middleware* parecem bem similares. Entretanto, as funcionalidades de um *middleware* deveriam ser invisíveis ao desenvolvedor, enquanto um *framework* fornece uma API de serviços e módulos testados pela comunidade científica de robótica.

Por fim, na robótica, uma arquitetura define abstratamente como os módulos de um sistema robótico deverão ser interligados e como eles se interagiram. O maior desafio de uma arquitetura é ser definida de modo a atender um grande número de sistemas robóticos, pois estes possuem extrema diversidade. São exemplos de arquiteturas para sistemas robóticos:

- **AuRA**: é uma arquitetura robótica híbrida, pois sua parte de alto nível é um componente deliberativo enquanto a de baixo nível possui um esquema de controle comportamental reativo (ARKIN; BALCH, 1997);
- **DCA**: é uma arquitetura distribuída para o controle de robôs, não só em diferentes executáveis, mas também em computadores diferentes também (PETERSSON; AUSTIN; CHRISTENSENI, 2001);
- e **Saphira**: é um sistema sensorial e de controle integrado com o ARIA que visa resolver problemas como navegação, reconhecimento de objetos e planejamento de trajetória (KONOLIGE *et al.*, 1997).

2.2 Sistema Multirrobô

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 multirrobô, do inglês *Multi Robot 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 multirrobô (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 um tarefa;
- além de exibir maior confiabilidade, flexibilidade, escalabilidade e versatilidade.

Parker e Tang (2006) e Khamis, Hussein e Elmogy (2015) afirmam que as motivações mais comuns para o desenvolvimento de soluções para sistemas multirrobô são:

- a complexidade da tarefa é muito alta para um único robô executá-la;
- a tarefa é inerentemente distribuída;
- construir vários robôs com recursos limitados é muito mais fácil do que ter um único robô poderoso;
- múltiplos robôs podem resolver problemas muito mais rápido usando paralelismo;
- e a introdução de múltiplos robôs acrescenta robustez através da redundância.

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

2.2.1 Composição: homogêneo *versus* heterogêneo

Um sistema multirrobô pode ser formado por um conjunto de robôs homogêneos ou heterogêneos. As capacidades individuais dos robô 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.

2.2.2 Cooperação: cooperativo *versus* competitivo

Os robôs do sistema podem responder a estímulos externos cooperativamente ou competitivamente. Quando há cooperação entre os robôs, eles 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 interagem competitivamente.

2.2.3 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-se a direita, parar em interseções, manter uma distância de segurança do robô da frente, etc. Por outro lado, 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.

2.2.4 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 a 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.

2.2.5 Organização: centralizada *versus* distribuída

Sistemas organizados em uma forma centralizada possuem um líder que observa todo o sistema 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. Logo, sistemas centralizados podem obter uma solução ótima, enquanto sistemas distribuídos só podem obter soluções sub-ótimas.

2.3 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.

2.3.1 Definição formal

Zlot e Stentz (2006) definem 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 de tarefas $\mathcal{T}$, um conjunto de robôs $\mathcal{R}$ e uma função de custo para cada subconjunto de robôs $r \in 2^{\mathcal{R}}$ que especifique o custo de performance para cada subconjunto de tarefas, $c_r : 2^{\mathcal{T}} \rightarrow \mathbb{R}_+ \cup \{\infty\}$: procure a alocação $\mathcal{A}^* \in 2^{\mathcal{R}} \times 2^{\mathcal{T}}$ que minimiza a função objetivo global $\mathcal{C} : 2^{\mathcal{R}} \times 2^{\mathcal{T}} \rightarrow \mathbb{R}_+ \cup \{\infty\}$. —

Para que um algoritmo consiga encontrar uma solução ótima para este problema, é necessário levar em consideração todo o espaço de alocação $2^{\mathcal{R}} \times 2^{\mathcal{T}}$, 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.

2.3.2 Taxonomias

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

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: *SR* (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 repeito 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.

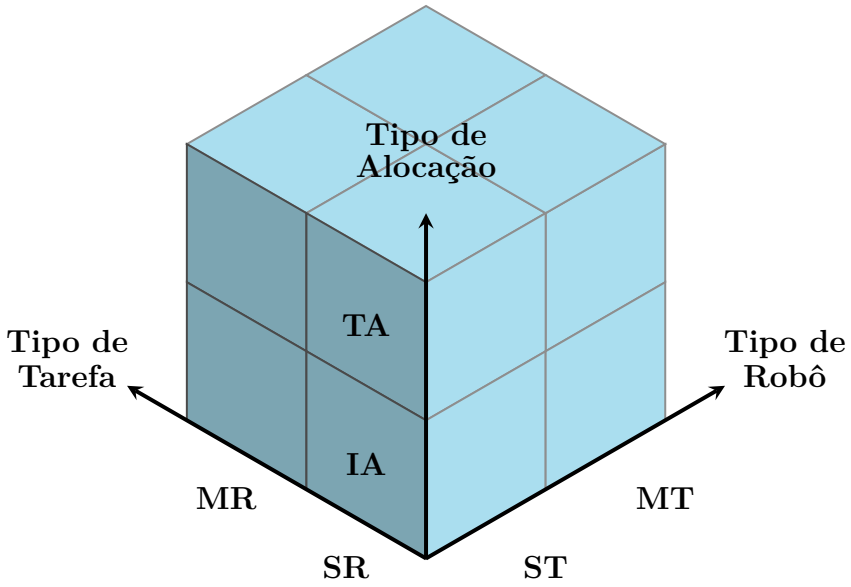


Figura 1 – Representação visual da taxonomia de três eixos sugerida por Gerkey e Matarić (2004).

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

A taxonomia de Gerkey e Matarić (2004) fundamentou a taxonomia denominada iTax de Korsah, Stentz e Dias (2013). Nesta nova taxonomia foi acrescentada uma camada acima da taxonomia de três eixos para classificar problemas de alocação levando em consideração as utilidades inter-relacionadas, restrições entre tarefas e tarefas complexas que podem ser decompostas em diversas maneiras. Desta forma, o iTax é composto de duas camadas: (1) a primeira camada possui apenas uma dimensão que define o grau de interdependência das utilidades dos robôs para as tarefas; enquanto, (2) a segunda camada fornece informação descritiva da configuração do problema utilizando a taxonomia de Gerkey e Matarić (2004).

A Figura 2 ilustra os diferentes graus de interdependência de utilidade que um problema MRTA pode ter. Verifica-se que tal dimensão pode assumir os seguintes valores:

- **ND**: acrônimo para *No Dependencies*, são problemas de alocação de tarefa em que a utilidade efetiva de um robô por uma tarefa não depende de outras tarefas ou agentes no sistema;
- **ID**: acrônimo para *Intra-Schedule Dependencies*, são problemas de alocação de tarefa em que a utilidade efetiva de um robô por uma tarefa depende das tarefas que ele está desempenhando, isto é, depende do seu escalonamento;
- **XD**: acrônimo para *Cross-Schedule Dependencies*, são problemas de alocação de tarefa em que a utilidade efetiva de um robô por uma tarefa depende não somente do seu próprio escalonamento, mas também dos escalonamentos dos demais robôs do sistema;
- e **CD**: acrônimo para *Complex Dependencies*, são problemas de alocação de tarefa em que a utilidade efetiva de um robô por uma tarefa depende dos escalonamentos dos outros robôs no sistema a partir da decomposição de tarefa escolhida de cada um deles.

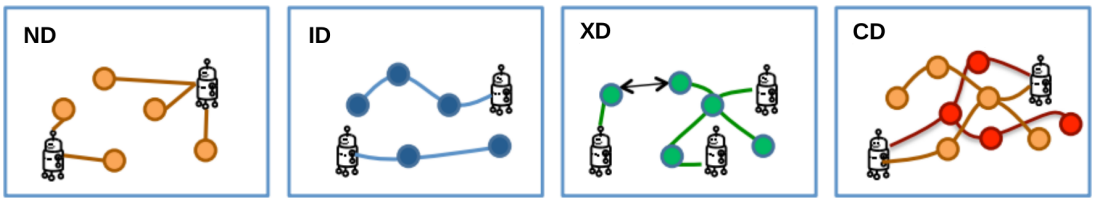


Figura 2 – Classificação de interdependência de utilidade entre as tarefas sugerida por Korsah, Stentz e Dias (2013).

A Figura 3 ilustra o domínio da taxonomia iTax. Sua camada 1 define o grau de interdependência entre as utilidades. E a camada 2 refina a configuração do problema com o grau de interdependência. Nota-se que o domínio do iTax não compreende o produto

cruzado total da taxonomia de Gerkey e Matarić (2004) com os graus de inter-relação entre as utilidades. Exemplificando, a sigla *XD [ST-SR-TA]* corresponde à categoria de problemas com dependência de escalonamento cruzado (*XD*) em que as atribuições podem ser agendadas (*TA*) de tarefas que exigem apenas um robô (*SR*) para a sua execução para robôs que podem executar apenas uma tarefa por vez (*ST*).

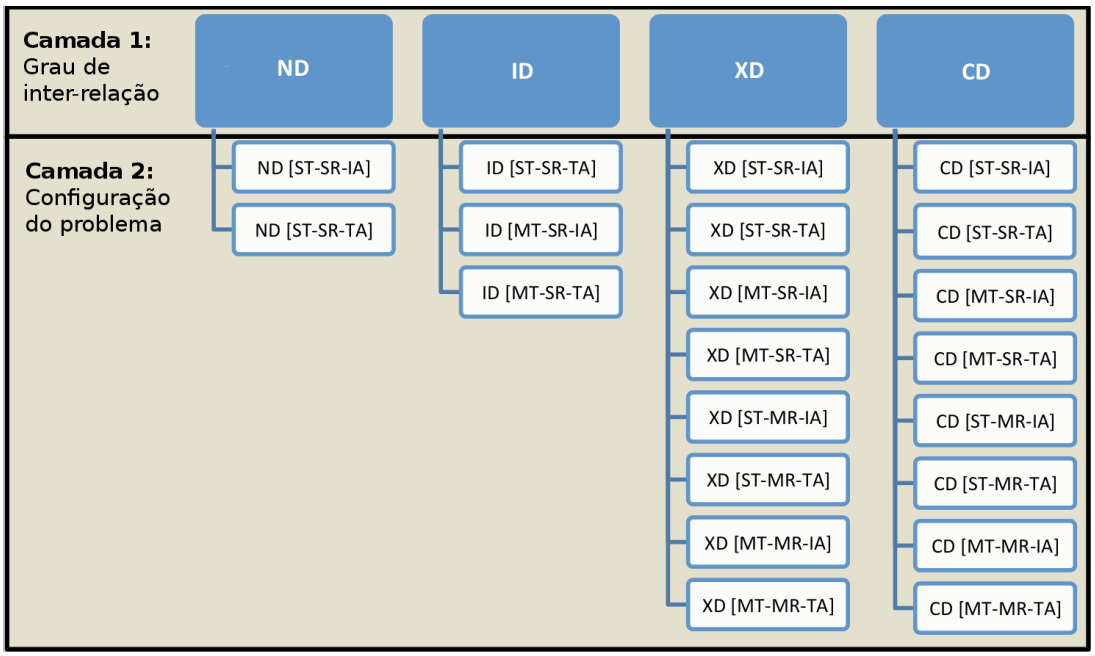


Figura 3 – Domínio da taxonomia iTax sugerida por Korsah, Stentz e Dias (2013).

Nunes *et al.* (2017) sugeriram MRTA/TOC, uma nova extensão da taxonomia de Gerkey e Matarić (2004). Essa taxonomia visa classificar problemas de alocação de tarefa em sistemas multirrobô que possuem restrições temporais e de ordenação. Neste caso, o eixo *TA* da taxonomia de Gerkey e Matarić (2004) foi expandido para incluí-las. As restrições temporais são expressas em forma de janelas de tempo (*TW*, acrônimo para *Time Window*), isto é, um intervalo temporal que começa no instante mais cedo em que a tarefa pode ser iniciada e termina no instante mais tarde em que a tarefa pode ser encerrada. As restrições de ordenação são expressas em forma de restrições de sincronização e de precedência (*SP*, acrônimo para *Synchronization and Precendence*), ou seja, restrições de sincronização determinam o instante inicial das tarefas, enquanto restrições de precedência especificam relações parciais de ordenação entre pares de tarefas.

2.4 Arquiteturas para Alocação de Tarefa em Sistema Multirrobô

As arquiteturas MRTA possuem a função de solucionar o problema de alocação de tarefas em um dado sistema multirrobô. Normalmente, elas se baseiam em modelos biológicos e econômicos, outras em métodos de otimização.

Khamis, Hussein e Elmogy (2015) afirmam que arquiteturas MRTA podem ser classificadas conforme o paradigma de organização do sistema: centralizado ou descentralizado. Em *arquiteturas centralizadas*, cada robô do sistema mantém conexão com um agente central, o qual processa todas informações do sistema e manda os comandos apropriados para os robôs do sistema. Esta forma de organização reduz a repetição de esforços, os custos e tempo. Além disso, arquiteturas centralizadas podem encontrar soluções ótimas para o sistema todo. Entretanto, sua maior desvantagem é a falta de robustez. Em outras palavras, se o agente central falhar, todo o sistema falha. Em *arquiteturas descentralizadas*, os papéis administrativos e de autoridades são distribuídos entre os robôs do sistema, de modo que não existe um único agente central que toma todas as decisões. A principal vantagem de arquiteturas distribuídas é sua robustez. Quando um robô do sistema falha, os outros robôs continuam trabalhando nas suas próprias tarefas; eventualmente, outro robô do sistema pode cobrir sua falha. Outras vantagens de arquiteturas distribuídas são escalabilidade e flexibilidade. Por outro lado, arquiteturas descentralizadas não são capazes de atribuir tarefas otimamente.

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 (1) *métodos de otimização*, (2) *comportamento* e (3) *negociação*. Arquiteturas baseadas em otimização são inspiradas em técnicas matemáticas de otimização cujo intuito é encontrar soluções ótimas para um conjunto de soluções disponíveis limitadas por um conjunto de restrições. Neste caso, a função objetivo é definida por critérios que descrevem quantitativamente o objetivo geral do sistema (HORST; PARDALOS; THOAI, 2000). As arquiteturas baseadas em comportamento e negociação são estudadas com profundidade em 2.4.1 e 2.4.2, respectivamente. Entretanto, as arquiteturas baseadas em técnicas de otimização não serão tratadas neste trabalho.

2.4.1 Arquiteturas baseadas em comportamento

As arquiteturas baseadas em comportamento geralmente são inspiradas em modelos biológicos para coordenar as atividades dos robôs no sistema. Nelas, os robôs possuem motivações internas que ditam o seu comportamento, isto é, qual tarefa será desempenhada por ele. A motivação interna de cada robô é influenciada pelo atual estado do próprio robô, do ambiente e o atual estado externo dos demais robôs. Cada robô do sistema percebe o estado do ambiente e o estado externo dos demais robôs através dos

seus sensores e a partir do sistema de comunicação, que permite o compartilhamento de informação entre os robôs.

Várias arquiteturas baseadas em comportamento foram propostas na literatura. É desejável entendê-las para verificar que características e mecanismos aparecem frequentemente nessas arquiteturas. Assim, serão revisadas as principais arquiteturas baseadas em comportamento encontradas na literatura.

Além de pesquisas realizadas nas principais bases de artigos científicos, essas arquiteturas também foram encontradas a partir dos principais artigos *surveys* e de taxonomia sobre alocação de tarefa em sistema multirrobô: (DUDEK *et al.*, 1996), (GERKEY; MATARIĆ, 2004), (KORSAH; STENTZ; DIAS, 2013) (YAN; JOUANDEAU; CHERIF, 2013), (KHAMIS; HUSSEIN; ELMOGY, 2015) e (NUNES *et al.*, 2017). A seguir, as principais arquiteturas de alocação de tarefa em sistema multirrobô baseadas em modelos de comportamento são revisadas.

2.4.1.1 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 $R = \{r_1, r_2, \cdots, r_n\}$, o conjunto de n robôs heterogêneos, e $A = \{a_1, a_2, \cdots, a_m\}$, 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 $C_i = \{c_{i1}, c_{i2}, \cdots, c_{ip}\}$. 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 $H : C_i \rightarrow A$, $H = \{h_1(c_{1k}), h_2(c_{2k}), \cdots, h_n(c_{nk})\}$.

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 2.1, tem como responsabilidade iden-

tificar 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.

$$aplicável_{ij}(t) = \begin{cases} 1, & \text{se o módulo de } \textit{feedback} \text{ sensorial da configuração de comportamento } c_{ij} \text{ do robô } r_i \text{ indicar que esta configuração é aplicável mediante ao estado atual do ambiente no instante } t; \\ 0, & \text{caso contrário.} \end{cases} \tag{2.1}$$

A Equação 2.2 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.

$$inibida_{ij}(t) = \begin{cases} 1, & \text{se outra configuração de comportamento } c_{ik} \text{ (com } k \neq j\text{) está ativa no robô } r_i \text{ no instante } t; \\ 0, & \text{caso contrário.} \end{cases} \tag{2.2}$$

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 2.3. 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.

$$recebida_{ij}(k, t_1, t_2) = \begin{cases} 1, & \text{se o robô } r_i \text{ recebeu mensagem do robô } r_k \\ & \text{referente à tarefa } h_i(c_{ij}) \text{ dentro do intervalo} \\ & \text{de tempo } [t_1; t_2], \text{ em que } t_1 < t_2; \\ 0, & \text{caso contrário.} \end{cases} \tag{2.3}$$

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 2.3 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.

$$reiniciada_{ij}(t) = \exists x, (recebida_{ij}(x, t - dt, t) \wedge \neg recebida_{ij}(x, 0, t - dt)) \tag{2.4}$$

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

A Equação 2.5 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.

$$ativa_{ij}(\Delta t, t) = \begin{cases} 1, & \text{se a configuração de comportamento } c_{ij} \text{ do} \\ & \text{robô } r_i \text{ estiver ativa por mais de } \Delta t \text{ unidades} \\ & \text{de tempo no instante } t; \\ 0, & \text{caso contrário.} \end{cases} \tag{2.5}$$

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 2.5. 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 2.6 define as condições em que r_i está aquiescente à desativação de c_{ij}.

$$\begin{aligned} aquiescente_{ij}(t) = (&ativa_{ij}(\psi_{ij}(t), t) \wedge \exists x, recebida_{ij}(x, t - \tau_i, t)) \\ &\vee ativa_{ij}(\lambda_{ij}(t), t) \end{aligned} \tag{2.6}$$

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 2.7. 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 $h_i(c_{ij})$.

$$impaciência_{ij}(t) = \begin{cases} \min\limits_{x} \delta_{slow_{ij}}(x,t), & \text{se } recebida_{ij}(x, t-\tau_i, t) \wedge \neg recebida_{ij}(x, 0, t- \\ & \phi_{ij}(x,t); \\ \delta_{fast_{ij}}(t), & \text{caso contrário.} \end{cases} \tag{2.7}$$

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 2.8 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.

$$\begin{aligned} motivação_{ij}(0) &= 0 \\ motivação_{ij}(t) &= (motivação_{ij}(t-dt) + impaciência_{ij}(t)) \\ &\quad \times aplicável_{ij}(t) \times inibida_{ij}(t) \\ &\quad \times reiniciada_{ij}(t) \times aquiescente_{ij}(t). \end{aligned} \tag{2.8}$$

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 2.9:

$$ativa_{ij}(t) = motivação_{ij}(t) \geq \theta \tag{2.9}$$

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.

$$\overline{\Delta t}_{ativação_{ij}} = \frac{\theta - motivação_{ij}(t)}{impaciência_{ij}(t)\rho_i} \tag{2.10}$$

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 2.10 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 2.9 e 2.10 podem ser rescritas como as Equações 2.11 e 2.12, respectivamente.

$$ativa_{ij}(t) = motivação_{ij}(t) == 1 \tag{2.11}$$

$$\overline{\Delta t}_{ativação_{ij}} = \frac{1 - motivação_{ij}(t)}{impaciência_{ij}(t)\rho_i} \tag{2.12}$$

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

2.4.1.2 L-ALLIANCE

O L-ALLIANCE (*Learning* 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).

Esta arquitetura faz duas suposições: (1) a média de desempenho na execução de uma tarefa específica nas últimas tentativas de um robô é um indicador razoável do seu desempenho esperado no futuro e (2) se um robô r_i está monitorando um conjunto de condições do ambiente C para avaliar o desempenho de outro robô r_k e as condições C mudam, então as mudanças são atribuíveis ao robô r_k.

O L-ALLIANCE estende a arquitetura ALLIANCE incorporando o uso de monitores de desempenho em cada comportamento motivacional dentro de cada robô. Cada monitor é responsável por observar, avaliar e catalogar o desempenho de qualquer robô do sistema (PARKER, 1996) sempre que este desempenha a tarefa correspondente à respectiva tarefa da configuração de comportamento do monitor em questão.

O L-ALLIANCE possui duas fases de controle na camada de alto nível de abstração: aprendizado ativo e aprendizado adaptativo. Durante a missão de treinamento, os robôs entram na fase de aprendizado ativo, enquanto quando eles estão realizando a missão ao vivo, eles estão na fase de aprendizado adaptativo.

2.4.1.3 BLE

A arquitetura BLE, acrônimo para *Broadcast Local Eligibility*, procura o par robô-tarefa (i, j), entre os robôs disponíveis e as tarefas a serem alocadas, que possui maior utilidade e aloca a tarefa j para o robô i enquanto existe robô disponível (WERGER; MATARIĆ, 2000).

O mecanismo utilizado pela arquitetura BLE envolve comparação de elegibilidade determinada localmente para uma dada tarefa com a melhor elegibilidade calculada por um par de comportamentos em todos os outros robôs que reivindicam a tarefa. Quando uma elegibilidade local de um robô é a melhor para algum comportamento, este inibe o par de comportamentos de todos os outros robôs que reivindicam a tarefa. Caso aconteça uma falha com o robô ou na tarefa, a falta de inibição resultante fará com que um outro robô tenha oportunidade de assumir o controle da tarefa.

A comunicação no BLE ocorre por difusão. Isto permite que o sistema tenha escalabilidade. Logo, qualquer número de robôs que utilizam o protocolo definido pelo BLE pode ser adicionado ao sistema e interagir apropriadamente.

2.4.1.4 MONAD

MONAD é uma arquitetura flexível para programação e execução orientada a um time de robôs (VU *et al.*, 2003). Esta arquitetura baseada em comportamentos integra controle comportamental hierárquico, mecanismos de coordenação multiagente e serviços de alocação de tarefas para os agentes. Esta arquitetura possui componentes que são configurados e programados através de um arquivo *script* de descrição do time e do programa do time e os códigos de execução de arbitragem e de comportamento. MONAD possui um sistema de coordenação distribuída chamado SCORE, do inglês *Synchronized CoORdination Engine*. A partir das arquivos de configuração e programação do time, este sistema cuida da coordenação do time em tempo de execução. Esta arquitetura foi testada através de uma aplicação de "Captura à Bandeira" no ambiente simulado Gamelotes (ADOBBATI *et al.*, 2015).

2.4.2 Arquiteturas baseadas em negociação

As arquiteturas baseadas em negociação são inspiradas em modelos econômicos para a coordenação das atividades dos robôs no sistema. De acordo com (ZLOT; STENTZ,

2006), o processo de negociação consiste em fazer com que os robôs do sistema procurem otimizar a função objetivo a partir de suas utilidades para desempenhar tarefas específicas.

Muitas arquiteturas baseadas em regras de negociação são variações do Protocolo 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. Para isso, este protocolo define quatro estágios:

- **Anúncio**: um agente assume o papel de coordenador (leiloeiro) e anuncia as tarefas ou o conjunto de tarefas que estão disponíveis para licitação;
- **Submissão**: após o cálculo dos valores de utilidade, cada agente (licitante) interessado comunica esse valor ao agente coordenador;
- **Seleção**: após o recebimento de todos os lances dos licitantes, o leiloeiro avalia os lances recebidos conforme uma estratégia de otimização para determinar qual será o agente ganhador;
- **Contrato**: o agente ganhador é atribuído através de um contrato para executar a tarefa.

Sempre que uma nova tarefa chega ao sistema, este processo acontece novamente. A Tabela 1 faz uma comparação entre três abordagens clássicas do CNP para negociação, as quais são:

- **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);
- **baseada em regras de leilão**: um leilão é um processo para a atribuição de um conjunto de bens ou serviços para um conjunto de licitantes de acordo com seus lances e os critérios do leilão (KHAMIS; HUSSEIN; ELMOGY, 2015). Conforme Gerkey e Mataric (2002), esta abordagem é uma excelente escolha para a alocação de tarefas em sistemas onde os recursos são escassos;
- e, por fim, **baseada em regras de comércio**: é composta 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).

Observa-se que todas as três abordagens utilizam o mesmo algoritmo para alocação de tarefas e possuem a mesma complexidade computacional e de computação. Arquiteturas baseadas em mercado consideram os benefícios de uma tarefa para o robô no cálculo

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	*publish / subscribe*	*publish / subscribe*	*apply / allocate*
Algoritmo de alocação de tarefa	algoritmo guloso	algoritmo guloso	algoritmo guloso
Habilidade 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	$\mathcal{O}(1)$/licitante, $\mathcal{O}(n)$/leiloeiro	$\mathcal{O}(1)$/licitante, $\mathcal{O}(n)$/leiloeiro	$\mathcal{O}(1)$/comprador, $\mathcal{O}(n)$/vendedor
Complexidade de computação	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

da utilidade enquanto as demais abordagens não levam isso em consideração. Arquiteturas baseadas em regras de leilão não permitem a reatribuição de tarefas já leiloadas, entretanto as outras duas abordagens permitem a realocação de tarefas. Abordagens baseadas em comércio possuem um modelo de comunicação durante negociação diferente das demais, além de permitir que seus agentes tenham a habilidade de tratar múltiplas tarefas simultaneamente. Por fim, a determinação da função dos robôs em arquiteturas baseadas em regra de comércio são negociadas enquanto que, nas arquiteturas de mercado e de leilão, os robôs se voluntariam para as funções disponíveis.

Outro fato interessante a respeito das arquiteturas de negociação é o fato que elas combinam a organização centralizada com a distribuída. Como consequência, elas apresentam uma combinação das vantagens e desvantagens de ambas organizações. Suas principais vantagens são:

- **eficiência**: a partir da utilização de informações locais e da preferência dos robôs, é possível alcançar soluções eficientes mesmo não tendo recursos;
- **robustez**: não existe um único agente responsável por analisar todos os dados do sistema e comandá-lo. Logo, o sistema é capaz de se adaptar para cobrir eventuais

falhas dos robôs;

- **escalabilidade**: mesmo que o número de entradas no sistema aumente, estas abordagens ainda fornecem uma solução eficiente;
- e **entrada *online***: novas tarefas podem ser introduzidas ao sistema a qualquer momento, ou seja, os robôs não conhecem as tarefas e nem quando elas chegam no sistema.

O mecanismo mais comum em arquiteturas de negociação é o leilão (ZLOT; STENTZ, 2006). Um conjunto de itens são oferecidos pelo leiloeiro durante a fase de anúncio do leilão. Cada participante pode fazer uma ofertar por estes itens ao submeter lances para o leiloeiro. Uma vez que todos os lances foram submetidos ou que o prazo predefinido tenha passado, o leiloeiro avalia os lances ofertados e determina quais itens serão premiados e para quem.

Dias *et al.* (2006) enfatizam que, em aplicações de robótica, os itens leiloados são tipicamente tarefas, papéis ou recursos. De modo que, o preço de um lance reflete os custos ou a utilidade de um robô que estão associados(a) com a completude de um tarefa, o cumprimento de um papel ou a utilização de um recurso.

O tipo mais simples de leilão é o aquele em que apenas um item é ofertado. Neste caso, cada participante (licitante) oferta um lance e aquele que ofertar o maior lance é premiado com o item pelo leiloeiro. Alternativamente, o leiloeiro retem o item caso nenhum lance cubra seu preço (chamado de preço de reserva).

Leilões do tipo lance selado e de protesto aberto se diferenciam na fase de submissão dos lances. Por um lado, em leilões de lance selado, apenas o leiloeiro tem conhecimento dos preço dos lances de cada licitante. Por outro lado, em leilão de grito aberto, os licitantes têm o benefício de ouvir os outros lances conforme cada um deles é ofertado.

Em leilões de primeiro preço, o preço de venda é o mesmo preço do lance ofertado pelo ganhador; enquanto que, em leilões de Vickrey, esse é o mesmo valor que o preço do segundo maior lance. O propósito desta última abordagem é de motivar os participantes a ofertar lances verdadeiros.

Outra abordagem deste mecanismo são leilões combinatoriais, onde múltiplos itens são leiloados pelo mesmo leiloeiro. Cada participante pode dar lance em cada combinação de subconjuntos dos itens ofertados. Dias *et al.* (2006) apontam que, em geral, o número de subconjuntos a serem considerados é muito grande. Deste modo, a avaliação dos lances, a comunicação e o fechamento do leilão tornam-se intratáveis, pois estes são problemas de ordem exponencial.

Várias arquiteturas baseadas em negociação foram propostas na literatura. É desejável entendê-las para verificar que características e mecanismos aparecem frequentemente

nessas arquiteturas. Assim, serão revisadas as principais arquiteturas baseadas em negociação encontradas na literatura.

Além de pesquisas realizadas nas principais bases de artigos científicos, essas arquiteturas também foram encontradas a partir dos principais artigos *surveys* e de taxonomia sobre alocação de tarefa em sistema multirrobô: (DUDEK *et al.*, 1996), (GERKEY; MATARIĆ, 2004), (DIAS *et al.*, 2006), (KORSAH; STENTZ; DIAS, 2013) (YAN; JOUANDEAU; CHERIF, 2013), (KHAMIS; HUSSEIN; ELMOGY, 2015) e (NUNES *et al.*, 2017). A seguir, as principais arquiteturas de alocação de tarefa em sistema multirrobô baseadas em modelos de negociação são revisadas.

2.4.2.1 DRA

A arquitetura Dynamic Role Assignment (DRA) foi proposta por Chaimowicz, Campos e Kumar (2002). O mecanismo utilizado nesta arquitetura permite coordenação em um sistema multirrobô durante a execução de tarefas cooperativas. Cada papel define um controlador de robô e uma atribuição de papel permite que os robôs mudem seus comportamentos dinamicamente durante a execução de uma tarefa. Este trabalho mostra que os robôs podem desempenhar as tarefas mais eficientemente se o sistema de coordenação permitir atribuição dinâmica e troca de papéis entre os agentes. Pois, assim, os robôs se adaptam aos eventos inesperados no ambiente, melhorando o desempenho individual deles em benefício do time.

A Figura 4 mostra os três tipos de atribuição de papéis do DRA:

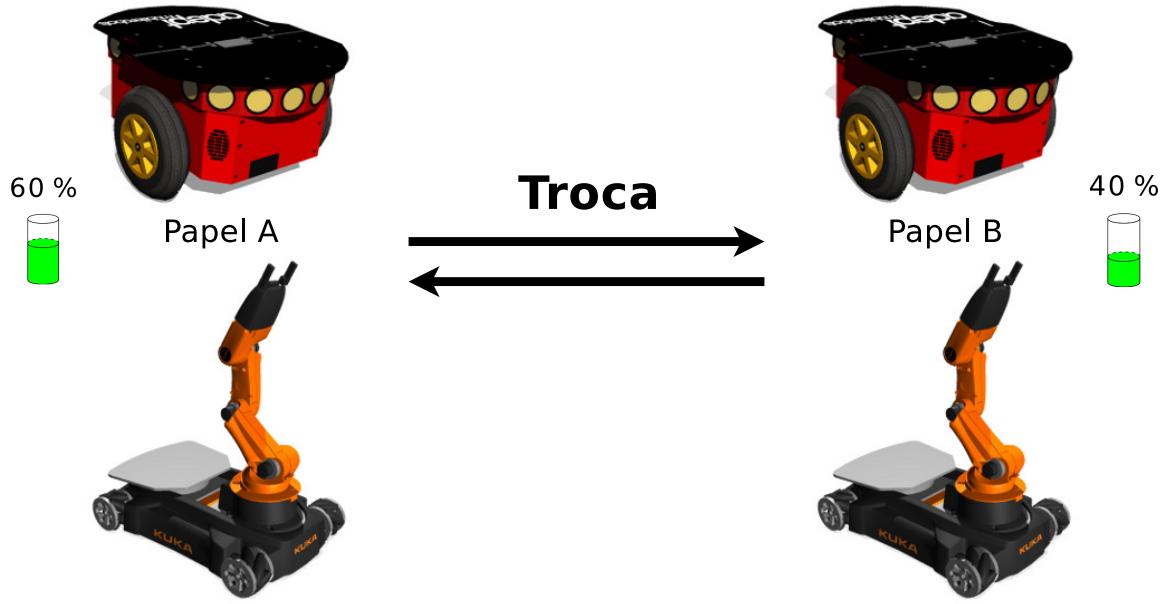
- **Alocação**: o robô assume um novo papel depois de terminar a execução de um outro papel, conforme mostra a Figura 4a;
- **Realocação**: o robô interrompe o desempenho de um papel para começar ou continuar a desempenhar outro papel, conforme mostra a Figura 4b;
- **Troca**: dois robôs se sincronizam para trocar seus papéis, de modo que cada um assume o papel desempenhado pelo outro.

2.4.2.2 M+

O M+ foi a primeira variação do CNP para a alocação e realização de tarefas em sistemas multirrobô. Esta arquitetura 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).

(a) Alocação de Papel

(b) Realocação de Papel



(c) Troca de Papéis

Figura 4 – Tipo de atribuição de papel da arquitetura DRA.

O protocolo da arquitetura M+ possui três módulos: *alocação de tarefa*, *reação cooperativa* a contingências e *execução de tarefa*. O módulo de alocação de tarefa do M+ é responsável por refinar e atribuir tarefas. Este módulo utiliza mecanismos de negociação que permitem que um robô escolha iterativamente a melhor tarefa para executar levando em consideração o contexto atual do sistema. O módulo de reação cooperativo é invocado quando ocorre uma falha durante a execução de uma dada tarefa. Este módulo é responsável por atualizar o estado do ambiente, gerenciar a troca de informação entre os robôs e controlar o replanejamento e pedidos por ajuda. Enquanto que o módulo de execução de

tarefa do M+ é responsável por controlar a execução de tarefas, bem como, sincronizar as tarefas e ações dos robôs.

Quando ocorre alguma falha no M+, primeiramente, o robô tenta encontrar outro conjunto de ações que atinja o objetivo; senão, é solicitado ajuda dos demais robôs. Quando requisitado, o robô tenta resolver o problema do colega em conjunto ao seu.

2.4.2.3 Murdoch

Murdoch é uma variante do CNP e foi a primeira aplicação de métodos de leilão para a coordenação de sistemas multirrobôs físico aplicado a várias tarefas. 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). Esta arquitetura se baseia em um modelo de comunicação *publish/subscribe* centrado em recursos que faz uso intenso e eficiente de comunicação explícita entre os robôs.

Seu objetivo não é explorar recursos de maneira ótima, mas sim investigar métodos práticos e eficientes para a alocação de tarefas para grupos de robôs autônomos heterogêneos. Para isso, a arquitetura Murdoch visa minimizar três aspectos ao alocar tarefas: (1) uso dos recursos; (2) tempo de conclusão das tarefas e (3) sobrecarga da comunicação. Lembrando que tais critérios não são ortogonais, isto é, eles influenciam fortemente um no outro.

Suas premissas são:

- o sistema é composto de robôs que possuem estrutura física;
- os robôs são heterogêneos, pois possuem capacidades distintas;
- os robôs podem se comunicar, mas as mensagens podem ser perdidas;
- os robôs cooperam honestamente;
- os robôs (ou parte deles) podem falhar a qualquer momento;
- um robô pode não estar ciente sobre a causa da sua falha;
- os robôs são multiúso e não específicos para uma tarefa;
- a sequência em que as tarefas chegam ao sistema não é conhecida;
- e se um robô é capaz de realizar uma dada tarefa, este também é capaz de determinar seu próprio progresso e a completude desta tarefa.

Em sua abordagem, Gerkey e Mataric (2002) definem um *modelo de comunicação*, uma *representação de tarefa* e um *modelo de leilão* para a alocação de tarefa. O modelo de

comunicação proposto determina que os robôs do sistema devem se comunicar enviando mensagens anonimamente em difusão. Deste modo, uma mensagem é enviada uma única vez e será recebida por todos através da rede. Além disso, tal esquema permite que os robôs possam entrar e sair dentro do raio de comunicação, que a rede caia temporariamente e, também, que os robôs experimentem diferentes tipos de falha. Também é proposto uma estrutura hierárquica para a representação de tarefa através de uma árvore. De modo que um nó raiz de uma tarefa específica pode ser um nó intermediário em outra árvore de tarefa. Uma relação pai-filho entre nós nesta árvore significa que a tarefa pai é responsável por alocar e monitorar sua tarefa filha. Enfim, o modelo de negociação usado (leilão) permite tratar incertezas com escalabilidade durante a atribuição de tarefas e recursos no sistema.

Murdoch é uma arquitetura distribuída apesar do leiloeiro centralizar as informações do leilão, pois qualquer robô do sistema pode gerenciar um leilão. O envio das mensagens nesta arquitetura é endereçado pelo seu conteúdo ao invés do seu destino. Isto produz uma associação anônima entre produtores e consumidores de dados. De um lado, um produtor de dado simplesmente classifica uma mensagem de acordo com o seu assunto e a publica na rede. Do outro lado, todo consumidor que registrou interesse neste assunto irá receber automaticamente a mensagem. Tal modelo de comunicação é chamado *publish/subscribe*. Os nomes dos assuntos das mensagens respeitam uma semântica. Como o Murdoch aloca tarefas em um grupo de robôs heterogêneos com potencial, a semântica utilizada para representar os assuntos das tarefas corresponde ao conjunto de recursos utilizados durante sua execução.

O protocolo de leilão distribuído utilizado pelo Murdoch (em inglês *first-price one-round auctions* possui os cinco seguintes passos:

- **Anúncio da tarefa**: um agente (trabalhando em favor de um usuário, alarme ou tarefa) atua como leiloeiro (*auctioneer*) publicando um mensagem de anúncio (*announcement*) da tarefa gerada. Esta mensagem contem dados pertinentes sobre a execução da tarefa, a qual é direcionada ao conjunto de assuntos que representa os recursos exigidos para executá-la; logo, apenas os robôs que são capazes de executar a tarefa receberão a mensagem;

- **Avaliação de métricas**: como haverá mais de um robô capaz disponível para designar a tarefa, torna-se necessário um parâmetro para compará-los. Assim, cada robô candidato pode determinar sua aptidão (*fitness*) para a tarefa leiloada;

- **Submissão do lance**: após o cálculo de aptidão, cada robô candidato (*bidder*) publica sua pontuação de aptidão como uma mensagem de lance (*bid*);

- **Encerramento do leilão**: alguns instantes após o anúncio do leilão, o leiloeiro

processa todos os lances recebidos até o dado momento. Todos os candidatos serão notificados do encerramento do leilão. O candidato que possuir maior pontuação de aptidão será o ganhador. Este é premiado com um contrato de tempo limitado para executar a tarefa, enquanto os demais retornam a espera por novas tarefas;

- **Monitoramento de progresso/Renovação do contrato**: enquanto o robô ganhador executa a tarefa, o leiloeiro monitora o progresso da tarefa. Sempre quando é verificado um progresso satisfatório, periodicamente, o leiloeiro envia uma mensagem de renovação (*renewal*) para o ganhador. Este responde com uma mensagem de confirmação (*acknowledgment*) até que a tarefa seja concluída.

Verifica-se, assim, que sempre que um contrato é firmado entre o leiloeiro e o robô ganhador, uma nova alocação de tarefa acontece no sistema. Se o contrato não for renovado, a tarefa poderá ser leiloada novamente. Isto significa que esta arquitetura permite a realocação de tarefas. Observa-se também que o Murdoch atua como um escalonador de tarefas instantâneo guloso. Com isso, dependendo da ordem em que as tarefas surgem no sistema, sua solução pode se encontrar bem distante da solução ótima.

2.4.2.4 ASyMTre

O ASyMTre (do inglês, *Automated Synthesis of Multi-robot Task solutions through software Reconfiguration*) é uma arquitetura centralizada que utiliza um protocolo de negociação para sintetizar soluções de tarefas de acordo com os requisitos das tarefas e a composição do time. Seu objetivo é aumentar as capacidades de solução de tarefa de times multirrobô heterogêneos trocando a abstração fundamental de uma abstração típica de "tarefa" para uma abstração de "esquema" e automaticamente reconfigurar os esquemas para tratar a tarefa em mãos.

A arquitetura ASyMTre possui algumas extensões:

- **ASyMTre-D**: proposta por Tang e Parker (2005), esta é uma versão distribuída da arquitetura ASyMTre baseada no CNP cujo propósito não é melhorar a versão original, mas dar capacidade de solução de tarefa autônoma para os robôs de forma distribuída;
- **IQ-ASyMTre**: proposta por Zhang e Parker (2010), esta extensão utiliza medidas de qualidade de informação (IQ) para guiar as coalizões de robôs para satisfazer um as limitações sensoriais enquanto as tarefas são executadas.

2.4.2.5 TraderBots

TraderBots é uma abordagem baseada em regras de comércio para a coordenação de múltiplos robôs que primeiramente foi concebida por Stentz e Dias (1999), o primeiro

trabalho a considerar uma abordagem baseada em regras de mercado para arquiteturas MRTA. Posteriormente, alguns avanços foram adquiridos em (DIAS; STENTZ, 2000) e, finalmente, ela foi aprimorada em (DIAS, 2004). Todos estes trabalhos utilizam uma extensão do CNP projetada por Sandholm, Lesser *et al.* (1995) e os conceitos gerais de agentes com conhecimentos de mercado de Wellman e Wurman (1998). Esta arquitetura é inerentemente distribuída, porém ela também forma sub-grupos centralizados para aumentar a eficiência do sistema.

Seja um time de robôs projetados para desempenhar um conjunto de tarefas em um ambiente dinâmico de modo que este time seja modelado como uma economia onde cada robô é um comerciante egoísta dentro dela. O objetivo do time é completar as tarefas eficazmente enquanto o *lucro* (que é dado pela *receita* menos *custo*) geral do sistema é maximizado ao mesmo tempo que cada robô tenta maximizar seu lucro individual. Para isso, os robôs irão realizar leilões e dar seus lances para que aconteçam atribuições das tarefas para os integrantes do time. Nesta economia, as diferentes tarefas e informações do sistema são as mercadorias negociadas (DIAS, 2004).

Os custos e as receitas determinam em grande medida o desempenho de abordagens baseadas em negociação. Para isso, são necessárias duas funções: (1) uma, $f_{receita}$, que mapeia possíveis resultados de tarefas em valores de receita e (2) outra, f_{custo} que mapeia esquemas possíveis para executar a tarefa em valores de custo. A partir delas é possível determinar o lucro esperado ao executar um dado plano $\mathbf{P}$: $f_{lucro}(\mathbf{P}) = f_{receita}(\mathbf{P}) - f_{custo}(\mathbf{P})$. As funções de receita e custo devem ser projetadas para refletir a natureza do domínio da aplicação. Logo, deve-se levar em conta as tarefas que possuem maior prioridade, prazos apertados, margens de erro aceitáveis, etc. Dado que o lucro total do sistema é a soma do lucro de cada robô, se cada robô atua estritamente em favor dos seus próprios interesses, robôs individuais não maximizam apenas seu próprio lucro mas também o lucro do time como um todo.

A execução de tarefas não é a única forma que os robôs têm para obter renda. Isto é, um robô também pode receber receita de outro robô em troca de bens e serviços (DIAS, 2004). Se um robô, por exemplo, não possui todos recursos e capacidades para completar uma tarefa, este pode contratar serviços de outros robôs do time que os possuem. Logo, dois robôs são incentivados a trabalhar cooperativamente se eles produzem mais lucro agregado juntos do que separados.

O montante de pagamento para um bem ou serviço é determinado por um *preço*. Dados:

- um robô r_1 gostaria de contratar um serviço do robô r_2;
- r_2 tem um custo Y para desempenhar tal serviço;

- e r_1 pode obter um adicional de X em sua receita se r_2 desempenhá-lo.

Se X for maior que Y $(X > Y)$, então, ambos robôs tem um incentivo para fechar um acordo. Uma das abordagens para negociar o preço de um bem ou serviço é a *licitação*. Este processo acontece até que um preço mutualmente aceitável seja encontrado. Por exemplo, r_1 poderia começar dando uma oferta de preço igual à Y, isto é, r_1 recebe todo o lucro. Em seguida, r_2 poderia recusar a oferta de r_1 e dar um lance de X, ou seja, r_2 recebe todo o lucro. E, assim, sucessivamente. Em outras palavras, a ideia geral é começar oferecendo um preço que é pessoalmente mais favorável e, paulatinamente, se retratar até que exista uma oferta mutuamente favorável.

Um dado robô pode realizar várias negociações com potencial simultaneamente. Inicialmente, este oferece o preço mais favorável para si para todas negociações. Gradativamente, este robô se retrata para um valor mais baixo. Assim que a primeira negociação satisfazer ambos, termina-se o processo de licitação. Uma negociação pode envolver múltiplas partes, requerendo que todas elas entrem em acordo antes que a negociação seja firmada.

Dias (2004) ressalta que o preço negociado de um bem/serviço que possui alta demanda ou baixa oferta será alta. A partir desta informação, vários outros fornecedores entrarão na concorrência, fazendo com que o preço deste bem/serviço abaixe. Caso contrário, se a demanda é baixa ou a oferta for alta, o baixo preço do serviço levará os fornecedores para uma outra linha de negócios, elevando o seu preço. Logo, observa-se que em um comércio eficiente em que as informações são perfeitas, o preço de cada bem e serviço será otimizado conforme sua oferta e demanda.

Portanto, a interação entre os robôs pode ser *cooperativa* ou *competitiva* (DIAS, 2004). De um lado, a interação cooperativa, recorrente entre robôs heterogêneos, acontece quando seus papéis são complementares, ou seja, quando os robôs envolvidos adquirem maior lucro trabalhando coletiva do que individualmente. Por outro lado, a interação competitiva, recorrente entre robôs homogêneos, ocorre quando os robôs envolvidos possuem o mesmo papel, isto é, se o lucro montante de um robô é afetado negativamente na presença de outro robô. Contudo, robôs heterogêneos podem competir se uma dada tarefa pode ser executada em diferentes maneiras. Assim como, robôs homogêneos podem cooperar se uma tarefa específica exige mais de um robô na sua execução.

A organização intencional de um sub-grupo de robôs para executar um conjunto de tarefas em um sistema multirrobô é chamada *formação de coalizão*. Na arquitetura (DIAS, 2004), os robôs têm a capacidade de se organizarem sozinhos para a formação de coalizão. Se um robô oferece seus serviços de liderança para um sub-grupo de robôs do time, este não tornará seu líder por coerção ou decreto. Este robô deve convencer o grupo de que eles farão mais dinheiro juntos se seguirem os planos dele do que se eles

atuassem individualmente ou em sub-grupos. Se ele possui realmente um bom plano, ele irá maximizar o lucro de todos os integrantes do grupo. Assim, o prospectivo líder pode usar este lucro grande para licitar os serviços dos membros do grupo e, logicamente, ele pode reter uma porção do lucro para si. Neste caso, todos os robôs relevantes devem submeter o plano para que este possa ser vendido. O líder pode oferecer ofertas não só contra os planos dos indivíduos, mas também contra planos de grupo produzidos por outros líderes em potencial. Por fim, um líder pode atuar simultaneamente como um agente benevolente e egoísta, pois ele receberá compensações pessoais pelos esforços ao beneficiar o grupo todo.

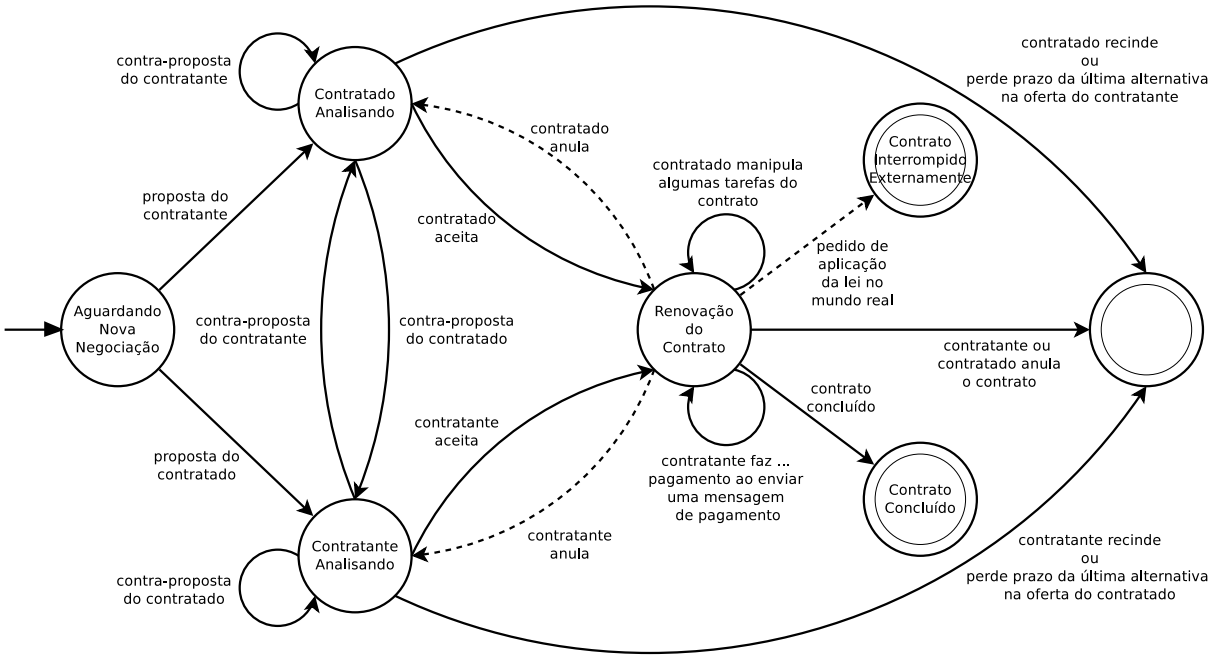


Figura 5 – Extensão do CNP sugerida por Sandholm, Lesser *et al.* (1995).

A Figura 5 apresenta o esquema de negociação proposto por Sandholm, Lesser *et al.* (1995) que é utilizado pela arquitetura TraderBots. Este esquema é controlado por uma máquina de estados que evolui mediante às ações dos agentes envolvidos, os quais podem assumir os papéis contratado e contratante. O contratante é o agente que possui interesse em algum produto ou serviço oferecido pelo contratado.

3 *Framework TAlMech*

Alguns mecanismos e características são recorrentes entre as arquiteturas de alocação de tarefa para sistemas de múltiplos robôs (MRTA). Assim, com o objetivo de facilitar o desenvolvimento de aplicações que envolvam MRTA, o *framework TAlMech* (acrônimo para *Task Allocation Mechanisms*) foi desenvolvido neste trabalho. Este *framework* agrupa e encapsula mecanismos que aparecem frequentemente nas arquiteturas de alocação de tarefa em sistemas com múltiplos robôs.

O *TAlMech* foi desenvolvido para ser utilizado em conjunto com o *framework* e *middleware* ROS. Logo, considera-se que toda a comunicação entre os agentes do sistema é realizada através dos recursos de *middleware* do ROS por meio de uma rede de comunicação. O Apêndice A revisa os conceitos necessários sobre o ROS para compreender o *framework* proposto. A linguagem de programação utilizada na implementação do *TAlMech* é o C++. Esta linguagem foi escolhida por permitir a utilização do paradigma de programação orientada a objetos e ser eficiente em tempo de execução. Além disso, a biblioteca C++ do ROS é a biblioteca mais utilizado por clientes do ROS e foi projetada para ser a biblioteca que possui maior desempenho (QUIGLEY *et al.*, 2009).

Todo o projeto do *framework TAlMech* foi versionado utilizando a ferramenta *git* e disponibilizado em um repositório[1] da plataforma GitHub para a contribuição e utilização da comunidade de robótica.

O Apêndice B define as mensagens que foram criadas para serem utilizadas pelo *framework TAlMech* para haver comunicação no ROS entre os agentes.

De modo geral, o *framework TAlMech* foi desenvolvido para facilitar sua manutenção, evolução, extensão e utilização.

3.1 Padrões de Projeto

Durante o desenvolvimento do *framework TAlMech* foram utilizados alguns dos padrões de projeto de Gamma *et al.* (1993). Tais padrões facilitam a comunicação entre os desenvolvedores de *software*, simplifica a documentação, facilita a manutenção do código, garante robustez e, em alguns casos, flexibilidade.

Dentre os padrões propostos por Gamma *et al.* (1993), foram utilizados nesse projeto:

- **Objeto Único** (*Singleton*):

[1] Repositório do *framework TAlMech*: <https://github.com/adrianohrl/TAlMech>.

  - Descrição:

    Garante que uma classe só tenha uma única instância e provê um ponto de acesso global a ela.

  - Consequências:
    * acesso central e extensível a recursos e objetos;
    * pode ter subclasses.
  - Aplicabilidade:
    * quando é preciso que apenas um objeto exista, independentemente do número de requisições para criá-lo.

- **Ponte** (*Bridge*):
  - Descrição:

    Desacopla uma abstração de sua implementação para que os dois possam variar independentemente.

  - Consequências:
    * os detalhes de implementação são totalmente inacessíveis aos clientes;
    * elimina-se dependências em tempo de compilação das implementações;
    * e a implementação pode ser configurada em tempo de execução.
  - Aplicabilidade:
    * quando for necessário evitar uma ligação permanente entre a interface e a implementação;
    * quando alterações na implementação não puderem afetar clientes;
    * quando tanto abstrações como implementações precisarem ser capazes de suportar extensão através de herança;
    * quando implementações são compartilhadas entre objetos desconhecidos do cliente.

- **Estado** (*State*):
  - Descrição:

    Permite a um objeto alterar o seu comportamento quanto o seu estado interno mudar. O objeto irá aparentar mudar de classe.

  - Consequências:
    * aparente mudança de classe quando o objeto muda internamente seu estado;
    * separa os diferentes estados em diferentes classes;

    - torna explícita a transição entre estados.
  - Aplicabilidade:
    - quando o comportamento de um objeto depender de um estado.

- **Decorador** (*Decorator*):
  - Descrição:

    Anexa responsabilidades adicionais a um objeto dinamicamente. *Decorators* oferecem uma alternativa flexível ao uso de herança para estender uma funcionalidade.
  - Consequências:
    - aumenta a legibilidade e a evolução do *software*;
    - decoradores podem ser retirados e acrescentados em execução;
    - não há necessidade de criar inúmeras subclasses.
  - Aplicabilidade:
    - quando se deseja expandir o conjunto de funcionalidades de um objeto em versões futuras;
    - quando é desejada a adição de funcionalidades dinamicamente a um objeto.

- **Modelo de Método** (*Template Method*):
  - Descrição:

    Define o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses. *Template Method* permite que suas subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura.
  - Consequências:
    - cria uma linguagem universal para os usuários da classe;
    - simplifica a implementação de variantes do método;
    - garante a chamada correta dos passos do algoritmo;
    - remove substancialmente a duplicação de código.
  - Aplicabilidade:
    - quando há repetição de código;
    - quando for necessário definir os passos exatos de um algoritmo (método) onde alguns desses passos, mais específicos, são implementados por subclasses.

- **Estratégia** (*Strategy*):

  - Descrição:

    Permite que algoritmos mudem independentemente dos clientes que os utilizam.

  - Consequências:
    * uma hierarquia de classes define uma família de algoritmos. Partes comuns podem ser colocadas nas superclasses;
    * algoritmos usados podem variar em execução.
  - Aplicabilidade:
    * quando se deseja permitir que o cliente implemente novas estratégias.

- **Fábrica Abstrata** (*Abstract Factory*):
  - Descrição:

    Provê uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

  - Consequências:
    * permite a troca de toda uma coleção de classes sem grandes modificações no programa fonte;
    * as classes concretas são conhecidas apenas pela fábrica abstrata;
    * é difícil colocar classes novas no programa;
    * as classes dos produtos devem ter uma interface comum.
  - Aplicabilidade:
    * quando o programa deve ser configurado por famílias de classes;
    * quando todas as classes de uma mesma família devem ser utilizadas em conjunto.

- **Fachada** (*Façade*):
  - Descrição:

    Oferece uma interface única para um conjunto de interfaces de um subsistema. *Façade* define uma interface de nível mais elevado que torna o subsistema mais fácil de usar.

  - Consequências:
    * o subsistema é mais fácil de usar;
    * o subsistema é mais independente dos clientes, pois não existe uma comunicação entre eles.

- Aplicabilidade:
  - quando clientes de um subsistema forem utilizar apenas parte das funcionalidades dele;
  - quando há muitas dependências entre dois subsistemas, dificultando a manutenção.

3.2 Classes Básicas

O *framework TAlMech* agrupa, em um mesmo *namespace*[2], um conjunto de classes utilitárias e comuns aos diferentes mecanismos implementados no *framework*. A Figura 6 mostra o diagrama UML (acrônimo para *Unified Modeling Language*) que define e relaciona as classes básicas do *framework TAlMech*.

3.2.1 Classes Utilitárias

O grupo de classes utilitárias do *framework TAlMech* é composto pelas classes: *Exception*, *Comparator*, *ToMsg*, *Controller*, *MachineController* e *MachineState*.

A classe *Exception* é utilizada para tratar exceções que podem acontecer em tempo de execução através do mal uso do *framework*. Sempre que um erro de consistência ocorre, é lançada uma nova exceção através de um objeto *Exception* contendo a descrição de sua causa. Quando isso acontece, também é registrado no ROS um novo *log* com severidade *FATAL*.

A interface genérica *Comparator* especifica a assinatura do método de comparação entre itens de um mesmo tipo. Este método é utilizado para verificar se um dado par de objetos de uma coleção obedece à regra de ordenação especificada pela classe.

A interface genérica *ToMsg* especifica os métodos que devem ser implementados por classes que armazenam dados de uma mensagem específica do ROS.

A classe abstrata *Controller* define sucintamente os métodos comuns entre os controladores que podem ser utilizados no *framework*. Classes que herdam dessa classe devem implementar o método que é chamado periodicamente para o processamento do controle desejado. Esta classe abstrata auxilia ainda na identificação da efetividade do controlador. Se desejado, pode-se utilizar tal recurso para a liberação de memória.

A classe *MachineController* em conjunto com a classe *MachineState* encapsulam o controle de uma máquina de estados para o uso do padrão de projeto *State*. Ambas classes são abstratas, de modo que, ao especializá-las, devem ser identificados os estados da máquina e definidas as funções de transição entre eles. A classe *MachineController*,

[2] *Namespace* é um recurso disponível em algumas linguagens de programação (do C++ inclusive) que fornece uma forma de evitar conflitos de nomes em grandes projetos.

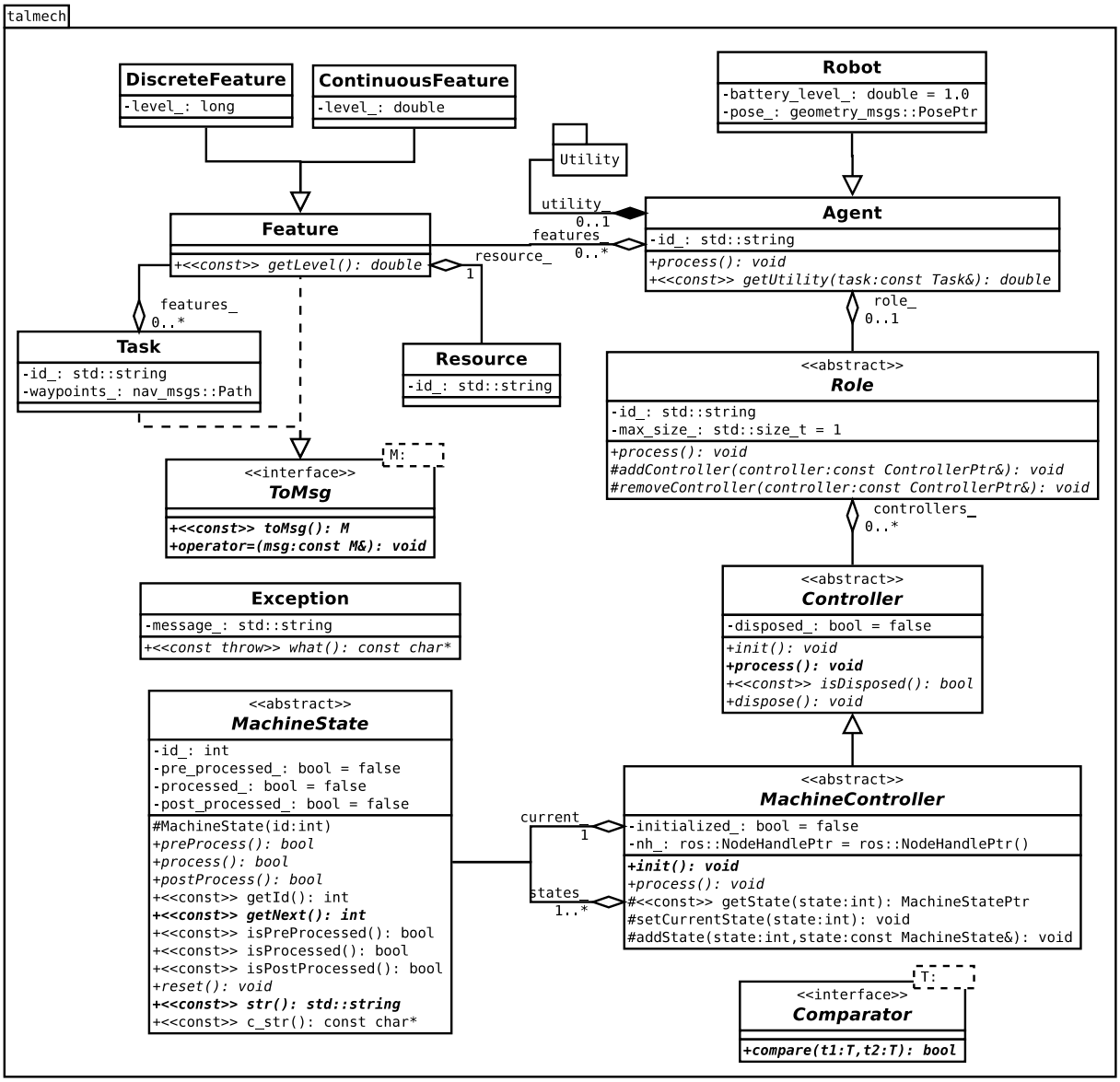


Figura 6 – Diagrama UML das classes básicas do *framework TAlMech*.

uma especialização da classe *Controller*, mapeia todas as instâncias de estados da máquina conforme suas identificações e controla a transição entre eles. Mas, para isso, é necessário identificar e instanciar cada estado desejado, bem como, definir o estado inicial da máquina durante a sua inicialização. A cada chamada de processamento da máquina de estados, o controlador pré-processa, processa ou pós-processa o estado corrente. Após o pós-processamento do estado corrente, é solicitado a identificação do próximo estado da máquina. Logo em seguida, o controlador configura esse estado como corrente. Este processo se repete continuamente até que o controlador deixe de ser efetivo.

3.2.2 Classes Comuns

As classes que não são específicas de algum mecanismo do *framework TAlMech* foram agrupas juntamente com as classes utilitárias dentro do *namespace talmech*. São

elas: *Resource*, *Feature*, *DiscreteFeature*, *ContinuousFeature*, *Task*, *Role*, *Agent* e *Robot*, conforme mostra a Figura 6.

A classe *Resource* é utilizada para classificar os recursos existentes no sistema. Cada recurso possui um identificador único representado por uma cadeia de caracteres. A classe *Feature* associa o grau de característica de um recurso particular. Objetos do tipo *Feature* são utilizados para representar o grau de característica requisitado durante a execução de uma tarefa, mas também para representar o grau de característica dos robôs e agentes do sistema com respeito a cada um dos seus recursos. Uma instância de *Feature* representa uma característica de grau unário, isto é, identificam a falta ou existência de um recurso. Assim, esta classe pode ser utilizada para representar situações como, por exemplo:

- quando se deseja especificar a utilização de uma câmera para executar uma tarefa que envolve processamento de imagens;
- quando a cinemática do robô é holonômica;
- ou quando a base de um manipulador robótico é fixa.

Contudo, a classe *Feature* pode ser especializada como *DiscreteFeature* e *ContinuousFeature*, as quais possuem um grau de característica discreto e contínuo, respectivamente. Desta forma, recursos como o número de processadores que um agente possui, por exemplo, devem ser representados por objetos do tipo *DiscreteFeature*. Por outro lado, recursos como o nível de bateria ou a capacidade de carga que um robô tem devem ser representados através de objetos do tipo *ContinuousFeature*. Logo, tanto recursos unários, quanto discretos e contínuos são considerados no *framework TAlMech*. Enfim, as classes *Feature*, *DiscreteFeature* e *ContinuousFeature* implementam a interface *ToMsg* para converter os dados armazenados por elas para uma mensagem do ROS do tipo *talmech_msgs/Feature*.

A classe *Task* reúne informações relevantes sobre a execução de uma tarefa. Cada tarefa possui um código identificador que deve ser único, um conjunto[3] de coordenadas cartesianas que descreve os pontos por onde o robô deverá passar durante sua execução e também uma coleção de recursos exigidos com seus respectivos graus de característica. Esta classe implementa a interface *ToMsg*, pois é possível gerar uma mensagem do ROS *talmech_msgs/Task* a partir de um objeto *Task*.

A classe *Role* é abstrata e deve ser herdada por classes que desejam implementar um comportamento ou um papel de um agente. Esta classe pode processar o controle de nenhum, um ou vários controladores do tipo *Controller*. A cada chamada de processamento de um objeto *Role*, é feita a chamada do processamento de cada controlador

[3] O conjunto de coordenadas cartesianas pode ser vazio.

do papel. Nota-se que esta é uma aplicação do padrão de projeto *Bridge*. Deste modo, é possível variar os tipos de implementação dos controladores independentemente do papel instanciado. Esta técnica garante flexibilidade na escolha dos tipos de controladores do papel instanciado, bem como, na expansão de novos tipos de controle para o papel. O controle do número de instâncias que um papel processa é controlado internamente na classe *Role*. Caso um controlador deixe de ser efetivo, este objeto é reciclado. Com isso, o número de controladores em processamento é decrementado.

A classe *Agent* armazena informações pertinentes sobre um agente do sistema. Cada agente possui um identificador que deve ser único e é utilizado nos mecanismos de alocação de tarefa para identificá-lo. Os agentes podem ter ou não um papel através da instância da classe *Role*. Caso este possua um papel, a cada chamada de processamento do agente, seu papel é processado. Novamente foi utilizado o padrão de projeto *Bridge* para se obter flexibilidade na escolha do papel do agente em tempo de execução e, ainda, possibilitar a expansão do *framework* para a implementação de novos papéis. Cada agente conta com uma coleção de objetos do tipo *Feature* que reúne o grau de característica para cada um dos seus recursos.

Finalmente, a classe *Robot* é uma especialização da classe *Agent* que representa os agentes do sistema que são robôs. Robôs são agentes que, além de se comunicar com outros agentes, interagem fisicamente com o ambiente, isto é, percebem o estado do sistema por meio dos seus sensores e que atuam no sistema através dos seus atuadores. Logo, robôs possuem mais características que um agente comum. Objetos do tipo *Robot* monitoram o nível de baterias e sua postura (posição e direção) dentro do ambiente. Os robôs podem representar seus sensores e atuadores através das classes *Feature*, *DiscreteFeature* e *ContinuousFeature*.

Agentes podem utilizar seus recursos no cálculo de utilidade para as tarefas de uma aplicação. Robôs, além de utilizar seus recursos (sensores, atuadores e outros), podem também utilizar o nível de bateria e a postura corrente para efetuar este cálculo. O cálculo de utilidade pode ser simples ou complexo, pode variar de um robô para outro e de uma aplicação robótica para outra. Pensando nisso, foi desenvolvido o *framework* de utilidade do *TAlMech*. A seguir, é detalhado o cálculo e a organização dos componentes de utilidade.

3.3 Cálculo de utilidade

O cálculo de utilidade é um conceito que não possui solução fechada em robótica. Seu valor pode ser subjetivo à um agente do sistema. Além disso, comparar utilidades em um sistema multirrobô heterogêneo pode ser um problema, pois as estruturas dos seus agentes são diferentes. Contudo, cada aplicação robótica possui uma métrica para avaliar a execução de uma tarefa para um dado robô.

Com o propósito de facilitar essa tarefa, foi desenvolvido um *framework* de utilidade dentro do *TAlMech*. Este *framework* decora o cálculo de utilidade com componentes modulares que realizam parte deste cálculo. Assim, cada componente contribui com uma parcela do cálculo de utilidade. Ao se requisitar o cálculo de utilidade para uma tarefa particular, esta é passada como parâmetro para cada componente. Deste modo, os componentes podem avaliar e comparar os dados da tarefa e as características do agente em questão conforme suas considerações. O nível de aptidão do agente para a tarefa dada é calculado a partir desta avaliação e comparação. Em seguida, este é somado à utilidade do agente para a tarefa. Ao final do cálculo de todos os componentes, é obtido o cálculo total de utilidade desta tarefa para o agente em questão. É importante que as unidades finais no cálculo de cada componente sejam as mesmas. Caso isso não aconteça, o valor de utilidade será inconsistente.

Para isso foi utilizado o padrão de projeto *Decorator* por meio das classes *UtilityComponent* e *UtilityDecorator*, conforme o diagrama de classes mostrado na Figura 7.

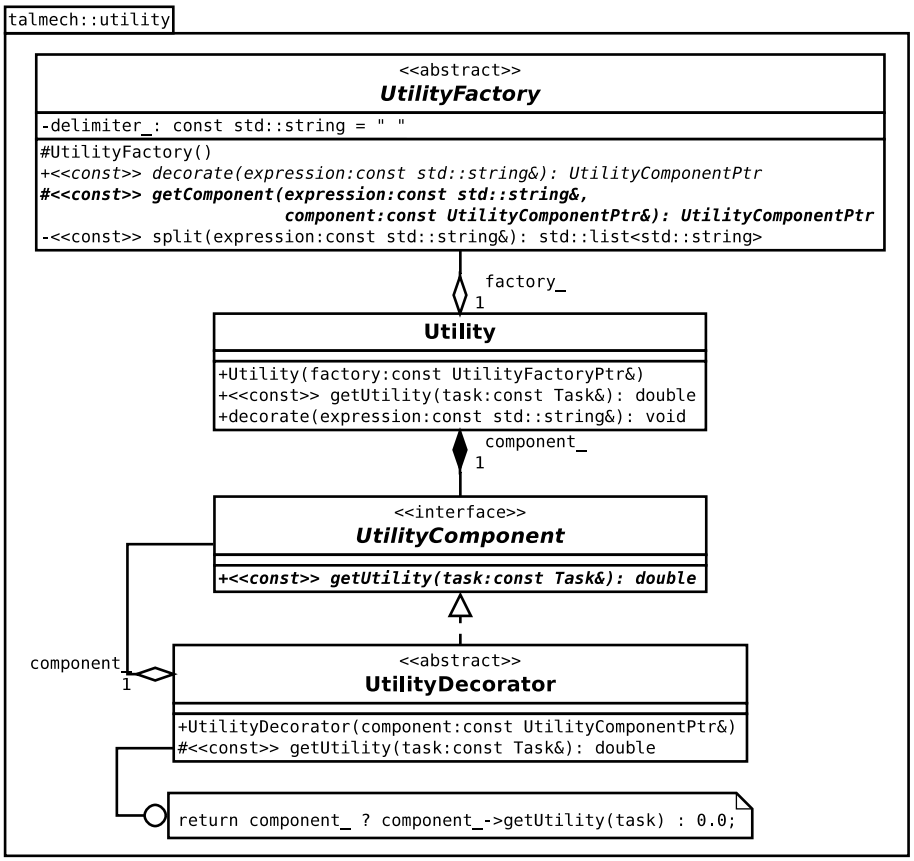


Figura 7 – Diagrama UML das classes do *framework TAlMech* para o cálculo de utilidade.

A interface *UtilityComponent* define a assinatura que os componentes de cálculo de utilidade devem implementar. O cálculo de utilidade é requisitado através da chamada deste método. A classe abstrata *UtilityDecorator* implementa a interface *UtilityComponent*. Portanto, objetos do seu tipo podem ser requisitados para fazer o cálculo de utilidade para uma dada tarefa. Esta classe possui um objeto do tipo *UtilityComponent* como seu

membro. A classe *UtilityDecorator* contem uma implementação do método para o cálculo de utilidade que só pode ser acessada através da herança. Isto se deve ao fato deste método apenas simplificar a chamada do cálculo de utilidade para a dada tarefa ao seu componente interno, caso este exista. Ou seja, esta implementação não contribui com uma parcela do cálculo de utilidade do robô para a tarefa. Logo, todas as classes que herdam de *UtilityDecorator* devem implementar o método com sua contribuição para o cálculo de utilidade do agente em questão para uma dada tarefa.

Na Figura 8, é mostrado os componentes modulares que foram desenvolvidos juntamente com o *framework* de utilidade do *TAlMech* para o cálculo básico de utilidade. Os decoradores básicos são definidos nas classes *FeatureUtility*, *BatteryUtility* e *DistanceUtility*. A classe *FeatureUtility* é uma classe filha de *UtilityDecorator*. Através dela é possível decorar o cálculo de utilidade para uma dada tarefa. Antes de ser usado, este componente de utilidade precisa ser inicializado com os dados do agente e os fatores de correção de cada um dos seus recursos. Os fatores de correção são utilizados para ajustar as unidades entre cada componente que influencia no cálculo de utilidade. A contribuição deste componente no cálculo de utilidade se dá a partir da comparação dos recursos exigidos durante a execução da tarefa com os recursos que o agente possui. Caso todos recursos exigidos forem encontrados, o grau de característica exigido é comparado com o grau de característica do agente em questão para cada um dos recursos exigidos. Ao final, tem-se a contribuição no cálculo de utilidade deste componente.

A classe *BatteryUtility* é um decorador de utilidade que também pode ser utilizado na configuração do cálculo de utilidade de uma dada tarefa. Entretanto, este componente de utilidade só pode ser utilizado por objetos do tipo *Robot*. Nota-se que na inicialização deste decorador são necessários os dados do robô e o fator de correção desejado. Este componente contribui com o cálculo de utilidade através da estimativa de bateria necessária para executar a tarefa dada e, também, da comparação desta estimativa com o nível atual de bateria do robô.

Por fim, a classe *DistanceUtility* também pode ser utilizada para decorar o cálculo de utilidade, pois esta classe herda de *UtilityDecorator*. Este decorador só pode ser utilizado por robôs, pois são necessários para esta parcela do cálculo os dados do robô e também o fator de correção desejado. Neste caso, são calculadas as distâncias entre as posições corrente do robô e de passagem exigidas pela tarefa. Estes componentes podem ser associados entre si ou com outros componentes desenvolvidos pelos usuários do *framework*.

As Figuras 7 e 8 mostram a estrutura das classes *UtilityFactory*, *BasicUtilityFactory*, *Utility* e *BasicUtility*. A classe abstrata *UtilityFactory* implementa os padrões de projeto *Template Method*, *Abstract Factory* e *Singleton*. Assim obteve-se uma fábrica de componentes de utilidade com um método robusto (isto é, fechado para modificação mas

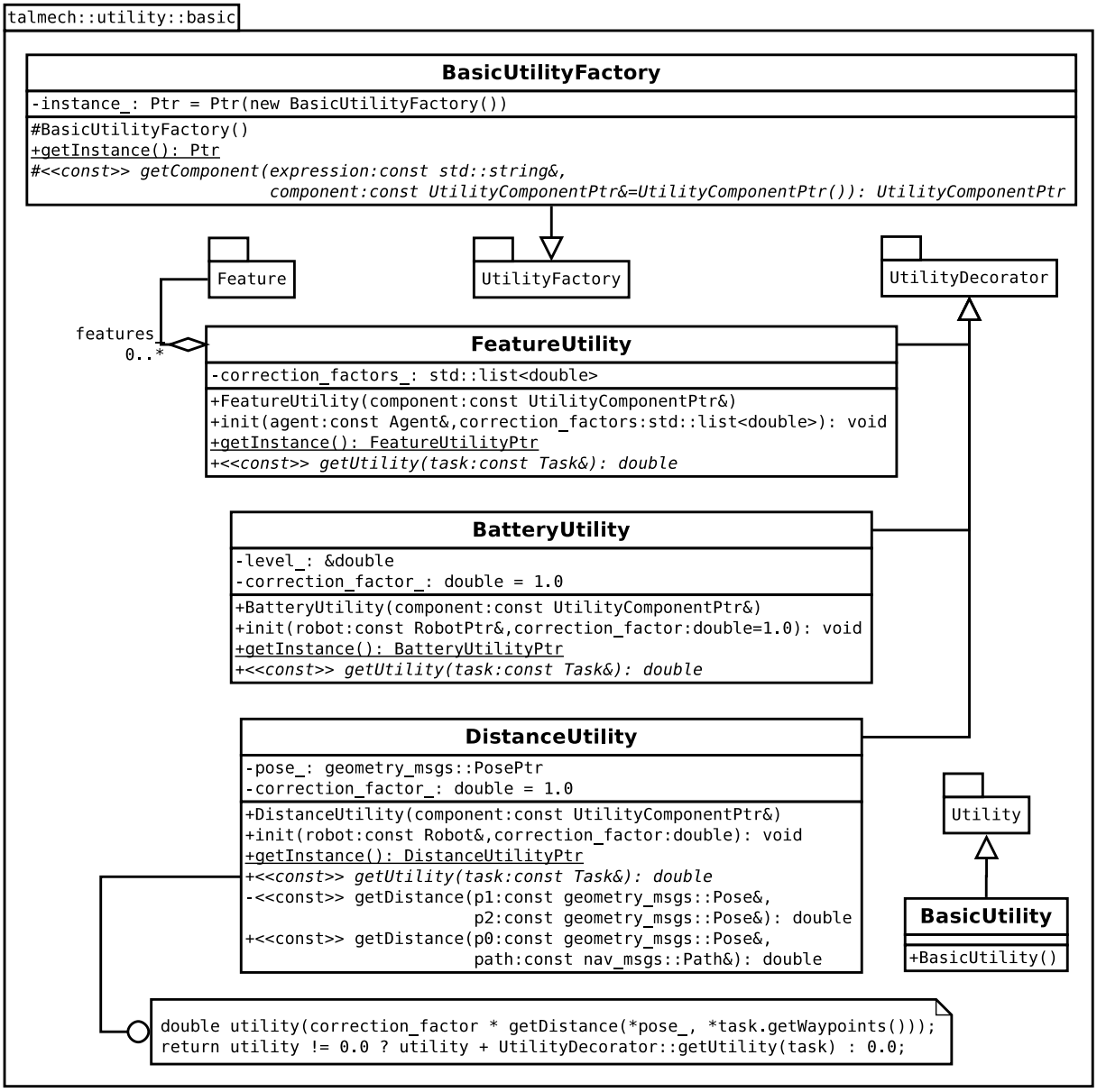


Figura 8 – Diagrama UML das classes do *framework TAlMech* para o cálculo de utilidade.

aberto para a extensão) para a decoração do cálculo de utilidade de um agente particular. Esta classe possui uma única instância de objeto em tempo de execução e tem como propósito facilitar a decoração do cálculo de utilidade.

A decoração acontece a partir da passagem de uma cadeia de caracteres contendo o nome dos componentes desejados para um dado robô. Cada nome de componente de utilidade deve ser separado por um delimitador. O delimitador padrão é o caractere espaço. É possível criar fábricas com novos componentes de utilidade ao herdar a classe abstrata *UtilityFactory* e implementar o método de criação dos novos componentes. *BasicUtilityFactory* é uma classe que herda de *UtilityFactory* e implementa o método de criação de componentes de utilidade para os decoradores *FeatureUtility*, *BatteryUtility* e *DistanceUtility*. Assim, ao utilizar a fábrica de utilidade básica pode-se fazer qualquer combinação

entre esses três componentes de utilidade para calcular a aptidão de um agente particular para uma dada tarefa. Esta classe também implementa o padrão de projeto *Singleton*, ou seja, só existe uma instância de objeto na memória deste tipo em tempo de execução. Recomenda-se aos clientes do *framework* de utilidade do *TAlMech* herde da classe *BasicUtilityFactory*, pois assim será possível utilizar os três componentes básicos de utilidade citados anteriormente. Finalmente, foi utilizado o padrão de projeto *Façade* na classe *Utility* para facilitar a utilização deste *framework* pelos agentes. Os agentes podem decorar e requisitar o cálculo de utilidade através do uso de apenas um objeto cujo tipo é *Utility*. Por fim, *BasicUtility* é uma classe especializada de *Utility* que utiliza como fábrica de componentes de utilidade um objeto do tipo *BasicUtilityFactory*.

A Figura 9 mostra exemplos de decoração do cálculo de utilidade. Estas decorações foram fabricadas a partir das cadeias de caractere "feature distance battery" e "battery feature", conforme as Figuras 9a e 9b ilustram, respectivamente. Ambas figuras mostram como o cálculo de utilidade para uma dada tarefa é requisitado. A Figura 9a mostra um caso em que o agente é capaz de realizar a tarefa dada. Por outro lado, a Figura 9b mostra uma situação em que o agente não é capaz de executá-la. Perceba que, neste caso, o cálculo de utilidade é interrompido assim que um de seus componentes retorna um valor nulo.

3.4 Negociação por Leilão

O *framework TAlMech* encapsula o mecanismo mais comum entre as arquiteturas de negociação, o leilão (ZLOT; STENTZ, 2006). O *TAlMech* implementa o protocolo de negociação entre leiloeiros e licitantes através dos recursos de *middleware* do *framework* ROS. Este mecanismo abstrai as regras de negociação da execução das tarefas da aplicação desejada. Isso permite que a arquitetura cliente seja independente do domínio, isto é, genérica.

A Figura 10 mostra o diagrama UML das principais classes utilizadas no desenvolvimento do *framework* para leilão, são elas: *Bidder*, *BidderAgent*, *BidderRobot*, *Auctioneer*, *AuctioneerAgent*, *AuctioneerRobot*, *Bid*, *HighestBid*, *AuctionEvaluator* e *Auction*. Serão descritos, a seguir, as características e os relacionamentos entre as classes que são responsáveis pelo controle do mecanismo de leilão dentro do *framework TAlMech*.

Um lance de leilão é ofertado por licitantes enquanto os leiloeiros o recebem para avaliar o vencedor. Um lance identifica o tempo em que ele foi ofertado, o licitante remetente, o leiloeiro destinatário, o leilão, o valor ofertado e código identificador. Essas informações são armazenadas em objetos do tipo *Bid*. Esta classe implementa a interface *ToMsg* pois seus objetos podem ser convertidos para a mensagem do ROS *talmech_msgs/Bid*. A classe *AuctionEvaluator* utiliza o padrão de projeto *Strategy*, pois abstrai a implementa-

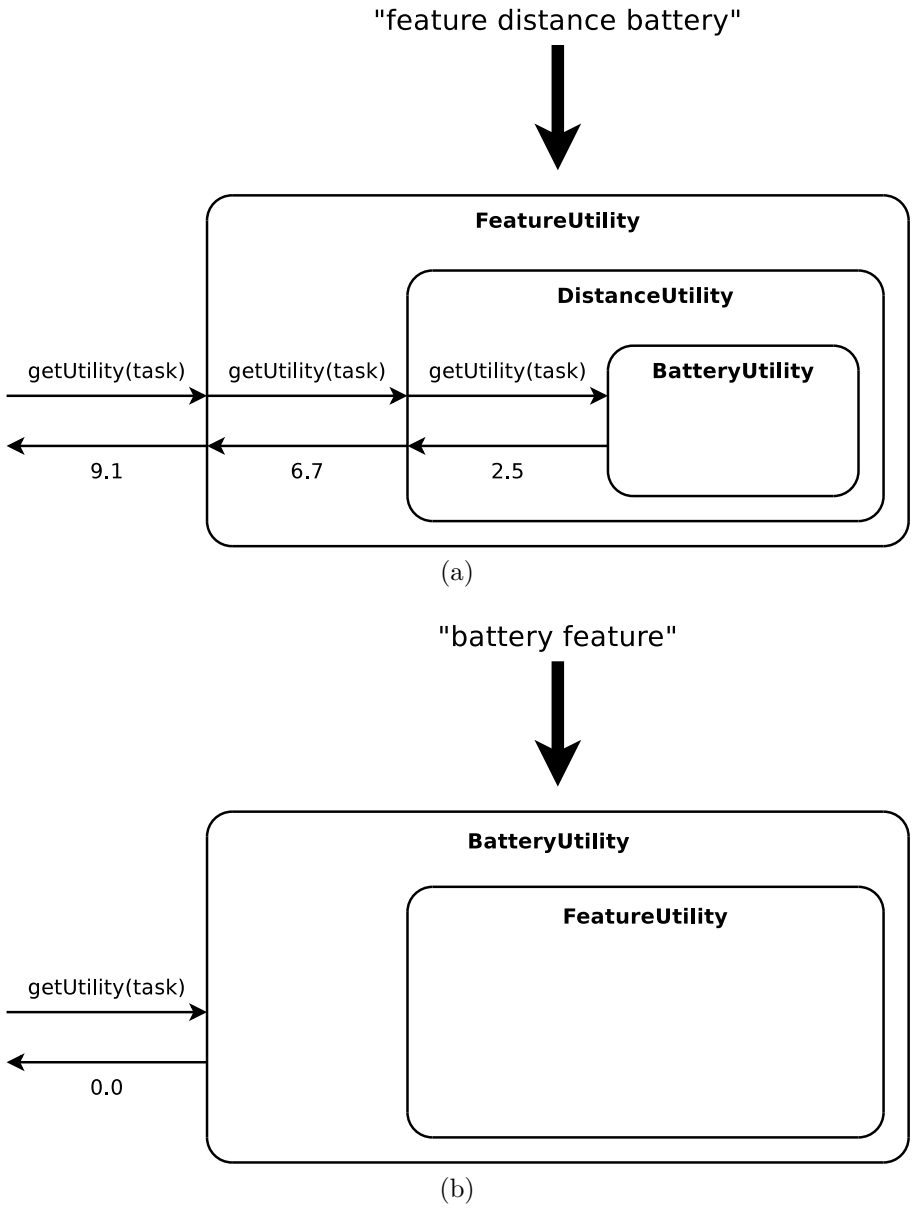


Figura 9 – Exemplos de decoração do cálculo de utilidade.

ção da política de avaliação dos lances pelo leiloeiro, permitindo que outras abordagens de avaliação desenvolvidas pelos clientes do *framework TAlMech* possam ser utilizadas com facilidade. O método de avaliação é definido pelo comparador escolhido. A classe *HighestBid* é um exemplo de comparador que implementa a interface *Comparator* com a política utilizada em leilões de primeiro preço e de Vickrey.

Os parâmetros de configuração de um leilão de uma dada tarefa são armazenados na classe *Auction*. Objetos do tipo *Auction* armazenam o código identificador do leilão, a identificação do seu leiloeiro, a tarefa em leilão, o valor do seu preço de reserva, seu instante de início, sua duração, o conjunto de lances direcionados para o leilão em questão, seu instante de encerramento, a identificação do licitante vencedor, a taxa de envio das mensagens de renovação de contrato, o instante da última renovação do contrato, o

instante em que o leilão foi abortado, seu instante de conclusão e o método de avaliação dos lances. Ainda é possível configurar se o modo de inserção de lances é ordenado, se é permitido leiloar uma tarefa cuja execução não foi concluída e se é permitido que um licitante atualize seu lance. Estas informações dão flexibilidade ao configurar as regras dos leilões. Enfim, esta é uma classe que implementa a interface *ToMsg*, pois seus objetos podem ser convertidos para mensagens do ROS *talmech_msgs/Auction*.

A classe *Bidder* é uma especialização da classe abstrata *Role*. Os agentes do sistema que configuram seu papel como uma instância de *Bidder* são capazes de licitar em um leilão. Os licitantes podem instanciar dinamicamente os controladores de licitação (*BiddingController*). Este tipo de controlador é descrito em 3.4.2. As classes *AgentBidder* e *RobotBidder* são, respectivamente, especializações das classes *Agent* e *Robot* que especificam a classe *Bidder* como papel do agente. Essas classes implementam os padrões de projeto *Façade* e Fábrica Abstrata para facilitar a configuração e a utilização de agentes licitantes através do *framework TAlMech*.

A classe *Auctioneer* é uma outra especialização da classe abstrata *Role*. Através do uso desta classe, os agentes do sistema podem desempenhar o papel de um leiloeiro em um leilão. As configurações de leilão são guardadas com ele para serem usadas na criação de novos leilões. Essas configurações são muito importantes, pois elas ditam o funcionamento do mecanismo de leilão e, consequentemente, o processo de alocação das tarefas no sistema. Além disso, os leiloeiros podem instanciar controladores de leilão (*AuctioningController*) dinamicamente. Este tipo de controlador é descrito em 3.4.1. As classes *AgentAuctioneer* e *RobotAuctioneer* são, respectivamente, especializações das classes *Agent* e *Robot* que especificam a classe *Auctioneer* como papel do agente. Essas classes implementam os padrões de projeto *Façade* e Fábrica Abstrata para facilitar a configuração e utilização de agentes leiloeiros através do *framework TAlMech*.

Os leiloeiros se comunicam com os licitantes através de mensagens transportadas entre os nós do ROS via tópico, conforme mostra a Figura 11 (vide Apêndice A para compreender os conceitos básicos do ROS). *AuctioneerNode* e *BidderNode* são nós que utilizam o *framework TAlMech* para alocar tarefas através do mecanismo de leilão.

O nó *AuctioneerNode* faz uso de um ponteiro inteligente para um objeto do tipo *AuctioneerAgent* que é responsável por controlar um agente que desempenha o papel de leiloeiro no sistema. Cada vez que o leiloeiro é requisitado para alocar uma tarefa, se possível, este cria uma nova instância de leilão (*Auction*) conforme suas configurações predefinidas e, também, uma nova instância de controlador de leilão para controlar cada etapa do leilão. O leiloeiro anuncia a tarefa na primeira etapa do leilão através do envio de uma única mensagem pelo tópico */auction/announcement*. Em seguida, na etapa de submissão das ofertas, ele fica aguardando o recebimento dos lances que os licitantes ofertam pela tarefa em leilão pelo tópico */auction/submission*. Logo após o final desta

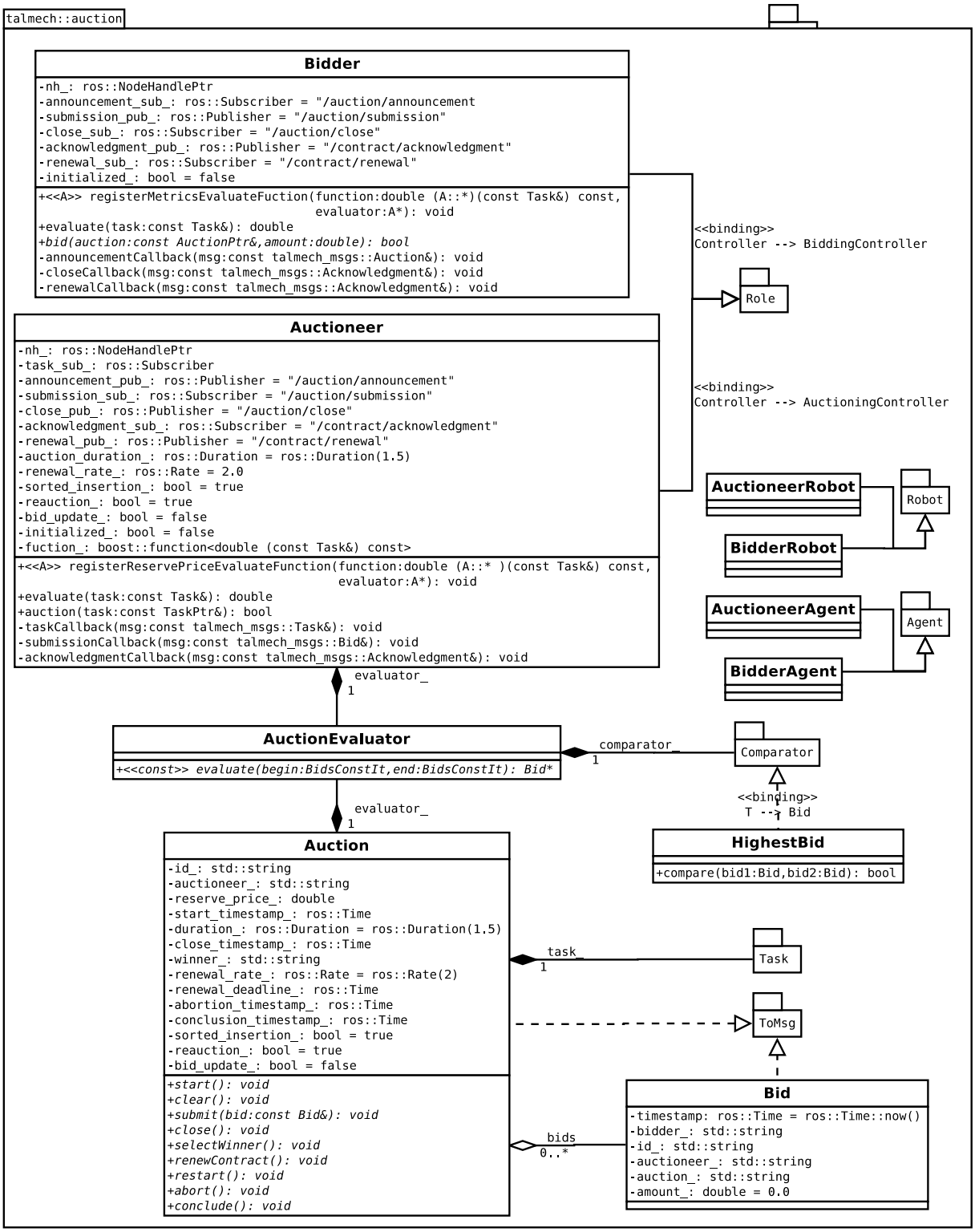


Figura 10 – Diagrama UML das classes básicas do mecanismo de leilão do *framework TAlMech*.

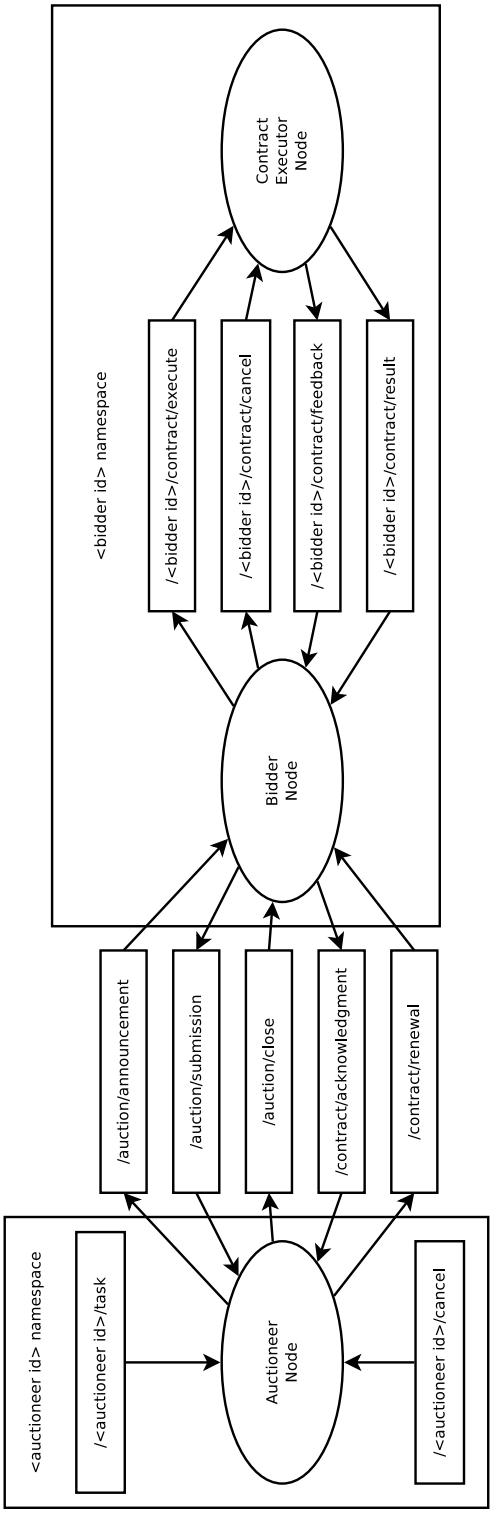


Figura 11 – Comunicação no ROS entre nós que utilizam o mecanismo de leilão para a alocação de tarefas em um sistema multirrobô: o nó *AuctioneerNode* utiliza um objeto do tipo *AuctioneerAgent* que encapsula as funcionalidades de um leiloeiro, o nó *BidderNode* utiliza um objeto do tipo *BidderRobot* que encapsula as funcionalidades de um licitante e o nó *ContractExecutorNode* controla a execução das tarefas que forem alocadas a ele. Os retângulos grandes, as elipses e os retângulos pequenos representam respectivamente os *namespaces*, os nós e os tópicos no ROS.

etapa, o leiloeiro anuncia o vencedor do leilão em questão, se houver, através do tópico */auction/close*, dando início à vigência de um contrato para a execução da tarefa leiloada. Durante esta etapa, o leiloeiro fica aguardando notificações relacionadas ao andamento da tarefa em execução por meio do tópico */contract/acknowledgment* e envia periodicamente a renovação do contrato pelo tópico */contract/renewal*. Este processo acontece sempre que o leiloeiro é notificado que a tarefa continua em bom andamento. Caso a comunicação entre o leiloeiro e o vencedor deixe de existir ou se a execução da tarefa for concluída, abortada ou cancelada, a vigência do contrato é encerrada.

Do outro lado, o nó *BidderNode* utiliza um ponteiro inteligente para uma instância de *BidderRobot*, a qual é responsável por controlar um robô do sistema que desenvolve o papel de licitante. Quando há disponibilidade para a execução de tarefas, o licitante fica observando os anúncios de tarefas enviados pelos leiloeiros através do tópico */auction/announcement*. Sempre que é identificado uma tarefa que pode ser executada pelo agente, o licitante cria uma nova instância de controlador de licitação. Ao fazê-lo, inicia-se a negociação entre este licitante com o leiloeiro. Primeiramente, o licitante envia sua oferta para o leiloeiro por meio do tópico */auction/submission*. Em seguida, o licitante fica aguardando o fechamento do leilão. Assim que o leilão é encerrado, o licitante recebe uma mensagem através do tópico */auction/close* identificando o vencedor. Caso o licitante seja o vencedor do leilão, este confirma o recebimento da mensagem de encerramento do leilão através do tópico */contract/acknowledgment*. Esta ação dá início à vigência do contrato de execução da tarefa. Enquanto o licitante executa a tarefa, este recebe periodicamente mensagens de renovação do contrato através do tópico */contract/renewal*. Logo em seguida, o vencedor envia uma mensagem no tópico */contract/acknowledgment* identificando o andamento da execução da tarefa. O contrato continua em vigor até que a comunicação entre o licitante vencedor e o leiloeiro deixe de existir ou quando a execução da tarefa é concluída, abortada ou cancelada.

Além de interagir com o nó *AuctioneerNode*, o nó *BidderNode* também interage com nós do tipo *ContractExecutorNode*. Os nós *BidderNode* e *ContractExecutorNode* se comunicam para que haja notificação do início, do progresso e do resultado da execução das tarefas alocadas ao agente em questão. A cada contrato firmado o licitante envia os dados da tarefa alocada ao nó executor através do tópico *contract/execute*. Note que o nome deste recurso é relativo (vide Apêndice A), de modo que o seu *namespace* é dado pelo código de identificação do licitante. Isto vale para todos os nós que interligam o par de nós *BidderNode* e *ContractExecutorNode*. O licitante pode enviar o cancelamento da execução de uma tarefa particular em execução através do tópico *contract/cancel*. O nó *BidderNode* recebe mensagens sobre o progresso e resultado da execução da tarefa através dos tópicos *contract/feedback* e *contract/result*, respectivamente. Caso a execução de uma dada tarefa entre em estado terminal, o licitante notifica apropriadamente o leiloeiro sobre o encerramento do contrato. Se a comunicação entre este par de nós se interromper

durante a execução de tarefas, o licitante notifica à cada leiloeiro envolvido sobre a quebra do contrato.

Opostamente, o nó *ContractExecutorNode* é responsável por controlar a execução das tarefas que são requisitadas pelo nó *BidderNode* a ele. O *framework TAlMech* apenas especifica a interface de comunicação que deve ser usada pelo nó que controla a execução da tarefa. A cada tarefa alocada ao licitante, o nó executor recebe os dados necessários para iniciar sua execução. Essas mensagens são recebidas por meio do tópico *contract/execute*. O nó *ContractExecutorNode* recebe requisições de cancelamento da execução de tarefas em execução através do tópico *contract/cancel*. Periodicamente, o executor envia informações sobre o andamento da execução da tarefa ao seu respectivo licitante através do tópico *contract/feedback*. Por fim, quando a execução de alguma tarefa passa a estar em estado terminal, o executor envia seu resultado ao licitante através do tópico *contract/result*.

3.4.1 Controlador de leilão

O controle de leilão é realizado por agentes que desempenham o papel de leiloeiro através de uma máquina de estados. Esta máquina é composta por cinco estados: *Anunciando Tarefa*, *Aguardando Prazo de Submissão*, *Selecionando Ganhador*, *Renovando Contrato* e *Aguardando Exclusão*, conforme mostra a Figura 12.

Anunciando Tarefa é o estado inicial desta máquina. Neste estado, o leiloeiro anuncia o início do leilão para os agentes licitantes do sistema. Todas as informações pertinentes para a execução da tarefa são enviadas para eles. Deste modo, cada licitante pode verificar se é capaz de realizar tal tarefa.

Após o anúncio da tarefa, o controlador da máquina configura o estado *Aguardando Prazo de Submissão* como seu estado corrente. O leiloeiro permanece neste estado durante um período de tempo predefinido. E, enquanto o estado está ativo, o leiloeiro coleta os lances que são ofertados pelos licitantes. Assim que o prazo de submissão é expirado, o leiloeiro cessa de coletar os lances dos licitantes e contabiliza o número de lances ofertados. Se nenhum lance tenha sido ofertado, a máquina passa a estar no estado *Aguardando Exclusão*. Caso contrário, o controlador configura o estado *Selecionando Ganhador* como estado corrente da máquina.

O leiloeiro avalia todos os lances ofertados no estado *Selecionando Ganhador*. Assim que o ganhador do leilão é selecionado, o leiloeiro anuncia o encerramento e o ganhador do leilão. Então a máquina avança para o estado *Renovando Contrato*.

Enquanto o leiloeiro permanece no estado *Renovando Contrato*, o contrato continua em vigor. O ganhador tem um prazo para notificar o andamento da execução da tarefa. Se o leiloeiro recebe a notificação antes do prazo estabelecido e a tarefa continua em perfeito andamento, o contrato é renovado e o seu prazo é redefinido. Este processo

se repete continuamente enquanto a máquina não muda de estado. O contrato deixa de ser vigente quando o contrato expira ou é cancelado, abortado ou concluído. Quando o contrato expira ou é abortado pelo ganhador, o estado da máquina move para *Anunciando Tarefa* se as regras do leilão permitem que a tarefa seja leiloada novamente; caso contrário, o próximo estado da máquina é *Aguardando Exclusão*. Porém, quando o contrato é cancelado ou concluído, o estado da máquina passar a ser *Aguardando Exclusão*.

Aguardando Exclusão é o estado final da máquina. Quando a máquina atinge o estado final, seu controlador deixa de ser efetivo. O leiloeiro fica apenas aguardando a reciclagem desta máquina de estado. Nenhuma ação é realizada neste estado.

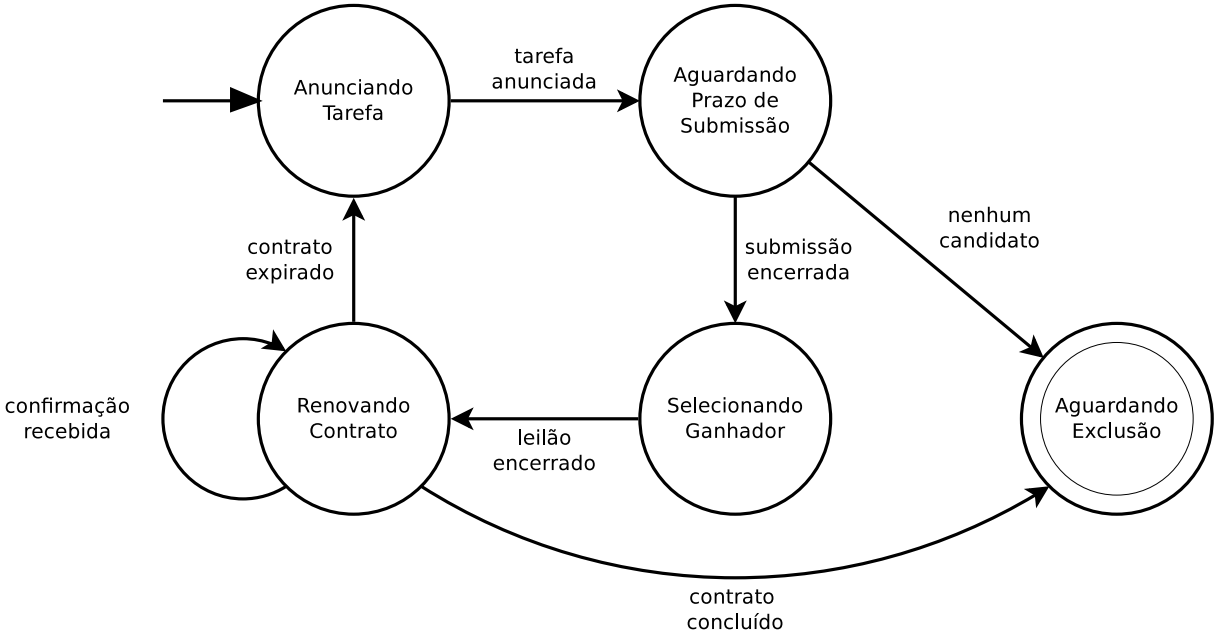


Figura 12 – Máquina de estado utilizada pelo leiloeiro enquanto leiloa uma tarefa.

O *framework TAlMech* disponibiliza o papel de leiloeiro para os agentes de um sistema para alocação de tarefas. Este papel possui configurações flexíveis que influenciam na forma com que o agente leiloa as tarefas. Uma dessas configurações é definida na classe abstrata *Role* e diz respeito ao número de leilões que o leiloeiro é capaz de tratar simultaneamente. Assim, o leiloeiro necessita verificar o número de leilão que ele está operando antes de atender à uma requisição de leilão de uma dada tarefa. Dada sua disponibilidade para atender à requisição, este cria uma nova instância de controlador de leilão.

A Figura 13 mostra um diagrama UML que define e relaciona as classes *AuctioningController*, *AuctioningState*, *AnnouncingTask*, *AwaitingAuctionDeadline*, *SelectingWinner*, *RenewingContract* e *AwaitingAuctioningDisposal*. Essas classes são responsáveis por controlar as ações dos leiloeiros durante um leilão.

A classe *AuctioningController* é uma especialização da classe *MachineController* que controla a transição dos estados representados na Figura 12. A responsabilidade desta

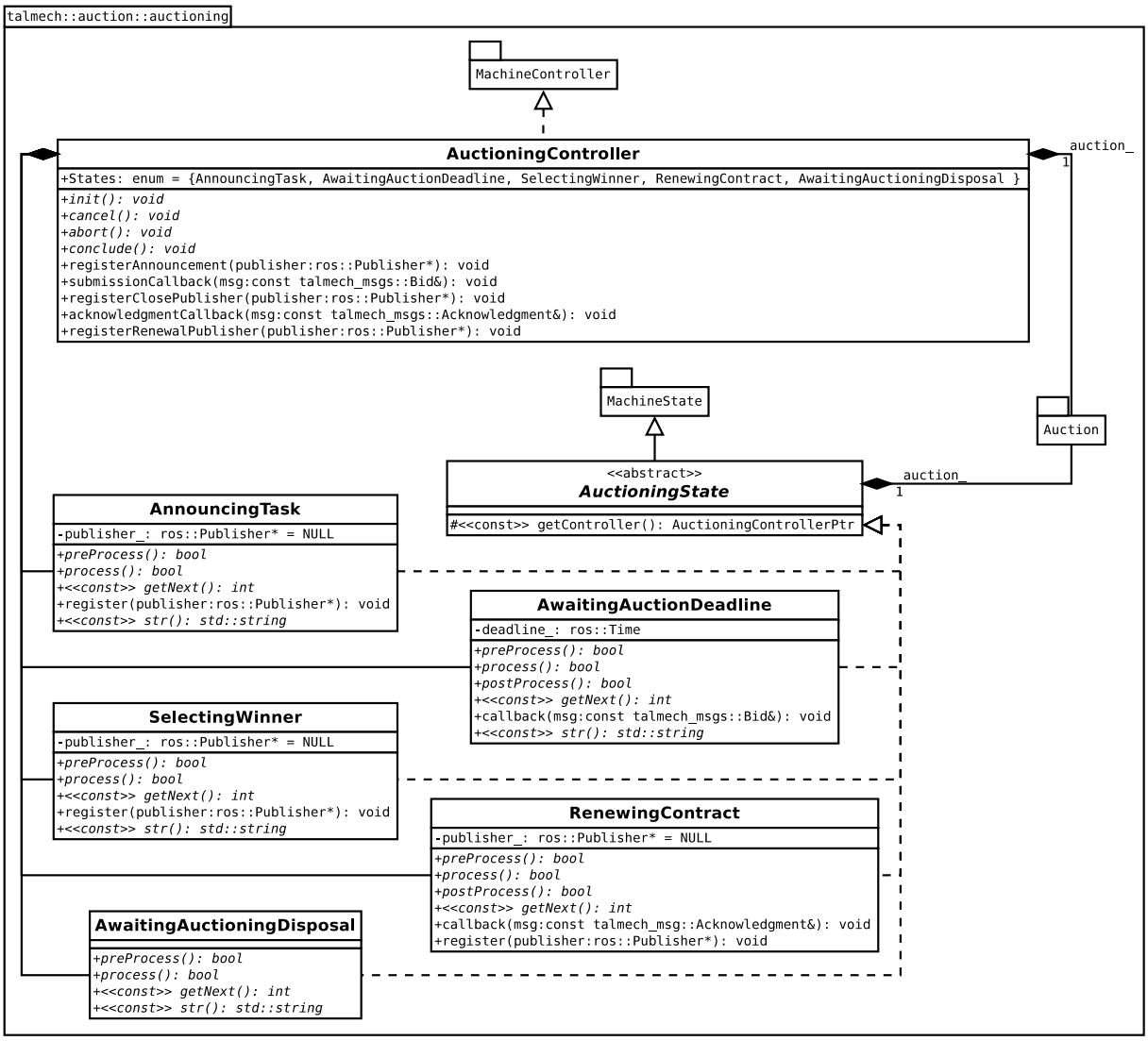


Figura 13 – Diagrama UML da máquina de estado utilizada pelo leiloeiro enquanto leiloa uma tarefa.

classe é inicializar os estados da máquina corretamente. Cada estado é identificado por um enumerado. Isso facilita a indexação dos estados na máquina de estados. Além disso, esta classe é responsável por registrar apropriadamente os publicadores do ROS nas instâncias de estado da máquina, bem como, encaminhar as mensagens dos licitantes para os estados corretamente.

Todos os estados do controlador são instâncias de *AuctioningState*, uma especialização da classe *MachineState*. Esta classe contém uma instância da classe *Auction* que pode ser acessada por herança para consultar as regras do leilão e atualizá-lo. Suas especializações são as classes *AnnouncingTask*, *AwaitingAuctionDeadline*, *SelectingWinner*, *RenewingContract* e *AwaitingAuctioningDisposal*, as quais implementam o comportamento do leiloeiro quando a máquina se encontra no estado *Anunciando Tarefa*, *Aguardando Prazo de Submissão*, *Selecionando Ganhador*, *Renovando Contrato* e *Aguardando Exclusão*, respectivamente.

3.4.2 Controlador de Licitação

O controle de licitação é realizado por agentes que desempenham o papel de licitante através de uma máquina de estados. Esta máquina é composta por três estados: *Aguardando Encerramento do Leilão*, *Aguardando Renovação do Contrato* e *Aguardando Exclusão*, conforme mostra a Figura 14.

A máquina é iniciada no estado *Aguardando Encerramento do Leilão*. Durante o preprocessamento deste estado, o licitante envia sua oferta com respeito à tarefa em leilão ao leiloeiro. Após esta ação, o licitante permanece neste estado enquanto ele não recebe a notificação do encerramento e do ganhador do leilão. Se este licitante não for o ganhador do leilão, a máquina avança para o estado *Aguardando Exclusão*.

Entretanto, se o licitante ganhar o direito de executar a tarefa leiloada, a máquina evolui para o estado *Aguardando Renovação do Contrato*. Durante o preprocessamento deste estado, o licitante envia um mensagem para o leiloeiro informando o início da execução da tarefa. Neste momento, o contrato passa a ser vigente. A licitação permanece neste estado enquanto o leiloeiro mantém o contrato em vigor, isto é, enquanto o leiloeiro envia a renovação do contrato. Cada vez que o licitante recebe uma mensagem de renovação de contrato, este precisa informar ao leiloeiro sobre a situação em que a execução da tarefa se encontra. Ao fazê-lo, o leiloeiro decide se o contrato será renovado. Caso o contrato não seja renovado após alguns instantes, o licitante considera que este contrato deixou de estar em vigor. Logo, o agente para de executar a tarefa em questão.

Quando o contrato é expirado, concluído, abortado ou cancelado, o estado da máquina passa a ser *Aguardando Exclusão*. Assim que a máquina atinge este estado, seu controlador deixa de ser efetivo. A licitação se encerra e permanece neste estado até que seu controlador seja excluído pelo papel do agente.

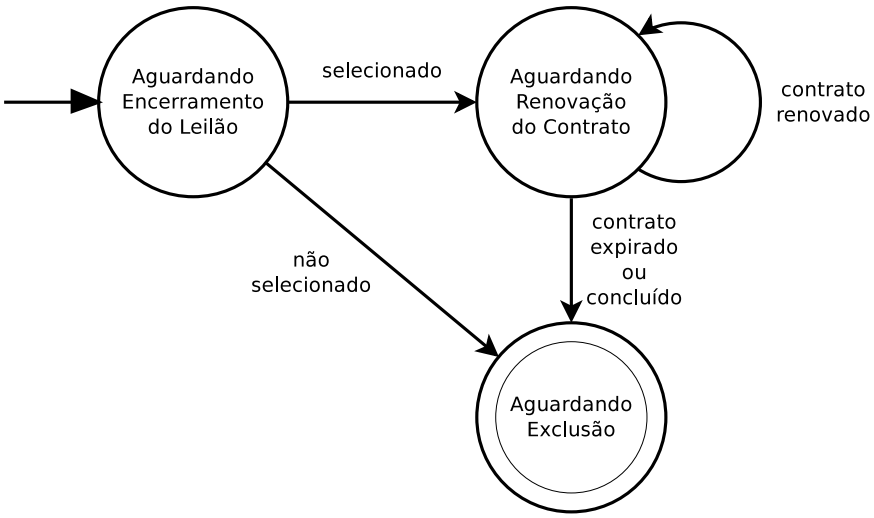


Figura 14 – Máquina de estado utilizada pelo licitante enquanto licita uma tarefa.

O *framework TAlMech* disponibiliza o papel de licitante para os agentes de um

sistema para alocação de tarefas. Este papel possui configurações flexíveis que influenciam na forma com que o agente licita as tarefas nos leilões. Uma dessas configurações é definida na classe abstrata *Role* e diz respeito ao número de leilões que o licitante é capaz de tratar simultaneamente. Este parâmetro deve estar de acordo com o número de tarefas que o agente pode executar paralelamente. Assim, o licitante necessita verificar o número de leilões que ele está processando antes de dar lance para uma outra tarefa anunciada. Dada a sua disponibilidade, ainda é verificada a aptidão do agente para executar a tarefa anunciada. Se o agente for capaz de executar esta tarefa, então é criada uma nova instância de controlador de licitação no papel de licitante do agente.

O diagrama de classes mostrado na Figura 15 define e relaciona as classes *BiddingController*, *BiddingState*, *AwaitingAuctionClose*, *AwaitingContractRenewal* e *AwaitingBiddingDisposal*, as quais são responsáveis por controlar as ações dos licitantes durante um leilão.

A classe *BiddingController* é uma especialização da classe *MachineController* que controla a máquina de estados da Figura 14. Esta especialização cria devidamente os estados da máquina e a inicializa apropriadamente. A indexação dos estados é feita por enumerados com o intuito de facilitar a identificação dos estados. Além disso, a classe *BiddingController* registra os publicadores do ROS nos estados adequados e também encaminha as mensagens recebidas pelo licitante para os devidos estados.

O controlador de licitação trata apenas estados do tipo *BiddingState*, uma classe abstrata que especializa a classe *MachineState*. Esta classe contém uma instância de *Auction* com as informações sobre o leilão e a tarefa leiloada. Ela também possui uma instância de *Bid* com os dados sobre o lance dado pelo licitante. As classes *AwaitingAuctionClose*, *AwaitingContractRenewal* e *AwaitingBiddingDisposal* são especializações da classe *BiddingState*, as quais implementam o comportamento do licitante quando a máquina se encontra no estado *Aguardando Encerramento do Leilão*, *Aguardando Renovação do Contrato* e *Aguardando Exclusão*, respectivamente.

3.5 Monitoramento de Comportamento

Em arquiteturas baseadas em comportamento, conforme foi visto na Subseção 2.4.1, os agentes são motivados internamente para ativar um de seus comportamentos mediante o seu próprio estado, do ambiente e dos demais robôs. Para isso, o *framework TAlMech* disponibiliza um mecanismo que pode ser usado nos componentes da função de ativação dos comportamentos dos agentes do sistema para verificar as atividades dos demais agentes do sistema.

Este mecanismo mantém um histórico com janela temporal dinâmica dos comportamentos dos agentes. Este histórico é adquirido a partir da comunicação entre os agentes

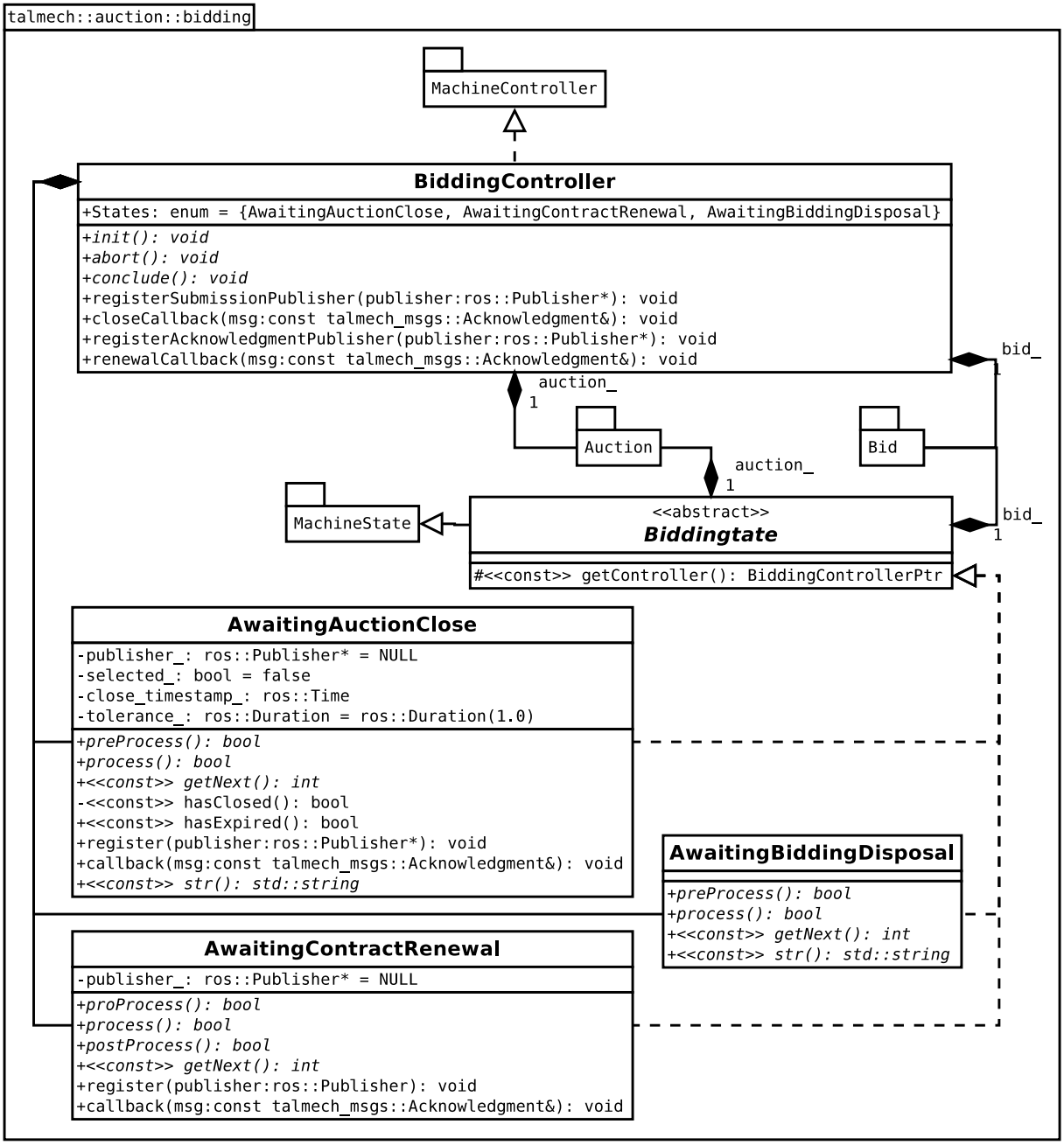
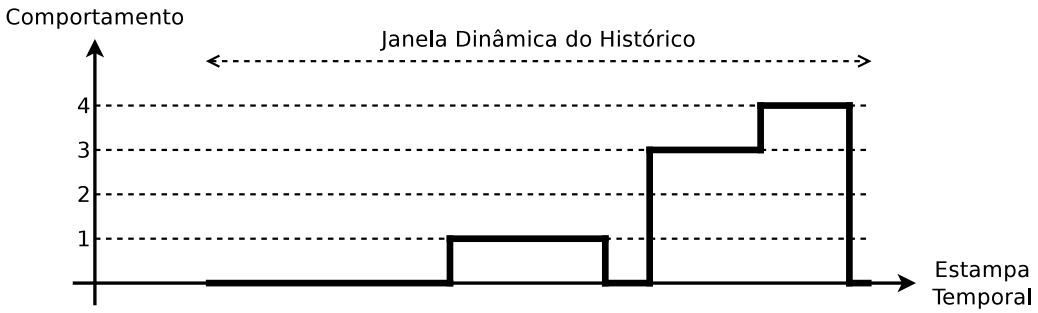


Figura 15 – Diagrama UML da máquina de estado utilizada pelo licitante enquanto licita uma tarefa.

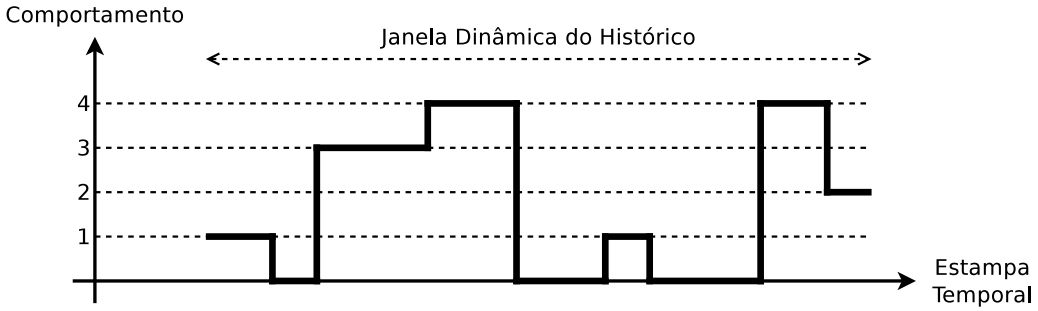
no sistemas durante a execução da aplicação. Cada agente pode assumir um ou nenhum comportamento por vez.

A Figura 16 exemplifica o histórico de comportamento de um agente que possui quatro comportamentos. As janelas mostradas na figura foram capturadas em três instantes diferentes. As Figuras 16a, 16b e 16c mostram respectivamente a primeira, a segunda e a terceira capturas de janela deste histórico. Os gráficos mostram funções temporais do comportamento do agente dentro de um intervalo predefinido. Este intervalo de tempo é denominado janela, a qual é dinâmica pois se move conforme a estampa temporal (isto é,

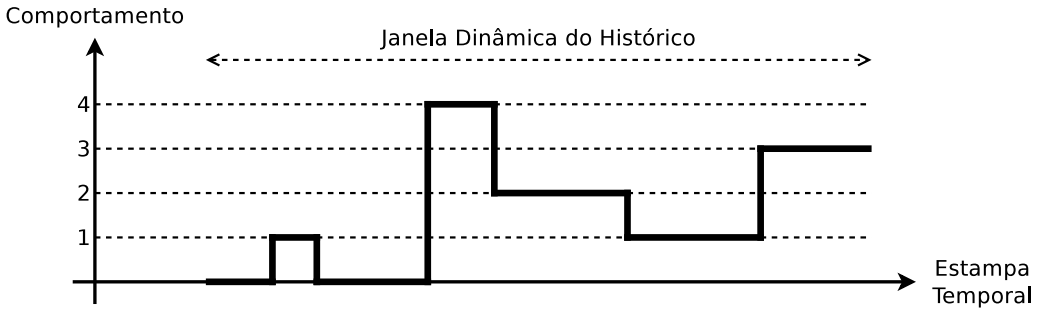
data e horário) atual se atualiza. As três figuras mostram a dinâmica da janela através do deslocamento da curva de comportamento.



(a) Primeira captura.



(b) Segunda captura.



(c) Terceira captura.

Figura 16 – Captura do histórico de um agente em três instantes diferentes. Em cada gráfico, o eixo das abcissas identifica as estampas temporais (isto é, data e horário) em que foram feitas as leituras e o eixo das ordenadas identifica o comportamento ativo do agente.

Contudo, a construção do histórico da curva de comportamento de um dado agente do sistema é realizada a partir do recebimento de mensagens através da rede. No *framework TAlMech*, essa troca de mensagens acontece no tópico */behavior* cujo tipo é *talmech_msgs/Behavior*. Este tipo de mensagem contém a identificação do agente que a enviou, do comportamento, a tarefa que está em execução e a estampa temporal gerada na sua criação.

A Figura 17 ilustra a construção do histórico de um agente que possui três comportamentos. As Figuras 17b, 17c e 17d mostram as estampas temporais das mensagens recebidas que dizem respeito aos comportamentos 1, 2 e 3 do agente em questão, respecti-

vamente. A Figura 17a mostra a curva de comportamento criado a partir das mensagens recebidas.

Primeiramente, uma mensagem do comportamento 1 do agente monitorado é recebida, conforme é mostrado na Figura 17b. Este pulso provoca uma rampa de subida na curva de comportamento, fazendo com que o comportamento do agente passe a estar com o comportamento 1 ativo ao invés de nenhum, conforme mostra a Figura 17a. Após alguns instantes (*timeout*), o agente volta a estar com nenhum comportamento ativo. O parâmetro *timeout* é a duração máxima que o agente pode manter-se em silêncio. Quando esta duração é excedida, o monitor considera que o agente está com nenhum comportamento ativo.

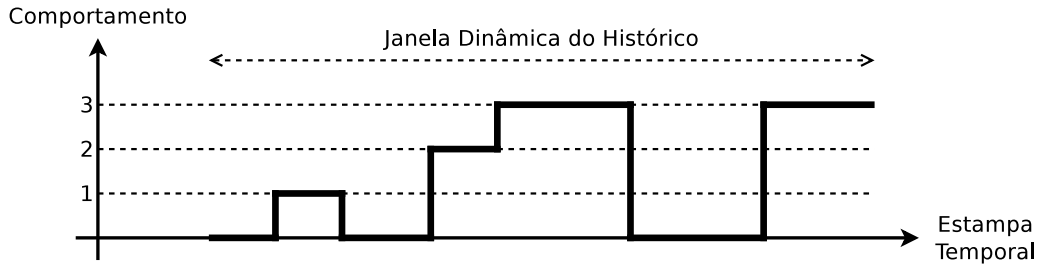
Na sequência, são recebidas várias mensagens relacionadas ao comportamento 2, de acordo com a Figura 17c. Isso faz com que o agente monitorado passe a estar com o comportamento 2 ativo.

Contudo, logo é recebido um conjunto de mensagens relacionadas ao comportamento 3, mediante Figura 17d. Com isso, o agente deixa de estar com o comportamento 2 ativo e passa a estar com o 3 ativo (vide Figura 17a). A seguir, o monitor passa a considerar que o agente está com nenhum comportamento ativo, pois este não recebe mais mensagens do agente em questão. Assim que a duração *timeout* é excedida, ocorre uma rampa de descida de 3 para 0 na curva de comportamento, conforme Figura 17a.

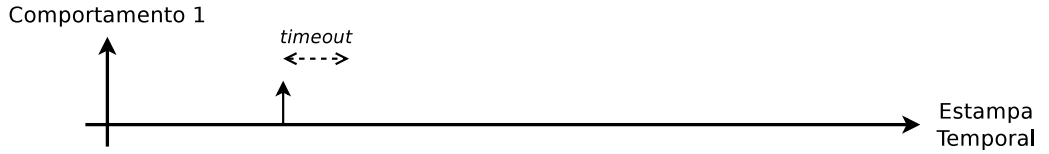
Por fim, o monitor torna a receber mensagens do agente em questão sobre o comportamento 3, conforme mostra a Figura 17d. Logo, acontece uma rampa de subida de 0 para 3 na curva de comportamento identificando que o agente ativou o comportamento 3 (vide Figura 17a).

A construção deste histórico no *framework TAlMech* é feita pelas classes: *TemporalPoint*, *TemporalBuffer* e *BehaviorMonitor*, conforme mostra o diagrama de classes da Figura 18. Primeiramente, a classe *TemporalPoint* armazena a estampa temporal e o índice do comportamento do agente no momento que ocorrem variações na sua curva de comportamento. Assim, toda vez que ocorre um rampa de subida ou descida, a classe *TemporalBuffer* adiciona um par ordenado (estampa temporal e índice do comportamento) na curva de comportamento do agente. Esta classe é responsável por eliminar pares ordenados que se encontram fora da janela.

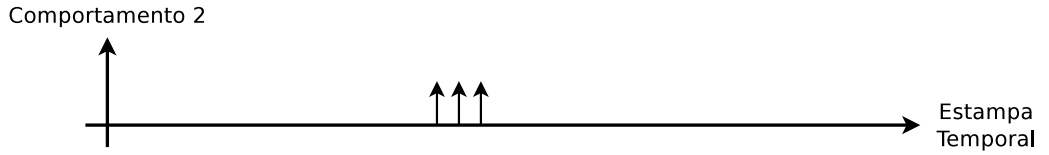
Enfim, a classe *BehaviorMonitor* cria dinamicamente novas instâncias de *TemporalBuffer* conforme são identificados novos agentes no sistema pelo agente que está monitorando. Esta classe deve ser instanciada em nós do ROS, pois esta classe recebe mensagens do tipo *talmech_msgs/Behavior* com o código de identificação do agente que enviou a mensagem, do seu comportamento ativo e sua respectiva tarefa, bem como, a estampa temporal do envio da mensagem. Essas mensagens são recebidas por meio do



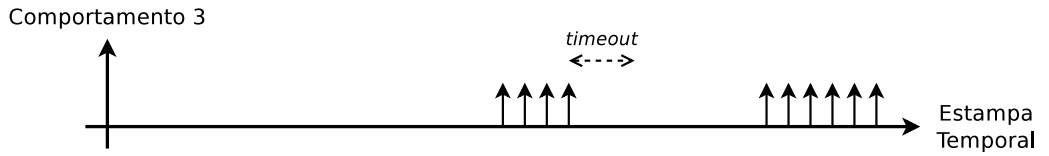
(a) Histórico construído a partir do recebimento das mensagens identificando o comportamento ativo ao longo do tempo do agente em questão.



(b) Trem de pulsos das mensagens recebidas do agente em questão quando este estava com o comportamento 1 ativo.



(c) Trem de pulsos das mensagens recebidas do agente em questão quando este estava com o comportamento 2 ativo.



(d) Trem de pulsos das mensagens recebidas do agente em questão quando este estava com o comportamento 3 ativo.

Figura 17 – Construção do histórico de um agente com três comportamentos.

tópico */behavior*. Portanto, é possível verificar se:

- um agente específico ativou algum comportamento dentro de um dado intervalo de tempo;
- um agente específico ativou o comportamento que executa uma tarefa particular em um dado intervalo de tempo;
- houve algum agente que ativou o comportamento que executa uma tarefa particular em um dado intervalo de tempo.

3.6 Controle de Ativação de Comportamento

As arquiteturas de alocação de tarefa para sistemas de múltiplos robôs que são baseadas em comportamento lidam com agentes comportamentais. Isto é, essas arquiteturas

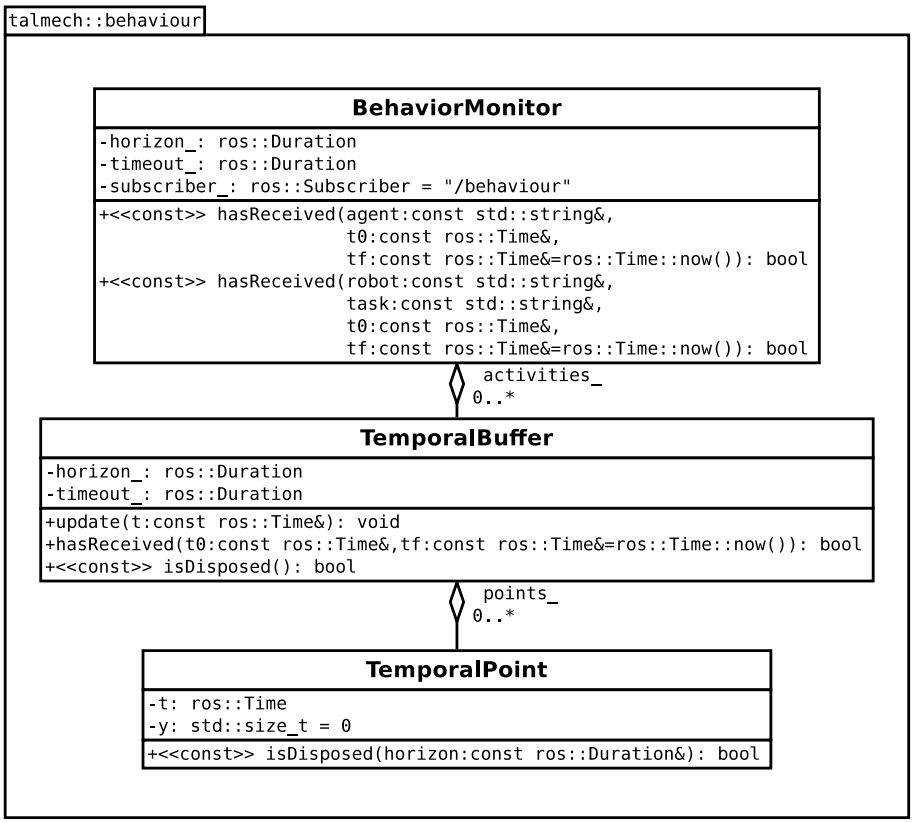


Figura 18 – Diagrama UML das classes que realizam o monitoramento dos comportamentos dos agentes do sistema.

consideram que os agentes do sistema possuem comportamentos. Quando um comportamento se torna ativo em um agente, este passa a executar um tarefa específica. Neste instante, ocorre uma alocação de tarefa no sistema.

Para isso, o *framework TAlMech* dispõe um mecanismo que realiza o controle da ativação de comportamentos de agentes comportamentais. O mecanismo proposto (1) publica periodicamente informações sobre o comportamento ativo do agente, (2) permite que no máximo um comportamento seja ativado por vez, (3) deve ser estendido para implementação de comportamentos característicos dos agentes para a aplicação e (4) é flexível quanto à representação do cálculo de motivação.

A Figura 19 define e relaciona as classes *MotivationComponent*, *MotivationDecorator*, *Behavior* e *Behaved*. Essas classes são utilizadas pelo *framework TAlMech* para controlar a ativação dos comportamentos de um dado agente.

A interface *MotivationComponent* define a assinatura do método que é utilizado para verificar se o agente está motivado a ativar o comportamento em questão. A classe abstrata *MotivationDecorator* implementa a interface *MotivationDecorator* e o padrão de projeto *Decorator*. Logo, a classe abstrata *MotivationDecorator* possui um objeto do tipo *MotivationComponent*. Esta formulação permite que a função de ativação de comportamento da arquitetura cliente do *framework TAlMech* possa ser representada estática ou

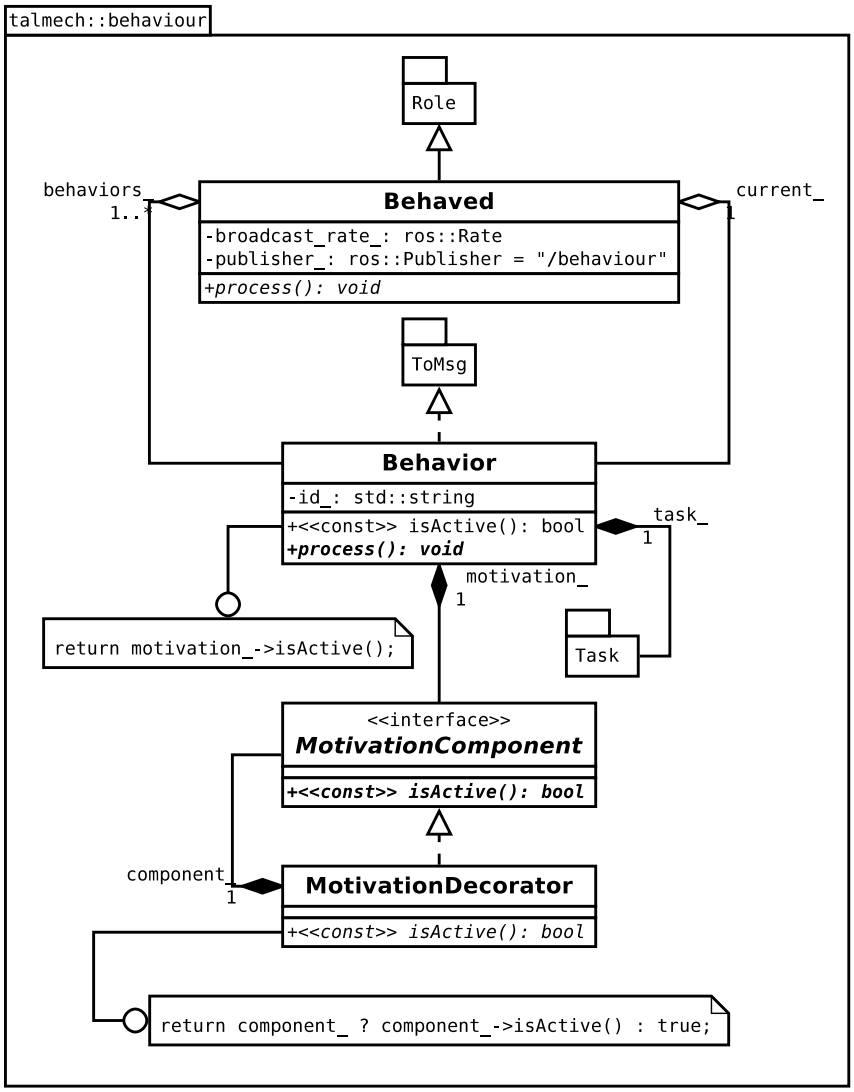
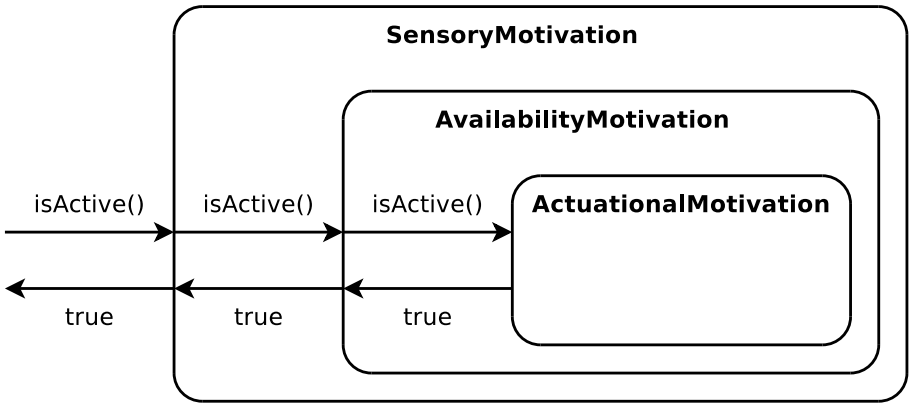


Figura 19 – Diagrama UML das classes que realizam o controle da ativação dos comportamentos dos agentes do sistema.

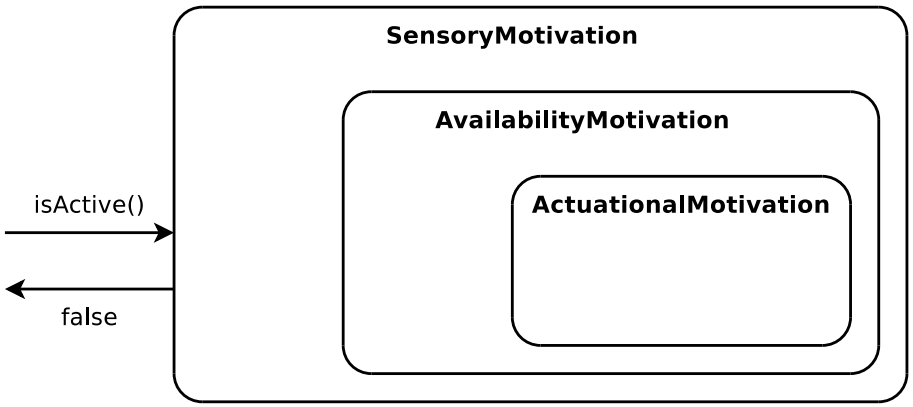
dinamicamente. Ou seja, a arquitetura pode definir o cálculo de motivação independente do comportamento e do agente ou dependente do comportamento e/ou do agente.

A Figura 20 mostra exemplos de decoração da função de ativação de comportamento e, também, como é feita a chamada pela verificação da motivação do agente para ativar o comportamento. Se a motivação para ativar o comportamento for verificada em todos os componentes da função de ativação, o comportamento será ativado, conforme mostra a Figura 20a. O cálculo de motivação é interrompido se um dos componentes da função retorna um valor falso, assim como mostra a Figura 20b.

A classe *Behavior* é uma especialização da interface *ToMsg* do *TAlMech*, pois as informações de objetos deste tipo podem ser convertidas para mensagens do tipo *talmech_msgs/Behavior*. Esta classe armazena informações pertinentes sobre comportamentos de um agente comportamental, como a decoração da sua função de ativação e a tarefa que está associada ao comportamento. A classe *Behavior* deve ser especializada por classes



(a) Agente motivado a ativar o comportamento.



(b) Agente sem motivação para ativar o comportamento.

Figura 20 – Exemplo de decoração da função de ativação de comportamento.

que implementam o processamento da execução das tarefas da aplicação desejada.

Finalmente, a classe *Behaved* é uma especialização da classe *Role* que pode ser incorporada por objetos do tipo *Agent* e *Robot*. Ao fazê-lo, o agente se torna comportamental, de modo que, este agente possua um conjunto de comportamentos definidos pela classe *Behavior*. A cada chamada de processamento desta classe, é feito o controle da publicação periódica, atualização e chamada de processamento do comportamento ativo do agente. O controle de publicação do comportamento do agente respeita uma frequência predefinida. Este controle permite que aconteça o monitoramento do comportamento dos agentes no sistema através do ROS. Para isso, são enviadas mensagens do tipo *talmech_msgs/Behavior* por meio do tópico */behavior*. Durante a atualização do comportamento do agente é verificado se o comportamento ativo anteriormente continua com a mesma motivação. Caso a motivação para a ativação de tal comportamento tenha se alterado, o comportamento em questão é desativado e, em seguida, é verificado se existe algum outro comportamento que deve ser ativado. Neste caso, o comportamento em questão é ativado. Caso contrário, o agente permanece com nenhum comportamento ativo. Por fim, é feita a chamada de processamento do comportamento ativo, caso exista.

4 Experimentos e Resultados

Este capítulo é dedicado a expor os testes realizados com os mecanismos de (1) cálculo de utilidade, (2) negociação por leilão, (3) monitoramento e (4) controle de ativação de comportamento. Primeiramente, é explicado o funcionamento do gerador randômico de tarefas, o qual é responsável por criar as tarefas que serão usadas para testar os mecanismos de utilidade e de leilão. Em seguida, sua eficácia foi verificada a partir de uma análise estatística e probabilística das tarefas geradas. Na sequência, o processo de amostragem de tarefas é exposto. Nesta fase, um conjunto de tarefas é armazenado para finalmente testar os mecanismo de utilidade e de leilão.

Quatro ensaios foram realizados para testar os mecanismos de utilidade e de leilão com as tarefas armazenadas. O mecanismo de negociação por leilão foi configurado distintamente em cada ensaio. Em um dos quatro ensaios, o mecanismo de leilão foi configurado de acordo com as características da arquitetura Murdoch. Enfim, os mecanismos relacionados a arquiteturas baseadas em comportamento foram testados. Para isso, é proposta uma aproximação da arquitetura ALLIANCE utilizando os mecanismos de controle de ativação e monitoramento de comportamento. Posteriormente, esta aproximação é testada em uma aplicação de patrulhamento.

4.1 Gerador Randômico de Tarefas Configurável

A fim de testar e validar os mecanismos de cálculo de utilidade e de negociação por leilão foi desenvolvido um gerador randômico de tarefas configurável. Este agente é responsável por gerar tarefas randomicamente. O intervalo entre a geração de tarefas é variável e segue uma distribuição normal. Pode ser definido um número máximo de tarefas geradas.

As tarefas geradas também são aleatórias. O número de pontos de passagem e o valor de cada coordenada de cada ponto de passagem obedecem distribuições normais. Além disso, a escolha de recurso cadastrado na geração da tarefa está associada à uma probabilidade. Enfim, a intensidade requerida de recursos discretos e contínuos também segue uma distribuição gaussiana.

O gerador é configurado através de uma conjunto de parâmetros que são lidos do servidor de parâmetros do ROS na inicialização do nó que instancia o gerador. Os parâmetros para configurar o gerador de tarefas são:

- $\sim max$: é um número inteiro cujo valor padrão é -1 que define quantidade de tarefas a serem geradas. Se for passado um valor negativo, serão geradas novas tarefas

enquanto o gerador estiver em execução;

- *~cycle_duration/mean*: é um número real cujo valor padrão é 5 que define a duração média dos intervalos entre a geração de tarefas;
- *~cycle_duration/standard_deviation*: é um número real cujo valor padrão é 1 que define o desvio padrão da duração dos intervalos entre a geração de tarefas;
- *~waypoints/mean*: é um número inteiro cujo valor padrão é 0 que define o número médio de pontos de passagem gerados nas tarefas;
- *~waypoints/standard_deviation*: é um número inteiro cujo valor padrão é 1 que define o desvio padrão do número de pontos de passagem gerados nas tarefas;
- *~waypoints/x/mean*: é um número real cujo valor padrão é 0 que define o valor médio na coordenada das abcissas dos pontos de passagem gerados nas tarefas;
- *~waypoints/x/standard_deviation*: é um número real cujo valor padrão é 1 que define o desvio padrão do valor na coordenada das abcissas dos pontos de passagem gerados nas tarefas;
- *~waypoints/y/mean*: é um número real cujo valor padrão é 0 que define o valor médio na coordenada das ordenadas dos pontos de passagem gerados nas tarefas;
- *~waypoints/y/standard_deviation*: é um número real cujo valor padrão é 1 que define o desvio padrão do valor na coordenada das ordenadas dos pontos de passagem gerados nas tarefas;
- *~features/size*: é um número real cujo valor padrão é 1 que define o número de características de tarefas;
- *~features/feature<index>/resource*: é uma cadeia de caracteres cujo valor padrão é uma cadeia vazia (‘’) que define o nome do recurso que a característica se referencia;
- *~features/feature<index>/type*: é um número inteiro cujo valor padrão é 0 que define o tipo do recurso da característica de índice *<index>*. Os valores válidos são: 0 para recursos unários, 1 para recursos discretos e 2 para recursos contínuos;
- *~features/feature<index>/level/mean*: utilizado apenas quando os recursos são discretos ou contínuos, é um número real cujo valor padrão é 0 que define o nível médio de intensidade requerida da característica de índice *<index>*;
- *~features/feature<index>/level/standard_deviation*: é um número real cujo valor padrão é 1 que define o desvio padrão do nível de intensidade requerida da característica de índice *<index>*;

- *~features/feature<index>/probability*: é um número real pertencente ao intervalo $[0; 1]$ cujo valor padrão é 1 que define a probabilidade da característica de índice *<index>* ser requisitada.

Os nomes dos parâmetros que contêm a etiqueta *<index>* devem substituí-la pelo índice da característica desejada.

Portanto, este gerador é utilizado para tornar o sistema estocástico. Deste modo, o mecanismo de leilão fica sujeito a momentos de saturação, mas também a situações de escassez. Em outras palavras, haverá momentos em que o leiloeiro será requisitado para leiloar tarefas além da sua capacidade, bem como, terá situações em que nenhum licitante estará disponível ou hábil para executar as tarefas. Logo, é possível verificar a robustez destes mecanismos.

4.1.1 Observador de Tarefas Geradas

Foi desenvolvido um observador do tópico por onde são enviadas as tarefas geradas para verificar o bom funcionamento do gerador de tarefas.

Este agente observa dados pertinentes para levantar dados estatísticos e probabilísticos sobre a geração de tarefas no sistema. Cada tipo de dado coletado é publicado em um tópico específico, os quais são:

- a duração entre as tarefas geradas, estes dados são publicados no tópico */analytics/cycles*;
- a quantidade de pontos de passagem criados durante a geração da tarefa, estes dados são publicados no tópico */analytics/waypoints*;
- o valor gerado para a coordenada das abcissas de um ponto de passagem criado, estes dados são publicados no tópico */analytics/waypoints/x*;
- o valor gerado para a coordenada das abcissas de um ponto de passagem criado, estes dados são publicados no tópico */analytics/waypoints/y*;
- verificação da utilização do recurso *<resource>*, estes dados são publicados no tópico */analytics/features/<resource>*;
- o nível de intensidade requirido durante a criação do recurso *<resource>*, estes dados são publicados no tópico */analytics/features/<resource>/level*.

Um novo par de publicadores dos tópicos */analytics/features/<resource>* e */analytics/features/<resource>/level* é criado cada fez que o agente observador de tarefas geradas identifica um novo recurso discreto ou contínuo. Entretanto, se o observador identifica um

novo recurso unário no sistema, este cria apenas um novo publicador do tópico */analytics/features/<resource>*. Nestes casos, a etiqueta *<resource>* é substituída pelo código identificador do recurso em questão.

4.1.2 Geração, Observação e Armazenamento de Tarefas

Cinco mil amostras de tarefas foram geradas e observadas para testar e validar o gerador randômico de tarefas. O gerador foi configurado para gerar tarefas permitindo a requisição dos seguintes recursos: (1) bateria (contínuo), (2) força (contínuo), (3) processador (discreto), (4) câmera (unário) e (5) *laserscan* (unário). Logo, o observador publicou dados para análise nos seguintes tópicos:

- */analytics/cycles*: dados aleatórios referentes à duração entre a geração de tarefas;
- */analytics/waypoints*: dados aleatórios referentes ao número de pontos de passagem gerados por tarefa;
- */analytics/waypoints/x*: dados aleatórios referentes ao valor da coordenada x dos pontos de passagem gerados;
- */analytics/waypoints/y*: dados aleatórios referentes ao valor da coordenada y dos pontos de passagem gerados;
- */analytics/features/battery*: trem de pulsos referente à requisição do recurso bateria;
- */analytics/features/battery/level*: dados aleatórios referentes à quantidade de bateria requisitada pelas tarefas geradas;
- */analytics/features/strength*: trem de pulsos referente à requisição do recurso força;
- */analytics/features/strength/level*: dados aleatórios referentes à capacidade de carga requisitada pelas tarefas geradas;
- */analytics/features/processor*: trem de pulsos referente à requisição do recurso processador;
- */analytics/features/processor/level*: dados aleatórios referentes ao número de processadores requisitados pelas tarefas geradas;
- */analytics/features/camera*: trem de pulsos referente à requisição do recurso câmera;
- */analytics/features/laserscan*: trem de pulsos referente à requisição do recurso *laserscan*;

As tarefas geradas foram publicadas pelo gerador no tópico */murdoch/task*. A Figura 21 mostra o esquema de comunicação no ROS dos nós que realizaram a geração, a observação e o armazenamento das tarefas criadas para testar os mecanismos de cálculo de utilidade e de negociação por leilão do *framework TAlMech*.

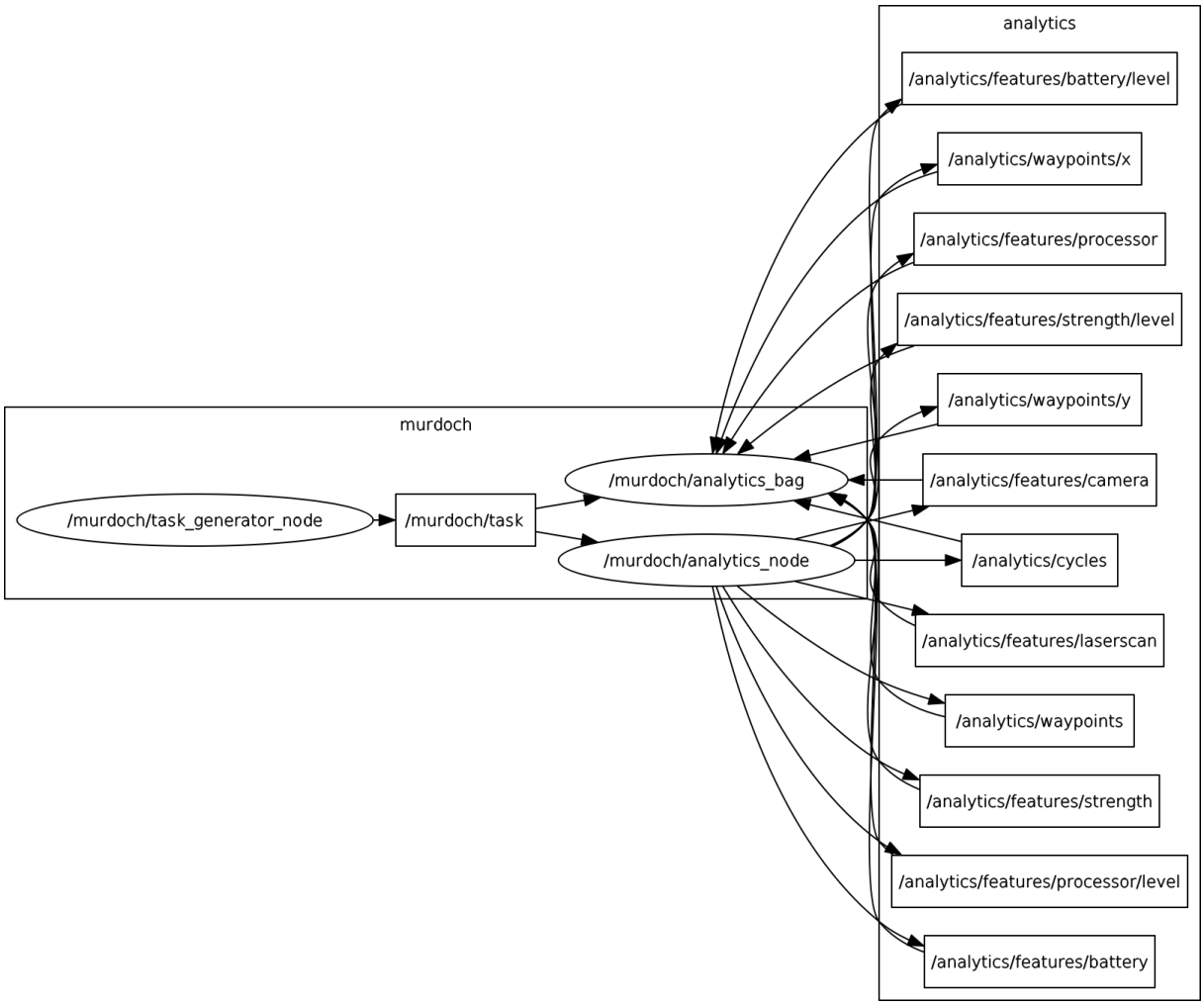


Figura 21 – Esquema de comunicação no ROS entre os nós de geração, observação e armazenamento de tarefas: os nós são representados por elipses e os tópicos são representados por retângulos.

O nó */murdoch/task_generator_node* é responsável por criar tarefas e publicá-las no tópico */murdoch/task* à medida que forem criadas. O nó */murdoch/analytics_node* é responsável por observar as tarefas que são recebidas do tópico */murdoch/task* e publicar suas observações pertinentes nos tópicos apropriados. O nó */murdoch/analytics_bag* é uma instância da ferramenta *rosbag*[1]. Esta ferramenta foi configurada para armazenar as tarefas criadas pelo gerador e as observações realizadas pelo observador em um arquio de extensão *bag* (vide Apêndice A).

As tarefas foram salvas para serem utilizadas no teste dos mecanismos de cálculo de utilidade e negociação por leilão do *framework TAlMech*. Deste modo, é possível comparar

[1] http://wiki.ros.org/rosbag

o desempenho de uma arquitetura ao variar seus parâmetros ou de arquiteturas distintas. Estes testes serão realizados no decorrer deste capítulo.

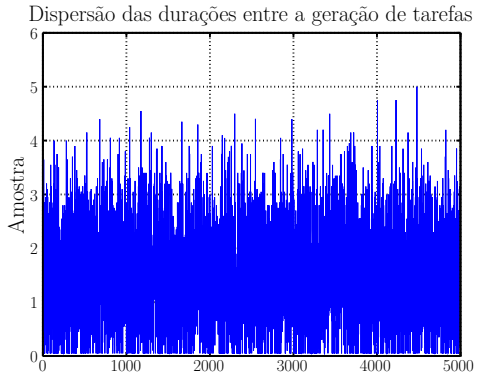
Porém, as observações foram armazenadas para a análise do gerador de tarefas com o propósito de validar o seu funcionamento. Para isso, o arquivo *bag* armazenado foi convertido em múltiplos arquivos de extensão *csv*[2]. De modo que as mensagens publicadas em cada tópico para análise foram convertidas e armazenadas em um arquivo *csv*. A partir desses arquivos *csv*, foram elaborados os gráficos expostos nas Figuras 22, 23, 24, 25, 26 e 27.

A Figura 22 mostra a dispersão e o histograma gerados a partir dos dados observados e publicados no tópico */analytics/cycles* pelo nó observador de tarefas. Cada amostra equivale à duração entre o recebimento de duas tarefas. O gráfico de dispersão da Figura 22a mostra que, aproximadamente, 5000 amostras foram coletadas (exatamente 4998 amostradas) e que a distribuição destas amostras se assemelha a uma distribuição normal. Durante a inicialização do gerador randômico de tarefas, a média e o desvio padrão da duração entre a geração das tarefas foram configurados para serem $1,50$ e $1,00$, respectivamente. Entretanto, uma distribuição normal de média $1,54$ e desvio padrão $0,93$ foi observada. Graficamente, é possível verificar que a distribuição observada (linha tracejada preta na Figura 22b) se aproxima da distribuição desejada (linha cheia vermelha na Figura 22b).

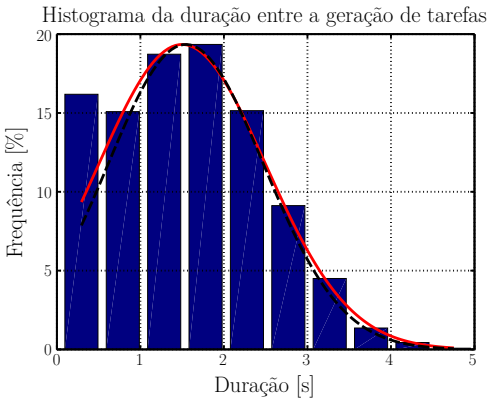
A Figura 23 mostra a dispersão e o histograma gerados a partir dos dados observados e publicados no tópico */analytics/waypoints* pelo nó observador de tarefas. Cada amostra equivale ao número de pontos de passagens que foram geradas em uma dada tarefa. O gráfico de dispersão da Figura 23a mostra que 5000 amostras foram coletadas e que a distribuição destas amostras se assemelha a uma distribuição normal. Durante a inicialização do gerador randômico de tarefas, a média e o desvio padrão do número de pontos de passagem por tarefa foram configurados para serem $4,00$ e $2,00$, respectivamente. Entretanto, uma distribuição normal de média $3,99$ e desvio padrão $1,97$ foi observada. Graficamente, é possível verificar que a distribuição observada (linha tracejada preta na Figura 23b) se aproxima da distribuição desejada (linha cheia vermelha na Figura 23b).

A Figura 24 mostra as dispersões e os histogramas gerados a partir dos dados observados e publicados nos tópicos */analytics/waypoints/x* (esquerda) e */analytics/waypoints/x* (direita) pelo nó observador de tarefas. Cada amostra equivale ao valor da coordenada das abcissas (esquerda) e das ordenadas (direita) de um ponto de passagem

[2] CSV (*Comma-Separated Values*) é um formato simples de armazenamento, que agrupa as informações de arquivos de texto em planilhas. Cada linha em um texto CSV representa uma linha em uma planilha. Cada célula é geralmente separada por vírgula ou um outro caractere, como tabulador (Fonte: <https://ajuda.rdstation.com.br/hc/pt-br/articles/205623999-Como-eu-crio-um-arquivo-CSV->, Acessada em: 3 de Março de 2018.).



(a) Dispersão da duração entre a geração das tarefas.

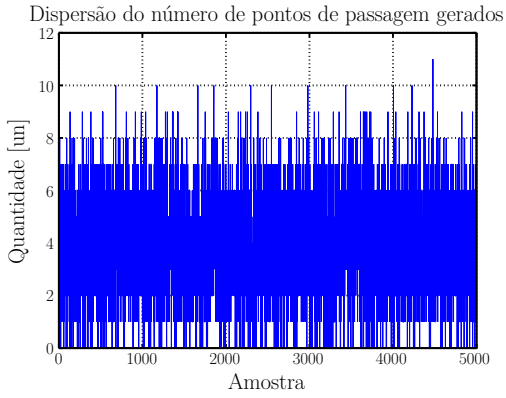


(b) Histograma da duração entre a geração das tarefas: as barras azuis mostram a frequência em que uma faixa de valor de duração ocorre, a linha cheia vermelha representa a distribuição normal desejada $X \sim \mathcal{N}(1,50; 1,00)$ e a linha tracejada preta representa a distribuição normal dos dados coletados $X \sim \mathcal{N}(1,54; 0,93)$.

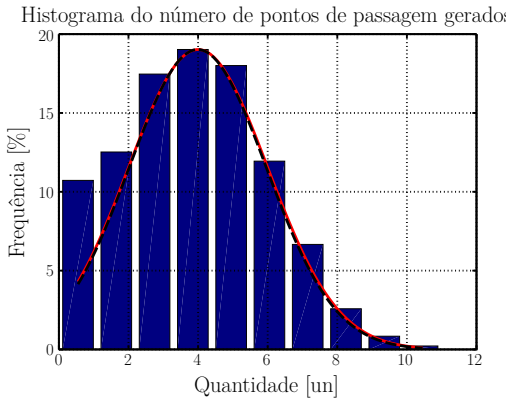
Figura 22 – Dispersão e histograma da duração entre as tarefas geradas.

gerado em uma dada tarefa. Os gráficos de dispersão das Figuras 24a e 24b mostram que, aproximadamente, 20000 amostras foram coletadas (exatamente 19950) e que a distribuição destas amostras se assemelha a uma distribuição normal. Durante a inicialização do gerador randômico de tarefas, a média do valor de ambas coordenadas dos pontos de passagem foi configurada para ser 0.00, enquanto o desvio padrão das coordenadas das abcissas e das ordenadas foi configurado para ser $10,00$ e $15,00$, respectivamente. Entretanto, uma distribuição normal com média $-0,10$ e desvio padrão $10,04$ foi observada para as coordenadas das abcissas e uma com média $-0,15$ e desvio padrão $15,06$ para as coordenadas das ordenadas. Graficamente, é possível verificar que, nos dois casos, a distribuição observada (linha tracejada preta nas Figuras 24c e 24d) se aproxima da distribuição desejada (linha cheia vermelha nas Figuras 24c e 24d).

A Figura 25 mostra a dispersão e o histograma gerados a partir dos dados ob-



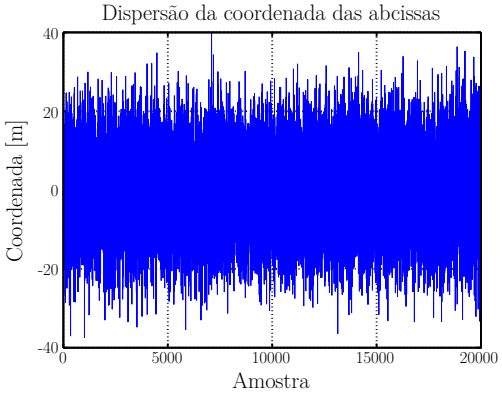
(a) Dispersão do número de pontos de passagem gerados por tarefa.



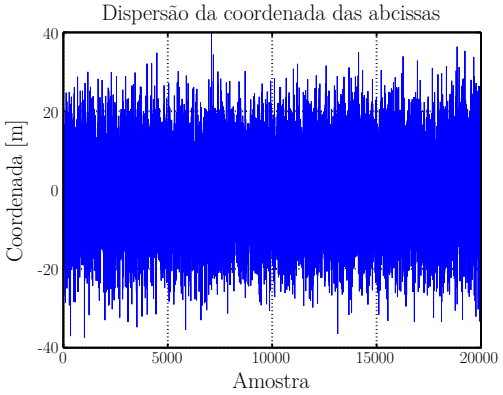
(b) Histograma do número de pontos de passagem gerados por tarefa: as barras azuis mostram a frequência em que uma faixa de quantidade ocorre, a linha cheia vermelha representa a distribuição normal desejada $X \sim \mathcal{N}(4,00; 2,00)$ e a linha tracejada preta representa a distribuição normal dos dados coletados $X \sim \mathcal{N}(3,99; 1,97)$.

Figura 23 – Dispersão e histograma do número de pontos de passagem gerados por tarefa.

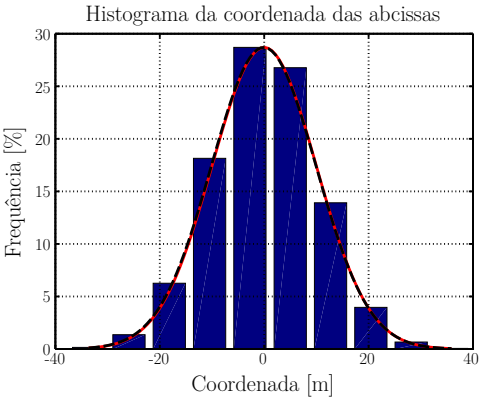
servados e publicados no tópico */analytics/features/battery/level* pelo nó observador de tarefas. Cada amostra equivale à quantidade de bateria demandada por uma dada tarefa. O gráfico de dispersão da Figura 25a mostra que, aproximadamente, 3500 amostras foram coletadas (exatamente 3504) e que a distribuição destas amostras se assemelha a uma distribuição normal. Durante a inicialização do gerador randômico de tarefas, a média e o desvio padrão da quantidade de bateria demandada para uma tarefa foram configurados para serem $0,60$ e $0,20$, respectivamente. De fato, uma distribuição normal de média $0,60$ e desvio padrão $0,20$ foi observada. Graficamente, é possível verificar que a distribuição observada (linha tracejada preta na Figura 25b) se aproxima da distribuição desejada (linha cheia vermelha na Figura 25b). Ademais, a probabilidade do recurso bateria ser requisitado foi configurada como $70,0\%$. Como resultado, observa-se que, pelo número de amostras coletadas, o recurso bateria foi requisitado $70,1\%$ das vezes em que uma tarefa



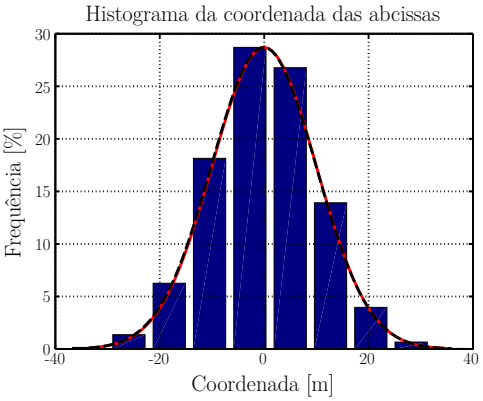
(a) Dispersão da coordenada das abcissas dos pontos de passagem gerados.



(b) Dispersão da coordenada das ordenadas dos pontos de passagem gerados.



(c) Histograma da coordenada das abcissas dos pontos de passagem gerados: as barras azuis mostram a frequência em que uma faixa de valor ocorre, a linha cheia vermelha representa a distribuição normal desejada $X \sim \mathcal{N}(0,00; 10,00)$ e a linha tracejada preta representa a distribuição normal dos dados coletados $X \sim \mathcal{N}(-0,10; 10,04)$.

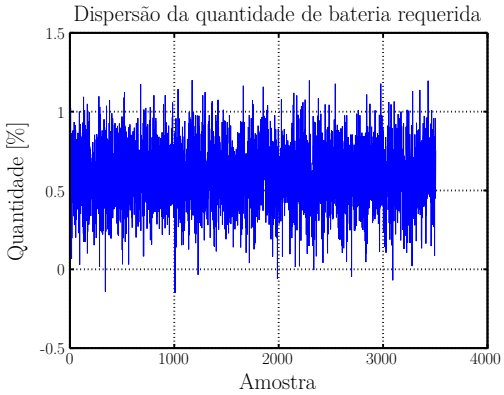


(d) Histograma da coordenada das ordenadas dos pontos de passagem gerados: as barras azuis mostram a frequência em que uma faixa de valor ocorre, a linha cheia vermelha representa a distribuição normal desejada $X \sim \mathcal{N}(0,00; 15,00)$ e a linha tracejada preta representa a distribuição normal dos dados coletados $X \sim \mathcal{N}(-0,15; 15,06)$.

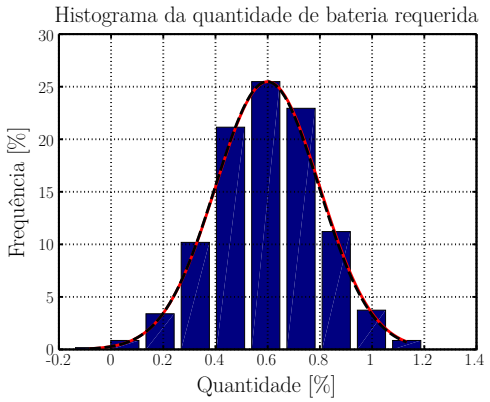
Figura 24 – Dispersão e histograma da coordenadas das abcissas (esquerda) e das ordenadas (direita) dos pontos de passagem gerados.

foi gerada.

A Figura 26 mostra a dispersão e o histograma gerados a partir dos dados observados e publicados no tópico */analytics/features/strength/level* pelo nó observador de tarefas. Cada amostra equivale à intensidade de força demandada por uma dada tarefa. O gráfico de dispersão da Figura 26a mostra que, aproximadamente, 2500 amostras foram coletadas (exatamente 2476) e que a distribuição destas amostras se assemelha a uma distribuição normal. Durante a inicialização do gerador randômico de tarefas, a média e o desvio padrão da intensidade de força demandada para uma tarefa foram configurados para serem $5,60$ e $3,40$, respectivamente. Entretanto, uma distribuição normal de média $5,57$ e desvio padrão $3,39$ foi observada. Graficamente, é possível verificar que a



(a) Dispersão da quantidade de bateria requerida por tarefa.

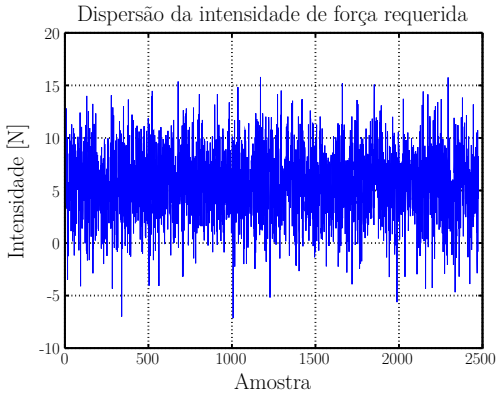


(b) Histograma da quantidade de bateria requerida por tarefa: as barras azuis mostram a frequência em que uma faixa de quantidade ocorre, a linha cheia vermelha representa a distribuição normal desejada $X \sim \mathcal{N}(0,60; 0,20)$ e a linha tracejada preta representa a distribuição normal dos dados coletados $X \sim \mathcal{N}(0,60; 0,20)$.

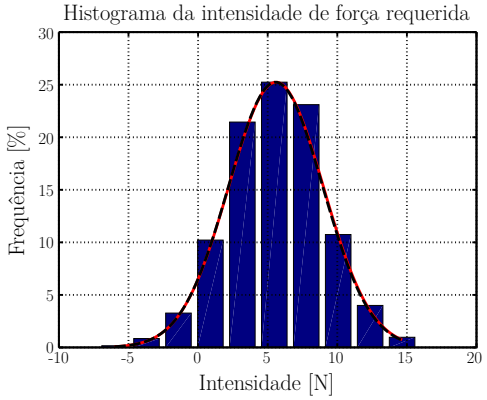
Figura 25 – Dispersão e histograma da quantidade de bateria requerida por tarefa.

distribuição observada (linha tracejada preta na Figura 26b) se aproxima da distribuição desejada (linha cheia vermelha na Figura 26b). Ademais, a probabilidade do recurso força ser requisitado foi configurada como $50,0\%$. Como resultado, observa-se que, pelo número de amostras coletadas, o recurso força foi requisitado $49,5\%$ das vezes em que uma tarefa foi gerada.

A Figura 27 mostra a dispersão e o histograma gerados a partir dos dados observados e publicados no tópico */analytics/features/processor/level* pelo nó observador de tarefas. Cada amostra equivale ao número de processadores demandados por uma dada tarefa. O gráfico de dispersão da Figura 27a mostra que, aproximadamente, 3000 amostras foram coletadas (exatamente 3002) e que a distribuição destas amostras se assemelha a uma distribuição normal. Durante a inicialização do gerador randômico de tarefas, a média e o desvio padrão da intensidade de força demandada para uma tarefa foram con-



(a) Dispersão da intensidade de força requerida por tarefa.

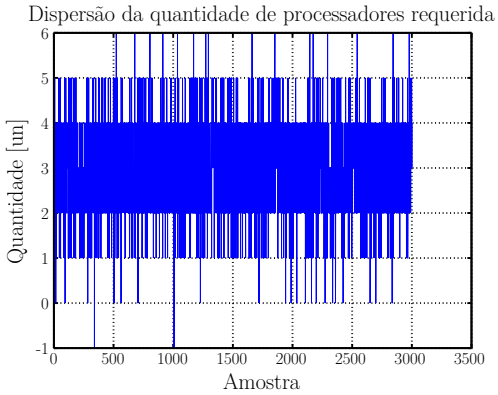


(b) Histograma da intensidade de força requerida por tarefa: as barras azuis mostram a frequência em que uma faixa de intensidade ocorre, a linha cheia vermelha representa a distribuição normal desejada $X \sim \mathcal{N}(5,60;3,40)$ e a linha tracejada preta representa a distribuição normal dos dados coletados $X \sim \mathcal{N}(5,57;3,39)$.

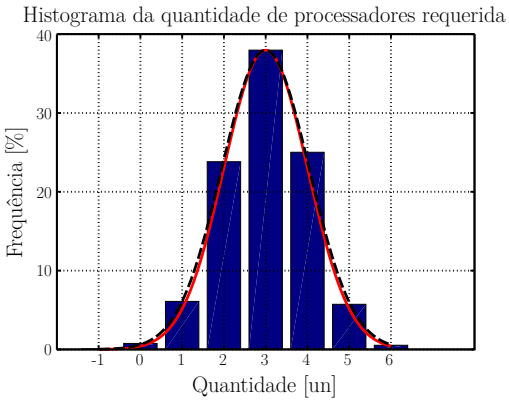
Figura 26 – Dispersão e histograma da intensidade de força requerida por tarefa.

figurados para serem 3 e 1, respectivamente. De fato, uma distribuição normal de média 3 e desvio padrão 1 foi observada. Graficamente, é possível verificar que a distribuição observada (linha tracejada preta na Figura 27b) se aproxima da distribuição desejada (linha cheia vermelha na Figura 27b). Ademais, a probabilidade do recurso processador ser requisitado foi configurada como $60,0\%$. Como resultado, observa-se que, pelo número de amostras coletadas, o recurso processador foi requisitado $60,1\%$ das vezes em que uma tarefa foi gerada.

Além disso, as probabilidades dos recursos câmera e *laserscan* serem requisitados na geração de uma tarefa foram configuradas para serem $75,0\%$ e $35,0\%$ durante a inicialização do gerador randômico de tarefas. Contudo, observa-se que, pelo número de amostras coletadas, os recursos câmera e *laserscan* foram requisitados $75,1\%$ e $34,8\%$, respectivamente, das vezes em que uma tarefa foi gerada.



(a) Dispersão da quantidade de processadores requerida por tarefa.



(b) Histograma da quantidade de processadores requerida por tarefa: as barras azuis mostram a frequência em que uma faixa de quantidade ocorre, a linha cheia vermelha representa a distribuição normal desejada $X \sim \mathcal{N}(3;1)$ e a linha tracejada preta representa a distribuição normal dos dados coletados $X \sim \mathcal{N}(3;1)$.

Figura 27 – Dispersão e histograma da quantidade de processadores requerida por tarefa.

Enfim, verifica-se que o gerador randômico de tarefas trabalhou segundo os parâmetros configurados. Isto é, a duração entre a geração das tarefas, o número de pontos de passagem gerados por tarefa, os valores das suas coordenadas, a quantidade de bateria requerida, a capacidade de carga requisitada e o número de processadores foram dispersos segundo a sua distribuição normal configurada. Bem como, os recursos foram requeridos pelas tarefas segundo a probabilidade configurada de cada um deles.

4.2 Mecanismos de Cálculo de Utilidade e Negociação por Leilão

Esta seção caracteriza os quatro ensaios realizados para testar e validar os mecanismos de cálculo de utilidade e de negociação por leilão do *framework TAlMech*. Em um dos ensaios, o mecanismo de negociação por leilão é configurado conforme as características estabelecidas na arquitetura Murdoch.

4.2.1 Murdoch

O Murdoch é uma das arquiteturas revisadas em 2.4.2. Esta arquitetura faz bom proveito do *framework TAlMech*. Tanto o mecanismo de cálculo de utilidade quanto o de negociação por leilão são utilizados nesta arquitetura baseada em negociação por leilão. Torna-se apenas necessário desenvolver os nós que (1) encapsulam os agentes que desempenharão os papéis de leiloeiro e licitante, (2) configuram os componentes do cálculo de utilidade e (3) mantêm o nível disponível de cada recurso de um dado agente atualizado.

Para que o mecanismo de negociação por leilão do *framework TAlMech* funcione conforme o Murdoch, é necessário configurar os papéis leiloeiro e licitante como segue. Na arquitetura Murdoch, leiloeiros não realocam tarefas e não permitem que licitantes ressubmetam um novo lance durante o período de submissão de lances. Ademais, licitantes podem executar apenas uma tarefa por vez no Murdoch.

4.2.2 Ensaios

Os mecanismos de cálculo de utilidade e negociação por leilão do *framework TAlMech* foram testados com as 5000 tarefas geradas pelo gerador randômico e, posteriormente, armazenadas em um arquivo *bag*, conforme descrito em 4.1.2. Este arquivo *bag* permitiu reutilizar as mesmas tarefas em cada teste dos mecanismos de cálculo de utilidade e negociação por leilão. Em todos ensaios, as tarefas foram publicadas com as mesmas taxas e características. Assim é possível comparar os ensaios e tirar conclusão a respeito das configurações alteradas.

O mesmo sistema multirrobô heterogêneo foi utilizado em todos os ensaios: um leiloeiro e quatro licitantes. Em todos os casos, os licitantes apresentaram a mesma configuração para o cálculo de utilidade e as mesmas características. A Tabela 2 mostra as configurações iniciais dos licitantes. Percebe-se que as configurações dos licitantes *bidder2*, *bidder3* e *bidder4* são muito parecidas. A maior diferença entre eles está na configuração dos fatores de correção no cálculo de utilidade.

Perceba que todos licitantes utilizam a mesma decoração de cálculo de utilidade. Por fim, cada licitante possui um conjunto de características que são usados no cálculo de utilidade para comparar com as características requeridas por cada tarefa. A decoração de cálculo de utilidade configurada para todos os licitantes faz com que os licitantes levem em consideração sua posição e os pontos de passagem dados pela tarefa para calcular a distância percorrida, bem como, compara todas as características exigidas pela tarefa com as suas para verificar se pode atender todos os requisitos da tarefa.

A execução das tarefas foi simulada em um nó separado para cada licitante. Para isso, a média, o desvio padrão da duração das tarefas e a probabilidade de falha de suas execuções foram configurados para o nó simulador de execução de tarefa de cada licitante,

Tabela 2 – Configuração inicial dos licitantes

Licitante	**Multi-tarefa**	**Utilidade**	**Características**		
			Intensidade	**Recurso**	**Fator**
bidder1	não	"distance feature"	- 70,0% 1 [un]	câmera bateria processador	0,4 1,8 3,2
bidder2	não	"distance feature"	- 85,0% 1 [un] - 4,6 [N]	câmera bateria processador *laserscan* força	5,0 1,0 7,0 0,4 5,4
bidder3	não	"distance feature"	- 85,0% 1 [un] - 4,6 [N]	câmera bateria processador *laserscan* força	1,2 1,0 2,0 0,4 5,4
bidder4	não	"distance feature"	- 50,0% 1 [un] - 4,6 [N]	câmera bateria processador *laserscan* força	5,0 1,0 4,0 0,4 8,0

conforme os dados da Tabela 3. Ao final da execução de uma tarefa, seu simulador atualiza

Tabela 3 – Configuração dos simuladores de execução de tarefa.

Licitante	**Média da duração das tarefas [s]**	**Desvio padrão da duração das tarefas [s]**	**Probabilidade de falha da execução das tarefas**
bidder1	5,0	2,5	15,0%
bidder2	3,0	1,5	20,0%
bidder3	3,0	2,5	5,0%
bidder4	6,0	4,0	35,0%

a posição do robô para o último ponto de passagem da tarefa.

Cada ensaio de leilão foi observado por um nó responsável por contabilizar cada ação possível dos leiloeiros e licitantes participantes. De modo que, ao final do ensaio, o número de leilões, realocações, contratos firmados, abortados e concluídos de cada leiloeiro, bem como, o número de submissões enviadas, contratos firmados, abortados e concluídos de cada licitante são obtidos.

A partir da ferramenta *rqt_graph* (vide Apêndice A), se obteve a Figura 28, a qual mostra o esquema de comunicação entre os nós utilizados em cada ensaio de leilão no ROS. Cada licitante possui seu próprio *namespace* sobre o qual rodam os nós */<bidder id>/murdoch/bidder_node* e */<bidder id>/murdoch/task_executor_node*. O nó */<bidder id>/murdoch/bidder_node* encapsula o papel de um licitante cujo código de identificação é *<bidder id>*. O nó */<bidder id>/murdoch/task_executor_node* simula a execução das tarefas que o seu respectivo licitante fechar contrato. Ao final de cada tarefa, este publica a nova posição do robô que executou a tarefa.

O nó */report_node* observa as atividades dos agentes ao desempenhar seus papéis no leilão. A partir de suas observações, é gerado um relatório. O leiloeiro possui seu próprio *namespace* (*/auctioneer1*), onde é executado o nó */auctioneer1/murdoch/auctioneer_node*. Este nó encapsula um agente com o papel de um leiloeiro, o qual recebe as tarefas pelo tópico */murdoch/task* para leiloar. As tarefas são publicadas pelo nó */play_<start timestamp>* no tópico */murdoch/task*.

O Ensaio 1 foi realizado com um leiloeiro com as seguintes configurações:

- **Duração do leilão**: $0,5$ [s];
- **Taxa de renovação**: $1,5$ [Hz];
- **Inserção Ordenada**: Não
- **Realocação**: Não
- **Atualização de Lance**: Não
- **Tamanho da pilha de tarefas**: 20
- **Número de leilões simultâneos**: 5

As configurações do leiloeiro e dos licitantes utilizadas neste ensaio tornam o funcionamento do mecanismo de leilão do *TAlMech* equivalente à arquitetura Murdoch.

Após a execução do Ensaio 1, o relatório das atividades do leiloeiro e dos licitantes foi obtido. O leiloeiro *auctioneer1* leiloou 4996 leilões, o que equivale à $99,9\%$ das tarefas geradas. Não houve realocação. $42,8\%$ das tarefas leiloadas se tornaram contratos, isto é, 2135 contratos foram firmados entre o leiloeiro *auctioneer1* e os licitantes do sistema; dos quais 370 foram abortadas e 1765 foram concluídas com sucesso, ou seja, $17,3\%$ dos contratos foram abortados enquanto que $82,7\%$ deles foram concluídos. A Tabela 4 relata as atividades dos robôs licitantes no sistema.

Nota-se que, apesar dos licitantes participarem consideravelmente de uma mesma quantia de leilões, a maioria dos contratos foram firmados com o licitante *bidder3*, fincando

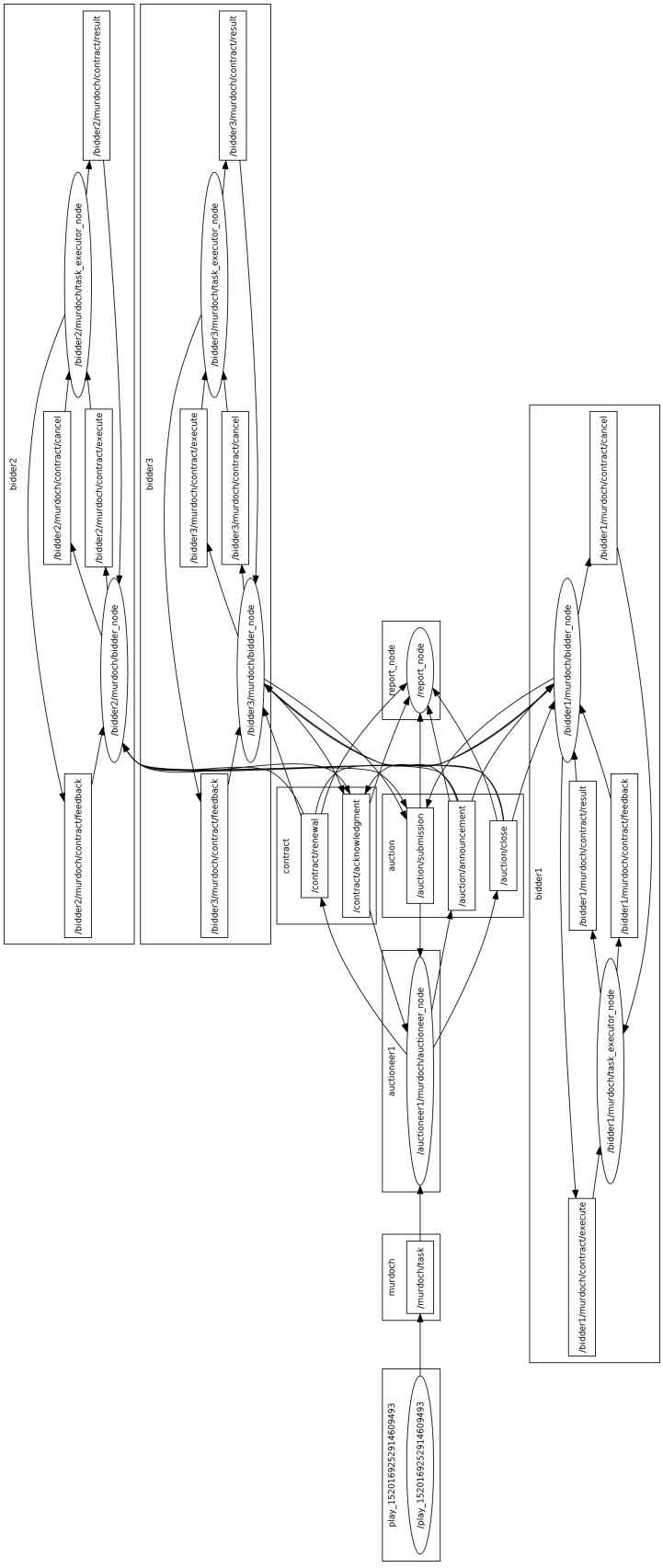


Figura 28 – Esquema de comunicação no ROS entre os nós dos licitantes, dos simuladores de execução, do leiloeiro e do observador.

atrás deste, o licitante *bidder2*. Conforme foi configurado na simulação, o licitante *bidder3* possui a maior taxa de contratos bem sucedidos. A taxa de participação dos licitantes em leilões pode estar atrelada à indisponibilidade dos licitantes durante o leilão ou à falta de destreza para executar a tarefa. Lembrando que os licitantes têm que ofertar pelo menos o valor de reserva da tarefa para poder executá-la.

Tabela 4 – Relatório das atividades dos licitantes durante o Ensaio 1.

Licitante	**Submissões**[*]	**Contratos**[**]	**Contratos Abortados**[***]	**Contratos Concluídos**[***]
bidder1	1133 (22, 7%)	337 (29, 7%)	54 (16, 0%)	283 (84, 0%)
bidder2	1297 (26, 0%)	623 (48, 0%)	135 (21, 7%)	488 (78, 3%)
bidder3	1297 (26, 0%)	766 (59, 1%)	37 (4, 8%)	729 (95, 2%)
bidder4	1066 (21, 3%)	409 (38, 4%)	144 (35, 2%)	265 (64, 8%)

[*] Porcentagem referente ao número de tarefas leiloadas.
[**] Porcentagem referente ao número de lances submetidos.
[***] Porcentagem referente ao número de contratos firmados.

O Ensaio 2 foi realizado com um leiloeiro com as seguintes configurações:

- **Duração do leilão**: 0, 5 [s];
- **Taxa de renovação**: 0, 5 [Hz];
- **Inserção Ordenada**: Sim
- **Realocação**: Sim
- **Atualização de Lance**: Não
- **Tamanho da pilha de tarefas**: 20
- **Número de leilões simultâneos**: 10

Após a execução do Ensaio 2, o relatório das atividades do leiloeiro e dos licitantes foi obtido. O leiloeiro *auctioneer1* leiloou 4999 leilões, o que equivale à 100, 0% das tarefas geradas. Houve 381 realocações, o que equivale à 7, 6% das tarefas leiloadas. 42, 9% das tarefas leiloadas se tornaram contratos, isto é, 2142 contratos foram firmados entre o leiloeiro *auctioneer1* e os licitantes do sistema; dos quais 381 foram abortadas e 1761 foram concluídas com sucesso, ou seja, 17, 8% dos contratos foram abortados enquanto que 82, 2% deles foram concluídos. A Tabela 5 relata as atividades dos robôs licitantes no sistema.

Houve uma pequena variação na participação dos licitantes em leilões neste ensaio. Isto era previsto, já que a taxa de entrada de tarefas no sistema era a mesma com o mesmo conjunto de amostras. A pequena variação existente se deve ao fato da duração simulada das execuções de tarefa ser aleatória. Contudo, houve um aumento no número de contratos firmados com os licitantes *bidder2* e *bidder4*. Consequentemente, houve um decréscimo na quantidade de contratos fechados com o licitante *bidder3*.

Tabela 5 – Relatório das atividades dos licitantes durante o Ensaio 2.

Licitante	**Submissões**[*]	**Contratos**[**]	**Contratos Abortados**[***]	**Contratos Concluídos**[***]
bidder1	1122 (22,4%)	283 (25,2%)	44 (15,5%)	239 (84,5%)
bidder2	1320 (26,4%)	687 (52,0%)	148 (21,5%)	539 (78,5%)
bidder3	1279 (25,6%)	725 (56,7%)	35 (4,8%)	690 (95,2%)
bidder4	1070 (21,4%)	447 (41,8%)	154 (34,5%)	293 (65,5%)

[*] Porcentagem referente ao número de tarefas leiloadas.
[**] Porcentagem referente ao número de lances submetidos.
[***] Porcentagem referente ao número de contratos firmados.

No Ensaio 3, os licitantes *bidder1* e *bidder4* são configurados para executar respectivamente três e duas tarefas ao mesmo tempo (robô de múltiplas tarefas *MT*), enquanto os demais licitantes podem executar apenas uma tarefa por vez (robôs de um tarefa *ST*). O *framework TAlMech* não trata alocação de uma tarefa para mais de um robô (tarefas de múltiplos robôs *MR*), isto é, cada tarefa só pode ser alocada para um único robô (tarefas de um robô *SR*). Além disso, o *framework* não possui recursos de escalonamento (*TA*). Logo, as alocações são instantâneas (*IA*). Neste caso, segundo a taxonomia de (GERKEY; MATARIĆ, 2004), o mecanismo de negociação por leilão do *framework TAlMech* é capaz de tratar problemas de alocação de tarefa em sistemas multirrobô dos tipos *ST-SR-IA* e *MT-SR-IA*.

Além disso, o Ensaio 3 foi realizado com um leiloeiro com as seguintes configurações:

- **Duração do leilão**: 0,5 [s];
- **Taxa de renovação**: 1,5 [Hz];
- **Inserção Ordenada**: Não
- **Realocação**: Não
- **Atualização de Lance**: Não

- **Tamanho da pilha de tarefas**: 20
- **Número de leilões simultâneos**: 8

Após a execução do Ensaio 3, o relatório das atividades do leiloeiro e dos licitantes foi obtido. O leiloeiro *auctioneer1* leiloou 5000 leilões, o que equivale à $100,0\%$ das tarefas geradas. Não houve realocação. $62,7\%$ das tarefas leiloadas se tornaram contratos, isto é, 3136 contratos foram firmados entre o leiloeiro *auctioneer1* e os licitantes do sistema; dos quais 659 foram abortadas e 2477 foram concluídas com sucesso, ou seja, $21,0\%$ dos contratos foram abortados enquanto que $79,0\%$ deles foram concluídos. A Tabela 6 relata as atividades dos robôs licitantes no sistema.

Houve uma aumento significativo na participação dos licitantes *bidder1* e *bidder4* neste ensaio. O número de submissões do *bidder1* quase triplicou, enquanto o número de submissões do *bidder4* duplicou. Dado que o aumento na participação de ambos licitantes foi proporcional ao aumento de capacidade de execução de tarefas simultâneas, o maior motivo dos licitantes não participarem dos leilões se dá pelo fato de não haver disponibilidade da parte deles para executar a tarefa durante o processo de leilão. Este argumento é reforçado ao analisar a Tabela 3 e verificar que a média da duração de execução das tarefas dos robôs foi configurada entre duas a quatro vezes a mais do que a taxa média de entrada de tarefa no sistema. Ou seja, o número de tarefas requisitadas no sistema é maior do que o número de robôs disponíveis para executá-las.

Tabela 6 – Relatório das atividades dos licitantes durante o Ensaio 3.

Licitante	**Submissões**[*]	**Contratos**[**]	**Contratos Abortados**[***]	**Contratos Concluídos**[***]
bidder1	2970 (59,4%)	1675 (56,4%)	260 (15,5%)	1415 (84,5%)
bidder2	1177 (23,6%)	278 (23,6%)	61 (21,9%)	217 (78,1%)
bidder3	1153 (23,1%)	267 (23,2%)	15 (5,6%)	252 (94,4%)
bidder4	2026 (40,5%)	916 (45,2%)	323 (35,2%)	593 (64,7%)

[*] Porcentagem referente ao número de tarefas leiloadas.
[**] Porcentagem referente ao número de lances submetidos.
[***] Porcentagem referente ao número de contratos firmados.

No Ensaio 4, todos os agentes licitantes foram configurados para poderem executar até duas tarefas ao mesmo tempo. Ademais, este ensaio foi realizado com um leiloeiro com as seguintes configurações:

- **Duração do leilão**: $0,5$ [s];
- **Taxa de renovação**: $1,5$ [Hz];

- **Inserção Ordenada**: Não
- **Realocação**: Sim
- **Atualização de Lance**: Não
- **Tamanho da pilha de tarefas**: 20
- **Número de leilões simultâneos**: 8

Após a execução do Ensaio 4, o relatório das atividades do leiloeiro e dos licitantes foi obtido. O leiloeiro *auctioneer1* leiloou 4999 leilões, o que equivale à $100,0\%$ das tarefas geradas. Houve 524 realocações, o que equivale à $10,5\%$ das tarefas leiloadas. $57,9\%$ das tarefas leiloadas se tornaram contratos, isto é, 2892 contratos foram firmados entre o leiloeiro *auctioneer1* e os licitantes do sistema; dos quais 524 foram abortadas e 2368 foram concluídas com sucesso, ou seja, $18,1\%$ dos contratos foram abortados enquanto que $81,9\%$ deles foram concluídos. A Tabela 7 relata as atividades dos robôs licitantes no sistema.

Percebe-se novamente que o aumento da capacidade de tarefas que podem ser executadas simultaneamente pelos robôs gerou um aumento proporcional no número de submissões realizadas por cada licitante. Neste ensaio, houve uma diminuição significativa do número de contratos firmados com o licitante *bidder1*, enquanto que o *bidder3* voltou a ter o maior número de contratos realizados.

Tabela 7 – Relatório das atividades dos licitantes durante o Ensaio 4.

Licitante	**Submissões**[*]	**Contratos**[**]	**Contratos Abortados**[***]	**Contratos Concluídos**[***]
bidder1	2106 $(42,1\%)$	324 $(15,4\%)$	51 $(15,7\%)$	273 $(84,3\%)$
bidder2	2321 $(46,4\%)$	910 $(39,2\%)$	194 $(21,3\%)$	716 $(78,7\%)$
bidder3	2274 $(45,5\%)$	989 $(43,5\%)$	44 $(4,4\%)$	945 $(95,6\%)$
bidder4	2049 $(41,0\%)$	669 $(32,7\%)$	235 $(35,1\%)$	434 $(64,9\%)$

[*] Porcentagem referente ao número de tarefas leiloadas.
[**] Porcentagem referente ao número de lances submetidos.
[***] Porcentagem referente ao número de contratos firmados.

Note que em todos os ensaios a taxa de falha na execução de tarefas é bem próxima da probabilidade configurada para cada robô licitante. Além disso, a soma dos contratos firmados por cada licitante coincide com o número de contratos firmados pelo leiloeiro em todos os ensaios.

Ao comparar o desempenho dos licitantes em cada ensaio, nota-se que o licitante *bidder3* possui maior característica do que os outros licitantes para solucionar o conjunto

Tabela 8 – Comparação entre os ensaios.

Ensaio	**Realocações**[*]	**Submissões**	**Contratos**[*]	**Contratos Abortados**[**]	**Contratos Concluídos**[**]
1	381 (7,6%)	4791	2142 (44,7%)	381 (17,8%)	1761 (82,2%)
2	0 (0,0%)	4793	2135 (42,8%)	370 (17,3%)	1765 (82,7%)
3	0 (0,0%)	7326	3136 (62,7%)	659 (21,0%)	2477 (79,0%)
4	524 (10,5%)	8750	2892 (57,9%)	524 (18,1%)	2368 (81,9%)

[*] Porcentagem referente ao número de tarefas leiloadas.
[**] Porcentagem referente ao número de contratos firmados.

de tarefas ensaiadas. Logo atrás dele está o licitante *bidder2* que possui configurações muito próximas do *bidder3*. Na sequência está o *bidder4* e depois o *bidder1* que obteve um desempenho bem inferior aos demais. Com exceção do Ensaio 3 onde a capacidade de execução do *bidder1* foi muito superior aos demais.

Portanto, verifica-se que o mecanismo de cálculo de utilidade do *TAlMech* é facilmente configurado a partir de uma cadeia de caracteres na inicialização dos nós que encapsulam a função de licitante para os agentes do sistema no ROS. Os componentes de cálculo de utilidade fornecidos pelo *framework TAlMech* auxiliaram a quantificar a aptidão de cada licitante para executar o conjunto de tarefas amostradas sempre quando disponível.

Além disso, verifica-se que o mecanismo de negociação por leilão do *TAlMech* pode ser configurado na inicialização dos nós que encapsulam a função de leiloeiro para os agentes do sistema no ROS. Este mecanismo ainda permite o tratamento de problemas de alocação de tarefas em sistemas multirrobô que envolvem robôs multi-tarefas. Logo, o mecanismo de negociação por leilão do *TAlMech* pode ser utilizados por arquiteturas MRTA que resolvem problemas *ST-SR-IA* e *MT-SR-IA*.

4.3 Mecanismos de Monitoramento e Controle de Ativação de Comportamento

Esta seção apresenta como os mecanismos de monitoramento e controle de ativação de comportamento do *framework TAlMech* são utilizados para implementar uma arquitetura baseada em comportamento. Primeiramente, é feita uma comparação entre os modelos de duas aproximações desta arquitetura: uma que não utiliza o *framework*

TAlMech e outra que o utiliza. Em seguida, é feito um teste da aproximação que utiliza o *TAlMech* em uma aplicação de patrulhamento realizada por múltiplos robôs.

4.3.1 ALLIANCE

O ALLIANCE é uma das arquiteturas revisadas em 2.4.1. Esta arquitetura faz bom proveito do *framework TAlMech*. A função de ativação de comportamento do ALLIANCE leva em consideração as atividades relacionadas a tarefa associada ao comportamento em questão. Para isso, pode-se utilizar o mecanismo de monitoramento de comportamento na implementação dos componentes da função de ativação dos comportamentos no ALLIANCE. O mecanismo seria responsável por identificar as atividades dos demais agentes no sistema ao longo do tempo e registrá-las em um histórico. Caberia ao componente da função de ativação de comportamento apenas consultar e analisar as informações registradas no histórico. Além disso, o ALLIANCE leva em consideração que apenas um comportamento pode estar ativado em um agente comportamental por vez. Neste caso, o mecanismo de controle de ativação de comportamento pode realizar este trabalho. Para isso, é necessário definir a decoração da função de ativação de comportamentos do ALLIANCE e a camada de baixo nível de cada comportamentos dos agentes. Por conseguinte, este mecanismo fica responsável por consultar as funções de ativação e manter apenas um comportamento ativo por vez. A chamada pelo processamento do comportamento ativo também é realizado pelo mecanismo de controle de ativação.

Um diagrama de classes modelado em um trabalho anterior[3] é apresentado na Figura 29. Este modelo é uma aproximação da camada de alto nível de abstração da arquitetura ALLIANCE. Este modelo pode ser simplificado ao se utilizar os mecanismos de monitoramento e controle de ativação de comportamento. A Figura 30 mostra uma simplificação do modelo apresentado na Figura 29.

Primeiramente, as classes *InterCommunication*, *Robot*, *BehaviorSet* e *MotivationalBehavior* foram eliminadas no novo modelo proposto na Figura 30. A classe *InterCommunication* foi substituída pelo mecanismo de monitoramento de comportamento do *framework TAlMech*. Todo o controle realizado pelas classes *Robot*, *BehaviorSet* e *MotivationalBehavior* foi substituído pelo mecanismo de controle de ativação de comportamento do *framework TAlMech*.

Entretanto, para que o mecanismo de controle de ativação de comportamento funcione corretamente, é necessário definir a função de ativação de comportamento da arquitetura. Para isso, os componentes do cálculo de motivação devem ser implementados como componentes de decoração da função de ativação de comportamento. Logo, as classes *SensoryFeedback*, *Impatience*, *ImpatienceReset*, *ActivitySuppression* e *Acquiescence* devem

[3] Repositório da arquitetura ALLIANCE para o ROS: <https://github.com/adrianohrl/alliance>. O Apêndice C faz uma breve descrição da utilização deste pacote.

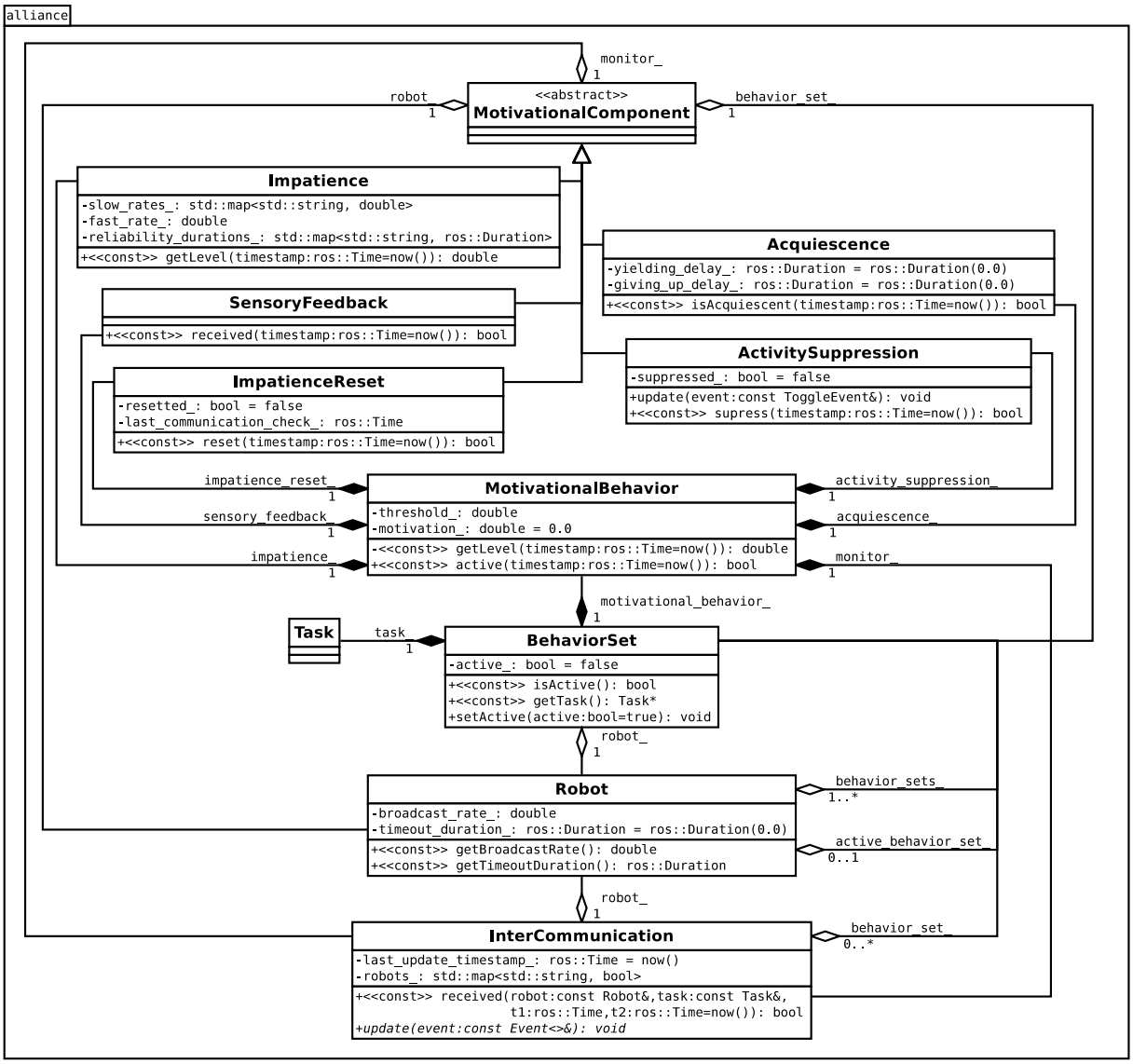


Figura 29 – Diagrama UML de uma aproximação genérica do ALLIANCE proposta em trabalho anterior.

herdar a classe *MotivationDecorator*. Cada uma dessas classes ainda precisa implementar o método de verificação de ativação, pois elas implementam a interface *MotivationComponent*. Portanto, a partir da chamada do método de verificação de cada componente da decoração, o mecanismo ativa e processa o comportamento ativo apropriadamente, conforme mostra a Figura 31.

Note que a arquitetura ALLIANCE define uma decoração estática para as funções de ativação dos comportamentos dos agentes. Em outras palavras, todos os agentes do sistema possuem a mesma função de ativação de comportamento. Entretanto, a dinâmica de cada função de ativação de um dado comportamento pode ser diferenciada das demais através da configuração dos seus parâmetros.

Perceba que os construtores das classes *SensoryFeedback*, *Impatience*, *ImpatienceReset*, *ActivitySuppression* e *Acquiescence* do novo modelo proposto na Figura 30 já

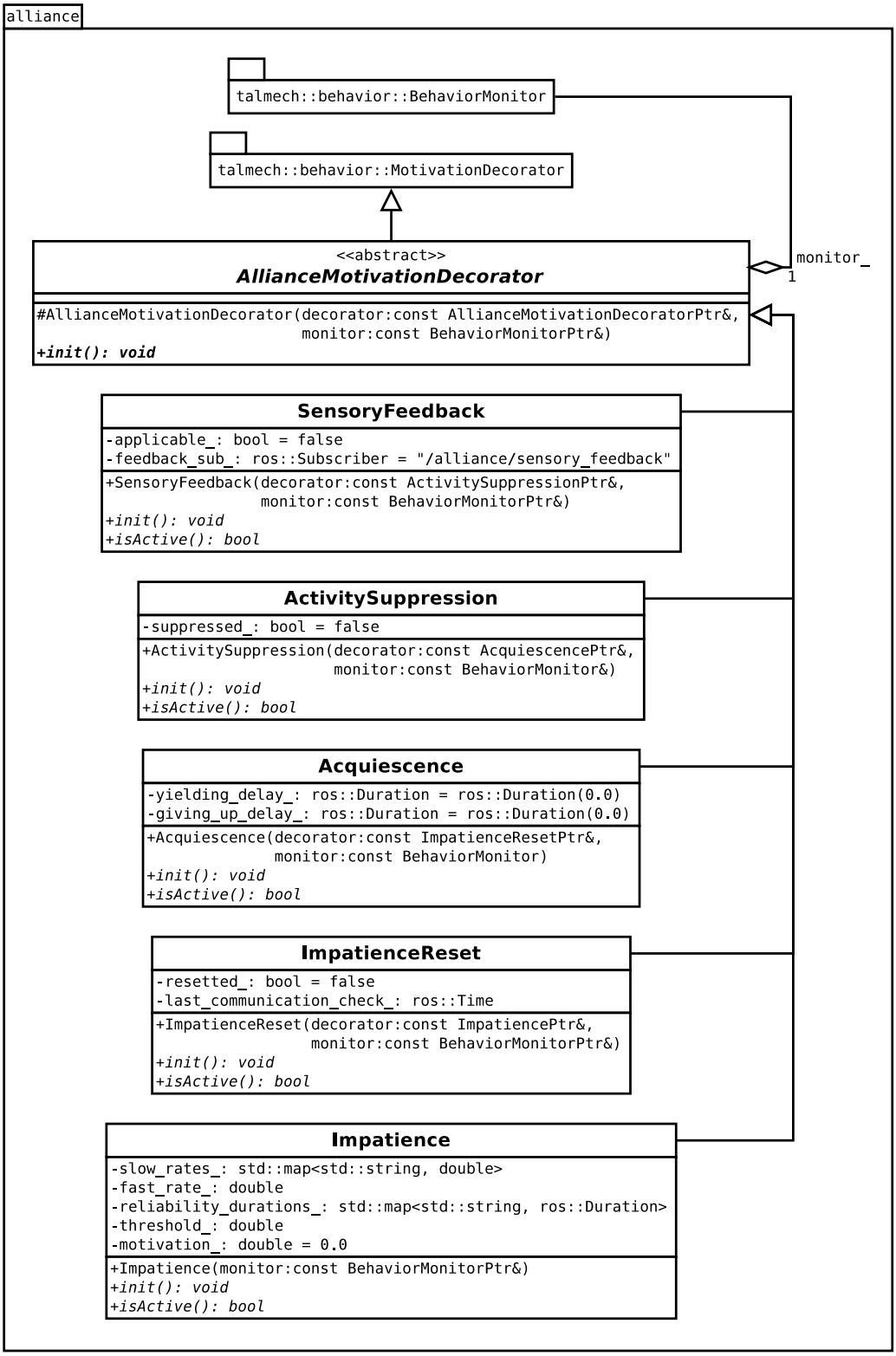
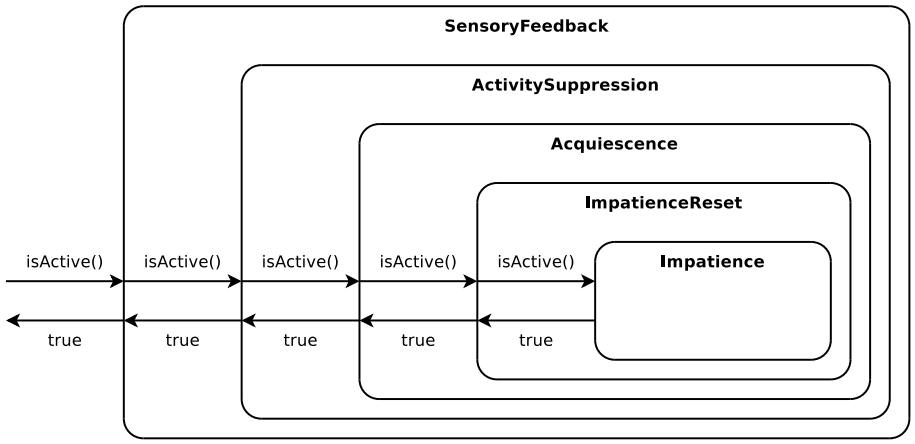
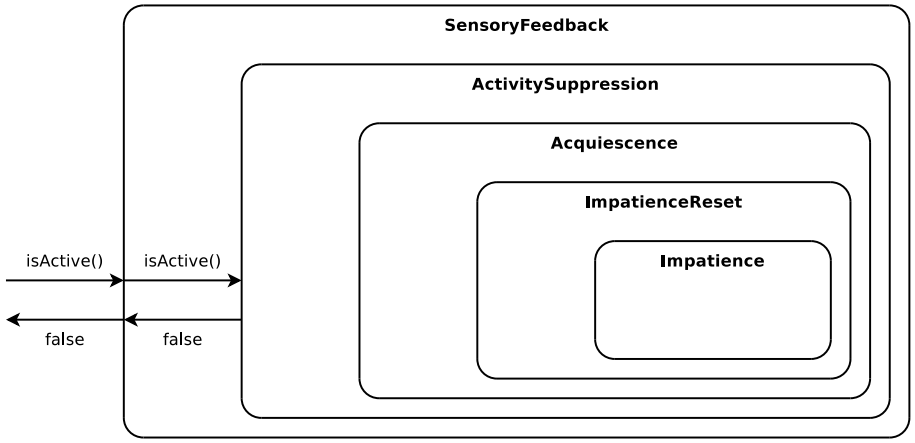


Figura 30 – Diagrama UML da aproximação genérica do ALLIANCE proposta anteriormente utilizando o *framework TAlMech*.



(a) Caso em que o comportamento está ativo.



(b) Caso em que o comportamento não está ativo.

Figura 31 – Exemplos de decoração da função de ativação de comportamento.

definem a forma de decoração permitida. O modelo define que a função de ativação de comportamento do ALLIANCE é dado pela seguinte decoração: (1) primeiramente, é verificado se o comportamento é aplicável através do componente *SensoryFeedback*, (2) se sim, então é verificado se o comportamento não está suprimido por outro comportamento do agente, (3) se o comportamento em questão não estiver suprimido, é verificado se o agente é aquiescente à desativação do comportamento (caso, esteja ativado), (4) se não, é verificado se a motivação do agente para o comportamento em questão deve ser zerada e, finalmente, (5) o nível de impaciência do agente com relação a este comportamento é atualizado mediante à taxa de impaciência apropriada. A Figura 31 ilustra a decoração permitida, segundo o modelo proposto na Figura 30. Caso todas as verificações sejam verdadeiras, o comportamento é ativado pelo mecanismo de controle de ativação do *TAlMech*. Por outro lado, caso uma das verificações seja falsa, as verificações são interrompidas e o mecanismo de controle de ativação mantém o comportamento desativado ou o desativa.

Estas duas aproximações possuem duas diferenças pequenas. Ambas diferenças estão relacionadas à função de ativação dos comportamentos. A função de ativação de comportamento é representada distintamente em cada modelagem. Na aproximação que

não utiliza o *TAlMech* cada componente pode ser acessado diretamente dentro da classe *MotivationalBehavior*. Por outro lado, a aproximação que usa o *TAlMech* tem acesso indireto dos componentes através da cadeia de decoração na classe *Behavior*. Por este motivo, a aproximação antiga (que não faz uso do *framework TAlMech*) consome menos memória. Além disso, a nova aproximação (que utiliza o *TAlMech*) interrompe o cálculo de motivação assim que um dos componentes da função de ativação indica que o comportamento deve ser (ou permanecer) desativado, conforme mostra a Figura 31b. Contudo, ambas diferenças são desprezíveis, pois a quantidade de componentes na função de ativação de comportamento é fixa.

Cabe ao desenvolvedor desta nova aproximação da arquitetura ALLIANCE relacionar a ativação da camada de alto nível de abstração do comportamento com as camadas de comportamento mais baixas através do mecanismo de subsunção proposto por Brooks (1986).

4.3.2 Ensaio

Foi realizado o ensaio de uma aplicação de patrulhamento para testar a nova aproximação da arquitetura ALLIANCE utilizando os mecanismo de monitoramento e controle de ativação de comportamento do *framework TAlMech*. O patrulhamento é realizado por cinco 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 MobileSim[4] da Adept MobileRobots foram utilizados cinco robôs diferenciais móveis Pioneer 3 DX (MOBILEROBOTS, 2017b), mostrado na Figura 32, para simular a dinâmica dos robôs em um ambiente fechado para a simulação de uma aplicação de patrulhamento. Dado que o foco deste trabalho está na avaliação do *framework TAlMech*, 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.

[4] <http://robots.mobilerobots.com/wiki/MobileSim>

Figura 32 – Robô Pioneer 3 DX da Adept MobileRobots (MOBILEROBOTS, 2017b).

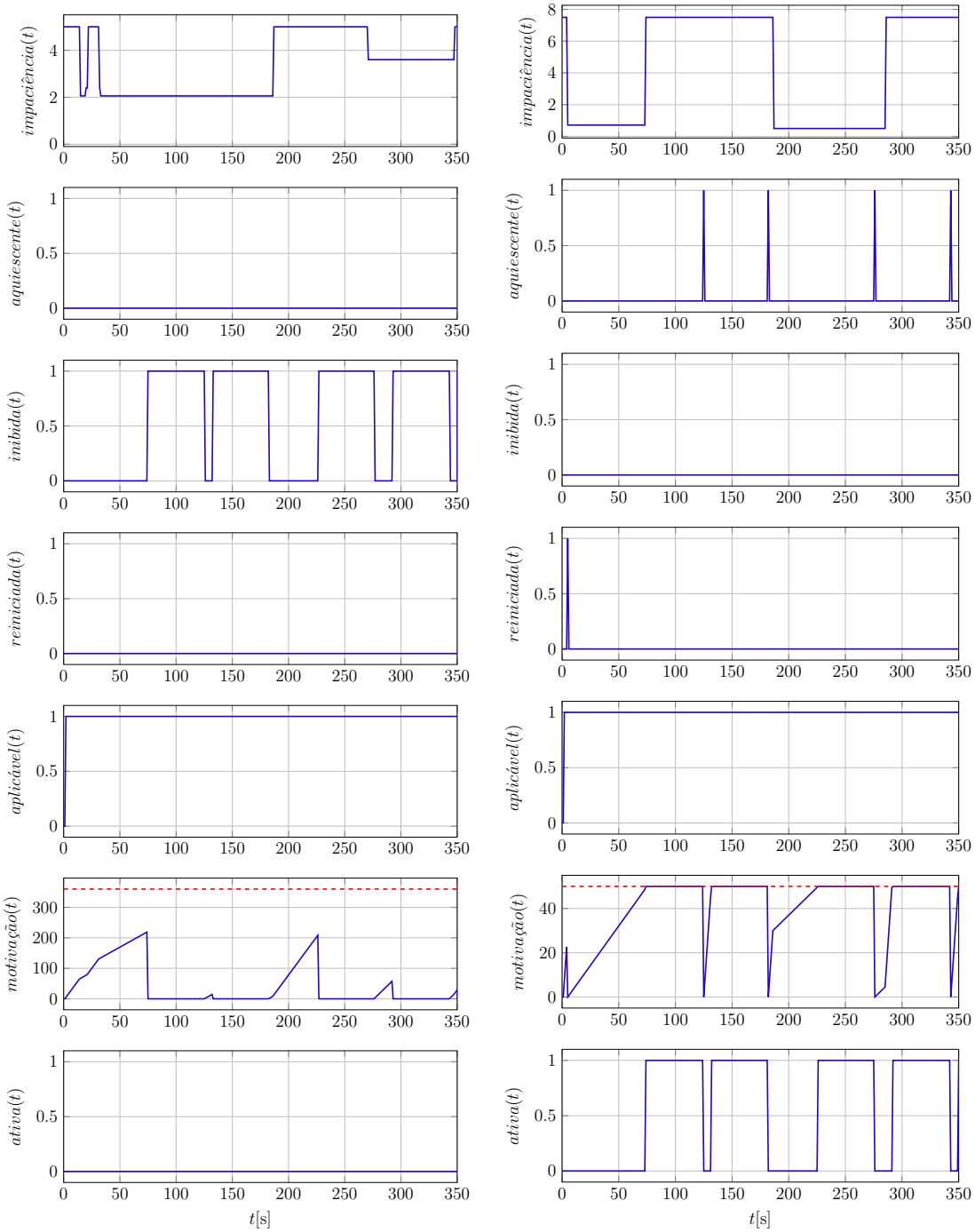
Também foi utilizado o pacote *RosAria*[5] do ROS para ter acesso aos sensores e atuadores dos robôs no ambiente simulado com o sistema de comunicação do ROS através de uma rede de computadores.

A contribuição de cada componente da função de ativação de cada comportamento dos agentes do sistema foi armazenada em um arquivo *bag* durante um trecho da execução do ensaio. As Figuras 33 e 34 mostram uma parcela dos dados armazenados neste arquivo *bag*. Cada figura é composta por sete gráficos, de cima para baixo: (1) taxa de impaciência de uma agente em relação à uma dada tarefa associada a um de seus comportamentos; (2) aquiescência para desativação do comportamento ao longo do tempo; (3) inibição do comportamento ao longo do tempo; (4) reinício do nível de impaciência acumulado ao longo do tempo; (5) aplicabilidade do comportamento ao longo do tempo; (6) intensidade de motivação para ativação do comportamento ao longo do tempo; e, enfim, (7) ativação do comportamento ao longo do tempo. Os cinco primeiros gráficos influenciam no gráfico de motivação conforme define a Equação 2.8.

As Figuras 33a e 33b mostram a contribuição de cada componente da função de ativação dos comportamentos *wander* e *report* do robô *robot3*, respectivamente. Nota-se que a dinâmica da sua configuração de comportamento *report* possui uma dinâmica mais rápida do que a configuração de *wander*. Com isso, o limiar de motivação para ativar o comportamento *report* é atingido diversas vezes, enquanto o de *wander* nunca é. Situação similar a de *wander* ocorre com a configuração de comportamento *border_protection*. Perceba que o gráfico de inibição do comportamento *wander* de *robot3* é equivalente ao seu gráfico de ativação do comportamento *report*. Isto acontece porque apenas um comportamento pode ser ativado por vez. Logo, a ativação do comportamento *report* de *robot3* inibe a ativação do seus comportamentos *wander* e *border_protection*. O gráfico de reinício da motivação acumulada do comportamento *report* de *robot3* possui um pulso unitário. Este pulso unitário acontece uma única vez para cada comportamento de um dado robô, conforme define a Equação 2.4. Quando o robô percebe que outro robô do sistema ativou o comportamento em questão, este pulso é gerado fazendo com que sua motivação acumulada para ativar este comportamento seja zerada. Enfim, percebe-se que o motivo para a desativação do comportamento *report* de *robot3* é causada pelos pulsos

[5] <http://wiki.ros.org/ROSARIA>

do seu gráfico de aquiescência.



(a) Motivação da configuração de comportamento */robot3/wander*.

(b) Motivação da configuração de comportamento */robot3/report*.

Figura 33 – Motivação das configurações de comportamento de *robot3*: na esquerda, o comportamento *wander*, na direita, o comportamento *report*.

As Figuras 34a e 34b mostram a contribuição de cada componente da função de ativação dos comportamentos *wander* e *report* do robô *robot4*, respectivamente. Este robô ativa diferentes comportamentos durante este trecho da aplicação. Inicialmente, seu comportamento *wander* foi ativado. Entretanto, logo em seguida ocorre um pulso unitário no

seu gráfico de reinício de motivação acumulada. Isto significa que o robô *robot4* identificou que outro robô do sistema ativou um comportamento referente à tarefa *wander*. Consequentemente, a motivação acumulada do comportamento *wander* de *robot4* foi zerado. Por conseguinte, este comportamento foi desativado. Contudo, rapidamente sua motivação cresce novamente até atingir o limiar para ativação. No instante em que isso ocorre, este é ativado e permanece assim até ficar aquiescente à desativação deste comportamento. Este momento ocorre evidentemente quando é registrado um pulso unitário no gráfico de aquiescência da Figura 34a. O comportamento *wander* de *robot4* é inibido porque a motivação para ativação do seu comportamento *report* atinge rapidamente o limiar enquanto a motivação de *wander* cresce paulatinamente. Quando isso acontece, o comportamento é ativado. Perceba que os gráficos de ativação dos comportamentos *wander* e *report* de *robot4* são equivalentes aos gráficos de inibição de seus comportamentos *report* e *wander*, respectivamente.

Por fim, a Figura 35 mostra o histórico registrado pelo mecanismo de monitoramento de comportamento do *framework TAlMech*. Estes históricos foram registrados no mesmo intervalo em que os gráficos das Figuras 33 e 34 foram registradas. Enquanto o eixo das abcissas indica o instante, em segundos, em relação ao início da execução da aplicação, o eixo das ordenadas indica o comportamento ativo ao longo do tempo. Os comportamentos *report*, *wander* e *border_protection* são representados no eixo das ordenadas destes gráficos pelos números 1, 2 e 3, respectivamente.

As Figuras 35a, 35b, 35c, 35d e 35e mostram respectivamente o histórico registrado pelo mecanismo de monitoramento de comportamento do *TAlMech*.

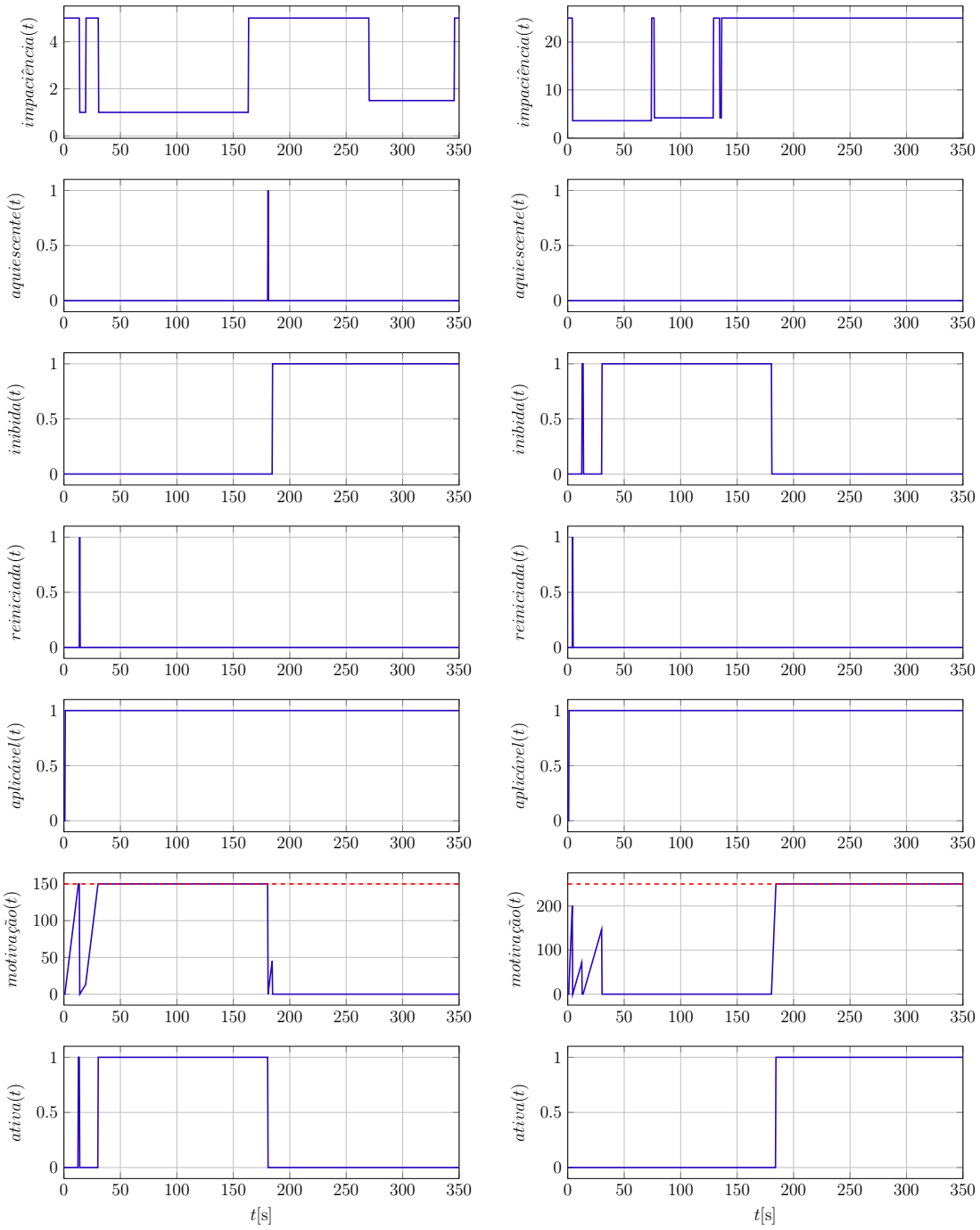
Percebe-se que o gráfico da Figura 35c é igual ao gráfico de ativação da Figura 33b. Dado que o comportamento *report* é representado pelo número 1 nos históricos registrados pelo monitor de comportamento, esta igualdade procede, pois a relação entre a função do histórico com as funções de comportamento é dada pela Equação 4.1.

$$comportamento(t) = ativa_{report}(t) + 2ativa_{wander}(t) + 3ativa_{border\_protection}(t) \quad (4.1)$$

Neste caso, a Equação 4.1 pode ser simplificada para a Equação 4.2, pois os comportamentos *wander* e *border_protection* do robô *robot3* não foram ativados em momento algum nesta janela temporal.

$$comportamento(t) = ativa_{report}(t) \quad (4.2)$$

Ao comparar os gráficos de ativação das Figuras 34a e 34b, verifica-se que o histórico de *robot4* registrado pelo mecanismo de monitoramento de comportamentos está em conformidade com a Equação 4.1, conforme mostra a Figura 35d. Rapidamente é ativado e desativado o comportamento *wander* de *robot4* no início da janela do histórico. Em

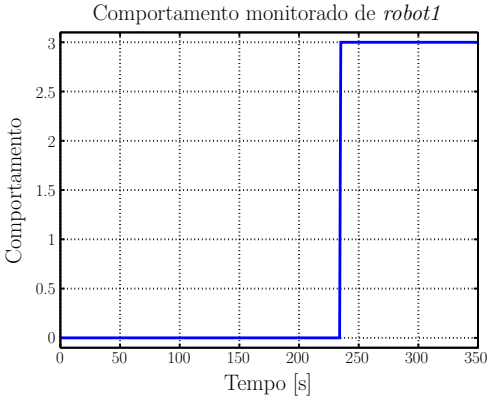


(a) Motivação da configuração de comportamento */robot4/wander*.

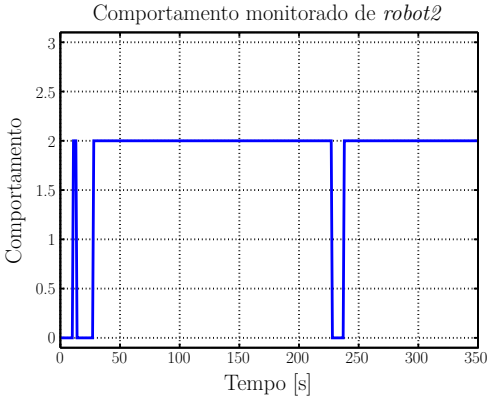
(b) Motivação da configuração de comportamento */robot4/report*.

Figura 34 – Motivação das configurações de comportamento de *robot4*: na esquerda, o comportamento *wander*, na direita, o comportamento *report*.

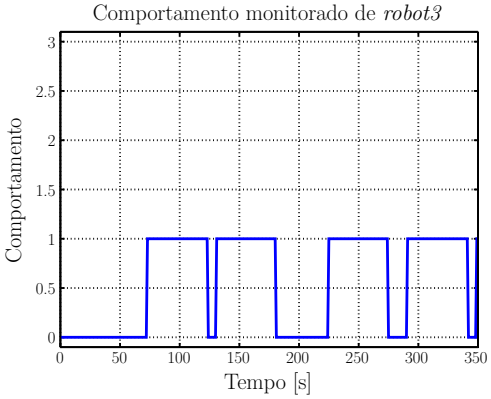
outras palavras, é registrado um pulso de intensidade 2 no início da janela do histórico de *robot4*. Alguns instante depois, a ativação do comportamento *wander* é registrada no monitor de comportamento sobre o robô *robot4*. Posteriormente, este comportamento é desativado e permanece assim por poucos segundos. Logo em seguida, o comportamento *report* de *robot4* é ativado e permanece assim até o final da janela.



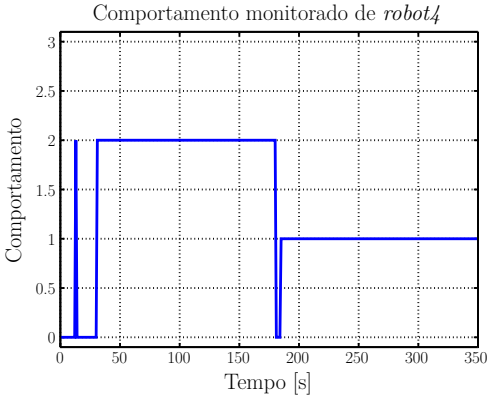
(a) Histórico de comportamento de *robot1*.



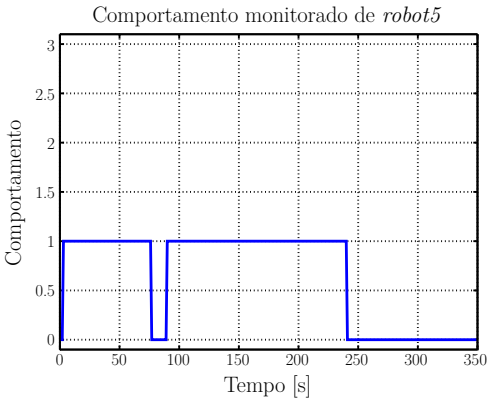
(b) Histórico de comportamento de *robot2*.



(c) Histórico de comportamento de *robot3*.



(d) Histórico de comportamento de *robot4*.



(e) Histórico de comportamento de *robot5*.

Figura 35 – Histórico de comportamento dos robôs durante o patrulhamento, nas ordenadas: (1) *report*, *wander* e *border_protection*.

Portanto, o mecanismo de controle de ativação de comportamento do *TAlMech* substituiu as classes que realizavam o controle dos comportamentos na antiga aproximação da arquitetura ALLIANCE. Além disso, o mecanismo de monitoramento de comportamentos do *framework TAlMech* substituiu o monitor de comunicação entre os agentes no ALLIANCE. Logo, a nova aproximação permitiu uma modelagem enxuta da arquitetura ALLIANCE, onde o projetista necessitou apenas definir a função de ativação de

comportamento utilizado pelo ALLIANCE e interligar a camada de alto nível de abstração com a camada de baixo nível para a ativação efetiva dos comportamentos, isto é, realização da leitura dos sensores, processamento dos dados sensoriais e atuação dos atuadores, conforme a configuração estabelecida pelo comportamento.

5 Conclusão e Trabalhos Futuros

5.1 Conclusão

Este trabalho apresentou o desenvolvimento de uma *framework* integrado com o ROS com o intuito de facilitar o desenvolvimento de arquiteturas de alocação de tarefas em sistemas multirrobôs. O *framework* encapsula quatro mecanismo comumente utilizados: (1) cálculo de utilidade, (2) negociação por leilão, (3) monitoramento de comportamento e (4) controle de ativação de comportamento.

Primeiramente, este trabalho apresentou as diferenças entre *frameworks*, *middlewares* e arquiteturas dedicados à robótica. Em seguida, um estudo sobre sistemas multirrobô foi feito, identificando suas características, bem como, suas vantagens perante um sistema de um único robô. Posteriormente, foi revisado o problema de alocação de tarefa nesses sistemas e como eles podem ser classificados. Por fim, foram revisadas algumas arquiteturas que resolvem problemas de alocação de tarefa, detalhando arquiteturas baseadas em comportamentos e em negociação.

Após a revisão teórica, foi apresentado no capítulo seguinte o desenvolvimento do *framework TAlMech*. Antes, a descrição, as consequências e a aplicabilidade de cada padrão de projeto utilizados no desenvolvimento do *TAlMech* foram listados. Os padrões listados foram utilizados de modo a facilitar a manutenção, evolução, extensão e utilização do *framework* desenvolvido. Na sequência, foram apresentadas as classes comuns e utilitárias aos mecanismos encapsulados. Posteriormente, cada mecanismo do *TAlMech* foi detalhado: (1) cálculo de utilidade, (2) negociação por leilão, (3) monitoramento de comportamento e (4) controle de ativação de comportamento. O mecanismo de cálculo de utilidade permitiu uma representação modular e flexível para os agentes do sistema quantificarem a receita/custo de executar uma dada tarefa. O mecanismo de negociação por leilão encapsula o protocolo de comunicação entre leiloeiros e licitantes durante o leilão de tarefas. O mecanismo de monitoramento de comportamento cria um histórico para consulta a partir da observação das atividades dos agentes comportamentais do sistema. Enfim, o mecanismo de controle de ativação de comportamento define uma representação genérica de função de ativação de comportamento. Esta função é utilizada pelo mecanismo para verificar se o comportamento deve ser/permanecer ativo. A chamada pelo processamento do comportamento também é realizada pela *TAlMech*. Além disso, o mecanismo de controle ainda garante que apenas um comportamento fique ativo por vez.

Para testar e validar os mecanismo de utilidade e leilão do *TAlMech*, quatro ensaios foram conduzidos utilizando um conjunto de 5000 tarefas amostradas aleatoriamente por

um gerador configurável. Em um dos ensaios, o mecanismo de leilão foi configurado de modo a obter uma aproximação da arquitetura Murdoch. Através destes ensaios, foi verificado que o *framework* auxilia a representação do cálculo de utilidade e, ainda, fornece componentes úteis para decorá-lo. Ademais, verificou-se que o mecanismo de negociação por leilão do *TAlMech* é configurável e pode ser utilizados em arquiteturas MRTA que negociam por leilão e lidam com problemas do tipo *SR-ST-IA* e *SR-MT-IA*.

Finalmente, os mecanismos para arquiteturas baseadas em comportamento foram utilizados para desenvolver uma aproximação da arquitetura ALLIANCE. A camada de alto nível de abstração ficou mais organizada com a nova representação da função de ativação dos comportamentos. Além disso, os mecanismos de monitoramento e controle de ativação substituíram diversas classes da antiga aproximação desta arquitetura. O mecanismo de controle de ativação de comportamento garantiu que apenas um comportamento estivesse ativo por vez em cada agente e que este fosse processado no tempo devido. O mecanismo de monitoramento de comportamento auxiliou no cálculo de motivação das funções de ativação de comportamento no ALLIANCE.

Concluindo, o *framework TAlMech* foi desenvolvido de modo a permitir a extensão de novos mecanismos e, também, novas estratégias de controle nos mecanismos existentes. As aproximações das arquiteturas ALLIANCE e Murdoch desenvolvidas a partir do *framework TAlMech* são exemplos de que o *TAlMech* possibilita representações independentes do domínio da aplicação que as utiliza. Sendo um projeto de código aberto, o *TAlMech* está disponível para a utilização e contribuição da comunidade de robótica.

5.2 Trabalhos Futuros

Os seguintes trabalhos futuros são propostos para o *framework TAlMech*:

- Implementar o mecanismo de negociação de comércio segundo o protocolo proposto por Sandholm, Lesser *et al.* (1995) e utilizado por Dias (2004);
- Criar um mecanismo de reserva de recursos para agendamento de tarefas;
- Disponibilizar o pacote ROS que contém a biblioteca com o *framework TAlMech* no repositório do ROS.
- Criar uma página do pacote que contém a biblioteca do *framework TAlMech*;
- Criar uma lista de arquiteturas que foram desenvolvidas a partir do *framework TAlMech* na página do *wiki.ros.org* com o intuito de facilitar a procura delas pelos seus clientes;

- Criar tutoriais no ROS sobre a utilização do *framework TAlMech* no desenvolvimento de arquiteturas para alocação de tarefas em sistema multirrobô;
- Disponibilizar o pacote ROS que contém a aproximação genérica da arquitetura ALLIANCE desenvolvida a partir do *framework TAlMech*;
- Disponibilizar o pacote ROS que contém a aproximação genérica da arquitetura Murdoch desenvolvida a partir do *framework TAlMech*;
- Desenvolver arquiteturas propostas na literatura utilizando o *TAlMech*;
- Integrar o *framework* com bibliotecas que encapsulam planejadores e técnicas de inteligência artificial;
- Criar interfaces gráficas para facilitar a configuração dos mecanismos do *framework TAlMech*;
- E criar interfaces gráficas das ferramentas métricas desenvolvidas.

A cada evolução do *framework TAlMech*, deve-se manter os objetivos estabelecidos neste trabalho de modo que este tenha um propósito bem definido para a comunidade de robótica.

Referências

ADOBBATI, R. *et al.* Gamebots: A 3d virtual world test-bed for multi-agent research-proceedings of the second international workshop on infrastructure for agents, mas, and scalable mas. *Anais... Montreal, Canada, 2001Disponível em:< http://cs. biu. ac. il/˜ galk/Publications/01/gamebots. pdf>. Acesso em*, v. 3, 2015. 35

ARKIN, R. C.; BALCH, T. Aura: Principles and practice in review. *Journal of Experimental & Theoretical Artificial Intelligence*, Taylor & Francis, v. 9, n. 2-3, p. 175–189, 1997. 22

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. 26

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. 17, 39

BROOKS, A. *et al.* Orca: A component model and repository. *Software engineering for experimental robotics*, Springer, p. 231–251, 2007. 21

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. 30, 101

BRUYNINCKX, H. Open robot control software: the orocos project. In: IEEE. *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*. [S.l.], 2001. v. 3, p. 2523–2528. 21

CALISI, D. *et al.* Openrdk: a modular framework for robotic software development. In: IEEE. *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*. [S.l.], 2008. p. 1872–1877. 21

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. 17

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. 17, 39

COLON, E.; SAHLI, H.; BAUDOIN, Y. Coroba, a multi mobile robot control and simulation framework. *International Journal of Advanced Robotic Systems*, SAGE Publications Sage UK: London, England, v. 3, n. 1, p. 13, 2006. 20

DIAS, M. B. Traderbots: A new paradigm for robust and efficient multirobot coordination in dynamic environments. *Robotics Institute*, p. 153, 2004. 44, 45, 109

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. 44

DIAS, M. B. *et al.* Market-based multirobot coordination: A survey and analysis. *Proceedings of the IEEE*, IEEE, v. 94, n. 7, p. 1257–1270, 2006. 38, 39

DUDEK, G. *et al.* A taxonomy for multi-agent robotics. *Autonomous Robots*, Springer, v. 3, n. 4, p. 375–397, 1996. 17, 30, 39

ESUBALEW, T. *et al.* A step towards developing adaptive robot-mediated intervention architecture (aria) for children with autism. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, IEEE, v. 21, n. 2, p. 289–299, 2013. 20

FOX, M.; LONG, D. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 2003. 17

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. 17

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. 47

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. 17, 36, 41

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. 17, 25, 26, 27, 28, 29, 30, 39, 93

HORST, R.; PARDALOS, P. M.; THOAI, N. V. *Introduction to global optimization*. [S.l.]: Springer Science & Business Media, 2000. 29

IÑIGO-BLASCO, P. *et al.* Robotics software frameworks for multi-agent robotic systems development. *Robotics and Autonomous Systems*, Elsevier, v. 60, n. 6, p. 803–821, 2012. 20

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. 17

KHAMIS, A.; HUSSEIN, A.; ELMOGY, A. Multi-robot task allocation: A review of the state-of-the-art. In: *Cooperative Robots and Sensor Networks 2015*. [S.l.]: Springer, 2015. p. 31–51. 23, 29, 30, 36, 39

KONOLIGE, K. *et al.* The saphira architecture: A design for autonomy. *Journal of experimental & theoretical artificial intelligence*, Taylor & Francis, v. 9, n. 2-3, p. 215–235, 1997. 22

KORSAH, G. A.; STENTZ, A.; DIAS, M. B. A comprehensive taxonomy for multi-robot task allocation. *The International Journal of Robotics Research*, SAGE Publications Sage UK: London, England, v. 32, n. 12, p. 1495–1512, 2013. 27, 28, 30, 39

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. 17

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. 18

LIMA, P. U.; CUSTODIO, L. M. Multi-robot systems. In: *Innovations in robot mobility and control*. [S.l.]: Springer, 2005. p. 1–64. 22

MAGNENAT, S. *et al.* Aseba: A modular architecture for event-based control of complex robots. *IEEE/ASME transactions on mechatronics*, IEEE, v. 16, n. 2, p. 321–329, 2011. 20

MANNE, A. S. On the job-shop scheduling problem. *Operations Research*, INFORMS, v. 8, n. 2, p. 219–223, 1960. 17

METTA, G.; FITZPATRICK, P.; NATALE, L. Yarp: yet another robot platform. *International Journal of Advanced Robotic Systems*, SAGE Publications Sage UK: London, England, v. 3, n. 1, p. 8, 2006. 21

MOBILEROBOTS, A. *ARIA*. 2017. Disponível em: <http://www.mobilerobots.com/Software/ARIA.aspx>. Acesso em: 11 janeiro 2018. 20

MOBILEROBOTS, A. *Pioneer 3DX*. 2017. Disponível em: <http://www.mobilerobots.com/ResearchRobots/PioneerP3DX.aspx>. Acesso em: 11 janeiro 2018. 101, 102

MOHAMED, N.; AL-JAROODI, J.; JAWHAR, I. Middleware for robotics: A survey. In: IEEE. *Robotics, Automation and Mechatronics, 2008 IEEE Conference on*. [S.l.], 2008. p. 736–742. 21

MONTEMERLO, M.; ROY, N.; THRUN, S. Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (carmen) toolkit. In: IEEE. *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*. [S.l.], 2003. v. 3, p. 2436–2441. 21

NUNES, E. *et al.* A taxonomy for task allocation problems with temporal and ordering constraints. *Robotics and Autonomous Systems*, Elsevier, v. 90, p. 55–70, 2017. 28, 30, 39

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. 34

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. 17, 18, 30, 131, 135

PARKER, L. E.; TANG, F. Building multirobot coalitions through automated task solution synthesis. *Proceedings of the IEEE*, IEEE, v. 94, n. 7, p. 1289–1305, 2006. 23

PETERSSON, L.; AUSTIN, D.; CHRISTENSENI, H. Dca: a distributed control architecture for robotics. In: IEEE. *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*. [S.l.], 2001. v. 4, p. 2361–2368. 22

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. 17, 21, 47, 117

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. 18

SANDHOLM, T.; LESSER, V. R. *et al.* Issues in automated negotiation and electronic commerce: Extending the contract net framework. In: *ICMAS*. [S.l.: s.n.], 1995. v. 95, p. 12–14. 9, 44, 46, 109

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. 17

SERRANO, M.; GALLESIO, E.; LOITSCH, F. Hop: a language for programming the web 2. 0. In: *OOPSLA Companion*. [S.l.: s.n.], 2006. p. 975–985. 21

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. 36

STENTZ, A.; DIAS, M. B. *A free market architecture for coordinating multiple robots*. [S.l.], 1999. 17, 43

TANG, F.; PARKER, L. E. Distributed multi-robot coalitions through asymtre-d. In: IEEE. *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*. [S.l.], 2005. p. 2606–2613. 43

TSARDOULIAS, E.; MITKAS, P. Robotic frameworks, architectures and middleware comparison. *arXiv preprint arXiv:1711.06842*, 2017. 20, 21

UTZ, H. *et al.* Miro-middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation*, IEEE, v. 18, n. 4, p. 493–497, 2002. 21

VOLPE, R. *et al.* The claraty architecture for robotic autonomy. In: IEEE. *Aerospace Conference, 2001, IEEE Proceedings*. [S.l.], 2001. v. 1, p. 1–121. 21

VU, T. *et al.* Monad: A flexible architecture for multi-agent control. In: ACM. *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*. [S.l.], 2003. p. 449–456. 35

WATKINS, C. J.; DAYAN, P. Q-learning. *Machine learning*, Springer, v. 8, n. 3-4, p. 279–292, 1992. 17

WELLMAN, M. P.; WURMAN, P. R. Market-aware agents for a multiagent world. *Robotics and Autonomous Systems*, Elsevier, v. 24, n. 3-4, p. 115–125, 1998. 44

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. 17, 35

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. 36

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. 22, 30, 37, 39

ZHANG, Y.; PARKER, L. E. Iq-asymtre: Synthesizing coalition formation and execution for tightly-coupled multirobot tasks. In: IEEE. *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. [S.l.], 2010. p. 5595–5602. 43

ZLOT, R.; STENTZ, A. Market-based multirobot coordination for complex tasks. *The International Journal of Robotics Research*, Sage Publications Sage CA: Thousand Oaks, CA, v. 25, n. 1, p. 73–101, 2006. 36

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. 17

ZLOT, R. M.; STENTZ, A. An auction-based approach to complex task allocation for multirobot teams. Carnegie Mellon University, The Robotics Institute, 2006. 25, 36, 38, 58

Apêndices

APÊNDICE A – 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 heterogêneas e (6) descoberta e configuração automática de recursos (QUIGLEY *et al.*, 2009). No intuito de cobrir todas exigências de um *middleware*, o ROS 2.0 tenta dar suporte a sistemas embarcados e dispositivos de baixo recurso.

No ROS, projetos atômicos são chamados *pacotes* e podem ser desenvolvidos em diversas linguagens de programação. Isso mostra que o ROS é flexível, pois seus usuários 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 robôs[3], sensores[4] e simuladores[5], planejadores[6], reconhecimento de voz[7], entre outras. Isso evidencia que os usuários de ROS podem focar no desenvolvimento de pesquisa de sua área e contribuir da melhor forma com essa comunidade.

[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>

Enfim, ROS disponibiliza diversas ferramentas para auxiliar no desenvolvimento de projetos e, também, verificar o funcionamento de aplicações. Suas ferramentas típicas são: *get* e *set* de parâmetros de configuração, vizualização 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.

A.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.

A.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:

- **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 raíz 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 enviada 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.

A.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.

- **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

[8] <http://wiki.ros.org/roscpp>
[9] <http://wiki.ros.org/rospy>

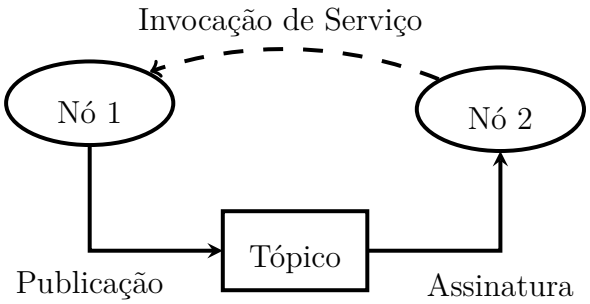


Figura 36 – Conceitos básicos de comunicação do ROS.

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 36 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 tópica 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 ROS[10].

A.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:

[10] <http://wiki.ros.org/common_msgs>

- **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.

A.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 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

[11] <http://wiki.ros.org>
[12] <https://answers.ros.org/questions/>
[13] <http://www.ros.org/news/>

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 recurso no ROS: *base*, *relativa*, *global* e *privada*. Segue, na ordem, um exemplo de cada um desses tipos.

- 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 de nós que trabalham em conjunto, pois eles podem ser escritos como se estivessem se comunicando no nível de *namespace* mais alto.

Tabela 9 – 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

A tabela 9 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.

Esses conceitos possuem extrema importância em sistemas multirrobô, 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.

A.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*.

A.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 37 foi extraida da página do metapacote *rqt* e mostra a aplicação de vários *plugins* que foram acoplados em uma mesma janela através do *rqt_gui*[15].

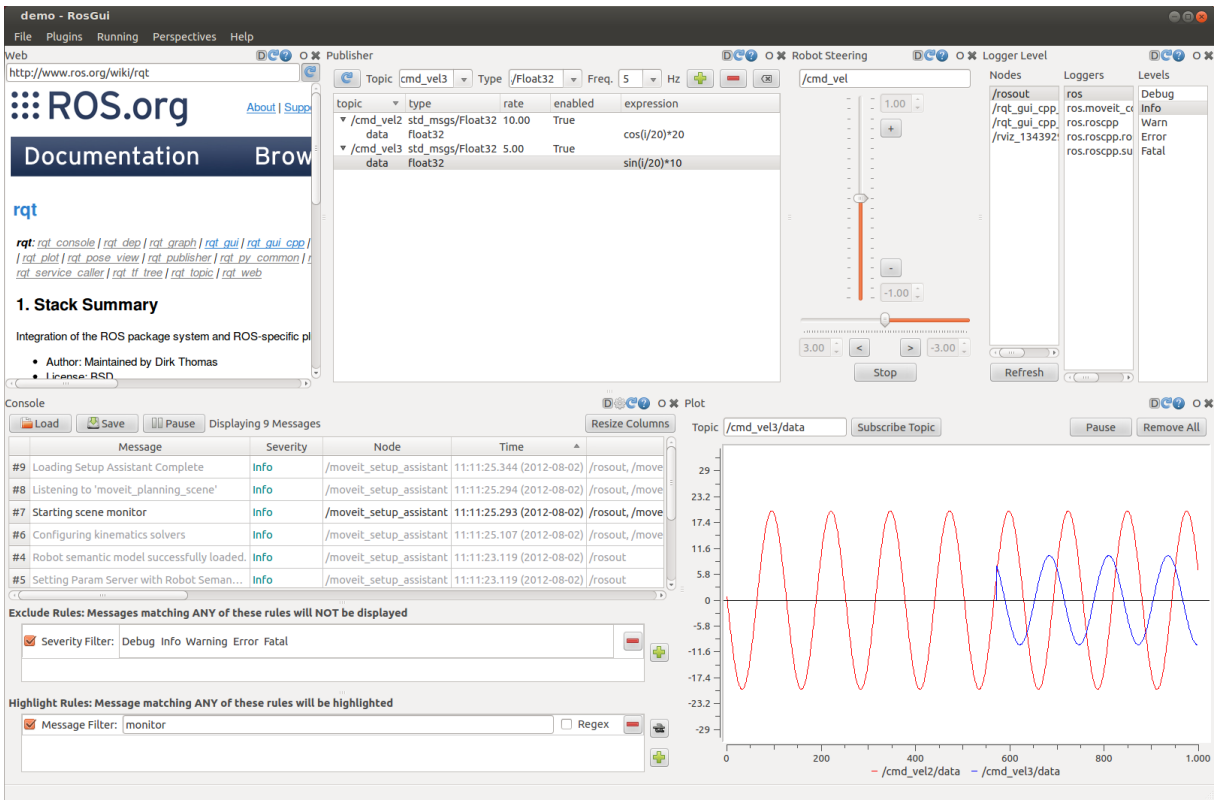


Figura 37 – Exemplo de ferramentas gráficas existentes no ROS.

14 <http://wiki.ros.org/rqt>
15 <http://wiki.ros.org/rqt_gui>

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;
- **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 aos 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 exemplos de ferramentas relacionadas com serviços;
- **Visualização**: agrupa ferramentas que traçam gráficos de dados numéricos no tempo, mostram imagens publicadas em tópicos e, também, sistemas supervisórios, como por exemplo os *plugins rqt_image_view*[18], *rqt_multiplot*[19] e *rqt_rviz*[20];
- e muitas outras.

[16] <http://wiki.ros.org/rqt_launch>
[17] <http://wiki.ros.org/rqt_reconfigure>
[18] <http://wiki.ros.org/rqt_image_view>
[19] <http://wiki.ros.org/rqt_multiplot>
[20] <http://wiki.ros.org/rqt_rviz>

APÊNDICE B – Pacote *talmech_msgs*

O pacote *talmech_msgs* define as mensagens que o *framework TAlMech* utiliza para que haja comunicação entre os agentes das arquiteturas. As mensagens definidas neste pacote são as mensagens (1) *talmech_msgs/Acknowledgment*, (2) *talmech_msgs/Auction*, (3) *talmech_msgs/Behavior*, (4) *talmech_msgs/Bid*, (5) *talmech_msgs/Contract*, (6) *talmech_msgs/Feature* e (7) *talmech_msgs/Task*. Estas mensagens foram criadas separadamente com o intuito de evitar problemas de dependência durante a construção e compilação do pacote *talmech*.

As mensagens *talmech_msgs/Feature* e *talmech_msgs/Task* são comuns e, por isso, são utilizadas por todos os mecanismos do *framework TAlMech*. Contudo, a mensagem *talmech_msgs/Behavior* é utilizada pelos mecanismos relacionados à arquiteturas baseadas em comportamento. As demais mensagens são utilizadas pelo mecanismo de negociação por leilão.

As seções seguintes detalham cada uma das mensagens definidas no pacote *talmech_msgs*, as quais são agrupadas na pasta *msg* na raiz do pacote.

B.1 Mensagem *talmech_msgs/Acknowledgment*

A mensagem *talmech_msgs/Acknowledgment* é utilizado pelo mecanismo de negociação por leilão do *framework TAlMech*. Esta mensagem é definida pelo arquivo *Acknowledgment.msg* localizado dentro da pasta *msg* na raiz do pacote *talmech_msgs*. O conteúdo deste arquivo é dado a seguir.

```
time timestamp
string id
string auctioneer
string auction
string bidder
time renewal_deadline
byte status   # 0: Ongoing, 1: Concluded, and 2: Aborted
```

Esta mensagem armazena informações sobre o reconhecimento de licitantes e leiloeiros durante a vigência de um contrato. Logo, os atributos desta mensagem são:

- **timestamp**: o qual informa a estampa temporal em que a mensagem em questão foi criada;

- **id**: o qual informa o código identificador da mensagem;
- **auctioneer**: o qual informa o código identificador do leiloeiro envolvido;
- **auction**: o qual informa o código identificador do leilão em contrato;
- **bidder**: o qual informa o código identificador do licitante envolvido;
- **renewal_deadline**: o qual informa o prazo limite para renovação do contrato;
- e **status**: o qual informa o *status* do contrato; se 0, o contrato está em andamento, se 1, o contrato foi concluído com sucesso ou se 2, o contrato foi abortado.

B.2 Mensagem *talmech_msgs/Auction*

A mensagem *talmech_msgs/Auction* é utilizado pelo mecanismo de negociação por leilão do *framework TAlMech*. Esta mensagem é definida pelo arquivo *Auction.msg* localizado dentro da pasta *msg* na raiz do pacote *talmech_msgs*. O conteúdo deste arquivo é dado a seguir.

```
string id
string auctioneer
talmech_msgs/Task task
float64 reserve_price
time start_timestamp
duration expected_duration
time expected_close_timestamp
float64 expected_renewal_rate
```

Esta mensagem armazena informações sobre um leilão em andamento. Logo, os atributos desta mensagem são:

- **id**: o qual informa o código identificador da mensagem;
- **auctioneer**: o qual informa o código identificador do leiloeiro que está leiloando o leilão em questão;
- **task**: o qual informa o código identificador da tarefa que está sendo leiloada no leilão em questão;
- **reserve_price**: o qual informa o valor mínimo que os licitantes podem dar de lance para a tarefa leiloada;
- **start_timestamp**: o qual informa a estampa temporal do início do leilão em questão;

- **expected_duration**: o qual informa a duração esperada da fase de submissão de lances do leilão em questão;
- **expected_close_timestamp**: o qual informa a estampa temporal esperada para o encerramento da fase de submissão de lances do leilão em questão;
- e **expected_renewal_rate**: o qual informa a frequência esperada de envio da renovação de contrato durante sua vigência.

B.3 Mensagem *talmech_msgs/Behavior*

A mensagem *talmech_msgs/Behavior* é utilizado pelos mecanismos de negociação por leilão do *framework TAlMech*. Esta mensagem é definida pelo arquivo *Behavior.msg* localizado dentro da pasta *msg* na raiz do pacote *talmech_msgs*. O conteúdo deste arquivo é dado a seguir.

```
time timestamp
string id
string agent
talmech_msgs/Task task
```

Esta mensagem armazena informações sobre o comportamento que está ativo em um dado agente comportamental. Logo, os atributos desta mensagem são:

- **timestamp**: o qual informa a estampa temporal em que a mensagem em questão foi criada;
- **id**: o qual informa o código identificador da mensagem;
- **agent**: o qual informa o código identificador do agente comportamental em questão;
- e **task**: o qual informa os dados da tarefa em que o comportamento ativo no agente em questão está relacionado.

B.4 Mensagem *talmech_msgs/Bid*

A mensagem *talmech_msgs/Bid* é utilizado pelo mecanismo de negociação por leilão do *framework TAlMech*. Esta mensagem é definida pelo arquivo *Bid.msg* localizado dentro da pasta *msg* na raiz do pacote *talmech_msgs*. O conteúdo deste arquivo é dado a seguir.

```
time timestamp
string id
```

```
string bidder
string auctioneer
string auction
float64 amount
```

Esta mensagem armazena informações sobre o lance submetido por um licitante durante a fase de submissão de um leilão. Logo, os atributos desta mensagem são:

- **timestamp**: o qual informa a estampa temporal em que a mensagem em questão foi criada;
- **id**: o qual informa o código identificador da mensagem;
- **bidder**: o qual informa o código identificador do licitante que submeteu o lance em questão;
- **auctioneer**: o qual informa o código identificador do leiloeiro para quem o lance em questão foi submetido;
- **auction**: o qual informa o código identificador do leilão para qual o lance em questão foi submetido;
- e **amount**: o qual informa o valor do lance em questão.

B.5 Mensagem *talmech_msgs/Contract*

A mensagem *talmech_msgs/Contract* é utilizado pelos mecanismos de monitoramento e controle de ativação de comportamento do *framework TAlMech*. Esta mensagem é definida pelo arquivo *Contract.msg* localizado dentro da pasta *msg* na raiz do pacote *talmech_msgs*. O conteúdo deste arquivo é dado a seguir.

```
time timestamp
talmech_msgs/Task task
byte status    # 0: Ongoing, 1: Concluded, and 2: Aborted
```

Esta mensagem armazena informações sobre o contrato que um dado licitante está executando. Logo, os atributos desta mensagem são:

- **timestamp**: o qual informa a estampa temporal em que a mensagem em questão foi criada;
- **task**: o qual informa os dados da tarefa em que o contrato em questão está relacionado;

- e **status**: o qual informa o *status* do contrato; se 0, o contrato está em andamento, se 1, o contrato foi concluído com sucesso ou se 2, o contrato foi abortado.

B.6 Mensagem *talmech_msgs/Feature*

A mensagem *talmech_msgs/Feature* é utilizado pelos mecanismos do *framework TAlMech*. Esta mensagem é definida pelo arquivo *Feature.msg* localizado dentro da pasta *msg* na raiz do pacote *talmech_msgs*. O conteúdo deste arquivo é dado a seguir.

```
string resource
byte type   # 0: Feature, 1: DiscreteFeature, and 2: ContinuousFeature
float64 level # not used when type=0
```

Esta mensagem armazena informações sobre uma dada característica. Logo, os atributos desta mensagem são:

- **resource**: o qual informa o código identificador do recurso relacionado à característica em questão;
- **type**: o qual informa o tipo do recurso; se 0, o recurso é unário, se 1, o recurso é discreto ou se 2, o recurso é contínuo.
- e **level**: o qual informa o nível do recurso.

B.7 Mensagem *talmech_msgs/Task*

A mensagem *talmech_msgs/Task* é utilizado pelos mecanismos do *framework TAlMech*. Esta mensagem é definida pelo arquivo *Task.msg* localizado dentro da pasta *msg* na raiz do pacote *talmech_msgs*. O conteúdo deste arquivo é dado a seguir.

```
string id
nav_msgs/Path waypoints
talmech_msgs/Feature[] features
```

Esta mensagem armazena informações sobre uma dada tarefa no sistema. Logo, os atributos desta mensagem são:

- **id**: o qual informa o código identificador da mensagem;
- **waypoints**: o qual informa os pontos de passagem durante a execução da tarefa em questão;

- e **features**: o qual informa as características requeridas pela tarefa em questão.

APÊNDICE C – Pacote *alliance*

O *alliance*[1] 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). Esta arquitetura resolve problemas 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 a 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 Subseção 2.4.1.1 dá mais detalhes sobre o funcionamento da arquitetura ALLIANCE.

Este pacote possui dois nós: *high_level* e *low_level*. Cada robô do sistema deve executar 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 aplica-

[1] https://github.com/adrianohrl/alliance

ção que atende as premissas desta arquitetura, utilizou-se o pacote *pluginlib*[2]. 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 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 o desenvolvimento de *plugins* de sensor, avaliação e camada, mostrando o cabeçalho em C++ da classe base de cada tipo de *plugin*.

C.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 terá 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,
14                           const std::string& name,
15                           const std::string& id);
16   virtual void readParameters();
17   std::string getNamespace() const;
```

[2] <http://wiki.ros.org/pluginlib>

```
18    std::string getName() const;
19    std::string getId() const;
20    virtual bool isUpToDate() const;
21    virtual bool operator==(const Sensor& sensor) const;
22    virtual bool operator!=(const Sensor& sensor) const;
23
24  protected:
25    std::string ns_;
26    std::string name_;
27    std::string id_;
28  };
29
30  typedef boost::shared_ptr<Sensor>
31    SensorPtr;
32  typedef boost::shared_ptr<Sensor const>
33    SensorConstPtr;
34  }
35
36  #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*.

C.1.1 *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 a 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 recurso 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_SENSORY_EVALUATOR_H_
2  #define _ALLIANCE_SENSORY_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>
14         BehavedRobotPtr;
15 typedef boost::shared_ptr<BehavedRobot const>
16         BehavedRobotConstPtr;
17
18 class SensoryEvaluator
19 {
20 public:
21   SensoryEvaluator();
22   virtual ~SensoryEvaluator();
23   virtual void initialize(const ros::NodeHandlePtr& nh,
24                           const BehavedRobotPtr& robot,
25                           const Task& task,
26                           const std::list<std::string> &sensors);
```

```
27    void process();
28    virtual bool isApplicable() = 0;
29    virtual SensorPtr getSensor(const std::string& sensor_id) const;
30
31  protected:
32    typedef std::list<SensorPtr>::iterator iterator;
33    typedef std::list<SensorPtr>::const_iterator const_iterator;
34    std::list<SensorPtr> sensors_;
35
36  private:
37    ros::NodeHandlePtr nh_;
38    ros::Publisher sensory_feedback_pub_;
39    alliance_msgs::SensoryFeedback sensory_feedback_msg_;
40    bool contains(const std::string &sensor_id) const;
41  };
42
43  typedef boost::shared_ptr<SensoryEvaluator>
44          SensoryEvaluatorPtr;
45  typedef boost::shared_ptr<SensoryEvaluator const>
46          SensoryEvaluatorConstPtr;
47  }
48
49  #endif // _ALLIANCE_SENSORY_EVALUATOR_H_
```

C.1.2 *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,
16                           const std::string& name);
17   virtual void setEvaluator(
18                       const SensoryEvaluatorPtr& evaluator);
19   virtual void readParameters();
20   virtual void process() = 0;
21   std::string getName() const;
22   bool operator==(const Layer& layer) const;
23   bool operator!=(const Layer& layer) const;
24
25 private:
26   std::string name_;
27   SensoryEvaluatorPtr evaluator_;
28 };
29
30 typedef boost::shared_ptr<Layer>
31   LayerPtr;
32 typedef boost::shared_ptr<Layer const>
33   LayerConstPtr;
34 }
35
36 #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.

C.2 Pacote *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 o instante em 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.

C.3 Pacote *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 38 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 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 constantes durante todo o intervalo.

A Figura 39, 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 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 os 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.

C.4 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:
```

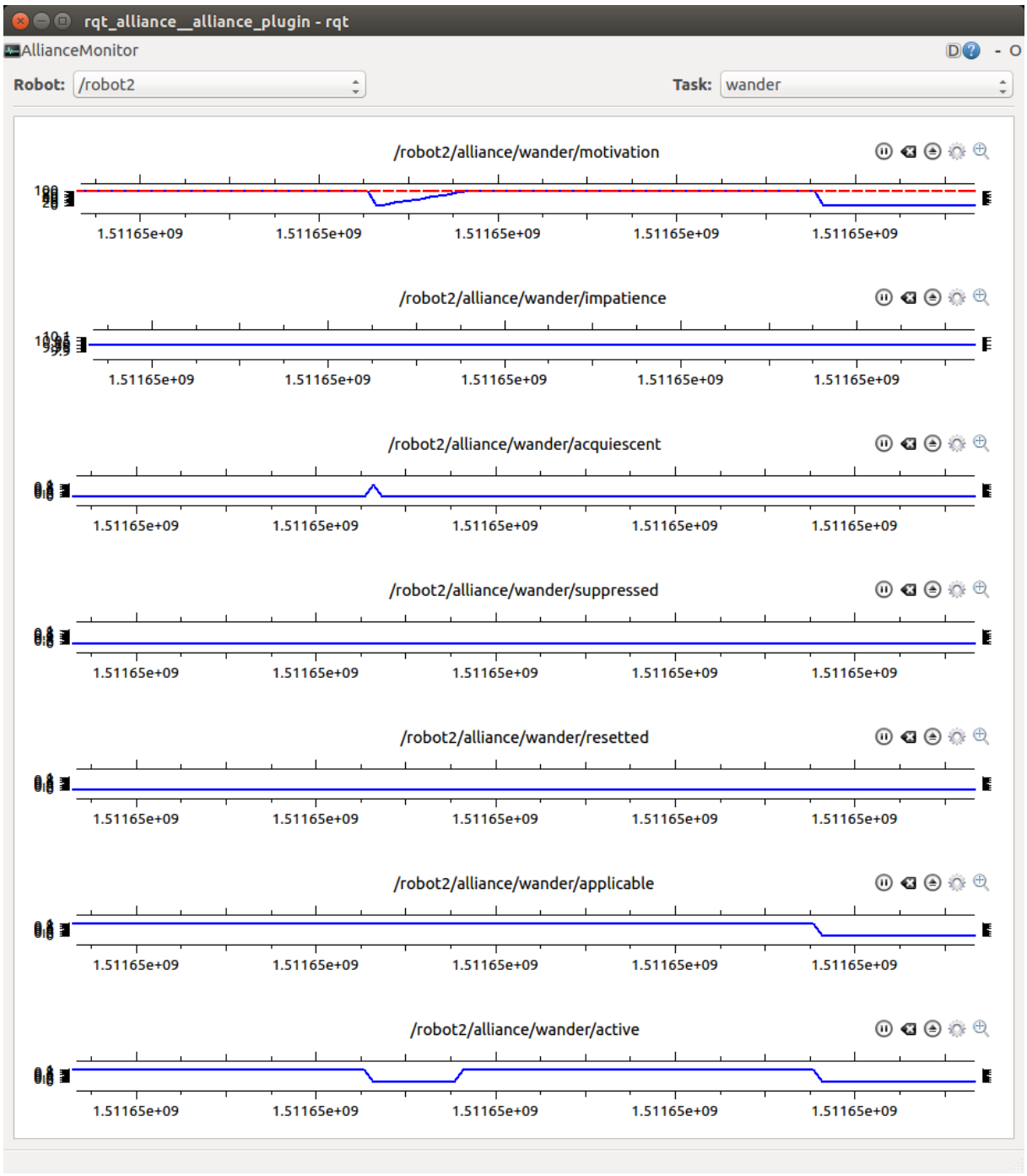


Figura 38 – Detalhamento da motivação */robot2/alliance/wander* ao longo do tempo.

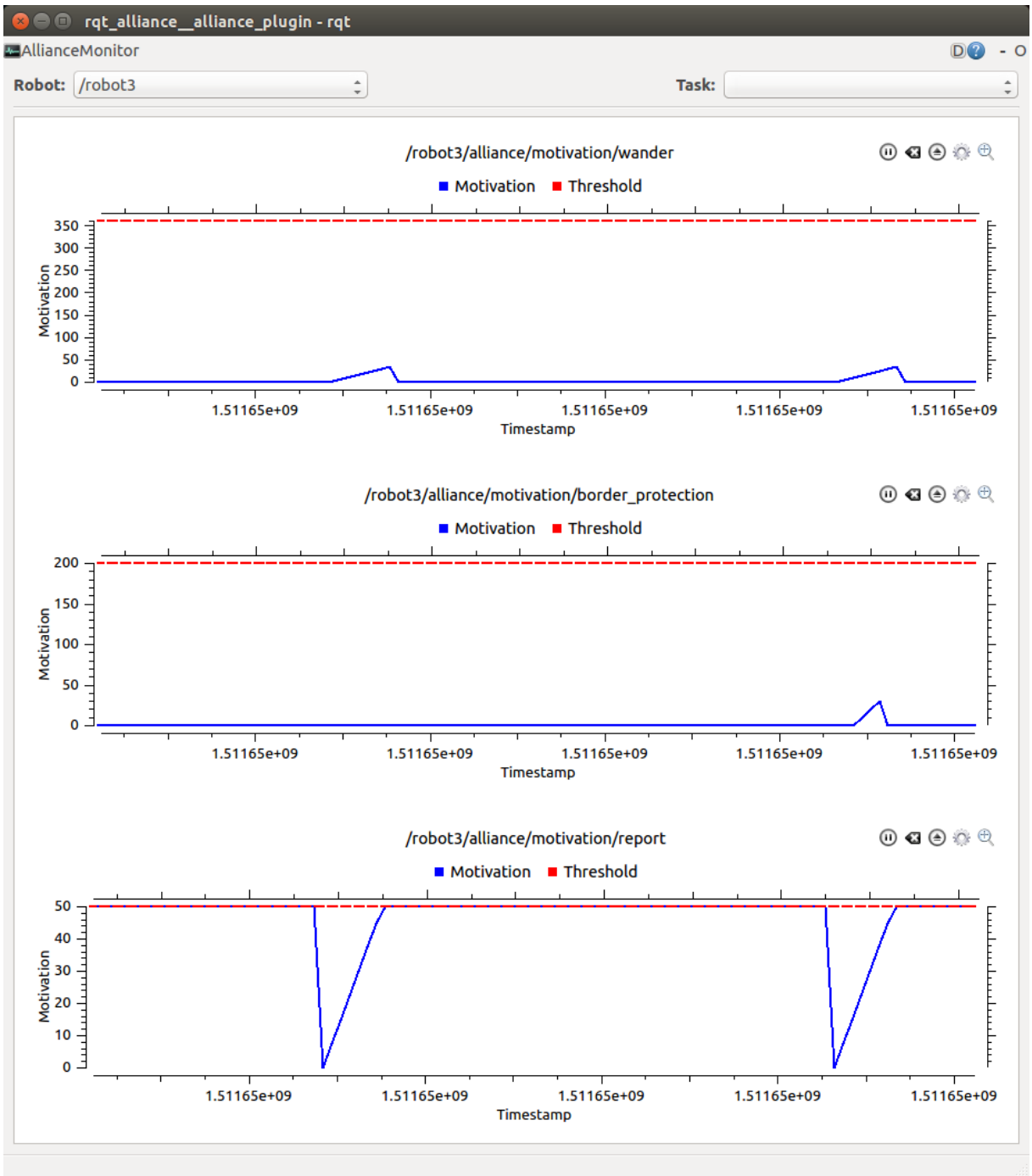


Figura 39 – Motivações das configurações de comportamento do robô */robot3*.

```
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
```