Tui Alexandre Ono Baraniuk

Concepção de uma Arquitetura Tolerante a Falhas para Robôs Submarinos Autônomos

Joinville
2018

Tui Alexandre Ono Baraniuk

CONCEPÇÃO DE UMA ARQUITETURA TOLERANTE A FALHAS PARA ROBÔS SUBMARINOS AUTÔNOMOS

Dissertação submetida ao Programa de Pós-Graduação em Engenharia e Ciências Mecânicas da Universidade Federal de Santa Catarina para a obtenção do Grau de Mestre em Engenharia e Ciências Mecânicas.
Orientador: Prof. Dr. Roberto Simoni
Coorientador: Prof. Dr. Lucas Weihmann

Joinville
2018

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Baraniuk, Tui Alexandre Ono
Concepção de uma arquitetura tolerante a falhas para robôs submarinos autônomos / Tui Alexandre Ono Baraniuk ; orientador, Roberto Simoni, coorientador, Lucas Weihmann, 2018.
143 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Campus Joinville, Programa de Pós Graduação em Engenharia e Ciências Mecânicas, Joinville, 2018.

Inclui referências.

1. Engenharia e Ciências Mecânicas. 2. Veículos Autônomos Submarinos. 3. Tolerância a Falhas. 4. Arquitetura Aberta. 5. Plataforma de Pesquisa. I. Simoni, Roberto. II. Weihmann, Lucas. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia e Ciências Mecânicas. IV. Título.

Tui Alexandre Ono Baraniuk

Concepção de uma Arquitetura Tolerante a Falhas para Robôs Submarinos Autônomos

Esta Dissertação foi julgada adequada para obtenção do Título de "Mestre em Engenharia e Ciências Mecânicas" e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia e Ciências Mecânicas da Universidade Federal de Santa Catarina.

Joinville, 28 de Agosto de 2018.

Prof. Breno Salgado Barra
Coordenador do Curso

Banca Examinadora:

Prof. Dr. Roberto Simoni
Orientador (videoconferência)
Universidade Federal de Santa Catarina

Prof. Dr. Lucas Weihmann
Coorientador
Universidade Federal de Santa Catarina

Prof. Andrea Piga Carboni
Universidade Federal de Santa Catarina

Prof. Maurício de Campos Porath
Universidade Federal de Santa Catarina

Prof. Pablo Andretta Jaskowiak
Universidade Federal de Santa Catarina

Dedico este trabalho aos meus amigos e a toda minha família, por todo apoio e carinho.

AGRADECIMENTOS

À minha mãe Maristela, ao meu pai James, à minha irmã Analin e à minha namorada Camila, pelo cuidado, pela ajuda, pelas ideias e incentivos.

Ao meu orientador, Prof. Dr. Roberto Simoni e ao meu coorientador, Prof. Dr. Lucas Weihmann, pela boa vontade, disponibilidade de tempo e material, que contribuíram no desenvolvimento deste trabalho.

Aos meus amigos e colegas de trabalho, pelo suporte e troca de experiências.

RESUMO

Veículos autônomos submarinos possuem aplicações em ambientes subaquáticos, sem a necessidade de operador externo. Grande parte das missões envolvem longos períodos de operação em ambientes dinâmicos, sendo um dos desafios na criação desses veículos projetar um sistema resiliente, capaz de detectar e tratar falhas. Este trabalho apresenta uma revisão do estado da arte de robôs submarinos autônomos (AUVs), analisando arquiteturas empregadas por AUVs encontrados na literatura. O trabalho apresenta também um estudo de técnicas de tolerância a falhas e possibilidades de abordagem a serem aplicadas em AUVs. Com base na revisão das arquiteturas foi proposta uma nova arquitetura tolerante a falhas para AUVs. A arquitetura foi implementada em uma bancada experimental e testada nos diversos modos. Os resultados dos testes são apresentados e discutidos.

Palavras-chave: AUV, arquitetura aberta, tolerância a falhas, plataforma de pesquisa.

ABSTRACT

Autonomous underwater vehicles (AUVs) can be used without the need of an external operator. Underwater missions usually have long duration and take place in dynamic locations, making the design of resilient, fault-tolerant AUVs a challenge. This study presents a review of the state of the art on AUVs, a survey on architectures found in scientific journals and an analysis on fault-tolerant techniques. In addition, a new fault tolerant architecture for AUVs is proposed and a prototype implementing it is presented as well.

Keywords: AUV, open architecture, fault tolerance, research platform.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AI	Artificial Intelligence
ASV	Autonomous Surface Vehicle
AUV	Autonomous Underwater Vehicle
CAN	Controller Area Network
CC	Corrente Contínua
CRC	Cyclic Redundancy Check
DVL	Doppler Velocity Log
ESC	Eletronic Speed Control
GPS	Global Positioning System
IMU	Inertial Measurement Unit
LBL	Long BaseLine
MCU	Microcontroller
PWM	Pulse Width Modulation
ROV	Remote Operated Underwater Vehicle
SLAM	Simultaneous Localization and Mapping
SBL	Short BaseLine
USBL	Ultra-short BaseLine

SUMÁRIO

1 INTRODUÇÃO

Com o aumento de aplicações submarinas relacionadas à indústria do petróleo e gás, ciências marinhas e prevenção de desastres marinhos, junto à evolução e ao crescimento das tecnologias, os veículos autônomos submarinos (AUVs) estão se tornando ferramentas cada vez mais úteis e acessíveis. Algumas das aplicações submarinas mais comuns são: na indústria de gás e petróleo, com a inspeção e reparos de infraestruturas submersas; busca e recuperação, com a localização e manipulação de objetos no fundo do mar; na arqueologia marinha, com a documentação e mapeamento em 2D e 3D de sítios localizados no fundo do mar; na ciência marinha, com a manutenção periódica de observatórios submarinos, assim como em pesquisa e coleta de amostras químicas, geológicas e biológicas.

A utilização de robôs não tripulados é relevante por permitir a inspeção e intervenção em ambientes hostis e de risco ao ser humano. Atualmente uma das ferramentas mais utilizadas pela indústria são os veículos subaquáticos remotamente operados (ROVs). Estes robôs são ligados à superfície via umbilical, sendo operados remotamente por uma equipe alocada a um veículo suporte de superfície. A necessidade de equipe especializada e veículos de suporte geram altos custos, tornando as missões envolvendo ROVs operações dispendiosas (INSAURRALDE; PETILLOT, 2015). O uso de robôs autônomos não tripulados é uma solução promissora, sendo que, quanto mais automatizados os processos, com redução de equipe e necessidade de intervenção humana, menores os riscos e custos gerados. A Figura 1 mostra alguns exemplos comerciais de AUVs e ROVs utilizados pela indústria de gás e petróleo.

Figura 1 – Diferentes tipos de ROVs e AUVs.

Fonte: Shukla e Karki (2016).

AUVs operam em ambientes dinâmicos, sem ou com suporte externo limitado e, quanto mais longa uma missão, maior a chance de ocorrência de uma falha. Em operações residentes, por exemplo, AUVs operam por longos períodos de tempo e, em um cenário ideal, deveriam ser capazes de operar sem nenhum suporte externo. Para minimizar os riscos de perda ou danos ao veículo e melhorar as chances de completar uma missão, é essencial que o sistema seja capaz de suportar e tomar ações quando exposto à eventos não previstos, reajustando a missão ou recuperando o AUV, quando necessário.

Este trabalho tem como propósito auxiliar pesquisadores e estudantes, fornecendo uma alternativa de arquitetura aberta para veículos autônomos submarinos, com foco em pesquisa, tolerância a falhas e resiliência.

Um projeto de robótica é multidisciplinar, variando de acordo com a aplicação e necessitando de uma equipe diversificada. O escopo

deste projeto foi limitado à soluções envolvendo conhecimentos principalmente na área de sistemas eletrônicos e computacionais, culminando no desenvolvimento de um protótipo que pode ser adaptado para executar em diferentes AUVs (ou conversão de ROVs para AUVs), capaz de fornecer uma base de trabalho na qual os usuários possam testar diferentes implementações de inteligência e controle, ao mesmo tempo em que executa rotinas de segurança em segundo plano.

O presente trabalho contém um levantamento do estado da arte dos AUVs, estratégias de arquitetura adotadas por outros projetos, estudo de possibilidades, proposta de arquitetura com tolerância a falhas para AUVs e criação de um protótipo de bancada.

1.1 IDENTIFICAÇÃO DOS OBJETIVOS

É considerada a aplicação da arquitetura em ambientes de pesquisa e experimentação. Em um ambiente de pesquisa universitário os usos e implementações no robô variam, dentre eles: experimentos com diferentes algoritmos de controle, inteligência artificial e equipamentos acoplados.

Deseja-se reduzir ao máximo a probabilidade de danos e perda do robô, pois o risco associado a ocorrência de falhas em testes com uma inteligência ou controle recém implementado são grandes, e os resultados raramente são conhecidos ou previstos com antecedência. Deste modo, um sistema complementar de segurança é desejado, com desempenho e funcionalidade já validado previamente, que seja capaz de minimizar danos na ocorrência de falhas das implementações em teste.

É desejado o desenvolvimento de uma arquitetura que contemple os seguintes requisitos:

- Desenvolvimento de uma arquitetura de hardware para AUVs que ofereça uma estratégia de estruturação e organização dos equipamentos;

- Desenvolvimento de uma arquitetura de software para AUVs que ofereça uma base na qual será inserida a inteligência do AUV;
- A arquitetura proposta deve prever a implementação de técnicas de detecção e tolerância a falhas, apresentando flexibilidade para modificações de modo a tratar eventos e falhas, esperados ou não;
- A plataforma desenvolvida deve ser flexível e aberta à modificação e desenvolvimento por terceiros.

1.1.1 Objetivo Geral

Propor uma arquitetura tolerante a falhas e aberta para robôs submarinos autônomos e implementação de protótipo de bancada.

1.1.2 Objetivos Específicos

Os seguintes objetivos específicos servirão de base para alcançar os objetivos gerais:

- Levantamento do estado da arte dos robôs subaquáticos não tripulados;
- Análise de arquiteturas de hardware para AUVs;
- Análise de arquiteturas de controle para AUVs;
- Análise de técnicas de detecção e tolerância a falhas;
- Propor uma arquitetura tolerante a falhas para AUVs;
- Implementação de bancada para testes;
- Testes de bancada.

1.2 ESTRUTURA DO TRABALHO

O restante do presente trabalho está estruturado da seguinte forma:

No Capítulo 2, são explicados os principais conceitos que embasam o projeto.

O Capítulo 3 apresenta o diagrama geral do projeto e metodologia de desenvolvimento empregada.

O Capítulo 4 apresenta os componentes utilizados, estrutura e montagem do protótipo de bancada.

O Capítulo 5 apresenta os testes realizados no protótipo de bancada e resultados obtidos.

O Capítulo 6 apresenta as considerações finais sobre o trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 CARACTERÍSTICAS DE AUV

Nesta seção serão apresentadas as principais e mais comuns características dos robôs subaquáticos autônomos.

2.1.1 Cinemática e Dinâmica

Em geral, um veículo subaquático possui seis graus de liberdade (sem levar em conta manipuladores), três definindo a posição e três indicando a orientação. A partir destas informações é possível identificar o estado de um veículo em relação à um sistema coordenado de referência.

No estudo da dinâmica do AUV deve ser analisada a hidrodinâmica, definida como o estudo das forças e momentos causados pelo fluido no qual o corpo está submerso; a gravidade e flutuabilidade, que são forças aplicadas aos corpos submersos em fluido; e os propulsores / lemes presentes no veículo. Maiores informações sobre este tema podem ser obtidas em Siciliano e Khatib (2008) e Antonelli (2016).

2.1.2 Sensores

Os veículos submarinos devem ser sensoreados para que a navegação e localização sejam possíveis, além de serem instalados sensores específicos voltados à necessidade da missão a ser executada. Combinações de diferentes sensores podem ser empregados para o mesmo fim, podendo ser utilizados aqueles que melhor se adequem a determinadas situações, além de poderem ser empregadas técnicas de fusão de sensores.

Dentre os sensores utilizados para navegação e localização temos:

- Compasso: Bússola giroscópica ou magnética utilizada para estimar a posição do norte geográfico ou magnético, respectivamente. O giroscópio costuma ser mais comum em aplicações marinhas por não ter seu desempenho afetado por objetos com forte assinatura eletromagnética;

- Magnetômetro: Capaz de identificar a intensidade e direção de um campo eletromagnético. Pode ser utilizado como detector de metais e compasso magnético / bússola;
- Unidade de Medição Inercial (IMU): Combinação de giroscópios e acelerômetros para estimar a velocidade linear e angular do robô, bem como as forças gravitacionais;
- Sensor de Profundidade / Pressão: A pressão da água pode ser medida para se determinar a profundidade do robô;
- Sensor de velocidade (DVL): Emite ondas acústicas contra o fundo do mar, medindo a reflexão das mesmas. É utilizado para determinar a velocidade do veículo em relação ao fundo do mar ou coluna de água;
- Sistema de Posicionamento Global (GPS): Quando o AUV se encontra na superfície é possível utilizar o GPS para se obter a localização do robô, calculada com base no tempo de emissão e recepção de sinais enviados via satélites sincronizados;
- Sonar: Sistemas de detecção de objetos na água baseados no uso de ondas acústicas. Podem ser utilizados na detecção de obstáculos, mapeamento e comunicação acústicas;
- Visão / Câmeras: Possuem inúmeras funções, incluindo detecção de objetos, obstáculos e mapeamento.

Dentre os sensores utilizados para segurança podem ser encontrados os seguintes itens:

- Consumo de energia;
- Pressão Interna;
- Temperatura Interna;
- Detector de Vazamento;

- Detector de Falha de Energia.

Sensores de segurança tem como objetivo auxiliar na detecção de problemas e minimizar danos ao robô, acionando sistemas de segurança e recuperação sempre que necessário. O Robô STARFISH (SANGEKAR; CHITRE; KOAY, 2008) possui sensores de vazamento e, ao detectar qualquer infiltração de água, automaticamente corta a alimentação dos sistemas elétricos.

O uso de câmeras e avanços no campo da visão computacional também trouxeram ganhos aos AUVs. Segundo Bonin-Font et al. (2015), a utilização de sistemas visuais traz benefícios claros nas capacidades de exploração e manipulação dos sistemas robóticos submarinos. Em sua pesquisa, criou-se um sistema, denominado Fugu-f, com o objetivo de providenciar informação visual a tarefas submarinas, tais como: navegação, inspeção, mapeamento e intervenção. O dispositivo desenvolvido no projeto é um módulo externo, podendo ser acoplado a qualquer veículo submarino com capacidade de carga.

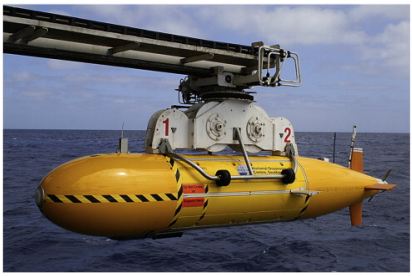
Apesar dos sensores visuais terem um alcance curto, em comparação com outros sensores como os acústicos, e sua performance ser limitada pelas condições do ambiente aquático, a inspeção visual é muito útil na detecção de objetos, principalmente a curto alcance. Kim et al. (2014) propõe um sistema visual para detectar pontos de referências em ambientes controlados.

2.1.3 Propulsores

Geralmente os veículos subaquáticos utilizam propulsores com hélices para se deslocar. Para que um AUV tenha 6 graus de liberdade, é necessária a instalação de pelo menos 6 propulsores. AUVs comumente empregados em exploração e inspeção marinha possuem formato de torpedos, utilizando um ou dois propulsores (À exemplo o NERC Autosub6000, apresentado na Figura 2).

Dependendo da aplicação, como as de intervenção, existem AUVs com designs diversificados, capazes de navegar por terrenos mais com-

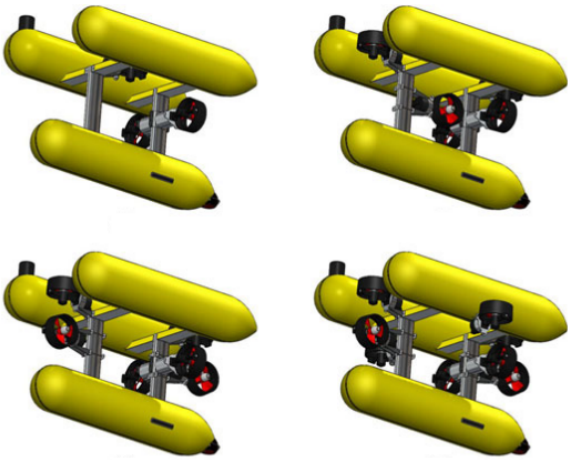
Figura 2 – NERC Autosub6000.



Fonte: Sorensen e Ludvigsen (2015).

plexos. A figura 3 apresenta diferentes configurações e posicionamentos de propulsores em um Girona 500.

Figura 3 – Algumas possibilidades de configuração de propulsores no Girona 500.



Fonte: Adaptado de Ribas, Ridao e Carreras (2012).

2.1.4 Navegação e Localização

A precisão de navegação é dada por quão bem um AUV é capaz de se deslocar de um ponto a outro. Precisão de localização é dada por quão bem um AUV se localiza dentro de um mapa. A navegação e localização de robôs submarinos são desafios atuais e temas de diversas pesquisas no mundo acadêmico e industrial, pois o uso de tecnologias frequentemente empregadas no meio terrestre, como o GPS e comunicação via rádio, não podem ser empregados no meio subaquático, devido principalmente à rápida atenuação destes sinais neste tipo de ambiente.

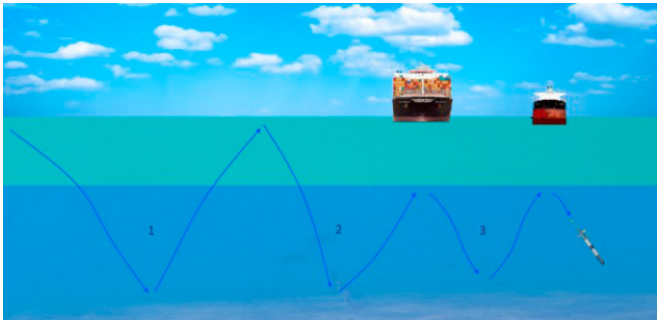
Em geral os métodos de navegação e localização submarina podem ser divididos em três categorias:

- Inercial;
- Uso de transpônderes e modems acústicos;
- Geofísico.

Segundo Braga, Calado e Sousa (2015), o sucesso e a eficiência de uma missão submarina depende de quão bem o robô é capaz de georeferenciar os dados coletados durante a operação. A maior parte das soluções do mercado utilizam métodos inerciais, estimando as posições do robô por meio da correlação dos dados coletados pelos sensores debaixo d'água e a estimativa do deslocamento, de acordo com os motores de propulsão, com a informação obtida pelo GPS quando o robô retorna à superfície. Quando o robô opera em águas turbulentas ou hostis, os AUVs podem perder sua posição e a precisão do local dos dados coletados. A Figura 4 mostra o deslocamento de um AUV *Glider*.

Para assegurar a qualidade dos dados obtidos, são buscadas formas e métodos para melhorar a precisão da localização e posicionamento dos AUVs. Uma possibilidade é utilizar um veículo de superfície não tripulado com um GPS acoplado, que seja capaz de identificar o posicionamento dos veículos subaquáticos, com um sistema de posicionamento acústico por triangulação, a exemplo do *ultrashort baseline* (USBL), que utiliza uma combinação de transceptores para emissão e leitura de

Figura 4 – AUV *Glider* em deslocamento padrão e desviando de obstáculos.



Fonte: Busquets-Mataix (2015).

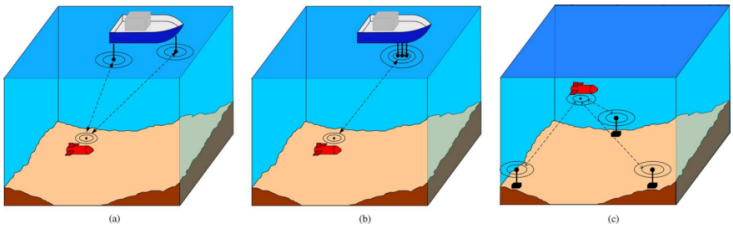
ondas acústicas (instalado no veículo de superfície) e modens acústicos montados no veículo submarino.

A Figura 5 apresenta métodos clássicos de navegação baseados no uso de transpônderes e sinalizadores, com a localização sendo calculada a partir do tempo de emissão e retorno das ondas acústicas. Em (a) é possível observar a aplicação do *short baseline* (SLB), no qual sinalizadores são instalados nas extremidades de um veículo de superfície. Em (b) é empregado o USBL, no qual os transdutores são posicionados próximos uns dos outros. Em (c) é aplicado o *long baseline* (LBL), com sinalizadores posicionados no fundo do mar e a localização sendo calculada pela triangulação dos sinais.

Métodos geofísicos utilizam características do ambiente externo para calcular a localização do robô. Um exemplo de aplicação desta técnica é dado por Bachmann e Williams (2003): quando o terreno e mapas batimétricos são conhecidos, é possível obter a localização do veículo medindo-se a profundidade do fundo do mar com um sonar.

O avanço de tecnologias de mapeamento e localização simultânea (SLAM) tem trazido maior flexibilidade e precisão na navegação dos AUVs, a custos reduzidos. Esta técnica se baseia na identificação de características no ambiente para conseguir localizar o veículo, podendo

Figura 5 – Métodos acústicos empregando transpônderes e sinalizadores: (a) SBL; (b) USBL; (c) LBL.



Fonte: Paull et al. (2014).

ser aplicada com câmeras ou sonares.

O uso de sensores inerciais, instalação de sinalizadores ou guias na região de interesse, ou voltar periodicamente o AUV à superfície resolvem apenas parcialmente o problema, pois são soluções com limitações específicas e, na maioria das vezes, com altos custos. A escolha do método de navegação a ser empregado depende do tipo de operação e local da missão, sendo que combinações de diferentes sensores e métodos podem ser utilizados, de modo a se obter melhores resultados.

2.1.5 Controle

Controle em veículos submarinos pode ser definido como a geração e aplicação de forças e momentos que possibilitem o controle do ponto de operação, rastreamento e estabilidade do robô. Isto é feito utilizando ferramentas computacionais, implementando leis de controle com *feedfoward*[1] e *feedback*[2] (SICILIANO; KHATIB, 2008).

Como os robôs subaquáticos enfrentam problemas de diferentes complexidades e incertezas, muitas vezes os sistemas de controle tradicionais não são suficientes para atender as necessidades do AUV. Uma possível solução é o uso de sistemas híbridos, formados pela combinação de múltiplos sistemas de controle.

[1] O sistema responde a um sinal de controle de maneira pré-definida.
[2] O sistema responde a um sinal de controle levando em conta o comportamento do sistema monitorado.

Busquets-Mataix (2015) apresenta um AUV que aproveita sistemas que utilizam o princípio de flutuabilidade para auxiliar no seu deslocamento, empregando uma combinação de gás (ar comprimido) e líquido (óleo) para otimizar e melhorar a capacidade de deslocamento, tendo como objetivo operar em longas missões no oceano.

2.1.6 Manipulação e Intervenção

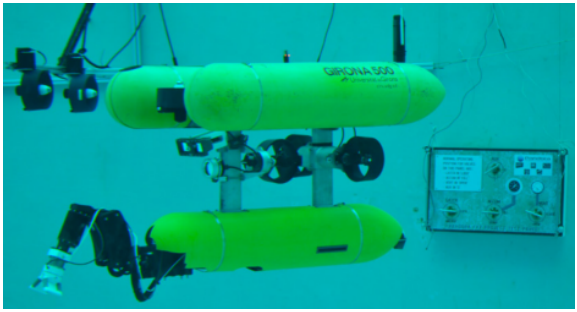
Uma linha da robótica submarina autônoma cuja demanda vem crescendo nos últimos anos é a de intervenção, a qual exige a instalação de manipuladores no veículo submarino. São exemplos de aplicações na área de manutenção de observatórios, cabos, recuperação de itens, abertura e fechamento de válvulas, conectar e desconectar conectores, etc. Nos dias atuais, estas tarefas têm sido comumente designadas a ROVs, mas, devido aos seus altos custos de equipamento e pessoal (RIDAO et al., 2015), as pesquisas e interesse por AUVs capazes de executar missões de intervenção vêm crescendo cada vez mais.

O projeto de Peñalver et al. (2015) estuda a manipulação de painéis com base fixa, necessitando que o AUV seja capaz de localizar o painel, acoplar-se a ele e realizar operações como conectar e rotacionar válvulas.

Um estudo de manipulação de painéis sem base fixa (que flutua livremente), realizado por Carrera et al. (2015), envolve a combinação das capacidades de deslocamento do AUV para compensar as limitações da área de alcance de operação do manipulador, com inteligência artificial (AI) e técnicas de aprendizado de máquina para treinar o AUV na reprodução de tarefas autonomamente. Um de seus experimentos pode ser observado na Figura 6.

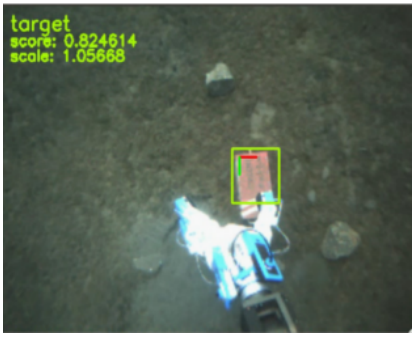
Bonin-Font et al. (2015) descreve a localização e recuperação de objetos utilizando tecnologias de visão computacional. A Figura 7 apresenta um de seus testes de recuperação e manipulação de objetos.

Figura 6 – Girona 500 AUV com um manipulador acoplado, em um experimento de intervenção em painel.



Fonte: Carrera et al. (2015).

Figura 7 – Identificação de objetos e intervenção.



Fonte: Bonin-Font et al. (2015).

2.1.7 Cooperação de AUVs

Sistemas cooperativos são úteis por permitirem maior flexibilidade na execução de missões, maior velocidade de mapeamento de áreas e oferecerem diferentes possibilidades de obtenção da localização dos AUVs (utilizando seus próprios sensores ou por meio da troca de informações com outros membros). Roumeliotis e Bekey (2002) provam que um grupo de agentes autônomos, mesmo sem acesso a informações de posicionamento global, são capazes de se localizar com maior precisão

se compartilharem estimativas de posições e dados de mapeamento individuais.

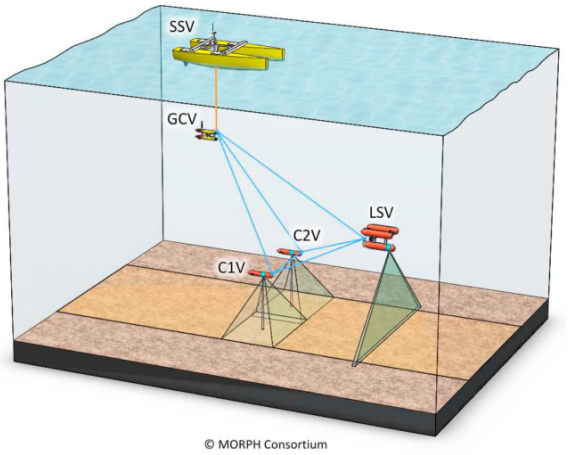
Sistemas cooperativos podem ser formados por times de AUVs heterogêneos ou homogêneos. No primeiro caso, AUVs equipados com diferentes sensores e funções podem ser utilizados. Esta estratégia é útil por permitir que sensores de alto custo sejam instalados em apenas alguns AUVs, que compartilham as informações das leituras obtidas com os outros. Uma possível formação heterogênea é utilizar um grupo de AUVs de suporte, com sensores de navegação mais avançados, e um grupo de inspeção, com sensores de posição mais simples, que tem como objetivo realizar medições específicas de acordo com os objetivos da missão. Em um grupo homogêneo de AUVs todos os membros do time são tratados igualmente, compartilhando informações.

Kalwa et al. (2015) propõe um sistema cooperativo heterogêneo cujos AUVs são capazes de se comunicar entre si via redes de comunicação acústicas. Neste sistema, as tarefas necessárias para mapear o fundo do mar são distribuídas entre robôs distintos, sendo que um deles é um veículo de superfície, com um GPS acoplado e um robô auxiliar embaixo d'água, posicionado longe da influência das ondas do mar e responsável por estabelecer a comunicação entre os robôs responsáveis pela coleta de dados. Estes robôs também utilizam o USBL como forma de se comunicar entre si e, como a distância é reduzida, ganha-se precisão. A Figura 8 mostra uma possível configuração do sistema cooperativo proposto:

2.1.8 Comunicação

Para que as soluções cooperativas de posicionamento e a localização sejam possíveis, é necessário que os agentes da missão consigam se comunicar de modo eficiente. Esta é uma tarefa complexa para os AUVs, pois o sinal deve se propagar pela água, estando sujeito a problemas tais como: propagação múltipla, atenuação, variação no tempo de propagação pelo meio, baixa largura de banda, frequente perda de dados, dentre outros (PAULL et al., 2014).

Figura 8 – Sistema de mapeamento cooperativo proposto por Kalwa et al. (2015).



Fonte: Kalwa et al. (2015).

Diversos estudos são voltados às comunicações submarinas, um meio no qual a comunicação é restrita e sujeita a perda de dados. Walls e Eustice (2014) apresentam um estudo em que se utiliza comunicação acústica com robôs sincronizados via *clock* do *hardware*, conseguindo utilizar a informação do horário de envio de mensagens e de recebimento para melhorar a navegação dos veículos submarinos. Petrioli et al. (2015) apresentam a ferramenta SUNSET, criada com o objetivo de auxiliar na comunicação entre equipamentos, desde simulações até testes em campo, permitindo configuração e controle remoto em tempo real, por meio de *links* acústicos.

2.1.9 Docking System e Recarga de Energia

Um limitador na versatilidade dos robôs subaquáticos autônomos são as restrições causadas pela necessidade de reposição energética. Veículos subaquáticos remotamente operados (ROVs) permanecem conectados à superfície por meio de um umbilical, com a desvantagem

dos custos adicionais associados à embarcação de apoio, operadores remotos e à conexão entre o ROV e a superfície. Já os AUVs, apesar de possuírem maior autonomia, possuem restrições associadas à capacidade de armazenamento energético e de dados. Uma das áreas de estudo dentro dos campos dos AUVs são os sistemas de acoplamento (*docking*) para veículos subaquáticos, que possam ser utilizados para recarga de energia, atualização da missão, coleta de dados, dentre outras possibilidades. Mintchev et al. (2014) apresentam e demonstram um sistema de *docking* para AUVs em miniatura, utilizando alinhamento magnético, tendo como desafio futuro realizar ajustes no projeto para possibilitar aplicações em maiores escalas.

2.1.10 Inteligência Artificial

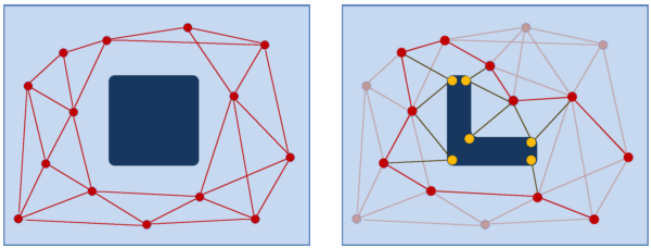
A autonomia de um robô é dada pela sua capacidade de operar por longos períodos de tempo em ambientes dinâmicos, sem intervenção humana. Para que isto seja possível, os robôs devem ter uma boa capacidade de planejamento, adaptação e interpretação de eventos.

Em geral um sistema inteligente é capaz de planejar e aprender. Durante o planejamento, o robô delibera sobre qual sequência de ações deve tomar para atingir objetivos. No aprendizado, o robô busca melhorar sua performance a partir das suas experiências (SICILIANO; KHATIB, 2008).

Ao iniciar missões de inspeção, o AUV possui apenas uma visão parcial do mundo, das posições e tipos de estruturas a serem encontradas. O AUV inicia sua missão visitando pontos de inspeção baseados no seu pré-conhecimento sobre o ambiente, mas, conforme dados são encontrados e interpretados, estes pontos podem mudar. A Figura 9 apresenta uma missão de inspeção de uma estrutura submersa, no qual é feito o pré-planejamento de trajetória com o mapa sendo atualizado a partir da leitura dos sensores. As linhas e pontos vermelhos indicam o caminho a ser percorrido pelo robô e os amarelos pontos de inspeção.

Em uma missão de intervenção em um painel subaquático, mesmo que o local de trabalho seja conhecido, a sua execução varia de acordo

Figura 9 – Missão de inspeção de uma estrutura submersa.



Fonte: Cashmore et al. (2015).

com as condições do ambiente e posição inicial do veículo, dentre outros fatores (CASHMORE et al., 2015).

Para que sejam reativos às mudanças no ambiente, os robôs autônomos devem ser capazes de tomar decisões em tempo real, levando em conta o objetivo da missão, leituras dos sensores e restrições de tempo. Muitas tarefas devem ser executadas em paralelo, por exemplo: enquanto se realiza uma tarefa com o manipulador o robô deve ser mantido estável através do acionamento dos propulsores, ou utilizar um segundo braço para se fixar em alguma estrutura.

2.2 PLATAFORMAS ABERTAS

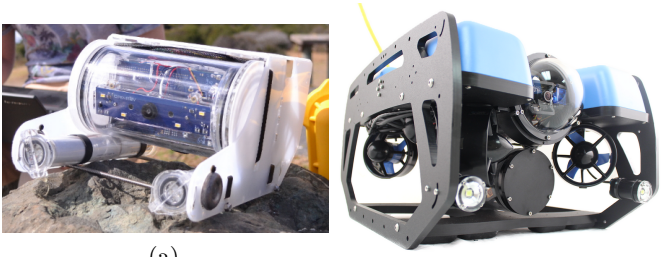
Plataformas abertas podem ser definidas como plataformas cuja tecnologia é disponível para modificação e desenvolvimento por terceiros. Para Eisenmann, Parker e Van Alstyne (2009) uma plataforma aberta:

1. Não possui restrições na participação do seu desenvolvimento, comercialização ou uso;

2. Qualquer restrição, como por exemplo o pagamento de um valor de licença, são razoáveis e aplicados igualmente a todos os participantes.

Uma plataforma com arquitetura aberta facilita o processo de desenvolvimento de novas tecnologias a partir de outras já existentes,

Figura 10 – Exemplos de ROVs de tecnologia aberta: (a) OpenROV 2.8; (b) BlueROV2.



(a) (b)

(a) Fonte: OpenROV[1]. (b) Fonte: BlueRobotics[2].

seguindo padrões já reconhecidos industrialmente (de interfaces de comunicação, por exemplo), permitindo trocas, adições e melhorias. Dentre os robôs submarinos de tecnologia aberta, podem ser citados como exemplo:

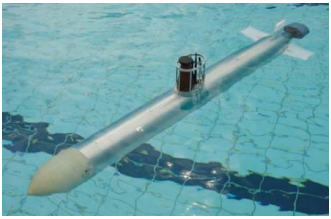
- ROVs
  - *OpenROV* (Figura 10a);
  - *BlueROV* (Figura 10b).
- AUVs
  - STARFISH (Figura 11a);
  - Alba-14 (Figura 11b);
  - SPARUS II (Figura 11c).

2.3 ARQUITETURAS

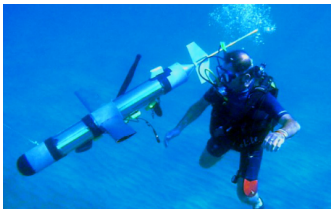
Nesta seção serão estudadas abordagens e arquiteturas de hardware e controle que possam atender os objetivos propostos.

[1] <http://www.openrov.com/>
[2] <http://www.bluerobotics.com/>
[3] Hardware aberto para integração de equipamento

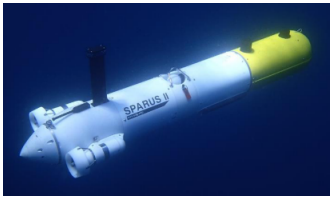
Figura 11 – Exemplos de AUVs de tecnologia aberta: (a) STARFISH; (b) Alba-14 HGL Glider; (c) Sparus II[1].



(a)



(b)



(c)

(a) Fonte: Sangekar, Chitre e Koay (2008). (b) Fonte: Busquets et al. (2015). (c) Fonte: Carreras et al. (2015).

2.3.1 Arquitetura de Hardware

A arquitetura de hardware define a distribuição dos sistemas e componentes físicos do robô, assim como as suas relações. De acordo com Jo et al. (2014) é possível classificar uma arquitetura como centralizada ou distribuída.

- **Arquitetura de hardware centralizada:**

  Um único setor computacional é responsável por múltiplas funções e tarefas. A unidade de controle deve ler e interpretar as entradas de sensores, tomar decisões e acionar atuadores. Esta estratégia possui custos reduzidos se comparada a uma arquitetura distribuída, com a programação sendo feita e atualizada em uma unidade central. Sua construção e montagem inicial é geralmente simples, sendo uma solução vantajosa para projetos de pequeno porte, baixa complexidade e baixo orçamento.

As principais desvantagens dessa arquitetura são a necessidade de uma unidade central com maior capacidade computacional, sua baixa robustez (caso a unidade central apresente problemas), baixa portabilidade e flexibilidade (unidades centrais são mais sofisticadas, de difícil manutenção e modificação).

- **Arquitetura de hardware distribuída:**

  Na arquitetura distribuída, múltiplos controladores podem ser empregados em subsistemas no AUV, compartilhando um sistema de comunicação comum. Esta é uma arquitetura mais robusta, sendo que, no caso de um setor apresentar erros, devidas ações podem ser tomadas pelos outros setores.

  Esta arquitetura possui flexibilidade e manutenção mais simples que uma arquitetura centralizada, com maior facilidade na identificação de problemas e permitindo a modularizarão de setores (cada módulo pode ser desenvolvido e testado independentemente dos outros). É possível reduzir o nível de ruído eletromagnéticos e custo de cabeamento dependendo do posicionamento dos módulos computacionais, deixando-os próximos aos sensores / atuadores com os quais interagem diretamente. Em contra partida, sistemas distribuídos costumam apresentar maiores custos que a arquitetura centralizada e uma arquitetura de hardware mais complexa.

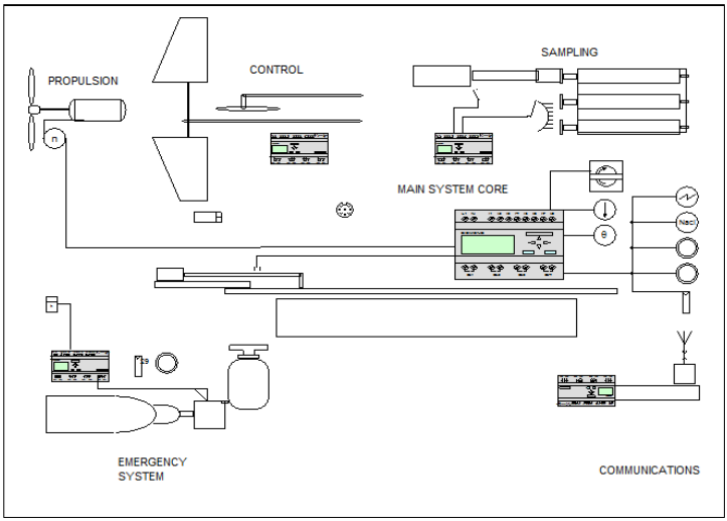
- **Exemplos de arquitetura de hardware apresentados na literatura:**

  O AUV Alba13 (BUSQUETS et al., 2015) utiliza um microcontrolador Arduino[1] para cada setor do veículo, conectados à uma placa central (Figura 12). Em caso de falha do processador central (*Main System Core*), os processadores periféricos são capazes de executar os procedimentos de emergência necessários.

  O AUV STARFISH (SANGEKAR; CHITRE; KOAY, 2008) também emprega uma arquitetura distribuída, utilizando como interface padrão o protocolo *RS-232* para comunicação entre setores.

[1] https://www.arduino.cc/

Figura 12 – Esquemático funcional do AUV Alba13.

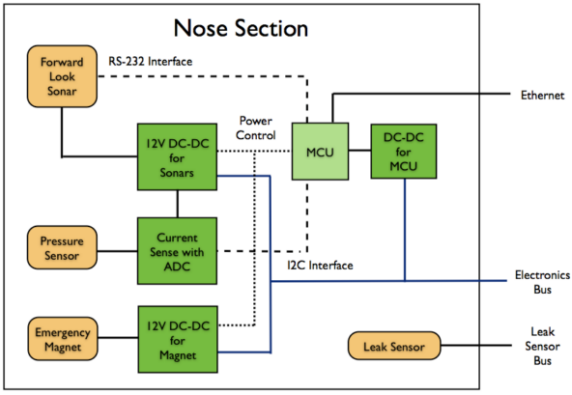


Fonte: Busquets et al. (2015).

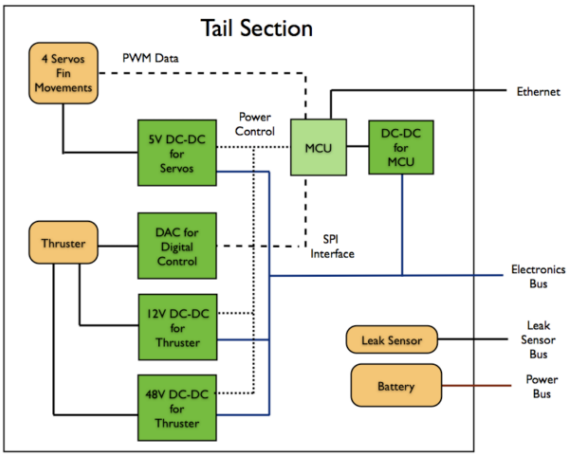
Cada setor é capaz de ler e interpretar seus sensores, compartilhando a informação processada com o resto do veículo via *Ethernet* (os atuadores e sensores também podem ser acessados diretamente por outros setores). Cada setor possui baterias e conversores de energia específicos para alimentar os seus sensores e atuadores, com a energia sendo fornecida por um setor principal, com baterias de maior capacidade. Exemplos de setores do STARFISH são apresentados na Figura 13.

Bian et al. (2012) utilizam um AUV chamado BSA-AUV para realizar missões em ambientes oceânicos. O AUV dispõe de diversos tipos de sensores, sendo capaz de medir diferentes parâmetros em diversos tipos de ambiente (considerando o oceano como uma área extensa, contendo sub-áreas distintas). O BSA-AUV é capaz de mergulhar até $200m$ e manter uma velocidade média de $1.5m/s$ por até $300km$, antes de necessitar recarregar suas baterias. Uma visão geral da sua arquitetura é apresentada na Figura 14.

Figura 13 – Exemplos de setores do STARFISH: (a) Nariz (*Nose Section*); (b) Calda do AUV (*Tail Section*).



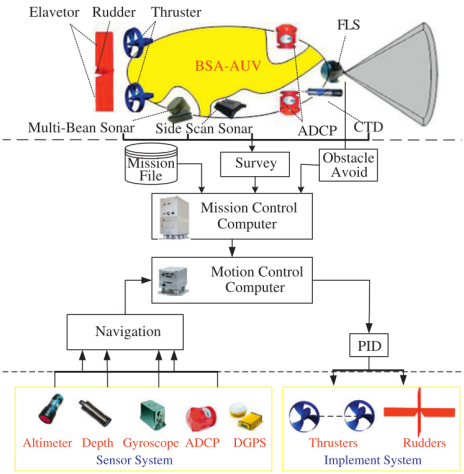
(a)



(b)

Fonte: Sangekar, Chitre e Koay (2008).

Figura 14 – Arquitetura do sistema do BSA-AUV.



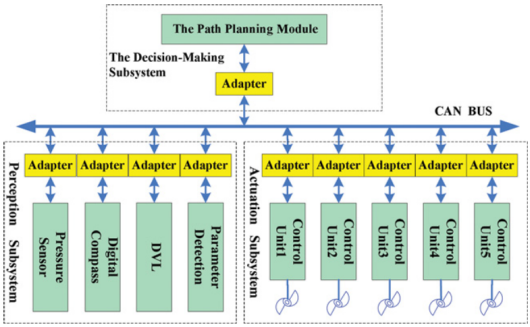
Fonte: Bian et al. (2012).

O BSA-AUV possui duas categorias de sensores, sensores utilizados para controlar da movimentação e localização do veículo e sensores utilizados nas tarefas de inspeção oceânica. Dois computadores industriais PC104 são utilizados no controle do AUV, sendo um responsável pela missão e decisões em alto nível (*Mission Control Computer*) e outro pelo controle em tempo real e decisões em baixo nível (*Motion Control Computer*).

He et al. (2013) apresentam uma arquitetura de hardware paralela distribuída (Figura 15), formada por quatro blocos: comunicação, decisão (*Decision-Making Subsystem*), percepção (*Perception Subsystem*) e atuador (*Actuation Subsystem*). A comunicação é feita via protocolo CAN (*Controller Area Network*), ao qual cada subsistema é conectado utilizando adaptadores. As tomadas de decisão são realizadas em um computador industrial PC104. O bloco de percepção é constituído pelos sensores do AUV (dentre eles compasso digital, DVL e medidor de pressão). O bloco atuador inclui os propulsores (acionados por motores de corrente contínua

sem escova, via PWM, utilizando um microcontrolador) e *drivers* de potência.

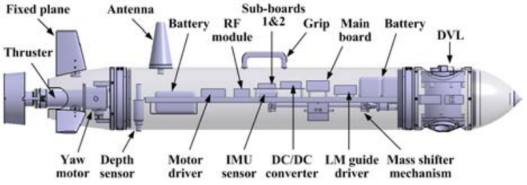
Figura 15 – Arquitetura de hardware paralela distribuída utilizada no controle de um AUV C-Ranger.



Fonte: He et al. (2013).

Loc et al. (2014) criaram o KAUV-1, um AUV de formato de torpedo, leve e de tamanho reduzido, que utiliza um sistema mecânico para alterar o seu centro de massa. O AUV é controlado por um computador principal e dois secundários, dividindo tarefas de leitura e interpretação de sensores, acionamento de atuadores e comunicação via rádio (quando o veículo se encontra na superfície). A estrutura interna do KAUV-1 pode ser observada na Figura 16.

Figura 16 – Vista interna do KAUV-1.



Fonte: Loc et al. (2014).

2.3.2 Arquitetura de Controle

A arquitetura de controle é responsável pelo comportamento do robô, na maneira como o robô percebe o ambiente a sua volta, planeja suas ações e reage a diferentes situações. Segundo Verhoeckx (2016) um sistema de controle ideal deve:

1. Realizar movimentações no decorrer do tempo utilizando trajetórias eficientes, com gasto mínimo de energia;
2. Evitar *deadlocks*;
3. Evitar colisões;
4. Apresentar flexibilidade em relação às possíveis rotas e destinos;
5. Ser de fácil implementação em um novo ambiente;
6. Possuir escalabilidade em relação ao número de agentes (robôs cooperativos);
7. Agir com previsibilidade;
8. Ser robusto contra perturbações;
9. Atingir os objetivos atribuídos;
10. Apresentar um sistema de controle modular.

Antonelli (2016) sita a possibilidade de um sistema de arquitetura modular utilizar estratégias hierárquicas, com tarefas ou componentes podendo ter diferentes prioridades de execução.

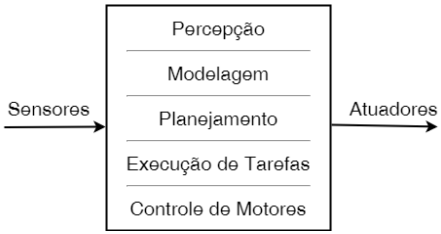
De acordo com Nakhaeinia et al. (2011), as arquiteturas de controle podem ser classificadas como deliberativas, reativas ou híbridas.

- **Arquitetura de controle deliberativo:**

  O sistema deliberativo possui três fases principais: sensoreamento, planejamento e ação. Na primeira fase, o robô cria um modelo simplificado do mundo ao seu redor a partir das informações

obtidas pelos sensores. Em seguida executa a fase de planejamento para atingir o seu objetivo. Finalmente, na fase de ação, o robô executa as ações planejadas. Este ciclo é repetido até que o robô atinja o seu objetivo final. Este é um método que necessita de alta capacidade computacional e memória, com seu desempenho dependendo da qualidade do modelo gerado, na maioria das vezes não sendo suficiente para operar em ambientes dinâmicos. As fases desta arquitetura podem ser observadas na Figura 17.

Figura 17 – Arquitetura de controle deliberativa.



Fonte: Adaptado de Palomeras et al. (2012).

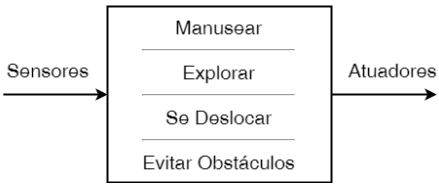
- **Arquitetura de controle reativo:**

  O sistema reativo de controle é capaz de fornecer uma reação rápida a estímulos externos em ambientes dinâmicos, com base na leitura em tempo real dos sensores e no modo de comportamento ativo no robô, sem possuir uma fase de planejamento (Figura 18). No método reativo não é necessário realizar um mapeamento completo do ambiente, possuindo uma menor necessidade de processamento que o controle deliberativo. Suas principais limitações são: capacidade de coordenação de ações (o comportamento do robô frente a um ambiente dinâmico é pouco previsível) e dificuldades no cumprimento de tarefas complexas.

- **Arquitetura de controle híbrido:**

  Em um sistema híbrido a arquitetura deliberativa e reativa são combinadas, sendo comumente formadas por três camadas, como

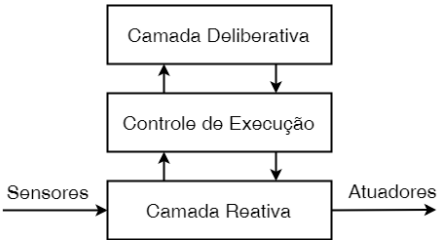
Figura 18 – Arquitetura de controle reativa.



Fonte: Adaptado de Palomeras et al. (2012).

mostra a Figura 19.

Figura 19 – Arquitetura de controle híbrida.



Fonte: Adaptado de Palomeras et al. (2012).

- Deliberativa: Executa o planejamento em alto nível, considerando a leitura dos sensores, o mapeamento e planejamento;
- Controle de execução: Responsável pela iteração entre as camadas de alto e o de baixo nível;
- Reativa: Gera as ações do robô em tempo real e interage com o ambiente.

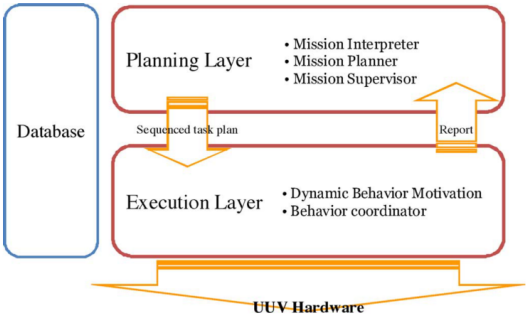
Segundo Nakhaeinia et al. (2011) a arquitetura híbrida é a opção mais promissora para navegação em ambientes desconhecidos e dinâmicos, apresentando flexibilidade e robustez adquiridos a partir da integração de características provindas de outras arquiteturas.

- **Exemplos de arquiteturas de controle híbrido:**

Belbachir, Ingrand e Lacroix (2012) apresentam uma arquitetura híbrida para controle de múltiplos AUVs e um veículo autônomo de superfície (ASV). Nesta arquitetura os AUVs compartilham suas informações, otimizando a navegação de acordo com estimativas de localização de objetivos e obstáculos, empregando estratégias de exploração adaptativas. O ASV é utilizado como uma interface de comunicação entre todos os AUVs e também para refinar a localização dos mesmos. Os AUVs e ASV adotam diferentes horizontes e tempos de deliberação (quanto menor o horizonte, mais reativo e menos deliberativo) no planejamento de cada atividade.

Outra arquitetura de controle híbrida e aberta é proposta por Han, Ok e Chung (2013), composta por duas camadas, um planejador (*Planning Layer*) e uma camada reativa (*Execution Layer*). O planejador cria um plano otimizado, considerando restrições como obstáculos, correntes marítimas e prioridades das tarefas. A camada reativa é responsável pela execução do plano, agindo de acordo com o modo de comportamento ativo. Um supervisor de missão decide quais são as tarefas a serem executadas, podendo modificar o plano original na ocorrência de eventos não previstos. A Figura 20 apresenta a estrutura da arquitetura proposta.

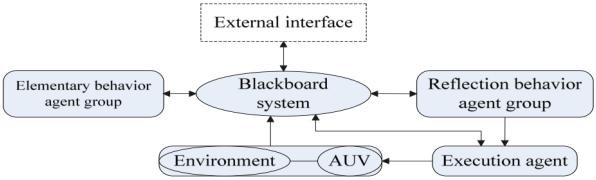
Figura 20 – Arquitetura de controle híbrida proposta por Han, Ok e Chung (2013).



Fonte: Han, Ok e Chung (2013).

Lei et al. (2013) propõe uma arquitetura híbrida composta por quatro partes: uma unidade central para processamento de dados e controle de comportamento (*Blackboard System*), uma unidade de controle responsável movimentação do robô (*Elementary Behavior*), uma unidade de segurança (*Reflection Behavior*) e uma de execução e acionamento dos atuadores (*Execution Agent*). A Figura 21 mostra a arquitetura de controle empregada no AUV ZT-AUV.

Figura 21 – Arquitetura de controle do ZT-AUV, propostos por Lei et al. (2013).



Fonte: Lei et al. (2013).

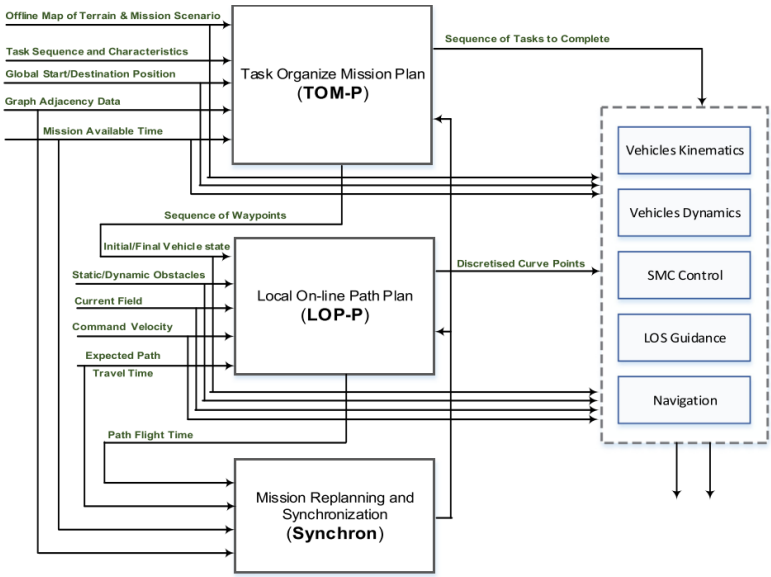
O sistema de controle do ZT-AUV executa cinco tipos de tarefas, utilizando o sistema operacional de tempo real *VxWorks*:

1. Leitura de dados brutos;
2. Processamento e filtragem da informação obtida;
3. Planejamento e controle da movimentação do robô;
4. Alimentação do sistema de planejamento e acionamento dos atuadores;
5. Ações de emergência.

O tempo da missão é levado em conta na priorização das tarefas executadas pela arquitetura híbrida proposta por MahmoudZadeh, Powers e Sammut (2016), composta por planejadores de alto e baixo nível (*TOM-P* e *LOP-P*, respectivamente). O módulo de alto nível é responsável pela priorização das tarefas, guiar o AUV até o seu destino final e gerar uma sequência de tarefas. O

baixo nível é responsável pela execução das tarefas e segurança do veículo, lidando com eventos como detecção de obstáculos ou correntes marítimas. Um módulo de sincronismo (*Synchron*) recebe informações de ambos os módulos e decide se é necessário replanejar o caminho ou missão executados pelo AUV, de acordo com o ambiente percebido e parâmetros de decisão (por exemplo, o tempo restante da missão). A arquitetura proposta pode ser observada na Figura 22.

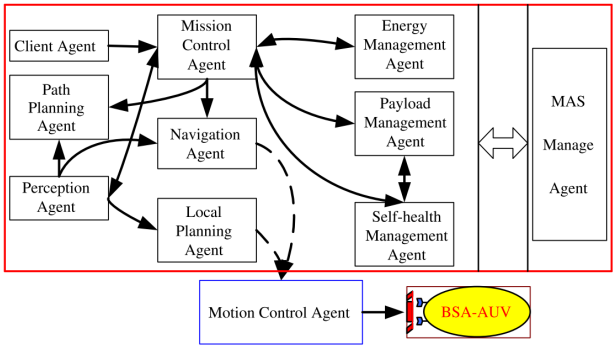
Figura 22 – Arquitetura de controle proposta por MahmoudZadeh, Powers e Sammut (2016).



Fonte: MahmoudZadeh, Powers e Sammut (2016).

Bian et al. (2012) utilizam uma estratégia multi-agente em uma arquitetura híbrida, com divisão de conhecimento e dados entre agentes independentes que cooperam entre si para alcançar objetivos e realizar missões. A arquitetura de controle proposta pode ser observada na Figura 23, no qual são ilustrados os 11 agentes que compõe a inteligência do AUV BSA-AUV.

Figura 23 – Arquitetura de controle multi-agente do BSA-AUV.

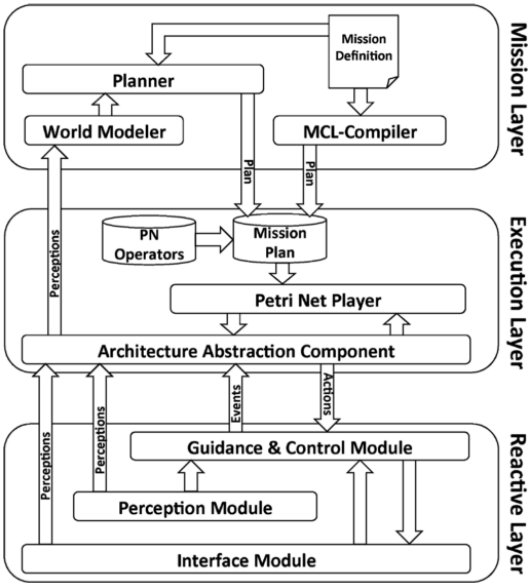


Fonte: Bian et al. (2012).

Palomeras et al. (2012) apresenta uma arquitetura de controle híbrida (COLA2) formada por três camadas: reativa (*Reactive Layer*), de execução (*Execution Layer*) e de missão (*Mission Layer*). Um de seus diferenciais em relação aos métodos convencionais é a aplicação de técnicas de aprendizado por reforço na camada reativa e a utilização de máquinas de estado para representar missões. O aprendizado por reforço possibilita que o robô aprenda através de interações com o ambiente, empregando um sistema de pontuação baseado no estado atual do veículo e da última ação tomada em relação ao objetivo. A arquitetura de controle do COLA2 pode ser observada na Figura 24.

A camada reativa do COLA2 (Figura 25) possui três módulos: interface, composta por *drivers* que interagem com sensores e atuadores; percepção, responsável por receber os dados da interface e processar a informação para detecção de obstáculos, estimar a posição do veículo e velocidade; controle, que inclui ações primitivas (como ir a determinado ponto ou atingir certa profundidade, por exemplo), interpretação das respostas geradas pelas ações e controle das velocidades dos propulsores do AUV. O aprendizado por reforço é utilizado para ajustar os parâmetros das ações primitivas,

Figura 24 – Arquitetura de controle do COLA2.



Fonte: Palomeras et al. (2012).

buscando melhorar a performance do AUV.

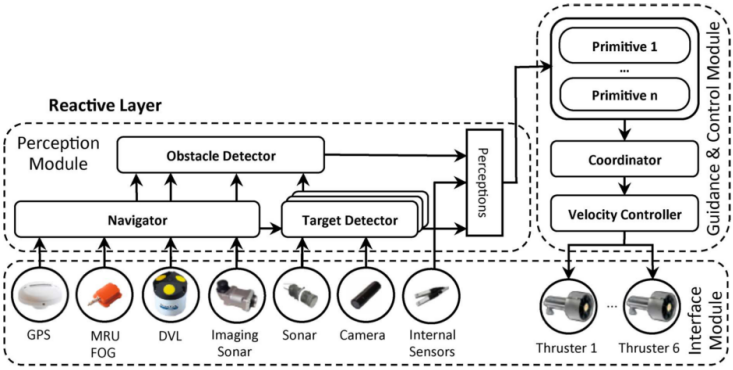
2.4 DETECÇÃO E TOLERÂNCIA A FALHAS

Em missões de longa duração em ambientes dinâmicos (exploração de oceanos (LEI et al., 2013), Antártida (DEARDEN; ERNITS, 2013)), uma falha pode ocasionar na perda do AUV. Deste modo, é importante que o sistema seja capaz de detectar falhas e tomar ações para abortar uma missão ou recuperar o veículo.

Antonelli (2016) definem detecção de falhas e tolerância a falhas da seguinte maneira:

- Detecção de falhas: processo de monitorar um sistema de modo a reconhecer a ocorrência ou presença de uma falha;

Figura 25 – Camada reativa da arquitetura do COLA2.



Fonte: Palomeras et al. (2012).

- Tolerância a falhas: capacidade do sistema completar uma missão mesmo na presença de falhas.

2.4.1 Resiliência

Resiliência pode ser definida como a capacidade do sistema lidar com interrupções, falhas e eventos, esperados ou não, durante uma missão (MARSHALL; ROBERTS; GRENN, 2017). Esta é uma habilidade relevante aos veículos autônomos devidos aos riscos e incertezas associados às missões. O sucesso de uma missão, e mesmo a sobrevivência do veículo, depende fortemente do tratamento e resposta a estes eventos.

Segundo Marshall, Roberts e Grenn (2017), um sistema resiliente possui as seguintes capacidades:

- Evasão: Ação de antecipar e evitar certo evento;
- Adaptação: Ações tomadas para contornar o evento sofrido;
- Resistência: Capacidade de resistir determinado evento (exemplo: margens de segurança);
- Recuperação: Restauração parcial ou completa do sistema após um evento.

Falhas podem ser causadas pelo *hardware*, *software*, ambiente ou interações do robô. Um levantamento dos tipos de falhas mais comuns em AUVs do tipo *glider* foi feito por Brito, Smeed e Griffiths (2014), que acompanhou 205 AUVs durante 4 anos, apontando como principais causas de falha os vazamentos, baterias e sistema de flutuabilidade.

Aslansefat, Latif-Shabgahi e Kamarlouei (2014) propõe organizar as falhas em AUVs em nove grupos:

- Sistema de energia;
- Sistema detecção de vazamentos;
- Sistema de mergulho;
- Sensores de detecção de ambiente;
- Sistemas de anti-colisão;
- Sistema computacional;
- Sistemas de propulsão;
- Sistema de comunicação;
- Sistema de navegação.

Segundo Seto e Bashir (2017), o nível de impacto da falha pode ser classificado como mínima (nenhuma ação necessária), baixa (ação corretiva bem sucedida), perda de funcionalidade (com ação corretiva mas sem recuperação total) e crítica (colisão, baixa bateria).

Koopman e Wagner (2017) argumenta que é possível reduzir a complexidade e custos do projeto, caso exista um estado de segurança secundário para o qual o veículo possa recorrer em caso de falha do sistema de controle principal. Isto é pertinente no uso de veículos autônomos em ambientes de pesquisa, pois concede flexibilidade no desenvolvimento dos sistemas de controle.

Durante uma missão o suporte externo provido aos AUVs é limitado, assim, quanto mais longa a missão, maior a probabilidade de

ocorrência de falhas e eventos não previstos. Isso ilustra a necessidade dos veículos autônomos possuírem uma arquitetura adaptável e robusta.

2.4.2 Redundância

Redundância pode ser definida pela disponibilidade de informação, material ou tempo adicional ao que seria necessário à operação correta do sistema em um cenário ideal (CASTANO; SCHAGAEV, 2015), e é uma das principais ferramentas utilizadas por sistemas tolerantes a falhas.

Um sistema tolerante a falhas deve ser capaz de detectar falhas e utilizar os recursos redundantes de modo a restaurar o funcionamento do sistema. Suas principais desvantagens são os custos adicionais, seja com componentes, maior consumo de energia, espaço que ocupa ou tempo de desenvolvimento (DUBROVA, 2013).

2.4.2.1 Redundância de Hardware

A redundância de *hardware* é obtida pelo uso de duas ou mais cópias físicas de um componente. Existem três técnicas de redundância de *hardware*: passiva, ativa e híbrida:

- Técnica passiva: mascara eventos sem necessitar que o sistema ou um operador atuem;
- Técnica ativa: a falha é detectada e só então tratada. Neste cenário curtas interrupções devido à erros são toleráveis, com a vantagem de minimizar a redundância necessária (equipamentos, custos) para tornar o sistema tolerante a falhas;
- Redundância híbrida: a técnica passiva e ativa são combinadas, permitindo a reconfiguração do sistema sem que ocorra uma interrupção do serviço.

2.4.2.2 Redundância de Software

A redundância de software pode ser obtida com a aplicação de técnicas de única ou múltiplas versões (DUBROVA, 2013). A primeira opção emprega mecanismos de detecção de falha, contenção e recuperação; já as técnicas de múltipla versão também incluem diversidade de versionamento e design. A seguir estas técnicas são descritas brevemente.

Técnicas de detecção de falha

As técnicas de detecção tem como objetivo verificar a ocorrência de falhas no sistema:

- Verificação de tempo: a performance do sistema é monitorada, verifica-se a ocorrência de *deadlocks*;
- Verificação de código: utilizado em dados que podem ser codificados, verifica-se a corrupção ou perda de dados (ex: verificação cíclica de redundância (CRC));
- Reversão: existindo a possibilidade de realizar o cálculo reverso para se obter a entrada a partir da saída, compara-se a entrada calculada com a entrada utilizada;
- Verificação estrutural: são comparadas as propriedades de estruturas com valores esperados ou previamente conhecidos (ex: tamanho de uma lista);
- Teste de razoabilidade: verifica-se se os valores de uma variável são coerentes com limites previamente estabelecidos.

Técnicas de contenção

Técnicas de contenção tem como objetivo evitar a propagação de uma falha pelo sistema:

- Divisão por módulos: o sistema é estruturado em submódulos, limitando sua comunicação e minimizando os recursos compartilhados;

- Mínima autorização: cada componente só recebe a mínima autorização / acesso necessário ao desempenho de sua função;
- Ações atômicas: as ações, durante a sua execução, são isoladas de outras funções paralelas.

Técnicas de recuperação

Técnicas de recuperação são procedimentos empregados pelo sistema após a ocorrência, detecção e contenção de falhas:

- *Checkpoint* e *Restart*: o módulo é reiniciado a um estado anterior. Solução utilizada para falhas do tipo intermitentes, com curta duração;
- Execução em pares: o programa é executado em dois processadores distintos, um principal e um reserva, que assume em caso de falha do primeiro.

Técnicas de múltiplo versionamento

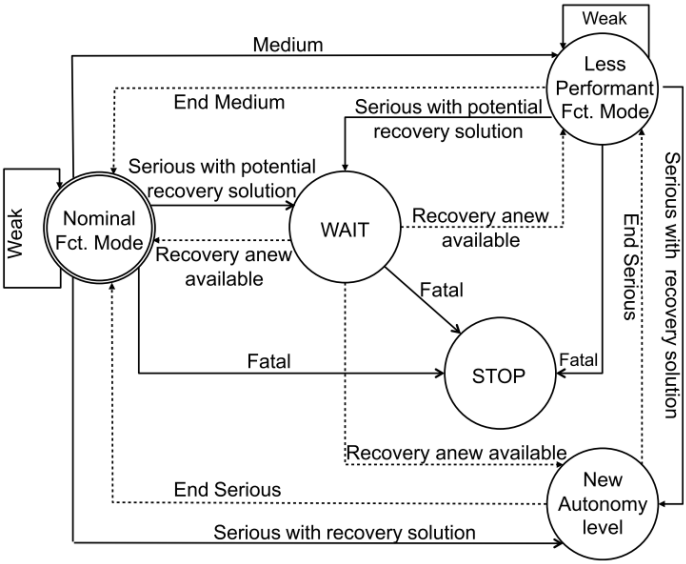
Técnicas de múltiplo versionamento utilizam diferentes times, algoritmos ou códigos de linguagem para minimizar a probabilidade de diferentes versões apresentarem o mesmo tipo de erro:

- Recuperação em blocos: quando detectada uma falha, uma outra versão do módulo (de funcionalidade equivalente) é executada;
- Programação em N-Versões: diversos módulos de mesma funcionalidade e diferente versões executam em paralelo, com um algoritmo selecionando quais serão utilizados para formar a saída;
- Programação auto verificadora: cada módulo possui um teste individual ou é utilizada a comparação em pares para verificar a execução correta dos blocos. O sistema é considerado operacional contanto que ao menos um dos módulos esteja funcionando corretamente.

2.4.3 Exemplos de Técnicas de Detecção e Tolerância a Falhas em AUVs

Crestani, Godary-Dejean e Lapierre (2015) demonstram a integração de princípios de tolerância a falhas no design da arquitetura de controle em tempo real de um robô. A Figura 26 e a Figura 27 apresentam o processo de recuperação de um robô autônomo e a arquitetura tolerante a falhas COTAMA, respectivamente.

Figura 26 – Processo de recuperação de um robô autônomo.

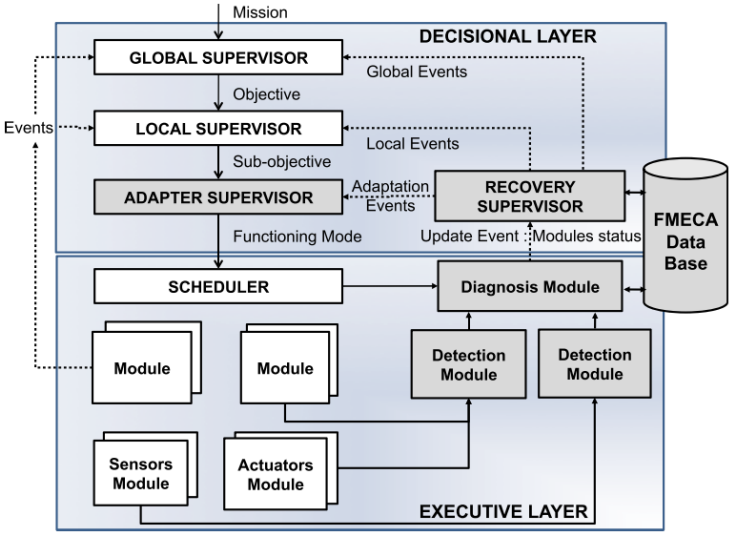


Fonte: Crestani, Godary-Dejean e Lapierre (2015).

Insaurralde e Petillot (2015) propõem o uso de uma arquitetura que permite a cooperação de robôs na execução de tarefas complexas, aproveitando as informações armazenadas para serem utilizadas pelos robôs durante a missão executada. Sua abordagem permite um planejamento da missão adaptável e reconfigurável durante a execução da mesma, sem necessidade de intervenção humana.

Seto e Bashir (2017) apresentam tipos de falhas que devem ser

Figura 27 – Arquitetura COTAMA.

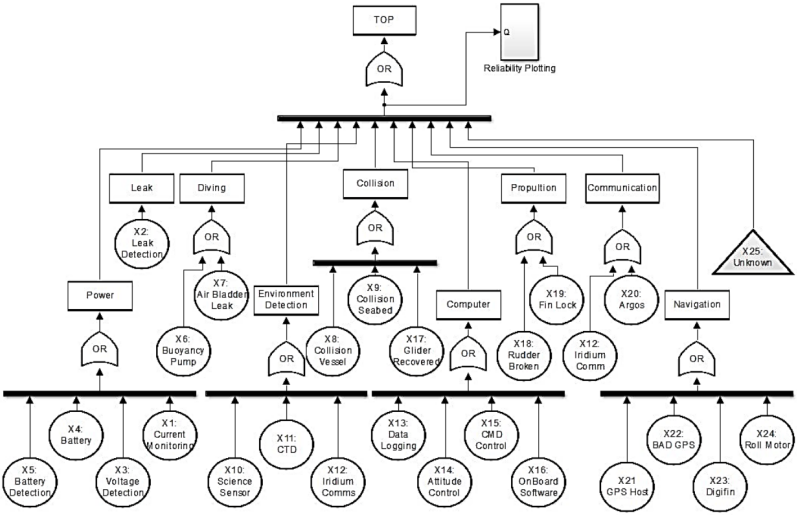


Fonte: Crestani, Godary-Dejean e Lapierre (2015).

consideradas durante o design de um AUV, de modo que o veículo seja capaz de se adaptar a modificações em si mesmo, no ambiente ou na missão. É analisada uma arquitetura aberta para AUVs, utilizando redes bayesianas e software de detecção, isolação e recuperação.

Aslansefat, Latif-Shabgahi e Kamarlouei (2014) apresentam técnicas tolerantes a falhas para o design de AUVs do tipo *glider*, analisando os subsistemas destes veículos e propondo uma árvore de detecção de falhas (Figura 28).

Figura 28 – Árvore de falhas para AUVs do tipo *glider*.



Fonte: Aslansefat, Latif-Shabgahi e Kamarlouei (2014).

3 METODOLOGIA

A metodologia de projeto empregada no desenvolvimento deste trabalho foi baseada na proposta de Back et al. (2008), na qual, após levantadas todas as características e especificações do produto, são geradas soluções que atendam a estes requisitos. As etapas da metodologia são apresentadas na Figura 29.

Figura 29 – Etapas da metodologia de projeto integrado de produtos de (BACK et al., 2008).

Fonte: Autoria própria.

De maneira resumida, as atividades de cada etapa envolvem:

- Planejamento do projeto: identificação das partes envolvidas no projeto e elaboração do plano de gerenciamento das comunicações;
- Projeto informacional: definição das especificações de projeto e levantamento de requisitos;
- Projeto conceitual: desenvolvimento da concepção do produto. O objetivo é definir a estrutura funcional do produto;
- Projeto preliminar: estabelecer o leiaute final do produto. Determinar as principais dimensões dos componentes, realizar testes com *mock-up*, avaliar os leiautes dimensionais sob o ponto de vista da viabilidade técnica do projeto e elaborar uma estrutura preliminar do protótipo;
- Projeto detalhado: aprovação do protótipo e finalização das especificações dos componentes;

Para esta dissertação, foram aplicadas as primeiras cinco etapas da metodologia de Back et al. (2008), pois o objetivo final é apresentar, em bancada experimental funcional, uma arquitetura tolerante a falhas para AUVs, não sendo necessárias as etapas de produção, lançamento e validação do produto no mercado. As próximas seções apresentam o enquadramento do tema da dissertação nas diversas fases da metodologia.

3.1 PLANEJAMENTO DO PROJETO

Nesta etapa foi analisado o problema a ser trabalhado e as partes envolvidas, investigando a solução desejada, identificação do público alvo e as áreas de conhecimento necessárias à execução do projeto.

As informações referentes ao planejamento do projeto foram apresentadas no Capítulo 1.

3.2 PROJETO INFORMACIONAL

Nesta etapa foram levantados os requisitos técnicos que o sistema deve contemplar.

Foram estudados projetos envolvendo AUVs, arquiteturas, sistemas críticos, tolerância a falhas e resiliência. Em paralelo com a análise do estado da arte, foi analisada a experiência, uso e dificuldades encontrados pelos pesquisadores e estudantes da UFSC na área de veículos submarinos.

As informações referentes ao projeto informacional desta dissertação foram apresentadas no Capítulo 2, na forma de uma revisão bibliográfica.

3.3 PROJETO CONCEITUAL

No projeto conceitual é definida a estrutura funcional do sistema, avaliadas as possibilidades de concepção, soluções alternativas e seleção da mais adequada.

Dentre as limitações consideradas na escolha da solução, as principais foram: custo, peso, volume, consumo de energia e complexidade (tempo de desenvolvimento, verificação e teste).

As soluções foram elaboradas utilizando como referência conceitos teóricos abordados na Seção 2.4 e arquiteturas abordadas na Seção 2.3, dentre eles:

- Arquitetura de hardware centralizada / distribuída;
- Arquitetura de hierárquica / não hierárquica;
- Arquitetura de software;
- Técnicas de tolerância a falhas, redundância.

Analisando as causas de falhas mais comuns em AUVs (ver Seção 2.4.1) e o uso previsto em ambientes acadêmicos de pesquisa, é desejado que as soluções sejam capazes de monitorar o sistema de energia, vazamentos, sistema de mergulho e computadores.

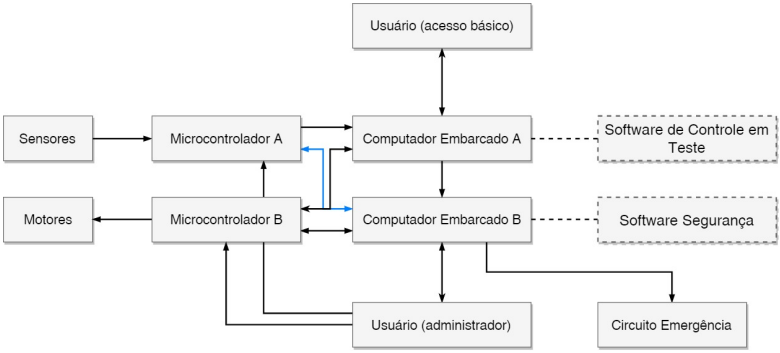
A seguir serão listados estudos de propostas e possibilidades de solução ao projeto. Por fim a solução mais promissora foi selecionada e utilizada nas etapas posteriores da metodologia de projeto.

3.3.1 Concepções de projeto

- **Concepção 1:**

  A Figura 30 apresenta uma proposta de arquitetura distribuída, hierárquica, modular e de múltiplo versionamento. Aqui um módulo é responsável por realizar a leitura e tratamento da entrada dos sensores, e outro pelos motores. Para controle em alto nível existem dois módulos compostos por computadores embarcados e de diferentes versões: um possui acesso básico e outro prioritário. Nesta proposta os módulos dos atuadores e sensores e um dos de inteligência seriam programados por um usuário com privilégios. Um circuito de emergência independente está presente, acionado em casos críticos.

Figura 30 – Proposta de arquitetura distribuída, hierárquica, modular e de múltiplo versionamento.



Fonte: Autoria própria.

- **Concepção 2:**

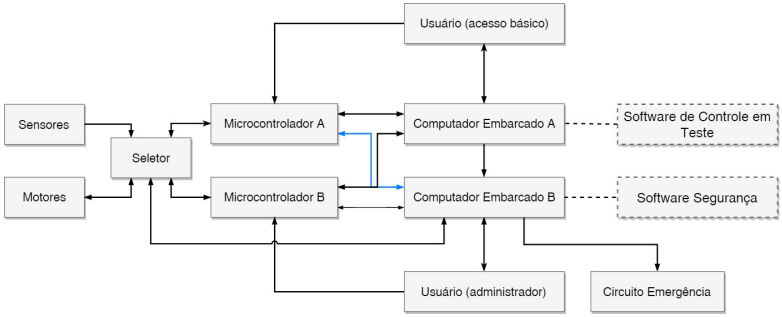
  A Figura 31 apresenta uma proposta de arquitetura distribuída, hierárquica, modular e de múltiplo versionamento. A leitura de sensores e acionamento dos motores é realizada por um dos módulos compostos por microcontroladores, ambos redundantes funcionalmente, mas programados por dois usuários ou equipes distintas, sendo que o programado pelo administrador possui maior prioridade. Um módulo seletor é responsável por escolher qual entrada será utilizada. Para controle em alto nível existem dois módulos compostos por computadores embarcados e de diferentes versões: um possui acesso básico e outro prioritário. Um circuito de emergência independente está presente, acionado em casos críticos.

- **Concepção 3:**

  A Figura 32 apresenta uma proposta de arquitetura centralizada. A leitura de sensores, acionamento dos motores e inteligência de controle é realizada por um único módulo central. Um circuito de emergência independente está presente, acionado em casos críticos.

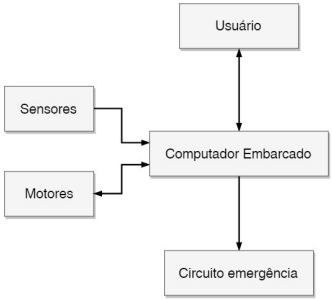
- **Concepção 4:**

Figura 31 – Proposta de arquitetura distribuída, hierárquica, modular e de múltiplo versionamento.



Fonte: Autoria própria.

Figura 32 – Proposta de arquitetura centralizada.



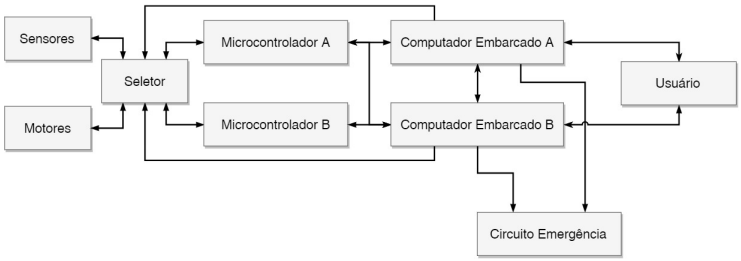
Fonte: Autoria própria.

A Figura 33 apresenta uma proposta de arquitetura distribuída, não hierárquica, modular e com redundância. Existem módulos programados pelo mesmo usuário, um principal e outro reserva, que pode assumir em caso de falha do primeiro. Um circuito de emergência independente está presente, acionado em casos críticos.

- **Concepção 5:**

  A Figura 34 apresenta uma proposta de arquitetura distribuída, hierárquica e com redundância. Existem módulos programados

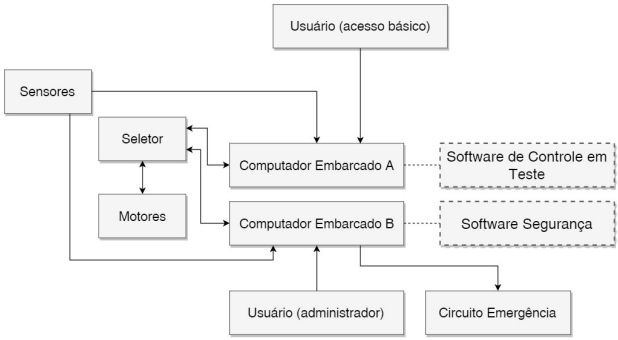
Figura 33 – Proposta de arquitetura distribuída, não hierárquica, modular e com redundância.



Fonte: Autoria própria.

por usuários distintos, um possuindo acesso básico e outro prioritário, assumindo em caso de falha do primeiro. Estes módulos são parcialmente centralizados, uma vez que são responsáveis pela leitura dos sensores, acionamento dos atuadores e inteligência. Um circuito de emergência independente está presente, acionado em casos críticos.

Figura 34 – Proposta de arquitetura distribuída, hierárquica e com redundância.



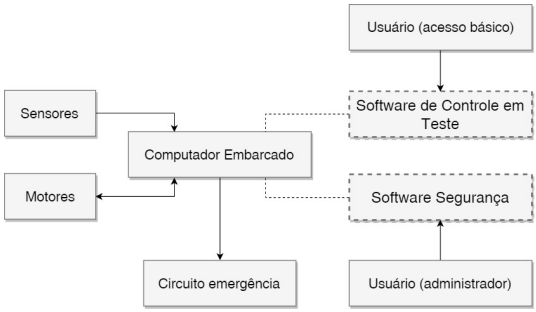
Fonte: Autoria própria.

- **Concepção 6:**

  A Figura 35 apresenta uma arquitetura de hardware centralizada,

mas com duas camadas de software, uma de teste e outra de segurança, que roda em segundo plano e assume em caso de falha do primeiro. Um circuito de emergência independente está presente, acionado em casos críticos.

Figura 35 – Proposta de arquitetura centralizada.

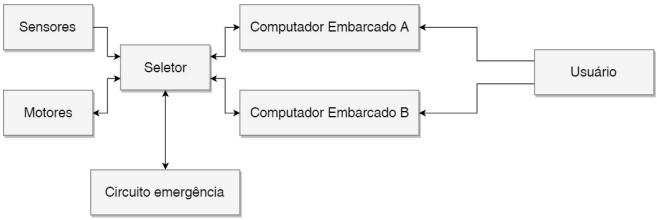


Fonte: Autoria própria.

- **Concepção 7:**

  A Figura 36 apresenta uma arquitetura distribuída, com redundância, composta por dois módulos, um principal e outro reserva, responsáveis pela leitura dos sensores, controle e atuação dos motores. Um circuito de emergência independente está presente, acionado em casos críticos.

Figura 36 – Proposta de arquitetura distribuída e com redundância.



Fonte: Autoria própria.

Tabela 1 – Propostas de solução de projeto.

Concepção	Arquitetura de Hardware	Arquitetura de Software	Comentários
1	Distribuída, modular, Hierárquico, suporta técnica passiva, ativa e híbrida.	Hierárquico, suporta técnicas de detecção variadas, contenção, recuperação e múltiplo versionamento.	O usuário básico tem acesso apenas à um módulo de inteligência. O usuário administrador tem acesso ao módulo de inteligência reserva e submódulos específicos. Flexibilidade de desenvolvimento, manutenção e implementação de técnicas de detecção e tolerância a falhas.
2	Distribuída, modular, Hierárquico, suporta técnica passiva, ativa e híbrida.	Hierárquico, suporta técnicas de detecção variadas, contenção, recuperação e múltiplo versionamento.	O usuário básico e administrador tem ambos acesso à módulos de inteligência e submódulos específicos. Flexibilidade de desenvolvimento, manutenção e implementação de técnicas de detecção e tolerância a falhas.
3	Centralizada, suporta técnicas passiva, ativa e híbrida.	Não hierárquico, suporta técnicas de detecção variadas.	Uma única equipe de desenvolvimento é responsável pelo controle do AUV. Arquitetura centralizada apresenta baixa flexibilidade no desenvolvimento, manutenção e implementação de técnicas de detecção e tolerância a falhas. Baixo custo de implementação.
4	Distribuída, modular, não hierárquica, suporta técnica passiva, ativa e híbrida.	Não hierárquico, suporta técnicas de detecção variadas, contenção e recuperação.	Uma única equipe de desenvolvimento é responsável pelo controle do AUV. A arquitetura apresenta flexibilidade no desenvolvimento, manutenção e implementação de técnicas de detecção e tolerância a falhas.
5	Distribuída, parcialmente modular, não hierárquica, suporta técnica passiva, ativa e híbrida.	Hierárquico, suporta técnicas de detecção variadas, contenção parcial, recuperação e múltiplo versionamento.	Possui um módulo para uma equipe com acesso básico e outro módulo para equipe com acesso prioritário. Não possui submódulos específicos.
6	Centralizada, suporta técnicas passiva, ativa e híbrida.	Hierárquico, suporta técnicas de detecção variadas, recuperação e múltiplo versionamento.	Arquitetura centralizada apresenta baixa flexibilidade no desenvolvimento de hardware, manutenção e implementação de técnicas de detecção e tolerância a falhas. Flexibilidade para suportar duas equipes desenvolvedoras a nível de software. Baixo custo de implementação.
7	Distribuída, parcialmente modular, suporta técnica passiva, ativa e híbrida.	Não hierárquico, suporta técnicas de detecção variadas, contenção parcial e recuperação.	Uma única equipe de desenvolvimento é responsável pelo controle do AUV utilizando dois módulos, um principal e outro reserva.

Fonte: Autoria Própria.

3.3.2 Proposta de solução

A Tabela 1 apresenta uma lista comparativa das concepções geradas.

Buscando a opção que melhor atendesse os objetivos, as soluções foram comparadas e analisadas. As concepções centralizadas (Concepções 3 e 6) apresentam custos reduzidos em relação às outras concepções, porém possuem menor flexibilidade no desenvolvimento, manutenção e robustez. Se o módulo central falhar o AUV dependerá do sistema de emergência adicional para se recuperar. A concepção 6 implementa a técnica de múltiplo versionamento a nível de software, mas é mais

vulnerável à falhas de hardware que as outras concepções modulares.

As concepções parcialmente distribuídas, mas sem a subdivisão de módulos específicos (Concepções 5 e 7), apesar de apresentarem menor complexidade e custo, ainda possuem flexibilidade de experimentação, desenvolvimento e tolerância a falhas inferior às outras propostas. A estratégia utilizada pela Solução 4 é interessante do ponto de vista de hardware, mas oferece menos possibilidades de tolerância a falhas a nível de software e controle quando comparada com as arquiteturas apresentadas nas concepções 1 e 2, que apresentam múltiplo versionamento de software.

Na concepção 1, o usuário possui acesso mínimo à um computador embarcado responsável por executar algoritmos experimentais de controle do AUV, enquanto que o computador embarcado reserva e submódulos específicos são de responsabilidade de um usuário administrador.

Por fim, a concepção 2 foi selecionada por permitir, tanto ao usuário básico quanto administrador, acesso aos submódulos específicos, oferecendo mais flexibilidade e sem agravar os riscos apresentados ao AUV, por sempre existir a possibilidade da rotina programada pelo usuário administrador assumir o controle do AUV. A escolha apresenta vantagens vinculadas à flexibilidade que a divisão por módulos traz na implementação e detecção de falhas, possibilidades de redundância e hierarquia para segurança e tolerância a falhas.

Nesta etapa foram propostas diferentes configurações e estratégias baseadas na revisão apresentada no Capítulo 2, de modo a se obter uma arquitetura cada mais robusta e confiável. O processo de refinamento e detalhamento será descrito nas seções que se seguem.

3.4 PROJETO PRELIMINAR

No projeto preliminar as possibilidades de solução capazes de atender o projeto conceitual foram analisadas.

A estratégia básica da arquitetura selecionada é dividir o controle do robô em dois módulos distintos, um experimental e outro reserva, que

é acionado em caso de pane do primeiro. Estes módulos principais são responsáveis pela inteligência do robô (ex.: planejamento da missão e trajetória). Para setores mais específicos do robô, são implementados sub-módulos responsáveis pelos sensores, emergência e atuadores (podendo ser adicionados módulos e feitas adaptações conforme a necessidade). Em último caso, em caso de falha do módulo reserva, um módulo de emergência seria acionado. A uma visão geral da estratégia adotada é apresentada na Figura 37.

Figura 37 – Visão geral da estratégia da arquitetura de controle do AUV proposta.

Fonte: Autoria própria.

Buscando experimentar ainda mais com as possibilidades de redundância, um módulo poderia assumir as responsabilidades de outro módulo, em caso de falhas. A desvantagem desta manobra é a divisão do processamento do primeiro módulo e aumento da complexidade de software e hardware do módulo.

A estratégia de divisão em módulos é interessante por incentivar a organização e divisão das tarefas e inteligência pelos programadores do veículo, além de facilitar a identificação e tratamento de erros. Maiores detalhes podem ser encontrados na Seção 2.3.

Para o controle do AUV foi previsto o uso de microcontroladores para os módulos com necessidade computacional reduzida e computadores embarcados para os que exigem maior capacidade de processamento. Algumas das opções de componentes de arquitetura aberta, populares e de baixo custo, que poderiam ser empregados no projeto são:

- Arduino MEGA 2560[1]: microcontrolador baseado no ATmega2560[2],

[1] <https://www.arduino.cc/>
[2] <http://www.atmel.com/>

possui 54 portas IO digitais (15 suportam PWM), 16 entradas analógicas, 4 UARTs (comunicação serial), *clock* de 16MHz e alimentação de 7-12V. Memória *flash* de 256kbytes, 8kbytes de SRAM e 4kbytes de EEPROM. Compatível com a maioria das placas de extensão do Arduino Uno.

- Raspberry Pi 3[1]: possui um processador 1.2GHz 64-bit *quad-core* ARMv8, 40 portas IO digitais, uma UART, comunicação via *Wireless*, HDMI, *Bluetooth* (4.1 e BLE) e USB (4 portas), memória RAM de 1GB e entrada de cartão *Micro SD*.

- BeagleBone Black[2]: processador ARM Cortex-A8 com 1GHZ de velocidade, 512MB de memória SDRAM, 4GB de memória *flash*, 4 UARTs, alimentação de 5V (1.2-2A), 65 portas IO digitais (8 suportam PWM) e 7 analógicas, comunicação via *Ethernet*, USB e HDMI. Suporta programação para sistemas de tempo real.

Para o protótipo de bancada foi previsto inicialmente a utilização dos sensores já adquiridos pelo projeto "Desenvolvimento de Tecnologias emergentes para exploração de petróleo no mar - CAPES 2013" e aqueles utilizados pelos robôs, dentre eles o IMU, sensor de pressão, sensor de vazamento e sensor de temperatura. Para os atuadores serão utilizados os propulsores T200 do *BlueROV2*.

Para ilustrar o acionamento de um sistema de emergência foi pensado em um circuito independente, com bateria reserva, capaz de acionar um *LED* de alta potência e baixo consumo, com acionamento intermitente. Para modificações futuras seria interessante que este circuito acionasse um módulo mecânico que fosse capaz de modificar o empuxo do veículo, de modo a fazê-lo flutuar (liberando pesos, por exemplo). Em um protótipo de bancada isto poderia ser reproduzido utilizando uma chave eletrônica (relé) que acione um mecanismo que facilite a visualização da funcionalidade.

[1] <https://www.raspberrypi.org/>
[2] <https://beagleboard.org/black>

Algumas possibilidades de modificação da arquitetura selecionada foram analisadas para a construção do protótipo, como o módulo dos sensores / atuadores comportando mais de uma rotina cada um, uma experimental e outra reserva (não sendo necessário reservar um dispositivo para cada modo de operação ou um dispositivo seletor de entrada ou saída adicional, operado pelo computador embarcado reserva). A rotina a ser executada pelos submódulos dependeria da rotina de controle do AUV em execução, experimental ou reserva.

3.5 PROJETO DETALHADO

No projeto detalhado foi finalizada a especificação de componentes e o protótipo foi construído. Os detalhes da construção e testes do protótipo são apresentados no Capítulo 4 e a análise de resultados no Capítulo 5.

4 PROTÓTIPO DE BANCADA

O desenvolvimento do protótipo foi dividido e organizado nas seguintes etapas:

- Análise, compra e teste individual de componentes;
- Implementação dos módulos e testes individuais;
- Comunicação entre módulos;
- Implementação e testes de tolerância a falhas;
- Integração dos módulos e preparação de bancada para apresentação.

4.1 ANÁLISE, COMPRA E TESTE INDIVIDUAL DE COMPONENTES

Para o módulo de inteligência e controle foram adquiridas duas placas BeagleBone Black, escolhidas pelo seu maior número de portas IO em relação ao Raspberry Pi. Foram adquiridos dois Arduino Mega 2560 para o controle dos módulos responsáveis pelos sensores e atuadores. Estes microcontroladores possuem uma capacidade de processamento e pinos mais do que suficientes para o protótipo de bancada, mas, devido à pouca diferença de preço, e por aumentar a flexibilidade de experimentação no protótipo, foram escolhidos estes ao invés de se adquirir um modelo mais simples, como um Arduino Due. Para o controle do módulo de emergência foi adquirido um Arduino Nano.

Em adição às placas de controle foram necessários componentes eletrônicos para implementação da arquitetura proposta, com a subsequente aquisição e testes individuais de funcionalidade dos mesmos. Para as entradas e saídas digitais e analógicas foram utilizados resistores de *pull-up* e *pull-down*, impedindo que o nível de tensão das portas flutuasse. Também foram utilizados transistores e diodos para implementação do chaveamento e isolamento dos módulos. Para possibilitar a

comunicação entre diferentes níveis de tensão, foram adquiridos conversores de nível lógico de tensão bidirecionais (3,3 - 5,5 V). No módulo de emergência foram utilizados um LED e uma sirene (simulando um *ping* de sonar), acionados em caso de falha do módulo de controle reserva. Para representar o estado de cada módulo e simulação de falhas, foram instalados LEDs variados e botões eletrônicos. Por fim, para teste do módulo responsável pela leitura dos sensores foi utilizado um acelerômetro MMA1270D 2.5g e, para o módulo responsável pelo acionamento dos atuadores, foi utilizado um propulsor T200 em conjunto com controlador de velocidade eletrônico ESC 30A, acionado via PWM.

4.2 IMPLEMENTAÇÃO DOS MÓDULOS E TESTES INDIVIDUAIS

Nesta etapa cada módulo da arquitetura foi construído e testado separadamente:

- Módulo Sensores - Arduino e Sensor Inercial;
- Módulo Atuadores - Arduino e Propulsor T200;
- Módulo de Inteligência Experimental - BeagleBone Black;
- Módulo de Inteligência Reserva - BeagleBone Black;
- Módulo de Emergência - Arduino Nano.

Os módulos dos sensores e atuadores foram programados utilizando uma interface de desenvolvimento aberta Arduino, sendo testados para leitura dos valores dos sensores, acionamento do propulsor T200, acionamento de LEDs, sirene e leitura de entradas digitais (como a do botão para a simulação de falha no módulo).

O módulo dos sensores são conectados ao sensor acelerômetro por meio de três pinos analógicos. A leitura é então tratada, com o cálculo variando de acordo com as especificação do sensor. No caso do protótipo de bancada, o cálculo de conversão para acelerômetro MMA1270D é dado por:

$$ValorFinal = ValorRecebido * 2,5/1024,0$$

sendo $ValorRecebido$ a leitura direta obtida pela leitura do sensor, 2,5 a sensibilidade do sensor e 1024 a resolução de leitura da entrada analógica.

O propulsor T200 utilizado pelo protótipo de bancada é o mesmo tipo de propulsor utilizado pelo BlueROV2. O T200 utiliza um motor CC sem escovas, necessitando de um controlador de velocidade para ser operado. Para o protótipo foi utilizado um controlador genérico ESC 30A, que recebe um PWM de 1200 à 1450 us, acionando o motor de 0 à 100% da velocidade máxima.

A sirene utilizada pelo módulo de emergência foi acionada com o auxílio da função *noTone*, disponibilizada pelo Arduino, que pode ser configurada com a frequência e duração do som a ser emitido pela sirene. Para testes de campo seria interessante substituir esta sirene por um sonar.

Os BeagleBones foram testados com rotinas programadas utilizando a linguagem de programação *Python*, configurados para executá-las durante a inicialização do módulo. É possível estabelecer comunicação entre o computador do usuário e o BeagleBone via acesso remoto utilizando os softwares *TightVNC*[1] e *PuTTY*[2]. Também é possível utilizar um software como o *WinSCP*[3] para facilitar a transferência de dados entre o computador do desenvolvedor e o computador embarcado. A configuração detalhada dos sistemas é descrita no Apêndice A.

Para os testes de bancada foi utilizada conexão cabeada, mas é possível estabelecer uma comunicação sem fio com a instalação e configuração de uma placa *wireless* adicional. Para configurar o BeagleBone para executar, durante a sua inicialização, os códigos transferidos, foi utilizada a ferramenta de gerenciamento de tarefas *Cron*[4]. Maiores detalhes em relação à programação das placas BeagleBones podem ser encontrados no Apêndice A.

Os botões foram configurados de modo a permanecerem com

[1] https://www.tightvnc.com/
[2] https://www.putty.org/
[3] https://winscp.net/
[4] https://crontab.guru/

o nível lógico baixo para quando não estão acionados e alto quando são pressionados. Tanto nas rotinas dos BeagleBones quanto Arduinos foram aplicadas na lógica de leitura do botão a filtragem de ruídos causados pelo *bounce* da chave.

Para a simulação de falha nos módulos, após detectar que o respectivo botão foi pressionado, a rotina força a finalização das atividades executadas pelo módulo, incluindo a comunicação do mesmo com outros módulos, e força a entrada em um *deadlock*.

4.3 COMUNICAÇÃO ENTRE MÓDULOS

Algumas das opções consideradas para realizar a comunicação entre os módulos de controle foram a serial, SPI e I2C. A opção serial possui limitações relacionadas à baixa velocidade e limite de número de dispositivos, quando comparada às outras opções. O protocolo SPI exige uma quantidade de pinos muito grande, principalmente para situações com múltiplos dispositivos (utilizando três pinos comuns à todos os componentes da rede e um adicional para cada dispositivo além do mestre), apesar de ser *full-duplex* e suportar *clocks* de até 10MHz. O protocolo I2C foi a opção escolhida para este projeto, sendo balanceado em relação a desempenho, velocidade e utilizando duas linhas para comunicação (ocupando dois pinos de cada dispositivo na rede).

O protocolo I2C entre múltiplos componentes envolve uma unidade mestre e um ou vários dispositivos escravos. A unidade mestre pode enviar e solicitar dados aos escravos, mas um escravo não envia dados diretamente ao mestre ou à outro escravo. Uma organização I2C pode suportar até 112 escravos (para endereços de 7 bits, sendo alguns reservados). Este protocolo possui um alcance de aproximadamente 2-3 metros para uma velocidade de comunicação de 100 kb/s, variando de acordo com o cabo utilizado (podendo aumentar se for reduzida a frequência de comunicação e sendo tratado o ruído agravado pelo longo cabeamento).

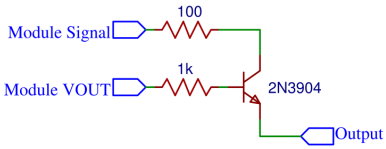
Para implementação da comunicação entre os módulos, inicialmente foram testados a comunicação em duplas (Arduino - Arduino;

Arduino - Beaglebone; Beaglebone - Beaglebone), e depois integrados todos os módulos. O protocolo I2C utiliza dois pinos para realizar a comunicação entre cada placa e um circuito terra comum, sendo utilizado um conversão de nível de tensão entre placas que apresentam níveis lógicos distintos (Arduino utiliza 5V, enquanto que o BeagleBone Black opera em 3.3V).

Nos testes e implementação da comunicação, os Arduinos foram configurados como escravos e os BeagleBones como mestres. Os Arduinos foram configurados utilizando a biblioteca *Wire*[1] e os BeagleBones foram configurados para comunicar utilizando a biblioteca SMBus[2], uma derivação do I2C.

Durante os testes de simulação de falhas dos módulos com perda de energia, foi verificado que a comunicação I2C do sistema como um todo ficava prejudicada. Para isolar os módulos de modo a impedir que falhas individuais comprometessem o todo, foram utilizados resistores de *pull-up* nas linhas conectadas aos Arduinos e no módulo experimental foram utilizados transistores, configurados para cortar a conexão do módulo na ocorrência de perda de energia deste (Figura 38).

Figura 38 – Esquemático de circuito de chave utilizando o transistor NPN 2n3904.



Fonte: Autoria própria.

Como camada de segurança adicional, para os testes de bancada os módulos responsáveis pelos sensores e atuadores foram configurados com diferentes endereços na rede I2C, variando de acordo com a rotina em execução: experimental ou reserva. Para ilustrar esta estratégia,

[1] <https://www.arduino.cc/en/Reference/Wire>
[2] <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/plain/Documentation/i2c/smbus-protocol>

quando o módulo reserva assume o controle do AUV, por exemplo, os módulos do sensor e atuador só responderão às mensagens enviadas ao endereço configurado para o estado de operação com o módulo reserva, com as mensagens dos dispositivos no estado experimental sendo ignoradas.

Além da comunicação I2C, o protótipo utiliza canais adicionais para emissão / leitura de sinais digitais para verificação de estado de funcionamento do módulo reserva, experimental, *reset* dos Arduinos e indicação de qual módulo de controle está atuando no AUV.

4.4 IMPLEMENTAÇÃO E TESTES DE TOLERÂNCIA A FALHAS

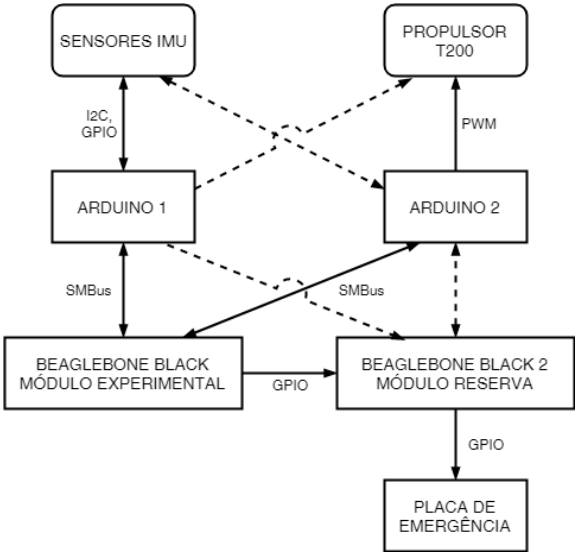
A visão geral da arquitetura em nível de *Hardware*, apresentada na Figura 39 ilustra a configuração dos módulos e comunicação. As linhas de comunicação tracejadas somente são acionadas na ocorrência de falhas (Por exemplo: quando o módulo de inteligência reserva deve assumir as funções do módulo experimental). A placa de emergência monitora o módulo de inteligência reserva e é acionado em caso de falhas deste.

No nível de software, os módulos seguem a seguinte estrutura:

- Módulo leitura sensores / acionamento atuadores: Possuem duas rotinas principais, uma acionada enquanto o módulo experimental de controle está ativo, e outra acionada se o módulo reserva assume o controle. Quando estes módulos inicializam eles verificam qual módulo de controle está em execução e inicializam a rotina correspondente. Em caso de falhas destes módulos, eles pode ser reinicializados remotamente pelo módulo de controle. Podem ser programados com sub-rotinas adicionais, a exemplo de rotinas executadas em caso de falha de outros módulos, podendo assumir mais responsabilidades, em troca de desempenho;

- Módulo experimental: Este módulo executa uma rotina de controle experimental a ser testada pelo AUV;

Figura 39 – Visão geral proposta arquitetura de *Hardware*.



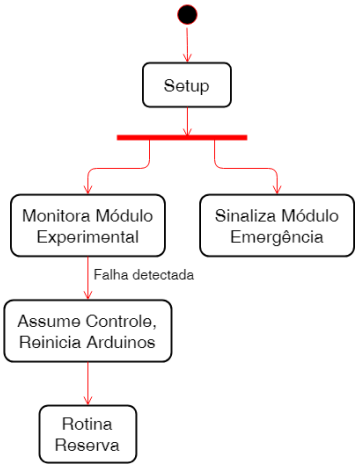
Fonte: Autoria própria.

- Módulo reserva: Este módulo monitora o módulo experimental, assumindo o controle do AUV ao detectar falhas. Deve ser capaz de reiniciar o módulo sensorial e o de operação dos propulsores. Em caso de falha de algum módulo, pode executar rotinas de emergência, reorganizando as funcionalidades dos módulos.

- Módulo de emergência: Monitora o módulo de controle reserva, acionando dispositivos de emergência em caso de detecção de falhas (no protótipo de bancada são acionados um LED e um sinal sonoro).

Em caso de falhas durante a execução da rotina experimental, o módulo de controle reserva assume o controle do AUV e reinicializa os Arduinos remotamente (pelo acionamento do pino de *reset*). Ao inicializar, os Arduinos verificam qual rotina devem executar (com base na leitura da entrada digital conectada ao módulo reserva), neste caso

Figura 40 – Rotina módulo reserva.



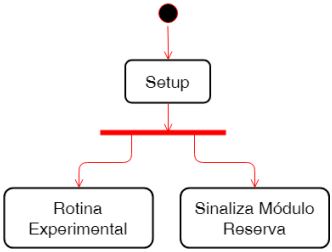
Fonte: Autoria própria.

executando as rotinas e lógicas reservas. A rotina simplificada do módulo reserva é ilustrada na Figura 40

O módulo reserva de controle verifica o estado de funcionamento dos Arduinos enviando mensagens do tipo *ping* regularmente via I2C. Já o módulo experimental (Figura 41) possui duas saídas digitais conectadas e monitoradas pelo reserva. Estas saídas são mantidas em nível lógico de tensão alto, emitindo pulsos em baixa periodicamente. Se um dos sinais cair ou ocorrer algum erro com a leitura dos pulsos (que podem, por exemplo, ser causados por um desligamento da placa experimental ou falha de execução do software experimental, respectivamente), o módulo reserva assume o controle do AUV, reiniciando e reconfigurando os módulos de leitura de sensores e acionamento de atuadores para executarem as rotinas reserva.

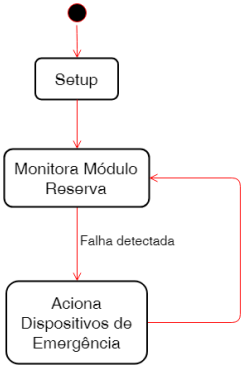
O módulo de emergência monitora o módulo de controle reserva (Figura 42) utilizando a mesma lógica que o reserva monitora o módulo experimental, lendo e analisando a entrada de dois pinos digitais, conectados ao módulo reserva. No protótipo de bancada este módulo aciona

Figura 41 – Rotina módulo experimental.



Fonte: Autoria própria.

Figura 42 – Rotina módulo emergência.

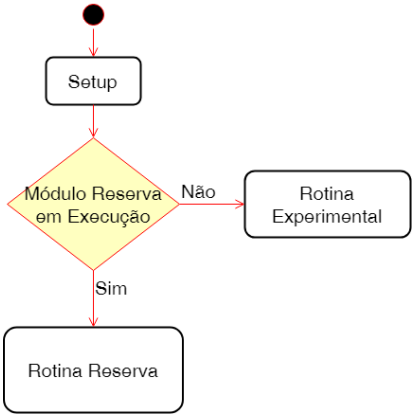


Fonte: Autoria própria.

um LED e uma Sirene quando detecta falha no módulo reserva.

Para implementar redundância entre módulos e hierarquia, os Arduinos foram programados com um código modular. Isto foi feito com a criação de uma rotina de backup e outra experimental, acionadas de acordo com a situação e com base no módulo de controle em execução (experimental ou reserva), representado na Figura 43 . O Arduino possui um pino de *reset* que pode ser acionado caso o seu módulo de software experimental apresente problemas, reiniciando e executando o código de reserva. Opcionalmente é possível adicionar sub-rotinas para casos especiais de funcionamento, como situações de falha de dispositivos (No

Figura 43 – Rotina módulo sensor/atuador.



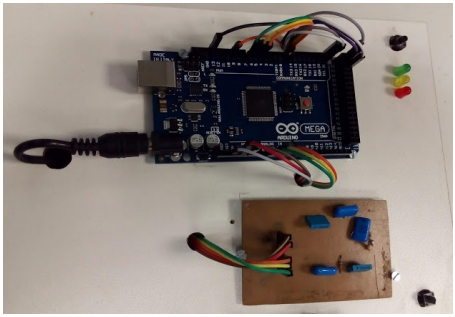
Fonte: Autoria própria.

protótipo de bancada esta funcionalidade foi testada dentro do modo de operação reserva do AUV, no qual o módulo responsável pelos sensores foi configurado de modo a ser capaz de acionar os atuadores, assim como o módulo de atuadores foi adaptado para conseguir ler os sensores, caso necessário).

A indicação de qual módulo de controle está no comando do AUV é dado pelo módulo reserva. Além de ser conectado aos pinos de *reset* dos Arduinos ele também é conectado à dois pinos digitais, que são colocado em alta quando este assume o controle. Deste modo, os Arduinos verificam estas entradas ao inicializarem, executando a rotina correspondente. Adicionalmente, ainda é possível incluir rotinas de emergência, para que o módulo seja capaz de assumir a funcionalidade de outros módulos, em caso de falhas.

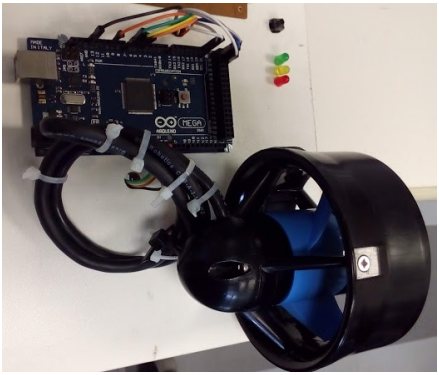
Nos testes do protótipo de bancada os módulos responsáveis pelos sensores / atuadores (Figura 44 e 45) e o módulo experimental foram configurados para inicializar a execução das rotinas de controle do AUV assim que são energizados. O módulo reserva e o de emergência (Ver Figura 46) aguardam um tempo mínimo de inicialização das outras placas, antes de iniciar o monitoramento. Para ilustrar a funcionalidade

Figura 44 – Módulo sensor.



Fonte: Autoria própria.

Figura 45 – Módulo atuador.



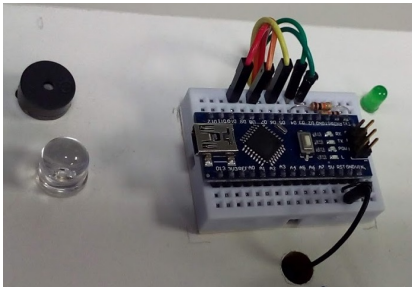
Fonte: Autoria própria.

do protótipo e facilitar a visualização e entendimento da arquitetura testada, foram adicionados ao protótipo de bancada LEDs que representam o estado de cada módulo, e botões para simular falhas nos módulos.

Os LEDs de estado dos módulos indicam os seguintes estados:

- Módulo experimental
  - Verde: Executando rotinas de controle do AUV;
  - Amarelo: Executando rotinas de inicialização;

Figura 46 – Módulo emergência.



Fonte: Autoria própria.

  - Vermelho: Modo simulação de falha.

- Módulo reserva
  - Verde: Assumindo controle e executando rotinas de controle do AUV;
  - Amarelo: Monitorando módulo experimental;
  - Vermelho: Modo simulação de falha.
- Módulo sensor / atuador
  - Verde: Executando rotinas módulo experimental;
  - Amarelo: Executando rotinas módulo reserva;
  - Vermelho: Modo simulação de falha.
- Módulo emergência
  - Verde: Monitorando módulo reserva.

No protótipo de bancada os Arduinos foram conectados ao módulo de controle reserva por três pinos adicionais, além dos dois necessários à comunicação I2C. Dois destes são de mesma função e redundantes, indicando se o módulo de controle reserva assumiu o controle do AUV, e o outro sendo conectado ao pino de *reset*, permitindo que as placas sejam reiniciadas remotamente.

O módulo de controle reserva monitora o módulo experimental verificando dois pinos digitais (redundantes), que enviam pulsos regularmente. Se o pulso deixar de ser enviado, uma falha é detectada e o módulo reserva assume o controle. A mesma lógica é utilizada para o monitoramento do módulo reserva pelo de emergência, que aciona os dispositivos de emergência caso detecte uma anomalia no sinal enviado.

Seguem a relação da alimentação dos componentes utilizados no protótipo:

- Arduino Mega 2560: alimentação de 7-12V, podendo ser suprido via conector *power jack*, pino VIN ou USB;
- Arduino Nano: Alimentação de alimentação de 7-20V no pino 30 ou 5V regulada no pino 27;
- Beaglebone Black: alimentado via conector CC, 5V ou Mini USB. Seu consumo é de 1A sem periféricos, do contrário é necessário suprir uma maior corrente;
- Propulsores T200: 6-22V, corrente de 25 amps (na terra) e máximo de 35 (na água);
- Sensor Acelerômetro MMA1270D: 5V.

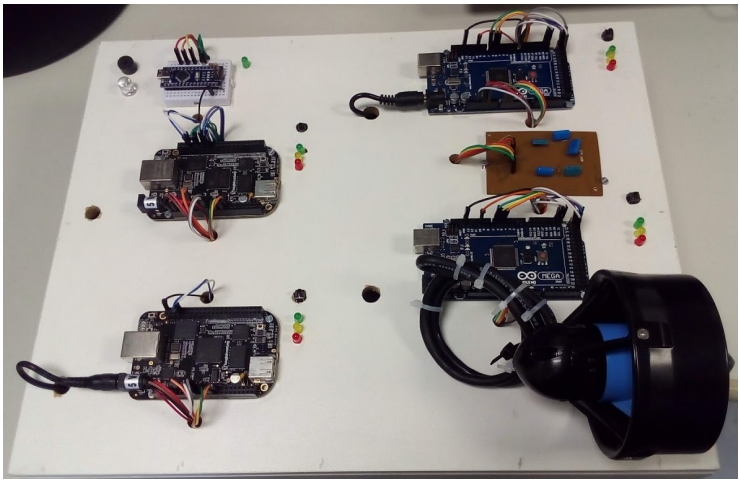
Para os testes de bancada foram utilizados uma fonte de tensão para cada módulo e uma para o propulsor e sensor.

4.5 INTEGRAÇÃO DOS MÓDULOS E PREPARAÇÃO DA BANCADA PARA APRESENTAÇÃO

Nesta etapa os módulos previamente testados individualmente ou em subconjuntos menores, foram conectados, integrados e testados em conjunto. A vista geral do protótipo de bancada é apresentada na Figura 47.

Os códigos e rotinas executadas pelos módulos do protótipo de bancada estão disponíveis em C e os diagramas das conexões e circuitos eletrônicos em B.

Figura 47 – Vista geral do protótipo de bancada.



Fonte: Autoria própria.

5 TESTES DO PROTÓTIPO INTEGRADO

O presente capítulo descreve os os testes e resultados do protótipo em bancada.

Os testes iniciais foram realizados seguindo princípios de *Unit testing* (Abran, A., Moore, J. W., Bourque, P., Dupuis, R., Tripp, 2001), no qual são testados blocos individuais de códigos ou módulos, de forma isolada. Para isto, cada canal e método de leitura ou emissão de sinais dos módulos foram verificados separadamente. Após os testes individuais, foram realizados testes de integração, inicialmente em pares de módulos, até o teste do conjunto integrado completo. Este processo é feito para isolar e auxiliar a identificação da causa de falhas (que no estágio de prototipagem incluem falhas causadas por erros de algoritmo, cabeamento, montagem, especificação ou falha de componente). As etapas de implementação, testes individuais e as de integração parcial são apresentadas na Tabela 2.

Para os testes do conjunto integrado completo, foi analisada a lógica de controle dos módulos, e se o protótipo se comportou de acordo. Resumidamente, a arquitetura proposta possui duas rotinas de controle distintas (que compartilham os recursos de hardware do robô) e uma de emergência. A ordem de operação básica é a execução da rotina experimental e, em caso de falha desta, a rotina reserva assume o controle do AUV. Se o módulo reserva falhar, é executado o módulo de emergência. Variações e diferentes estratégias podem ser experimentadas, algumas exigindo atualizações e modificações nos componentes eletrônicos do AUV.

Os testes funcionais do protótipo integrado foram realizados com a simulação de falhas nos módulos, empregando diferentes combinações e ordem de falha. O estado e funcionamento de cada módulo foram monitorados por meio da análise dos LEDs de estado e também diagnosticando e validando os resultados com o auxílio do computador do desenvolvedor (via comunicação Serial com os Arduinos e SSH com os BeagleBones), para verificação da comunicação, variáveis e execução da rotina de cada módulo. A lista de testes funcionais do protótipo,

Tabela 2 – Etapas de implementação do protótipo.

Etapa		Descrição
Individual	1	Programação Arduino
	1.1.1	Módulo Atuador - Acionamento Propulsor (T200, ESC30A)
	1.1.2	Módulo Atuador - Acionamento LEDs
	1.1.3	Módulo Atuador - Leitura Botão
	1.1.4	Módulo Atuador - Modo Simulação de Falha
	1.2.1	Módulo Sensor - Leitura Sensores (MMA1270D)
	1.2.2	Módulo Sensor - Acionamento LEDs
	1.2.3	Módulo Sensor - Leitura Botão
	1.2.4	Módulo Sensor - Modo Simulação de Falha
	1.3.1	Módulo Emergência - Acionamento LEDs
	1.3.2	Módulo Emergência - Acionamento Sirene
	2	Programação Beaglebone Black
	2.1.1	Módulo Experimental - Acionamento LEDs
	2.1.2	Módulo Experimental - Leitura Botão
	2.1.3	Módulo Experimental - Modo Simulação de Falha
	2.2.1	Módulo Reserva - Acionamento LEDs
	2.2.2	Módulo Reserva - Leitura Botão
	2.2.3	Módulo Reserva - Modo Simulação de Falha
Integração	3.1.1	Comunicação I2C Arduino - Arduino
	3.1.2	Comunicação I2C Arduino - Nivelador de Tensão - BeagleBone Black
	3.1.3	Comunicação GPIO BeagleBone Black - Nivelador de Tensão - Arduinos
	3.1.4	Comunicação GPIO BeagleBone Black - BeagleBone Black
	3.1.5	Comunicação I2C Múltiplos Arduinos - Nivelador de Tensão - BeagleBone Black
	4.1	Módulo Atuador - Leitura Sensores (MMA1270D)
	4.2	Módulo Sensor - Acionamento Propulsor (T200, ESC30A, Circuito OR)
	4.3	Módulo Experimental - Leitura Sensor, Acionamento Propulsor
	4.4.1	Módulo Reserva - Nivelador de Tensão - Reset Arduinos
	4.4.2	Módulo Reserva - Monitoramento Módulo Experimental
	4.4.3	Módulo Reserva - Assumindo Controle AUV
	4.4.4	Módulo Reserva - Leitura Sensor, Acionamento Propulsor
	4.4.5	Módulo Reserva - Falha módulo Sensor / Atuador
	4.5	Módulo Emergência - Monitoramento Módulo Reserva

Fonte: Autoria Própria.

contendo o estado do sistema e módulos, é apresentada na Tabela 3. O Teste 1 é detalhado na Seção 5.1 (Inicialização do Protótipo de Bancada), os Testes 2 e 3 na Seção 5.2 (Rotina Experimental - Reserva) e o Teste 4 na Seção 5.3 (Rotina de Emergência).

5.1 INICIALIZAÇÃO DO PROTÓTIPO DE BANCADA

Numa inicialização em um cenário sem falhas, os módulos responsáveis pelos sensores e atuadores executam suas rotinas de inicialização e então aguardam o recebimento de mensagens enviadas pelo módulo de controle. O módulo de controle experimental inicializa e, durante sua rotina, envia pedidos de leitura dos sensores e percentagem de velocidade a ser aplicado ao propulsor, ao mesmo tempo que envia o sinal contendo pulsos periódicos ao módulo reserva, para indicar que está em funcionamento. Os módulos reserva e de emergência, após aguardarem um tempo de inicialização pré definido, monitoram o módulo experimental e reserva, respectivamente, sendo acionados em caso de detecção de falhas.

A inicialização do sistema foi validada pela observação da correta inicialização dos módulos dos sensores / atuadores executando a rotina experimental, e o módulo de emergência e reserva aguardando e monitorando os outros módulos. O estado de cada módulo pode ser verificado pela observação dos LEDs de estado ou conectando-os diretamente com o computador do desenvolvedor e enviando mensagens contendo os dados necessários para validação do seu funcionamento.

5.2 ROTINA EXPERIMENTAL - RESERVA

Os módulos responsáveis pelos sensores e atuadores foram configurados para, durante a rotina experimental, serem capazes de realizar a leitura dos sensores e acionar o propulsor, respectivamente. No modo reserva, foram configurados para terem um estado adicional de emergência, no qual são capazes de assumir a função do outro módulo, ou seja, cada um pode tanto realizar a leitura do sensor quanto acionar o

Tabela 3 – Testes funcionais do protótipo.

Teste	Estado Sistema	Módulo Experimental	Módulo Reserva	Módulo Sensores	Módulo Atuadores	Módulo Emergência	Acelerômetro	Propulsor
1	Inicializando Protótipo	Setup, Estabelecendo Conexão	Setup, Estabelecendo Conexão	Setup Rotina Experimental, Estabelecendo Conexão	Setup Rotina Experimental, Estabelecendo Conexão	Setup, Estabelecendo Conexão	Executando	Inicializando
2.1	Rotina Experimental	Loop: Lendo Sensor Módulo Sensor; Enviando PWM Módulo Atuador;	Loop: Monitorando Módulo Experimental	Loop Experimental: Lendo Sensores	Loop Experimental: Controlando Propulsor	Loop: Monitorando Módulo Reserva	Executando	Executando
2.2	Rotina Experimental, Falha Módulo Sensor	Loop: Detectada Falha no módulo Sensor; Enviando PWM Módulo Atuador;	Loop: Monitorando Módulo Experimental	Simulando Falha	Loop Experimental: Controlando Propulsor	Loop: Monitorando Módulo Reserva	Executando	Executando
2.3	Rotina Experimental, Falha Módulo Atuador	Loop: Lendo Sensor Módulo Sensor; Detectada Falha no módulo Atuador;	Loop: Monitorando Módulo Experimental	Loop Experimental: Lendo Sensores	Simulando Falha	Loop: Monitorando Módulo Reserva	Executando	Parado
2.4	Rotina Experimental, Falha Módulo Sensor e Atuador	Loop: Detectada Falha no módulo Sensor; Detectada Falha no módulo Atuador;	Loop: Monitorando Módulo Experimental	Simulando Falha	Simulando Falha	Loop: Monitorando Módulo Reserva	Executando	Parado
2.5	Rotina Experimental, Falha Módulo Reserva	Loop: Detectada Falha no módulo Sensor; Detectada Falha no módulo Atuador;	Simulando Falha	Loop Experimental: Lendo Sensores	Loop Experimental: Lendo Sensores	Acionando Dispositivos de Emergência	Executando	Executando
3.1	Inicialização Rotina Reserva	Simulando Falha	Detectada Falha no Módulo Experimental; Reinicia Submódulos Sensores / Atuadores; Comunica que está assumindo controle do AUV;	Reiniciando, Setup Rotina Reserva	Reiniciando, Setup Rotina Reserva	Loop: Monitorando Módulo Reserva	Executando	Parado
3.2	Rotina Reserva	Simulando Falha	Loop: Lendo Sensor Módulo Sensor; Enviando PWM Módulo Atuador;	Loop Reserva: Lendo Sensores	Loop Reserva: Controlando Propulsor	Loop: Monitorando Módulo Reserva	Executando	Executando
3.3	Rotina Reserva, Falha Módulo Sensor	Simulando Falha	Loop: Detectada Falha no Módulo Sensor; Comunica módulo Atuador; Enviando PWM Módulo Atuador; Lendo Sensores Módulo Atuador;	Simulando Falha	Loop Emergência: Controlando Propulsor, Lendo Sensores	Loop: Monitorando Módulo Reserva	Executando	Executando
3.4	Rotina Reserva, Falha Módulo Atuador	Simulando Falha	Loop: Detectada Falha no Módulo Atuador; Comunica módulo Sensor; Enviando PWM Módulo Sensor; Lendo Sensores Módulo Sensor;	Loop Emergência: Controlando Propulsor, Lendo Sensores	Simulando Falha	Loop: Monitorando Módulo Reserva	Executando	Executando
3.5	Rotina Reserva, Falha Módulo Sensor e Atuador	Simulando Falha	Loop: Detectada Falha no Módulo Atuador e Sensor; Reinicia ambos os Módulos;	Simulando Falha	Simulando Falha	Loop: Monitorando Módulo Reserva	Executando	Parado
3.6	Rotina Reserva, Módulo Experimental Reiniciado	Loop: Detectada Falha no módulo Sensor; Detectada Falha no módulo Atuador;	Loop: Lendo Sensor Módulo Sensor; Enviando PWM Módulo Atuador;	Loop Reserva: Lendo Sensores	Loop Reserva: Controlando Propulsor	Loop: Monitorando Módulo Reserva; Acionando	Executando	Executando
3.7	Reserva, Falha Módulo Reserva	Sem impacto	Simulando Falha	Loop Reserva: Lendo Sensores	Loop Reserva: Controlando Propulsor	Dispositivos de Emergência	Executando	Executando último comando
4.1	Falha Módulo Reserva e Experimental	Simulando Falha	Simulando Falha	Loop Reserva: Lendo Sensores	Loop Reserva: Controlando Propulsor	Dispositivos de Emergência	Executando	Executando último comando
4.2	Falha Módulo Reserva, Experimental e Submódulos	Simulando Falha	Simulando Falha	Simulando Falha	Simulando Falha	Acionando Dispositivos de Emergência	Desligado	Parado
4.3	Falha Módulo Emergência	Sem impacto	Sem impacto	Sem impacto	Sem impacto	Simulando Falha	Sem impacto	Sem impacto

Fonte: Autoria Própria.

propulsor, se necessário.

O módulo de controle experimental realiza a leitura dos sensores, solicitando e recebendo a informação via comunicação I2C com o módulo responsável pelos sensores, e envia mensagens ao módulo do atuador informando o PWM a ser aplicado no acionamento do propulsor. Na ocorrência de falha no módulo experimental, o módulo reserva assume o controle do AUV: O módulo reserva possui uma rotina de funcionamento semelhante ao módulo experimental, mas implementado com a função adicional de monitorar o estado dos Arduinos por meio de mensagens, e sendo capaz de reiniciá-los. Para os testes de bancada, se algum módulo de sensores ou atuadores falhar, e o módulo reserva estiver em ação, este indica ao módulo ainda funcional que assuma as funções do módulo em falha. Se ambos falharem o módulo de controle irá reinicia-los.

Para verificação destes itens, os Arduinos foram conectados com o computador do desenvolvedor via porta USB e comunicação serial, e os Beaglebones Black via SSH. Todas as mensagens recebidas e enviadas pelos módulos eram comunicadas diretamente ao desenvolvedor.

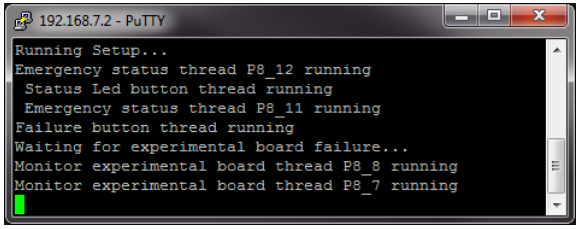
Após todos os módulos estarem funcionando e a rotina experimental estar em execução, foi forçada a falha no módulo de controle experimental (pressionando o botão de simulação de falhas do módulo). O módulo reserva então assumiu o controle e reiniciou os Arduinos, que iniciaram executando a rotina experimental. Então foram forçadas as falhas dos Arduinos, observando se o outro módulo assumia as suas funções. A Figura 48 apresenta um exemplo de comunicação via *PuTTY*[1] (Ver Apêndice A) entre o computador do desenvolvedor e o Beaglebone Black do módulo reserva, enquanto este inicializa e monitora o módulo experimental.

5.3 ROTINA DE EMERGÊNCIA

O módulo de emergência foi configurado para acionar um LED de emergência e sirene, simulando um *ping* de sonar, em caso de falha do módulo reserva.

[1] <https://www.putty.org/>

Figura 48 – Módulo reserva inicializando e monitorando o módulo experimental.



Fonte: Autoria própria.

O funcionamento deste módulo foi validado observando sua correta inicialização e posterior monitoramento do módulo reserva, e se executava a rotina de emergência, acionando a sirene e LED de emergência, quando forçada a falha do módulo reserva.

Para modificações e testes futuros, o módulo responsável pelo acionamento do atuador pode ser programado para forçar a parada do atuador, caso não receba mensagens de atualização de velocidade após um limite de tempo pré determinado. Isso também poderia ser feito pelo próprio módulo de emergência, com as devidas modificações no circuito elétrico.

6 CONCLUSÃO

No presente trabalho, foi desenvolvida uma arquitetura tolerante a falhas para AUVs, com implementação e testes em um protótipo de bancada, atendendo o objetivo geral e específicos, como o levantamento do estado da arte de robôs subaquáticos não tripulados, análise de arquiteturas empregadas em AUVs e considerando o contexto de aplicação e uso em ambiente acadêmico.

Foram consideradas diversas propostas e possibilidades de arquitetura, sendo escolhida a que ofereceu mais oportunidades de experimentação e aplicasse mais técnicas de tolerância a falhas. Esta proposta foi então refinada, detalhada e colocada em prática em um protótipo de bancada. De modo geral, a proposta final aplicou e atendeu as seguintes técnicas de tolerância a falhas:

- Redundância de Hardware:
  - Técnica Híbrida, as falhas são detectadas e tratadas, alguns canais e ligações foram duplicados (Ver Figura 39 na Página 85).
- Redundância de Software:
  - Detecção: Verificação de tempo de resposta e análise de sinal para detecção de falha em módulos;
  - Contenção: Divisão por módulos, a falha de um módulo não deve ser propagada e causar a falha de outros módulos;
  - Recuperação: *Reset* de módulos e execução em pares;
  - Múltiplo versionamento: Recuperação em blocos, diferentes versões de software e rotinas.

Para que o AUV suporte a falha de um módulo sensor ou atuador, ambos podem ser programados para assumir a função do outro, ou mudar sua rotina padrão, de acordo com a lógica de controle do AUV desejada pelo usuário. Dependendo da estratégia a ser utilizada,

alterações no circuito elétrico deverão ser feitas. Durante os testes do protótipo de bancada esta possibilidade foi experimentada durante a rotina de controle do módulo reserva.

Correlacionando com as técnicas de redundância estudadas, pode ser observada a presença de redundância entre o módulo responsável pelos sensores e o dos atuadores. Com as devidas modificações no circuito também existe a possibilidade de instalar camadas adicionais de redundância, uma delas utilizando o próprio módulo de controle para assumir as responsabilidades de outros módulos e centralizando as responsabilidades, se necessário.

Foi possível verificar o comportamento hierárquico do sistema, sendo o módulo de emergência a última camada a ser acionada na ocorrência de falha do restante do sistema, e o módulo reserva possuindo a capacidade de assumir o controle do AUV em caso de falha do módulo experimental, e tendo controle sobre o funcionamento dos módulos dos sensores e atuadores. A técnica de múltiplo versionamento é aplicada, suportando a aplicação de lógicas distintas nos módulos dos sensores / atuadores (contendo cada um, no mínimo, uma rotina experimental e outra reserva), e nos módulos de controle e inteligência (um executando a lógica experimental e outro a reserva), podendo ser implementadas por diferentes equipes. Um cuidado deve ser tomado em relação às modificações nas ligações dos componentes eletrônicos e pinagens, pois os programas executados pelos módulos devem ser atualizados de acordo.

Os componentes utilizados pelo protótipo de bancada não são definitivos, qualquer módulo pode ser substituído e outros componentes empregados, sendo esta apenas uma das possibilidades de aplicação da arquitetura. Modificações e otimização de software e hardware podem ser efetuadas de acordo com a aplicação do AUV.

O sistema é flexível, permitindo que os usuários implementem alterações e testem diferentes configurações, tanto em nível de hardware quanto de software. A nível de hardware, os módulos são cambiáveis, se respeitadas as interfaces de comunicação entre eles e atentando-se aos níveis lógicos de tensão. A nível de software, a linguagem de programação e códigos entre módulos são independentes uns dos outros,

mas também é necessário respeitar o protocolo de comunicação entre módulos estabelecido pela equipe.

Em caso de estratégia semelhante à utilizada nos submódulos do protótipo de bancada, nos quais foram implementadas duas rotinas em cada um (reserva e experimental), quaisquer modificações a nível de software podem ser feitas nas suas respectivas divisões, porém, no caso de alterações a nível de hardware, deve-se verificar o impacto destas mudanças em ambas as rotinas.

Modificações na implementação da arquitetura proposta podem ser feitas de acordo com as missões a serem cumpridas pelo AUV e estratégia da equipe responsável. Módulos adicionais podem ser empregados em setores específicos do AUV, podendo utilizar o canal I2C para comunicação com os módulos de controle. Rotinas adicionais podem ser programadas para diferentes situações que o AUV possa encontrar. Mais informações em relação à programação do protótipo podem ser encontradas no Apêndice A.

Sempre que forem feitas modificações no protótipo recomenda-se realizar simulações e testes de falhas no módulo, verificando e assegurando que falhas não se propaguem para outros módulos do AUV.

Como próximos passos é prevista a adaptação de um ROV de arquitetura aberta para operar como um AUV, empregando a arquitetura proposta e sendo realizados testes em campo.

REFERÊNCIAS

Abran, A., Moore, J. W., Bourque, P., Dupuis, R., Tripp, L. L. *SWEBOK: A Project of the Software Engineering Coordinating Committee*. Los Alamitos, California: IEEE, 2001. v. 1.00. 137–145 p. ISBN 0769510000. 93

ANTONELLI, G. *Underwater Robots*. Italy: Springer, 2016. ISBN 9783319028767. 31, 51, 58

ASLANSEFAT, K.; LATIF-SHABGAHI, G.; KAMARLOUEI, M. A Strategy for Reliability Evaluation And Fault Diagnosis Of Autonomous Underwater Gliding Robot Based On Its Fault Tree. n. October, p. 978–93, 2014. 60, 65, 66

BACHMANN, A.; WILLIAMS, S. Terrain aided underwater navigation-a deeper insight into generic monte carlo localization. *Australasian Conference on Robotics & Automation (ACRA)*, p. 1–7, 2003. Disponível em: <http://www.araa.asn.au/acra/acra2003/papers/22.pdf>. 36

BACK, N. et al. *Projeto Integrado de Produtos*. Brazil: Manole, 2008. ISBN 9788520422083. 16, 67, 68

BELBACHIR, A.; INGRAND, F.; LACROIX, S. A cooperative architecture for target localization using multiple AUVs. *Intelligent Service Robotics*, v. 5, n. 2, p. 119–132, 2012. ISSN 18612776. 54

BIAN, X. et al. Mission management and control of BSA-AUV for ocean survey. *Ocean Engineering*, v. 55, p. 161–174, 2012. ISSN 00298018. 47, 49, 56, 57

BONIN-FONT, F. et al. Visual sensing for autonomous underwater exploration and intervention tasks. *Ocean Engineering*, Elsevier, v. 93, p. 25–44, 2015. ISSN 00298018. Disponível em: <http://dx.doi.org/10.1016/j.oceaneng.2014.11.005>. 33, 38, 39

BRAGA, J.; CALADO, P.; SOUSA, J. An Inside Perspective on LAUV Control and Localization Layers. *IFAC-PapersOnLine*, Elsevier Ltd., v. 48, n. 2, p. 143–148, 2015. ISSN 24058963. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S2405896315002621>. 35

BRITO, M.; SMEED, D.; GRIFFITHS, G. Underwater glider reliability and implications for survey design. *Journal of Atmospheric and Oceanic Technology*, v. 31, n. 12, p. 2858–2870, 2014. ISSN 15200426. 60

BUSQUETS, J. et al. Low-cost AUV Alba13 as multi-sensor platform with water sampler capabilities, for application in multi-agent ocean research applications. *2014 Oceans - St. John's, OCEANS 2014*, n. 1, 2015. 45, 46, 47

BUSQUETS-MATAIX, J. Combined gas-fluid buoyancy system for improved attitude and maneuverability control for application in underwater gliders. *IFAC Proceedings Volumes (IFAC-PapersOnline)*, Elsevier Ltd., v. 48, n. 2, p. 281–287, 2015. ISSN 14746670. Disponível em: <http://dx.doi.org/10.1016/j.ifacol.2015.06.046>. 36, 38

CARRERA, A. et al. Free-floating panel intervention by means of learning by demonstration. *IFAC Proceedings Volumes (IFAC-PapersOnline)*, Elsevier Ltd., v. 48, n. 2, p. 38–43, 2015. ISSN 14746670. Disponível em: <http://dx.doi.org/10.1016/j.ifacol.2015.06.007>. 38, 39

CARRERAS, M. et al. Testing Sparus II AUV, an open platform for industrial, scientific and academic applications. *Instrumentation Viewpoint*, n. 18, p. 54–55, 2015. ISSN 1886-4864. Disponível em: <http://upcommons.upc.edu/handle/2117/77636>. 45

CASHMORE, M. et al. Artificial intelligence planning for AUV mission control. *IFAC Proceedings Volumes (IFAC-PapersOnline)*, v. 48, n. 2, p. 262–267, 2015. ISSN 14746670. 43

CASTANO, V.; SCHAGAEV, I. *Resilient computer system design*. Stevenage, UK: Springer, 2015. 1–256 p. ISBN 9783319150697. 61

CRESTANI, D.; GODARY-DEJEAN, K.; LAPIERRE, L. Enhancing fault tolerance of autonomous mobile robots. *Robotics and Autonomous Systems*, Elsevier B.V., v. 68, p. 140–155, 2015. ISSN 09218890. Disponível em: <http://dx.doi.org/10.1016/j.robot.2014.12.015>. 64, 65

DEARDEN, R.; ERNITS, J. Peer-Reviewed Technical Communication Automated Fault Diagnosis for an Autonomous Underwater Vehicle. v. 38, n. 3, p. 484–499, 2013. 58

DUBROVA, E. *Fault-tolerant design*. Sweden: Springer, 2013. 1–185 p. ISBN 9781461421139. 61, 62

EISENMANN, T. R.; PARKER, G.; Van Alstyne, M. Opening Platforms: How, When and Why? *Platforms, Markets and Innovation*, p. 131–162, 2009. ISSN 08987629. Disponível em:

<http://search.ebscohost.com/login.aspx?direct=true{&}db=bth{&}AN=43456312{&}site=e>. 43

HAN, J.; OK, J.; CHUNG, W. K. An ethology-based hybrid control architecture for an autonomous underwater vehicle for performing multiple tasks. *IEEE Journal of Oceanic Engineering*, v. 38, n. 3, p. 514–521, 2013. ISSN 03649059. 54

HE, B. et al. A Distributed Parallel Motion Control for the Multi-Thruster Autonomous Underwater Vehicle. *Mechanics Based Design of Structures and Machines*, v. 41, n. 2, p. 236–257, 2013. ISSN 1539-7734. Disponível em: <http://www.tandfonline.com/doi/abs/10.1080/15397734.2012.726847>. 49, 50

INSAURRALDE, C. C.; PETILLOT, Y. R. Capability-oriented robot architecture for maritime autonomy. *Robotics and Autonomous Systems*, Elsevier B.V., v. 67, p. 87–104, 2015. ISSN 09218890. Disponível em: <http://dx.doi.org/10.1016/j.robot.2014.10.003>. 25, 64

JO, K. et al. Development of Autonomous Car–Part I: distributed system architecture and development process. *IEEE Trans.\ Ind.\ Electronics*, v. 61, n. 12, p. 7131–7140, 2014. 45

KALWA, J. et al. EU project MORPH: Current status after 3 years of cooperation under and above water. *IFAC Proceedings Volumes (IFAC-PapersOnline)*, Elsevier Ltd., v. 48, n. 2, p. 119–124, 2015. ISSN 14746670. Disponível em: <http://dx.doi.org/10.1016/j.ifacol.2015.06.019>. 40, 41

KIM, D. et al. Artificial landmark-based underwater localization for AUVs using weighted template matching. *Intelligent Service Robotics*, v. 7, n. 3, p. 175–184, 2014. ISSN 18612784. 33

KOOPMAN, P.; WAGNER, M. Autonomous Vehicle Safety: An Interdisciplinary Challenge. *IEEE Intelligent Transportation Systems Magazine*, v. 9, n. 1, p. 90–96, 2017. ISSN 19391390. 60

LEI, Z. et al. An AUV for Ocean Exploring and its Motion Control System Architecture. *Open Mechanical Engineering Journal*, v. 7, p. 40–47, 2013. ISSN 1874155X. Disponível em: <https://benthamopen.com/contents/pdf/TOMEJ/TOMEJ-7-40.pdf>. 55, 58

LOC, M. B. et al. Development and control of a new AUV platform. *International Journal of Control, Automation and Systems*, v. 12, n. 4, p. 886–894, 2014. ISSN 20054092. 50

MAHMOUDZADEH, S.; POWERS, D. M.; SAMMUT, K. An autonomous reactive architecture for efficient AUV mission time management in realistic dynamic ocean environment. *Robotics and Autonomous Systems*, Elsevier B.V., v. 87, p. 81–103, 2016. ISSN 09218890. Disponível em: <http://dx.doi.org/10.1016/j.robot.2016.09.007>. 55, 56

MARSHALL, C.; ROBERTS, B.; GRENN, M. Intelligent Control & Supervision for Autonomous System Resilience in Uncertain Worlds. p. 438–443, 2017. 59

MINTCHEV, S. et al. Towards docking for small scale underwater robots. *Autonomous Robots*, v. 38, n. 3, p. 283–299, 2014. ISSN 09295593. 42

NAKHAEINIA, D. et al. A review of control architectures for autonomous navigation of mobile robots. *International Journal of the Physical Sciences*, v. 6, n. 2, p. 169–174, 2011. ISSN 1992-1950. 51, 53

PALOMERAS, N. et al. COLA2: A control architecture for AUVs. *IEEE Journal of Oceanic Engineering*, v. 37, n. 4, p. 695–716, 2012. ISSN 03649059. 52, 53, 57, 58, 59

PAULL, L. et al. AUV navigation and localization: A review. *IEEE Journal of Oceanic Engineering*, v. 39, n. 1, p. 131–149, 2014. ISSN 03649059. 37, 40

PEÑALVER, A. et al. Visually-guided manipulation techniques for robotic autonomous underwater panel interventions. *Annual Reviews in Control*, v. 40, p. 201–211, 2015. ISSN 13675788. 38

PETRIOLI, C. et al. The SUNSET framework for simulation, emulation and at-sea testing of underwater wireless sensor networks. *Ad Hoc Networks*, Elsevier B.V., v. 34, p. 224–238, 2015. ISSN 15708705. 41

RIBAS, D.; RIDAO, P.; CARRERAS, M. Girona 500 AUV : From Survey to Intervention. v. 17, n. 1, p. 46–53, 2012. 34

RIDAO, P. et al. Intervention AUVs: The next challenge. *Annual Reviews in Control*, Elsevier Ltd, v. 40, p. 227–241, 2015. ISSN 13675788. Disponível em: <http://dx.doi.org/10.1016/j.arcontrol.2015.09.015>. 38

ROUMELIOTIS, S. I.; BEKEY, G. A. Distributed multirobot localization. *IEEE Transactions on Robotics and Automation*, v. 18, n. 5, p. 781–795, 2002. ISSN 1042296X. 39

SANGEKAR, M.; CHITRE, M.; KOAY, T. B. Hardware architecture for a modular autonomous underwater vehicle STARFISH. *Oceans 2008*, 2008. 33, 45, 46, 48

SETO, M. L.; BASHIR, A. Z. Fault tolerance considerations for long endurance AUVs. *Proceedings - Annual Reliability and Maintainability Symposium*, 2017. ISSN 0149144X. 60, 64

SHUKLA, A.; KARKI, H. Application of robotics in offshore oil and gas industry-A review Part II. *Robotics and Autonomous Systems*, Elsevier B.V., v. 75, p. 508–524, 2016. ISSN 09218890. Disponível em: <http://dx.doi.org/10.1016/j.robot.2015.09.013>. 26

SICILIANO, B.; KHATIB, O. (Ed.). *Springer Handbook of Robotics*. 2. ed. Heidelberg: Springer, 2008. ISBN 978-3-540-23957-4. 31, 37, 42

SORENSEN, A. J.; LUDVIGSEN, M. Towards integrated autonomous underwater operations. *IFAC Proceedings Volumes (IFAC-PapersOnline)*, v. 48, n. 2, p. 107–118, 2015. ISSN 14746670. 34

VERHOECKX, P. B. Comparison of Control Architectures for Autonomous Mobile Robots. n. 0663727, 2016. 51

WALLS, J. M.; EUSTICE, R. M. An origin state method for communication constrained cooperative localization with robustness to packet loss. *International Journal of Robotics Research*, v. 33, n. 9, p. 1191–1208, 2014. ISSN 0278-3649. 41

Apêndices

APÊNDICE A – PROGRAMAÇÃO BEAGLEBONE BLACK

A.1 ACESSO VIA PROMPT DE COMANDO

Para comunicar o PC do desenvolvedor e o BeagleBone Black via Prompt de Comando é possível instalar o software PuTTY, disponível para Linux e Windows. Inicializar o programa e selecionar SSH, entrando o endereço do BeagleBone, cujo padrão é: ip 192.168.7.2 e porta 22. A senha de acesso padrão é "root", sem necessitar de senha no sistema operacional Debian.

A.2 ACESSAR ÁREA DE TRABALHO REMOTAMENTE

Para acessar remotamente a área de trabalho do Beaglebone é possível instalar o TightVNC server na placa, com os comandos:

```
sudo apt-get update && sudo apt-get upgrade
sudo apt-get install tightvncserver
```

Após a instalação digite "vncserver"no Promt de Comando do BeagleBone, a primeira vez que executar o comando será necessário cadastrar uma senha para o usuário atual. Para os BeagleBones testados em bancada a senha utilizada foi "beaglebone".

No computador do desenvolvedor instalar e executar o TightVNC Java Viewer, informando os dados configurados no BeagleBone quando forem peerguntados pelo software:

```
Remote Host: 192.168.7.2
Port: 5901
Use SSH tunneling: Yes
SSH Server: 192.168.7.2
SSH Port: 22
SSH User: root
```

Depois será pedida a senha configurada no TightVNC server instalada na placa BeagleBone e a área de trabalho da placa será acessada no computador do usuário.

A.3 TRANSFERÊNCIA DE ARQUIVOS VIA WINDOWS

Para transferir arquivos para o BeagleBone (por exemplo: o código de inteligência a ser utilizado pelo módulo) é possível realizar pelo próprio TightVNC via Linux, ou utilizando um software como o WinSCP pelo Windows.

Após instalado o aplicativo WInSCP, configurar o endereço IP do BeagleBone, usuário root e nenhuma senha (para a configuração padrão). Ao conectar uma janela com dois painéis será aberta, uma correspondente ao BeagleBone e outra do computador do usuário. Para transferir arquivos é possível arrastar os arquivos de um painel ao outro.

A.4 EXECUTAR ROTINA AO INICIALIZAR

Para que o BeagleBone Black execute as rotinas ao inicializar, semelhante ao funcionamento do Arduino, acessar o BeagleBone via PuTTy, instalar o crontab:

```
sudo apt-get install crontab
```

Abrir o editor crontab:

```
crontab -e
```

Ao final do arquivo adicionar um comando contendo o endereço do arquivo a ser executado durante a inicialização, por exemplo:

```
@reboot sudo  python /root/Code/NomeArquivo.py &
```

Não esquecer de adicionar o **&** após o endereço do arquivo, este comando faz com que o processo rode em segundo plano quando executado.

APÊNDICE B – DIAGRAMA ELÉTRICO

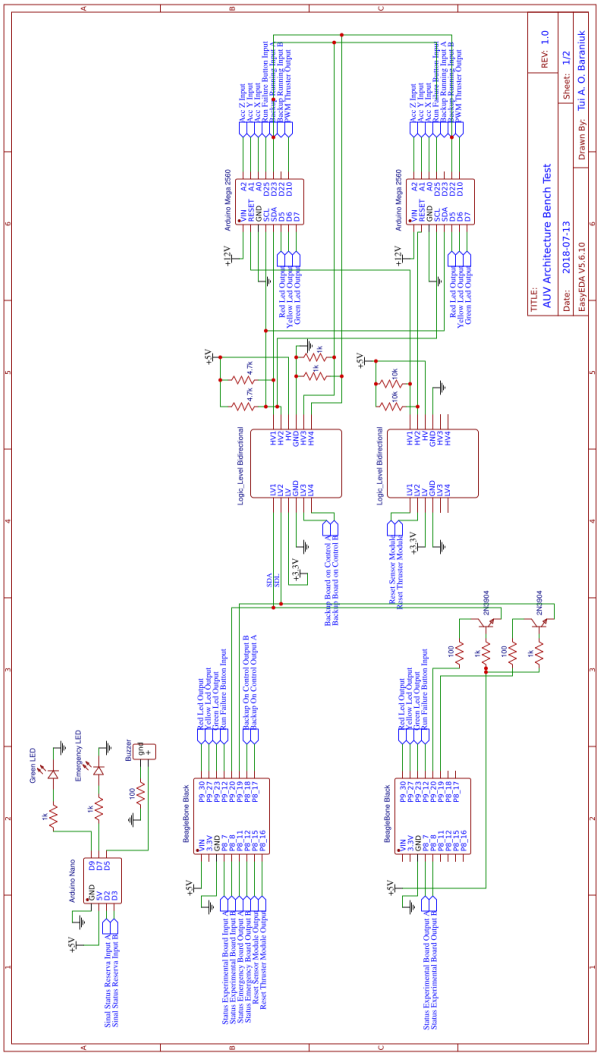


Figura B.49 – Esquemático do circuito do protótipo de bancada, parte 1. Fonte: Autoria própria.

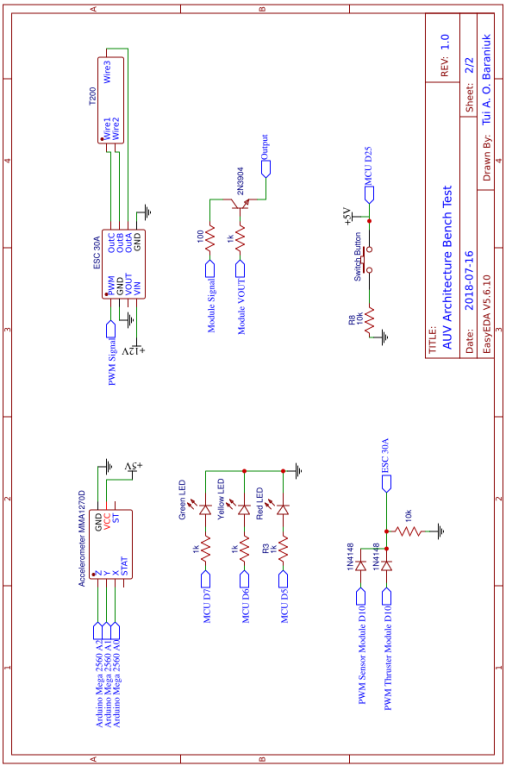


Figura B.50 – Esquemático do circuito do protótipo de bancada, parte 2. Fonte: Autoria própria.

APÊNDICE C – CÓDIGOS PROTÓTIPO

C.1 MÓDULO DE CONTROLE RESERVA

```
# Fault Tolerance Backup BeagleBone Black Test
# By: Tui Alexandre Ono Baraniuk
import Adafruit_BBIO.GPIO as GPIO
import smbus
import time
import thread
import random
import struct

# Constant variables
SENSORS_MODULE_ADDRESS = 8
THRUSTER_MODULE_ADDRESS = 9

IM_ALIVE_EXPERIMENTAL_PIN_A = "P8_7"
IM_ALIVE_EXPERIMENTAL_PIN_B = "P8_8"
MY_STATUS_EMERGENCY_PIN_A = "P8_11"
MY_STATUS_EMERGENCY_PIN_B = "P8_12"
SENSOR_MODULE_RESET_PIN = "P8_15"
THRUSTER_MODULE_RESET_PIN = "P8_16"
BACKUP_RUNNING_STATUS_PIN_A = "P8_17"
BACKUP_RUNNING_STATUS_PIN_B = "P8_18"
RUN_FAILURE_TEST_PIN = "P9_12"
LED_GREEN_STATUS_PIN = "P9_23"
LED_YELLOW_STATUS_PIN = "P9_27"
LED_RED_STATUS_PIN = "P9_30"

PING = 1
GET_ACCELEROMETER_DATA = 2
GET_GYROSCOPE_DATA = 3
SET_THRUSTER_PWM = 4
SET_RUN_EMERGENCY_ROUTINE = 5

EXPERIMENTAL_BOARD_WAIT_STARTUP_DELAY = 60

# 0 sensors module, 1 thruster module
arduino_status = [False, False]

bus = smbus.SMBus(1)
standby_status = True

active_led_status_pin = LED_YELLOW_STATUS_PIN
running_deadlock_failure_mode = False
emergency_activated = False

def setup():
    print("Running Setup...")
    setup_pins()
    # test_routine()
    turn_on_arduinos()
    set_backup_running_status(GPIO.LOW)

    start_check_failure_button_thread()
```

```
53      start_status_led_thread()
54      start_emergency_board_thread()
55
56      # wait for experimental board to initialize before checking if
         its running correctly
57      time.sleep(EXPERIMENTAL_BOARD_WAIT_STARTUP_DELAY)
58
59      wait_experimental_board_failure()
60      taking_control_setup()
61      time.sleep(1)
62
63  def loop():
64      print("Entering Backup Module Control Loop...")
65      time_between_readings = 5
66      while True:
67          pwm_percentage_test = random.randint(0, 70)
68          if not running_deadlock_failure_mode:
69              monitor_arduino()
70              get_accelerometer_data()
71              set_thruster(pwm_percentage_test)
72              time.sleep(time_between_readings)
73          pass
74
75  def setup_pins():
76      GPIO.setup(RUN_FAILURE_TEST_PIN, GPIO.IN)
77      GPIO.setup(IM_ALIVE_EXPERIMENTAL_PIN_A, GPIO.IN)
78      GPIO.setup(IM_ALIVE_EXPERIMENTAL_PIN_B, GPIO.IN)
79      GPIO.setup(MY_STATUS_EMERGENCY_PIN_A, GPIO.OUT)
80      GPIO.setup(MY_STATUS_EMERGENCY_PIN_B, GPIO.OUT)
81      GPIO.setup(SENSOR_MODULE_RESET_PIN, GPIO.OUT)
82      GPIO.setup(THRUSTER_MODULE_RESET_PIN, GPIO.OUT)
83      GPIO.setup(BACKUP_RUNNING_STATUS_PIN_A, GPIO.OUT)
84      GPIO.setup(BACKUP_RUNNING_STATUS_PIN_B, GPIO.OUT)
85      GPIO.setup(LED_GREEN_STATUS_PIN, GPIO.OUT)
86      GPIO.setup(LED_YELLOW_STATUS_PIN, GPIO.OUT)
87      GPIO.setup(LED_RED_STATUS_PIN, GPIO.OUT)
88
89  def set_backup_running_status(status):
90      GPIO.output(BACKUP_RUNNING_STATUS_PIN_A, status)
91      GPIO.output(BACKUP_RUNNING_STATUS_PIN_B, status)
92
93  def start_status_led_thread():
94      thread.start_new_thread(blink_status_led, ())
95
96  def blink_status_led():
97      interval = 0.5
98      print("Status Led button thread running")
99      while True:
100         global active_led_status_pin
101         new_state = not GPIO.input(active_led_status_pin)
102         GPIO.output(active_led_status_pin, new_state)
103         time.sleep(interval)
104
105 def set_status_pin(led_pin):
106     global active_led_status_pin
107     active_led_status_pin = led_pin
108     GPIO.output(LED_GREEN_STATUS_PIN, GPIO.LOW)
```

```
109     GPIO.output(LED_YELLOW_STATUS_PIN, GPIO.LOW)
110     GPIO.output(LED_RED_STATUS_PIN, GPIO.LOW)
111
112 def start_check_failure_button_thread():
113     thread.start_new_thread(monitor_failure_button, ())
114
115 def monitor_failure_button():
116     print("Failure button thread running")
117
118     global running_deadlock_failure_mode
119     global active_led_status_pin
120     # initialise a previous input variable to 0 (assume button not
        pressed last)
121     last_input = 0
122     last_active_led_pin = active_led_status_pin
123     debounce_time = 0.05
124
125     while True:
126         # take a reading
127         input = GPIO.input(RUN_FAILURE_TEST_PIN)
128         # if the last reading was low and this one high, print
129         if (not last_input) and input:
130             print("Button pressed")
131             if not running_deadlock_failure_mode:
132                 running_deadlock_failure_mode = True
133                 last_active_led_pin = active_led_status_pin
134                 set_status_pin(LED_RED_STATUS_PIN)
135                 print("Running Failure Simulation Mode")
136             else:
137                 running_deadlock_failure_mode = False
138                 set_status_pin(last_active_led_pin)
139                 print("Returning to Normal Mode")
140
141         last_input = input
142         time.sleep(debounce_time)
143
144 def monitor_arduino():
145     global emergency_activated
146     arduino_status[0] = check_slave_board_status(
        SENSORS_MODULE_ADDRESS)
147     arduino_status[1] = check_slave_board_status(
        THRUSTER_MODULE_ADDRESS)
148     if (not arduino_status[0]) and arduino_status[1] and not
        emergency_activated:
149         print("Error detected on Sensor Module")
150         print("Setting Thruster Module to emergency mode")
151         GPIO.output(SENSOR_MODULE_RESET_PIN, GPIO.LOW)
152         set_arduino_emergency(1, THRUSTER_MODULE_ADDRESS)
153         emergency_activated = True
154         # turn off sensor module
155
156     elif arduino_status[0] and (not arduino_status[1]) and not
        emergency_activated:
157         print("Error detected on Thruster Module")
158         print("Setting Sensor Module to emergency mode")
159         GPIO.output(THRUSTER_MODULE_RESET_PIN, GPIO.LOW)
160         set_arduino_emergency(1, SENSORS_MODULE_ADDRESS)
```

```
161         emergency_activated = True
162         # turn off thruster module
163
164     elif not (arduino_status[0] or arduino_status[1]):
165         print("Error detected on both Thruster and Sensor Module")
166         emergency_activated = True
167         reset_arduinos()
168
169     elif arduino_status[0] and arduino_status[1] and
    emergency_activated:
170         print("Setting arduinos to exit emergency mode")
171         set_arduino_emergency(0, SENSORS_MODULE_ADDRESS)
172         set_arduino_emergency(0, THRUSTER_MODULE_ADDRESS)
173         emergency_activated = False
174
175 def start_emergency_board_thread():
176     thread.start_new_thread(send_my_status_emergency_board, (
    MY_STATUS_EMERGENCY_PIN_A,))
177     thread.start_new_thread(send_my_status_emergency_board, (
    MY_STATUS_EMERGENCY_PIN_B,))
178
179 def wait_experimental_board_failure():
180     thread.start_new_thread(monitor_experimental_board_status, (
    IM_ALIVE_EXPERIMENTAL_PIN_A,))
181     thread.start_new_thread(monitor_experimental_board_status, (
    IM_ALIVE_EXPERIMENTAL_PIN_B,))
182     print("Waiting for experimental board failure...")
183     global standby_status
184     while standby_status:
185         pass
186     print("Experimental board failure detected!")
187
188 # Fault Tolerance Logic
189 def monitor_experimental_board_status(pin):
190     print("Monitor experimental board thread {0} running ".format(pin
    ))
191     global standby_status
192     limit_down_time = 0.5
193     limit_up_time = 2
194     while standby_status:
195         start_time_high = time.time()
196         elapsed_time_on_high = 0
197         while GPIO.input(pin) & (elapsed_time_on_high < limit_up_time
    ):
198             elapsed_time_on_high = time.time() - start_time_high
199
200         start_time_low = time.time()
201         elapsed_time_low = 0
202         while (not GPIO.input(pin)) and elapsed_time_low <
    limit_down_time:
203             elapsed_time_low = time.time() - start_time_low
204
205         # if board is not responding, exit loop
206         if elapsed_time_low > limit_down_time or elapsed_time_on_high
     > limit_up_time:
207             standby_status = False
208
```

```
209 def check_slave_board_status(address):
210     global bus
211     boards_status = True
212     try:
213         bus.write_byte(address, PING)
214     except:
215         print("Ping Error on Address: ", address)
216         boards_status = False
217     return boards_status
218 
219 def taking_control_setup():
220     # Signalize to Sensor and Actuator modules that backup module is
        taking control
221     print("Backup board is taking control, reseting arduinos...")
222     set_status_pin(LED_GREEN_STATUS_PIN)
223     set_backup_running_status(GPIO.HIGH)
224     reset_arduinos()
225 
226 def turn_off_arduinos():
227     GPIO.output(SENSOR_MODULE_RESET_PIN, GPIO.LOW)
228     GPIO.output(THRUSTER_MODULE_RESET_PIN, GPIO.LOW)
229 
230 def turn_on_arduinos():
231     GPIO.output(SENSOR_MODULE_RESET_PIN, GPIO.HIGH)
232     GPIO.output(THRUSTER_MODULE_RESET_PIN, GPIO.HIGH)
233 
234 def reset_arduinos():
235     print("Reseting arduinos...")
236     turn_off_arduinos()
237     # signal sensor and thruster module that the backup control is
        running
238     GPIO.output(BACKUP_RUNNING_STATUS_PIN_A, GPIO.HIGH)
239     GPIO.output(BACKUP_RUNNING_STATUS_PIN_B, GPIO.HIGH)
240     time.sleep(1)
241     turn_on_arduinos()
242     # Give arduinos some time to reset
243     time.sleep(2)
244 
245 def send_my_status_emergency_board(pin):
246     print("Emergency status thread {0} running".format(pin))
247     low_pulse_delay = 0.1
248     high_pulse_delay = 0.5
249 
250     while True:
251         global running_deadlock_failure_mode
252         if not running_deadlock_failure_mode:
253             GPIO.output(pin, GPIO.LOW)
254             time.sleep(low_pulse_delay)
255             GPIO.output(pin, GPIO.HIGH)
256             time.sleep(high_pulse_delay)
257 
258 # Backup AUV Control Routine
259 def get_accelerometer_data():
260     global bus
261     num_bytes = 12
262     if arduino_status[0]:
263         active_module = SENSORS_MODULE_ADDRESS
```

```
264     elif arduino_status[1]:
265         active_module = THRUSTER_MODULE_ADDRESS
266     else:
267         print("Error getting accelerometer_data: Arduinos aren't
        responding!")
268         return
269     bus.write_byte(active_module, GET_ACCELEROMETER_DATA)
270     accelerometer_data = bus.read_i2c_block_data(active_module, 0,
        num_bytes)
271     converted_data = [get_float(accelerometer_data, 0), get_float(
        accelerometer_data, 1),
272                       get_float(accelerometer_data, 2)]
273     print("{0} answer to get_accelerometer_data Request: {1}".format(
        active_module, converted_data))
274
275 def get_float(data, index):
276     byte_array = data[4*index:(index+1)*4]
277     return struct.unpack('f', "".join(map(chr, byte_array)))[0]
278
279 def get_gyroscope_data():
280     global bus
281     num_bytes = 12
282     if arduino_status[0]:
283         active_module = SENSORS_MODULE_ADDRESS
284     elif arduino_status[1]:
285         active_module = THRUSTER_MODULE_ADDRESS
286     else:
287         print("Error getting get_gyroscope_data: Arduinos aren't
        responding!")
288         return
289     bus.write_byte(active_module, GET_GYROSCOPE_DATA)
290     gyroscope_data = bus.read_i2c_block_data(active_module, 0,
        num_bytes)
291     converted_data = [get_float(gyroscope_data, 0), get_float(
        gyroscope_data, 1),
292                       get_float(gyroscope_data, 2)]
293     print("{0} answer to get_gyroscope_data Request: {1}".format(
        active_module, converted_data))
294
295 def set_thruster(pwm_value):
296     global bus
297     if arduino_status[1]:
298         print("Setting new Pwm Thruster value to Thruster_Module: ",
        pwm_value)
299         bus.write_block_data(THRUSTER_MODULE_ADDRESS,
        SET_THRUSTER_PWM, [pwm_value])
300     elif arduino_status[0]:
301         print("Setting new Pwm Thruster value to Sensors_Module: ",
        pwm_value)
302         bus.write_block_data(SENSORS_MODULE_ADDRESS, SET_THRUSTER_PWM
        , [pwm_value])
303     else:
304         print("Error setting PWM: Arduinos aren't responding!")
305
306 def set_arduino_emergency(new_state, address):
307     global bus
308     bus.write_block_data(address, SET_RUN_EMERGENCY_ROUTINE, [
```

```
        new_state])
309     print('Setting {0} emergency mode to {1}'.format(address,
        new_state))
310     time.sleep(0.5)
311
312 def run_exit_routine():
313     print("Running exit routine")
314     turn_off_arduinos()
315     GPIO.cleanup()
316
317 def main():
318     setup()
319     loop()
320
321 if __name__ == "__main__":
322     try:
323         main()
324     except BaseException as e:
325         print(str(e))
326         run_exit_routine()
```

C.2 MÓDULO DE CONTROLE EXPERIMENTAL

```
1  # Fault Tolerance Experimental BeagleBone Black Test
2  # By: Tui Alexandre Ono Baraniuk
3  import Adafruit_BBIO.GPIO as GPIO
4  import smbus
5  import time
6  import thread
7  import random
8  import struct
9
10 # Constant variables
11 SENSORS_MODULE_ADDRESS = 4
12 THRUSTER_MODULE_ADDRESS = 5
13 IM_ALIVE_EXPERIMENTAL_PIN_A = "P8_7"
14 IM_ALIVE_EXPERIMENTAL_PIN_B = "P8_8"
15 RUN_FAILURE_TEST_PIN = "P9_12"
16 LED_GREEN_STATUS_PIN = "P9_23"
17 LED_YELLOW_STATUS_PIN = "P9_27"
18 LED_RED_STATUS_PIN = "P9_30"
19
20 PING = 1
21 GET_ACCELEROMETER_DATA = 2
22 GET_GYROSCOPE_DATA = 3
23 SET_THRUSTER_PWM = 4
24 STARTUP_DELAY = 10
25
26 bus = smbus.SMBus(1)
27
28 active_led_status_pin = LED_YELLOW_STATUS_PIN
29 running_deadlock_failure_mode = False
30
31 def setup():
32     print("Running Setup...")
33     setup_pins()
```

```
34
35      start_check_failure_button_thread()
36      start_status_led_thread()
37      start_send_my_status_thread()
38
39      # wait for other modules to initialize
40      time.sleep(STARTUP_DELAY)
41
42  def loop():
43      print("Entering Experimental Module Control Loop...")
44      time_between_readings = 5
45      set_status_pin(LED_GREEN_STATUS_PIN)
46      while True:
47          pwm_percentage_test = random.randint(0, 70)
48          if not running_deadlock_failure_mode:
49              try:
50                  get_accelerometer_data()
51              except:
52                  print("Error getting Accelerometer data")
53
54              try:
55                  set_thruster(pwm_percentage_test)
56              except:
57                  print("Error setting Thruster PWM")
58          time.sleep(time_between_readings)
59
60  def setup_pins():
61      GPIO.setup(RUN_FAILURE_TEST_PIN, GPIO.IN)
62      GPIO.setup(IM_ALIVE_EXPERIMENTAL_PIN_A, GPIO.OUT)
63      GPIO.setup(IM_ALIVE_EXPERIMENTAL_PIN_B, GPIO.OUT)
64      GPIO.setup(LED_GREEN_STATUS_PIN, GPIO.OUT)
65      GPIO.setup(LED_YELLOW_STATUS_PIN, GPIO.OUT)
66      GPIO.setup(LED_RED_STATUS_PIN, GPIO.OUT)
67
68  def start_status_led_thread():
69      thread.start_new_thread(blink_status_led, ())
70
71  def blink_status_led():
72      interval = 0.5
73      print("Status Led button thread running")
74      while True:
75          global active_led_status_pin
76          new_state = not GPIO.input(active_led_status_pin)
77          GPIO.output(active_led_status_pin, new_state)
78          time.sleep(interval)
79
80  def set_status_pin(led_pin):
81      global active_led_status_pin
82      active_led_status_pin = led_pin
83      GPIO.output(LED_GREEN_STATUS_PIN, GPIO.LOW)
84      GPIO.output(LED_YELLOW_STATUS_PIN, GPIO.LOW)
85      GPIO.output(LED_RED_STATUS_PIN, GPIO.LOW)
86
87  def start_check_failure_button_thread():
88      thread.start_new_thread(monitor_failure_button, ())
89
90  def start_send_my_status_thread():
```

```
91      thread.start_new_thread(send_my_status_backup_board, (
        IM_ALIVE_EXPERIMENTAL_PIN_A,))
92      thread.start_new_thread(send_my_status_backup_board, (
        IM_ALIVE_EXPERIMENTAL_PIN_B,))
93
94  def send_my_status_backup_board(pin):
95      print("Sending my status to backup board thread pin {0} running "
        .format(pin))
96      low_pulse_delay = 0.1
97      high_pulse_delay = 0.4
98
99      while not running_deadlock_failure_mode:
100         GPIO.output(pin, GPIO.LOW)
101         time.sleep(low_pulse_delay)
102         GPIO.output(pin, GPIO.HIGH)
103         time.sleep(high_pulse_delay)
104
105 def monitor_failure_button():
106     print("Failure button thread running")
107
108     global running_deadlock_failure_mode
109     global active_led_status_pin
110     # initialise a previous input variable to 0 (assume button not
        pressed last)
111     last_input = 0
112     last_active_led_pin = active_led_status_pin
113     debounce_time = 0.05
114
115     while True:
116         # take a reading
117         input = GPIO.input(RUN_FAILURE_TEST_PIN)
118         # if the last reading was low and this one high, print
119         if (not last_input) and input:
120             print("Button pressed")
121             if not running_deadlock_failure_mode:
122                 running_deadlock_failure_mode = True
123                 last_active_led_pin = active_led_status_pin
124                 set_status_pin(LED_RED_STATUS_PIN)
125                 print("Running Failure Simulation Mode")
126             else:
127                 running_deadlock_failure_mode = False
128                 set_status_pin(last_active_led_pin)
129                 print("Returning to Normal Mode")
130
131         last_input = input
132         time.sleep(debounce_time)
133
134 # Control Routine
135 def get_accelerometer_data():
136     global bus
137     num_bytes = 12
138
139     bus.write_byte(SENSORS_MODULE_ADDRESS, GET_ACCELEROMETER_DATA)
140     accelerometer_data = bus.read_i2c_block_data(
        SENSORS_MODULE_ADDRESS, 0, num_bytes)
141     converted_data = [get_float(accelerometer_data, 0), get_float(
        accelerometer_data, 1),
```

```
                        get_float(accelerometer_data, 2)]
    print("{0} answer to get_accelerometer_data Request: {1}".format(
    SENSORS_MODULE_ADDRESS, converted_data))

def get_float(data, index):
    byte_array = data[4*index:(index+1)*4]
    return struct.unpack('f', "".join(map(chr, byte_array)))[0]

def get_gyroscope_data():
    global bus
    num_bytes = 12

    bus.write_byte(SENSORS_MODULE_ADDRESS, GET_GYROSCOPE_DATA)
    gyroscope_data = bus.read_i2c_block_data(SENSORS_MODULE_ADDRESS,
    0, num_bytes)
    converted_data = [get_float(gyroscope_data, 0), get_float(
    gyroscope_data, 1),
                        get_float(gyroscope_data, 2)]
    print("{0} answer to get_gyroscope_data Request: {1}".format(
    SENSORS_MODULE_ADDRESS, converted_data))

def set_thruster(pwm_value):
    global bus
    print("Setting new Pwm Thruster value to Thruster_Module: ",
    pwm_value)
    bus.write_block_data(THRUSTER_MODULE_ADDRESS, SET_THRUSTER_PWM, [
    pwm_value])

def run_exit_routine():
    print("Running exit routine")
    GPIO.cleanup()

def main():
    setup()
    loop()

if __name__ == "__main__":
    try:
        main()
    except BaseException as e:
        print(str(e))
        run_exit_routine()
```

C.3 MÓDULO SENSORES

```
/*
 * Fault Tolerance Sensors Module Implementation Test
 * By: Tui Alexandre Ono Baraniuk
*/

#include <Wire.h>
#include <TimedAction.h>
#include <Servo.h>

/*
 * Pins variables are organized here.
```

```
12  * Any eletronic connection changes should be done with the
        surpevision of the backup routine team.
13  */
14
15 const int backupStatusInputApin = 22;
16 const int backupStatusInputBpin = 23;
17 const int accelerometerPins[] = {0, 1, 2};
18
19 // Led Status Pin for Bench Tests
20 const int greenLedPin = 7;
21 const int yellowLedPin = 6;
22 const int redLedPin = 5;
23
24 const int pwmThrusterPin = 10;
25 const int runFailureButtonPin = 25;
26
27
28 /*****************************/
29
30 bool backupControlStatusOnSetup;
31
32 void setup() {
33   // Bench tests setup
34   Serial.begin(9600); // start serial for output for TESTS DEBUGING
35   benchTestsPinsSetup();
36
37   // Control setup
38   backupControlStatusOnSetup = getBackupControlStatus(); // Verify
        what routine to run
39   if(backupControlStatusOnSetup)
40   {
41     backupSetup();
42   }
43   else
44   {
45     experimentalSetup();
46   }
47 }
48
49 void loop() {
50   testLoopRoutine();
51   if(backupControlStatusOnSetup)
52   {
53     backupLoop();
54   }
55   else
56   {
57     experimentalLoop();
58   }
59 }
60
61 // FAULT TOLERANCE
62 bool getBackupControlStatus()
63 {
64   bool backupStatus = false;
65   pinMode(backupStatusInputApin, INPUT);
66   pinMode(backupStatusInputBpin, INPUT);
```

```
67    if(digitalRead(backupStatusInputApin) == HIGH || digitalRead(
          backupStatusInputBpin) == HIGH)
68    {
69      backupStatus = true;
70    }
71
72    return backupStatus;
73  }
74
75
76  /*
77   * BACKUP SENSORS MODULE ROUTINE
78   */
79  const int backupI2CbusAddress = 8;
80
81  const int backupPing = 1;
82  const int backupGetAccelerometerData = 2;
83  const int backupGetGyroscopeData = 3;
84  const int backupSetThrusterPWM = 4;
85  const int backupEmergencyRoutine = 5;
86
87  bool backupRunEmergencyRoutine;
88  int backupLastRequestedInfo;
89  Servo backupEsc; // Need one for each Thruster
90
91  void backupSetup()
92  {
93    setStatusPin(yellowLedPin);
94    Serial.println("Backup Routine Running");
95    backupSetupCommunication(backupI2CbusAddress);
96    backupSetupSensors();
97  }
98
99  void backupLoop()
100 {
101   if(backupRunEmergencyRoutine)
102   {
103     backupEmergencyLoopRoutine();
104   }
105   else
106   {
107     backupDefaultLoopRoutine();
108   }
109 }
110
111
112 // BACKUP LOGIC
113 void backupSetupThruster()
114 {
115   backupEsc.attach(pwmThrusterPin);
116 }
117
118 void backupSetupSensors()
119 {
120   int numberAccPins = 3;
121   for(int i = 0; i < numberAccPins; i ++)
122   {
```

```
123     pinMode(accelerometerPins[i], INPUT);
124   }
125 }
126
127 void backupDefaultLoopRoutine()
128 {
129   // Backup loop routine goes here
130 }
131
132 bool backupSetRunEmergencyRoutine(int newValue)
133 {
134   if (newValue == 1)
135   {
136     backupRunEmergencyRoutine = true;
137   }
138   else
139   {
140     backupRunEmergencyRoutine = false;
141   }
142
143   backupRunEmergencyRoutine = newValue;
144   if(backupRunEmergencyRoutine){
145     Serial.println("Emergency Routine Activated");
146     backupEmergencyStateSetup();
147   }
148   else
149   {
150     Serial.println("Default Routine Activated");
151     backupReturnToDefaultStateSetup();
152   }
153 }
154
155 void backupEmergencyStateSetup()
156 {
157   backupSetupThruster();
158   backupUpdateThrusterSpeed(backupEsc, 0); // Stop Thruster
159 }
160
161 void backupEmergencyLoopRoutine()
162 {
163   // Emergency loop routine goes here
164 }
165
166 void backupReturnToDefaultStateSetup()
167 {
168   backupEsc.detach();
169   backupUpdateThrusterSpeed(backupEsc, 0); // Stop Thruster
170 }
171
172 /*
173  * Acc Sensor: MMA1270D
174  * +- 2.5g
175  * raw 1024 = 0.0025g
176  */
177 void backupReadAccelerometer(float accReading[], int numberAccPins)
178 {
179   float sens = 2.5 / 1024.0;
```

```
180   for(int i = 0; i < numberAccPins; i ++){
181     int rawValue = analogRead(accelerometerPins[i]);
182     accReading[i] = rawValue*sens;
183   }
184 }
185
186 void backupUpdateThrusterSpeed(Servo thrusterEsc, int speedPercentage
        )
187 {
188   // map(value, fromLow, fromHigh, toLow, toHigh)
189   int mappedValue = map(speedPercentage, 0, 100, 1200, 1450); //
        limits set after bench tests
190   thrusterEsc.writeMicroseconds(mappedValue);
191 }
192
193
194 // COMMUNICATION
195 void backupSetupCommunication(int myAddress)
196 {
197   Wire.begin(myAddress);
198   Wire.onReceive(backupReceiveEvent);
199   Wire.onRequest(backupRequestEvent);
200 }
201
202 void backupRequestEvent()
203 {
204   Serial.println("Request event running ");
205   switch (backupLastRequestedInfo)
206   {
207     case backupGetGyroscopeData:
208     {
209       // No Gyroscope Installed
210     }
211     case backupGetAccelerometerData:
212     {
213       int accNumberReadings = 3;
214       float accData[accNumberReadings];
215       backupReadAccelerometer(accData, accNumberReadings);
216       Serial.print("Accelerometer data: ");
217       for(int i = 0; i < accNumberReadings; i++)
218       {
219         Serial.print(accData[i]);
220         Serial.print(";  ");
221       }
222       Serial.println();
223       // send string as a series of bytes
224       Wire.write((byte*) &accData, accNumberReadings*sizeof(float));
225     }
226   }
227 }
228
229 void backupReceiveEvent(int numBytes) {
230   while (Wire.available()) {
231     Serial.print("Num bytes: ");
232     Serial.println(numBytes);
233     int receivedMsg = Wire.read(); // receive byte as a int
234     Wire.read(); // discard extra byte
```

```
235     Serial.print("Received message: ");
236     Serial.println(receivedMsg);
237     switch (receivedMsg)
238     {
239       case backupPing:
240         Serial.println("Ping received");
241         // do nothing
242         break;
243       case backupGetAccelerometerData:
244       {
245         Serial.println("Reading Acc");
246         backupLastRequestedInfo = backupGetAccelerometerData;
247         break;
248       }
249       case backupSetThrusterPWM:
250       {
251         int newPwmValue = Wire.read();
252         Serial.print("Setting new PWM percentage: ");
253         Serial.println(newPwmValue);
254         backupUpdateThrusterSpeed(backupEsc, newPwmValue);
255         break;
256       }
257       case backupEmergencyRoutine:
258       {
259         int setEmergencyRoutine = Wire.read();
260         Serial.print("Setting emergency mode: ");
261         Serial.println(setEmergencyRoutine);
262         backupSetRunEmergencyRoutine(setEmergencyRoutine);
263         break;
264       }
265     }
266   }
267 }
268
269
270 /*
271  * EXPERIMENTAL SENSORS MODULE ROUTINE
272  */
273
274 const int experimentalI2CbusAddress = 4;
275
276 const int experimentalPing = 1;
277 const int experimentalGetAccelerometerData = 2;
278 const int experimentalGetGyroscopeData = 3;
279
280 int experimentalLastRequestedInfo;
281
282 void experimentalSetup()
283 {
284   setStatusPin(greenLedPin);
285   Serial.println("Experimental Routine Running");
286   experimentalSetupCommunication(experimentalI2CbusAddress); // join
        i2c bus with address #4
287   experimentalSetupSensors();
288 }
289
290 void experimentalLoop()
```

```
{
  // Experimental loop routine goes here
}


// EXPERIMENTAL LOGIC

void experimentalSetupSensors()
{
  int numberAccPins = 3;
  for(int i = 0; i < numberAccPins; i ++)
  {
    pinMode(accelerometerPins[i], INPUT);
  }
}

void experimentalReadAccelerometer(float accReading[], int
    numberAccPins)
{
  float sens = 2.5 / 1024.0;
  for(int i = 0; i < numberAccPins; i ++){
    int rawValue = analogRead(accelerometerPins[i]);
    accReading[i] = rawValue*sens;
  }
}

void experimentalSetupCommunication(int myAdress)
{
  Wire.begin(myAdress);
  Wire.onReceive(experimentalReceiveEvent);
  Wire.onRequest(experimentalRequestEvent);
}

void experimentalRequestEvent()
{
  Serial.println("Request event running ");
  switch (experimentalLastRequestedInfo)
  {
    case experimentalGetGyroscopeData:
    {
      // No Gyroscope installed
    }
    case experimentalGetAccelerometerData:
    {
      int accNumberReadings = 3;
      float accData[accNumberReadings];
      experimentalReadAccelerometer(accData, accNumberReadings);
      Serial.print("Accelerometer data: ");
      for(int i = 0; i < accNumberReadings; i++)
      {
        Serial.print(accData[i]);
        Serial.print(";  ");
      }
      Serial.println();
      // send string as a series of bytes
      Wire.write((byte*) &accData, accNumberReadings*sizeof(float));
    }
```

```
347   }
348 }
349
350 void experimentalReceiveEvent(int numBytes) {
351
352   while (Wire.available()) {
353     Serial.print("Num bytes: ");
354     Serial.println(numBytes);
355     int receivedMsg = Wire.read(); // receive byte as a int
356     Wire.read(); // discard extra byte
357     Serial.print("Received message: ");
358     Serial.println(receivedMsg);
359     switch (receivedMsg)
360     {
361       case experimentalPing:
362         Serial.println("Ping received");
363         // do nothing
364         break;
365       case experimentalGetAccelerometerData:
366       {
367         Serial.println("Reading Acc");
368         experimentalLastRequestedInfo =
      experimentalGetAccelerometerData;
369         break;
370       }
371     }
372   }
373 }
374
375
376 /*
377  * Routines for the bench tests
378  */
379 const int statusLedBlinkingInterval = 500;
380 const int runFailureCheckInterval = 200;
381 TimedAction blinkStatusLedThread = TimedAction(
      statusLedBlinkingInterval, blinkStatusLed);
382 TimedAction checkRunFailureThread = TimedAction(
      runFailureCheckInterval, checkRunFailure);
383 boolean testLedState = false;
384 int activeStatusPin;
385
386 int lastButtonState = LOW;   // the previous reading from the input
      pin
387
388 unsigned long lastDebounceTime = 0;  // the last time the output pin
      was toggled
389 unsigned long debounceDelay = 50;    // the debounce time; increase
      if the output flickers
390
391 void benchTestsPinsSetup()
392 {
393   pinMode(greenLedPin, OUTPUT);
394   pinMode(yellowLedPin, OUTPUT);
395   pinMode(redLedPin, OUTPUT);
396   pinMode(runFailureButtonPin, INPUT);
397 }
```

```
398
399 void checkRunFailure()
400 {
401   if(checkRunFailureButtonState())
402   {
403     int lastActiveStatusPin = activeStatusPin;
404     runFailureDeadlock();
405     setStatusPin(lastActiveStatusPin);
406   }
407 }
408
409 bool checkRunFailureButtonState()
410 {
411   bool runFailure = false;
412   // READ BUTTON WITH DEBOUNCE LOGIC
413   int buttonReading = digitalRead(runFailureButtonPin);
414
415   // button pressed logic
416   if(buttonReading == HIGH && lastButtonState == LOW)
417   {
418     runFailure = true;
419     delay(debounceDelay); // for debouncing
420   }
421
422   lastButtonState = buttonReading;
423   return runFailure;
424 }
425
426 void testLoopRoutine()
427 {
428   blinkStatusLedThread.check();
429   checkRunFailureThread.check();
430 }
431
432 void blinkStatusLed()
433 {
434   testLedState ? testLedState=false : testLedState=true;
435   digitalWrite(activeStatusPin, testLedState);
436 }
437
438 void setStatusPin(int ledPin)
439 {
440   activeStatusPin = ledPin;
441   digitalWrite(greenLedPin, LOW);
442   digitalWrite(yellowLedPin, LOW);
443   digitalWrite(redLedPin, LOW);
444 }
445
446 void runFailureDeadlock()
447 {
448   setStatusPin(redLedPin);
449   Serial.println("Running error deadlock");
450   Wire.end();
451   while(!checkRunFailureButtonState())// Wait for anoter button press
        , to exit state
452   {
453     blinkStatusLedThread.check();
```

```
454   }
455 }
```

C.4 MÓDULO ATUADORES

```
1  /*
2   * Fault Tolerance Thruster Module Implementation Test
3   * By: Tui Alexandre Ono Baraniuk
4  */
5
6  #include <Wire.h>
7  #include <TimedAction.h>
8  #include <Servo.h>
9
10 /*
11  * Pins variables are organized here.
12  * Any eletronic connection changes should be done with the
        surpevision of the backup routine team.
13  */
14
15 const int backupStatusInputApin = 22;
16 const int backupStatusInputBpin = 23;
17 const int accelerometerPins[] = {0, 1, 2};
18 // Led Status Pin for Bench Tests
19 const int greenLedPin = 7;
20 const int yellowLedPin = 6;
21 const int redLedPin = 5;
22
23 const int pwmThrusterPin = 10;
24
25 const int runFailureButtonPin = 25;
26
27 /*****************************************************************
       */
28
29 bool backupControlStatusOnSetup;
30
31 void setup() {
32   // Bench tests setup
33   Serial.begin(9600); // start serial for output for TESTS DEBUGING
34   benchTestsPinsSetup();
35
36   // Control setup
37   backupControlStatusOnSetup = getBackupControlStatus(); // Verify
       what routine to run
38   if(backupControlStatusOnSetup)
39   {
40     backupSetup();
41   }
42   else
43   {
44     experimentalSetup();
45   }
46 }
47
48 void loop() {
```

```
49    testLoopRoutine();
50    if(backupControlStatusOnSetup)
51    {
52      backupLoop();
53    }
54    else
55    {
56      experimentalLoop();
57    }
58  }
59
60  // FAULT TOLERANCE
61  bool getBackupControlStatus()
62  {
63    bool backupStatus = false;
64    pinMode(backupStatusInputApin, INPUT);
65    pinMode(backupStatusInputBpin, INPUT);
66    if(digitalRead(backupStatusInputApin) == HIGH || digitalRead(
        backupStatusInputBpin) == HIGH)
67    {
68      backupStatus = true;
69    }
70
71    return backupStatus;
72  }
73
74
75  /*
76   * BACKUP THRUSTER MODULE ROUTINE
77   */
78  const int backupI2CbusAddress = 9;
79  const int backupPing = 1;
80  const int backupGetAccelerometerData = 2;
81  const int backupGetGyroscopeData = 3;
82  const int backupSetThrusterPWM = 4;
83  const int backupEmergencyRoutine = 5;
84
85  bool backupRunEmergencyRoutine;
86  int backupLastRequestedInfo;
87  Servo backupEsc; // Need one for each Thruster
88
89  void backupSetup()
90  {
91    setStatusPin(yellowLedPin);
92    Serial.println("Backup Routine Running");
93    backupSetupCommunication(backupI2CbusAddress);
94    backupSetupThruster();
95    backupUpdateThrusterSpeed(backupEsc, 0);
96  }
97
98  void backupLoop()
99  {
100   if(backupRunEmergencyRoutine)
101   {
102     backupEmergencyLoopRoutine();
103   }
104   else
```

```
105   {
106     backupDefaultLoopRoutine();
107   }
108 }
109
110
111 // BACKUP LOGIC
112 void backupSetupThruster()
113 {
114   backupEsc.attach(pwmThrusterPin);
115 }
116
117 void backupSetupSensors()
118 {
119   int numberAccPins = 3;
120   for(int i = 0; i < numberAccPins; i ++)
121   {
122     pinMode(accelerometerPins[i], INPUT);
123   }
124 }
125
126 void backupDefaultLoopRoutine()
127 {
128   // Backup loop routine goes here
129 }
130
131 bool backupSetRunEmergencyRoutine(int newValue)
132 {
133   if (newValue == 1)
134   {
135     backupRunEmergencyRoutine = true;
136   }
137   else
138   {
139     backupRunEmergencyRoutine = false;
140   }
141
142   backupRunEmergencyRoutine = newValue;
143   if(backupRunEmergencyRoutine){
144     Serial.println("Emergency Routine Activated");
145   }
146   else
147   {
148     Serial.println("Default Routine Activated");
149   }
150 }
151
152 void backupEmergencyStateSetup()
153 {
154   backupSetupSensors();
155 }
156
157 void backupEmergencyLoopRoutine()
158 {
159   // Emergency loop routine goes here
160 }
161
```

```
162 void backupReturnToDefaultStateSetup()
163 {
164
165 }
166
167 /*
168  * Acc Sensor: MMA1270D
169  * +- 2.5g
170  * raw 1024 = 0.0025g
171  */
172 void backupReadAccelerometer(float accReading[], int numberAccPins)
173 {
174   float sens = 2.5 / 1024.0;
175   for(int i = 0; i < numberAccPins; i ++){
176     int rawValue = analogRead(accelerometerPins[i]);
177     accReading[i] = rawValue*sens;
178   }
179 }
180
181 void backupUpdateThrusterSpeed(Servo thrusterEsc, int speedPercentage
      )
182 {
183   // map(value, fromLow, fromHigh, toLow, toHigh)
184   int mappedValue = map(speedPercentage, 0, 100, 1200, 1450);
185   thrusterEsc.writeMicroseconds(mappedValue);
186 }
187
188
189 // COMMUNICATION
190 void backupSetupCommunication(int myAddress)
191 {
192   Wire.begin(myAddress);
193   Wire.onReceive(backupReceiveEvent);
194   Wire.onRequest(backupRequestEvent);
195 }
196
197 void backupRequestEvent()
198 {
199   Serial.println("Request event running ");
200   switch (backupLastRequestedInfo)
201   {
202     case backupGetGyroscopeData:
203     {
204       // No Gyroscope installed
205     }
206     case backupGetAccelerometerData:
207     {
208       int accNumberReadings = 3;
209       float accData[accNumberReadings];
210       backupReadAccelerometer(accData, accNumberReadings);
211       Serial.print("Accelerometer data: ");
212       for(int i = 0; i < accNumberReadings; i++)
213       {
214         Serial.print(accData[i]);
215         Serial.print(";  ");
216       }
217       Serial.println();
```

```
218       // send string as a series of bytes
219       Wire.write((byte*) &accData, accNumberReadings*sizeof(float));
220     }
221   }
222 }
223
224 void backupReceiveEvent(int numBytes) {
225   while (Wire.available()) {
226     Serial.print("Num bytes: ");
227     Serial.println(numBytes);
228     int receivedMsg = Wire.read(); // receive byte as a int
229     Wire.read(); // discard extra byte
230     Serial.print("Received message: ");
231     Serial.println(receivedMsg);
232     switch (receivedMsg)
233     {
234       case backupPing:
235         Serial.println("Ping received");
236         // do nothing
237         break;
238       case backupGetAccelerometerData:
239       {
240         Serial.println("Reading Acc");
241         backupLastRequestedInfo = backupGetAccelerometerData;
242         break;
243       }
244       case backupSetThrusterPWM:
245       {
246         int newPwmValue = Wire.read();
247         Serial.print("Setting new PWM percentage: ");
248         Serial.println(newPwmValue);
249         backupUpdateThrusterSpeed(backupEsc, newPwmValue);
250         break;
251       }
252       case backupEmergencyRoutine:
253       {
254         int setEmergencyRoutine = Wire.read();
255         Serial.print("Setting emergency mode: ");
256         Serial.println(setEmergencyRoutine);
257         backupSetRunEmergencyRoutine(setEmergencyRoutine);
258         break;
259       }
260     }
261   }
262 }
263
264
265 /*
266  * EXPERIMENTAL THRUSTER MODULE ROUTINE
267  */
268
269 const int experimentalI2CbusAddress = 5;
270 Servo experimentalEsc; // Need one for each Thruster
271
272 void experimentalSetup()
273 {
274   setStatusPin(greenLedPin);
```

```
275   Serial.println("Experimental Routine Running");
276   experimentalSetupCommunication(experimentalI2CbusAddress); // join
        i2c bus with address #4
277   experimentalSetupThrusters();
278   experimentalUpdateThrusterSpeed(experimentalEsc, 0);
279 }
280
281 void experimentalLoop()
282 {
283   // Experimental loop routine goes here
284 }
285
286
287 // EXPERIMENTAL LOGIC
288 void experimentalSetupThrusters()
289 {
290   experimentalEsc.attach(pwmThrusterPin);
291 }
292
293 void experimentalUpdateThrusterSpeed(Servo thrusterEsc, int
      speedPercentage)
294 {
295   // map(value, fromLow, fromHigh, toLow, toHigh)
296   int mappedValue = map(speedPercentage, 0, 100, 1200, 1450); //
        limits set after bench tests
297   experimentalEsc.writeMicroseconds(mappedValue);
298 }
299
300 void experimentalSetupCommunication(int myAdress)
301 {
302   Wire.begin(myAdress);
303   Wire.onReceive(experimentalReceiveEvent);
304 }
305
306 void experimentalReceiveEvent(int numBytes) {
307   const int ping = 1;
308   const int setThrusterPWM = 4;
309
310   while (Wire.available()) {
311     int receivedMsg = Wire.read(); // receive byte as a int
312     Wire.read(); // discard extra byte
313     Serial.print("Received message: ");
314     Serial.println(receivedMsg);
315     switch (receivedMsg)
316     {
317       case ping:
318         break;
319       case setThrusterPWM:
320       {
321         int newPwmValue = Wire.read();
322         Serial.print("Setting new PWM percentage: ");
323         Serial.println(newPwmValue);
324         experimentalUpdateThrusterSpeed(backupEsc, newPwmValue);
325         break;
326       }
327     }
328   }
```

```
329 }
330
331
332 /*
333  * Routines for the bench tests
334  */
335 const int statusLedBlinkingInterval = 500;
336 const int runFailureCheckInterval = 200;
337 TimedAction blinkStatusLedThread = TimedAction(
        statusLedBlinkingInterval, blinkStatusLed);
338 TimedAction checkRunFailureThread = TimedAction(
        runFailureCheckInterval, checkRunFailure);
339 boolean testLedState = false;
340 int activeStatusPin;
341
342 int lastButtonState = LOW;   // the previous reading from the input
        pin
343
344 unsigned long lastDebounceTime = 0;  // the last time the output pin
        was toggled
345 unsigned long debounceDelay = 50;    // the debounce time; increase
        if the output flickers
346
347 void benchTestsPinsSetup()
348 {
349   pinMode(greenLedPin, OUTPUT);
350   pinMode(yellowLedPin, OUTPUT);
351   pinMode(redLedPin, OUTPUT);
352   pinMode(runFailureButtonPin, INPUT);
353 }
354
355 void checkRunFailure()
356 {
357   if(checkRunFailureButtonState())
358   {
359     int lastActiveStatusPin = activeStatusPin;
360     runFailureDeadlock();
361     setStatusPin(lastActiveStatusPin);
362   }
363 }
364
365 bool checkRunFailureButtonState()
366 {
367   bool runFailure = false;
368   // READ BUTTON WITH DEBOUNCE LOGIC
369   int buttonReading = digitalRead(runFailureButtonPin);
370
371   // button pressed logic
372   if(buttonReading == HIGH && lastButtonState == LOW)
373   {
374     runFailure = true;
375     delay(debounceDelay); // for debouncing
376   }
377
378   lastButtonState = buttonReading;
379   return runFailure;
380 }
```

```
381
382 void testLoopRoutine()
383 {
384   blinkStatusLedThread.check();
385   checkRunFailureThread.check();
386 }
387
388 void blinkStatusLed()
389 {
390   testLedState ? testLedState=false : testLedState=true;
391   digitalWrite(activeStatusPin, testLedState);
392 }
393
394 void setStatusPin(int ledPin)
395 {
396   activeStatusPin = ledPin;
397   digitalWrite(greenLedPin, LOW);
398   digitalWrite(yellowLedPin, LOW);
399   digitalWrite(redLedPin, LOW);
400 }
401
402 void runFailureDeadlock()
403 {
404   setStatusPin(redLedPin);
405   Serial.println("Running error deadlock");
406   Wire.end();
407   while(!checkRunFailureButtonState())// Wait for anoter button press
        , to exit state
408   {
409     blinkStatusLedThread.check();
410   }
411 }
```

C.5 MÓDULO DE EMERGÊNCIA

```
1  /*
2   * Emergency Board Routine Test - Bench Test
3   * By: Tui Alexandre Ono Baraniuk
4  */
5
6  #include <TimedAction.h>
7
8  // Function Declaration
9  void blinkEmergencyLed();
10 void playBuzzer();
11 void blinkStatusLed();
12 void updateBackupBoardStatus();
13
14 // Connected Pins
15 const int buzzerPin = 5;
16 const int backupBoardStatusPinA = 2;
17 const int backupBoardStatusPinB = 3;
18 const int ledEmergencyPin = 7;
19 const int ledGreenPin = 9;
20
21 // Atuators parameters
```

```
22 const int buzzerFrequencyHz = 1000;
23 const int buzzerDelayMs = 1000;
24 const int buzzerDurationMs = 500;
25
26 const int ledDelayMs = 1000;
27
28 // Emergency parameters
29 const int waitBackupBoardInitializationDelay = 30000;
30 bool emergencyModeRunning = false;
31
32 // Bench tests
33 int statusLedBlinkingInterval = 800;
34
35 TimedAction blinkLedThread = TimedAction(ledDelayMs,
       blinkEmergencyLed);
36 TimedAction playBuzzerThread = TimedAction(buzzerDelayMs, playBuzzer)
       ;
37 TimedAction blinkStatusLedThread = TimedAction(
       statusLedBlinkingInterval, blinkStatusLed);
38
39 void setup() {
40   Serial.begin(9600); // start serial for output for TESTS DEBUGING
41
42   pinMode(backupBoardStatusPinA, INPUT);
43   pinMode(backupBoardStatusPinB, INPUT);
44   pinMode(ledGreenPin, OUTPUT);
45   pinMode(ledEmergencyPin, OUTPUT);
46
47   Serial.println("Emergency board running");
48   Serial.println("Waiting for the backup board to initialize");
49   delay(waitBackupBoardInitializationDelay);
50   Serial.println("Monitoring backup board");
51 }
52
53 void loop() {
54   monitorBackupBoard();
55 }
56
57
58 // Checks if it needs to turn emergency mode
59 void monitorBackupBoard()
60 {
61   bool emergencyMode = !checkBackupBoardStatus();
62   if(emergencyMode)
63   {
64     // activates buzzer and blinks emergency led
65     blinkLedThread.check();
66     playBuzzerThread.check();
67     emergencyModeRunning = true;
68   }
69   else if (emergencyModeRunning)
70   {
71     exitEmergencyMode();
72     emergencyModeRunning = false;
73   }
74
75   blinkStatusLedThread.check();
```

```
76 }
77
78 // Turn off buzzer and emergency LED
79 void exitEmergencyMode()
80 {
81   digitalWrite(ledEmergencyPin, LOW);
82   noTone(buzzerPin);
83   Serial.println("Exiting Emergency Mode");
84 }
85
86 bool checkBackupBoardStatus()
87 {
88   bool boardStatus = true;
89   if(pulseIn(backupBoardStatusPinA, LOW) == 0 && pulseIn(
       backupBoardStatusPinB, LOW) == 0  )
90   {
91     Serial.println("Emergency Mode Detected");
92     boardStatus = false;
93   }
94   return boardStatus;
95 }
96
97
98 void playBuzzer()
99 {
100   static bool buzzerLastState;
101   static unsigned long buzzerPreviousMillis;
102   const long interval = buzzerDelayMs;
103   unsigned long currentMillis = millis();
104
105   if (currentMillis - buzzerPreviousMillis >= interval) {
106     if(buzzerLastState)
107     {
108       tone(buzzerPin, buzzerFrequencyHz);// play frequency in
      specified pin
109     }
110     else
111     {
112       noTone(buzzerPin);
113     }
114     buzzerLastState = !buzzerLastState;
115     buzzerPreviousMillis = currentMillis;
116   }
117 }
118
119
120 void blinkEmergencyLed()
121 {
122   static bool ledLastState = false;
123   static unsigned long ledPreviousMillis = 0;
124   const long interval = ledDelayMs;
125   unsigned long currentMillis = millis();
126
127   if (currentMillis - ledPreviousMillis >= interval) {
128     if(ledLastState)
129     {
130       digitalWrite(ledEmergencyPin, HIGH);
```

```
131     }
132     else
133     {
134       digitalWrite(ledEmergencyPin, LOW);
135     }
136     ledLastState = !ledLastState;
137     ledPreviousMillis = currentMillis;
138   }
139 }
140 
141 
142 void blinkStatusLed()
143 {
144   static bool ledState;
145   if(ledState || emergencyModeRunning)
146   {
147     ledState = false;
148   }
149   else
150   {
151     ledState=true;
152   }
153   digitalWrite(ledGreenPin, ledState);
154 }
```