Universidade Federal da Bahia
Instituto de Matemática/Escola Politécnica
Departamentos de Engenharia Mecânica & Ciência da Computação

Pós-graduação em Mecatrônica

DETECTORES ADAPTATIVOS DE DEFEITOS PARA SISTEMAS DE CONTROLE DE TEMPO REAL CRÍTICOS

Alirio Santos de Sá

DISSERTAÇÃO DE MESTRADO

Salvador
6 de outubro de 2006

Universidade Federal da Bahia
Instituto de Matemática/Escola Politécnica
Departamentos de Engenharia Mecânica & Ciência da Computação

Alirio Santos de Sá

DETECTORES ADAPTATIVOS DE DEFEITOS PARA SISTEMAS DE CONTROLE DE TEMPO REAL CRÍTICOS

Trabalho apresentado ao Programa de Pós-graduação em Mecatrônica dos Departamentos de Engenharia Mecânica & Ciência da Computação da Universidade Federal da Bahia como requisito parcial para obtenção do grau de Mestre em Mecatrônica.

Orientador: *Prof. Dr. Raimundo José de Araújo Macêdo*

Salvador
6 de outubro de 2006

Ficha catalográfica elaborada pela Biblioteca Bernadete Sinay Neves,
Escola Politécnica da UFBA

Sá, Alirio Santos de
S111d Detectores adaptativos de defeitos para sistemas de controle de tempo real críticos / Alirio Santos de Sá. – Salvador, 2006.

—p. : il.

Orientador: Prof. Dr. Raimundo José de Araújo Macêdo.
Dissertação (mestrado) – Universidade Federal da Bahia, Escola Politécnica, 2006.

1. Sistemas distribuídos. 2. Sistemas de tempo real. 3. Teoria de controle. I. Macedo, Raimundo José de Araújo. II. Universidade Federal da Bahia. Escola Politécnica. III Título.

CDD 20.ed. 003.83

TERMO DE APROVAÇÃO

Título da Dissertação: *Detectores Adaptativos de Defeitos para Sistemas de Controle de Tempo Real Críticos*

Estudante: Alirio Santos de Sá

Dissertação aprovada como requisito parcial para obtenção do grau de Mestre em Mecatrônica, Universidade Federal da Bahia, pela seguinte banca examinadora:

Raimundo José de Araújo Macêdo - Orientador
Ph.D University of Newcastle Upon Tyne, Inglaterra
Professor Titular do Departamento de Ciência da Computação da Universidade Federal da Bahia

---

Rômulo Silva de Oliveira
Doutor em Engenharia Elétrica da Universidade de Santa Catarina
Professor Adjunto do Instituto de Informática da Universidade Federal de Santa Catarina

---

Taisy Silva Weber
Pos-Doctor in Informatics at the University of Karlsruhe, Alemanha
Professor Associado do Instituto de Informática da Universidade Federal do Rio Grande do Sul

---

George Marconi Lima
Ph.D., University of York, Inglaterra
Professor Adjunto do Departamento de Ciência da Computação da Universidade Federal da Bahia

06 de outubro de 2006

Aos
meus pais, Antonio e Neuza, por terem me encorajado em meus primeiros passos e me apoiado nos momentos difícies. À minha esposa, Margarete, e minha filha, Maísa, por terem tolerado a minha ausência, mudanças de humor e por reaviavarem minha esperança e coragem.

AGRADECIMENTOS

Ao Professor Raimundo José de Araújo Mâcedo, pelo excelente, incançável e compreensível acompanhamento e orientação do trabalho realizado nesta dissertação.

Aos Professores Marcelo Embiruçu e Leizer Schmitman, pela ajuda durante a fase inical sobre os conceitos pertinentes à teoria de controle.

À Fundação de Amparo a Pesquisa do Estado da Bahia (FAPESB) pelo apóio financeiro.

Aos amigos Fred, Geovani, Leopoldo, Neima, Sandro e Rodrigo pelas frutificantes discussões e troca de ídeias.

A todos aqueles que direta ou inderetamente colaboraram para o desenvolvimento deste trabalho.

A cada salto que damos na tentativa de encostar os dedos no céu, nos aproximamos e nos distanciamos daquilo que por convenção denominamos "A Verdade". E por mais estranho que possa parecer, o processo científico não está nas pontas dos dedos das mãos, e sim nas pontas dos dedos dos pés, quando estes se afastam e retornam ao chão, denotando um fluxo enfadonho, ardúo e belo entre "Teoria" e "Realidade".

—ALIRIO S. DE SÁ (2006)

RESUMO

Aplicações de controle de processos são exemplos típicos de sistemas de tempo real, uma vez que necessitam operar de forma correta atendendo aos prazos ditados pela dinâmica da planta que está sendo controlada. A indústria moderna de controle e automação de processos tem tirado proveito do avanço das redes e arquiteturas de computadores para promover soluções distribuídas que permitam uma maior integração entre as plantas e que sejam mais flexíveis, interoperáveis e produtivas. Além disso, tais soluções devem permitir a automação de plantas cada vez mais complexas, com menores custos operacionais. O projeto dessas aplicações de tempo real sobre rede, ou distribuídas, deve levar em consideração questões pertinentes ao algoritmo de controle, ao atendimento dos requisitos temporais e à comunicação sobre rede. Algumas aplicações de controle sobre rede, ditas de *missão crítica*, necessitam de mecanismos que garantam o funcionamento contínuo mesmo na presença de falhas. Mecanismos de tolerância a falhas são fundamentais para incrementar a confiabilidade de sistemas críticos. Todavia, a implementação desses mecanismos pode tornar o projeto ainda mais complexo e comprometer o desempenho do sistema de controle. Detectores de defeitos são blocos básicos na construção de mecanismos de tolerância a falhas: seja para ativar mecanismos de recuperação, seja para permitir a reconfiguração do sistema após a ocorrência de uma falha. Dotar esses detectores da capacidade de adaptação é importante para permitir um serviço de detecção de defeitos mais rápido e confiável. O casamento do grau de adaptabilidade dos detectores com a qualidade do desempenho das aplicações críticas de controle é crucial para garantir o atendimento dos requisitos funcionais e temporais de tais aplicações. Nesse contexto, esta dissertação traz uma proposta de detecção de defeitos adaptável baseada em redes neurais e contrapõe tal proposta com as principais abordagens de detecção adaptativa existentes na literatura. De outro lado, a dissertação avalia o impacto da abordagem proposta sobre o desempenho do sistema de controle, indicando os limites que devem ser respeitados pelo detector de defeitos de modo a não comprometer a estabilidade do controle em questão.

Palavras-chave: Detecção de Defeitos, Tolerância a Falhas, Sistemas de Tempo Real Críticos, Sistemas Distribuídos, Sistemas de Controle sobre Rede.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

ALGORITMOS

CAPÍTULO 1

INTRODUÇÃO

Sistemas de Controle baseado em Computador – aqui referenciados simplesmente por Sistemas de Controle – são sistemas que, em instantes discretos de tempo, monitoram e atuam sobre uma ou mais variáveis que compõem uma planta, fazendo com que as variáveis de tal planta se comportem da maneira esperada (OGATA, 1995). De forma simplificada, uma planta pode ser traduzida por qualquer objeto físico, ou conjunto de objetos físicos, que contenha elementos a serem controlados (OGATA, 1990), como um reator químico, um avião, um automóvel, um forno, um robô etc. Em geral, quando se faz referência à planta, se está referenciando o conjunto de variáveis que podem ser manipuladas e/ou observadas. Sem perda de generalidade, um sistema de controle baseado em computador pode ser entendido por um sistema de controle discreto (OGATA, 1995) cujo algoritmo de controle executa em um dispositivo com potencial de processamento.

Os sistemas de controle são exemplos típicos de sistemas de tempo real (FARINES; FRAGA; OLIVEIRA, 2000; KOPETZ, 1997), uma vez que devem atender a prazos que são ditados pela própria dinâmica do objeto físico a ser controlado. Em geral, esses sistemas são formados por elementos sensores, que percebem o atual estado das variáveis a serem controladas; elementos controladores, responsáveis por, a partir do estado percebido, fornecer a ação de controle requerida para manter o estado das variáveis manipuladas dentro do desejado; e elementos atuadores responsáveis por capturar ação de controle e promover a efetiva atuação sobre as variáveis que estão sendo controladas.

Além de possibilitar a execução de diversos algoritmos diferentes e, conseqüentemente,

o controle de diversas plantas usando uma mesma unidade de processamento, a implementação baseada em computador permite que algoritmos de controle mais eficientes e adaptativos possam ser construídos.

O avanço das redes de computadores tem dado subsídios para que a indústria de controle e automação de processos implementem soluções de controle distribuído utilizando redes de comunicação. Tais soluções, entre outras coisas, facilitam a interoperabilidade entre componentes de fabricantes distintos; diminuem a complexidade da interconexão entre componentes dos sistemas; permitem um monitoramento e supervisão remotos mais eficientes; e diminuem os custos operacionais (LIAN; MOYNE; TILBURY, 2001).

1.1 MOTIVAÇÃO

Apesar de todos os benefícios apontados, novos desafios são encontrados quando se realiza a implementação do sistema de controle de tempo real sobre rede: o sistema está sujeito a incertezas relacionadas às variações no tempo de computação do algoritmo de controle (atraso de computação) e envio e entrega das mensagens pelo subsistema de comunicação (atraso de comunicação).

Em um sistema de controle baseado em computador, o algoritmo de controle é implementado como uma tarefa de tempo real, estando, assim, sujeito às variações e preempções provocadas pela execução das demais tarefas sob a coordenação do sistema operacional de tempo real(CERVIN et al., 2004), o que impacta diretamente no tempo necessário para a computação da ação de controle.

O atraso na comunicação, por sua vez, está relacionado não só à largura de banda, mas também ao tamanho das mensagens, o tempo de acesso ao meio, o número de colisões, a probabilidade de perdas, entre outros(LIAN; MOYNE; TILBURY, 2001; OTANEZ et al., 2002; TANENBAUM, 2003).

Os atrasos de computação e comunicação promovem variações entre a aquisição do

estado da planta e realização da ação de controle, podendo, dessa forma, influenciar no desempenho do controle realizado e tornar o projeto do sistema mais complexo (LIAN; MOYNE; TILBURY, 2002; YOOK; TILBURY; SOPARKART, 2000).

Algumas aplicações de controle distribuído, ditas *de missão crítica*, precisam de mecanismos de tolerância a falhas que possibilitem o funcionamento contínuo, produzindo resultados corretos e atendendo aos limites temporais estabelecidos. Todavia, os algoritmos relacionados à implementação dos mecanismos de tolerância a falhas em sistemas distribuídos necessitam de um processamento extra e de uma troca adicional de mensagens para cumprirem seus objetivos com segurança (LYNCH, 1996). Esse processamento e troca adicional de mensagens torna o projeto do sistema ainda mais difícil, visto que o aumento do tráfego e do processamento podem implicar, em alguns casos, em atrasos não determinísticos quando se utiliza dispositivos e redes de comunicação convencionais.

Para a implementação de mecanismos de tolerância a falhas, um serviço de detecção de defeitos[1] é fundamental, seja para ativar procedimentos de recuperação, seja para permitir a reconfiguração do sistema (JALOTE, 1994). Para aplicações críticas de controle distribuídos sobre redes convencionais, é essencial promover soluções de detecção de defeitos adaptáveis e os algoritmos de adaptação utilizados devem contornar ou minimizar os possíveis efeitos dos atrasos (não determinísticos) impostos pelo processamento e pela rede de comunicação.

Em ambiente de tráfego ou tempo de computação variados, os detectores adaptativos de defeitos podem produzir informações mais precisas sobre o estado dos dispositivos do sistema, evitando, assim, que os mecanismos de tolerância a falhas tomem decisões que venham a prejudicar o desempenho do sistema de controle. O caráter adaptativo

[1]Existe uma distinção na literatura entre os conceitos de defeitos, erros e falhas. Os mecanismos de tolerância a falhas devem tolerar falhas em componentes evitando que haja um desvio no serviço a ser prestado pelo sistema (defeito). O erro é definido como um estado, oriundo de uma falha, que quando ativado pode levar o sistema a apresentar um defeito. Em Laprie (1985) existe uma discussão detalhada sobre estas terminologias.

de tais detectores se dá de forma estática[2], através de mapeamento matemático das características do tráfego no subsistema de comunicação e da velocidade dos dispositivos, ou de forma dinâmica[3], reconhecendo as mudanças e sugerindo novos modelos a cada mudança *brusca* em tal comportamento.

Informações errôneas produzidas pelos detectores de defeitos podem, por exemplo, iniciar procedimentos de recuperação ou de reconfiguração, os quais consomem recursos do sistema (SAMPAIO et al., 2003), aumentando o tempo para atuação do controle. Variações no instante de atuação podem implicar em perda de desempenho ou até mesmo em instabilidade (BUTTAZZO et al., 2004).

1.2 OBJETIVO E ESTRUTURA DA DISSERTAÇÃO

A presente dissertação se propõe a avaliar o impacto da implementação de mecanismos de tolerância a falhas sobre o desempenho de sistemas de controle de *missão crítica* sobre redes de comunicação, focando, para tanto, no impacto causado pela implementação do serviço de detecção de defeitos.

Sendo assim, no **capítulo 2** abordam-se questões básicas sobre sistemas de controle, apresentam-se pontos relevantes a respeito da implementação de sistemas de controle de tempo real sobre rede, discute-se como variações no tempo de computação e comunicação podem influenciar o projeto e o desempenho de tais sistemas e, por fim, verifica-se como a estabilidade e o desempenho dessa categoria de sistema pode ser avaliada.

No **capítulo 3**, por sua vez, apresentam-se conceitos pertinentes aos modelos de sistemas distribuídos e aos mecanismos de tolerância a falhas. Em seguida, abordam-se pontos pertinentes à implementação de detectores adaptativos para ambientes distribuídos e como métricas de qualidade de serviço podem ser utilizadas para avaliar o serviço de detecção prestado por tais abordagens.

[2]e.g. modelos de Chen, Toueg e Aguilera (2002) e Macêdo e Lima (2004)
[3]e.g. modelo de Bertier, Marin e Sens (2003) e Jacobson (1988)

No **capítulo 4**, apresentam-se algumas discussões sobre redes neurais artificiais, focando, entretanto, no modelo de rede utilizado nesta dissertação.

No **capítulo 5**, apresentam-se as avaliações e as considerações pertinentes às simulações realizadas nesta dissertação. Para tais simulações, realizou-se a implementação, em ambiente simulado, de diferentes abordagens de detecção de defeitos e analisou-se o impacto de cada uma das abordagens sobre o desempenho de sistemas de controle de tempo-real implementados sobre redes *CAN*, *Shared-Ethernet* e *Switched-Ethernet*. Utilizou-se, para tanto, métricas para a verificação de desempenho de sistemas de controle existentes na literatura. Além disso, averigüou-se que nível de qualidade de serviço é oferecido pelas diferentes abordagens de detecção, comparando-as entre si através de métricas atualmente em uso na literatura. A fim de se obter um mecanismo de detecção mais confiável em cenários com variações nos tempos de computação e comunicação, implementou-se e avaliou-se uma abordagem de detecção adaptativa baseada em redes neurais artificiais. Por fim, nesse capítulo discute-se como, em um cenário de falhas, o desempenho do sistema de controle é garantido.

Por fim, no **capítulo 6**, são feitas as conclusões acerca do trabalho realizado e apontados possíveis trabalhos futuros a serem desenvolvidos.

Detalhes do ambiente de simulação utilizado e dos principais algoritmos implementados são apresentados no **apêndice A**.

1.3 CONTRIBUIÇÕES

O trabalho realizado nesta dissertação traz algumas contribuições, que servem como ponto de partida para o projeto de sistemas de controle de missão crítica sobre rede. Essas contribuições são apresentadas nas subseções a seguir.

1.3.1 Detecção Adaptativa Baseada em Redes Neurais Artificiais

A primeira contribuição trazida nesta dissertação é a proposta de um mecanismo adaptativo de detecção baseado em Redes Neurais Artificiais aplicável a um Sistema de Controle sobre Rede de Comunicação. Em particular, os seguintes aspectos foram avaliados:

- Apresenta uma metodologia para a construção de um detector baseado em *Redes Neurais Artificiais*.
- Analisa o desempenho da abordagem quando contraposta a abordagens convencionais existentes na literatura. Para tanto, utiliza métricas de qualidade de serviço para detecção de defeitos definidas por Chen, Toueg e Aguilera (2002).
- Avalia o impacto no desempenho de um ambiente com sistema de controle simples e em um ambiente no qual a rede é compartilhada com múltiplos sistemas de controle.
- Avalia o desempenho do detector de defeitos em diferentes redes de comunicação.

1.3.2 Avaliação da Estabilidade e Desempenho do Sistema de Controle sob Diferentes Parâmetros de Detecção

A segunda contribuição desta dissertação aponta como o conceito de *Margem de Jitter*, de Cervin et al. (2004), pode ser utilizado para avaliar o impacto da implementação de detectores de defeitos adaptativos na construção de sistemas de controle de *missão crítica*. Tal abordagem é utilizada para verificar a estabilidade do sistema sob diferentes condições de tráfego na rede e de processamento nos dispositivos. Além disso, verifica-se, utilizando métricas convencionais, como o desempenho do controle é afetado pela carga das tarefas de emissão de mensagens de *heartbeats* do detector de defeitos. Assim:

- É demonstrado como os índices de desempenho do sistema de controle são afetados através da mudança do período de emissão de *heartbeats*.

- Avalia-se o desempenho e a estabilidade do sistema de controle usando diferentes redes de comunicação e diferentes períodos de emissão de *heartbeats*.

1.3.3 Outras Contribuições

Além das contribuições apresentadas, as implementações e análises realizadas implicaram em contribuições mais específicas, listadas abaixo:

- Discute como atrasos de computação inerentes ao escalonamento das tarefas afetam o desempenho dos detectores de defeitos, podendo provocar falsas estimativas.

- Avalia as restrições impostas pelo sistema de controle no tempo para a detecção de defeitos em componentes.

- Apresenta um *framework* para simulação de sistemas de controle de missão crítica sobre rede.

1.3.4 Publicações Relacionadas a Esta Dissertação

Alguns dos resultados obtidos foram discutidos em artigos publicados em congressos científicos de modo a fortalecer o estudo acerca do tema. Tais publicações são apresentadas a seguir:

(a) De Sá, Alirio, Macêdo, Raimundo J. de A. *Avaliando o Impacto de Detectores na Estabilidade de Controle de Tempo-Real sobre Redes Convencionais*, **VII Workshop Brasileiro de Testes e Tolerância a Falhas, SBRC**, maio 2006, Curitiba, PR, Brasil.

- Nesse artigo utiliza-se o conceito de *Margem de Jitter*, de Cervin et al. (2004), para verificar condições suficientes para que a implementação do detector de defeitos não leve o sistema de controle à instabilidade. Além disso, verifica-se a qualidade do controle utilizando, para tanto, índices de desempenho convencionais como o IAE (*Integral Absolute Error*) e ISE (*Integral Square Error*)(OGATA, 1990). Avalia-se também o desempenho dos detectores de defeitos sob um ambiente com múltiplos sistemas de controle, no qual os dispositivos de cada sistema (controladores, atuador e sensor) se comunicam através de uma rede de comunicação *shared bus Ethernet*.

(b) De Sá, Alirio, Macêdo, Raimundo J. de A. *Avaliando Detectores de Defeitos em Sistemas de Controle Distribuídos de Tempo Real sobre Redes Convencionais*, **Proc. In Third Workshop On Theses And Dissertations III, WTD-ladc2005**, pp. 61-66, out. 2005, Salvador, Ba, Brazil.

- O artigo verifica o desempenho da abordagem de detecção baseado em RNA, aplicada a um ambiente com múltiplos sistemas de controle, nos quais sensores, atuadores e controladores se comunicam através de uma rede de comunicação *shared bus Ethernet*. As simulações são conduzidas observando o desempenho, em termo das métricas de qualidade de serviço para detecção, sugeridas por Chen, Toueg e Aguilera (2002), e a abordagem proposta é comparada a abordagens convencionais existentes na literatura. Além disso, é observado o impacto da implementação do detector de defeitos sobre a atuação dos sistemas de controle. Para esse propósito, calcula-se o atraso médio inserido na atuação do controle com e sem a atuação do bloco detector de defeitos no sistema.

(c) De Sá, Alirio, Macêdo, Raimundo J. de A. *An Adaptive Failure Detection Approach for Real-Time Distributed Control Systems over Shared Ethernet*, **XVIII Inter-**

national Congress of Mechanical Engineering, COBEM 2005, nov. 2005, Ouro Preto, MG, Brazil.

- O artigo verifica o desempenho da abordagem de detecção baseado em *RNA*, aplicada a um ambiente com um único sistema de controle, onde sensor, atuador e controladores, se comunicam através de uma rede *shared Ethernet*. As simulações são conduzidas observando o desempenho, em termo das métricas de qualidade de serviço para detecção, sugeridas por Chen, Toueg e Aguilera (2002), e a abordagem proposta é comparada a outras abordagens existentes na literatura. Além disso, é analisado o impacto da implementação do detector de defeitos sobre a atuação dos sistemas de controle. Para esse propósito, calcula-se o atraso médio inserido na atuação do controle com e sem a atuação do bloco detector de defeitos no sistema.

(d) De Sá, Alirio, Macêdo, Raimundo J. de A. *Detectores de Defeitos Adaptáveis para Sistemas de Controle Distribuídos de Tempo Real sobre Redes Ethernet*, **VII Brazilian Workshop of Real-Time Systems, sbrc 2005**, may 2005, Fortaleza, CE, Brazil (Short Paper).

- Versão resumida do artigo (c); não são apresentados os detalhes da implementação da rede neural.

(e) Andrade, Sandro S.; Macêdo, Raimundo J. De A.; De Sá, Alirio S.. *ARCOS: A Component-based Architecture for the Construction of Robust Control and Supervision Applications*, **Proc. In First Latin-american Workshop On Dependable Automation Systems, WDAS-LADC2005**, pp. 10-16, out. 2005, Salvador, Ba, Brazil.

- Nessa publicação, discute-se a implementação de um detector de defeitos adaptativo baseado em componentes. Para tanto, utiliza-se uma plataforma para

implementação de aplicações de controle e supervisão, proposta por Andrade e Macêdo (2005), denominada *ARCOS*.

(f) Macedo, R., Lima, G., Barreto, L., Andrade, A., De Sá, Alirio, Barboza, F., Albuquerque, R., Andrade, S. *Tratando a Previsibilidade em Sistemas de Tempo real Distribuídos: Especificação, Linguagens, Middleware, Mecanismos Básicos.* **Capítulo 3 do Livro Texto do Mini-curso Realizado no 22º Simpósio Brasileiro de Redes de Computadores, SBRC2004**, pp. 105-163, ISBN:85-88442-82-5, 10 À 14 De Maio De 2004, Gramado, RS.

- Essa publicação constitui um trabalho colaborativo que aborda o Tratamento da Previsibilidade na Implementação de Sistemas de Tempo Real Distribuídos. O trabalho desenvolvido durante a dissertação contribui com uma discussão sobre técnicas de Tolerâncias a Falhas para Sistemas de Tempo Real Críticos.

1.4 TRABALHOS CORRELATOS

Em Lian, Moyne e Tilbury (2001) avalia-se o desempenho de diferentes tipos de redes de comunicação usadas em sistemas de controle distribuídos. Nesse trabalho, discute-se como redes *Ethernet*, *ControlNet* e *DeviceNet* podem satisfazer a requisitos de tempo de resposta e garantir a qualidade e a confiabilidade da comunicação entre os dispositivos de um sistema de controle. Uma vez que as características temporais da rede influenciam diretamente as aplicações de controle, para cada protocolo avaliado, os autores verificaram, em diversos cenários, como tais características temporais são afetadas pela velocidade da rede, pela prioridade e tamanho das mensagens, pela política de acesso ao meio de comunicação, pela quantidade de bits de controle adicionados a cada mensagem etc.

Lian, Moyne e Tilbury (2002) discutem a importância da escolha da *taxa de amostragem*, evidenciando a relação de compromisso existente entre o desempenho do sistema de

controle e o número de mensagens que trafegam na rede. Pequenas *taxas de amostragem* são interessantes para garantir um desempenho do controle dentro do limite aceitável. Entretanto, *taxas de amostragem* muito pequenas podem incrementar em demasia o número de mensagens que trafegam na rede e, dessa forma, implicar em um aumento considerável no tempo de acesso ao meio de comunicação. A conseqüência disso é um maior atraso na comunicação *fim-a-fim* entre os dispositivos do sistema e uma degradação na qualidade do desempenho do controle.

Nos trabalhos de Lian, Moyne e Tilbury (2002) e Lian, Moyne e Tilbury (2001), entretanto, não é discutida a implementação de mecanismos de tolerância a falhas e como esses mecanismos podem ser usados de forma eficiente mesmo na presença de redes de comunicação não determinísticas. O impacto no desempenho do controle quando se necessita da implementação de mecanismos de tolerância a falhas não é analisado.

Conforme mencionado anteriormente, para sistemas de controle sobre redes de *missão crítica*, além das garantias para o desempenho do controle, o projeto deve focar a implementação de mecanismos que garantam a *confiança no funcionamento* (*Dependability*) (LAPRIE, 1985). Em Macêdo et al. (2004) é encontrada uma discussão geral sobre os mecanismos básicos e técnicas principais para que sistemas de tempo real críticos (*safety-critical systems*) tenham um comportamento previsível. Entretanto, a utilização de redes de controle não é abordada e a implementação de mecanismos adaptativos de detecção de defeitos não é discutida.

O detector de defeitos tem sido usado como um bloco básico para solucionar diversos problemas em sistemas distribuídos tolerantes a falhas. Em especial, citam-se os detectores de defeitos não confiáveis de Chandra e Toueg (1996) propostos como uma alternativa à impossibilidade, verificada por Fischer, Lynch e Paterson (1985), em solucionar o problema do Consenso (GUERRAOUI; SCHIPER, 1997) em sistemas distribuídos assíncronos[4]

[4]Em um sistema assíncrono não existem limites temporais para o atraso na transmissão das mensagens nem para o processamento realizado em cada componente. Uma discussão mais detalhada sobre modelos de sistemas pode ser encontrada em (LYNCH, 1996), (GÄRTNER, 1999) e (VERÍSSIMO; RODRIGUES, 2000)

em que um componente do sistema pode falhar por omissão infinita(*crash fail*).

Os detectores não confiáveis encapsulam a sincronia necessária para a resolução dos problemas em sistemas assíncronos e funcionam como oráculos, provendo informações sobre os processos do sistema. Esses detectores são ditos não confiáveis uma vez que podem suspeitar de componentes que efetivamente nunca falharam (ditos componentes corretos) ou ainda não suspeitar de componentes falhos. Chandra e Toueg (1996) propuseram oito classes para tais detectores de defeitos, as quais variam de acordo com duas propriedades básicas: *Completude* (*Completeness*), referindo-se à capacidade dos detectores de defeitos de suspeitar, em algum momento no futuro, de componentes que efetivamente falharam; e *Precisão* (*Accuracy*), relacionada à capacidade do detector em evitar falsas suspeitas.

A classificação dos detectores de defeitos de Chandra e Toueg (1996) é baseada em comportamentos temporais futuros não quantificáveis, sendo, portanto, não apropriada para sistemas que necessitem de restrições temporais mais fortes (sistemas síncronos ou sistemas de tempo real). Para tais sistemas é necessário que as características dos detectores de defeitos sejam quantificáveis de modo a assegurar o atendimento dos requisitos temporais das aplicações.

Nesse contexto, Chen, Toueg e Aguilera (2002) propuseram métricas probabilísticas de Qualidade de Serviço (*QoS, Quality of Service*) para detecção de defeitos que avaliam o quão rápido as falhas são detectadas (velocidade) e quão capaz o detector é de evitar falsas suspeitas (precisão).

Alguns trabalhos como Chen, Toueg e Aguilera (2002), Bertier, Marin e Sens (2002), Bertier, Marin e Sens (2003), Macêdo e Lima (2004) e Nunes e Jansch-Pôrto (2004) propõem e avaliam, em sistemas assíncronos, o desempenho de abordagens de detectores adaptativos de defeitos baseados nas métricas de *QoS* propostas por (CHEN; TOUEG; AGUILERA, 2002). Todavia, nesses trabalhos não se discute a implementação dessas abordagens para um sistema de controle de tempo real.

Apesar das abordagens de detecção de defeitos citadas acima estarem diretamente

ligadas a sistemas assíncronos (e parcialmente síncronos), Hermant e Lann (2002) demonstraram que tais abordagens podem ser utilizadas para projetar sistemas distribuídos de tempo real críticos. No trabalho, os autores discutem como, através de um procedimento denominado *Ligação Tardia* (*Late Binding*), pode-se utilizar algoritmos adotados na solução de problemas em sistemas assíncronos para maximizar a capacidade (cobertura) do sistema de tempo real em: percorrer apenas estados corretos (propriedade *safety*), poder alcançar todos os estados em algum momento no futuro (propriedade *liveness*) e progredir atendendo aos prazos preestabelecidos (propriedade *timeliness*)[5].

[5]As propriedades de safety, liveness e timeliness caracterizam a corretude dos sistemas distribuídos. Maiores detalhes sobre tais propriedades podem ser encontrados em Lamport e Lynch (1989), Lynch (1996) e Lann e Schmid (2003)

CAPÍTULO 2

SISTEMAS DE TEMPO REAL INDUSTRIAIS DE CONTROLE

2.1 INTRODUÇÃO

Controle e supervisão são mecanismos utilizados na indústria para, não só otimizar a utilização e produção das plantas ativas, mas também incrementar a qualidade e a confiabilidade dos produtos construídos.

O desenvolvimento industrial tem demandado novas tecnologias que possibilitem o controle e a supervisão de plantas cada vez maiores e mais complexas. Essas plantas necessitam de facilidades de gerenciamento, computação e comunicação cada vez mais robustos e eficientes, de modo que se possa maximizar o desempenho do processo produtivo e reduzir os custos de operação e manutenção. Além disso, em muitos casos, as plantas devem trabalhar de forma coordenada e integrada, o que evidencia e aumenta a criticalidade de aspectos como: correção na execução das operações, computação distribuída, restrições temporais das operações e tolerância a falhas.

Acomodar todos os requisitos demandados pelos sistemas industriais modernos é um desafio que tem motivado fabricantes e acadêmicos na busca de soluções. Diversos trabalhos como Wittenmark e Törngren (1994), por exemplo, discutem técnicas de projetos de sistemas de controle com restrições temporais; outros trabalhos como Piuri (1994) e Törngren (1998) abordam questões pertinentes a sistemas de controle distribuídos. Em Andrade e Macêdo (2005) podem ser encontradas discussões focadas no desenvolvi-

mento de uma arquitetura confiável para o desenvolvimento de sistemas de tempo real distribuídos de controle e supervisão. Questões como confiança no funcionamento e tolerância a falhas são abordadas em trabalhos como Piuri (1994), Kim e Shin (1994) e Elks, Dugan e Johnson (2000).

Seguindo esse contexto, este capítulo foca a discussão de aspectos relacionados a implementação de sistemas de tempo real industriais de controle.

2.2 SISTEMAS DE CONTROLE

O objetivo de um sistema de controle é manter as variáveis do processo a ser controlado dentro dos limites pré-estabelecidos, mesmo quando tal processo esteja sujeito a certas perturbações externas. Por conta disso, o sistema deve observar as variáveis que mapeiam o atual estado do processo e atuar em variáveis que permitam que o objetivo do controle seja alcançado[1].

De modo geral, os sistemas de controle podem ser classificados em sistemas de controle de tempo contínuo, ou simplesmente sistema de controle contínuo, e sistemas de controle de tempo discreto, ou simplesmente sistema de controle discreto. Um sistema de controle contínuo é um sistema de controle que observa e manipula sinais[2] definidos sobre uma faixa contínua de tempo, ditos sinais contínuos (HSU, 2004). Um sistema de controle discreto, por outro lado, é um sistema que controla sinais de tempo discreto, ditos sinais discretos. Um sinal discreto é aquele definido apenas em instantes discretos de tempo.

Em geral, para se entender e controlar os sistemas físicos (plantas), é necessário que sejam obtidos modelos matemáticos que determinem o relacionamento entre as variáveis do sistema (COLOM, 2002). Sistemas físicos dinâmicos são, normalmente, descritos através

[1] Aqui se aborda sistema de controle como sinônimo para sistema de controle realimentado. Todavia, existem outras categorias de sistemas de controle, tais como os sistemas de controle de malha aberta, que não serão abordados nesta discussão. Ver Ogata (1990) para maiores detalhes.

[2] Segundo HSU (2004), um sinal é uma função que representa uma quantidade ou variável física e contém informações sobre o comportamento ou natureza de um fenômeno.

de equações diferenciais (BIRKHOFF, 1927).

Sistemas físicos são, em sua grande maioria, não lineares. Um sistema é dito não linear se o relacionamento entre suas variáveis não é linear. Todavia, em vários casos, tais sistemas podem trabalhar dentro de uma região na qual apresentam um comportamento aproximadamente linear. Sendo assim, no projeto do algoritmo (ou lei) de controle é comum utilizar um modelo aproximado representando o sistema físico como linear (OGATA, 1990).

Além disso, as propriedades do sistema físico podem mudar com o tempo. Um sistema físico é dito variante no tempo se os coeficientes das equações que descrevem o sistema mudam com o tempo; de outro modo, o sistema é dito invariante no tempo (HSU, 2004).

Um sistema físico (ou objeto controlado) modelado como um sistema dinâmico linear contínuo invariante no tempo e de ordem n (*LTI – Linear Time Invariant*), por exemplo, pode ser definido em termos de equações diferenciais, como segue:

$$\sum_{i=0}^{n} a_i \frac{d^i y(t)}{dt^i} = \sum_{j=0}^{m} c_j \frac{d^j u(t)}{dt^j} \tag{2.1}$$

onde a e c são constantes reais, enquanto que $y(t)$ representa a variável de saída e $u(t)$ a variável de entrada, que pode ser manipulada pelo sistema de controle. Assim, conhecendo-se as condições iniciais do sistema e a entrada $u(t)$, o comportamento de $y(t)$ poderá ser obtido resolvendo a equação diferencial 2.1. O modelo equivalente para um sistema discreto seria:

$$\sum_{i=0}^{n} a_i y[n-i] = \sum_{j=0}^{m} c_j u[n-j] \tag{2.2}$$

$y[n]$ e $u[n]$ representam a n-éssima amostra das variáveis de saída e de entrada, respectivamente.

Em muitos casos, essas equações diferenciais são de difícil resolução e através das

mesmas é relativamente trabalhoso analisar o comportamento do sistema. Por isso, na grande maioria dos casos, os sistemas são analisados no domínio da freqüência.

A transformada de *Laplace*(OGATA, 1990) é uma ferramenta bastante utilizada para, a partir de um modelo contínuo no domínio do tempo, encontrar seu modelo equivalente no domínio da freqüência. Logo, após algumas simplificações o modelo equivalente no domínio da freqüência da equação 2.1 será dado por:

$$\sum_{i=0}^{n} a_i s^i Y(s) = \sum_{j=0}^{m} c_j s^j U(s) \tag{2.3}$$

onde $s = \sigma + jw$ representa uma freqüência complexa e, $U(s)$ e $Y(s)$ representam a entrada e a saída do sistema no domínio da freqüência.

O modelo que descreve o objeto controlado é, geralmente, expresso através de uma função de transferência. O uso de funções de transferências está diretamente associado ao estudo de sistemas dinâmicos, nos casos em que se relacionam entrada e saída. A função de transferência $G(s)$ é definida como a relação entre $Y(s)$ e $U(s)$, portanto:

$$G(s) = \frac{Y(s)}{U(s)} = \frac{\sum_{j=0}^{m} c_j s^j}{\sum_{i=0}^{n} a_i s^i} \tag{2.4}$$

A figura 2.1 apresenta a representação em termos de diagrama em blocos da função $G(s)$.

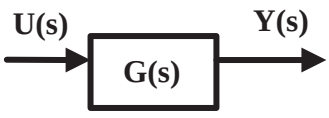


Figura 2.1. Diagrama em bloco da função de transferência $G(s) = \frac{Y(s)}{U(s)}$

A transformada Z é uma ferramenta equivalente à transformada de *Laplace*, só que para um modelo discreto (OGATA, 1995). Assim:

$$\sum_{i=0}^{n} a_i z^{-i} Y(z) = \sum_{j=0}^{m} c_j s^{-j} U(z) \tag{2.5}$$

onde $z = e^{hs}$ e h representa o período de amostragem do sistema[3]. Com isso, o equivalente discreto da função de transferência é expresso por:

$$G(z) = \frac{Y(z)}{U(z)} = \frac{\sum_{j=0}^{m} c_j z^{-j}}{\sum_{i=0}^{n} a_i z^{-i}} \tag{2.6}$$

A mesma representação apresentada na figura 2.1 é válida quando se trabalha no domínio discreto z.

Dado que, a partir da função de transferência $G(s)$ (ou $G(z)$) se chega facilmente à equação 2.1 (ou a 2.2), pode-se afirmar que tal função caracteriza de forma completa o sistema dinâmico (OGATA, 1990).

Em sistema de controle de processos, a ação de controle é gerada com base no atual estado da planta, ou seja, o estado atual é a referência para determinar a ação a ser gerada para determinar o estado futuro e assim por diante. Dessa forma, o sistema deve ser representado por um diagrama em blocos com realimentação (malha fechada). A figura 2.2 apresenta um exemplo de um sistema de controle realimentado (ou de malha fechada), no qual $P(s)$ e $C(s)$ representam, respectivamente, a planta e o dispositivo controlador, enquanto que $U(s)$ e $A(s)$ representam o sinal de referência de entrada (*set point*) e a ação de controle sobre a planta, respectivamente.

A função de transferência de malha aberta $G_A(s)$ é dada pela relação entre o sinal de saída da planta $Y(s)$ e o erro atuante no sistema $E(s)$. Assim:

$$G_A(s) = P(s).C(s) = \frac{Y(s)}{E(s)} \tag{2.7}$$

[3]O período de amostragem representa o intervalo entre dois eventos discretos consecutivos. Para um sistema de controle baseado em computador, por exemplo, esse período de amostragem representa o intervalo entre os instantes usados para observar o estado das variáveis da planta (OGATA, 1995)

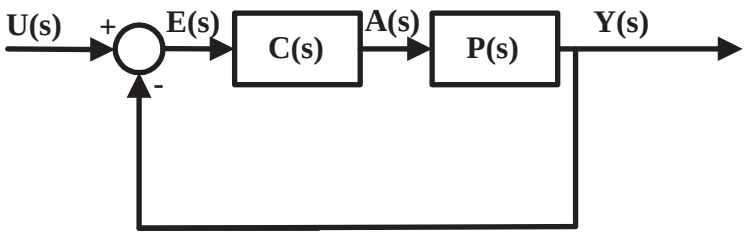


Figura 2.2. Sistema de controle realimentado

A função de transferência em malha fechada $G_F(s)$ pode ser obtida da forma a seguir. Analisando o diagrama da figura 2.2, tem-se:

$$Y(s) = P(s).A(s) \tag{2.8}$$

e,

$$A(s) = C(s).E(s) \tag{2.9}$$

assim,

$$Y(s) = P(s).C(s).E(s) \tag{2.10}$$

mas,

$$E(s) = U(s) - Y(s) \tag{2.11}$$

Portanto:

$$Y(s) = P(s).C(s).\left[U(s) - Y(s)\right] \tag{2.12}$$

Com isso, calcula-se:

$$Y(s).[1 + P(s).C(s)] = P(s).C(s).U(s) \tag{2.13}$$

Por fim,

$$G_F(s) = \frac{Y(s)}{U(s)} = \frac{P(s).C(s)}{1 + P(s).C(s)} \tag{2.14}$$

ou,

$$G_F(s) = \frac{G_A(s)}{1 + G_A(s)} \tag{2.15}$$

Análises importantes das características desejadas para o objeto físico a ser controlado podem ser extraídas a partir de G_F.

Por fim, um outro ponto importante a ser mencionado é que, em geral, para facilitar as análises, G_A é escrito como uma divisão de polinômios:

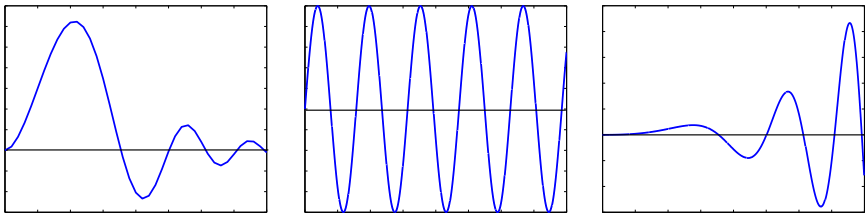
$$G_A(s) = K.\frac{\prod_{i=1}^{m}(s - c_i)}{\prod_{k=1}^{n}(s - p_k)} \tag{2.16}$$

Onde K é uma constante real qualquer, enquanto c e p representam, respectivamente, os zeros e pólos da função.

2.2.1 Analisando a Estabilidade e o Desempenho de Sistemas de Controle

O projeto do controlador é uma peça importante na implementação do sistema de controle. Tal projeto visa levar a malha fechada de controle a ter as características desejadas, as quais estão relacionadas à resposta do sistema físico controlado e à sua estabilidade (COLOM, 2002). A resposta e a estabilidade do sistema físico podem ser obtidas especificando a localização dos pólos da malha fechada de controle. Os pólos da malha fechada são as raízes da equação característica $1 + G_A = 0$ (o denominador da

função de transferência de malha fechada G_F). Os pólos de G_F representam o comportamento autônomo do sistema (COLOM, 2002), determinando a trajetória que o sistema controlado traçará quando sujeito a pertubações externas ou a incertezas intrínsecas ao modelo matemático proposto (SIMON, 2002). Os pólos de um sistema contínuo ($G_F(s)$) e seu equivalente discreto ($G_F(z)$) possuem uma relação direta assumindo um determinado período de amostragem h (OGATA, 1995).



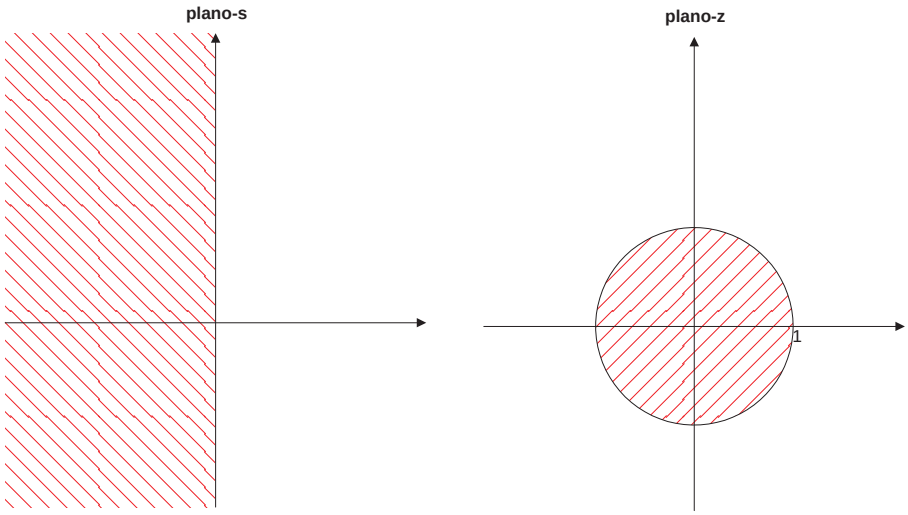
(a) (b) (c)

Figura 2.3. 2.3(a) sistema estável, 2.3(b) marginalmente estável e 2.3(c) instável.

Quando se fala em desempenho, a estabilidade é um requisito fundamental para o projeto de qualquer sistema de controle. Um sistema de controle é dito estável (ou em equilíbrio) se na ausência de qualquer perturbação permanece no mesmo estado ou ainda retorna para o estado de equilíbrio quando sujeito a alguma pertubação (ou condição inicial) (COLOM, 2002), figura 2.3(a). Se a saída do sistema não converge para o equilíbrio nem torna-se ilimitada, o sistema é dito marginalmente estável, figura 2.3(b). Por outro lado, se a saída do sistema torna-se ilimitada, o sistema é dito instável, figura 2.3(c).

Conforme mencionado anteriormente, a estabilidade do sistema de controle realimentado pode ser determinada através da localização dos pólos de G_F. Um sistema de tempo contínuo será estável se todos os pólos de G_F estão no semiplano esquerdo do plano-s, (figura 2.4(a)), (OGATA, 1990). Para um sistema de tempo discreto, a estabilidade é verificada se todos os pólos estão contidos no círculo unitário centrado na origem do plano-z, (figura 2.4(b)),(OGATA, 1990).

Uma outra técnica clássica para avaliar a estabilidade de um sistema de controle



(a) (b)

Figura 2.4. 2.4(a) plano-s, 2.4(b) plano-z.

realimentado é o critério de Nyquist (1932). Esse critério é baseado no teorema da teoria de funções de variáveis complexas conhecido como *Teorema de Cauchy*(OGATA, 1990), o qual é enunciado como segue:

Teorema 2.2.1 (Teorema de Cauchy) *Se um contorno Γ_S no plano-s cerca Z zeros e P pólos da Função $F(s)$ e não cruza nenhum pólo ou zero de $F(s)$, e ainda, a orientação do contorno é no sentido horário, o contorno correspondente Γ_F no plano-$F(s)$ cerca a origem do plano-$F(s)$ $N = Z - P$ vezes no sentido horário.*

A figura 2.5 apresenta um exemplo da aplicação do teorema de Cauchy, no qual o contorno Γ_S no plano-s circunda apenas um pólo da função $F(s)$; como conseqüência, se tem um contorno Γ_F no plano-$F(s)$ circulando uma única vez a origem do plano. Observe que, como nenhum zero de $F(s)$ foi contornado por Γ_S, a origem será contornada $-P$ vezes no sentido horário, o que equivale a P vezes no sentido anti-horário.

O critério de Nyquist (1932) procura determinar se existe algum zero na equação característica em malha fechada $F = 1+G_A = 0$ em uma certa região do plano complexo. Assim, o critério de Nyquist para sistemas de tempo contínuo pode ser enunciado da forma

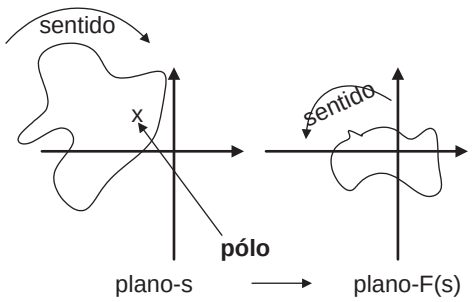


Figura 2.5. Exemplo do teorema de Cauchy, onde o contorno Γ_S circunda apenas um pólo.

a seguir:

Teorema 2.2.2 (Critério de Nyquist - I) *Um sistema de controle realimentado é estável se, e somente se, o contorno $\Gamma_{F'}$ no plano-F' não envolve o ponto $(-1, 0)$ quando o número de pólos de G_A instáveis no plano-s é nulo ($P = 0$), sendo $F' = F - 1 = G_A$.*

Quando o número de pólos de G_A no lado direito do plano-s é não nulo, o critério de (NYQUIST, 1932) pode ser enunciado como segue:

Teorema 2.2.3 (Critério de Nyquist - II) *Um sistema de controle realimentado é estável se, e somente se, para o contorno $\Gamma_{F'}$ no plano-F', o número de cercos no sentido anti-horário do ponto $(-1, 0)$ é igual ao número de pólos de G_A instáveis no plano-s ($N = -P$)*

2.2.1.1 Margens de Ganho e de Fase Os critérios enuciados por Nyquist (1932) podem ser utilizados para determinar o quão robusta é a estabilidade do sistema quando submetida a alterações de ganho e fase incertos e não previstos (SIMON, 2002). Para tanto, Nyquist propôs os conceitos de margem de ganho e de fase, os quais têm sido usados como conceitos clássicos para analisar a robustez da estabilidade de sistemas de controle lineares invariantes no tempo do tipo entrada e saída simples (SISO, *Single Input and Single Output*[4]), quando sujeitos a perturbações externas ou incertezas provenientes

[4]ver Ogata (1990) para maiores detalhes.

das imperfeições do modelo matemático proposto para representar a planta.

As margens de ganho e de fase podem definir a estabilidade em relação ao ponto crítico $(-1, 0)$(SIMON, 2002). Assim, as margens são enunciadas como segue (OGATA, 1990):

Definição 2.2.1 (Margem de Ganho (g_m)) *É o incremento no ganho do sistema quando a fase é* -180^o *que resultará em um sistema marginalmente estável com a intersecção com o ponto* $(-1, 0)$ *no diagrama de Nyquist.*

A margem de ganho pode ser calculada usando a freqüência de cruzamento de fase w_{cp}. Em que w_{cp} representa a freqüência na qual G_A sofre um deslocamento de fase de -180^o. Desse modo:

$$g_m = \frac{1}{|G_A(jw_{cp})|} \tag{2.17}$$

Definição 2.2.2 (Margem de Fase (φ_m)) *É a quantidade de deslocamento de fase de* F *na magnitude* 1 *que resultará em um sistema marginalmente estável com a intersecção com o ponto* $(-1, 0)$ *no diagrama de Nyquist.*

A margem de fase pode ser calculada usando a freqüência de cruzamento de ganho w_{cg}. Em que w_{cg} representa a freqüência na qual $|G_A| = 1$. Desse modo:

$$\varphi_m = 180^o + \angle G_A(jw_{cg}) \tag{2.18}$$

A figura 2.6 apresenta uma representação gráfica das margens de ganho e de fase. Nessa figura, pode-se observar que a margem de ganho é determinada quando o contorno $\Gamma_{F'}$ cruza o eixo real do plano-F', enquanto a margem de fase é determinada através do ângulo formado entre o eixo real do plano-F' e o ponto de interseção entre o contorno $\Gamma_{F'}$ e um círculo unitário com centro em $(0, 0)$.

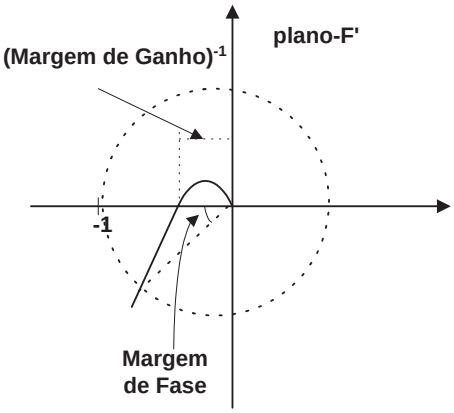


Figura 2.6. Exemplo das Margens de Ganho e de Fase

2.2.2 Avaliando o Desempenho do Sistema de Controle

Uma vez que se pode garantir a estabilidade do sistema, o próximo passo é verificar como o controlador pode minimizar os desvios (erros do sistema) entre o estado desejado (r) e o estado obtido (y) das variáveis do sistema físico controlado (figura 2.7).

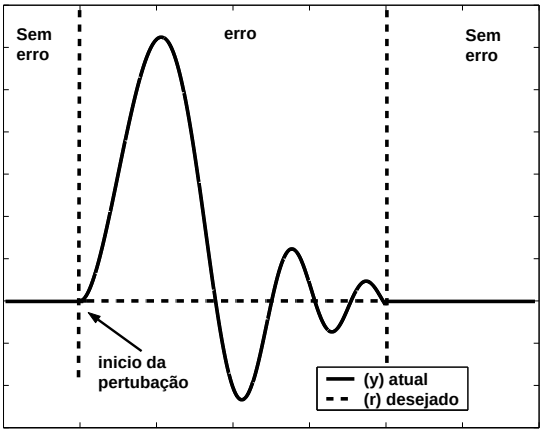


Figura 2.7. Exemplo de uma resposta do sistema a uma perturbação.

Alguns critérios têm sido usados para averigüar o índice de desempenho do sistema de controle em função dos erros quando sujeito a perturbações externas (ou incertezas internas)[5]. Esses índices de desempenho são os seguintes (OGATA, 1990):

[5]No caso dos índices de desempenho, o termo *erro* refere-se à diferença entre o valor desejado e o valor real para o estado de uma determinada variável da planta que está sendo monitorada.

- Integral do valor absoluto do erro (IAE, *Integral of the Absolute value of the Error*):

$$IAE = \int_0^\infty |r(t) - y(t)|\, dt \tag{2.19}$$

- Integral do erro quadrático (ISE, *Integral of Square Error*):

$$ISE = \int_0^\infty (r(t) - y(t))^2 dt \tag{2.20}$$

- Integral do valor absoluto do erro ponderado pelo tempo(ITAE, *Integral of Time-weighted Absolute Error*):

$$ITAE = \int_0^\infty t\,|r(t) - y(t)|\, dt \tag{2.21}$$

- Integral do erro quadrático ponderado pelo tempo (ITSE, *Integral of Time-weighted Square Error*):

$$ITSE = \int_0^\infty t(r(t) - y(t))^2 dt \tag{2.22}$$

Esses critérios medem o desempenho do sistema, mas usam diferentes abordagens. Por exemplo, os critérios que ponderam o erro pelo tempo ($ITAE$ e $ITSE$) penalizam mais drasticamente os últimos erros. Os demais critérios (IAE e ISE) ponderam todos os erros igualmente[6].

2.3 SISTEMA DE CONTROLE DE TEMPO REAL

Incrementar a capacidade produtiva é um elemento chave que vem sendo abordado como um dos principais pontos pela indústria de automação e controle de processos. Com isso, o desenvolvimento de algoritmos ótimos que incrementem a produtividade, aten-

[6]Uma discussão mais detalhada pode ser encontrada em (OGATA, 1990)

dendo às restrições impostas pelos dispositivos de hardware, pela dinâmica do processo ou ainda pela capacidade de investimento disponível, têm sido um ponto amplamente estudado[7].

A necessidade de implementação de algoritmos melhores e mais eficientes demandou dispositivos com um maior poder de computação e comunicação. Isso inspirou, entre outras coisas, a inserção de computadores na malha de controle. Naturalmente, novas possibilidades surgiram: um sistema de controle baseado em computador, não só permite o desenvolvimento de algoritmos mais robustos e flexíveis, mas também possibilita que vários algoritmos de controle compartilhem uma mesma unidade de processamento. Assim sendo, diversas plantas podem ser controladas por um único dispositivo, o que promove uma diminuição dos custos e possibilita um melhor gerenciamento.

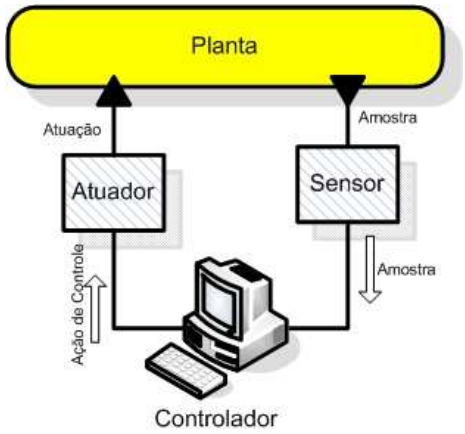


Figura 2.8. Sistema de controle por computador

Sistemas de controle baseados em computador são sistemas discretos cujo sinal é quantizado em amplitude (sinal digital). Um sinal quantizado em amplitude é aquele no qual a amplitude varia dentro de um conjunto limitado de valores. No caso do sistema baseado em computador, o sinal digital usa codificação binária e a amplitude pode assumir apenas dois valores distintos. A figura 2.8 mostra um exemplo de um sistema controlado

[7]Exemplos de alguns dos diversos estudos existentes nesse contexto podem ser encontrados em: (PAGANINI, 1996), (ÅSTRÖM, 2000), (GOODWIN; GRAEBE; SALGADO, 2000), (YOOK; TILBURY; SOPARKART, 2000), (MARTí et al., 2001), (SETO; LEHOCZKY; SHIN, 2001), (MARTí et al., 2002) e (BUTTAZZO et al., 2004)

por computador. Nesse exemplo, o sistema de controle percebe o atual estado do processo através das informações (amostras) providas pelo dispositivo sensor. As amostras são processadas pelo algoritmo de controle, o qual envia ao dispositivo atuador a ação de controle a ser executada sobre as variáveis que estão sendo manipuladas.

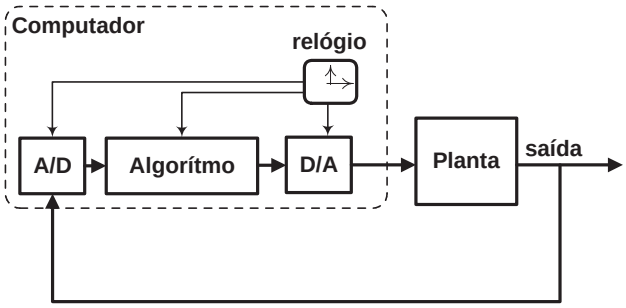


Figura 2.9. Esquema de um sistema de controle por computador

Em geral, os sinais obtidos das variáveis do processo são analógicas[8]; logo, existe a necessidade de uma conversão Analógico-Digital (A/D) e Digital-Analógico (D/A) no *interfaceamento* entre o computador e o processo. A figura 2.9 apresenta um esquema geral de um sistema de controle baseado em computador. O relógio (*clock*) registra e dita os instantes em que os eventos ocorrem (ou devem ocorrer).

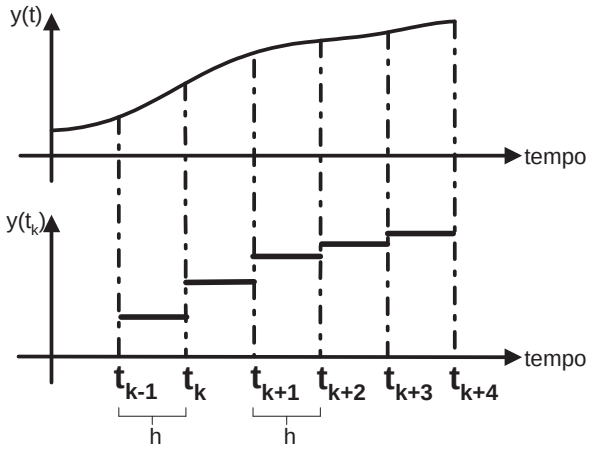


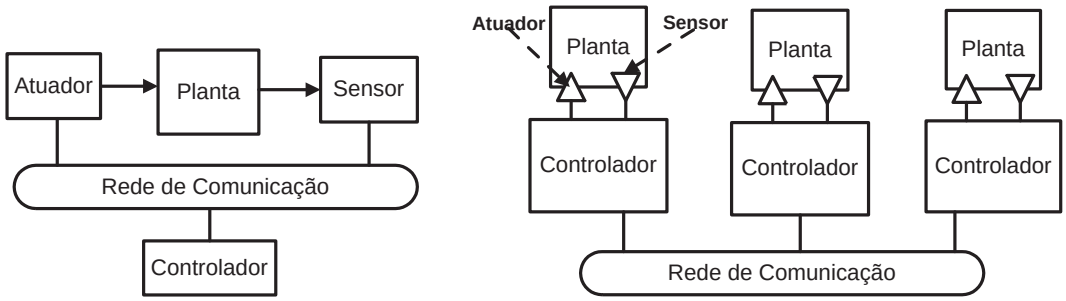
Figura 2.10. Intervalo entre coleta de amostras

[8]Variáveis de tempo contínuo e amplitude não quantizada .

As amostras são coletas em períodos regulares h (período de amostragem) de tempo, como pode ser visto na figura 2.10. Nessa figura, $y(t)$ e $y(t_k)$ representam a saída analógica e sua versão digitalizada, respectivamente. t_k é o instante em que a k-ésima amostra foi coletada.

O controle por computador é, em muitos casos, constituído por sistema operacional com *kernel* multi-tarefa de tempo real e suporte a comunicação em rede. Para atender às demandas de mercado, esse sistema deve suportar implementações complexas com o menor custo possível (CERVIN et al., 2003).

Diversas implementações de sistemas controlados por computador são modeladas concebendo componentes distribuídos, onde sensores, atuadores e sistema de controle se comunicam através de uma rede de comunicação (figura 2.11(a)). Esse modelo de sistema vem sendo chamado na literatura de Sistema de Controle sobre Rede (**NCS**, ***N****etworked* ***C****ontrol* ***S****ystems*). Em alguns cenários, os sistemas de controle podem, ainda, comunicar entre si para promover a interação entre diversas plantas e formar um modelo mais eficiente e confiável (figura 2.11(b)). Tais modelos vêm sendo denotados na literatura por Sistema de Controle Distribuído (**DCS**, ***D****istributed* ***C****ontrol* ***S****ystems*)[9].



(a) *NCS* (b) *DCS*

Figura 2.11. Exemplo de *NCS* e *DCS*.

Tanto os modelos de sistema NCS quanto os modelos DCS podem se utilizar das

[9] *DCS* e *NCS* têm sido bastante discutidos em diferentes aspectos; como exemplo podem ser citados os trabalhos de (GLASER; KORDECKI; REMBOLD, 1989), (HILMER; KOCHS; DITTMAR, 1997), (YOOK; TILBURY; SOPARKART, 2000), (ANDRADE; MACêDO, 2005) e (JI; KIM, 1997).

técnicas e métodos da área de sistemas distribuídos[10] e tolerância a falhas[11] na resolução dos problemas.

As abordagens de controle baseadas em computador com componentes distribuídos (**NCS** e **DCS**) trazem alguns aspectos importantes, pertinentes à computação e à comunicação, que precisam ser considerados durante o projeto do sistema (GAMBIER, 2004). Tais aspectos são listados a seguir:

- Durante o processo de conversão **A/D** e **D/A**, erros de quantização[12] são inseridos. Tais erros podem impactar no desempenho do controle realizado. Em geral, esses erros de conversão podem ser minimizados através da utilização de conversores mais precisos ou com uma faixa de conversão mais adequada à magnitude do sinal que está sendo manipulado. Esse é um problema inerente aos sistemas de controle discretos(OGATA, 1995) e é herdado pelo modelo de tempo real distribuído.

- Necessidade de desenvolvimento de software verificáveis de modo a minimizar os erros de programação. O desenvolvimento de algoritmos mais confiáveis é um problema que pode ser atacado através da utilização de metodologias mais sofisticadas de desenvolvimento de software, que aumentem a reusabilidade do código produzido e incrementem a manutenabilidade. Nesse tópico é interessante a adoção de ambientes de execução (middleware, sistema operacional etc.) e linguagens que permitam um mapeamento mais fácil do modelo de ativação de tarefas, das restrições temporais e dos requisitos de distribuição e confiabilidade das aplicações de controle de tempo real(ANDRADE; MACêDO, 2005; MACÊDO et al., 2004).

- É preciso que o projeto do sistema de controle considere o atraso necessário ao processamento do algoritmo (atraso de computação) e na comunicação. Tais atrasos representam um problema, que quando não considerados durante o projeto, podem

[10] Ver Lamport e Lynch (1989) e Veríssimo e Rodrigues (2000)
[11] Ver Cristian (1991), Jalote (1994) e Gärtner (1999)
[12] A quantização é o processo pelo qual um sinal digital qualquer é transformado em um sinal binário.

levar ao desenvolvimento de um algoritmo de controle ineficiente. Mas, apesar da literatura convencional não abordar diretamente tal problema, derivar soluções que considerem o tempo de computação não é uma tarefa difícil. Contudo, esses atrasos podem exigir não apenas um reprojeto do algoritmo, mas também a utilização de arquiteturas de hardware e software básico mais sofisticadas, de modo a melhorar o tempo de resposta e diminuir o efeito do atraso para aplicações de controle críticas de tempo. Isto, todavia, pode encarecer o projeto do sistema como um todo.

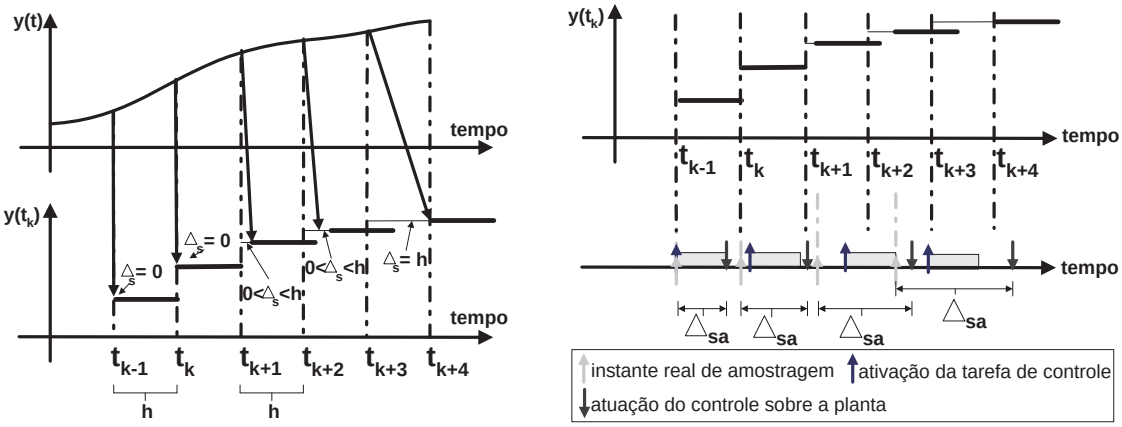
- Introdução de atrasos variados nos instantes de ativação da tarefa de controle, provocado pela utilização de um ambiente multitarefa e baseado em prioridades ou ainda pela utilização de plataformas convencionais de hardware, software e/ou comunicação. Em geral, a utilização de soluções especializadas podem implicar no encarecimento do valor final do sistema. Isso tem motivado a utilização de componentes de prateleira (*COTS*, ***C****ommercial* ***O****ff-****T****he-****S****helf*) que, apesar do baixo custo, pode incrementar a complexidade de implementação do projeto e aumentar as componentes de incertezas temporais do sistema.

A existência de atrasos variados e a necessidade do desenvolvimento de algoritmos eficientes, mesmo quando sujeitos a tais atrasos, é um ponto bastante relevante que tem incentivado o desenvolvimento de técnicas de *codesign*, onde controle e tempo real são implementados em conjunto (CERVIN, 2004; CERVIN et al., 2004; BUTTAZZO et al., 2004).

2.4 SISTEMAS DE CONTROLE DE TEMPO REAL SUJEITO A VARIAÇÕES TEMPORAIS

A implementação do controlador como uma tarefa de tempo real insere incertezas que são inerentes à plataforma de computação(CERVIN et al., 2004; CERVIN, 2004) e de comunicação(LIAN; MOYNE; TILBURY, 2001). Tais incertezas estão relacionadas às variações no

tempo de execução das tarefas (atraso de computação), no envio e entrega das mensagens pelo subsistema de comunicação (atraso de comunicação).



(a) *Atraso na amostragem* (b) *Atraso na ativação e atuação do controle*

Figura 2.12. Exemplo de interferência do atraso $\Delta = \Delta_s + \Delta_{sa}$

As figuras 2.12(a) e 2.12(b) apresentam um exemplo da influência dos atrasos no funcionamento do sistema de controle[13]. Nessas figuras, Δ_s e Δ_{sa} representam o atraso no sensoriamento e na atuação do sistema de controle. O atraso no sensoriamento é a diferença entre o instante desejado e real no processo de amostragem. O atraso na atuação, por sua vez, é a diferença entre o instante no qual o controle foi ativado até o instante em que a atuação é realizada sobre a planta. O atraso no sensoriamento pode ser provocado pelo tempo necessário para que o dispositivo sensor realize a transdução[14] (e, em alguns casos, a conversão A/D) do sinal e/ou ainda pelo tempo necessário para que a amostra chegue até o dispositivo controlador (atraso de comunicação). O atraso na atuação pode ser desmembrado em outras componentes:

- **atrasos de liberação e de execução da tarefa de controle**, provocados pela

[13]As figuras 2.12(a) e 2.12(b) são baseadas em (MARTí et al., 1999). Na figura 2.12(b), a ativação do controle é indicada por uma seta em preto voltada para cima, enquanto a atuação é indicada por uma seta em preto voltada para baixo. O instante real de amostragem é indicado por uma seta em branco voltada para cima.

[14]Transformação de uma grandeza física (como temperatura, pressão, velocidade) em uma grandeza elétrica (como corrente ou voltagem).

política de escalonamento de tarefas adotada (FARINES; FRAGA; OLIVEIRA, 2000).

- **atraso de transferência da ação de controle**, intervalo de tempo necessário para que a ação de controle chegue ao dispositivo atuador.

- **atraso de execução da atuação**, tempo necessário para que o atuador receba a ação de controle e efetivamente atue sobre a planta (podendo envolver o tempo necessário para uma conversão **D/A**).

Essas variações podem alterar o comportamento do controle, degradando o desempenho e podendo levar o sistema a comportar-se de forma instável (MARTÍ et al., 2001, 2002). O projeto clássico do sistema de controle, muitas vezes, não leva em consideração a existência de tais incertezas, ou ainda pressupõe que estas são muito pequenas para serem consideradas. Na prática, tais incertezas afetam consideravelmente o sistema, principalmente quando tal sistema é implementado como controle em rede (*NCS*) ou como um controle distribuído (*DCS*). Essas variações de tempo implicam diretamente em variações nos instantes de atuação e, mesmo quando não levam à instabilidade, alteram o desempenho do controle realizado.

O problema torna-se ainda mais complexo quando aplicações de controle de *missão-crítica* precisam ser implementadas. Essas aplicações necessitam de mecanismos que atribuam um maior grau de confiança ao funcionamento, habilitando o sistema com a capacidade de evitar ou tolerar a existência de falhas. Na ocorrência de falhas em componentes do sistema, mecanismos de tolerância a falhas devem detectar o erro, localizar e confinar a falha e promover a recuperação e ou reconfiguração do sistema. Assim sendo, a adição desses mecanismos pode incrementar consideravelmente o atraso computacional e de comunicação, podendo fazer com que o sistema viole as condições necessárias para a garantia da estabilidade (KIM; SHIN, 1994; ELKS; DUGAN; JOHNSON, 2000).

Conforme discutido anteriormente, medidas clássicas como a *margem de fase* e a *margem de ganho* são usadas para medir a estabilidade e descrever a sensibilidade da malha

de controle quando a planta apresenta distúrbios em seu comportamento (CERVIN et al., 2004). Entretanto, tais medidas são definidas sem levar em consideração a existência de incertezas na plataforma computacional, ou falhas de dispositivos. Diversos trabalhos têm sido realizados para promover algoritmos mais eficientes (KAO; LINCOLN, 2004; LINCOLN, 2003; BUTTAZZO et al., 2004; MARTÍ et al., 2001), bem como alguns outros trabalhos visam o desenvolvimento de metodologias de *codesign* para o desenvolvimento de controle em tempo real (MARTÍ et al., 2001, 2002; CERVIN, 2004), diminuindo a lacuna existente entre as duas áreas.

Algumas técnicas aplicáveis ao projeto de sistemas de controle de tempo real sujeitos a atrasos variados e falhas serão apresentadas nas seções a seguir.

2.4.1 Margem de Atraso e de Jitter

Para analisar o desempenho de um sistema de controle com variações no atraso de computação, Cervin (2004) desenvolveu o conceito de *margem de jitter*, a qual representa a maior variação do atraso que pode ser tolerada pelo sistema sem que o mesmo perca a estabilidade.

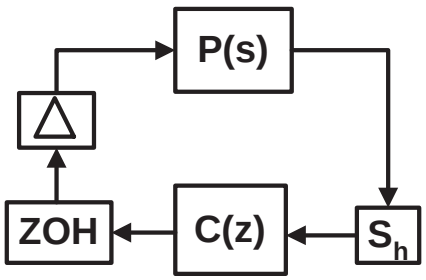


Figura 2.13. Sistema de controle com $P(s)$ e $C(z)$ e atrasos na atuação do controle

A *margem de jitter* é obtida considerando uma malha de controle formada por uma planta contínua, $P(s)$, controlada por um controlador discreto, $C(z)$. Nessa malha de controle o tempo de amostragem h não sofre variações, entretanto, atrasos variados (*jitter*) podem ocorrer nos instantes de atuação do controle sobre a planta. Tal modelo é

explicitado na figura 2.13, na qual: ZOH representa um *Zero-Order-Hold*(OGATA, 1995), S_h é um *sample-and-hold*(OGATA, 1995) com um período de h segundos e Δ é um atraso variante no tempo.

Segundo Cervin (2004), para a malha de controle da figura 2.13, a estabilidade pode ser obtida seguindo o teorema de Kao e Lincoln (2004):

Teorema 2.4.1 (Estabilidade sob *jitter* de saída) *O sistema de malha fechada da figura 2.13 é estável para qualquer atraso variante no tempo* $\Delta \in [0, N.h]$, *onde* $N \in \Re_+^*$, *se*

$$\left| \frac{P_{alias}(W)C(e^{iw})}{1 + P_{ZOH}(e^{iw})C(e^{iw})} \right| < \frac{1}{\widehat{N}\,|e^{iw} - 1|}, \forall w \in [0, \pi]\,, \tag{2.23}$$

onde $\widehat{N} = \sqrt{\lfloor N \rfloor^2 + 2\,\lfloor N \rfloor\, g + g}$; $g = N - \lfloor N \rfloor$; $P_{ZOH}(z)$ *é a discretização*[15] *zero-order hold de* $P(s)$*(OGATA, 1995) e*

$$P_{alias}(w) = \sqrt{\sum_{k=-\infty}^{\infty} \left| P\left(i(w + 2\pi k)\frac{1}{h} \right) \right|^2} \tag{2.24}$$

O modelo adotado por (CERVIN, 2004), considera uma tarefa periódica de controle executando em um sistema de tempo real, no qual assume-se que a planta é amostrada quando a tarefa é liberada e que o sinal de controle é enviado ao atuador quando a tarefa é finalizada. Sendo assim, define-se um atraso de entrada (coleta da amostra) e saída (atuação do controle) que pode ser dividido em duas partes. A primeira parte representa um atraso constante ($L \geq 0$) que pode ser definida como o tempo de atuação no melhor caso. A segunda parte se refere a um atraso variado ($J \geq 0$), o qual representa um instante de atuação que pode acontecer entre os tempos de atuação no melhor e no pior caso, conforme apresentado na figura 2.14.

Dessa forma, o menor atraso possível será dado por L e o maior atraso permitido será

[15]Processo de transformação de um sinal analógico em um sinal digital(discreto).

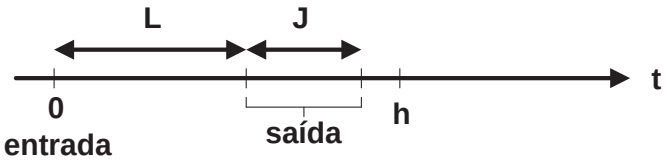


Figura 2.14. Atraso de entrada e saída dividido em constante e variado

dado por $L + J$. Quando $J = 0$, L representa a margem de atraso (*tempo morto*[16]) da teoria clássica de controle, na qual:

Definição 2.4.1 (Margem de Atraso) *A margem de atraso é definida com o maior número L_m para o qual a estabilidade é garantida assumindo um atraso constante $\Delta = L_m$.*

Para sistemas de controle contínuos, a margem de atraso sem *jitter* é dada por:

$$L_m = \frac{\varphi_m}{w_{cg}}, \tag{2.25}$$

em que φ_m representa a margem de fase e w_{cg} a freqüência de cruzamento do ganho.

Para sistemas onde atrasos variados estão presentes, a margem de *jitter* é apresentada de acordo com a definição a seguir (CERVIN, 2004):

Definição 2.4.2 (Margem de *Jitter*) *A margem de jitter é definida como o maior número $J_m(L)$ para o qual a estabilidade é garantida para qualquer atraso variante no tempo $\Delta \in [L, L + J_m(L)]$.*

A margem de *jitter* possui as seguintes propriedades:

Proposição 2.4.1 *Se $L \geq L_m$, então $J_m(L) = 0$.*

Proposição 2.4.2 *$J_m(L) \leq L_m$, $\forall L$.*

Proposição 2.4.3 *$J_m(L) + L \geq L$.*

[16] ver Ogata (1990) para maiores detalhes.

O conceito de margem de *jitter* pode ser usado para verificar a estabilidade do sistema. Assumindo uma componente de atraso constante L e uma componente de atraso variado (*jitter*) J de uma tarefa de controle, a estabilidade é garantida se $J_m(L) > J$. Assim:

Definição 2.4.3 (Margem de Atraso para sistemas com atraso e *jitter*) *Assumindo um atraso constante L e jitter J, a margem de atraso é definida como o maior número L_m para o qual a estabilidade da malha fechada de controle com jitter é garantida para qualquer atraso variante no tempo $\Delta \in [L + L_m, L + L_m + J]$.*

Segundo (CERVIN, 2004), a margem de atraso pode ser analisada em termos da margem de *jitter* como sendo o menor valor de L_m que soluciona a equação 2.26.

$$J_m(L + L_m) = J \tag{2.26}$$

Através da margem de *jitter* é possível atribuir os *deadlines*, de modo que se consiga garantir a estabilidade da malha de controle. Uma vez que o atraso mínimo L é conhecido, o *deadline* relativo D pode ser calculado por:

$$D = L + J_m(L) \tag{2.27}$$

Para sistemas de controle com *jitter*, o conceito de margem de fase não está definido. Assim, (CERVIN, 2004) definiu o conceito de margem de fase aparente como uma derivação da margem de fase para sistemas com *jitter* de entrada e saída:

Definição 2.4.4 (Margem de Fase Aparente) *Assumindo um atraso constante L e o jitter J, a margem de fase aparente é definida como o maior número $\widehat{\varphi}_m$ para o qual a estabilidade de um sistema de malha fechada com jitter de saída pode ser garantida para qualquer atraso variante no tempo $\Delta \in \left[L + \frac{\widehat{\varphi}_m}{w_{cg}}, L + \frac{\widehat{\varphi}_m}{w_{cg}} + J\right]$, em que w_{cg} representa a freqüência de cruzamento do ganho.*

Uma vez definida a margem de fase aparente, a equação 2.26 pode ser reescrita da seguinte forma:

$$J_m(L + \frac{\widehat{\varphi}_m}{w_{cg}}) = J \tag{2.28}$$

Ou seja, o problema torna-se encontrar a menor $\widehat{\varphi}_m$ que atenda a equação 2.28.

Assumindo que na ausência de atraso ou jitter o sistema é estável e possui o desempenho desejado, a relação entre a margem de fase φ_m e a margem de fase aparente $\widehat{\varphi}_m$ pode ser usada, em conjunto com outras métricas, como uma medida do desempenho do sistema: quanto mais próxima a margem de fase aparente estiver da margem de fase convencional, mais estável e melhor desempenho terá o sistema. Portanto, se deseja maximizar a relação de desempenho $QoP = \frac{\widehat{\varphi}_m}{\varphi_m}$.

2.5 CONSIDERAÇÕES FINAIS

Neste capítulo, foram apresentados alguns conceitos básicos relacionados à implementação de sistemas de controle de tempo real. De acordo com o que foi abordado, a implementação de tais sistemas deve levar em consideração uma relação de compromisso entre eficiência do algoritmo de controle e o atendimento às restrições temporais.

Sistemas de controle implementados em computador ou sistemas de controle sobre rede enfrentam barreiras que podem impactar no desempenho do controle realizado. Tais barreiras estão relacionadas ao atraso de computação e de comunicação. O atraso de computação é influenciado pela política de escalonamento de tarefas adotada e pelas restrições pertinentes aos recursos computacionais. O atraso de comunicação, por outro lado, está diretamente ligado à política de acesso ao meio e às características da rede, como taxa de transmissão, confiabilidade etc.

Esses atrasos são extremamente nocivos ao desempenho do sistema de controle e se agravam ainda mais quando componentes de prateleira são utilizados na implementação

do sistema. Componentes convencionais, apesar de serem bastante atraentes por conta do custo e por permitir uma integração mais facilitada, podem impor atrasos de comunicação e computação não determinísticos.

As abordagens tradicionais utilizadas para a construção de sistemas de controle, não levam em consideração ambientes sujeitos a tais variações, possuindo, dessa forma, uma aplicabilidade bastante limitada em tais cenários. O projetista do sistema de controle deve possuir, portanto, uma metodologia eficiente para construir algoritmos eficientes e que levem em consideração as condições impostas pelo ambiente computacional de tempo real. Nesse contexto, o conceito de *margem de jitter* se mostra uma ferramenta interessante para guiar o projeto do sistema ou, ainda, avaliar o desempenho do sistema implementado.

CAPÍTULO 3

DETECÇÃO DE DEFEITOS: ADAPTABILIDADE E QUALIDADE DE SERVIÇO

3.1 INTRODUÇÃO

Muitos dos problemas associados à área de sistemas distribuídos críticos são herdados pelos sistemas críticos de controle sobre rede e pelos sistemas críticos de controle distribuídos. Assim, muitas soluções e conceitos básicos abordados na área de sistema distribuídos podem ser aplicados a *NCS* e a *DCS*. Desse modo, este capítulo faz uma breve introdução a respeito dos conceitos básicos na construção de sistemas distribuídos críticos e, em seguida, foca na discussão sobre a implementação dos mecanismos de detecção de defeitos.

3.1.1 Sistemas Distribuídos

Um sistema distribuído pode ser definido por um conjunto de unidades de processamento dotadas de potencial de armazenamento e interconectadas através de um canal de comunicação. Cada unidade de processamento possui um relógio local e independente dos demais, o qual é utilizado para registrar o momento de ocorrência de eventos locais. Sobre essas unidades de processamento executam um conjunto de processos que se comunicam através de troca de mensagens e colaboram para atingir um objetivo comum(VERÍSSIMO; RODRIGUES, 2000).

Em geral, os sistemas distribuídos são representados por modelos abstratos que de-

finem as formas de interação entre os processos do sistema (LAMPORT; LYNCH, 1989). Diversos modelos podem apresentar diferentes visões de um mesmo sistema e, além disso, podem ser classificados de acordo com as hipóteses assumidas. Algumas dessas hipóteses dizem respeito às condições de sincronia do sistema (ponto chave na especificação de sistemas de tempo real).

De acordo com a sincronia, um modelo de sistema distribuído pode ser classificado em síncrono ou assíncrono. Em um modelo de sistema completamente assíncrono os limites temporais são desconhecidos ou inexistentes. Assim, entre outras coisas, não se pode fazer considerações sobre os tempos de processamento e de viagens das mensagens. Em um sistema síncrono, por outro lado, os limites temporais são conhecidos a priori.

Uma vez que os modelos de sistemas assíncronos são implementados livres de restrições temporais, existe um maior potencial de reutilização e adaptação da solução em ambientes com um alto grau de incerteza. Entretanto, se falhas de processos ou meios de comunicação forem considerados, a solução de certos problemas básicos em um modelo puramente assíncrono se torna uma tarefa difícil e sem solução determinística (FISCHER; LYNCH; PATERSON, 1985).

Os modelos de sistemas síncronos são bastante convenientes para sistemas em que as restrições temporais são imperativas para o correto funcionamento do sistema. Entretanto, as definições dos limites temporais podem ser bastante difíceis de serem obtidas. Por exemplo, o tempo máximo de execução de um algoritmo depende da implementação da plataforma de hardware, do número de instruções geradas pelo compilador, da política de acesso ao meio de comunicação etc. Além disso, erros durante a especificação de limites temporais incertos podem levar os sistemas a falhar durante a sua execução.

Modelos síncronos e assíncronos ocupam dois extremos em termos de sincronia e, de certo modo, ambos não são realísticos. Sistemas reais tipicamente possuem alguma hipótese temporal e, mesmo quando as restrições temporais são fortes, algumas componentes de tempo são incertas. Esse fato tem motivado o desenvolvimento de modelos que

ocupem uma posição intermediária entre os modelos síncronos e assíncronos. A exemplo disso, estão os modelos assíncronos com tempo de Cristian e Fetzer (1999) e os assíncronos com detectores de defeitos de Chandra e Toueg (1996). Além desses, existem arquiteturas híbridas como o *TCB* (*Timely Computing Base*) de Almeida, Veríssimo e Casimiro (1998) e o modelo híbrido e adaptativo de Gorender, Mâcedo e Raynal (2005). Esses modelos são ditos parcialmente síncronos, haja vista que possuem alguma informação de tempo, mas essa informação pode não ser exata.

3.1.2 Confiança no Funcionamento

Para alguns sistemas, a falha de um dos seus componentes pode levar a situações catastróficas. Esses sistemas são ditos sistemas críticos e é necessário atribuir confiança em seu funcionamento (*Dependability*). Um sistema é confiável se provê o serviço de acordo com a sua especificação. Existem duas técnicas básicas para se atribuir confiança no funcionamento de um sistema (LAPRIE, 1985): Prevenção e Tolerância a Falhas.

A Prevenção de falhas se baseia na utilização de componentes e métodos robustos durante a construção do sistema de modo a evitar a ocorrência de falhas durante o funcionamento de tal sistema. Entretanto, por melhores que sejam os componentes e as técnicas utilizadas é impossível prevenir todas as falhas. Alguns sistemas podem ser extremamente complexos e de composição completamente heterogênea de maneira que prevenir a ocorrência de falhas se torna muito complicado (ELDER, 2001). Além disso, eventos físicos externos podem levar o sistema a falhar.

Tolerância a falhas se baseia na utilização da redundância necessária para capacitar o sistema da habilidade de prover um serviço confiável mesmo após a ocorrência de uma falha. Prevenção e Tolerância a falhas são mecanismos essenciais e devem ser aplicados em conjunto para atribuir confiança ao funcionamento de um sistema.

3.1.2.1 Falhas, Erros e Defeitos Falhas, Erros e Defeitos são elementos nocivos à confiabilidade de um sistema crítico que devem ser considerados durante a sua implementação(LAPRIE, 1985). Para caracterizar a ocorrência de um desses elementos nocivos, necessita-se de uma especificação bem definida dos requisitos do sistema (CHILLAREGE, 1986). A especificação do sistema é a base para validação dos requisitos e das hipóteses de falhas.

Conceitualmente, um defeito denota um desvio entre o serviço atualmente provido e o que foi especificado (LAPRIE, 1985). Tal desvio é originado de um estado errôneo(erro) ao qual o sistema é levado na presença de uma falha. Um erro pode permanecer latente no sistema até que algum evento promova a sua ativação. A falha é um evento indesejado que pode inserir um erro latente no sistema. Como já foi dito, o erro latente, quando ativado, faz com que o defeito aconteça. Dessa forma, pode-se afirmar que a falha é a causa indireta e primária do defeito.

O defeito em um nível do sistema pode levar à falha em um componente ou sistema usuário de tal serviço, provocando um encadeamento de falhas, erros e defeitos. A figura 3.1, adaptada de Deswarte, Kanoun e Laprie (1998), apresenta um exemplo do encadeamento entre falha, erro e defeito.

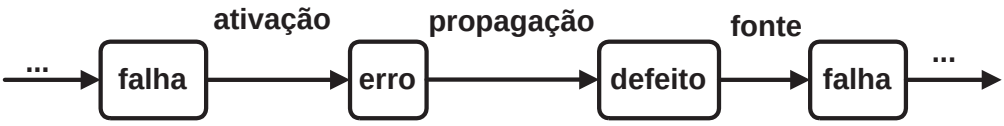


Figura 3.1. Cadeia de Falhas, erros e defeitos

3.1.2.2 Fases em Tolerância a Falhas Do ponto de vista do projeto de um sistema crítico, além das especificações funcionais e temporais que um sistema crítico deve atender, capacitá-lo com mecanismos de tolerância a falhas exige que os projetistas considerem fases adicionais pertinentes à implementação de tais mecanismos. Apesar de

serem fases pertencentes apenas à implementação dos mecanismos de tolerância a falhas, as mesmas estarão intimamente ligadas aos requisitos que o sistema deve atender (JALOTE, 1994). Essas fases dizem respeito às ações que devem ser tomadas por um conjunto de mecanismos de tolerância a falhas para tratar as falhas dos componentes do sistema, de modo a garantir a continuidade do serviço. Tais fases são (JALOTE, 1994): detecção do erro (*error detection*) no estado de algum dos componentes, confinamento dos riscos (*damage confinement*) de propagação do erro, processamento do erro (*error processing*) e tratamento da falha (*fault treatment*).

Conforme mencionado, a manifestação de uma falha pode levar o sistema a um estado de erro. A ativação de um estado de erro é a causa da manifestação de um defeito no sistema. Desse modo, para evitar que o defeito aconteça é necessário que um processo para detecção de erro seja utilizado. Após detectar-se o erro, é necessário calcular sua extensão para realizar um confinamento do mesmo de modo a limitar sua propagação. Uma vez que o erro está confinado, é necessário realizar um processamento que possa recuperar o sistema do estado de erro (*error recovery*) ou compensar o erro através de uso de redundância (*error compensation* ou *error masking*). Dado que o processamento do erro foi efetuado, é preciso que o tratamento da falha causadora do erro seja executado, evitando que novos erros aconteçam em decorrência de tal falha. Esse tratamento implica na identificação do componente defeituoso ou na reconfiguração do sistema de modo a prevenir que tal componente venha a ocasionar a falha de forma recorrente (JALOTE, 1994).

3.2 DETECÇÃO DE DEFEITOS

A detecção de erros consiste em realizar verificações no estado do sistema a fim de detectar estados errôneos. Uma vez que o estado de erro é verificado através do defeito em um dos componentes do sistema, e o defeito de um componente representa uma

falha quando observado o sistema como um todo, a detecção de erros também pode ser denominada por detecção de defeitos de componentes ou detecção de falhas no sistema.

Um detector de defeitos em componentes, ou simplesmente detector de defeitos, é um mecanismo elementar na construção de sistemas distribuídos tolerantes a falhas. Os detectores de defeitos são elementos que provêem informações sobre os estados dos componentes do sistema. Tais informações são geralmente compostas por uma lista de componentes falhos (ou suspeitos de terem falhado (CHEN; TOUEG; AGUILERA, 2002)). Contudo, em certos modelos de sistemas, em que não há garantias nos limites temporais ou em que alguns limites temporais são incertos, essa lista de componentes pode não estar atualizada ou correta, uma vez que os detectores de defeitos (devido a atraso no canal de comunicação, perda de mensagens etc.) podem errar, suspeitando de componentes corretos ou ainda deixando de suspeitar de componentes que efetivamente falharam.

Nos ambientes onde existem variações (ou incertezas) nos tempos de processamento ou de comunicação, a adaptabilidade é uma característica importante para garantir uma detecção de defeitos mais rápida e confiável. Por outro lado, uma vez que as informações de tais detectores podem ser imprecisas, métricas de qualidade de serviço constituem uma ferramenta importante para quantificar a rapidez e a confiabilidade do serviço de detecção. Desse modo, as próximas subseções discutem pontos relevantes sobre a detecção adaptativa de defeitos e aborda um *framework* de métricas proposto na literatura para avaliar a qualidade do serviço de detecção.

3.2.1 Detecção Distribuída de Defeitos

Em um ambiente distribuído, a comunicação entre os dispositivos do sistema é realizada através da troca de mensagens. Assim, a implementação de um detector de defeitos deve considerar um monitoramento remoto de componente baseado em tal forma de comunicação. O monitoramento do estado do componente pode ser feito de duas formas

básicas: monitoramento *Pull*, no qual o detector de defeitos envia, periodicamente, uma mensagem questionando o estado atual do componente (mensagens de *Are you alive?*); e monitoramento *Push*, no qual o componente monitorado envia, de forma periódica, uma mensagem informando seu atual estado – mensagens de *I am alive!* ou *heartbeat* – (FELBER, 1998).

No modelo *Pull*, figura 3.2, uma vez recebida a mensagem do detector de defeitos (monitor), o componente monitorado deve responder com seu atual estado (mensagem de *I am alive!* ou *heartbeat*). A cada mensagem de monitoramento a ser recebida, o detector de defeitos deve estimar o intervalo de tempo necessário (*timeout*) para a chegada da mensagem de resposta oriunda do componente monitorado. Caso a mensagem não chegue dentro do intervalo esperado, o detector passa a suspeitar da falha do componente.

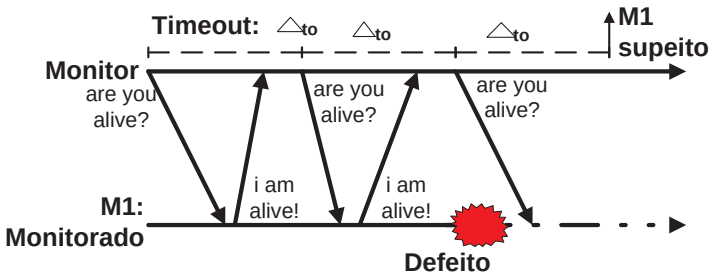


Figura 3.2. Modelo de monitoramento *Pull* (Δ_{to}:tempo estimado para chegada de um *I am alive!*)

No modelo *Push*, por outro lado, o componente monitor espontaneamente envia seu estado atual (ver figura 3.3). Baseado no intervalo entre chegada das mensagens, o detector deve estimar o instante de chegada da próxima mensagem de detecção. Caso a mensagem não chegue dentro do intervalo esperado, o detector suspeita da falha do componente.

Uma vez que as estimativas são baseadas no relógio local do componente monitor, utilizando o modelo *Pull*, o detector consegue calcular mais facilmente os *timeouts*, além de poder controlar melhor o ritmo do monitoramento. No modelo *Push*, entretanto, o

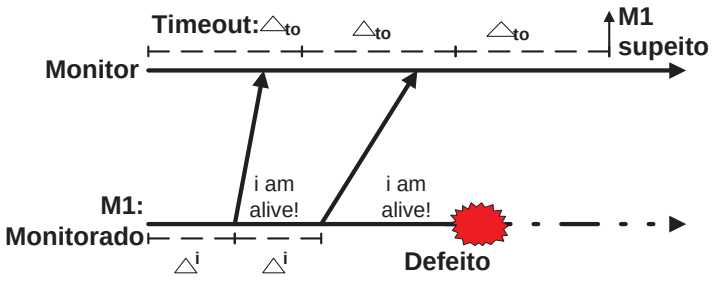


Figura 3.3. Modelo de monitoramento *Push* (Δ^i:período de emissão de *I am alive!*)

número de mensagens de detecção trocadas entre componente monitorado e detector de defeitos é menor e, portanto, consome menos recursos do canal de comunicação.

3.2.2 Adaptabilidade do Detector de Defeitos

As abordagens de detecção de defeitos apresentadas nesta seção utilizam o modelo de monitoramento *Push*, baseado em mensagens de *heartbeats* (AGUILERA; CHEN; TOUEG, 1997), entretanto, as questões aqui apontadas podem facilmente ser estendidas para o modelo de monitoramento *Pull*.

A fim de facilitar a discussão a seguir, considera-se a existência de dois componentes p e q, em que o componente q possui um módulo detector de defeitos embutido e monitora falhas do componente p. A cada Δ^i unidades de tempo, p envia para q uma mensagem, seqüencialmente assinalada e denotada por *heartbeat*, informando que está funcionando corretamente. Além disso, os marcos temporais, tais como os instantes de envio e recebimento de mensagens, são analisados do ponto de vista de um relógio global e independente dos relógios dos componentes de p e q. Entretanto, para simplificar a notação usada, a referência a esse tempo global será omitida.

A cada *heartbeat* assinalado por k recebido (m_k^{hb}), q calcula o intervalo de tempo (Δ^{to}) necessário para a chegada do próximo *heartbeat* (m_{k+1}^{hb}). Se m_{k+1}^{hb} não chega dentro de Δ^{to} unidades de tempo, q coloca p em sua lista de suspeitos. Por outro lado, caso q receba

um *heartbeat* com um número seqüencial igual ou superior ao do *heartbeat* esperado, q remove p de sua lista de suspeitos.

O componente p envia mensagens m^{hb} em instantes denotados por σ, dessa forma: m_k^{hb} é enviada no instante σ_k, m_{k+1}^{hb} em σ_{k+1}, m_{k+2}^{hb} em σ_{k+2} e assim sucessivamente (ver figura 3.4). Para quaisquer dois instantes consecutivos σ_k e σ_{k+1}, tem-se:

$$\sigma_{k+1} - \sigma_k = \Delta^i \tag{3.1}$$

Os instantes de chegada das mensagens m^{hb} são denotados por A, ou seja: m_k^{hb} chega em A_k, m_{k+1}^{hb} em A_{k+1}, m_{k+2}^{hb} em A_{k+2} e assim por diante(ver figura 3.4). Sendo *delay* o tempo necessário para a viagem de uma mensagem m_k^{hb}, logo:

$$A_k = \sigma_k + delay \tag{3.2}$$

e,

$$\Delta A^{delay} = A_{k+1} - A_k = \Delta^i \tag{3.3}$$

ΔA^{delay} representa o intervalo entre as chegadas das mensagens m_k e m_{k+1}.

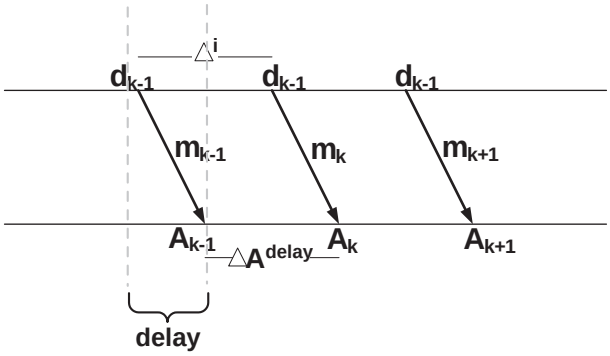


Figura 3.4. Ambiente com atraso constante.

Como pode ser visto na figura 3.4, cada *heartbeat* enviado por p, chega em q envelhe-

cido de pelo menos *delay* unidades de tempo.

Em um ambiente onde não há variação no atraso nem perda de mensagens no subsistema de comunicação, os *heartbeats* chegam em q espaçados entre si de exatamente Δ^i unidades de tempo. Se p envia m_k^{hb} em σ_k e falha em seguida, q receberá m_k^{hb} e só suspeitará da falha do componente p quando não receber m_{k+1}^{hb}. Para o componente q, entretanto, p pode ter falhado em qualquer instante de tempo entre σ_k e σ_{k+1}. Desse modo, o menor intervalo de tempo no qual q pode começar a suspeitar da falha de p com segurança não pode ser menor que $delay + \Delta^i$. Portanto, o tempo mínimo de detecção é:

$$T_D^{min} = delay + \Delta^i \tag{3.4}$$

Em que T_D^{min} representa o tempo mínimo para uma detecção segura de falha.

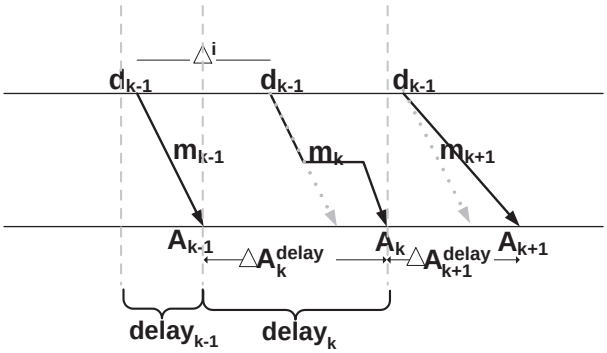


Figura 3.5. Ambiente com atraso variado.

Se variações no atraso são consideradas, cada mensagem m_k^{hb} terá um $delay_k$ associado (ver figura 3.5). Como $A_k = \sigma_k + delay_k$, observa-se que:

$$A_{k+1} - A_k = \Delta^i + (delay_{k+1} - delay_k) \tag{3.5}$$

ou,

$$\Delta A_{k+1}^{delay} = \Delta^i + (delay_{k+1} - delay_k) \tag{3.6}$$

Referencia-se aqui por ΔA^{jitter} a variação observada no intervalo de tempo entre chegadas das mensagens de *heartbeat*[1], em que:

$$\Delta A^{jitter}_{k+1} = \Delta A^{delay}_{k+1} - \Delta A^{delay}_{k} \tag{3.7}$$

Quando os tempos de viagem das mensagens m^{hb} não são conhecidos e não há relógios sincronizados, as estimativas do detector de defeitos não são precisas e não é possível estimar o tempo de detecção com exatidão. Sendo EA_k a estimativa realizada pelo detector para o instante de chegada de um *heartbeat* m^{hb}_k, quanto mais próximo EA_k estiver de A_k menor será o tempo de detecção. A relação $EA_k \geq A_k$ deve ser satisfeita para que o detector evite falsas suspeitas. Assim, a relação entre EA e A é uma indicação do desempenho do detector em termos de tempo de detecção.

Uma vez que variações extras no atraso podem provocar sub-estimativas, utiliza-se uma margem de segurança (α) que compense variações não previstas no atraso da rede (JACOBSON, 1988). Desta forma, o marco estimado no tempo FP (*Freshness Point* (CHEN; TOUEG; AGUILERA, 2002)) para chegada do próximo *heartbeat* é definido por:

$$FP_{k+1} = EA_{k+1} + \alpha_{k+1} \tag{3.8}$$

Portanto, a adaptabilidade do detector consiste em ajustar EA e α (NUNES; JANSCH-PôRTO, 2004). Sendo assim, recebido m^{hb}_k em A_k, quanto mais precisa a estimativa de EA_{k+1} e α_{k+1} mais próximo FP_{k+1} estará de A_{k+1}.

Nas subseções seguintes serão abordados os algoritmos de adaptação utilizados pelos detectores de defeitos analisados durante o desenvolvimento desta dissertação.

[1] A variável ΔA^{jitter} não deve ser confundida com o conceito de *jitter* usado na área de sistemas de tempo real ou com o conceito usado na área de redes de computadores. Aqui denomina-se ΔA^{jitter} qualquer variação no atraso entre os instantes de chegada das mensagens e não a máxima variação possível.

3.2.2.1 O Algoritmo de Jacobson (1988) No protocolo de comunicação TCP (TANENBAUM, 2003), a cada mensagem enviada, o componente transmissor estipula um intervalo de tempo para a chegada de uma mensagem de confirmação oriunda do componente para o qual a mensagem transmitida foi destinada. Caso a confirmação não chegue dentro do intervalo de tempo estipulado, o componente transmissor realiza uma retransmissão da mensagem. Jacobson (1988) sugere um algoritmo para evitar que retransmissões desnecessárias de mensagens colaborem para o congestionamento em redes *TCP*. Tal algoritmo é descrito a seguir. Sejam rtt_k^M e rtt_k^C o atraso (*RTT*, *Round Trip Time*) medido e estimado no instante k, respectivamente. Uma estimativa do próximo atraso rtt_{k+1}^C pode ser feita por:

$$error_k = rtt_k^M - rtt_k^C \tag{3.9}$$

$$rtt_{k+1}^C = rtt_k^C + \mu.error_k, \tag{3.10}$$

onde *error* e μ representam, respectivamente, o erro medido na última estimativa e a confiança atribuída à estimativa realizada para o erro.

Considerando a existência de variações adicionais no atraso da rede, a nova estimativa para o atraso (dita Δ_{k+1}^{to}) na chegada de uma mensagem pode ser dado pelas equações 3.11 e 3.12.

$$var_{k+1} = var_k - \mu.\left(|error_k| - var_k\right) \tag{3.11}$$

$$\Delta_{k+1}^{to} = \beta.rtt_{k+1}^C - \phi.var_{k+1} \tag{3.12}$$

Δ_{k+1}^{to} representa o atraso estimado para a chegada do próximo *I am alive!* e var_k denota a variação adicional no atraso da rede de comunicação medida no instante k. ϕ

e β são, respectivamente, a confiança na variação do atraso e na estimativa do atraso calculado.

Para a implementação de um detector usando o algoritmo de adaptação de (JACOBSON, 1988), a variável rtt_k^C equivale a EA_k, enquanto rtt_k^M corresponde a A_k.

Uma vez calculado o atraso estimado, sendo A_k o instante de chegada da última mensagem, presume-se que a próxima mensagem chegará em:

$$EA_{k+1} = A_k + \Delta_{k+1}^{to} \tag{3.13}$$

Como poderá ser visto na subseção a seguir, o algoritmo de Jacobson (1988) também foi usado por Bertier, Marin e Sens (2002) para a estimativa da margem de segurança α.

3.2.2.2 O Algoritmo de Bertier, Marin e Sens (2002) Bertier, Marin e Sens (2002) propuseram um detector adaptativo (AFD, *Adaptative Failure Detector*) implementado em duas camadas de modo que a camada detecção (inferior) possa prover o serviço básico de detecção de defeitos baseado nas métricas de qualidade de serviço propostos por (CHEN; TOUEG; AGUILERA, 2002), enquanto a camada de adaptação (superior) ajusta a qualidade de serviço prestado pela camada inferior, de acordo com as necessidades da aplicação.

A implementação do serviço de detecção de defeitos na camada de detecção é baseada na estimativa do momento de chegada da mensagem proposta por (CHEN; TOUEG; AGUILERA, 2002). Tal implementação traz algumas modificações em relação à proposta inicial, no que diz respeito à estimativa da margem de segurança e à estimativa do tempo de chegada da mensagem durante a chegada das n primeiras mensagens.

O instante de tempo limite esperado para chegada de uma mensagem m_{k+1}^{hb} é definido por:

$$FP_{k+1} = EA_{k+1} + \alpha_{k+1} \tag{3.14}$$

EA_{k+1} e α_{k+1} representam, respectivamente, uma estimativa para o instante de chegada e a margem de segurança para a próxima mensagem de *heartbeat* m^{hb}_{k+1}. Sendo Δ^i o período entre a emissão de dois *heartbeats* e A_k o momento do recebimento da mensagem m^{hb}_k, a estimativa EA_{k+1} das n primeiras mensagens a serem recebidas é calculada da forma a seguir (BERTIER; MARIN; SENS, 2002):

$$U_{k+1} = \frac{A_k}{k+1}.\frac{k.U_k}{k+1} \tag{3.15}$$

em que $U_1 = A_0$.

$$EA_{k+1} = U_{k+1} + \frac{k+1}{2}.\Delta^i \tag{3.16}$$

Após o recebimento de n mensagens, os autores sugerem a estimativa apresentada a seguir:

$$V_{k+1} = \frac{1}{n}\left[A_k - A_{k-n-1}\right] \tag{3.17}$$

$$EA_{k+1} \approx V_{k+1} + (k+1) * \Delta^i \tag{3.18}$$

Na equação 3.18, $(k+1) * \Delta^i$ representa o momento de envio da mensagem m_{k+1}, enquanto V_{k+1} (equação 3.17) é uma estimativa do tempo médio de viagem da mensagem de *heartbeat*.

Considerando o erro ($error_k$) obtido no cálculo da última estimativa, o instante (A_k) de recebimento da última mensagem, a estimativa para o atraso ($delay_k$) chegadas de *heartbeats* e var_k uma estimativa da variação no cálculo do erro, os autores de (BERTIER; MARIN; SENS, 2002) calculam a margem de segurança com base na estimativa de Jacobson (1988), da seguinte forma:

$$error_k = A_k - EA_k - delay_k \tag{3.19}$$

$$delay_{k+1} = delay_k - \mu.error_k \tag{3.20}$$

$$var_{k+1} = var_k - \mu.\left(|error_k| - var_k\right) \tag{3.21}$$

$$\alpha_{k+1} = \beta.delay_{k+1} - \phi.var_{k+1} \tag{3.22}$$

3.2.2.3 O Algoritmo Baseado em Redes Neurais de Macêdo e Lima (2004)

Macêdo e Lima (2004) propuseram um detector de defeitos (*ANN-FD, Artificial Neural Network Based Failure Detector*) baseado na utilização de Rede Neural Artificial[2] (**RNA**). Tal detector utiliza como parâmetro de entrada para a rede neural variáveis disponibilizadas pelo protocolo **SNMP** (***S**imple **N**etwork **M**anagement **P**rotocol*)[3]. Tais variáveis fornecem uma visão, a cada instante, das características do tráfego no sistema de comunicação.

O **ANN-FD** utiliza uma rede neural do tipo *Feedforward Multilayer Perceptron*(HAYKIN, 1994) com quatro camadas organizadas da forma a seguir. Cada camada é formada por neurônios artificiais, os quais têm sua saída conectada às entradas dos neurônios da camada seguinte. As conexões entre neurônios (conexões sinápticas) possuem pesos que definem o grau de excitação que a saída de um neurônio provocará no neurônio da camada posterior. Não existem conexões entre neurônios de uma mesma camada. Na camada de retina (camada entrada) são utilizados seis neurônios que recebem em suas entradas os valores das variáveis da **MIB** (***M**anagement **I**nformation **B**ase*) que são usadas para

[2]Uma discussão mais detalhada sobre as redes neurais será apresentada na seção 4.1.
[3]Ver (STALLINGS, 1999) para maiores detalhes sobre este protocolo.

caracterizar o comportamento do tráfego no sistema de comunicação. Tais variáveis são:

- $IfnUcastPkts$, número de pacotes de subredes unicast entregues;
- $IfOutUcastPkts$, número total de pacotes transmitidos (incluindo os pacotes descartados);
- $IfOutQLen$, tamanho da fila de pacotes de saída;
- $udpInDatagrams$, número de datagramas *UDP* entregues a usuários *UDP*;
- $udpOutDatagrams$, número total de datagramas *UDP* enviados de uma entidade;
- $udpNoPorts$, número total de datagramas *UDP* recebidos que não estão destinados a uma porta que não é utilizada por qualquer aplicação;

As camadas intermediárias (segunda e terceira camadas) são compostas por nove e quatro neurônios, respectivamente. A camada de saída (quarta camada) é composta por apenas um neurônio e é responsável por apresentar o resultado do processamento realizado pela *RNA*.

Antes da sua utilização na predição dos momentos de chegada das mensagens de *heartbeat*, a *RNA* deve ser treinada; tal processo é descrito a seguir. O *ANN-FD* coleta os valores das variáveis da *MIB* e em seguida estima um valor para o instante de chegada do próximo *heartbeat* m_{k+1}^{hb}. O valor estimado é comparado com o instante real, e o erro (diferença entre a estimativa e valor real) é utilizado para ajustar os pesos das conexões sinápticas. O procedimento de ajuste dos pesos sinápticos utiliza o algoritmo de retropropagação (*Backpropagation*). A fase de treinamento é realizada até que o grau de aprendizado da rede seja considerado satisfatório.

Uma vez treinada, a *RNA* pode ser usada para, a partir do padrão de carga no sistema de comunicação fornecido pelas variáveis da *MIB*, predizer o instante de chegada

do próximo *heartbeat* m_{k+1}^{hb}. O tempo máximo esperado para chegada de m_{k+1} é definido conforme a equação 3.23.

$$FP_{k+1} = EA_{k+1} + \alpha \tag{3.23}$$

A análise realizada pelos autores demonstra que o *ANN-FD* apresenta um melhor desempenho em relação ao detector adaptativo de Bertier, Marin e Sens (2002) em condições de sobrecarga do sistema de comunicação. Em condições de carga moderada, entretanto, o *AFD* se mostrou mais eficiente.

3.2.3 Qualidade de Serviço em Detecção de Defeitos

A detecção de defeitos em ambiente distribuído está diretamente ligada à velocidade dos processos e aos atrasos no canal de comunicação. Sendo assim, se existem variações não determinísticas no tempo de processamento ou no tempo de transmissão das mensagens, o detector pode cometer erros em sua detecção. Por conta disso, tais detectores são ditos não confiáveis, uma vez que podem suspeitar de componentes que efetivamente nunca falharam (ditos componentes corretos) ou ainda não suspeitar de componentes que efetivamente falharam.

Chandra e Toueg (1996) propuseram uma classificação para os detectores de defeitos, as quais variam de acordo com duas propriedades básicas: *Completude* (*Completeness*), referindo-se à capacidade dos detectores de suspeitar, em algum momento no futuro, de componentes que efetivamente falharam; e *Precisão* (*Accuracy*), relacionada à capacidade do detector em evitar falsas suspeitas.

A classificação proposta por Chandra e Toueg (1996), entretanto, é baseada em comportamentos temporais futuros não quantificáveis, sendo, portanto, não apropriada para algumas aplicações críticas que necessitem de garantias de detecção mais fortes. Tais aplicações precisam de métricas que possam mensurar a qualidade de serviço prestada

pelos mecanismos de detecção de defeitos(CHEN; TOUEG; AGUILERA, 2002; HERMANT; LANN, 2002). A exemplo disso, tem-se os sistemas críticos de tempo real, os quais necessitam de mecanismos que assegurem um certo grau de previsibilidade para que possam atender às restrições temporais impostas pela aplicação.

Chen (2000) propôs métricas probabilísticas de Qualidade de Serviço (*QoS, Quality of Service*) para medir a capacidade do detector em prevenir falsas suspeitas (*precisão*) e a rapidez com a qual o mesmo detecta a falha dos dispositivos (*velocidade*). Nesse trabalho, Chen (2000) considera a implementação de um serviço de detecção baseado no modelo de monitoramento *Push*, no qual cada processo monitor é dotado por um detector de defeitos com saída binária. A saída de cada detector de defeitos em um tempo t é S (suspeito) ou T (correto). Uma transição *S-Transition* ocorre quando a saída do detector muda de T para S, assim como, uma transição *T-Transition* ocorre quando a saída do detector muda de S para T.

As métricas, propostas por Chen (2000), estão divididas em dois grupos:

- **Métricas Primárias**, as quais não podem ser deduzidas de quaisquer outras métricas, mas a partir das quais se deduz as demais. Essas métricas são:

  - Tempo de detecção (T_D, *Detection Time*): Tempo necessário para que o dispositivo monitor (referenciado na figura 3.6 por q) suspeite que o dispositivo monitorado (referenciado na figura 3.6 por p) falhou; tal tempo é contado a partir do instante no qual o dispositivo monitorado efetivamente falhou.

    A figura 3.6 apresenta um cenário no qual o dispositivo p falha imediatamente após ter enviado uma mensagem para o dispositivo q. Essa mensagem, entretanto, chega atrasada em q, fazendo com que a saída do detector nele instalado oscile entre falho e correto. Por outro lado, como a próxima mensagem de p não chegará, q passará a suspeitar de p definitivamente.

  - Intervalo entre falsas suspeitas (T_{MR}, *Mistake Recurrence Time*): Tempo entre

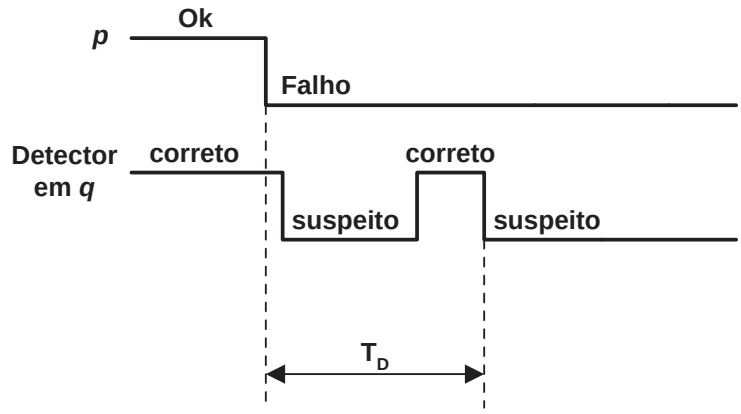


Figura 3.6. Possíveis alterações na saída do detector até que a falha seja percebida

duas falsas suspeitas consecutivas (ver figura 3.7).

– Duração das falsas suspeitas (T_M, *Mistake duration*): Tempo que o detector leva para corrigir uma falsa suspeita (ver figura 3.7).

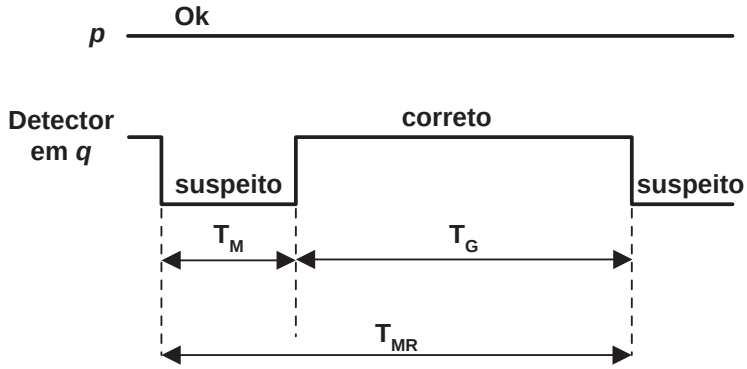


Figura 3.7. Exemplo das métricas primárias T_M e T_{MR}, e da métrica secundária T_G

- **Métricas Derivadas**, métricas secundárias obtidas a partir das métricas primárias:

  – Taxa de falsas suspeitas (λ_M, *Avarage Mistake Rate*): Indica a taxa de falsas suspeitas cometidas pelo detector.

  – Probabilidade de consulta correta (P_A, *Query Accurace Probability*): Indica a probabilidade que um detector tem em produzir a saída correta em um instante qualquer.

– Período Bom (T_G, *Good Period Duration*): Indica o tempo médio em que o detector de defeito produz a saída correta (ver figura 3.7).

– Intervalos entre períodos bons (T_{FG}, *Foward Good Period Duration*): Indica o intervalo médio de tempo entre dois períodos bons T_G.

3.2.3.1 Discutindo a Relação Entre as Métricas Sejam $Pr(A)$, $E(x)$, $E(xk)$, $V(x)$ a probabilidade de um evento A ocorrer, o valor esperado de x, o momento k e a variância de x, respectivamente, as relações entre as métricas podem ser mapeadas com base no seguinte teorema (CHEN; TOUEG; AGUILERA, 2002):

Teorema 3.2.1 (Da relação entre as métricas de *QoS*) *Para qualquer detector de defeito ergótico*[4]*, os seguintes resultados se mantém:*

i) $T_G = T_{MR} - T_M$.

ii) Se $0 < E(T_{MR}) < \infty$*, então* $\sigma_M = \frac{1}{E(T_{MR})}$ *e* $P_A = \frac{E(T_G)}{E(T_{MR})}$.

iii) Se $0 < E(T_{MR}) < \infty$ *e* $E(T_G) = 0$*, então* T_{FG} *é sempre* 0. *Se* $0 < E(T_{MR}) < \infty$ *e* $E(T_G) \neq 0$*, então:*

a. $\forall x \in [0, \infty)$, $Pr(T_{FG} \leq x)$, $\frac{1}{E(T_G)} \int_0^x Pr(T_G > y)dy$.

b. $E(T_G k) = \frac{E(T_G k+1)}{[(k+1)E(T_G)]}$, *em particular* $E(T_{FG}) = [\frac{1+V(T_G)}{E(T_G)^2}]E(T_G)/2$

Baseado no teorema 3.2.1, algumas considerações podem ser tecidas:

- Naturalmente, os bons períodos acontecem nos intervalos em que o detector não comete falsas suspeitas.
- A taxa de falsas suspeitas cometidas pelo detector de defeitos representa o número de falsas suspeitas por unidade de tempo.

[4]Um detector é ergótico se em rodadas livres de defeitos, tal detector lentamente *esquece* o histórico de suas saídas; desta forma, seu comportamento pode depender apenas de seu comportamento mais recente.

- A probabilidade do detector produzir um valor correto é a relação entre o valor esperado para períodos bons e o valor esperado para intervalo entre falsas suspeitas.
- Se existe a possibilidade do detector não cometer erros, então o valor esperado para o próximo período bom será a relação entre o somatório das probabilidades de todos os possíveis intervalos de períodos bons e valor esperado para o período bom.

3.2.3.2 Sintonia de Detectores de Defeitos com QoS O processo de sintonia do detector de defeitos deve determinar, através do comportamento probabilístico do canal de comunicação[5], a taxa de emissão de mensagens de monitoramento (Δ^i) e a margem de segurança para atrasos extras na comunicação(α).

A sintonia do detector de defeitos está sujeita ao nível de qualidade de serviço desejado; assim, deve-se definir uma tupla (T_D^U, T_{MR}^L, T_M^U), representando, respectivamente, o tempo máximo de detecção, o intervalo mínimo entre falsas suspeitas recorrentes e o tempo máximo para duração de uma falsa suspeita. Esta tupla especifica a *QoS* desejada para o detector. Para essa tupla, as seguintes relações devem ser válidas:

$$T_D < T_D^U \tag{3.24}$$

$$E(T_{MR}) \geq T_{MR}^L \tag{3.25}$$

$$E(T_M) \leq T_M^U \tag{3.26}$$

O processo de sintonia é um problema de pesquisa operacional, no qual se deve encontrar o máximo Δ^i sujeito às restrições impostas pelas relações em 3.24, 3.25 e 3.26.

De posse de Δ^i e T_D^U, pode-se facilmente encontrar uma margem de segurança inicial α_0. Chen, Toueg e Aguilera (2002) sugerem o procedimento de configuração a seguir.

[5]Probabilidade de perdas p_L de mensagens de monitoramento dentro do tempo esperado $Pr(D \leq x)$.

(a) Calcular

$$q_0^{'} = (1 - p_L)Pr(D < T_D^U) \tag{3.27}$$

em que $q_0^{'}$ representa a probabilidade de uma mensagem de monitoramento m_k chegar dentro do intervalo $[k, k+1)$. Assim, é possível calcular a máxima taxa de emissão de mensagens de monitoramento tolerada.

$$\Delta_{max}^i = q_0^{'} T_M^U \tag{3.28}$$

Se $\Delta_{max}^i = 0$, então a QoS desejada não pode ser encontrada, senão o passo **(b)** deve ser realizado.

(b) Encontrar o maior $\Delta^i < \Delta_{max}^i$ tal que $f(\Delta^i) \geq T_{MR}^L$, em que

$$f(\Delta^i) = \Delta^i \left\{ q_0^{'} \prod_{j=1}^{\left\lceil \frac{T_D^U}{\Delta^i} - 1 \right\rceil} \left[p_L + (1 + p_L)Pr(D > T_D^U - j\Delta^i) \right] \right\}^{-1} \tag{3.29}$$

(c) Por fim, obtem-se $\alpha_0 = T_D^U - \Delta^i$

3.3 CONSIDERAÇÕES FINAIS

A rapidez e a confiabilidade do detector de defeitos é extremamente dependente do comportamento dos atrasos de comunicação e computação. Sendo assim, nos cenários em que as hipóteses consideradas durante a construção do sistema podem ser violadas ou em que o ambiente de comunicação ou de computação podem apresentar atrasos não determinísticos, dotar o detector de defeitos da capacidade de adaptação é uma característica importante para promover uma maior rapidez e confiabilidade na detecção

de falhas.

Na construção de sistemas de tempo real críticos, além da capacidade de adaptação, é necessário que a qualidade do serviço prestado pelo detector seja avaliada. As métricas de qualidade de serviço, propostas por Chen (2000), se mostram uma ferramenta bastante interessante para averigüar a precisão, a confiabilidade e a velocidade do detector na detecção de falhas. A partir de tais métricas se pode avaliar com mais simplicidade o potencial de adaptação do detector e sua coesão com as necessidades da aplicação de tempo real.

CAPÍTULO 4

REDES NEURAIS ARTIFICIAIS

Neste capítulo são discutidos conceitos teóricos básicos sobre Redes Neurais Artificiais. Serão comentados alguns aspectos básicos, mas será dado um enfoque maior nas Redes Neurais Artificiais Multicamada do tipo *Feedforward* com aprendizado supervisionado, um dos modelos mais utilizados na predição e aproximação universal de funções. Por conta disso, escolheu-se tal modelo de Rede Neural para a implementação da abordagem de detecção de defeitos proposta no capítulo 5.

4.1 INTRODUÇÃO

Uma **R**ede **N**eural **A**rtificial (*RNA*) é uma das técnicas da inteligência artificial utilizada para solucionar problemas relacionados a reconhecimento de padrões, predição, otimização, controle entre outros(JAIN; MAO; MOHIUDDIN, 1996).

A *RNA* é representada através de um conjunto de nós (neurônios) interconectados e capazes de aprender, armazenar nas conexões (ou *sinapses*) e reproduzir características específicas de um determinado domínio de problema. Segundo Haykin (1994), uma *RNA* pode ser definida como uma estrutura massivamente paralela e distribuída, que tem por característica natural armazenar conhecimento e fazê-lo disponível para uso. Essa definição remete-se a uma comparação entre a *RNA* e a rede neurológica humana, no que diz respeito à capacidade de adquirir conhecimento através de um processo de aprendizado e armazenar tal conhecimento através das interconexões (sinapses) entre estruturas simples denominadas neurônios.

Do ponto de vista prático, entretanto, uma Rede Neural Artificial pode ser vista como uma representação de funções matemáticas utilizando elementos computacionais aritméticos simples, sem, no entanto, possuir relações diretas com a modelagem do sistema nervoso (BITTENCOURT, 2001).

Bittencourt (2001) cita algumas características interessantes das redes neurais quando aplicadas à solução de problemas, dentre as quais se pode destacar as seguintes:

- Capacidade de *aprender* através de exemplos e de generalizar o aprendizado de maneira a reconhecer instâncias similares que nunca lhes foram apresentadas antes.
- Bom desempenho em tarefas mal definidas, em que falta o conhecimento explícito da solução.
- Em geral, não requer conhecimento a respeito de eventuais modelos matemáticos dos domínios da aplicação.

4.2 ARQUITETURA BÁSICA DE UMA RNA

4.2.1 Modelo do Neurônio Artificial

A estrutura da *RNA* está fundamentada em neurônios e em suas conexões; assim, definir o modelo matemático do neurônio artificial é um ponto importante para entender o funcionamento da Rede Neural.

De acordo com Haykin (1994), um neurônio é uma unidade de processamento de informação, no qual se pode identificar três elementos básicos (ver figura 4.1), descritos a seguir:

i) Um conjunto de sinapses ou conexões, as quais possuem pesos que caracterizam a força de tal conexão. Assim, um valor x_j na entrada de uma sinapse j conectada

a um neurônio k é multiplicado por um peso sináptico w_{kj}. O peso w_{kj} é dito excitador se é positivo ou inibidor em caso contrário.

ii) Um somador para agregar as entradas ponderadas por seus respectivos pesos sinápticos. A variável u_k representa essa combinação linear obtida na saída do somador do neurônio k.

iii) Uma função de ativação (f) responsável por delimitar a intensidade do valor y de saída do neurônio (y_k representa a saída do neurônio k). Essa função pode ainda ser controlada ou auxiliada por um limiar de ativação denominado *bias* (b).

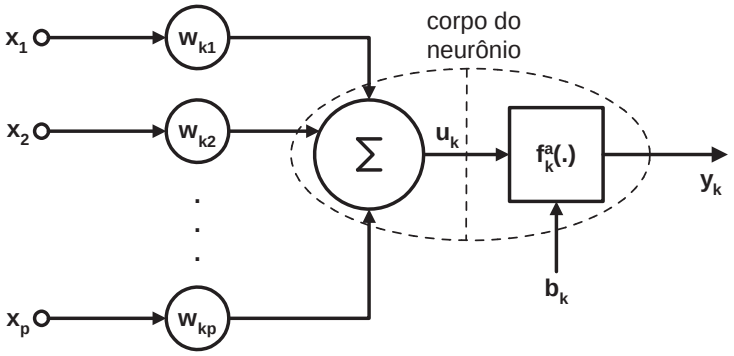


Figura 4.1. Modelo conceitual de um neurônio artificial

Um neurônio k qualquer pode ser definido matematicamente pelo seguinte par de equações 4.1 e 4.2 (HAYKIN, 1994):

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{4.1}$$

e,

$$y_k = f_k\left(u_k - b_k\right) \tag{4.2}$$

em que x_1, x_2, ..., x_m são as entradas do neurônio k, enquanto w_{k1}, w_{k2}, ..., w_{km} são os seus pesos sinápticos. u_k é a saída da combinação linear das entradas. y_k, f_k e b_k

representam, respectivamente, a saída, a função de ativação e o limiar de ativação do neurônio k.

A equação 4.1 define como as entradas são combinadas e a equação 4.2 define como a saída é processada.

4.2.2 Arquitetura de RNA Feedforward Multicamada

4.2.2.1 *Feedforward* Simples A arquitetura da Rede Neural Artificial do tipo *Feedforward* é composta por camadas, as quais são compostas por um conjunto de neurônios. Uma *RNA Feedforward* simples contém apenas dois tipos de camadas básicas. A primeira é a camada de entrada, a qual possui o conjunto de neurônios responsáveis por permitir a entrada de dados na Rede Neural. Essa também é conhecida por camada de retina, e os seus neurônios não realizam qualquer tipo de processamento. A segunda é a camada de saída, a qual contém o conjunto de neurônios responsáveis por realizar o processamento. Os neurônios da camada de entrada são conectados aos neurônios da camada de saída, mas o contrário não é verdade (HAYKIN, 1994).

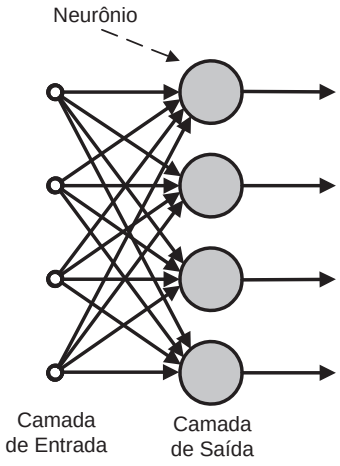


Figura 4.2. Exemplo de *RNA Feedforward* simples com 4 neurônios em cada camada

A figura 4.2, adaptada de (HAYKIN, 1994), apresenta um exemplo de *RNA Feedforward*

simples com quatro neurônios em cada camada. Como pode ser visto nessa figura, a saída dos neurônios da camada de entrada é interconectada a todos os neurônios da camada seguinte, a camada de saída.

4.2.2.2 *Feedforward* Multicamada Uma Rede Neural *Feedforward* Multicamada acrescenta à *Feedforward* simples uma ou mais camadas ocultas, também chamadas de camadas intermediárias. Nessa categoria de rede, os neurônios de um camada são conectados aos neurônios da camada subseqüente.

A camada intermediária é composta por neurônios que podem realizar processamento. Dessa forma, a adição de camadas intermediárias aumenta o poder de generalização da rede, uma vez que a mesma pode, agora, armazenar mais conhecimento. Todavia, essa adição de camadas torna o processo de aquisição do conhecimento mais demorado(JANG; SUN; MIZUTANI, 1997).

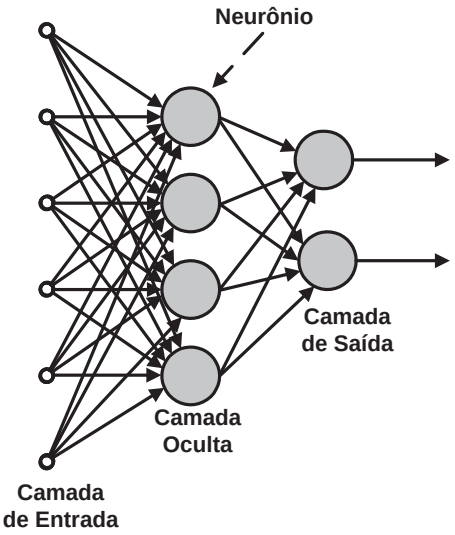


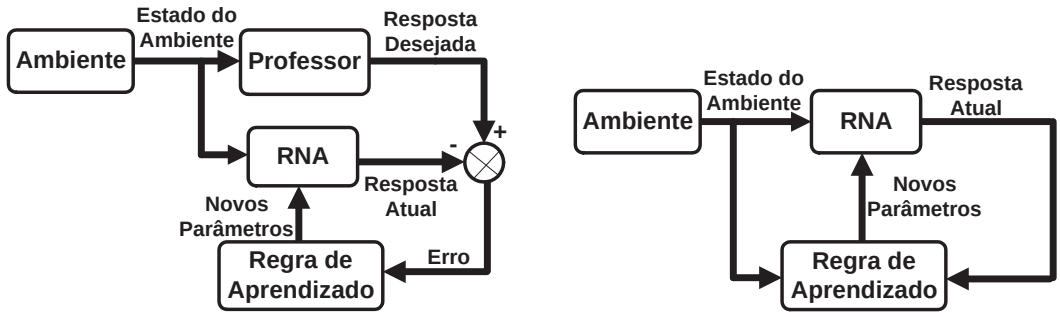
Figura 4.3. Exemplo de *RNA Feedforward* multicamada 6-4-2

A figura 4.3, adaptada de (HAYKIN, 1994), apresenta um exemplo de uma RNA Multicamada com seis neurônios na camada de entrada, uma camada oculta com quatro neurônios e uma camada de saída com dois neurônios.

4.2.3 Processo de Aprendizado Supervisionado

O processo pelo qual a *RNA* realiza a aquisição de conhecimento é denominado de treinamento ou aprendizado. De modo pragmático, o processo de aprendizado consiste em um ajuste paramétrico da rede.

Em geral, o processo de aprendizado pode ser dividido em supervisionado e não supervisionado. O aprendizado supervisionado é caracterizado pela existência de um agente externo (professor) que supervisiona e guia o aprendizado da rede, apresentando exemplos confiáveis a serem assimilados (figura 4.4(a)). Esse método é realizado antes que a rede seja posta de fato em funcionamento. No aprendizado não supervisionado (figura 4.4(b)), por outro lado, as redes neurais aprendem com os dados que lhes são apresentados durante a sua execução (*on-line*).



(a) (b)

Figura 4.4. 4.4(a) rede supervisionada, 4.4(b) rede não supervisionada

Diferente do aprendizado supervisionado, o aprendizado não supervisionado não contém uma entidade que represente o papel de professor.

4.2.3.1 Algoritmo de Retropropagação Existem diferentes regras que podem ser usadas no processo de aprendizado(HAYKIN, 1994). Essas regras se diferenciam pelo modo como os parâmetros da rede (pesos sinápticos, limiar etc) são atualizados.

Uma regra de aprendizado simples e bastante utilizada no treinamento de *RNA Fe-*

edforward multicamada é a Retropropagação (*Backpropagation*). Em redes neurais artificiais do tipo *Feedforward* com Retropropagação, o aprendizado se dá em três etapas:

a. O padrão π, pertencente ao conjunto de padrões de treinamento Π, é apresentado à camada de entrada, sendo repassado e processado pelos neurônios subseqüentes. Assim, os neurônios de uma camada m enviam os sinais de saída (resultados) dos seus processamentos para todos os neurônios da camada subseqüente $m + 1$.

b. O supervisor verifica o resultado apresentado na saída da RNA e indica a mesma diferença entre o padrão obtido y^π e o padrão desejado d^π como resultado de saída, calculando assim o desvio (ou erro e^π) entre esses padrões.

c. Os pesos das interconexões entre os neurônios são reajustados em função do erro encontrado. Com isso, deseja-se que o padrão obtido y^π seja o mais próximo possível do padrão desejado d^π. Para tanto, deve-se minimizar o erro de modo que a distância entre y^π e d^π esteja dentro do mínimo aceitável.

A regra de Retropropagação realiza a atualização dos pesos sinápticos de acordo com os passos a seguir (HAYKIN, 1994).

Calcula-se o erro $e_j^\pi(n)$ da saída do neurônio j para o padrão π por:

$$e_j^\pi = d_j^\pi - y_j^\pi \tag{4.3}$$

O valor instantâneo do erro quadrático para o neurônio j é $\frac{1}{2}\left[e_j^\pi\right]^2$. Aplicando essa regra à camada de saída, pode-se calcular o erro instantâneo da rede em relação a π:

$$\xi^\pi = \frac{1}{2}\sum_{j\in S}\left[e_j^\pi\right]^2 \tag{4.4}$$

O conjunto S inclui todos os neurônios da camada de saída.

Sendo N o número de padrões de entrada contidos no conjunto de treinamento Π, pode-se medir o desempenho do treinamento da rede usando:

$$\xi = \frac{1}{N} \sum_{\pi \in \Pi} [\xi^{\pi}]^2 \tag{4.5}$$

O objetivo do treinamento é ajustar os parâmetros da rede de modo a minimizar ξ, deixando-os dentro do limite aceitável, ou seja, $\xi \leq \epsilon$ (o pior desempenho aceitável para rede).

Conforme apresentado anteriormente, o valor na saída do neurônio j pode ser calculado por:

$$y_j^{\pi} = f_j\left(u_j^{\pi}\right) \tag{4.6}$$

em que,

$$u_j^{\pi} = \sum_{i=1}^{p} w_{ji}^{\pi} y_i^{\pi} \tag{4.7}$$

p é o número total de entradas sem levar em consideração o limiar de ativação b_j do neurônio j.

O algoritmo de Retropropagação aplica uma correção Δw_{ji}^{π} ao peso sináptico w_{ji}^{π}. Essa correção é proporcional ao gradiente do erro instantâneo $\partial \xi^{\pi} / \partial w_{ji}^{\pi}$, o qual pode ser calculado como segue (HAYKIN, 1994):

$$\frac{\partial \xi^{\pi}}{\partial w_{ji}^{\pi}} = -e_j^{\pi} f_j'\left(u_j^{\pi}\right) y_i^{\pi} \tag{4.8}$$

Assim, a correção aplicada aos pesos sinápticos é calculada por:

$$\Delta w_{ji}^{\pi} = -\eta \frac{\partial \xi^{\pi}}{\partial w_{ji}^{\pi}} \tag{4.9}$$

ou,

$$\Delta w_{ji}^{\pi} = \eta \delta_j^{\pi} y_i^{\pi} \tag{4.10}$$

com,

$$\delta_j^{\pi} = e_j^{\pi} f_j^{'}\left(u_j^{\pi}\right) \tag{4.11}$$

δ_j^{π} é o gradiente descendente local, a saída do neurônio j e η representa a taxa de aprendizado. O sinal negativo da equação 4.9 refere-se ao sentido gradiente.

As equações acima são relativamente fáceis de derivar quando se trata da camada de saída, para a qual o padrão de saída é conhecido. No caso das camadas ocultas, entretanto, o valor de referência para a saída de seus neurônios deve ser encontrada em função do erro obtido na camada de saída (ou camada subseqüente, quando se trata de uma RNA como mais de uma camada intermediária). Assim, segundo Hecht-Nielsen (1989), o gradiente de saída para um neurônio na camada intermediaria pode ser calculado por:

$$\delta_j^{\pi} = f_j^{'}\left(u_j^{\pi}\right) \sum_{k \in K} \delta_k^{\pi} w_{kj}^{\pi} \tag{4.12}$$

K representa o conjunto de neurônios da camada subseqüente m e j refere-se a um neurônio da camada oculta $m-1$.

De forma resumida, a atualização dos pesos sinápticos em uma *RNA Feedforward* com Retropropagação é calculado como segue (HECHT-NIELSEN, 1989):

$$\Delta w_{ji}^{\pi} = \eta \delta_j^{\pi} y_i^{\pi} \tag{4.13}$$

$$\delta_j^{\pi} = \begin{cases} e_j^{\pi} f_j^{'}\left(u_j^{\pi}\right) & , \quad se\ (j \in S) \\ f_j^{'}\left(u_j^{\pi}\right) \sum_{k \in K} \delta_k^{\pi} w_{kj}^{\pi} & , \quad se\ (j \in H) \end{cases} \tag{4.14}$$

S e H são, respectivamente, o conjunto dos neurônios da camada de saída e o conjunto de neurônios de uma determinada camada oculta.

4.2.3.2 Algoritmo de Propagação Elástica A escolha da taxa de aprendizado tem um papel importante na eficiência do algoritmo de Retropropagação. Uma taxa de aprendizado muito pequena faz com que o ajuste paramétrico tenha uma convergência lenta, precisando de muitas iterações até que a rede apresente um bom desempenho ($\xi \leq \epsilon$). Por outro lado, uma taxa de aprendizado muito grande, apesar de acelerar o aprendizado, pode fazer com que a rede não consiga convergir e oscile em torno do ponto desejado (HAYKIN, 1994).

O algoritmo de *Propagação Elástica(RPROP, Resilient Propagation)*, proposto por Riedmiller e Braun (1993), é uma alternativa ao algoritmo de Retropropagação, que propõe um esquema mais eficiente para o ajuste dos pesos sinápticos (SCHIFFMANN; JOOST; WERNER, 1994).

Durante o processo de aprendizado, o *RPROP* introduz a cada peso sináptico uma taxa individual de adaptação Δ_{ji}. Essa taxa de adaptação é modificada de acordo com a derivada do erro ξ. Tal modificação segue a seguinte regra:

$$\Delta_{ji}(k) = \begin{cases} \eta^{+} * \Delta_{ji}(k-1) & , \quad se \ \frac{\partial \xi(k-1)}{\partial w_{ji}} * \frac{\partial \xi(k)}{\partial w_{ji}} > 0 \\ \eta^{-} * \Delta_{ji}(k-1) & , \quad se \ \frac{\partial \xi(k-1)}{\partial w_{ji}} * \frac{\partial \xi(k)}{\partial w_{ji}} < 0 \\ \Delta_{ji}(k-1) & , \quad se \ \frac{\partial \xi(k-1)}{\partial w_{ji}} * \frac{\partial \xi(k)}{\partial w_{ji}} = 0 \end{cases} \tag{4.15}$$

em que,

- w_{ji} representa o peso da conexão sináptica do neurônio i para o neurônio j.
- $\frac{\partial \xi}{\partial w_{ji}}$ refere-se à derivada parcial do erro.

- Δ_{ji} é o fator de correção do peso sináptico w_{ji}.

- η representa o fator de atualização de Δ_{ji}. Sendo que $\eta+$ é utilizado quando a derivada parcial correspondente ao peso w_{ji} é positiva, enquanto $\eta-$ é usado quando a mesma é negativa. η^+ e η^- devem atender à seguinte relação $0 < \eta^- < 1 < \eta^+$.

O conceito associado à regra de modificação da taxa de adaptação é apresentado a seguir. Quando a derivada parcial correspondente ao peso w_{ji} tem o seu valor trocado, significa que a atualização realizada no peso sináptico foi muito grande, logo o valor de Δ_{ji} deve ser decrementado pela constante η^-. Por outro lado, se a derivada parcial mantém o seu sinal significa o valor Δ_{ji} deve ser incrementado suavemente para garantir uma convergência mais rápida.

Uma vez encontrado o valor do fator de atualização Δ_{ji}, a atualização dos pesos sinápticos ocorre de acordo com as equações 4.16 e 4.17, se a derivada parcial mantém o seu sinal. Por outro lado, se a derivada parcial muda de sinal, a equação 4.16 é substituída pela equação 4.18

$$\Delta w_{ji}^k = \begin{cases} -\Delta_{ji}(k) & , \quad se \ \frac{\partial \xi(k)}{\partial w_{ji}} > 0 \\ +\Delta_{ji}(k) & , \quad se \ \frac{\partial \xi(k)}{\partial w_{ji}} < 0 \\ 0 & , \quad se \ \frac{\partial \xi(k)}{\partial w_{ji}} = 0 \end{cases} \tag{4.16}$$

$$w_{ji}(k+1) = w_{ji}(k) + \Delta w_{ji}(k) \tag{4.17}$$

$$\Delta w_{ji}(k) = -\Delta w_{ji}(k-1) \quad , \quad se \ \frac{\partial \xi(k-1)}{\partial w_{ji}} * \frac{\partial \xi(k)}{\partial w_{ij}} < 0 \tag{4.18}$$

O funcionamento básico da regra *RPROP* pode ser sumarizado através do algoritmo 4.1(RIEDMILLER; BRAUN, 1993). Nesse algoritmo, os operadores $max(.,.)$ e $min(.,.)$

retornam, respectivamente, o valor máximo e o valor mínimo entre dois números. O operador $sinal(.)$, por outro lado, retorna 1, se o número é positivo, -1 se o número é negativo, ou 0 caso contrário.

Algoritmo 4.1: Propagação elástica

Entrada:
Δ_{max}: o maior valor permitido para Δ_{ji}
Δ_{min}: o menor valor permitido para Δ_{ji}
η^+, η^-: fatores de atualização de Δ_{ji}
Dados:
W: conjunto de todos os pesos sinápticos.
B: conjunto de todos os limiares.

Escolha um valor inicial pequeno para $\Delta_{ji}(0)$
para todo $w_{ji} \in W$ *e* $b \in B$ **faça**
  se $\frac{\partial \xi}{\partial w_{ji}}(k-1) * \frac{\partial \xi}{\partial w_{ji}}(k) > 0$ **então**
    $\Delta_{ji}(k) \longleftarrow \mathbf{min}(\Delta_{ji}(k-1) * \eta^+, \Delta_{max})$
    $\Delta w_{ji}(k) \longleftarrow -\mathbf{sinal}(\frac{\partial \xi}{\partial w_{ji}}(k)) * \Delta_{ji}(k)$
    $\Delta_{ji}(k+1) \longleftarrow w_{ji}(k) + \Delta w_{ji}(k)$
  senão se $\frac{\partial \xi}{\partial w_{ji}}(k-1) * \frac{\partial \xi}{\partial w_{ji}}(k) < 0$ **então**
    $\Delta_{ji}(k) \longleftarrow \mathbf{max}(\Delta_{ji}(k-1) * \eta^-, \Delta_{min})$
    $w_{ji}(k+1) \longleftarrow w_{ji}(k) - \Delta w_{ji}(k-1)$
    $\frac{\partial \xi}{\partial w_{ji}}(k) \longleftarrow 0$
  senão
    $\Delta_{ji}(k) \longleftarrow \mathbf{sinal}(\frac{\partial \xi}{\partial w_{ji}}(k)) * \Delta_{ji}(k)$
    $w_{ji}(k+1) \longleftarrow w_{ji}(k) + \Delta w_{ji}(k)$
  fim
fim

Assim como na Retropropagação, a derivada do erro pode ser encontrada usando-se a regra da cadeia. Os resultados obtidos com o algoritmo independem do valor inicial $\Delta_{ji}(0)$(SCHIFFMANN; JOOST; WERNER, 1994).

4.3 CONSIDERAÇÕES FINAIS

As redes neurais artificiais têm sido uma estratégia bastante utilizada nos casos em que se tem pouco conhecimento a respeito do problema a ser solucionado. Em geral, tais problemas estão relacionados à determinação de um modelo matemático (aproximação de funções), à descoberta de padrões e ao agrupamento de elementos com características

similares. Apesar da implementação de um modelo de uma rede neural ser bastante simples, determinar tal modelo é uma tarefa bastante complicada, uma vez que as estratégias encontradas na literatura, apesar de compravarem a aplicabilidade da rede a um determinado domínio de problema, não apresentam considerações objetivas de como tais redes podem ser obtidas.

CAPÍTULO 5

PROPOSTA DE DETECÇÃO ADAPTATIVA BASEADA EM REDES NEURAIS ARTIFICIAIS

O presente capítulo discute os aspectos referentes à implementação da abordagem de detecção baseada em redes neurais artificiais.

5.1 ABORDAGEM DE DETECÇÃO BASEADA EM REDES NEURAIS

Assim como a abordagem de Macêdo e Lima (2004), a proposta de detecção aqui apresentada utiliza uma *RNA Feedforward multicamada*. Todavia, a abordagem proposta neste trabalho não necessita do protocolo *SNMP*, realizando a adaptação baseada apenas nos instantes de chegada das mensagens de *heartbeat*.

Adotou-se a *RNA Feedforward multicamada*, por conta da facilidade de sua implementação e sua vasta utilização na literatura como um mecanismo para aproximação de funções (HAYKIN, 1994; JAIN; MAO; MOHIUDDIN, 1996; JANG; SUN; MIZUTANI, 1997).

Na subseção 5.1.1 serão apresentados os detalhes da implementação da RNA e na subseção 5.1.2 será abordado o detector adaptativo baseado em RNA.

5.1.1 Implementação da RNA

Uma das dificuldades na implementação da *RNA* é determinar o número de camadas e o número de neurônios em cada camada. Uma vez que a *RNA* deve fornecer a estimativa para o atraso entre chegadas de *heartbeats*, é necessário apenas um neurônio na camada

de saída. Para que se possa determinar o número de neurônios na camada de entrada é necessário conhecer que variáveis serão utilizadas pela Rede Neural.

Na construção do detector, utilizou-se o modelo de monitoramento *push*, por conta do menor número de mensagens de detecção usadas nesse modelo. Todavia, nesse modelo, é difícil precisar o atraso e a variação do atraso no canal de comunicação. As variáveis ΔA_k^{delay} e ΔA_k^{jitter} são influenciadas diretamente pelo atraso e pela variação do atraso de comunicação. Portanto, pressupõe-se que a utilização dessas variáveis deve refletir indiretamente o comportamento do tráfego no canal de comunicação ou ainda possíveis atrasos de computação nos dispositivos.

Para o cálculo de ΔA_k^{delay} e ΔA_k^{jitter}, é necessário que se utilize os três últimos instantes de chegada: A_k, A_{k-1}, A_{k-2}, ou seja:

$$\Delta A_k^{delay} = A_k - A_{k-1} \tag{5.1}$$

e,

$$\Delta A_{k-1}^{delay} = A_{k-1} - A_{k-2} \tag{5.2}$$

mas,

$$\Delta A_k^{jitter} = \Delta A_k^{delay} - \Delta A_{k-1}^{delay} \tag{5.3}$$

com isso,

$$\Delta A_k^{jitter} = A_k - 2.A_{k-1} + A_{k-2} \tag{5.4}$$

Em um ambiente perfeitamente síncrono, as mensagens de *heartbeats* chegam espaçadas de Δ^i unidades de tempo. Entretanto, quando variações no atraso de comunicação ou de computação são consideradas, esse espaçamento pode sofrer alterações. Todavia, Δ^i

pode ainda servir como base para o espaçamento entre as chegadas dos *heartbeats*. Além disso, o conhecimento do período de emissão de *heartbeats* pode auxiliar a *RNA* em encontrar a sobrecarga de comunicação e computação causadas pelo próprio detector de defeitos: quanto menor Δ^i, mais mensagens de *heartbeats* serão geradas pelo mecanismo de detecção.

Dessa forma, pode-se chegar a uma camada de entrada com três neurônios, para que se possa capturar os valores de ΔA^{delay}, ΔA^{jitter} e Δ^i.

O número de camadas ocultas e o número de neurônios em cada uma dessas camadas são difíceis de serem encontrados. Tais números foram encontrados de forma empírica e o procedimento usado é descrito a seguir.

Considera-se um conjunto $A = \{A_1, A_2, ..., A_n\}$ dos instantes de chegada das mensagens de *heartbeats* coletados durante a comunicação entre o módulo monitor e monitorado do detector. Nesse conjunto, A_i representa o instante de chegada da i-éssima mensagem de *heartbeat*(i é um número de seqüência associada a mensagem). Seleciona-se um subconjunto de A, $A^{'} = \{A_{i+1}, A_{i+2}, ..., A_{i+k}\}$, para executar o treinamento da *RNA*. k é o tamanho do subconjunto $A^{'}$. Como a entrada da *RNA* necessita de pelo menos 3 instantes de chegada consecutivos, k deve ser maior ou igual a 3. Os padrões de entrada são subconjuntos de $A^{'}$ com agrupamentos de 3 em 3 instantes de chegada consecutivos.

De posse de A', executa-se o procedimento de seleção da *RNA*, conforme apresentado no algoritmo 5.1. Nesse procedimento, procura-se obter uma Rede Neural que atenda ao requisito mínimo de desempenho adotado ξ^{min}. O procedimento seleciona uma *RNA* e então executa o processo de treinamento da mesma. O treinamento será finalizado caso o desempenho desejado para Rede Neural tenha sido obtido ou caso o número máximo de épocas($\wp^{max}$) tenha sido alcançado. Uma época de treinamento é finalizada após todo o conjunto de treinamento ter sido apresentado à *RNA* (HAYKIN, 1994).

A *RNA* selecionada deve sugerir valores de ΔA^{delay} com uma margem de segurança para evitar falsas suspeitas. Assim, um parâmetro $\hbar < 1$ é utilizado para criar tal margem

de segurança entre o padrão desejado e a saída da Rede Neural (ver linha 18 do algoritmo 5.1).

Para garantir uma resposta confiável e um treinamento eficiente, é necessário normalizar os valores de entrada e saída da *RNA*, de modo a garantir que esses valores estejam dentro de uma faixa conhecida (DEMUTH; BEALE, 1998). Desta forma, utiliza-se o parâmetro ζ como fator de normalização dos valores de entrada e saída da Rede Neural.

Após a execução do algoritmo 5.1, encontra-se uma *RNA* com o número de camadas ocultas e o número de neurônios em cada camada definidos.

Algoritmo 5.1: Procedimento de Seleção da Rede Neural Artificial

1 *Selecione um desempenho mínimo* ξ^{min} *para RNA*
2 *Selecione um número máximo* $\wp^{max}$ *de épocas de treinamento*
3 *Selecione um valor para* $\hbar$
4 *Selecione um valor para* ζ
5 **repita**
6 *Selecione uma configuração para RNA*
7 *Inicialize aleatoriamente os pesos sinápticos da RNA*
8 $\wp \leftarrow 0$
9 **repita**
10 $\Xi \leftarrow \{\}$
11 **para** $j \leftarrow 4$ **até** k **faça**
12 *Calcule* $A_j^{delay} \leftarrow A_j - A_{j-1}$
13 *Calcule* $A_{j-1}^{delay} \leftarrow A_{j-1} - A_{j-2}$
14 *Calcule* $A_{j-2}^{delay} \leftarrow A_{j-2} - A_{j-3}$
15 *Calcule* $A_{j-1}^{jitter} \leftarrow A_{j-1}^{delay} - A_{j-2}^{delay}$
16 *Obtenha* Δ_{j-1}^{i}
17 *Construa o padrão de entrada* $\pi^E \leftarrow \begin{pmatrix} A_j^{delay} \\ A_j^{jitter} \\ \Delta_j^i \end{pmatrix} * \frac{1}{\zeta}$
18 *Construa o padrão de saída* $\pi^S \leftarrow A_j^{delay} * \frac{(1+\hbar)}{\zeta}$
19 *Verifique o padrão na saída da rede* $\pi^C \leftarrow RNA\left(\pi^E\right)$
20 *Verifique erro* $\xi^{\pi} \leftarrow \frac{1}{2}(\pi^S - \pi^C)^2$
21 *Realize o ajustes dos pesos sinápticos da* RNA *seguindo a regra adotada*
22 $\Xi \leftarrow \Xi \cup \{\xi^{\pi}\}$
23 **fim**
24 *Calcule:* $\xi^{RNA} \leftarrow \frac{1}{k-3}\left[\sum_{\xi \in \Xi} \xi^2\right]$
25 $\wp \leftarrow \wp + 1$
26 **até** $\xi^{RNA} \leq \xi^{min}$ ***ou*** $\wp > \wp^{max}$;
27 **até** $\xi^{RNA} \leq \xi^{min}$;

Conforme sugerido em Haykin (1994), os pesos sinápticos da rede são selecionados aleatoriamente[1] dentro da faixa:

$$\left(-\frac{2.4}{f_j^{in}}, \frac{2.4}{f_j^{in}}\right) \tag{5.5}$$

em que f_j^{in} representa o número total de entradas para o neurônio j. Sendo assim, o peso de cada conexão sináptica é selecionado individualmente.

Como função de ativação, adotou-se em cada neurônio a função tangente hiperbólica:

$$f(\frac{x}{2}) = \frac{1 - e^{-x}}{1 + e^{-x}} \tag{5.6}$$

O procedimento de seleção da configuração da *RNA* foi realizado de forma empírica. Para tanto, foram testadas diversas configurações com uma e duas camadas ocultas.

As configurações com apenas uma camada oculta, como sugerido por alguns critérios na literatura[2], têm uma convergência mais rápida, mas não obtiveram um bom desempenho com base no critério adotado. Tais critérios são extremamente complexos e não apresentam uma proposta clara para a obtenção do número adequado de neurônios na camada oculta.

Configurações com duas camadas ocultas demandaram um maior tempo para a convergência, todavia apresentaram um melhor desempenho em relação ao critério adotado.

Para o ajuste dos pesos sinápticos, utilizou-se o algoritmo de *Propagação Elástica*. Diversos algoritmos foram testados, mas o *RPROP* foi o que trouxe um menor tempo de convergência em todos os casos, quando comparado aos demais.

[1] Modelo de seleção paramétrica do tipo *caminho mais fácil*, ver Cerqueira (2003).
[2] ver Irie e Miyake (1988), Pinkus (1999), Haykin (1994), Tikk, Kóczy e Gedeon (2003).

5.1.2 Detector de Defeitos Adaptativo baseado em Feedforward Multicamada

A *RNA* utilizada possui quatro camadas e realiza suas estimativas observando apenas a taxa de emissão(Δ^i) e o comportamento entre os instantes de chegadas das mensagens de *heartbeats*.

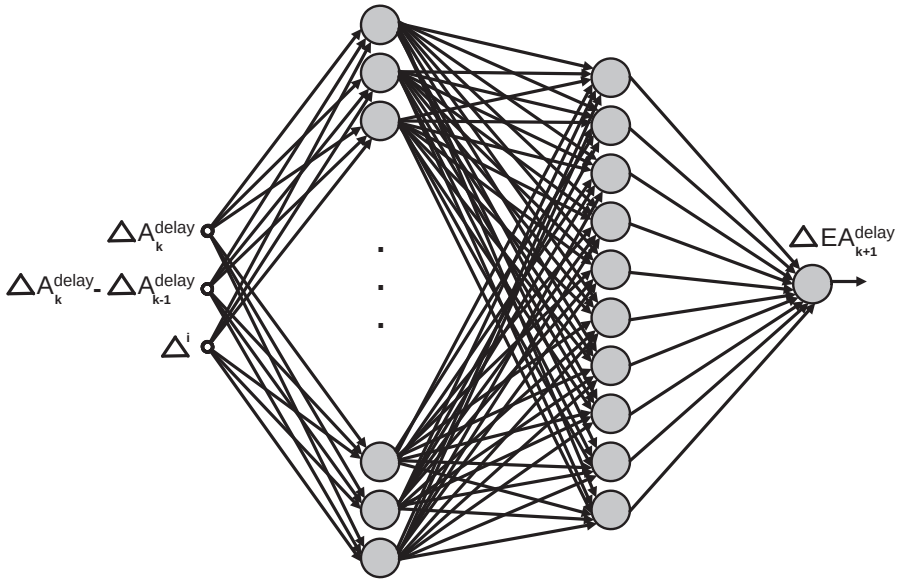


Figura 5.1. Modelo de rede neural MLP 3-30-10-1

Como pode ser visto na figura 5.1, o modelo matemático implementado pela Rede Neural é composto por: uma camada de entrada com três neurônios; uma camada de saída com um neurônio; e duas camadas ocultas com trinta e dez neurônios, respectivamente.

A camada de entrada recebe o atraso (ΔA_k^{delay}) entre os instantes de chegada dos *heartbeats* m_k^{hb} e m_{k-1}^{hb}, a variação desse atraso (ΔA_k^{jitter}) e a taxa de emissão de *heartbeats* (Δ^i). Essas variáveis devem ser processadas pela RNA para que a mesma possa estimar e informar, através do neurônio da camada de saída, o intervalo de tempo necessário para a chegada da próxima mensagem de *heartbeat*. Em média, a RNA necessita de $0.7\mu s$ para realizar o processamento das entradas e sugerir e realizar a estimativa desejada.

O detector adaptativo utiliza a estimativa da *RNA* para realizar a predição dos instantes de chegada das mensagens de *heartbeats*. Tal processo é executado da forma a

seguir. Calcula-se ΔA_k^{delay}, ΔA_k^{jitter} e Δ_k^i de modo a se obter o padrão de entrada para a **RNA**:

$$\pi_k = \begin{bmatrix} A_k^{delay} \\ \Delta A_k^{jitter} \\ \Delta_k^i \end{bmatrix} * \frac{1}{\zeta} \tag{5.7}$$

Para as estimativas, considera-se que, após o treinamento da Rede Neural, esta assimilará a função $\Omega(\pi_k)$ que sugere o próximo intervalo esperado para a chegada da próxima mensagem de *heartbeat* m_{k+1}^{hb}. Assim, o detector de defeitos computará:

$$EA_{k+1} = EA_k + \zeta * \Omega(\pi_k) \tag{5.8}$$

$\Omega(\pi_k)$ representa a função Ω aplicada aos parâmetros de entrada da Rede Neural no instante k.

5.2 SIMULAÇÕES REALIZADAS

Nesta seção, será avaliada a abordagem adaptativa baseada em redes neurais, contrapondo-a a outras abordagens existentes na literatura. Todas as avaliações foram conduzidas em ambientes simulados e os detalhes das simulações, bem como os resultados obtidos, serão apresentados nas subseções a seguir.

5.2.1 Modelo do Sistema

Os sistemas analisados nas simulações são modelados da forma a seguir. Para cada sistema, utiliza-se um sensor ($node^{sns}$), um atuador ($node^{atd}$) e um controlador ($node^{ctrl}$). Cada dispositivo é dotado de um sistema operacional de tempo real multitarefa e são co-

nectados através de um subsistema de comunicação para interagir via troca de mensagens e colaborar para compor um sistema de controle sobre rede(NCS).

A figura 5.2 apresenta o modelo de sistema proposto.

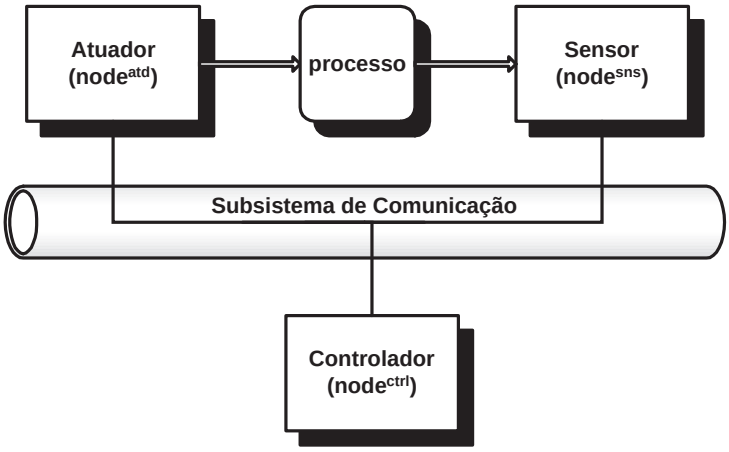


Figura 5.2. Sistema de controle sobre rede

No sensor se executa uma tarefa periódica de aquisição de dados (τ^{sns}). A cada amostra coletada por τ^{sns}, uma mensagem com a amostra é enviada para o elemento controlador. No controlador, a tarefa de controle (τ^{ctrl}) é esporádica e é acionada a cada mensagem recebida do sensor. De posse dos dados adquiridos do objeto controlado, τ^{ctrl} executa um algoritmo de controle e então envia uma mensagem com a informação de controle para o atuador. A tarefa τ^{atd} é ativada no elemento atuador a cada evento de recepção de uma mensagem de controle. Recebida a informação de controle, a tarefa de atuação realiza a efetiva atuação sobre o processo que está sendo controlado.

Assume-se a possibilidade de parada do controlador. Desse modo, o dispositivo é replicado, formando um esquema redundante para tolerar uma única falha por omissão infinita ($crash$[3]). Um dos controladores é dito controlador primário ($node_p^{ctrl}$) e o outro é dito secundário ($node_s^{ctrl}$). O controlador primário tem por responsabilidade receber as mensagens oriundas do sensor; executar um algoritmo que garanta a consistência do estado do mesmo com o estado atual do controlador secundário; e enviar a ação de

[3]Ver Veríssimo e Rodrigues (2000) e Laprie (1985) para maiores detalhes.

controle ao atuador. O controlador secundário possui um detector de defeitos embarcado (FD^{emb}) para verificar falhas no controlador primário. O algoritmo de detecção em FD^{emb} é modelado conforme indicado na seção 3.2.2.

A figura 5.3 apresenta o novo modelo de *NCS* com a adição dos mecanismos de tolerância a falhas.

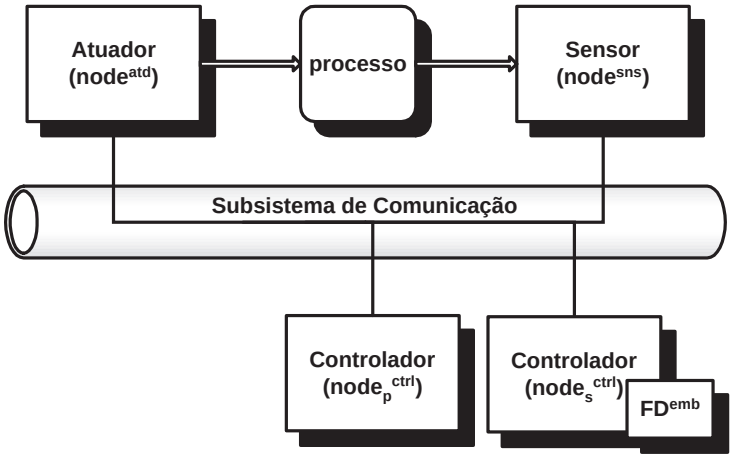


Figura 5.3. Sistema de controle sobre rede com mecanismos de tolerância a falhas

O subsistema de comunicação garante que quando um processo p_s envia mensagens para um processo p_r, se p_s envia m_k antes de m_{k+1} e ambas as mensagens m_k e m_{k+1} são entregues a p_r, o recebimento de m_k precede o recebimento m_{k+1}. Não é possível a existência de partições de rede, mas altas condições de tráfego podem implicar em perda de mensagens.

5.2.2 A Ferramenta de Simulação Simulink/TrueTime.

O modelo apresentado foi implementado utilizando o *ToolBox TrueTime* versão *1.3* de Henriksson e Cervin (2003). Esse *Toolbox* foi desenvolvido para o pacote de software *Simulink*(The Mathworks, 2004) do *Matlab* (The Mathworks, 2002)[4].

[4]Mais detalhes sobre o ambiente *Matlab*, o pacote *Simulink* e o *Toolbox TrueTime* podem ser encontrados no apêndice A.

O *TrueTime* permite que estações com sistemas operacionais multitarefas e de tempo real sejam simuladas. Além disso, é possível interligar tais estações através de redes de comunicação, para as quais se pode, não apenas selecionar o protocolo a ser utilizado, mas também uma série de outros parâmetros que ditam as características da rede (como tamanho de mensagem, atraso de processamento, probabilidade de perdas, entre outros).

Para disponibilizar tais facilidades, o *TrueTime* dispõe de dois blocos principais: o bloco para Estação com *Kernel* multitarefa e de tempo real e um bloco Rede para comunicação em tempo real. Usando o conjunto de blocos especiais disponíveis no *Simulink*, instâncias desses blocos podem ser interligadas para formar redes de controle de tempo real.

A figura 5.4 apresenta o sistema de controle sobre rede usado nas simulações realizadas. Nesse sistema estão os dispositivos sensor, controlador primário, controlador secundário, atuador e planta, conforme apresentado na seção 5.2.1

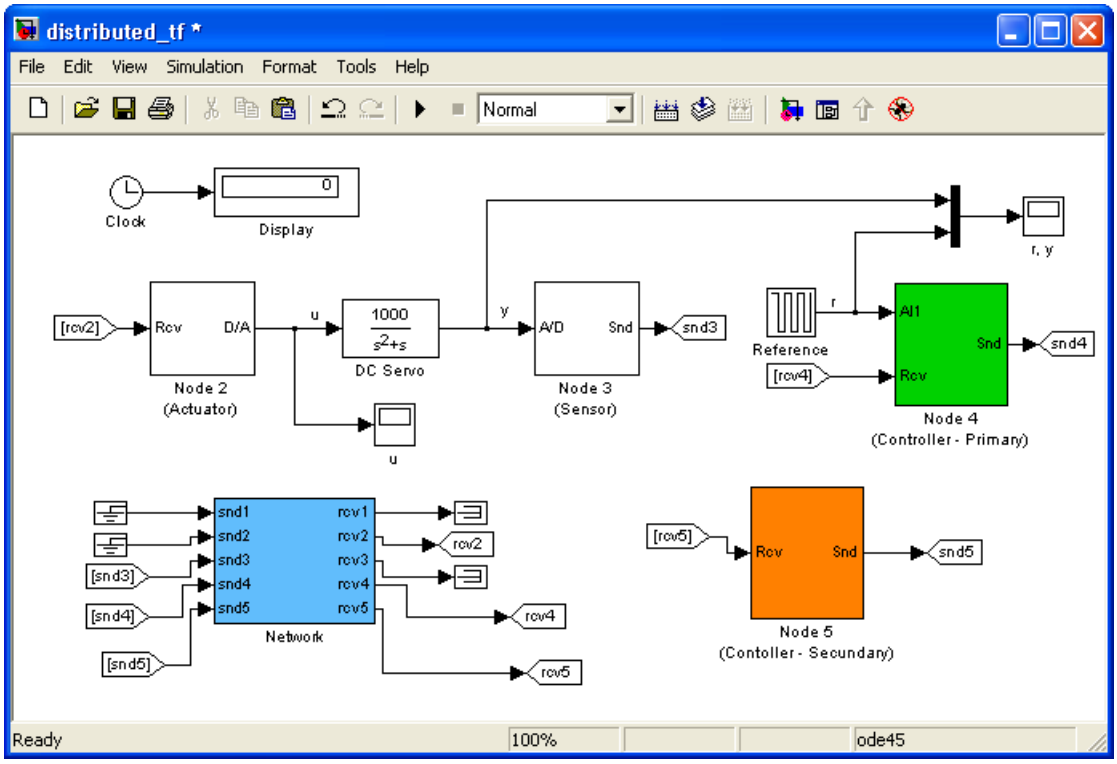


Figura 5.4. Modelagem de um dos sistemas simulados usando o *Toolbox TrueTime/Simulink*

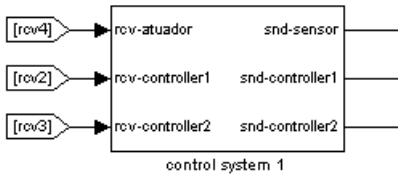
Para cada bloco de dispositivo, se tem um algoritmo de inicialização associado. Através desse algoritmo, pode-se decidir a política de escalonamento a ser utilizada, especificar o número de portas de entrada e saída, e definir as características das tarefas a serem executadas na estação. Por exemplo, o nó sensor possui uma porta de entrada através da qual o mesmo adquire as amostras do processo controlado. O atuador, por outro lado, possui uma porta de saída para enviar a ação de controle ao processo. Uma descrição mais detalhada do processo de modelagem, configuração e simulação de redes de controle de tempo real usando o *TrueTime* pode ser encontrada no apêndice A ou em Henriksson e Cervin (2003).

5.2.3 Configurando o Ambiente de Simulação.

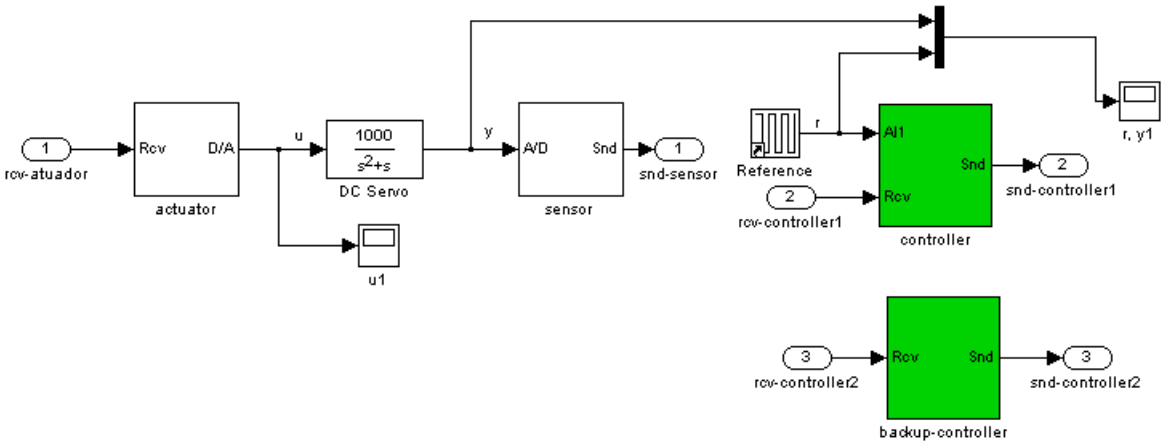
Para a realização das simulações, algumas configurações precisam ser realizadas. Primeiramente precisa-se definir o número de subsistemas a serem utilizados. Para tanto, utilizou-se uma facilidade do *Simulink*, através da qual um subsistema inteiro pode ser mapeado como um bloco. A figura 5.5 mostra a representação de um subsistema como bloco do *Simulink* (figura 5.5(a)) e sua representação interna (figura 5.5(b)). Cada subsistema mapeado contém dois dispositivos controladores (um primário e outro secundário), um dispositivo sensor, um dispositivo atuador e um *DC-Servo*(conforme descrito em 5.2.1). O *DC-Servo* representa a planta e é mapeado por um sistema linear invariante no tempo de segunda ordem, baseado em um dos exemplos apresentados em (HENRIKSSON; CERVIN, 2003).

Os dispositivos representados pelo bloco de subsistema de controle são conectados ao bloco de rede, como pode ser visto na figura 5.6(a). Nessa figura, é apresentado um modelo no qual quatro sistemas de controle de tempo real compartilham um mesmo sistema de comunicação.

Como cada sistema de controle é composto por 4 dispositivos, um sensor, um atu-



(a) Bloco subsistema



(b) dispositivos internos do subsistema

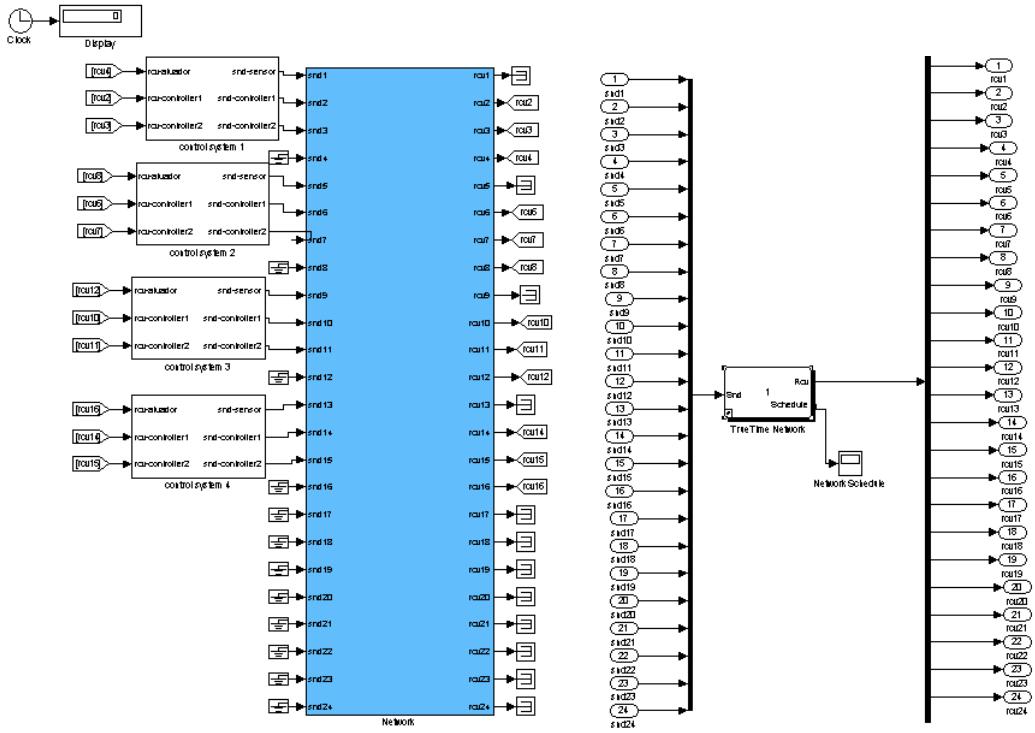
Figura 5.5. Subsistema de controle mapeado no Simulink. Como um macro bloco na figura 5.5(a) e seu conjunto de dispositivos na figura 5.5(b).

ador e dois controladores ligados através da rede, o exemplo da figura 5.6(a) possui 16 dispositivos. As simulações realizadas consideram cenários com quatro, oito e vinte dispositivos, ou seja, um, dois, e cinco sistemas de controle, respectivamente, compartilhando um mesmo canal de comunicação. A figura 5.6(b) mostra o bloco de rede do *TrueTime* configurado com vinte e quatro portas para atender às simulações realizadas.

Uma vez que a quantidade de dispositivos foi definida, é necessário realizar a configuração do bloco de rede com os parâmetros desejados. Dentre os tipos de rede utilizados nas simulações, selecionou-se as redes: *Shared-Bus Ethernet*(*CSMA-CD*), CAN(CSMA/AMP), e *Switched-Ethernet*.

Os parâmetros de rede apresentados na tabela 5.1 foram fixados durante as simulações.

O ambiente de simulação foi configurado seguindo o exemplo da figura 5.7. Maiores detalhes sobre esses parâmetros podem ser encontrados no apêndice A.



(a) Modelo com 4 sistemas compartilhando uma rede de comunicação

(b) Visão interna do bloco de rede com 24 portas

Figura 5.6. Modelo com 4 sistemas compartilhando uma rede de comunicação e representação interna da rede usada

Tabela 5.1. Parâmetros fixados na configuração do bloco de rede

Parâmetro	**Valor**
Identificador de rede (*network number*)	1
Número de nós	24
Atraso de entrada (*preprocessing delay*)	0
Atraso de saída (*posprocessing delay*)	0
Probabilidade de Perda (*Loss probability*)	0

Além da configuração da rede, para que a simulação seja realizada, é necessário que dispositivos sejam configurados. Os algoritmos utilizados na inicialização de cada um dos sistemas operacionais de tempo real podem ser encontrados na seção A.4.2.1.

O modelo de tarefas do sistema é especificado conforme indicado na tabela 5.2.

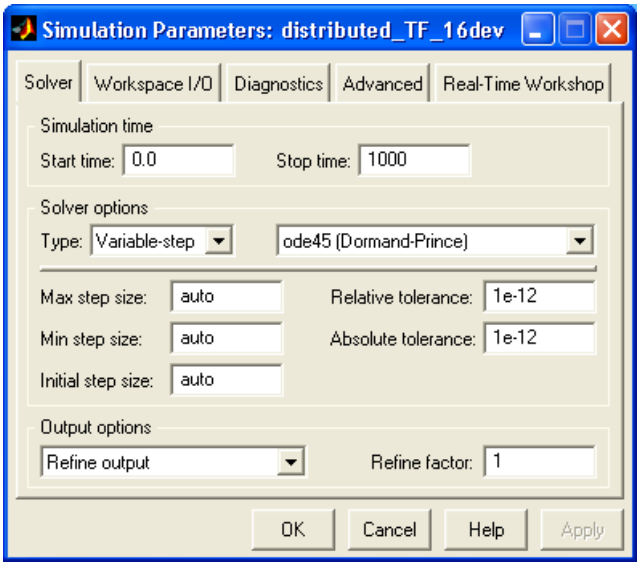


Figura 5.7. Configuração do ambiente simulink

Tabela 5.2. Especificação das tarefas do sistema

Tarefa	**Dispositivo**	**Tipo de Ativação**	**Evento**	**Deadline** (D)	**Período** (T)	**WCET**
Controle (τ_{ctr})	Controlador	Esporádica	Recepção da mensagem enviada por τ_{sns}	$6ms$	-	$0.5ms$
Aquisição (τ_{snr})	Sensor	Periódica	-	$10ms$	$10ms$	$0.4ms$
Atuação (τ_{atd})	Atuador	Esporádica	Recepção da mensagem enviada por τ_{ctrl}	$4ms$	-	$0.5ms$
Emissor de *heartbearts*	Controlador Primário	Periódica	-	Δ^i	Δ^i	$0.005ms$
Monitor de *heartbeats*	Controlador Secundário	Esporádica	Recepção de um *heartbeat*	Δ^i	-	$0.005ms$

5.2.4 Métricas de Detecção Observadas e Configuração dos Detectores nas Simulações.

Antes de iniciar a discussão dos resultados, é necessário tecer mais alguns comentários sobre as variáveis observadas nas simulações. Cada simulação é realizada durante a troca de 440 mensagens entre os módulos do detector de defeitos instalados nos controladores primários e secundários dos sistemas de controle (10% dessas mensagens são usadas para

inicializações dos algoritmos de detecção). Avalia-se o desempenho do detector, para diferentes taxas de emissão de *heartbeats* $\Delta^i = 0.1ms$, $0.5ms$, $0.9ms$, $1ms$, $2ms$, *e* $5ms$. Por conta da velocidade das redes de comunicação de *heartbeats*, períodos de emissão inferiores a $0.1ms$ não são tolerados, enquanto períodos de emissão superiores a $5ms$, como será discutido nas subseções seguintes, não garantem os requisitos estabelecidos pelo sistema de controle.

Nas simulações realizadas, comparou-se o desempenho da abordagem baseada em **RNA** com os algoritmos de Jacobson (1988) e de Bertier, Marin e Sens (2002)[5]. Para tais algoritmos, configurou-se $\beta = 1$, $\phi = 2$, $\mu = 0.1$, como originalmente proposto pelos autores.

O treinamento e execução da *RNA* foram realizados utilizando os seguintes parâmetros:

- $\eta^+ = 1.2$, $\eta^- = 0.2$, $\Delta_{max} = 50$ e $\Delta_{min} = 0.08$, conforme recomendações de Riedmiller e Braun (1993);
- $\Delta_{ji}(0) = 0.08$, $\wp^{max} = 5 * 10^5$ e $\xi^{min} = 10^{-13}$;
- $\hbar = 0.025$ e $\zeta = 10^4$;

O desempenho dos algoritmos de detecção foi observado em termos dos valores máximos, médios e desvios padrões das métricas primárias de detecção de defeitos (T_D, T_M, T_{MR}). Além disso, observou-se o número de falsas suspeitas cometidas (N^{fs}) por cada detector durante as simulações.

Define-se o valor máximo (max) de um conjunto de amostras $X = \{x_1, x_2, x_3, ..., x_N\}$ por:

$$max(X) = \left\{ x_i | x_i \geq x_j, \forall (x_i, x_j) \in X \right\} \tag{5.9}$$

Da mesma forma, pode-se definir o valor médio ($mean$) e o desvio padrão (std) por:

[5] vide seção 3.2.2

$$mean(X) = \frac{1}{N}\left(\sum_{k=1}^{N} x_k\right) \tag{5.10}$$

e,

$$std(X) = \sqrt{\frac{1}{N}\sum_{p=1}^{N}\left[x_p - mean(X)\right]^2} \tag{5.11}$$

5.2.5 Modelo para Avaliação do Impacto da Detecção no Desempenho do Sistema de Controle

As simulações realizadas assumem um motor $Servo - CC$ representado pela função de transferência abaixo:

$$G(s) = \frac{1000}{s(s+1)}$$

Além disso, considera-se um controlador PID(Proporcional-Intregal-Derivativo[6]), implementado conforme abaixo:

$$e(k) = r(k) - y(k)$$

$$P(k) = K.e(k)$$

$$I(k+1) = I(k) + \frac{Kh}{T_i}e(k)$$

$$D(k) = a_d D(k-1) + b_d(y(k-1) - y(k))$$

$$u(k) = P(k) + I(k) + D(k)$$

$P(k)$, $I(k)$ e $D(k)$ representam os ganhos proporcional, integral e derivativo, em que $a_d = \frac{T_d}{Nh+T_d}$ e $b_d = \frac{NKT_d}{Nh+T_d}$. O modelo da planta e do controlador PID são os mesmos

[6]Ver Ogata (1990) para maiores detalhes.

usados em Henriksson e Cervin (2003). Assim, os parâmetros usados na sintonia do controlador PID foram: $N = 10$, $T_d = 0.035$, $T_i = 1$, $h = 10ms$ e $K = 1.5$.

Diferente da metodologia inicialmente utilizada em Sá e Macêdo (2005), o impacto na qualidade do controle foi observado em termos de índices clássicos de desempenho IAE e ISE[7]. Além disso, utilizou-se também o conceito de margem de *jitter*, proposto por Cervin et al. (2003), como uma medida adicional para verificar o impacto do detector no desempenho do sistema de controle.

A avaliação usando o conceito de margem de *jitter* é realizada através dos seguintes passos:

i) Projeta-se uma malha de controle fechada composta por um controlador discreto $C(z)$, com período de amostragem h, e planta discretizada $P(z)$. Uma malha $C(z)P(z)$ deve ser estável.

ii) Calcular P_{alias}, realizando uma pequena adaptação do sugerido por (LINCOLN, 2002):

   a. Assuma um ganho máximo K_{max} e uma freqüência máxima w_{max}.

   b. Considerando a faixa de freqüência $w \in W = [-w_{max}, w_{max}]$, obtenha

$$P_{alias}(w) = \sqrt{\sum_{k=-K_{max}}^{K_{max}} \left| P\left(i(w + 2\pi k)\frac{1}{T_s} \right) \right|^2}$$

iii) Calcular os autovalores $B(w)$ de

$$\frac{-e^{jwT_s} + 1}{e^{jwT_s}}$$

iv) Calcular, para a malha de controle, o ganho máximo G_{max}

[7] Ver seção 2.2.2

$$G_{max} = min(P_{alias}. * B)$$

em que a operação $P_{alias}. * B$ é o produto elemento a elemento dos vetores P_{alias} e B . $min(.)$ representa o menor valor de um vetor.

v) Calcular g

$$g = \frac{G_{max}^2 - \lfloor G_{max} \rfloor^2}{1 + 2 * \lfloor G_{max} \rfloor}$$

vi) Obter $N = g + \lfloor G_{max} \rfloor$

vii) Calcular a *margem de jitter*

$$J_m = N * T_s$$

5.2.6 Avaliando o Desempenho e o Impacto dos Detectores em NCS sobre CAN

Nesta subseção, avalia-se o desempenho dos detectores de defeitos e o impacto de sua implementação na qualidade de NCS com dispositivos interconectados através de uma rede que implementa o protocolo *CAN, Control Area Network* (MACKAY, 2004). Esse protocolo disciplina o acesso ao meio físico de comunicação através de uma política de prioridades. As prioridades são atribuídas de acordo com o identificador do dispositivo na rede; quanto maior o identificador menor, a prioridade.

As simulações usando as redes *CAN* consideraram cenários nos quais essa rede é compartilhada por um, dois e cinco sistemas de controle. Cada sistema de controle foi configurado conforme descrito nas subseções anteriores. Em cenários com múltiplos sistemas de controle, os sistemas com menor identificador possuem os dispositivos com maior prioridade. Desse modo, os sistemas com os maiores identificadores (e dispositivos com as menores prioridades) serão penalizados durante a execução das simulações.

Os sistemas de controle são configurados de forma que o dispositivo sensor tenha maior

prioridade que o controlador primário, e que esse, por sua vez, possua uma prioridade maior que o controlador secundário. E, por fim, o controlador secundário possui maior prioridade que o dispositivo atuador.

As avaliações consideram uma rede *CAN* com taxa de transferência de $500Kbps$, taxa padrão usada em redes industriais *DeviceNet*(MACKAY, 2004). As mensagens enviadas através da rede possuem um tamanho máximo de 2 *bytes*. Assim, cada quadro transmitido terá cerca de 10 *bytes*, considerando 47 *bits* de controle e nenhum *stuff bit*(TINDELL; HANSSMON; WELLINGS, 1994).

Por fim, na implementação dos detectores de defeitos foram usados períodos de emissão de *heartbeats* $\Delta^i = 0.1$, 0.5, 0.9, 1.0, 2.0, e 5.0, todos medidos em milissegundos. Os gráficos e tabelas, apresentados para as avaliações com a rede CAN nas subseções a seguir, são observados a partir do pior caso, ou seja, através do sistema com os dispositivos com as menores prioridades.

5.2.6.1 Avaliando o Desempenho dos Algoritmos de Adaptação Usados nos Detectores de Defeitos.

Considerando quadros com 10 *bytes* e uma taxa de transferência de *500Kbps*, pode-se verificar que cada mensagem será transmitida em aproximadamente $0.16ms$. Portanto, períodos de emissão de *heartbeats* inferiores a $0.16\mu s$[8] fazem com que o sistema de controle 1 ocupe toda a banda passante disponível. Assim, em um cenário com vários sistemas de controle, nenhum dos demais sistemas terá dispositivos acessando o meio de comunicação. Isso justifica o fato das tabelas que demonstram o desempenho dos detectores não possuírem informações para os sistemas 2, 3, 4 e 5, nas simulações em que são utilizados períodos de emissão de *heartbeats* $\Delta^i = 0.1ms$. A mesma análise justifica a ausência de dados para os sistemas 4 e 5 quando se utiliza um período de emissão de *heartbeats* $\Delta^i = 0.5ms$.

[8]Na realidade, considerando a inserção de *stuff bits*, o tempo de transferência de um quadro será um pouco superior a $0.16\mu s$.

Por possuir dispositivos com as maiores prioridades, o sistema 1 não tem suas transmissões penalizadas por conta das disputas pelo uso do meio de transmissão. Conforme apresentado em subseções anteriores, o controlador primário possui duas tarefas periódicas: a tarefa de controle e a tarefa de emissão de *heartbeats*. A tarefa de controle possui maior prioridade que a tarefa de emissão de *heartbeats*. Por conta disso, alguns *heartbeats* serão enviados com atraso quando essas duas tarefas disputarem pelo uso do processador do dispositivo de controle.

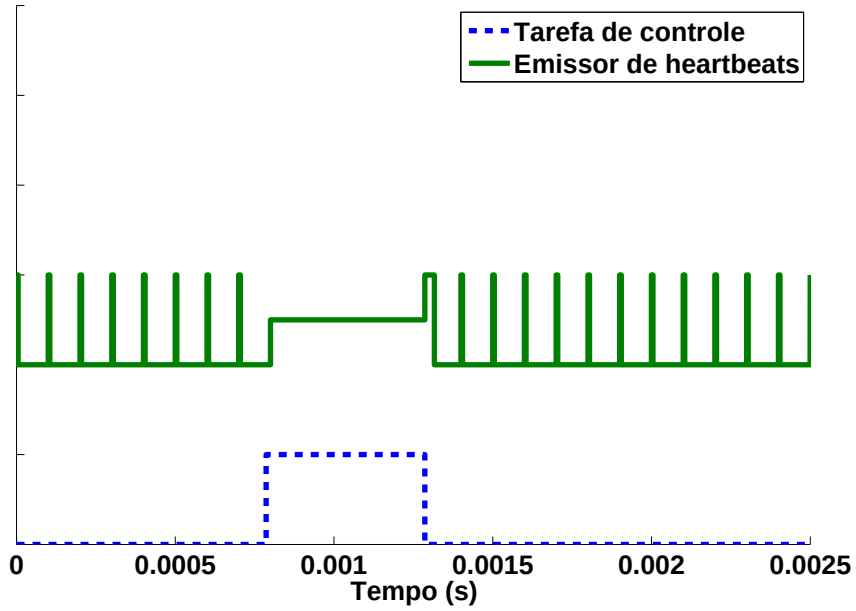
Assim, uma vez que a tarefa de controle será ativada com períodos de 10 a $10.16ms$ aproximadamente, pode-se, desse modo, afirmar que ocorrerá uma disputa a cada:

$$T_{disputa} = \left\lfloor \frac{10.16}{\Delta^i} \right\rfloor \tag{5.12}$$

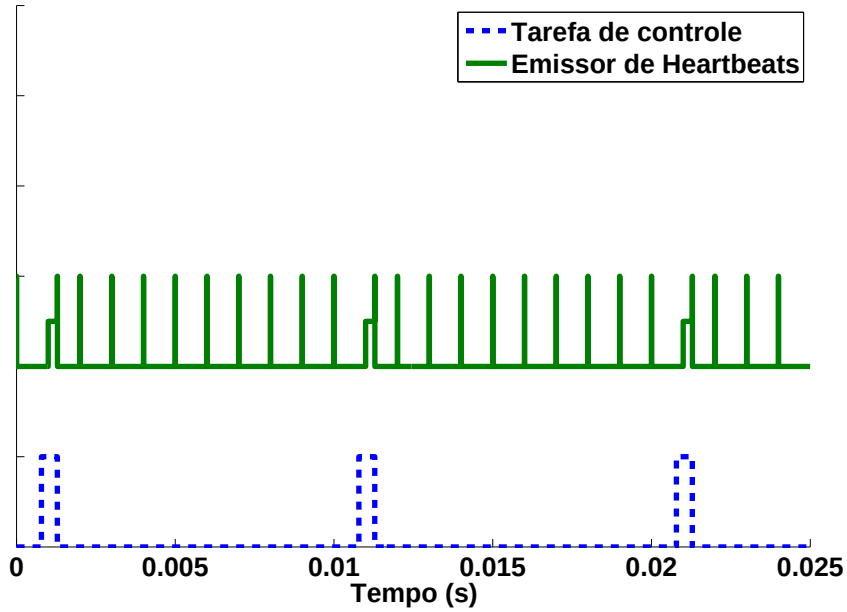
$T_{disputa}$ representa os períodos de disputa para uso do processador no dispositivo de controle. Esse período é válido para uma *NCS* com redes *CAN* e para as demais redes a serem analisadas nas próximas seções.

As figuras 5.8(a) e 5.8(b) apresentam gráficos que exemplificam o escalonamento das tarefas no controlador primário para períodos de emissão de *heartbeats* de $0.1ms$ e $1ms$, respectivamente. As curvas em verde(contínuas) representam a tarefa de emissão de *heartbeats*, enquanto as curvas em azul(tracejadas) representam a tarefa de controle. São utilizadas diferentes escalas de tempo para as duas figuras, dessa forma se pode visualizar com maior facilidade a influência das disputas no controlador. Nos gráficos, quando a curva assume valor baixo, significa que a tarefa não precisa ser escalonada no momento; uma curva com o valor alto significa que a tarefa está executando; e um valor intermediário significa que a tarefa está aguardando para fazer uso do processador.

Como pode ser visto na figura 5.8(a), para um período de emissão de *heartbeats* $\Delta^i = 0.1ms$ acontecerá uma disputa a cada $10ms$ aproximadamente, e a cada disputa mais de 5 *heartbeats* serão atrasados, ou perdidos. Por outro lado, para uma taxa de



(a)



(b)

Figura 5.8. Escalonamento das tarefas de controle e de emissão de *heartbeats*: 5.8(a)$\Delta^i = 0.1ms$, 5.8(b)$\Delta^i = 1ms$

emissão de *heartbeats* $\Delta^i = 1ms$, figura 5.8(b), apenas 1 *heartbeat* será atrasado. Nesse caso, o período $\Delta^i = 1ms$ é suficiente para acomodar o tempo de execução da tarefa de controle em caso de disputa.

Essa análise demonstra o efeito do atraso de computação na emissão de *heartbeats*. Assim sendo, mesmo em cenários nos quais não existem atrasos provocados por acesso ao meio de comunicação, é necessário que o algoritmo de adaptação usado pelo detector de defeitos contorne os atrasos de computação de modo a evitar falsas suspeitas.

À medida que o período de emissão de *heartbeats* aumenta, o número de disputas pela unidade de processamento do dispositivo controlador diminui (*discretamente*) e o número de *heartbeats* afetados também é decrementado. Para períodos de emissão de *heartbeats* menores que 0.9, as simulações realizadas mostram que esse efeito ainda significativo. Assim, o efeito do atraso de computação se torna ainda é significativo. Maior é, portanto, a probabilidade do detector de defeitos cometer uma falsa suspeita. Por outro lado, quando o período de emissão de *heartbeats* aumenta ($\Delta^i > 0.9$), a relação entre o mesmo e o tempo de execução da tarefa de controle se torna maior, menos *heartbeats* serão atrasados e, conseqüentemente, menor será a influência do atraso de computação.

O número de *heartbeats* atrasados por disputas pelo uso do processador pode ser obtido por:

$$N_{hb}^{delay} = \left\lceil \frac{T_{espera}}{\Delta^i} \right\rceil \tag{5.13}$$

em que N_{hb}^{delay} é o número aproximado de *heartbeats* atrasados e T_{espera} representa o somatório dos tempos de computação das tarefas de maior prioridade que a tarefa de emissão de *heartbeats* que participam da disputa. Esse tempo pode ser obtido através de uma análise do escalonamento das tarefas. No caso especial da simulação, T_{espera} equivale ao tempo de execução da tarefa de controle.

Com o aumento do período de emissão de *heartbeats*, mais dispositivos têm a opor-

tunidade de iniciar uma transmissão. As disputas pelo acesso ao meio ainda continuarão sendo vencidas pelos dispositivos com as maiores prioridades. Mas, por outro lado, se um dispositivo de menor prioridade conseguir ter acesso ao meio de comunicação, dispositivos com as maiores prioridades terão que aguardar o término de tal transmissão antes de tomarem posse do canal de comunicação. Sendo assim, quanto maior for o período de emissão de *heartbeats*, maior será a possibilidade dos dispositivos com as menores prioridades terem acesso ao meio e maior será a probabilidade de um dispositivo ter sua transmissão atrasada por conta de uma outra transmissão que esteja em andamento.

Assim, o efeito do atraso de comunicação sobre os dispositivos com as maiores prioridades se torna mais evidente, uma vez que a probabilidade de se ter alguns *heartbeats* atrasados por conta de uma transmissão em andamento é maior. Entretanto, quando o período de emissão de *heartbeats* é incrementado, $\Delta^i > 2ms$, o efeito do atraso de comunicação provocado pela emissão de *heartbeats* se torna muito pequeno e, assim, o atraso de comunicação é influenciado pura e simplesmente pelo tráfego gerado pelas transmissões das aplicações de controle.

A figura 5.9 apresenta o escalonamento das mensagens em uma rede *CAN* compartilhada por dois sistemas de controle e com detectores de defeitos configurados para usar um período de emissão de heartbeats de $1ms$. Nesse gráfico, quando a curva assume valor baixo, significa que o dispositivo não possui mensagens a serem transmitidas; uma curva com o valor alto significa que o dispositivo está realizando uma transmissão; e um valor intermediário significa que o dispositivo está aguardando a liberação do meio de comunicação para transmitir suas mensagens.

Como no exemplo da figura 5.9, os sistemas de controle possuem um mesmo período de amostragem $h = 10ms$, a cada $10ms$ os sensores (sensor 1, curva de cor azul – composta por traços e pontos–, e sensor 2, curva de cor violeta – curva pontilhada) disputam o acesso ao meio de comunicação (intervalos entre 0 e $1ms$ e entre 10 e $11ms$). Uma vez que o sensor do sistema 1 possui maior prioridade, esse assume o canal enviando a

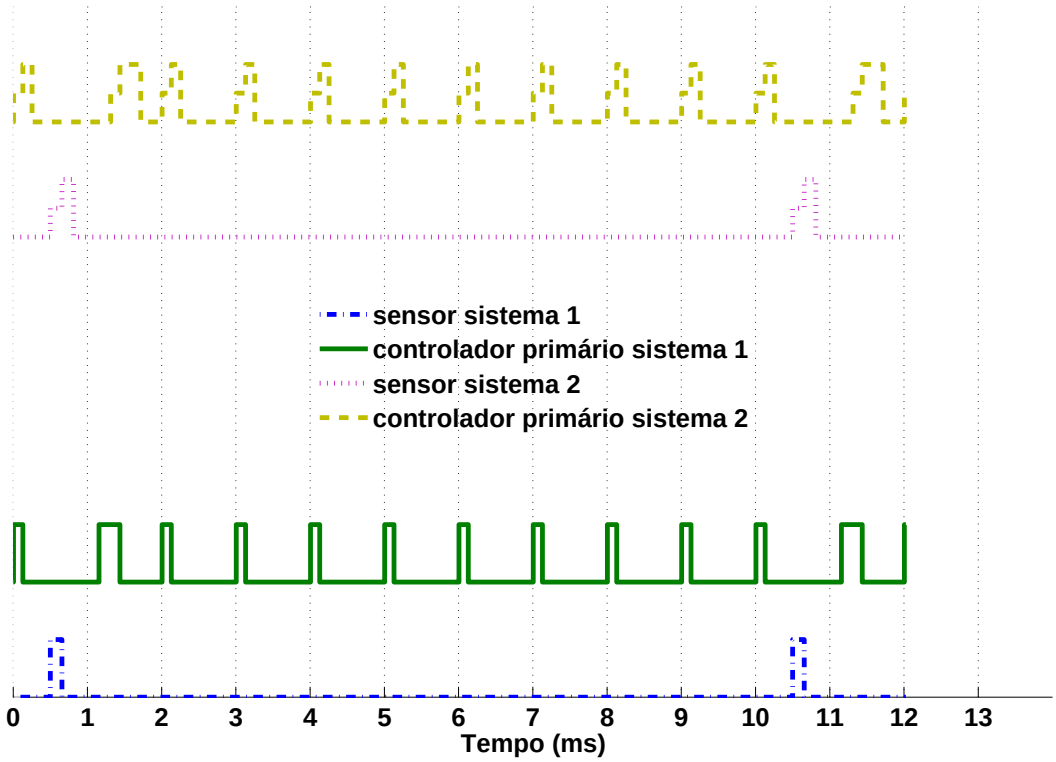


Figura 5.9. Escalonamento das mensagens na rede para um ambiente com 2 sistemas de controle($\Delta^i = 1ms$)

amostra para o controlador 1 (curva em verde – curva contínua), o que provocará, em alguns casos, no controlador 1, uma disputa entre a tarefa de controle e de emissão de *heartbeats* pelo uso do processador. Observa-se que os controladores entram em disputa pela ocupação do meio de comunicação a cada $\Delta^i = 1ms$. Nessa disputa, o controlador 2 (representado pela curva em amarelo e tracejada) é sempre penalizado. Esse controlador sofre uma punição maior em instantes em que o controlador primário necessita emitir uma ação de controle e, logo em seguida, mensagens *heartbeats* (ver intervalos entre 1 e $2ms$ e entre 11 e $12ms$).

A primeira avaliação efetiva do desempenho dos algoritmos de adaptação usados por cada detector é apresentada na tabela 5.3. Nessa tabela, é avaliado o número de falsas suspeitas cometidas por cada detector. Assim, são sintetizados os resultados obtidos para os ambientes com 1, 2 e 5 sistemas de controle. Como pode ser observado, a *RNA*

Tabela 5.3. Número de falsas suspeitas observadas na execução dos algoritmos de detecção para uma rede *CAN* com taxa de transferência de $500Kbps$ compartilhada por múltiplos sistemas.

Δ^i (ms)	Detector	Sistema 1	Sistema 2	Sistema 5
0.1	**Bertier**	38		
	Jacobson	21		
	RNA	21		
0.5	**Bertier**	65	66	
	Jacobson	80	80	
	RNA	0	0	
0.9	**Bertier**	46	66	41
	Jacobson	56	64	82
	RNA	32	64	12
1.0	**Bertier**	66	66	67
	Jacobson	80	80	80
	RNA	0	0	8
2.0	**Bertier**	0	121	132
	Jacobson	156	159	160
	RNA	0	0	80
5.0	**Bertier**	0	0	0
	Jacobson	0	0	0
	RNA	0	0	0

cometeu um menor número de falsas suspeitas na maioria das simulações realizadas. Isso denota a sua confiabilidade em relação aos demais algoritmos analisados.

Os demais algoritmos oscilam em termos de desempenho, mas, de modo geral, a abordagem adaptativa proposta por Bertier obteve um melhor desempenho, em termos de número de falsas suspeitas, quando comparado ao algoritmo adaptativo de Jacobson, mostrando-se mais confiável na maioria dos casos.

O algoritmo adaptativo de Bertier promove altas estimativas durante a fase de inicialização; isso faz com que o seu tempo máximo de detecção se eleve (ver tabela 5.6) e seu número inicial de falsas suspeitas seja bastante reduzido. Após a inicialização, entretanto,

o algoritmo de Bertier adapta suas estimativas pelo *janelamento* da média dos atrasos entre os últimos instantes de chegada. Assim, desconsiderando a margem de segurança, o algoritmo oscila em torno da média.

Conforme apresentado no capítulo 3, o algoritmo de Bertier usa a estratégia de Jacobson para ajustar a margem de segurança α. Sendo assim, após a sua inicialização, o desempenho da estratégia de Bertier dependerá da estratégia de Jacobson, e isso se torna mais evidente nos casos em que existem oscilações temporárias nos atrasos entre os instantes de chegada. Por conta disso, salvo os casos em que uma estimativa um pouco superior à média seja suficiente para evitar falsas suspeitas, o algoritmo de Bertier terá seu número de falsas suspeitas oscilando em conjunto com o algoritmo de adaptação de Jacobson; esse fato é evidenciado na tabela 5.3.

Na tabela 5.4 avalia-se o desempenho dos detectores em termos de tempo de recuperação para os casos nos quais o detector comete falsas suspeitas de detecção. Os valores apresentados na tabela estão em milissegundos e com precisão de duas casas decimais. Nessa tabela, também é apresentada uma síntese das simulações em cenários com 1, 2 e 5 sistemas de controle. As células da tabela preenchidas com um sinal de menos $(-)$ indicam a impossibilidade do cálculo do T_M por conta da inexistência de falsas suspeitas.

Focando na métrica T_M, pode-se observar que, além de cometer um número menor de falsas suspeitas, a abordagem baseada em *RNA*, na grande maioria dos casos, corrige os erros de detecção mais rapidamente que as demais abordagens de detecção.

A abordagem de Bertier foi, em geral, melhor em termos de T_M que a abordagem de Jacobson, principalmente em cenários com leves variações no atraso entre os instantes de chegada dos *heartbeats* ($\Delta^i = 2ms$). O contrário acontece com a abordagem baseada em *RNA*, que em cenários com variações moderadas em ΔA^{delay}, ele corrige suas suspeitas mais lentamente. Isso poderá ser justificado mais adiante quando observados os tempos de detecção, que para a adaptação com *RNA* são superiores às demais abordagens.

Uma outro quadro que resume a confiabilidade das abordagens de detecção é apre-

Tabela 5.4. Tempo de recuperação (T_M) observado na execução dos algoritmos de detecção para uma rede *CAN* com taxa de transferência de $500Kbps$ compartilhada por múltiplos sistemas. T_M^U, $\overline{T_M}$ e T_M^{std} referem-se, respectivamente, ao T_M máximo, médio e ao seu desvio padrão.

Δ^i (ms)	Detector	Sistema 1			Sistema 2			Sistema 5		
		T_M^U	$\overline{T_M}$	T_M^{std}	T_M^U	$\overline{T_M}$	T_M^{std}	T_M^U	$\overline{T_M}$	T_M^{std}
0.1	**Bertier**	0.10	0.04	0.03						
	Jacobson	0.16	0.14	0.01						
	RNA	0.09	0.08	0.00						
0.5	**Bertier**	0.32	0.20	0.12	0.25	0.24	0.02			
	Jacobson	0.16	0.13	0.04	0.30	0.23	0.07			
	RNA	-	-	-	-	-	-			
0.9	**Bertier**	0.51	0.26	0.15	0.47	0.19	0.14	0.90	0.39	0.27
	Jacobson	0.45	0.24	0.15	0.45	0.20	0.15	3.91	3.00	0.30
	RNA	0.37	0.27	0.06	0.53	0.21	0.17	4.61	1.74	1.50
1	**Bertier**	0.32	0.31	0.04	0.44	0.43	0.03	0.20	0.20	0.02
	Jacobson	0.34	0.34	0.00	0.46	0.46	0.00	2.29	2.29	0.00
	RNA	-	-	-	-	-	-	0.01	0.01	0.00
2	**Bertier**	-	-	-	0.01	0.01	0.00	0.42	0.42	0.03
	Jacobson	0.01	0.01	0.00	0.01	0.01	0.00	0.58	0.58	0.00
	RNA	-	-	-	-	-	-	4.62	3.66	0.70
5	**Bertier**	-	-	-	-	-	-	-	-	-
	Jacobson	-	-	-	-	-	-	-	-	-
	RNA	-	-	-	-	-	-	-	-	-

sentada na tabela 5.5, na qual verifica-se o desempenho dos detectores em termos do intervalo entre falsas suspeitas (T_{MR}). Os valores apresentados nessa tabela estão em milissegundos e com precisão de uma casa decimal. Assim como na tabela 5.3, é apresentada uma síntese das simulações em cenários com 1, 2 e 5 sistemas de controle. As células da tabela preenchidas com um sinal de menos ($-$) representam a impossibilidade do cálculo do T_{MR} por conta da inexistência de falsas suspeitas.

A métrica T_{MR} confirma a confiabilidade da abordagem baseada em RNA em relação

Tabela 5.5. Intervalo entre falsas suspeitas (T_{MR}) observado na execução dos algoritmos de detecção para uma rede *CAN* com taxa de transferência de $500Kbps$ compartilhada por múltiplos sistemas. T^{U}_{MR}, $\overline{T_{MR}}$ e T^{std}_{MR} referem-se, respectivamente, ao T_{MR} máximo, médio e ao seu desvio padrão.

Δ^i (ms)	Detector	Sistema 1			Sistema 2			Sistema 5		
		T^{U}_{MR}	$\overline{T_{MR}}$	T^{std}_{MR}	T^{U}_{MR}	$\overline{T_{MR}}$	T^{std}_{MR}	T^{U}_{MR}	$\overline{T_{MR}}$	T^{std}_{MR}
0.1	**Bertier**	9.8	2.7	3.7						
	Jacobson	10.0	6.2	3.4						
	RNA	10.0	6.2	3.4						
0.5	**Bertier**	9.2	5.0	4.3	8.9	4.9	3.9			
	Jacobson	9.2	5.0	4.3	8.9	5.0	3.9			
	RNA	-	-	-	-	-	-			
0.9	**Bertier**	29.2	12.7	6.6	29.2	8.9	7.6	50.5	16.5	11.1
	Jacobson	29.2	12.9	6.8	20.2	11.3	3.4	10.8	10.0	0.3
	RNA	60.0	22.9	22.2	19.1	11.2	3.5	19.8	6.7	4.2
1	**Bertier**	10.0	10.0	0.0	10.0	10.0	0.0	10.0	10.0	0.0
	Jacobson	10.0	10.0	0.0	10.0	10.0	0.0	10.0	10.0	0.0
	RNA	-	-	-	-	-	-	10.0	10.0	0.0
2	**Bertier**	-	-	-	10.0	10.0	0.0	10.0	10.0	0.0
	Jacobson	10.0	10.0	0.0	10.0	10.0	0.0	10.0	10.0	0.0
	RNA	-	-	-	-	-	-	6.0	3.3	1.9
5	**Bertier**	-	-	-	-	-	-	-	-	-
	Jacobson	-	-	-	-	-	-	-	-	-
	RNA	-	-	-	-	-	-	-	-	-

às demais abordagens analisadas. Através da tabela 5.5, mostra-se que, na grande maioria dos casos, a adaptação usando *RNA* possui altos valores de T_{MR} quando comparada aos demais, reforçando o fato de que a recorrência de erros do detector com o algoritmo de adaptação baseado em *RNA* acontece em intervalos maiores que os demais, ou seja, uma vez que o detector comete um erro de detecção, o intervalo para que um próximo erro aconteça é, na maioria dos casos, superior ao das demais abordagens.

Em resumo, as avaliações em termos de N^{fs}, T_M e T_{MR} demonstram a confiabilidade

do detector em relação aos demais detectores analisados, tendo esse um desempenho melhor, na grande maioria dos casos, nessas três métricas.

Tabela 5.6. Tempo de detecção (T_D) observado na execução dos algoritmos de detecção para uma rede *CAN* com taxa de transferência de $500Kbps$ compartilhada por múltiplos sistemas. T_D^U, $\overline{T_D}$ e T_D^{std} referem-se, respectivamente, ao T_D máximo, médio e ao seu desvio padrão.

Δ^i (ms)	Detector	Sistema 1			Sistema 2			Sistema 5		
		T_D^U	$\overline{T_D}$	T_D^{std}	T_D^U	$\overline{T_D}$	T_D^{std}	T_D^U	$\overline{T_D}$	T_D^{std}
0.1	**Bertier**	1.5	0.4	0.4						
	Jacobson	0.2	0.2	0.0						
	RNA	0.3	0.2	0.0						
0.5	**Bertier**	3.6	1.0	0.8	3.6	1.0	0.8			
	Jacobson	0.8	0.7	0.0	0.9	0.8	0.1			
	RNA	1.2	0.8	0.2	1.8	0.9	0.3			
0.9	**Bertier**	6.6	1.7	1.6	6.7	1.7	1.6	9.6	4.0	1.7
	Jacobson	1.4	1.1	0.1	1.3	1.1	0.1	4.0	2.9	0.4
	RNA	1.7	1.2	0.2	2.0	1.3	0.3	9.9	2.6	1.9
1	**Bertier**	7.2	2.0	1.7	7.3	2.0	1.7	7.7	3.6	1.4
	Jacobson	1.4	1.3	0.1	1.5	1.4	0.1	2.9	2.4	0.3
	RNA	2.1	1.2	0.4	2.4	1.3	0.5	7.6	2.0	2.1
2	**Bertier**	14.5	3.6	3.6	14.5	3.6	3.6	14.7	4.7	3.2
	Jacobson	2.0	2.0	0.0	2.0	2.0	0.0	3.9	3.7	0.2
	RNA	2.1	2.1	0.0	2.1	2.1	0.0	2.2	0.2	1.6
5	**Bertier**	36.1	5.9	4.4	36.1	5.9	4.4	36.6	7.5	4.1
	Jacobson	5.0	5	0	5.0	5	0	8.3	8.3	0.1
	RNA	5.5	5.5	0.0	5.5	5.5	0.0	7.1	5.3	1.7

A tabela 5.6 apresenta o desempenho dos algoritmos de adaptação em função do tempo de detecção. Todos os valores sintetizados na tabela estão em milissegundos e com precisão de uma casa decimal. Assim como na tabela 5.3, essa tabela apresenta uma síntese das simulações em cenários com 1, 2 e 5 sistemas de controle.

Tomando como base o tempo de detecção, conforme tabela 5.6, o algoritmo de Jacob-

son possui um melhor desempenho, haja vista que, em geral, consegue tempos máximos e médios de detecção menores que os demais. O algoritmo de Bertier, entretanto, apresentou os maiores tempos de detecção quando comparado aos demais algoritmos. De acordo com o discutido anteriormente, essa abordagem é prejudicada pelo procedimento usado durante a fase de inicialização do algoritmo de adaptação.

Em termos de tempo de detecção, a *RNA* obteve um desempenho intermediário em relação às demais abordagens de adaptação. Todavia, os tempos máximos e médios de detecção apresentados pela *RNA* estão relativamente próximos daqueles produzidos pela abordagem de Jacobson. Na maioria dos casos, o tempo de detecção da abordagem baseada em *RNA* foi superior à abordagem de Jacobson cerca de 5% a 50% quando observados os tempos máximos de detecção, e 0% a 50% quando analisados os tempos médios.

Os valores mais altos, em termos de tempo de detecção, produzidos pela *RNA*, são calculados nos cenários em que os atrasos se tornam mais *moderados*. Isso se deve em parte à estratégia usada durante o ajuste dos parâmetros da *RNA*, o qual é realizado de modo que a Rede Neural sugira valores um pouco acima daqueles sugeridos em seu padrão de treinamento, evitando, dessa forma, falsas suspeitas, mas incrementando o tempo de detecção. Essa estratégia melhor se adequa a cenários nos quais os atrasos são bastante *severos*.

5.2.6.2 Avaliando o Impacto da Implementação dos Detectores sobre o Desempenho do Controle. Nas simulações realizadas, avaliou-se também o impacto da implementação das abordagens de detecção de defeitos sobre o desempenho dos sistemas de controle. Para tanto, foram considerados cenários nos quais a rede *CAN* era compartilhada por 1, 2 e 5 sistemas de controle. Nas avaliações, deve ser considerado o impacto gerado pela carga adicional de processamento e comunicação provocada pelo mecanismo

de detecção de defeitos.

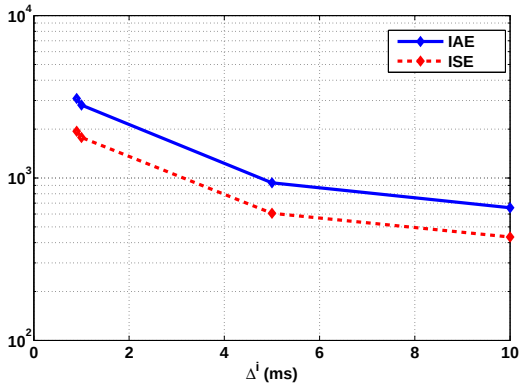
É importante ressaltar que as análises realizadas nesta subseção consideram o desempenho no pior caso. Isso quer dizer que nos diversos cenários foram coletados e calculados os índices de erro, o jitter e o desempenho do sistema de controle cujos dispositivos possuíam as menores prioridades.

As sobrecargas de processamento e de comunicação estão diretamente relacionadas ao período de emissão de *heartbeats* (Δ^i) usado na configuração do detector. Quanto menor Δ^i, maior será o percentual de utilização da CPU do dispositivo de controle e também maior será o número de mensagens a serem enviadas na rede. Por outro lado, períodos de emissão muito grandes podem comprometer o controle realizado na ocorrência de uma falha, uma vez que esse período de emissão é um parâmetro determinante para a magnitude do tempo de detecção.

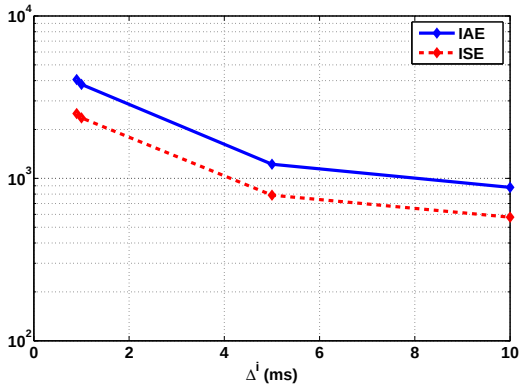
Dado que a tarefa de controle foi configurada com uma prioridade maior que a tarefa de emissão de *heartbeats*, a emissão de *heartbeats* não interfere no processamento da tarefa de controle. Entretanto, o mesmo não pode ser afirmado no que diz respeito à sobrecarga de comunicação.

Os gráficos das figuras 5.10(a), 5.10(b) e 5.10(c) utilizam os índices IAE e ISE para sintetizar a influência do período de emissão de *heartbeats* no desempenho do sistema de controle. Nesses gráficos, a magnitude dos índices de erro é apresentada usando uma escala logarítmica.

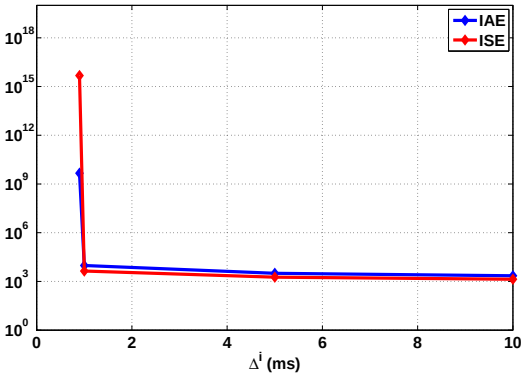
Em todos os gráficos, os índices de erro diminuem com o aumento do período de emissão de *heartbeats*. Além disso, pode-se notar que, com o aumento do número de sistemas compartilhando o mesmo meio de comunicação, o controle se torna mais sensível à redução de Δ^i. Para um ambiente com 1 ou 2 sistemas de controle e $\Delta^i < 2$, os erros são da ordem de 10^3 (ver figuras 5.10(a) e 5.10(b)). Para um ambiente com 5 sistemas com $\Delta^i < 2$, entretanto, os índices de erro crescem consideravelmente, chegando a atingir valores superiores a 10^{15}.



(a) Rede *CAN* com 1 Sistema



(b) Rede *CAN* com 2 Sistemas



(c) Rede *CAN* com 5 Sistemas

Figura 5.10. Comportamento dos índices de erro *IAE* e *ISE* em função do período de emissão Δ^i nas simulações com rede *CAN*

Além dos índices clássicos de desempenho, avaliou-se o impacto da implementação do detector sobre os sistemas de controle usando o conceito de *margem de jitter* (J_m). Na avaliação, é necessário considerar um atraso mínimo de $322.5\mu s$, equivalente ao somatório

dos tempos mínimos de execução do processo de sensoriamento (configurado na simulação como $5\mu s$), de execução da tarefa de controle (configurado como $5\mu s$), de transferência da mensagem entre sensor-controlador ($156.25\mu s$) e entre controlador-atuador ($156.25\mu s$).

Nas simulações, obteve-se $J_m(322.5\mu s) = 9.7ms$, ou seja, $9.7ms$ representa a máxima variação permitida para o atraso. Assim, para os cenários com 1 e 2 sistemas, o controle permanece estável para períodos de emissão de *heartbeats* superiores a $0.9ms$. Para $\Delta^i > 0.9ms$, o *jitter* observado é $1.3ms$ e $1.6ms$ para cenários com 1 e 2 sistemas de controle, respectivamente. Para um cenário com 5 sistemas de controle, por outro lado, esse período de emissão deve ser superior a $1ms$, para o qual se obtem um $jitter \leq 5.9$. Quando o *jitter* é superior ao apontado pela *margem de jitter*, o sistema de controle é instável, caso contrário o controle apresenta um comportamento estável.

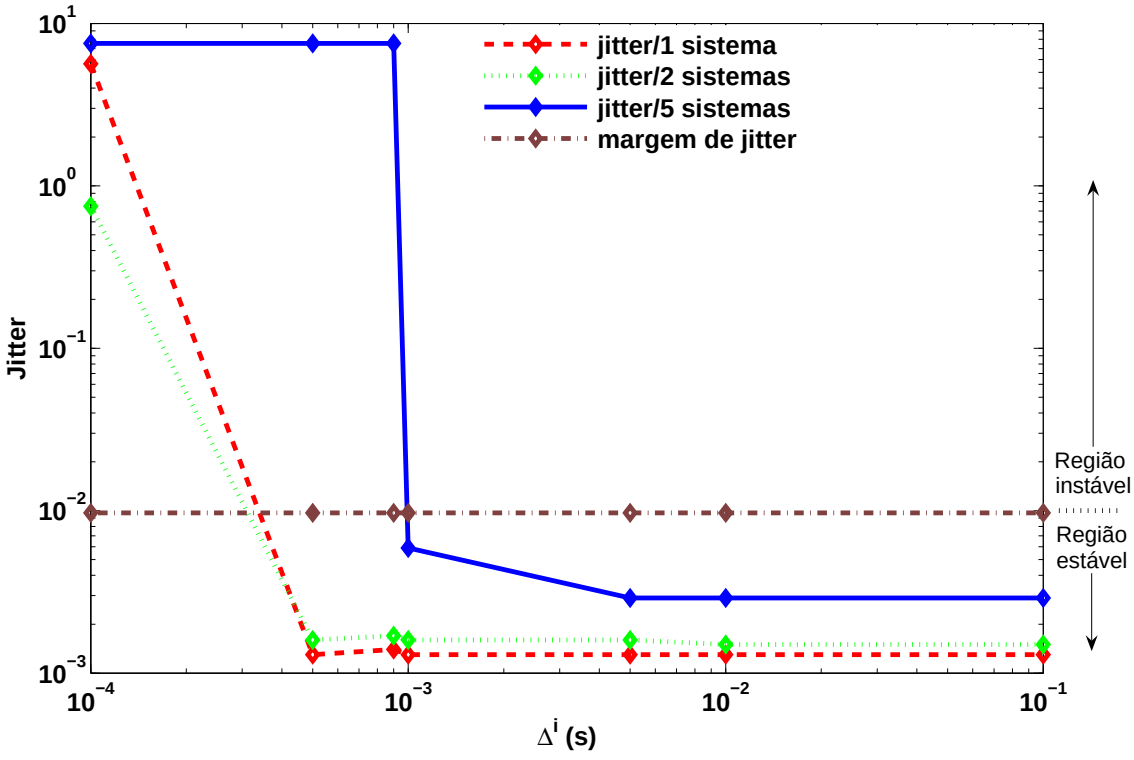


Figura 5.11. Jitter observado nas simulações com rede *CAN*

A figura 5.11 apresenta o *jitter* observado com a variação do período de emissão de

heartbeats. Os cenários com 1, 2 e 5 sistemas de controle são representados por curvas em vermelho, em verde e em azul, respectivamente. A curva em amarelo representa o *jitter* máximo permitido, calculado através do conceito de margem de *jitter*. O *jitter* e o período Δ^i são apresentados no gráfico usando escalas logarítmicas.

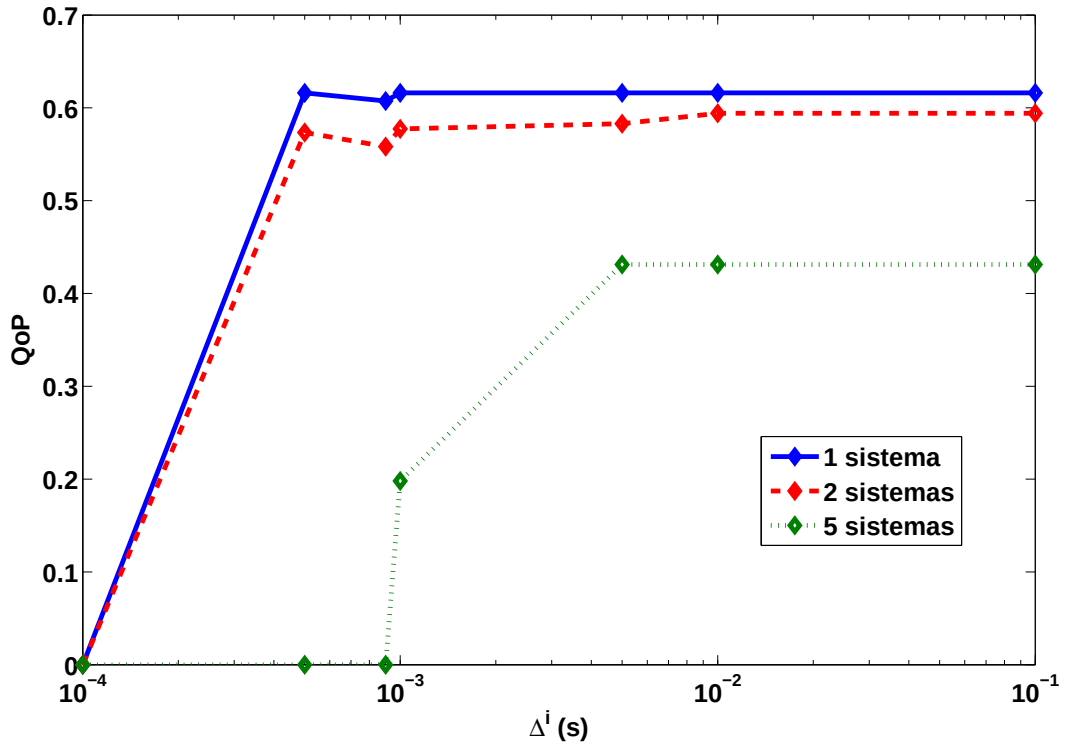


Figura 5.12. Qualidade do desempenho do controle medido ($QoP = \frac{\overline{\varphi}}{\varphi}$) nas simulações com rede *CAN*

Por fim, a figura 5.12 apresenta um gráfico através do qual se pode avaliar o desempenho do controle com a variação de Δ^i. Os valores utilizados no gráfico para representar o período de emissão de *heartbeats* são apresentados em escala logarítmica.

O índice de desempenho QoP[9] é medido usando a relação entre a *margem de fase* (φ) e a *margem de fase aparente*[10] ($\overline{\varphi}$) do sistema de controle, conforme recomendação de Cervin et al. (2004). A curva em azul representa a QoP para um cenário no qual a rede de

[9] ver seção 2.4
[10] Ver seção 2.2.1.1

comunicação é compartilhada com apenas 1 sistema de controle, enquanto as curvas em vermelho e verde representam a QoP para cenários com 2 e 5 sistemas, respectivamente.

Como pode ser observado na figura 5.12, a qualidade do desempenho do controle aumenta na medida em que o Δ^i é incrementado. Para períodos de emissão de *heartbeats* iguais ou inferiores a $0.1ms$, tem-se uma $QoP = 0$, significando que o sistema é instável, isto para os cenários com 1 e 2 sistemas de controle. Para cenários com 5 sistemas, por outro lado, o controle possui desempenho zero quando utilizados períodos de emissão de *heartbeats* iguais ou inferiores a $1ms$.

5.2.7 Avaliando o Desempenho dos Detectores em uma Rede Switched-Bus-Ethernet

Uma vez avaliada uma rede industrial de fato como a rede *CAN*, nesta subseção, e na próxima, avalia-se o desempenho dos detectores de defeitos e o impacto de sua implementação na qualidade de *NCS* com dispositivos interconectados através de uma rede de prateleira como a *Ethernet/CSMA-CD*.

Nessa subseção, em particular, serão avaliadas as redes do tipo *Switched-Bus-Ethernet* (TANENBAUM, 2003; SONG; KOUBAA; SIMONOT, 2002), as quais, diferentemente das redes CAN, não estabelecem uma política de acesso ao meio baseada em prioridades. Todavia, suas altas taxas de transferência e a capacidade de isolamento de tráfego entre pares de dispositivos, torna a *Switched-Bus-Ethernet* bastante atraente para implementação de sistemas de controle distribuídos ou sobre rede (SONG; KOUBAA; SIMONOT, 2002).

Na rede *Switched-Bus-Ethernet*, cada nó possui uma conexão com um dispositivo comutador central (*switch*). O *Switch* armazena as mensagens em um *buffer* e, então, as encaminha aos dispositivos para os quais foram destinadas(TANENBAUM, 2003). Quando mais de uma mensagem é transmitida para um mesmo nó, cada uma dessas mensagens é enfileirada em ordem *FIFO*. Em cenários com alto tráfego, as filas de mensagens podem se tornar longas e a capacidade do *buffer* pode ser excedida, necessitando que a mensagem

seja descartada ou retransmitida.

As simulações usando as redes *Switched-Bus-Ethernet* consideram os mesmos cenários que foram adotados para as simulações com redes CAN. Entretanto, não existe qualquer prioridade associada às mensagens transmitidas durante a comunicação entre os dispositivos.

As avaliações consideram uma rede *Switched-Bus-Ethernet* com taxa de transferência de $10Mbps$. As mensagens enviadas através da rede possuem um tamanho mínimo de 64 *bytes*. Além disso, utiliza-se um *buffer* para enfileiramento das mensagens de $2MB$ de capacidade. Em caso de *buffer cheio*, as mensagens serão descartadas. Na implementação dos detectores de defeitos foram considerados períodos de emissão de *heartbeats* $\Delta^i =$ 0.1, 0.5, 0.9, 1.0, 2.0, e 5.0, todos medidos em milissegundos.

Nos cenários avaliados, não existe comunicação entre dispositivos de diferentes sistemas de controle. Sendo assim, o tráfego gerado por um sistema não interfere no tráfego gerado pelos demais. Por conta disso, todos os sistemas de controle percebem os mesmos atrasos de comunicação. Dessa forma, os dados contidos nas tabelas e nos gráficos apresentados a seguir, podem ser extraídos de qualquer um dos sistemas de controle envolvidos na simulação.

5.2.7.1 Avaliando o desempenho dos algoritmos de adaptação usados nos detectores de defeitos.

Considerando quadros com 64 *bytes* e uma taxa de transferência de $10Mbps$, pode-se verificar que cada mensagem será transmitida em aproximadamente $51.2\mu s$. Portanto, períodos de emissão de heartbeats inferiores à $51.2\mu s$ provocarão um enfileiramento excessivo no *buffer* e farão com que algumas mensagens sejam descartadas.

A ausência de uma política de priorização para as mensagens torna o acesso ao meio mais democrático. Sendo assim, diferentemente da rede CAN, na *Switched-Bus-Ethernet* todos os sistemas são penalizados com atrasos de comunicação com magnitudes aproxi-

madamente equivalentes. Dessa forma, o desempenho dos detectores depende apenas do seu algoritmo de adaptação.

Para a rede *Switched-Ethernet*, a primeira avaliação de desempenho dos algoritmos de adaptação usados por cada detector é apresentada na tabela 5.7. Nessa tabela é avaliado o número de falsas suspeitas cometido por cada detector, sendo sintetizados os resultados obtidos para os ambientes com 1, 2 e 5 sistemas de controle. Como pode ser observado, na maioria das simulações, a RNA cometeu o menor número de falsas suspeitas; isso denota a sua confiabilidade em relação aos demais algoritmos analisados.

Tabela 5.7. Número de falsas suspeitas observadas na execução dos algoritmos de detecção para uma rede *Switched-Ethernet* com taxa de transferência de $10Mbps$ compartilhada por múltiplos sistemas.

Δ^i (ms)	Detector	Sistema 1	Sistema 2	Sistema 5
0.1	**Bertier**	34	62	62
	Jacobson	4	8	8
	RNA	4	8	8
0.5	**Bertier**	32	32	32
	Jacobson	40	40	40
	RNA	0	0	0
0.9	**Bertier**	32	32	32
	Jacobson	40	40	40
	RNA	0	0	0
1.0	**Bertier**	64	64	67
	Jacobson	80	80	80
	RNA	0	0	0
2.0	**Bertier**	0	0	0
	Jacobson	0	0	0
	RNA	0	0	0
5.0	**Bertier**	0	0	0
	Jacobson	0	0	0
	RNA	0	0	0

Os demais algoritmos oscilam em termos de desempenho, mas, de modo geral, a

abordagem adaptativa proposta por Bertier obteve um melhor desempenho em termos do número de falsas, suspeitas quando comparado ao algoritmo adaptativo de Jacobson, se mostrando mais confiável na grande maioria dos casos. Por outro lado, no cenário de pior caso ($\Delta^i = 0.1ms$) o algoritmo de Jacobson apresentou um desempenho melhor que o algoritmo de Bertier.

Um efeito interessante, que também acontece nos demais cenários, mas que é mais facilmente percebido na rede *Switched-Ethernet*, é o da escolha do período de emissão de *heartbeats* Δ^i no atraso da rede. Analisando a tabela 5.7, pode-se perceber que para $\Delta^i = 1ms$, o número de falsas suspeitas é, aproximadamente, o dobro do número de falsas suspeitas cometidas para $\Delta^i < 1ms$. O efeito se justifica pelo fato do período de emissão de *heartbeats* ser múltiplo do período de amostragem h usado pelo sistema de controle. Sendo assim, a cada $h = 10ms$, as mensagens de *heartbeats* e as mensagens de controle competem pelo uso do *buffer* do *switch*. Como, no dispositivo controlador, a tarefa de controle tem maior prioridade, a cada $10ms$, aproximadamente, um *heartbeat* terá sua transmissão atrasada e, por conta disso, observará um atraso maior que o atraso dos demais *heartbeats*. Uma vez que o atraso na rede sofre apenas pequenas oscilações, as abordagens de Jacobson e de Bertier calculam um valor em torno da média. Sendo assim, nesses instantes, acontecerá uma falsa suspeita adicional. Para um período de $0.9ms$, por outro lado, uma competição pelo meio de comunicação só acontecerá a cada $90ms$. O mesmo efeito observado para o período de emissão de *heartbeats* $\Delta^i = 1ms$ deveria ser observado para $\Delta^i = 0.1ms$ e para $\Delta^i = 0.5ms$. Entretanto, esses períodos de emissão sofrem influência do escalonamento da tarefa no dispositivo. O atraso de computação para esses períodos de emissão de *heartbeats* é mais severo. Isso faz com que a tarefa de emissão de *heartbeat* tenha sua execução atrasada, deslocando o instante de emissão do *heartbeat* e, conseqüentemente, diminuindo as competições pelo meio de comunicação. Com isso, o número de falsas suspeitas é reduzido. Os tempos de resposta das tarefas

de emissão de *heartbeats* para $\Delta^i = 0.1ms$ e $\Delta^i = 0.5ms$ são idênticos[11] e promovem competições pelo canal de comunicação em períodos superiores a $10ms$.

Tabela 5.8. Tempo de recuperação (T_M) observado na execução dos algoritmos de detecção para uma rede *Switched-Ethernet* com taxa de transferência de $10Mbps$ compartilhada por múltiplos sistemas. T_M^U, $\overline{T_M}$ e T_M^{std} referem-se, respectivamente, ao T_M máximo, médio e ao seu desvio padrão.

Δ^i (ms)	Detector	Sistema 1			Sistema 2			Sistema 5		
		T_M^U	$\overline{T_M}$	T_M^{std}	T_M^U	$\overline{T_M}$	T_M^{std}	T_M^U	$\overline{T_M}$	T_M^{std}
0.1	**Bertier**	0.48	0.09	0.15	0.49	0.10	0.16	0.49	0.10	0.16
	Jacobson	0.49	0.49	0.00	0.49	0.49	0.00	0.49	0.49	0.00
	RNA	0.27	0.27	0.00	0.37	0.37	0.00	0.37	0.37	0.00
0.5	**Bertier**	0.08	0.08	0.00	0.08	0.08	0.00	0.08	0.08	0.00
	Jacobson	0.09	0.09	0.00	0.09	0.09	0.00	0.09	0.09	0.00
	RNA	-	-	-	-	-	-	-	-	-
0.9	**Bertier**	0.39	0.24	0.11	0.39	0.24	0.11	0.39	0.24	0.11
	Jacobson	0.41	0.25	0.11	0.41	0.25	0.11	0.41	0.25	0.11
	RNA	-	-	-	-	-	-	-	-	-
1	**Bertier**	0.06	0.06	0.01	0.06	0.06	0.01	0.06	0.06	0.01
	Jacobson	0.07	0.07	0.01	0.07	0.07	0.01	0.07	0.07	0.00
	RNA	-	-	-	-	-	-	-	-	-
2	**Bertier**	-	-	-	-	-	-	-	-	-
	Jacobson	-	-	-	-	-	-	-	-	-
	RNA	-	-	-	-	-	-	-	-	-
5	**Bertier**	-	-	-	-	-	-	-	-	-
	Jacobson	-	-	-	-	-	-	-	-	-
	RNA	-	-	-	-	-	-	-	-	-

Na tabela 5.8, avalia-se o desempenho dos detectores em termos de tempo de recuperação para os casos nos quais o detector comete falsas suspeitas de detecção. Os valores apresentados na tabela estão em milissegundos e com precisão de duas casas decimais.

[11] Isso pode ser verificado calculando o tempo de resposta da tarefa de emissão de *heartbeats*. Tal cálculo pode ser feito usando o teorema para a obtenção do tempo de resposta desenvolvido por Joseph e Pandya (1986) e a estratégia para a resolução desse teorema desenvolvida por Audsley et al. (1993).

Nessa tabela, também é apresentada uma síntese das simulações em cenários com 1, 2 e 5 sistemas de controle. As células da tabela preenchidas com um sinal de menos (−) indicam a impossibilidade do cálculo do T_M por conta da inexistência de falsas suspeitas.

Assim como foi percebido nas redes *CAN*, observa-se que, além de cometer um número menor de falsas suspeitas, a abordagem baseada em *RNA*, na grande maioria dos casos, corrige os erros de detecção mais rapidamente que as demais abordagens de detecção. A abordagem de Bertier obteve um melhor desempenho em termos de T_M que a abordagem de Jacobson.

Na tabela 5.9, verifica-se o desempenho dos detectores em termos do intervalo entre falsas suspeitas (T_{MR}). Os valores apresentados nessa tabela estão em milissegundos e com precisão de uma casa decimal. A tabela 5.9 apresenta uma síntese das simulações em cenários com 1, 2 e 5 sistemas de controle. As células da tabela preenchidas com um sinal de menos (−) representam a inviabilidade do cálculo do T_{MR} por conta da inexistência de falsas suspeitas.

Assim como foi observado na rede *CAN*, a métrica T_{MR} confirma a confiabilidade da abordagem baseada em RNA em relação às demais abordagens analisadas. A tabela 5.9 mostra que a adaptação usando RNA possui valores de T_{MR} levemente superiores quando comparada às demais em cenários em que o detector é configurado para utilizar um período de emissão de *heartbeats* $\Delta^i = 0.1ms$. Nos demais casos, o T_{MR} não pode ser calculado, posto que a *RNA* não cometeu falsas suspeitas.

Por fim, as simulações em redes *Switched-Bus-Ethernet* confirmam a confiabilidade do detector baseado em RNA, em termos das métricas N^{fs}, T_M e T_{MR}, quando comparado aos demais detectores.

A tabela 5.10 apresenta o desempenho dos algoritmos de adaptação em função do tempo de detecção. Todos os valores sintetizados na tabela estão em milissegundos e com precisão de uma casa decimal. Assim como na tabela 5.7, essa tabela apresenta uma síntese das simulações em cenários com 1, 2 e 5 sistemas de controle.

Tabela 5.9. Intervalo entre falsas suspeitas (T_{MR}) observado na execução dos algoritmos de detecção para uma rede *Switched-Ethernet* com taxa de transferência de $10Mbps$ compartilhada por múltiplos sistemas. T_{MR}^{U}, $\overline{T_{MR}}$ e T_{MR}^{std} referem-se, respectivamente, ao T_{MR} máximo, médio e ao seu desvio padrão.

Δ^i (ms)	Detector	Sistema 1			Sistema 2			Sistema 5		
		T_{MR}^{U}	$\overline{T_{MR}}$	T_{MR}^{std}	T_{MR}^{U}	$\overline{T_{MR}}$	T_{MR}^{std}	T_{MR}^{U}	$\overline{T_{MR}}$	T_{MR}^{std}
0.1	**Bertier**	8.4	0.9	2.4	7.3	1.0	2.2	7.3	1.0	2.2
	Jacobson	10.0	10.0	0.0	10.0	10.0	0.0	10.0	10.0	0.0
	RNA	10.0	10.0	0.0	10.0	10.0	0.0	10.0	10.0	0.0
0.5	**Bertier**	10.0	10.0	0.0	10.0	10.0	0.0	10.0	10.0	0.0
	Jacobson	10.0	10.0	0.0	10.0	10.0	0.0	10.0	10.0	0.0
	RNA	-	-	-	-	-	-	-	-	-
0.9	**Bertier**	50.0	17.7	16.1	50.0	17.7	16.1	50.0	17.7	16.1
	Jacobson	50.0	17.2	15.6	50.0	17.2	15.6	50.0	17.2	15.6
	RNA	-	-	-	-	-	-	-	-	-
1	**Bertier**	10.0	10.0	0.0	10.0	10.0	0.0	10.0	10.0	0.0
	Jacobson	10.0	10.0	0.0	10.0	10.0	0.0	10.0	10.0	0.0
	RNA	-	-	-	-	-	-	-	-	-
2	**Bertier**	-	-	-	-	-	-	-	-	-
	Jacobson	-	-	-	-	-	-	-	-	-
	RNA	-	-	-	-	-	-	-	-	-
5	**Bertier**	-	-	-	-	-	-	-	-	-
	Jacobson	-	-	-	-	-	-	-	-	-
	RNA	-	-	-	-	-	-	-	-	-

Tomando como base o tempo de detecção, conforme tabela 5.10, o algoritmo de Jacobson possui um melhor desempenho, haja vista que, em geral, consegue tempos máximos e médios de detecção menores que os demais. O algoritmo de Bertier, entretanto, apresentou os maiores tempos de detecção quando comparados aos demais algoritmos. De acordo com o discutido anteriormente, essa abordagem é prejudicada pelo procedimento usado durante a fase de inicialização do algoritmo de adaptação.

Novamente, em termos de tempo de detecção, a abordagem baseada em *RNA* obteve

Tabela 5.10. Tempo de detecção (T_D) observado na execução dos algoritmos de detecção para uma rede *Switch-Bus-Ethernet* com taxa de transferência de $10Mbps$ e compartilhada por múltiplos sistemas. T_D^U, $\overline{T_D}$ e T_D^{std} referem-se, respectivamente, ao T_D máximo, médio e ao seu desvio padrão.

Δ^i (ms)	Detector	Sistema 1			Sistema 2			Sistema 5		
		T_D^U	$\overline{T_D}$	T_D^{std}	T_D^U	$\overline{T_D}$	T_D^{std}	T_D^U	$\overline{T_D}$	T_D^{std}
0.1	**Bertier**	0.7	0.3	0.2	0.8	0.2	0.2	0.8	0.2	0.2
	Jacobson	0.3	0.1	0.0	0.3	0.1	0.0	0.3	0.1	0.0
	RNA	0.6	0.3	0.1	0.6	0.2	0.1	0.6	0.2	0.1
0.5	**Bertier**	3.6	0.9	0.9	3.6	0.9	0.9	3.6	0.9	0.9
	Jacobson	0.5	0.5	0.0	0.5	0.5	0.0	0.5	0.5	0.0
	RNA	1.6	1.5	0.1	1.0	1.0	0.1	1.1	1.0	0.1
0.9	**Bertier**	6.5	1.7	1.6	6.5	1.7	1.6	6.5	1.7	1.6
	Jacobson	1.2	1.0	0.1	1.2	1.0	0.1	1.2	1.0	0.1
	RNA	3.2	2.7	0.3	2.1	1.7	0.1	2.1	1.7	0.1
1	**Bertier**	7.2	1.8	1.8	7.2	1.8	1.8	7.2	1.8	1.1
	Jacobson	1.0	0.0	0.0	1.1	1.0	0.0	1.1	1.0	0.0
	RNA	3.1	3.0	0.1	1.9	1.8	0.1	1.9	1.8	0.1
2	**Bertier**	14.5	3.6	3.6	14.5	3.6	3.6	14.5	3.6	3.6
	Jacobson	2.0	2.0	0.0	2.0	2.0	0.0	2.0	2.0	0.0
	RNA	2.0	2.0	0.0	2.0	2.0	0.0	2.0	2.0	0.0
5	**Bertier**	36.2	28.5	4.1	36.2	8.9	8.9	36.6	8.9	8.9
	Jacobson	7.3	5.2	0.5	5.0	5.0	0.0	5.0	5.0	0.0
	RNA	5.0	5.0	0.0	5.0	5.0	0.0	5.0	5.0	0.0

um desempenho intermediário em relação às demais abordagens de adaptação. Os tempos máximos e médios de detecção apresentados pelo algoritmo com *RNA* são relativamente altos quando comparados ao do algoritmo de Jacobson, chegando, em alguns casos, a ser entre 100% e 300% superior. Ao contrário do observado quando se avaliou a abordagem em uma rede *CAN*, a *RNA* consegue tempos de detecção equivalentes à abordagem de Jacobson em cenários com carga leve ou moderada ($\Delta^i > 1ms$).

5.2.7.2 Avaliando o Impacto da Implementação dos Detectores sobre o Desempenho do Controle. Nas simulações realizadas, avaliou-se também o impacto da implementação das abordagens de detecção de defeitos sobre o desempenho dos sistemas de controle. Para tanto, foram considerados cenários nos quais a rede *Switched-Bus-Ethernet* era compartilhada por 1, 2 e 5 sistemas de controle.

A mesma análise realizada para a rede *CAN* para a sobrecarga de processamento também é válida para a rede *Switched-Bus-Ethernet*. Por outro lado, o atraso de comunicação possui um perfil diferenciado.

Em termos dos índices de desempenho IAE e ISE foram obtidos aproximadamente os mesmos resultados para os cenários com 1, 2 e 5 sistemas de controle. O desempenho em termos de tais índices, entretanto, aumenta com o aumento do período de emissão de *heartbeats* (Δ^i).

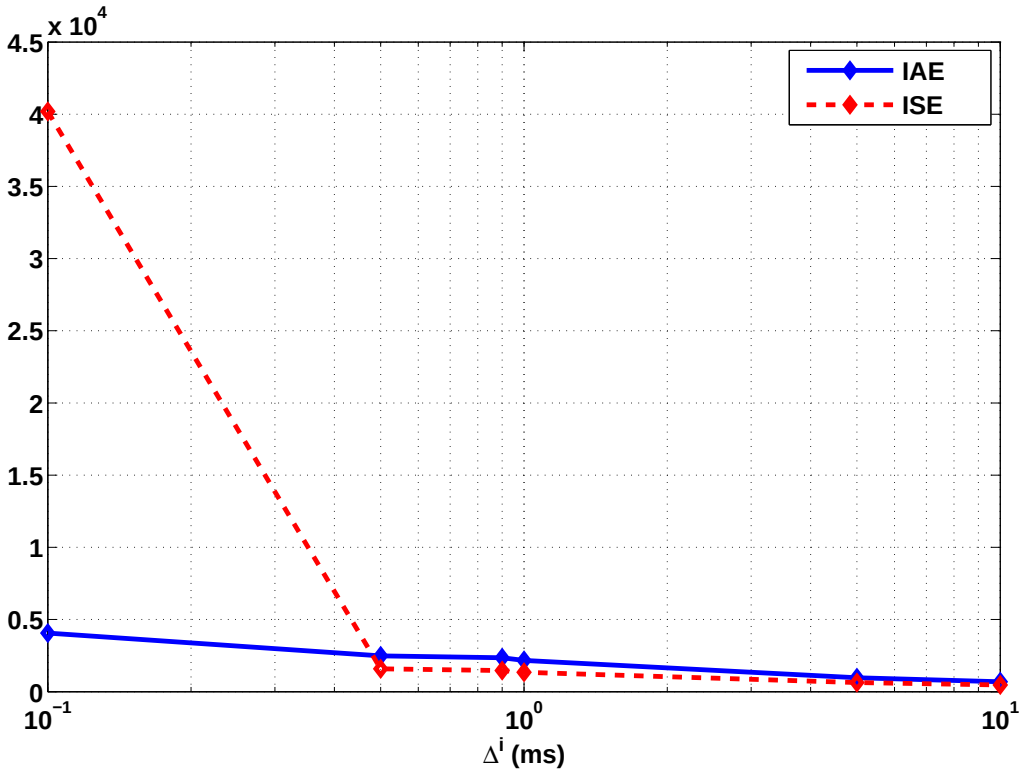


Figura 5.13. Comportamento dos índices de erro IAE e ISE em função do período de emissão Δ^i nas simulações com rede *Switched-Bus-Ethernet*

A figura 5.13 apresenta o comportamento dos índices de desempenho com o incremento de Δ^i. Nessa figura, o período de emissão de *heartbeats* é apresentado em escala logarítmica.

Na avaliação do impacto no desempenho usando o conceito de *margem de jitter*, é necessário considerar um atraso mínimo de $112.4\mu s$, equivalente ao somatório dos tempos mínimos de execução do processo de sensoriamento (configurado na simulação como $5\mu s$), de execução da tarefa de controle (configurado como $5\mu s$), de transferência da mensagem entre sensor-controlador ($51.2\mu s$) e entre controlador-atuador ($51.2\mu s$).

Nas simulações, obteve-se $J_m(112.4\mu s) = 9.7ms$, o qual representa a máxima variável permitida para o atraso. Para todos os cenários, o *jitter* observado foi de $1.1ms$ e todos os sistemas de controle permaneceram estáveis, independente do período de emissão de *heartbeats* utilizado. O índice de desempenho em todas as simulações permaneceu constante e equivalente a $QoP = 0.65$.

5.2.8 Avaliando o Desempenho dos Detectores em uma Rede Shared-Bus-Ethernet

Nessa subseção, avalia-se o desempenho dos detectores de defeitos e o impacto de sua implementação na construção de *NCS* sobre *Shared-Bus-Ethernet* (SCHNEIDER; PARDO-CASTOLLOTE; HAMILTON, 1999; TANENBAUM, 2003).

Na *Shared-Bus-Ethernet*, não há uma disciplina para acesso ao meio de comunicação; antes de enviar uma mensagem, os dispositivos verificam se o meio está livre e, em caso afirmativo, iniciam a transmissão. Quando dois ou mais dispositivos, interessados em enviar uma mensagem, encontram o meio de comunicação disponível e, simultaneamente, iniciam uma transmissão, ocorre uma colisão. Após uma colisão, os dispositivos cessam imediatamente suas transmissões e utilizam um protocolo para solucionar a disputa pelo acesso ao meio denominado *recuo binário exponencial* (*BEB*, *Binary Exponential Backoff*). No *BEB*, após uma colisão, cada dispositivo sorteia um número aleatório de *slots*

de tempo com tamanho fixo e então tenta uma nova transmissão após um período de acordo com o número de *slots* sorteados (TANENBAUM, 2003). Se uma nova colisão acontece, o número de slots de tempo é incrementado e uma nova tentativa é realizada. Esse procedimento se repetirá até que um dispositivo consiga acessar o meio de comunicação ou até que o número máximo de tentativas tenha se esgotado e o dispositivo cancele a transmissão. Isso implica, portanto, em possíveis perdas de mensagens. O procedimento *BEB* faz com que o atraso em uma rede *Shared-Bus-Ethernet* tenha um comportamento não determinístico.

As simulações usando as redes *Shared-Bus-Ethernet* consideram os mesmos cenários adotados nas demais simulações, isso observando o papel e a interação entre os dispositivos. Assim como na rede *Switched-Ethernet*, as avaliações consideram uma taxa de transferência de $10Mbps$ e as mensagens enviadas através da rede possuem um tamanho mínimo de 64 *bytes*. A implementação dos detectores de defeitos consideram períodos de emissão de *heartbeats* $\Delta^i = 0.1$, 0.5, 0.9, 1.0, 2.0, e 5.0, todos medidos em milissegundos.

Na rede *Shared-Bus-Ethernet*, dependendo do desenrolar da disputa pelo acesso ao meio, os sistemas podem observar diferentes magnitudes para o atraso de comunicação. Sendo assim, os dados apresentados nas tabelas e gráficos a seguir se referem ao sistema que foi submetido às piores condições de tráfego.

5.2.8.1 Avaliando o Desempenho dos Algoritmos de Adaptação Usados nos Detectores de Defeitos.

Dentre as redes analisadas, a rede *Shared-Bus-Ethernet* é a mais democrática, e o intervalo de tempo necessário para a transmissão de uma mensagem dependerá dos resultados dos *sorteios* realizados durante as disputas pelo acesso ao meio de comunicação.

Seguindo a mesma abordagem utilizada na análise das demais redes de comunicação, avalia-se o desempenho dos algoritmos de adaptação usados por cada detector em termos

do número de falsas suspeitas cometidas.

A tabela 5.11 sintetiza os resultados obtidos, em termos do número de falsas suspeitas, para os ambientes com 1, 2 e 5 sistemas de controle. Na maioria das simulações, a abordagem baseada em *RNA* cometeu um menor número de falsas suspeitas, reforçando sua confiabilidade quando comparado as demais abordagens.

Tabela 5.11. Número de falsas suspeitas observadas na execução dos algoritmos de detecção para uma rede *Shared-Bus-Ethernet* com taxa de transferência de $10Mbps$ compartilhada por múltiplos sistemas.

Δ^i (ms)	Detector	Sistema 1	Sistema 2	Sistema 5
0.1	**Bertier**	62	46	38
	Jacobson	16	31	25
	RNA	16	7	7
0.5	**Bertier**	64	66	46
	Jacobson	80	64	51
	RNA	40	59	26
0.9	**Bertier**	32	57	52
	Jacobson	40	62	63
	RNA	32	50	49
1.0	**Bertier**	64	63	49
	Jacobson	80	60	55
	RNA	0	3	1
2.0	**Bertier**	0	49	62
	Jacobson	0	62	53
	RNA	0	4	8
5.0	**Bertier**	0	52	58
	Jacobson	0	62	64
	RNA	0	0	1

Assim como nas demais simulações, os detectores construídos com as abordagens de adaptação de Bertier e de Jacobson oscilam em termos de desempenho; todavia, a abordagem de Bertier obteve um desempenho melhor na maioria dos casos. No cenário de pior caso ($\Delta^i = 0.1ms$), o algoritmo de Jacobson, mais uma vez, apresentou um

desempenho melhor que o algoritmo de Bertier.

Diferente do comportamento apresentado na *Switched-Ethernet*, nos cenários com maior estresse da rede o número de falsas suspeitas diminuem. Esses cenários são aqueles em que a rede é compartilhada por 5 sistemas de controle e os períodos de emissão de *heartbeats* estão entre 0.1 e 0.9 milissegundos. Isso confirma a adequação das abordagens adaptativas aos cenários de tráfego severo.

O desempenho dos detectores em termos do intervalo de tempo necessário para corrigir uma falsa suspeita é apresentado na tabela 5.12. Os valores dessa tabela também estão em milissegundos e com precisão de duas casas decimais. As células da tabela preenchidas com um sinal de menos (−) representam a impossibilidade do cálculo do T_M por conta da inexistência de falsas suspeitas.

A abordagem baseada em *RNA*, na grande maioria dos casos, não só comente um menor número de falsas suspeitas, mas também corrige os erros de detecção mais rapidamente que as demais abordagens de detecção. As abordagens de Bertier e de Jacobson alternam-se em termos de desempenho, quando comparados em função dessa métrica. Entretanto, nos cenários de tráfego mais severo o algoritmo de Jacobson obteve os melhores resultados.

Na tabela 5.13 verifica-se o desempenho dos detectores em termos do intervalo entre falsas suspeitas (T_{MR}). Os valores apresentados nessa tabela estão em milissegundos e com precisão de uma casa decimal. Novamente, as células da tabela preenchidas com um sinal de menos (−) representam a inviabilidade do cálculo do T_{MR} por conta da inexistência de falsas suspeitas.

Como nas demais avaliações, a abordagem baseada em RNA consegue um melhor desempenho em termos da métrica T_{MR} na maioria dos casos.

Na tabela 5.14 é apresentado o desempenho dos detectores em função do tempo de detecção. De acordo com essa métrica, a abordagem de Jacobson, mais uma vez, possui um melhor desempenho que as demais. A adaptação usando RNA obteve um melhor

Tabela 5.12. Tempo de recuperação (T_M) observado na execução dos algoritmos de detecção para uma rede *Shared-Bus-Ethernet* com taxa de transferência de $10Mbps$ compartilhada por múltiplos sistemas. T_M^U, $\overline{T_M}$ e T_M^{std} referem-se, respectivamente, ao T_M máximo, médio e ao seu desvio padrão.

Δ^i (ms)	Detector	Sistema 1			Sistema 2			Sistema 5		
		T_M^U	$\overline{T_M}$	T_M^{std}	T_M^U	$\overline{T_M}$	T_M^{std}	T_M^U	$\overline{T_M}$	T_M^{std}
0.1	**Bertier**	0.47	0.10	0.15	99.28	6.30	18.71	6.10	0.62	1.31
	Jacobson	0.43	0.23	0.20	99.37	3.86	18.01	4.08	0.46	1.02
	RNA	0.04	0.03	0.01	99.09	16.84	36.85	4.54	1.57	1.87
0.5	**Bertier**	0.06	0.04	0.02	0.86	0.15	0.16	133.96	13.24	28.82
	Jacobson	0.04	0.03	0.01	0.76	0.14	0.16	133.59	5.66	22.26
	RNA	0.04	0.04	0.00	0.75	0.16	0.16	133.06	10.86	30.39
0.9	**Bertier**	0.39	0.24	0.11	1.05	0.21	0.20	4.19	0.77	0.96
	Jacobson	0.41	0.25	0.11	1.02	0.20	0.19	3.82	0.61	0.89
	RNA	0.39	0.24	0.11	0.79	0.17	0.19	3.75	0.55	0.83
1	**Bertier**	0.06	0.06	0.01	1.72	0.17	0.24	4.39	0.70	0.87
	Jacobson	0.07	0.07	0.00	1.25	0.17	0.21	4.44	0.60	0.83
	RNA	-	-	-	1.49	0.58	0.79	0.06	0.06	0.00
2	**Bertier**	-	-	-	1.84	0.20	0.28	1.26	0.32	0.30
	Jacobson	-	-	-	1.83	0.19	0.27	1.26	0.33	0.28
	RNA	-	-	-	1.26	1.05	0.16	1.33	0.55	0.42
5	**Bertier**	-	-	-	1.73	0.17	0.24	2.65	0.50	0.56
	Jacobson	-	-	-	1.59	0.14	0.21	3.32	0.51	0.63
	RNA	-	-	-	-	-	-	1.54	1.54	0.00

desempenho em termos do tempo de detecção médio, quando realiza-se uma comparação da mesma com a abordagem de Bertier.

Como pode ser visto na tabela, a abordagem baseada em RNA obteve, em alguns poucos casos, tempos de detecção máximos elevados. Esses tempos de detecção podem ser justificados por um treinamento ineficiente ou ainda por variações excessivas dos atrasos no canal de comunicação.

Tabela 5.13. Intervalo entre falsas suspeitas (T_{MR}) observado na execução dos algoritmos de detecção para uma rede *Shared-Bus-Ethernet* com taxa de transferência de $10Mbps$ compartilhada por múltiplos sistemas. T^{U}_{MR}, $\overline{T_{MR}}$ e T^{std}_{MR} referem-se, respectivamente, ao T_{MR} máximo, médio e ao seu desvio padrão.

Δ^i (ms)	Detector	Sistema 1			Sistema 2			Sistema 5		
		T^{U}_{MR}	$\overline{T_{MR}}$	T^{std}_{MR}	T^{U}_{MR}	$\overline{T_{MR}}$	T^{std}_{MR}	T^{U}_{MR}	$\overline{T_{MR}}$	T^{std}_{MR}
0.1	**Bertier**	7.3	1.0	2.1	99.9	3.7	15.2	9.6	1.5	2.6
	Jacobson	9.5	4.7	4.6	100.0	5.6	18.2	6.4	2.3	2.0
	RNA	8.8	4.7	4.0	109.0	24.4	42.2	10.6	8.3	2.4
0.5	**Bertier**	9.5	4.9	4.5	24.3	4.9	4.4	135.6	10.8	24.5
	Jacobson	9.5	4.9	4.5	24.3	6.3	4.9	135.6	9.8	22.8
	RNA	10.0	10.0	0.0	32.0	6.5	6.0	163.6	19.6	37.0
0.9	**Bertier**	50.0	17.7	16.1	35.6	10.1	6.8	41.0	11.7	11.5
	Jacobson	50.0	17.2	15.6	35.6	11.6	7.6	38.6	11.3	7.4
	RNA	60.0	21.3	21.3	46.1	14.0	10.4	101.8	13.4	20.7
1	**Bertier**	10.0	10.0	0.0	35.1	10.2	7.9	41.3	13.2	10.6
	Jacobson	10.0	10.0	0.0	39.3	13.4	8.4	40.2	14.6	9.2
	RNA	-	-	-	7.3	3.7	5.1	-	-	-
2	**Bertier**	-	-	-	79.8	25.0	18.8	56.0	20.7	14.4
	Jacobson	-	-	-	93.8	26.0	18.4	70.7	24.8	16.9
	RNA	-	-	-	634.2	220.7	358.1	214.6	105.5	88.2
5	**Bertier**	-	-	-	158.74	60.9	46.8	208.7	53.6	41.5
	Jacobson	-	-	-	229.9	64.26	49.6	228.7	62.9	45.1
	RNA	-	-	-	-	-	-	-	-	-

5.2.8.2 Avaliando o Impacto da Implementação dos Detectores Sobre o Desempenho do Controle.

Nesta avaliação foram considerados cenários nos quais a rede *Shared-Bus-Ethernet* era compartilhada por 1, 2 e 5 sistemas de controle.

Os gráficos das figuras 5.14 e 5.15 apresentam o desempenho da rede em termos dos índices *IAE* e *ISE*, respectivamente. Nesses gráficos, os índices de desempenho são apresentados em escala logarítmica.

Observa-se que o desempenho do sistema de controle aumenta na medida em que o

Tabela 5.14. Tempo de detecção (T_D) observado na execução dos algoritmos de detecção para uma rede *Shared-Bus-Ethernet* com taxa de transferência de $10Mbps$ compartilhada por múltiplos sistemas. T_D^U, $\overline{T_D}$ e T_D^{std} referem-se, respectivamente, ao T_D máximo, médio e ao seu desvio padrão.

Δ^i (ms)	Detector	Sistema 1			Sistema 2			Sistema 5		
		T_D^U	$\overline{T_D}$	T_D^{std}	T_D^U	$\overline{T_D}$	T_D^{std}	T_D^U	$\overline{T_D}$	T_D^{std}
0.1	**Bertier**	0.8	0.2	0.2	48.0	8.0	4.2	4.3	0.4	0.7
	Jacobson	0.3	0.1	0.0	30.0	0.8	3.3	2.6	0.1	0.3
	RNA	1.0	0.1	0.1	410.7	1.3	19.4	0.5	0.2	0.1
0.5	**Bertier**	3.6	0.9	0.9	3.7	1.1	0.8	64.9	8.6	13.2
	Jacobson	0.5	0.5	0.0	1.0	0.8	0.1	41.2	2.3	5.2
	RNA	0.6	0.5	0.2	1.8	1.8	0.2	67.6	2.0	9.6
0.9	**Bertier**	6.5	1.7	1.6	6.5	1.8	1.5	7.1	2.2	1.5
	Jacobson	1.2	1.0	0.1	1.5	1.2	0.1	3.0	1.6	0.3
	RNA	1.5	1.0	0.7	4.3	1.3	0.3	3.8	2.1	1.0
1	**Bertier**	7.2	1.8	1.8	7.2	1.9	1.7	7.3	2.3	1.7
	Jacobson	1.1	1.0	0.0	1.9	1.3	0.1	3.1	1.7	0.3
	RNA	1.2	1.1	0.0	6.8	2.1	0.4	3.5	1.0	1.1
2	**Bertier**	14.5	3.6	3.6	14.5	3.7	3.5	14.5	3.9	3.4
	Jacobson	2.0	2.0	0.0	3.2	2.3	0.1	3.4	3.4	2.6
	RNA	2.1	2.1	0.0	9.1	4.2	0.6	11.4	4.1	1.0
5	**Bertier**	36.2	8.9	8.9	36.2	9.1	8.9	36.2	9.3	8.8
	Jacobson	5.0	5.0	0.0	5.9	5.2	0.1	7.4	5.7	0.3
	RNA	5.4	5.4	0.0	14.5	10.5	0.5	16.2	10.0	1.8

período de emissão de *heartbeats* é incrementado. Além disso, observa-se que, quanto maior o número de dispositivos, maior o índice de erro do sistema de controle.

Por outro lado, o caráter não determinístico da rede *Shared-Bus-Ethernet* pode, em alguns casos, fazer com que em cenários com menor carga o controle apresente um desempenho menor que em cenários de tráfego mais severo. É o que mostram os gráficos das figuras 5.14 e 5.15, quando são comparados os índices *IAE* e *ISE* dos cenários com 2 e 5 sistemas de controle para um período de emissão de *heartbeats* $\Delta^i = 0.1ms$.

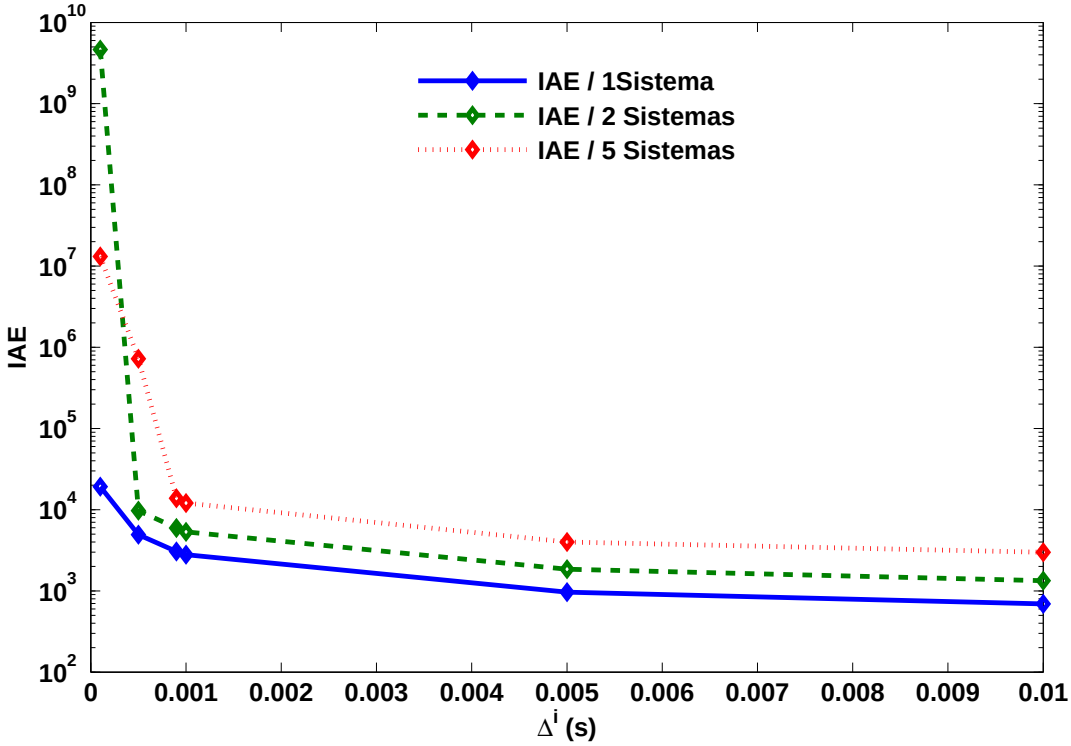


Figura 5.14. Comportamento do índice de erro *IAE* em função do período de emissão Δ^i nas simulações com rede *Shared-Bus-Ethernet*

O gráfico da figura 5.16 apresenta o jitter imposto, no pior caso, a um sistema de controle, quando a rede é compartilhada por 1, 2 e 5 sistemas, respectivamente. Nesse gráfico, tanto o *jitter* quanto Δ^i são apresentados em escala logarítmica.

Para um cenário com apenas 1 sistema de controle, para qualquer período de emissão de *heartbeats* igual ou superior a $0.1ms$, o *jitter* final do sistema se mantém constante e equivalente a $1.1ms$, usando o conceito de *margem de jitter* e considerando um atraso mínimo de $112.4\mu s$, equivalente ao somatório dos tempos mínimos de execução do processo de sensoriamento (configurado na simulação como $5\mu s$), de execução da tarefa de controle (configurado como $5\mu s$), de transferência da mensagem entre sensor-controlador ($51.2\mu s$) e entre controlador-atuador ($51.2\mu s$). Assim, $J_m(112.4\mu s) = 9.7ms > 1.1ms$, ou seja, no cenário com a rede *Shared-Bus-Ethernet* sendo usada por apenas 1 sistema, o controle se mantém estável para todos os períodos de emissão de *heartbeats* utilizados.

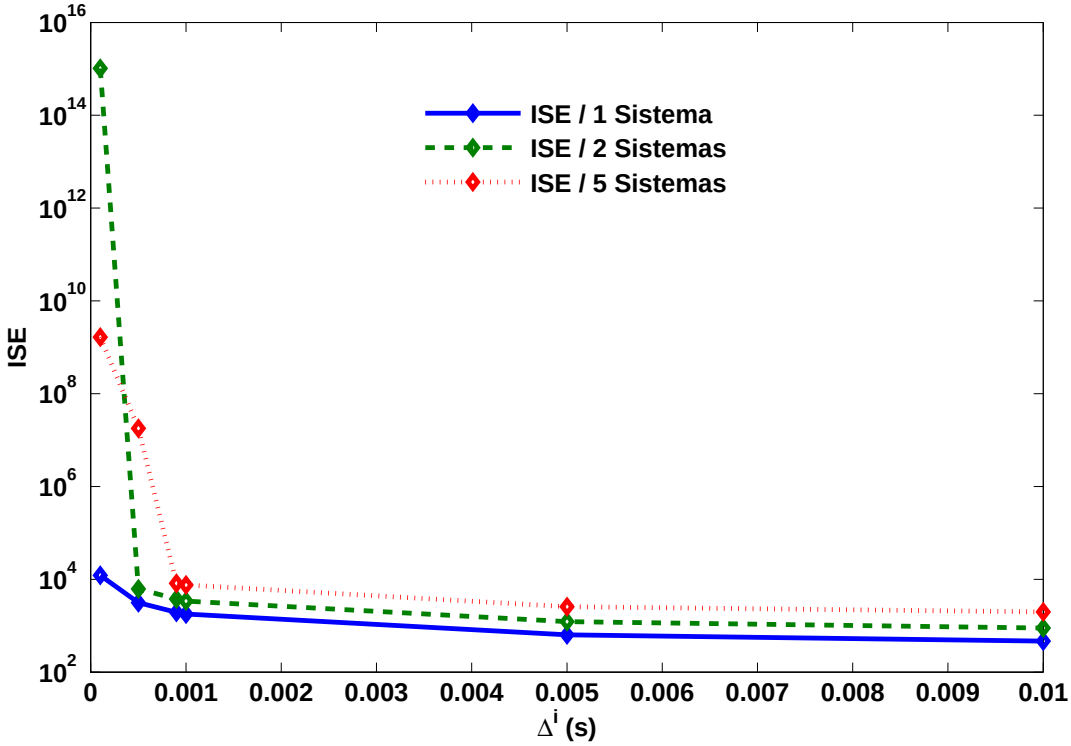


Figura 5.15. Comportamento do índice de erro *ISE* em função do período de emissão Δ^i nas simulações com rede *Shared-Bus-Ethernet*

Para os cenários com 2 e 5 sistemas, a estabilidade do controle só é garantida para períodos de emissão de *heartbeats* superiores a $0.5ms$ e $0.9ms$, respectivamente.

O gráfico na figura 5.17 apresenta a qualidade do desempenho dos sistemas de controle em cenários em que a rede é compartilhada com 1, 2 e 5 sistemas de controle. Como pode ser observado, o índice de desempenho do sistema aumenta à medida em que o período de emissão de *heartbeats* aumenta ou o número de sistemas de controle compartilhando a rede diminuem.

5.3 CONSIDERAÇÕES FINAIS

Apesar da rede *CAN* ser um padrão de fato para construção de *NCS*, a rede *Switched-Ethernet* apresentou um melhor desempenho em todas as análises. Além de possibilitar

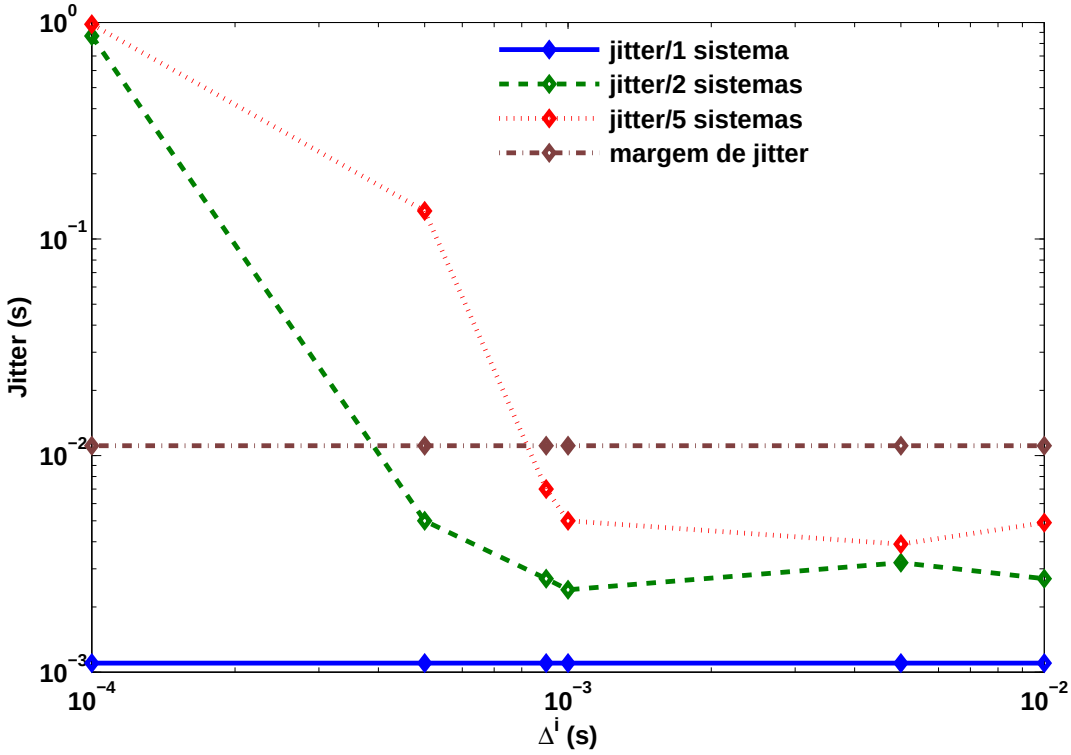


Figura 5.16. Jitter observado nas simulações com rede *Shared-Bus-Ethernet*

que a implementação dos mecanismos de detecção de defeitos tenham um menor impacto sobre *NCS*, essa rede permitiu que os detectores tivessem desempenhos melhores. Entretanto, o fato de não possuir, de forma natural, um mecanismo para priorização de mensagens, a rede *Switched-Ethernet* pode representar um problema para sistemas que precisam de uma distinção entre a urgência na entrega das mensagens e o mesmo vale para a *Shared-Bus-Ethernet*.

Em geral, o ambiente industrial é hostil às redes de comunicação *Ethernet*, dado que são inúmeros os possíveis elementos nocivos: fontes de interferência elétrica e eletromagnética, contato com óleo e outros fluidos, radiação ultravioleta, vibração, temperaturas extremas, entre outras (MACKAY, 2004). A proposta de cabeamento e equipamentos mais comumente encontrados no mercado para a *Switched-Ethernet* são muito susceptíveis a tais elementos nocivos. As redes *Shared-Ethernet*, tradicionalmente usadas

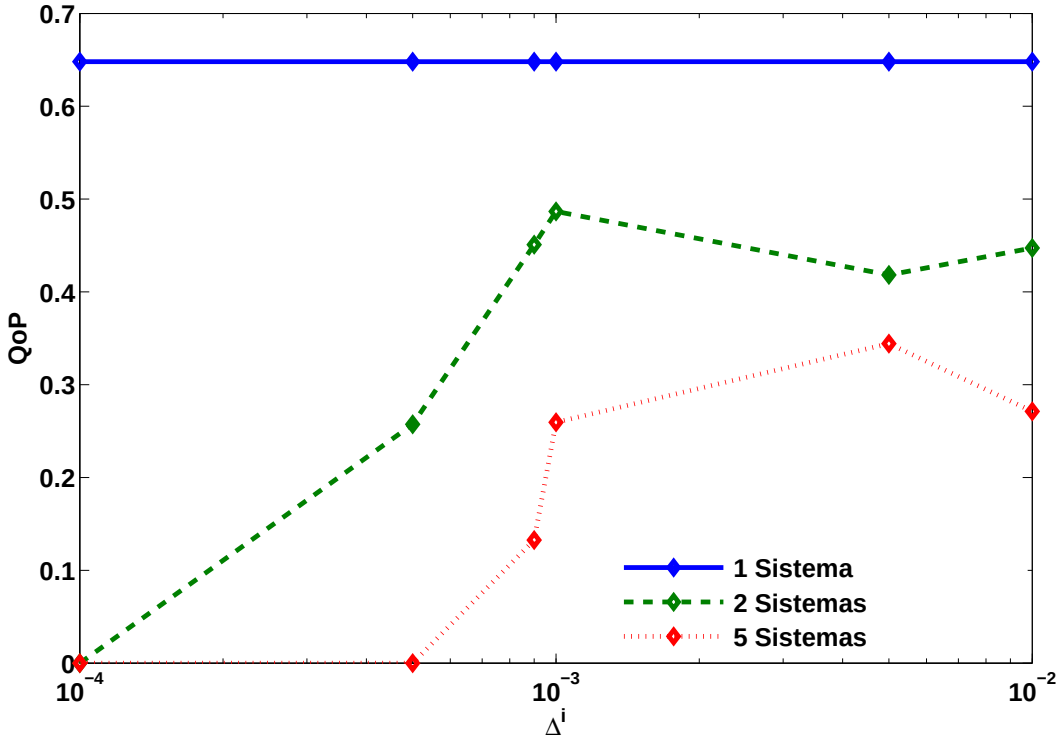


Figura 5.17. Qualidade do desempenho do controle medido ($QoP = \frac{\overline{\varphi}}{\varphi}$) nas simulações com rede *Shared-Bus-Ethernet*

em escritórios, possuem o mesmo problema, todavia, para essa rede, existem alternativas naturais previamente desenvolvidas. Exemplos dessas alternativas são os padrões $10Base2$ e $10Base5$(MACKAY, 2004), entretanto, tais padrões possuem uma série de limitações, mas são mais resistentes a interferências.

Uma outra análise importante é a relação entre os tempos de detecção medidos para cada detector e o tempo necessário para a atuação dos demais mecanismos de tolerância a falhas. Na implementação dos sistemas de controle de missão crítica, quando a falha é detectada, é necessário realizar a reconfiguração ou recuperação do sistema. O intervalo de tempo para realizar esses procedimentos deve ser tal que não interfira no desempenho do sistema de controle. De modo simplificado, considerando o tempo total para atuação dos mecanismos de tolerância a falhas (T_{TF}) como sendo o somatório do tempo necessário para que a falha seja percebida ou detectada (T_D) e mais o tempo necessário para que a

comutação entre os controladores aconteça (T_{RE}), tem-se:

$$T_{TF} = T_D + T_{RE}$$

Uma vez que a amostra foi coletada no dispositivo sensor, *a margem de jitter* será o tempo máximo para atuação do sistema de controle sobre a planta. Sendo assim, T_{TF} é limitado pela *margem de jitter* do sistema. Logo:

$$T_{TF} \leq J_m(L)$$

ou,

$$T_D \leq J_m(L) - T_{RE} \tag{5.14}$$

com $T_D \geq T_D^{min}$, $J_m(L) \geq 0$ e $T_{RE} \geq 0$.

A equação 5.14 apresenta uma condição suficiente para que os mecanismos de tolerância a falhas atendam às restrições temporais impostas pelo sistema de controle. Sendo assim, analisando os dados das tabelas referentes ao tempo de detecção observado para os detectores de defeitos, pode-se observar que, em alguns casos, apesar da implementação do detector não impactar no desempenho do controle, o tempo de detecção é extremamente alto e que, após a ocorrência de uma falha, a estabilidade do controle não será garantida nesses casos (haja vista que $T_D > J_m(L)$ e, conseqüentemente, $T_D > J_m(L) - T_{RE}$).

Esse fato é verificado para as simulações dos sistemas sobre redes *Shared-Bus-Ethernet* e *CAN*. É válido observar que, dentre as abordagens de detecção, a abordagem baseada em *RNA* e a abordagem de *Bertier* são as que apresentam os piores desempenhos, quando se observa a relação entre T_D e T_{TF}.

Analisando o tempo de resposta da tarefa de detecção, pode-se concluir que (com

base nos dados da tabela 5.2), o período mínimo de emissão de *heartbeats* é $0.51ms$. Entretanto, as avaliações consideradas na seção anterior apresentam períodos de emissão de *heartbeats* iguais a 0.1 e $0.5ms$. Logo, para tais períodos de emissão, o conjunto de tarefas que executam no controlador primário não é escalonável. Isso, contudo, não influencia a estabilidade do sistema de controle. Isso se dá pelo fato de que, mesmo após uma falha, o tempo máximo para atuação do sistema não é ultrapassado. Por outro lado, períodos de emissão de *heartbeats* com tais magnitudes apenas reforçam o poder de adaptação das abordagens de detecção de defeitos.

Apesar disso, em termos do tempo de detecção, observa-se que na maioria dos casos os mecanismos adaptativos estimam intervalos entre chegadas com magnitudes muito inferiores à magnitude máxima estimada, ressaltando, assim, que a adaptabilidade do detector, em geral, reduz o tempo necessário para a detecção de uma falha no sistema.

CAPÍTULO 6

CONCLUSÕES E TRABALHOS FUTUROS

6.1 CONCLUSÕES ACERCA DO TRABALHO DESENVOLVIDO

Projetar sistemas de controle de missão-crítica sobre rede é uma atividade interdisciplinar que envolve, entre outras coisas, conhecimento do projeto do sistema de controle, mecanismos de tolerância a falhas e redes de computadores. Tais projetos são extremamente difíceis, uma vez que existe uma relação de compromisso entre o desempenho desejado para o sistema de controle e a implementação dos mecanismos de tolerância a falhas.

Essa relação de compromisso é evidente quando se observa a implementação de detectores de defeitos. Para um bom desempenho do sistema de controle sobre rede é necessário que a influência dos atrasos de comunicação e computação seja o menor possível, de modo que tal sistema tenha um desempenho que se assemelhe aos dos sistemas de controle tradicionais. Todavia, para uma rápida detecção de falhas, é necessário que o detector de defeitos possua períodos de emissão de mensagens de monitoramento bastante pequenos, o que além de aumentar a carga computacional, implica em um aumento do consumo dos recursos da rede e em conseqüentes aumentos nos atrasos de comunicação e computação. Esse aumento em tais atrasos implica em uma diminuição do desempenho do sistema de controle, podendo, em alguns casos, levar o sistema a um estado de instabilidade.

Dentro desse contexto, o conceito de *margem de jitter* se mostrou uma ferramenta bastante eficiente para adequar os requisitos de tolerância a falhas ao requisito de estabilidade do sistema de controle.

A implementação de mecanismos adaptativos de detecção de defeitos ajuda a incrementar a confiabilidade do sistema. Se por um lado, a detecção pode se tornar mais rápida e confiável, por outro se pode acomodar de modo mais eficiente os procedimentos de detecção e restabelecimento do sistema, uma vez que evita tempos de detecção superestimados.

Nesse cenário, a construção de detectores de defeitos baseados em Redes Neurais Artificiais se mostra uma alternativa muito atraente quando se deseja aumentar a confiabilidade do detector nos cenários em que a magnitude do atraso é desconhecida ou quando o atraso varia de forma não determinística.

Todavia, apesar da confiabilidade obtida com a utilização dessa ferramenta da inteligência artificial, o processo de treinamento e obtenção da *RNA* pode ser bastante demorado e tedioso.

6.2 PROPOSTA PARA TRABALHOS FUTUROS

Esta dissertação contribuiu na avaliação da implementação de mecanismos de detecção de defeitos no âmbito dos sistemas de controle sobre rede. Para isso foi proposta uma abordagem adaptativa baseada em Redes Neurais Artificiais e avaliado o desempenho de tal abordagem através da comparação com outras propostas para a detecção adaptativa existentes na literatura. Além disso, neste trabalho é inserido o conceito de *margem de jitter* como uma ferramenta para a avaliação do impacto da construção de mecanismos de tolerância a falhas na construção de sistemas de controle de *missão-crítica*.

Apesar das contribuições aqui apresentadas, alguns estudos e melhorias podem ser realizados de forma a trazer novas contribuições. Sendo assim, algumas propostas de trabalhos futuros são discutidas a seguir.

6.2.1 Possíveis Melhorias na Abordagem de Detecção Baseada em RNA

6.2.1.1 Verificar a Utilização de Outras Variáveis de Modo a Incrementar a Capacidade de Adaptação da Abordagem Baseada em Redes Neurais Artificiais Um ponto para trabalho futuro é analisar o melhor conjunto de variáveis utilizado na abordagem de adaptação baseada em *RNA*, de modo a fazer com que esta consiga um melhor desempenho em termos das métricas de detecção adotadas. Para tanto, é necessário escolher variáveis de acordo com o tipo de rede de comunicação e do tipo de escalonamento utilizado nos dispositivos. Por exemplo, em uma rede *CAN* pode ser interessante que a prioridade dos dispositivos ou a velocidade da rede sejam usados como parâmetros de entrada no processo de ajuste dos tempos estimados para chegadas das mensagens de *heartbeats*. Para rede *Ethernet*, por outro lado, verificar o número de colisões, probabilidade de ocorrência de uma nova colisão, a probabilidade de descarte de uma mensagem e o número de dispositivos pode ser uma alternativa mais interessante.

6.2.1.2 Avaliar Técnicas que Possam Ajustar no Processo de Seleção da Rede Neural para o Caso da Detecção Adaptativa O processo de seleção da arquitetura da rede neural é bastante tedioso e demorado, implicando em várias tentativas no intuito de obter uma *RNA* com desempenho satisfatório.

Técnicas como algoritmos genéticos (BITTENCOURT, 2001) têm sido usadas com o intuito de oferecer uma alternativa para a obtenção de modelos de redes neurais que melhor se adequem a determinados domínios de problema (SANTOS et al., 1999; FREITAS, 2004). A utilização de tais técnicas pode automatizar e tornar menos dispendioso o processo de seleção da *RNA*.

6.2.2 Pesquisar o Potencial de Outros Modelos de Redes Neurais na Construção de um Detector Adaptativo

Nesta dissertação, optou-se pela utilização das redes *MLP*, por conta da sua vasta utilização e de referências na literatura da utilização de tais redes como aproximador universal(PINKUS, 1999; TIKK; KÓCZY; GEDEON, 2003). Todavia, a eficiência de outros modelos de RNA na construção de detectores adaptativos não foi avaliada. É interessante, portanto, validar o desempenho de outros modelos de RNA e compará-los ao desempenho do modelo de Rede Neural utilizado nesta dissertação.

6.2.3 Novas Abordagens de Adaptação Usando Técnicas de Inteligência Artificial

6.2.3.1 Abordagem Baseada em Modelo Fuzzy ou *Neuro-Fuzzy* Durante o desenvolvimento desta dissertação foram conduzidos alguns trabalhos de orientação de conclusão de curso de graduação no sentido de avaliar a construção de detectores de defeitos baseados em lógica *fuzzy* (JANG; SUN; MIZUTANI, 1997).

Esses trabalhos demonstraram que um preditor baseado em tal abordagem pode ser uma alternativa bastante interessante. A construção de modelos *Fuzzy* que se adaptem ao padrão de tráfego pode ser uma abordagem bastante viável para contornar algumas limitações do modelo baseado em *RNA*. Por exemplo, o modelo de *RNA* utilizado nesta dissertação exige um novo treinamento se o padrão do tráfego da rede muda drasticamente; isso pode ser impeditivo para a utilização da abordagem baseada em *RNA* em alguns cenários.

Uma outra alternativa que pode ser avaliada é a utilização de um sistema *fuzzy* na condução do ajuste paramétrico da *RNA*. Tal ajuste pode possibilitar que a Rede Neural se adeque a condições sazonais da rede de comunicação.

6.2.4 Proposta para Modelagem de Sistemas Críticos de Controle

Usando o conceito de margem de *jitter* e a análise do tempo máximo de viagem da mensagem é possível, através do procedimento proposto por Chen, Toueg e Aguilera (2002) determinar, a priori, se a qualidade de serviço do detector de defeitos poderá atender ao sistema de controle.

Por exemplo, Tindell, Hanssmon e Wellings (1994) e Tindell, Burns e Wellings (1995), apresentam uma proposta para o cálculo do atraso de comunicação em redes CAN. Em Schneider, Pardo-Castollote e Hamilton (1999), por sua vez, pode ser encontrada uma análise probabilística do atraso de comunicação em uma rede *Shared-Bus-Ethernet*. Por fim, Song, Koubaa e Simonot (2002) analisam o atraso inserido por uma rede *Switch-Bus-Ethernet*.

Tais metodologias, aliadas ao conceito de *margem de jitter*, proposto por Cervin et al. (2004), e considerando a estratégia introduzida e discutida nesta dissertação (a exemplo das análises e considerações realizadas nas seções 5.2 e 5.3), podem colaborar para um modelagem simples e mais elaborada para o projeto de sistemas de controle críticos.

6.2.5 Desenvolvimento de uma Abordagem Adaptativa Baseada em Componentes

Como discutido anteriormente, as modernas aplicações de controle e supervisão demandam soluções que permitam interconexões entre dispositivos de diferentes fabricantes, facilidade de monitoramento, flexibilidade para distribuição, possibilidade de implementação de mecanismos de tolerância a falhas etc. Para atender a tais demandas, necessita-se de soluções de software que facilitem interoperabilidade, permitam composições adaptáveis às mais diversas configurações dos dispositivos e facilitem a implementação de mecanismos básicos focados no desempenho e na confiabilidade.

A engenharia de software baseada em componentes tem sido vista como uma alter-

nativa bastante promissora no desenvolvimento das modernas aplicações industriais. A engenharia de componentes de software pode ser uma alternativa bastante convidativa para a implementação de detectores de defeitos adaptáveis e sintonizáveis às necessidades das aplicações industriais. Pode, além disso, permitir que diferentes abordagens de detecção sejam utilizadas com intuito de possibilitar uma melhor aderência do serviço de detecção de defeitos às necessidades da aplicação.

Atualmente, está sendo desenvolvido no LaSiD, uma arquitetura para suporte ao desenvolvimento de aplicações industriais de controle e supervisão denominada ARCOS (ANDRADE; MACêDO, 2005). Nessa plataforma, além dos mecanismos de aquisição de dados, estão sendo implementados mecanismos de reconfiguração e recuperação pró-ativos. Em paralelo a essas implementações, está sendo desenvolvido um detector de defeitos multicamada baseado em componentes. Tal detector deve permitir a sintonia da qualidade de serviço de detecção e prover as informações necessárias para que os demais mecanismos de tolerância sejam ativados.

APÊNDICE A

DESCREVENDO DETALHES DO AMBIENTE DE SIMULAÇÃO

Este capítulo descreve a ferramenta *Simulink/Matlab* e o *Toolbox TrueTime*. Além disso, apresenta os algoritmos implementados e as questões de projeto adotadas para a execução das simulações.

A.1 O AMBIENTE MATLAB

Matlab (The Mathworks, 2002) é um acrónimo para ***Mat****rix* ***Lab****oratory* e foi originalmente proposto como uma linguagem a ser utilizada em problemas que envolvessem manipulação de vetores e matrizes. Atualmente, o *Matlab* evoluiu para um ambiente interativo que permite solucionar problemas técnicos de computação, especialmente aqueles que envolvem formulações compostas por matrizes e vetores. O ambiente *Matlab* possibilita integrar computação, visualização e programação na resolução dos problemas. Computação e matemática, desenvolvimento de algoritmos, aquisição de dados, modelagem, simulação e prototipagem, entre outros, são casos típicos nos quais o *Matlab* pode ser utilizado.

O ambiente *Matlab* é composto por cinco partes básicas:

- **Ambiente de desenvolvimento**, com um conjunto de facilidades e ferramentas para ajudar no uso das funções e arquivos do *Matlab*;

- **Biblioteca de funções matemáticas**, a qual contém uma ampla coleção de algoritmos computacionais envolvendo funções elementares (como soma, multiplicação, seno, aritmética complexa etc.) e funções avançadas (como funções para inversão de matrizes, cálculo de auto-valores, transformadas de fourier etc.);

- **Linguagem Matlab**, uma linguagem de alto nível que permite a manipulação de vetores e matrizes, além de conter declarações para fluxo de controle, funções, estruturas de dados, declarações para entrada/saída e características para a programação orientada a objetos;

- **Gráficos**, define um conjunto de facilidades para manipulação e visualização de gráficos 2-D e 3-D;

- **Matlab API**, contém um conjunto de bibliotecas para que programas escritos em linguagens como *C/C++*, *Fortran*, *Java* e *Visual Basic* interajam com o *Matlab*.

O *Matlab* possui uma família de ferramentas (*toolboxes*) compostas por funções do *Matlab* (*M-files*) para solução de problemas em domínios específicos. Os *toolboxes* permitem o aprendizado e a aplicação de tecnologias específicas, incluindo comunicação, sistemas de controle, aquisição de dados, redes neurais, processamento de sinal, sistemas de tempo real etc.

A.1.1 O Simulink

O *Simulink*(The Mathworks, 2006) é um pacote de software, integrado ao *Matlab* que possibilita a modelagem, simulação e análise de sistemas dinâmicos, os quais mudam suas saídas em função do tempo (por exemplo, sistemas elétricos, mecânicos, termodinâmicos etc).

A simulação dos sistemas é feita em dois processos básicos. Primeiramente, o sistema é modelado utilizando o editor de modelos do *Simulink*. O modelo deve refletir as relações

matemáticas entre as entradas, saídas e estados do sistema. Em seguida, o *Simulink* deve ser usado para simular o comportamento do sistema em função do tempo, usando as informações contidas no modelo (The Mathworks, 2004).

No *Simulink*, um diagrama em blocos é usado como uma representação gráfica do modelo matemático de um sistema dinâmico. Esse modelo matemático é, em geral, composto por um conjunto de equações conhecidas como equações algébricas, equações diferenciais ou sistemas de equações diferenciais.

Os modelos de blocos são representados por um conjunto de blocos interconectados por linhas, através das quais trafegam os sinais emitidos por cada bloco. No *Simulink*, um bloco pode pertencer a duas classes: virtuais e não virtuais. Blocos não virtuais representam subsistemas elementares, como integradores, somadores, multiplexadores etc. Blocos virtuais, por sua vez, representam uma composição de blocos e sinais, visualizados no modelo como um único bloco. Dessa forma, blocos não virtuais podem ser agrupados para representar subsistemas maiores (blocos virtuais), os quais podem novamente ser agrupados, em diagramas com blocos virtuais e/ou não virtuais, para compor um subsistema ainda maior, e assim por diante.

Em geral, para serem diferenciados das outras formas de diagramas de blocos existentes no *Simulink*, diagramas de blocos usados para modelar sistemas dinâmicos são denotados por diagramas de blocos baseados em tempo (*time-based block diagrams*). Tais blocos realizam suas operações da seguinte forma (The Mathworks, 2004):

- O diagrama de blocos do *Simulink* define o relacionamento, no tempo, entre os sinais (variáveis de entrada e saída) e as variáveis de estado do sistema;
- Os sinais representam quantidades que mudam no tempo e que são definidas em todos os instantes entre o início e o término da simulação;
- E os relacionamentos entre os sinais e as variáveis de estado são definidos por um conjunto de equações representadas por blocos.

Em um modelo no *Simulink*, dois tipos de estados podem ocorrer: estados discretos e estados contínuos. Esses estados são usados para modelar o comportamento de sistemas discretos e contínuos, respectivamente. Um estado contínuo muda continuamente em função do tempo; um estado discreto, por sua vez, é uma aproximação de um estado contínuo e muda em intervalos periódicos ou aperiódicos de tempo. Um estado discreto com intervalo de atualização igual a zero equivale a um estado contínuo.

O estado é uma variável que determina a saída de um bloco e o seu valor corrente é uma função dos valores dos estados anteriores e (ou) das entradas atuais. Para computar o estado atual, um bloco deve memorizar os estados anteriores. Os blocos que não têm estado (ditos livres de estado ou *stateless*) não precisam de memória.

O relacionamento temporal entre entradas, estados e saídas de um bloco é realizado através de um conjunto de funções. Tal conjunto inclui: uma função de saída (f_o), a qual relaciona entradas, saídas e estados do sistema no tempo; uma função de atualização f_u, a qual relaciona os valores futuros de estados discretos do sistema ao tempo, entrada e estados correntes; e uma função derivativa f_d, através da qual as derivadas dos estados contínuos do sistema no tempo são relacionadas aos valores dos estados e entradas correntes. Assim, as funções podem ser expressas por:

$$y = f_o(t, x, u) \tag{1.1}$$

$$x_{d_{k+1}} = f_o(t, x, u) \tag{1.2}$$

$$x'_c = f_d(t, x, u) \tag{1.3}$$

em que,

$$x = \begin{bmatrix} x_c \\ \\ x_{d_k} \end{bmatrix} \tag{1.4}$$

t representa o tempo corrente, x, y, u representam, respectivamente, os estados, as saídas e as entradas do bloco. x_d representa as derivadas discretas do bloco e x'_c representa as derivadas dos estados contínuos dos blocos.

Para que se possa calcular um estado contínuo é necessário que se tenha o conhecimento acerca de sua taxa de mudança (derivada). Desde que a derivada de um estado contínuo muda continuamente com o tempo, calcular o valor atual de um estado contínuo requer a integração de suas derivadas desde o início da simulação(The Mathworks, 2004). Estados de sistemas dinâmicos reais podem matematicamente ser representados por equações diferenciais ordinárias e, em geral, é bastante difícil realizar a integração de tais equações. Por conta disso, a integração dos estados requer o uso de métodos numéricos denominados ODE (*Ordinary Differential Equation solvers*). O uso dos métodos ODE exige que seja atendida uma relação de compromisso entre precisão e carga computacional. O método consiste em estabelecer a granularidade entre duas marcações (passos de tempo) consecutivas do relógio usado na simulação. A precisão dos métodos numéricos de integração depende do tamanho dos intervalos de tempo entre os passos de tempo usados na resolução do modelo(The Mathworks, 2004). Quanto menor o passo em relação ao tempo, mais precisos serão os resultados produzidos na simulação. Alguns métodos de resolução de ODE, denominados métodos de resolução de tempo variável, podem variar o tamanho do passo de tempo, baseado na derivada do estado, para obter o nível de precisão especificada durante a execução da simulação.

Para o cálculo de estados discretos, o *Simulink* usa blocos especiais chamados blocos discretos. Os estados discretos são calculados de forma similar aos estados contínuos; entretanto, cada passo de tempo deve ser largo o suficiente para acomodar as taxas de

amostragem de todos os estados do modelo.

A simulação de sistemas dinâmicos consiste em computar os estados e saída do sistema dentro de um intervalo de tempo, usando as informações contidas no modelo (The Mathworks, 2006). O processo de simulação pode ser dividido em três fases: *Compilação do modelo*, *Ligação* e *Execução*.

A *Compilação do modelo* consiste na transformação do modelo em uma forma executável. Tal fase envolve (The Mathworks, 2006):

- Avaliação das expressões dos blocos do modelo para determinar seus valores;
- Determinação dos atributos dos sinais e checagem da coerência dos sinais conectados às entradas de cada um dos blocos;
- Propagação dos atributos, a fim de verificar possíveis atributos não especificados;
- Realização de possíveis otimizações no diagrama de blocos;
- Substituição de blocos virtuais pelos blocos contidos nos mesmos;
- Determinação de todos os blocos no modelo cujas taxas de amostragem não foram explicitamente especificadas;

Uma vez finalizada a compilação do modelo, o *Simulink* deve iniciar a fase de *Ligação*. Nessa fase, o Simulink aloca a memória necessária para a execução da simulação. Durante a *Ligação* é a alocada e inicializada a memória para estruturas de dados que armazenam as informações de cada bloco. Além disso, deve criar e configurar as listas de execução; para tanto, são usadas as listas geradas durante a fase de *Compilação do modelo*.

Por fim, a *Execução* do modelo de simulação é finalmente realizada. Essa fase consiste em um laço (ou rodada) de simulação no qual os estados e as saídas de cada bloco são sucessivamente calculadas a cada passo de tempo. A fase de *Execução* é subdividida em duas subfases: a *Inicialização* e a *Iteração*.

A *Inicialização* ocorre uma única vez, no início do laço da Execução. Nessa subfase, são definidos os estados e saídas iniciais do sistema a ser simulado.

Durante a *Iteração*, as seguintes ações são executadas (The Mathworks, 2004):

i) São calculadas as saídas do modelo. Nessa ação, um método do *Simulink* (*Simulink Output Method*) passa os argumentos para cada bloco. Esses argumentos são os ponteiros para as estruturas de dados de cada bloco e um ponteiro para a memória principal de trabalho do *Simulink*(*SimBlock*).

ii) São calculados os estados do modelo. Nessa ação, o *Simulink* usa um mecanismo chamado (*Simulink Engine*) para invocar um método de resolução (*solver*); isso irá depender dos tipos de blocos existentes no modelo: blocos contínuos, discretos ou ambos.

Se o modelo possui apenas estados discretos, o *Simulink* invoca um *solver* discreto selecionado pelo usuário e, então, o *solver* calcula o tamanho do passo de tempo necessário para acomodar os tempos de amostragem do modelo. Após essa ação, um método de atualização do sistema (*Update Method System*) é invocado e esse, por sua vez, invoca o método de atualização de cada bloco contido na lista gerada durante a fase de *Ligação*.

Se o modelo contém apenas estados contínuos, o *Simulink* invoca o *solver* especificado para o modelo. Dependendo do *solver*, esse poderá invocar um dos métodos de derivação (*Derivative method*) do modelo uma única vez ou entrar em subciclos do passo de tempo e invocar continuamente os métodos Derivatives e Outputs para calcular as saídas e derivadas do modelo no passo de tempo. Isso é feito para incrementar a precisão do cálculo realizado.

iii) Opcionalmente, o *Simulink* checa discontuidades nos estados contínuos de um bloco. Essas descontinuidades são checadas usando a técnica chamada de detecção de

zero (*zero-crossing detection*). Esse método se baseia no fato de que, em geral, descontinuidades podem ser indicadas por mudanças significativas nos estados de um sistema dinâmico.

Para usar essa técnica, cada bloco deve registrar um conjunto de variáveis, dito conjunto *zero-crossing*, que podem apresentar descontinuidades. Ao final de cada passo de tempo na simulação, o *Simulink* verifica se, no último passo, alguma mudança de sinal aconteceu nos valores de alguma das variáveis do conjunto *zero-crossing*. Se qualquer passagem por zero é detectada, o Simulink, então, interpola valores entre o valor anterior e o valor corrente de cada variável na qual uma mudança de sinal tenha ocorrido.

iv) O intervalo de tempo necessário para o cálculo do próximo passo de tempo é, então, computado.

A.1.2 O ToolBox TrueTime

O *TrueTime* é um *toolbox*, desenvolvido por Henriksson e Cervin (2003), que permite simular sistemas de controle de tempo real. Nesse *toolbox*, é possível simular o comportamento temporal de sistemas operacionais com *kernel* multitarefa de tempo real, contendo tarefas de controle. Além disso, o *TrueTime* permite estudar os efeitos da utilização da *CPU* e da rede de comunicação sobre o desempenho do sistema de controle.

O *Truetime* oferece quatro tipos de blocos(ver figura A.1.2): o bloco kernel (*Kernel block*), bloco rede(*Network block*), bloco de energia (*Battery block*) e bloco para rede sem fio (*Wireless network block*). Todos esses blocos, existentes no *TrueTime*, são gatilhados por eventos (*event-driven*), sejam esses eventos internos ou externos (HENRIKSSON; CERVIN; ARZEN, 2002). Eventos internos são aqueles provocados pelas interrupções do relógio, enquanto eventos externos correspondem às interrupções provocadas por sinais oriundos das portas externas, como conversores A/D e *interfaces* de Rede.

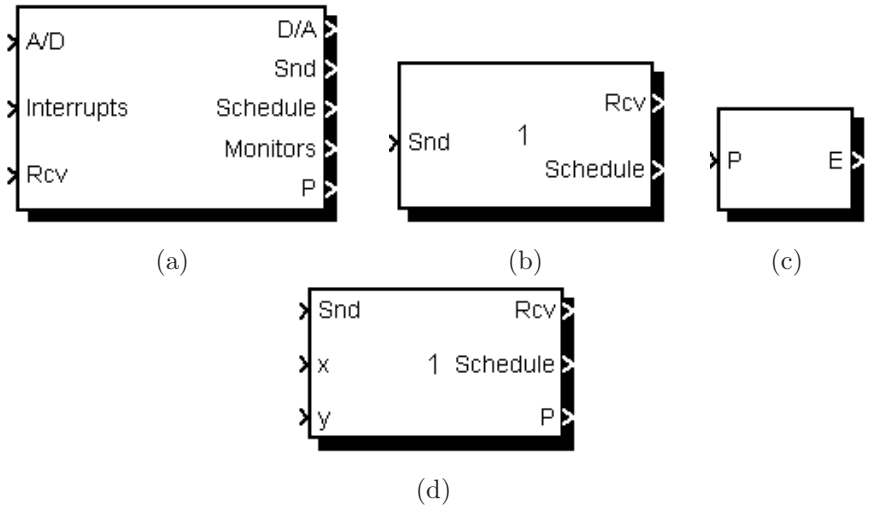


Figura A.1. Blocos disponíveis no TrueTime: (a) Bloco *Kernel*; (b) Bloco *Network*; (c) Bloco *Battery*;(d) Bloco *Wireless Network*.

O bloco *Kernel*, figura A.1(a), simula um computador com sistema operacional multitarefa e *kernel* de tempo real. Esse bloco executa tarefas e manipuladores de interrupção definidos pelo usuário. As tarefas, executadas pelo bloco *Kernel*, podem ser periódicas ou aperiódicas e podem ser usadas para modelar, entre outras coisas, tarefas de um controlador, bem como tarefas de comunicação. Além disso, o bloco mantém diversas estruturas de dados (como filas, registros, monitores etc.) naturalmente encontradas em sistemas operacionais com *kernels* de tempo real.

Dentre as facilidades existentes no bloco *kernel*, a execução das tarefas pode se dar em três diferentes níveis de prioridades: nível de interrupção, nível de kernel e nível de tarefas. O nível de interrupção e o nível de tarefas representam, respectivamente, o mais alto e o mais baixo níveis de prioridade disponíveis. No nível de interrupção são executados os manipuladores de interrupção, os quais são escalonados seguindo uma política baseada em prioridade fixa. No nível de tarefas, por outro lado, o escalonamento pode ser realizado seguindo uma política baseada em prioridades dinâmicas. Dentre as políticas de escalonamento existentes, estão: *RM*(*Rate Monotonic*), *DM* (*Deadline Monotonic*) e EDF (*Earliest Deadline First*).

O bloco *Network*, figura A.1(b), executa toda vez que uma mensagem é enviada ou recebida. Uma fila de transmissão mantém todas as mensagens enviadas em um determinado instante. Essas mensagens são mantidas nessa fila até que a transmissão seja finalizada. Uma mensagem deve conter informações acerca dos nós transmissor e receptor, os dados do usuário, o instante da transmissão e, opcionalmente, atributos de tempo real (tais como *prazo* ou prioridade). O bloco *Network* simula um meio de acesso e transmissão de pacotes em uma rede local e suporta diferentes implementações de protocolos de comunicação; como exemplo podem ser citados: *CSMA/CD*, *CSMA/AMP*, *Round Robin*, *FDMA*, *TDMA* e *Switched Ethernet*.

O bloco *Wireless Network*, figura A.1(d), é uma extensão do bloco *Network* para suportar protocolos usados em redes sem fio, como: *IEEE 802.11b/g* e *802.15.4*. Além disso, esse bloco permite simular características peculiares a esse tipo de rede, como: perda de sinal, interferência entre nós etc.

O bloco *Battery*, figura A.1(c), é usado em conjunto com o bloco *Kernel* para simular dispositivos que possuam restrições de consumo de energia. Para tanto, pode-se configurar nesse bloco a quantidade inicial de energia em *Watts*. Durante a simulação, uma tarefa de controle de energia pode ser criada para verificar ou administrar o consumo de energia (HENRIKSSON; CERVIN, 2003).

A.2 DESCREVENDO A CONFIGURAÇÃO DO AMBIENTE DE SIMULAÇÃO

Conforme descrito na seção 5.2.1, as simulações foram conduzidas utilizando sistemas com quatro dispositivos: dois controladores, um atuador e um sensor. Esses dispositivos compartilham um subsistema de comunicação para realizar troca de mensagens. Os blocos que representam os dispositivos de um subsistema de controle foram agrupados de modo a melhor organizar os elementos no diagrama da simulação.

A figura A.2 apresenta como cada bloco representante de um subsistema de controle

foi esquematizado. Dessa maneira, se tornou mais simples agrupar diversos sistemas de controle compartilhando uma mesma rede de comunicação.

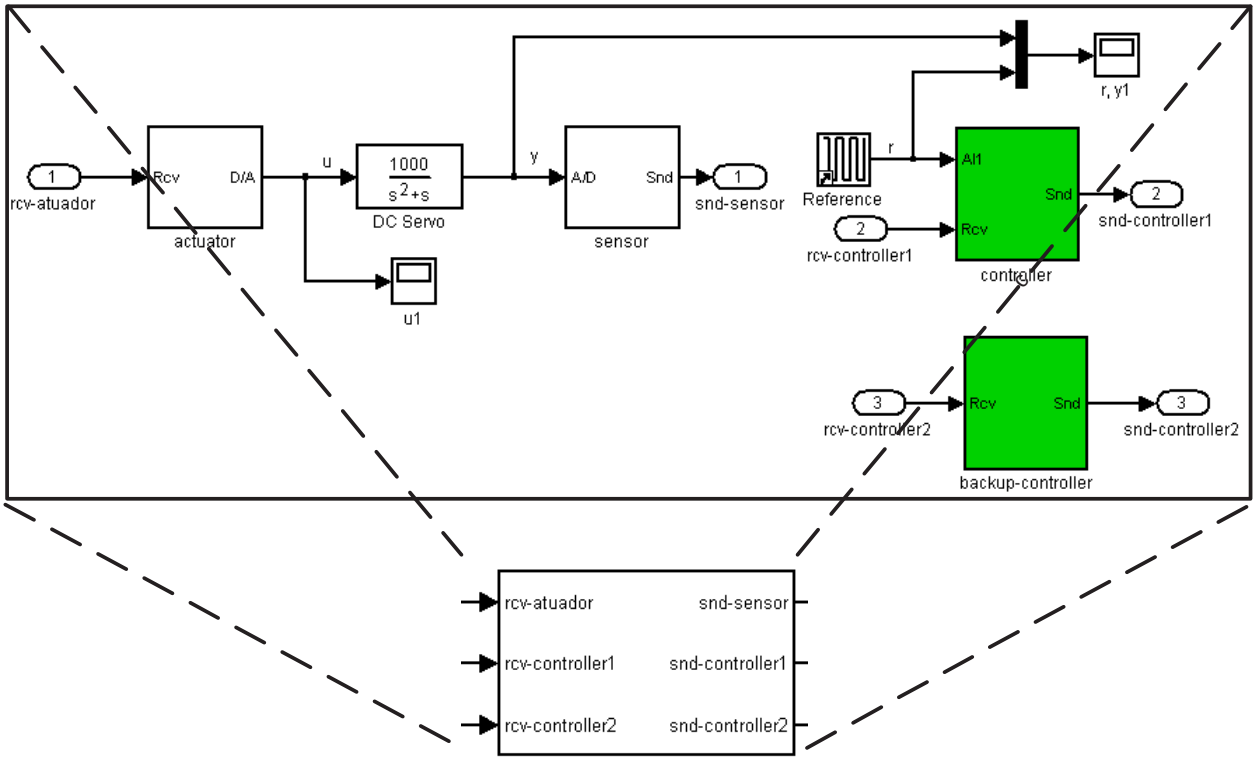


Figura A.2. Bloco virtual representando um subsistema de controle

Os parâmetros de configuração do bloco *Network* foram alterados seguindo o tipo de rede de comunicação utilizada. Os seguintes parâmetros, disponíveis para a configuração de tal bloco (ver figura A.3), foram usados nas simulações:

- *Network type*, através desse parâmetro determina-se o tipo de rede de comunicação a ser utilizada. A seleção do tipo de rede torna disponível ou indisponibiliza alguns parâmetros de configuração do bloco *Network*. Os tipos de redes selecionados foram: *CSCMA/CD* (*Ethernet*), *CSMA/AMP* (*CAN*) e *Switched Ethernet*;

- *Network Number*, identificador do bloco de rede. Durante a simulação, esse parâmetro foi sempre configurado com valor igual a *1*;

- *Number of nodes*, representa o número de nós que estão conectados à rede. Nas simulações realizadas, esse parâmetro foi configurado para *4*, *8* e *20* para os cenários com *1*, *2* e *5* subsistemas de controle, respectivamente;

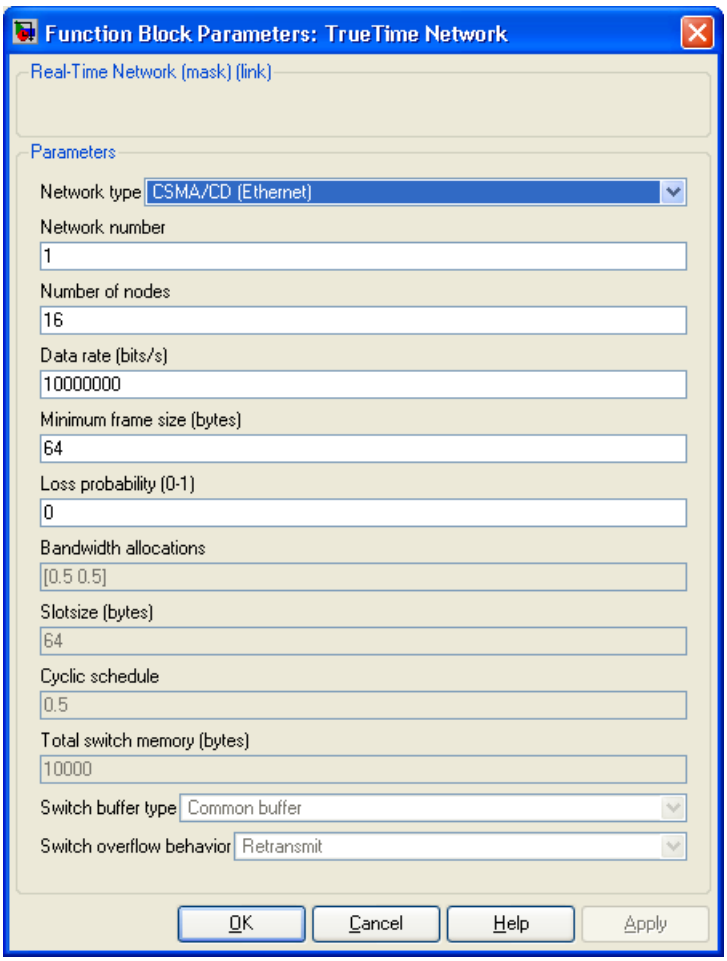


Figura A.3. Exemplo de configuração do bloco *Network*

- *Data rate*, representa a velocidade da rede. Foram utilizadas as velocidades de *10Mbps* para as redes da família *Ethernet* e *500Kbps* para rede *CAN*;

- *Minimum frame size*, representa o tamanho mínimo de um pacote a ser enviado através da rede. Esse parâmetro foi configurado da seguinte forma: *64 bytes* para as redes da família *Ethernet* e *10 bytes* para as redes *CAN*.

- *Total switch memory*, define a quantidade de memória disponível em um *Switched*

Ethernet. Durante as simulações com esse tipo de rede, tal parâmetro foi configurado com *2Mbytes*.

- *Switch Buffer Type*, descreve como a memória é alocada no *Switch*. Configurado como *Commom buffer*.
- *Switch overflow behavior*, define a ação a ser tomada quando o *buffer* de mensagens do switch está cheio. Configurado como *Drop*.

Por fim, todas as simulações no ambiente de Simulink foram configuradas de acordo com a figura A.4

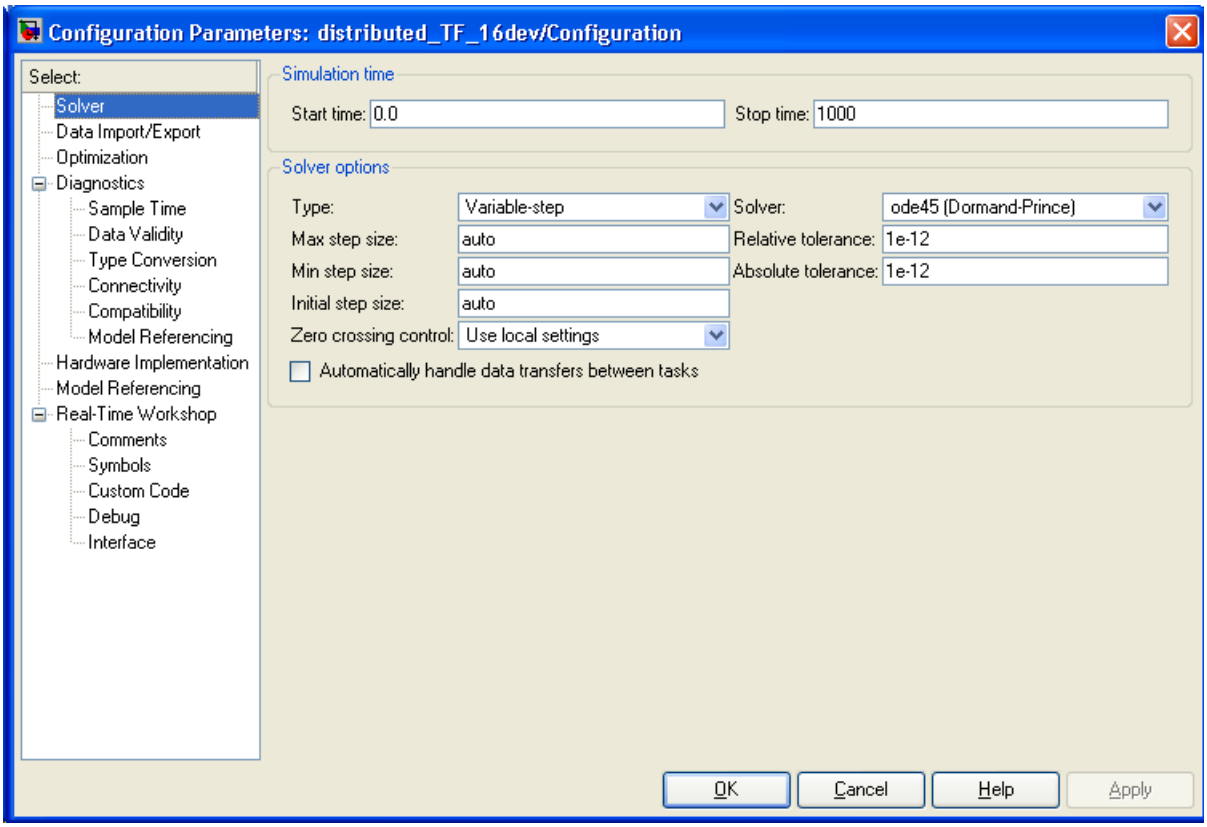


Figura A.4. Configuração da simulação no *Simulink*

Observa-se que, para garantir uma maior precisão, optou-se por utilizar um *solver* baseado em passo de tempo variado, o *solver ode45*. Tal *solver* e suas opções de configuração são descritos com maiores detalhes em The Mathworks (2004).

A.3 PASSOS PARA EXECUÇÃO E OBTENÇÃO DOS RESULTADOS DA SIMULAÇÃO

A fim de facilitar a análise dos dados e diminuir o tempo e consumo de recursos durante a execução das simulações, a mesma foi divida em fases (ver figura A.5).

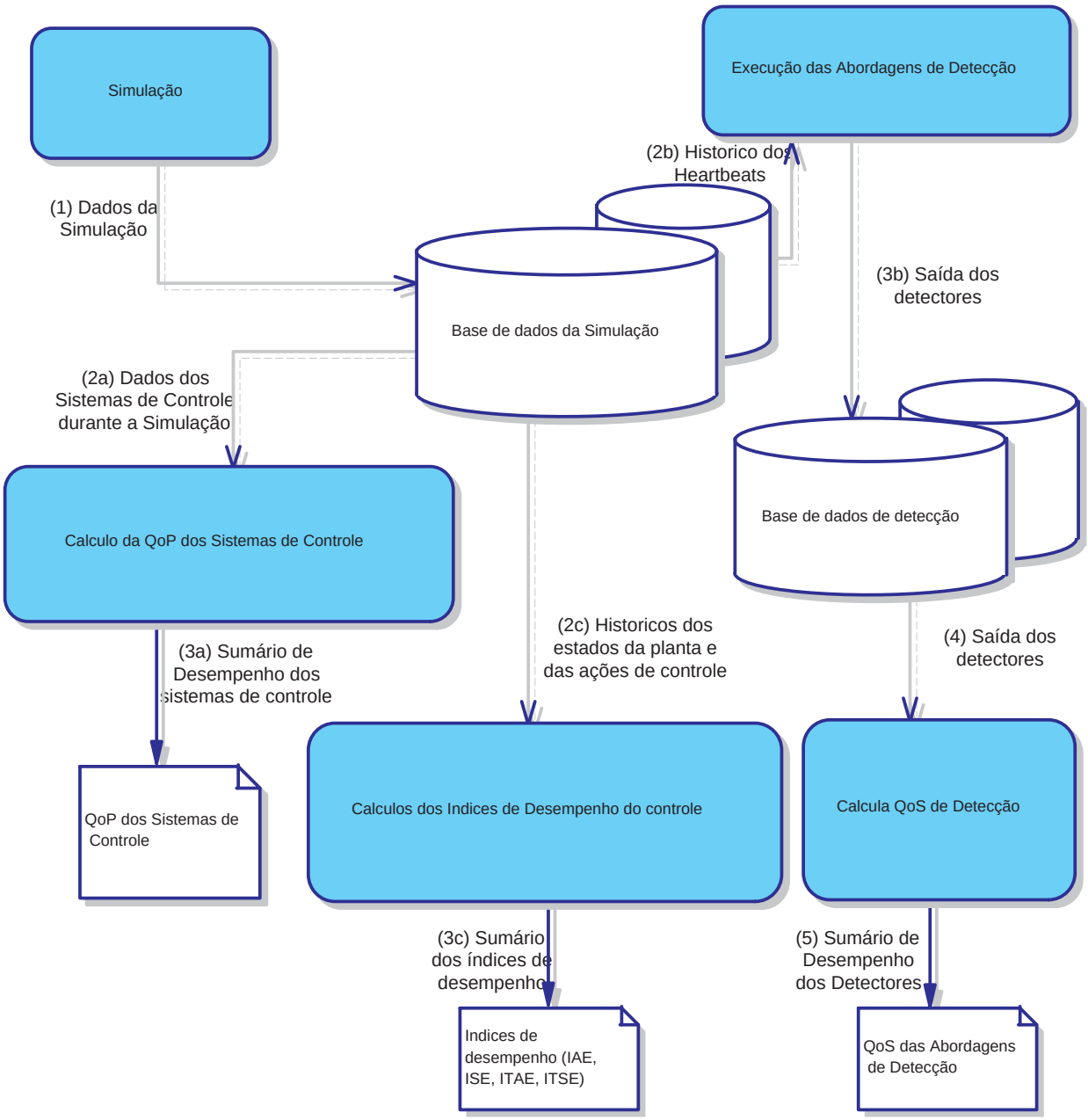


Figura A.5. Passos usados para geração dos resultados da simulação

Essas fases são descritas a seguir.

1) Na primeira fase, o modelo de sistema é simulado e os dados da simulação são coletados. Ao final da simulação, os dados coletatos são armazenados em disco (figura A.5 item *1*).

2) Na segunda fase, os dados armazenados em disco são lidos e em seguida um processamento é realizado. O tipo de processamento realizado dependerá do tipo de análise a ser feita.

   a. Para o cálculo da qualidade de desempenho ou dos índices de erro do sistema de controle, os dados armazenados em disco são lidos e as informações pertinentes às ações de controle são processadas (figura A.5 itens *2a* e *2c*). Após o processamento, um sumário dos resultados obtidos são armazenados em disco (figura A.5 itens *3a*, *3c*).

   b. Para a verificação do desempenho dos detectores de defeitos, por sua vez, necessita-se de dois processamentos adicionais. No primeiro processamento, os dados armazenados em disco são lidos e o histórico das emissões e recepções de *heartbeats* são processados (figura A.5 item *2b*). Nesse processamento, as estratégias de detecção são postas em execução e tuplas contendo as transições na saída de cada detector e os marcos no tempo, referentes a essas transições, são armazenadas novamente em disco (figura A.5 item *3b*).

3) A última fase é requerida apenas para o cálculo da qualidade de serviço dos detectores de defeitos. As saídas que foram armazenadas na fase *2b* são lidas (figura A.5 item *4*), um novo processamento é realizado e os indicadores de qualidade de serviço de cada detector são produzidos. Em seguida, um sumário desse processamento é novamente armazenado em disco (figura A.5 item *5*).

A.4 DESCREVENDO A IMPLEMENTAÇÃO DOS ALGORITMOS

Para que a simulação pudesse ser realizada de acordo com o modelo de sistema desejado, algumas entidades e algoritmos foram implementados utilizando a linguagem de *script* disponível no *Matlab*. Essa linguagem permite não só a integração dos algoritmos implementados com o ambiente *Simulink*, mas também a codificação de entidades usando o paradigma orientado a objetos.

As subseções seguintes trazem alguns detalhes sobre as entidades e os algoritmos implementados.

A.4.1 Projeto das Classes Usadas na Simulação

A figura A.6 apresenta o diagrama das classes implementadas na simulação. O projeto das classes foi realizado de modo a permitir que modificações nos cenários ou nos objetivos da simulação sejam executadas de forma simples e rápida.

Para maior coerência da implementação com o modelo proposto, foram desenvolvidas algumas funções que realizam a instanciação dos objetos seguindo os relacionamentos indicados. Por exemplo, para criar um detector de defeitos baseado na abordagem de (BERTIER; MARIN; SENS, 2002), foi implementada uma função que instancia um detector de defeitos e associa ao mesmo um modelo de monitoramento *push* e uma estratégia de detecção que utiliza o algoritmo de Bertier, Marin e Sens (2002).

Cada uma das principais classes implementadas serão descritas a seguir, enquanto o código fonte das funções utilizadas são apresentados na subseção A.4.2.

A.4.1.1 Classes de Simulação Essas classes estão diretamente relacionadas aos objetivos e ao ambiente de simulação utilizados.

- *Simulation*. Representa a simulação realizada. Essa classe tem como papel fazer a

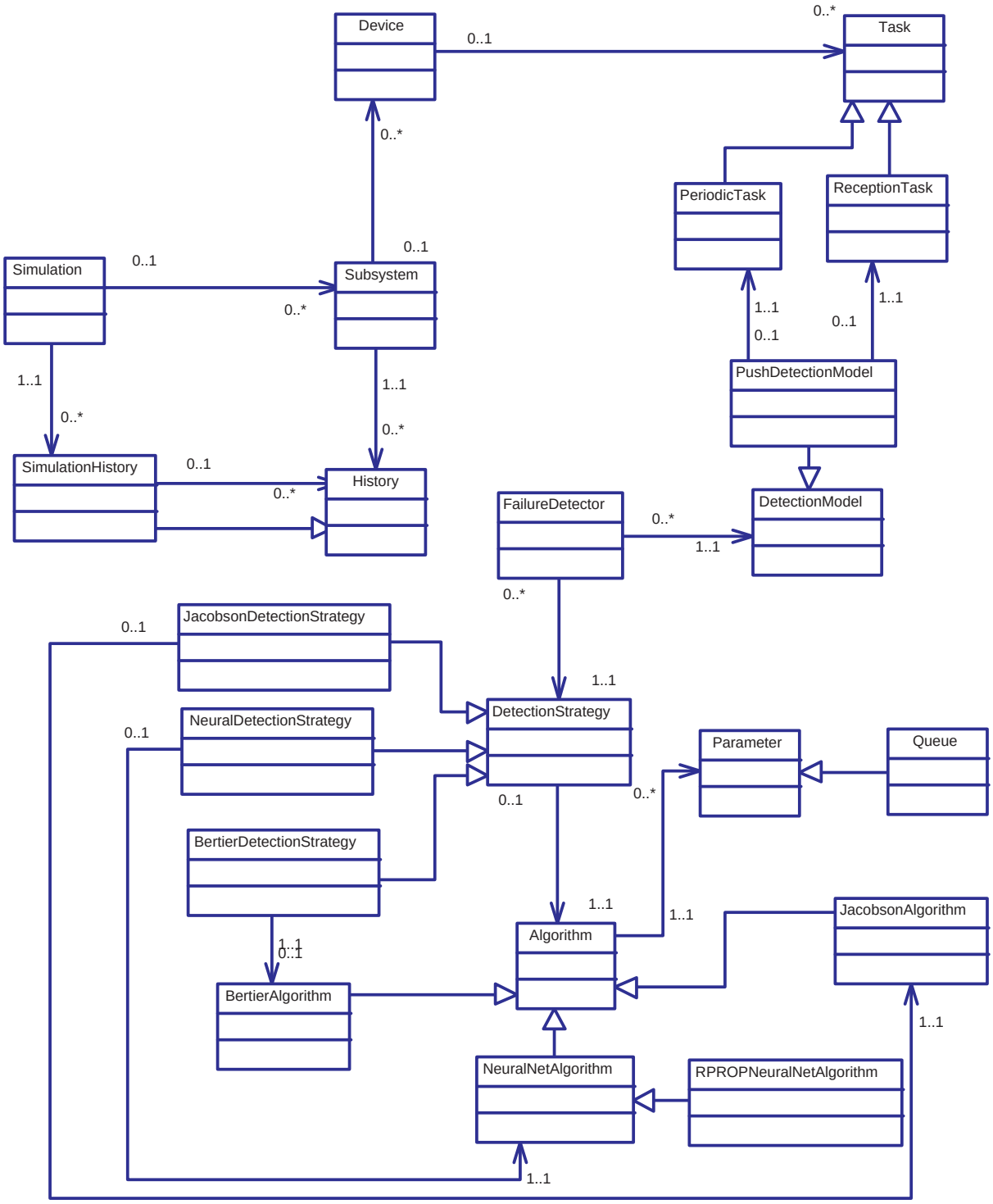


Figura A.6. Modelo de classes usado na simulação

interface com o ambiente *Simulink*, de modo a gerenciar a execução da simulação. Além disso, é papel dessa classe armazenar em histórico as variáveis da simulação a serem utilizadas durante a análise dos resultados e verificar os critérios para o término da simulação.

Os principais métodos dessa classe são descritos a seguir:

- *add_simulation_history*. adiciona uma variável ao histórico da simulação. Instante de chegada e de emissão de *heartbeats*, instantes de sensoriamento e de atuação, entre outros, são exemplos de variáveis que podem ser adicionadas ao histórico da simulação. Em geral, não existem restrições para que uma variável pertença ao histórico; isto será indicado pelos objetivos da simulação.
- *get_simulation_history*. Captura uma variável do histórico da simulação.
- *get_subsystem*. Captura um subsistema integrante do cenário simulado.
- *get_subsystem_device*. Captura um dispositivo específico de um subsistema inserido na simulação.
- *get_subsystem_history_sample*. Captura uma amostra do histórico de estados de uma variável; para tanto, deve-se informar o subsistema ao qual a variável está relacionada.
- *is_empty_subsystem_history*. Verifica se um histórico dos estados de uma variável está vazio.
- *is_finished*. Verifica se os objetivos da simulação foram alcançados e se, portanto, a mesma poderá ser finalizada. Esse método deve ser *sobrecarregado* para diferentes tipos de simulação.
- *is_undefined_subsystem*. Verifica se um determinado subsistema foi realmente definido na simulação.
- *set_subsystem*. Define um subsistema no cenário a ser simulado.

  - *stop*. Interage com o *Simulink* com o intuito de parar uma simulação em andamento.
  - *update_simulation_history*. Atualiza o histórico de estados de uma variável observada na simulação.
  - *update_subsystem_history_sample*. Atualiza uma amostra no histórico de estados de uma variável observada na simulação.

- *Simulation_History*. Representa o histórico das variáveis armazenadas durante a simulação. Todo *Simulation_History* é um *History*.

  Seus principais métodos são:

  - *array2samples*. Permite que uma matriz de amostras seja transformada em item de um histórico de estados de uma determinada variável observada.
  - *get_next_sample*. Funciona como um *iterator*, permitindo uma navegação no conjunto de amostras de um histórico.
  - *get_sample*. Permite que uma amostra específica seja selecionada a partir de sua posição no histórico.
  - *is_full*. Uma vez que durante a construção de um *SimulationHistory* pode-se determinar o número máximo de amostras a serem acomodadas no histórico, esse método permite que se possa verificar se o número máximo de amostras informadas foi alcançado.
  - *set_next_sample*. Funciona como um *iterator*, permitindo uma atribuição seqüencial de valores ao histórico de amostras.
  - *set_sample*. Permite especificar o valor de uma determinada amostra dentro do histórico de amostras.

A.4.1.2 Classes do Modelo de Sistema Esse conjunto de classes estão relacionadas ao modelo do sistema proposto nas simulações.

- *SubSystem*. Representa um subsistema pertencente ao cenário simulado. Todo subsistema mantém um histórico das ações de interesse realizadas durante sua execução. Um subsistema é formado por uma coleção de dispositivos. Os principais métodos dessa classe são descritos a seguir:
  - *add_history*. Adiciona uma instância de histórico de estados de variável a um subsistema.
  - *get_device*. Captura um dispositivo ao subsistema.
  - *get_history*. Captura um histórico de estados de uma variável da lista de históricos do subsistema.
  - *get_history_sample*. Permite que uma amostra específica de um histórico seja selecionada a partir de sua posição no histórico.
  - *init*. Inicializa a execução de um subsistema. Esse método faz com que cada um dos dispositivos associados ao subsistema seja iniciado.
  - *is_empty_history*. Verifica se um histórico de estados está vazio.
  - *is_full_history*. Verifica se um histórico de estados está cheio.
  - *set_device*. Atribui um dispositivo ao subsistema.
  - *update_history*. atualiza um histórico de estados com um novo estado de variável.
  - *update_history_sample*. Possui a mesma responsabilidade que o método *update_history*, com a exceção de que cede o controle da amostra a ser atualizada para a entidade usuária do método.

- *History*. Representa o histórico das ações em um subsistema. Seus principais métodos são:

  - *add_sample*. Adiciona uma amostra ao histórico.
  - *array2samples*. Permite que uma matriz de amostras seja transformada em item em um histórico de estados de uma determinada variável observada.
  - *compact*. Realiza a compactação do histórico, removendo possíveis estados inconsistentes ou posições de histórico vazias.
  - *diff*. Calcula diferenças (distâncias) entre dois estados subseqüentes existentes no histórico. Nesse método, pode-se parametrizar o atributo da amostra a ser usado no cálculo das diferenças.
  - *get_next_sample*. Funciona como um *iterator*, permitindo uma navegação no conjunto de amostras de um histórico.
  - *get_sample*. Permite que uma amostra específica seja selecionada a partir de sua posição no histórico.
  - *is_empty*. Verifica se o histórico está vazio.
  - *is_full*. Uma vez que durante a construção de um *History* pode-se determinar o número máximo de amostras a serem acomodadas no histórico, esse método permite que se possa verificar se o número máximo de amostras informadas foi alcançado.
  - *max*. Captura o maior dentre os valores de variáveis armazenadas no histórico.
  - *min*.Captura o menor dentre os valores de variáveis armazenadas no histórico.
  - *minmax*. Captura o maior e o menor dentre os valores de variáveis armazenadas no histórico.
  - *reset*. Inicializa o histórico, tornando todas as posições vazias.

  - *set_next_sample*. Funciona como um *iterator*, permitindo uma atribuição seqüencial de valores ao histórico de amostras.
  - *set_sample*. Permite especificar o valor de uma determinada amostra dentro do histórico de amostras.

- *Device*. Representa um dispositivo. Todo dispositivo pode executar uma ou mais tarefas. Controlador, sensor e atuador são exemplos de dispositivo. Os principais métodos suportados por essa classe são descritos a seguir:

  - *add_task*. Adiciona uma tarefa ao conjunto de tarefas a serem executadas pelo dispositivo.
  - *get_task*. Captura uma tarefa do conjunto de tarefas do dispositivo.
  - *init*. Inicializa a execução do dispositivo, pondo o kernel de seu sistema operacional em execução. O código fonte desse método pode ser visto a seguir.

Algoritmo A.1. Implementação do método *init* da classe *Device*

```
1 function my_device = init(my_device)
2
3 ttInitKernel(my_device.inputs, my_device.outputs, my_device.schedule);
```

  - *init_tasks*. Inicializa(ponhe em execução) o conjunto de tarefas do dispositivo. Sua implementação pode ser vista a seguir.

Algoritmo A.2. Implementação do método *init_tasks* da classe *Device*

```
1 function my_device = init_tasks(my_device)
2
3 ntasks = length(my_device.tasks);
4
5 for task_index = 1 : ntasks
6     my_task = get_task(my_device, task_index);
7     my_task = init(my_task);
8 end
```

  - *set_task*. Substitue uma tarefa previamente associada a um dispositivo.

- *Task*. Representa uma tarefa executada em um dispositivo. Essa classe tem como papel *interfacear* com o pacote *TrueTime* de modo a garantir a criação e gestão das tarefas no bloco de tempo real. Seu principal método é:

  – *init*. Usado para por uma tarefa previamente configurada para a execução.

- *Periodic_Task* e *Reception_Task*. Representam respectivamente uma tarefa periódica e uma tarefa esporádica preparada para a recepção de mensagens. *Periodic_Task* e *Reception_Task* são subtipos de *Task*. Os métodos de inicialização (*init*) dessas classes são apresentados a seguir.

Algoritmo A.3. Implementação do método *init* da classe *Periodic_Task*

```
function my_task = init(my_task)
ttCreatePeriodicTask( ...
    get(my_task, 'name'),     get(my_task, 'release'), get(my_task, 'period'), ...
    get(my_task, 'priority'), get(my_task, 'code'),    get(my_task, 'data') ...
);
```

Algoritmo A.4. Implementação do método *init* da classe *Reception_Task*

```
function my_task = init(my_task)

task_name   = get(my_task, 'name');
my_device   = get(my_task, 'device');
dev_address = get(my_device, 'address');

ttCreateInterruptHandler( ...
    task_name, ...
    get(my_task, 'priority'), ...
    get(my_task, 'code'), ...
    get(my_task, 'data') ...
);

ttInitNetwork(dev_address, task_name);
```

A.4.1.3 Classes de Detecção Essas classes estão relacionadas às abordagens de detecção utilizadas durante as simulações e análises.

- *Failure_Detector*. Representa um módulo detector de defeitos. Todo módulo detector segue um modelo de detecção e realiza sua atividade seguindo uma estratégia de detecção previamente definida. O principal método dessa classe é:

– *is_fail*. Determina o atual estado de um dispositivo que está sendo monitorado. O retorno é um *booleano* que, quando definido como verdadeiro, indica que o dispositivo é suspeito de ter falhado; caso contrário, indica que o dispositivo está correto. Esse método é implementado como segue:

Algoritmo A.5. Implementação do método que reporta o estado de um dispositivo no sistema

```
1  function fail = is_fail(fd, device)
2
3  now = ttCurrentTime;
4
5  monitor_name = strcat(device, '_', monitor);
6
7  [fd,   currentEA] = query(fd, monitor_name);
8
9  fail = currentEA < now;
```

- *Detection_Model*. Representa um modelo para detecção de defeitos. O principal método contido nessa classe é:

  – *init*. Usado para iniciar o modelo de monitoramento, colocando as tarefas responsáveis por monitoramento e indicação de estado de dispositivo em execução. O tipo e o conjunto de tarefas a serem iniciadas dependerá do modelo de monitoramento implementado.

- *Push_Detection_Model*. É um modelo de detecção. Representa o modelo *push* de monitoramento de falhas. Esse modelo é composto por uma tarefa periódica de emissão de *heartbeats* e uma tarefa esporádica de recepção de *heartbeats*. *Push_Detection_Model* é um subtipo de *Detection_Model*.

  Os principais métodos dessa classe são descritos a seguir:

  – *create_monitor*. Cria uma tarefa de monitoramento de falhas e associa ao dispositivo que deseja verificar o estado dos demais dispositivos. A implementação desse método é apresentada a seguir.

Algoritmo A.6. Implementação do modelo de monitoramento *Push*

```
function [my_push_detection_model, monitor] = create_monitor(my_push_detection_model)

it = get_interrogation_time;

my_push_detection_model.monitor = create_sporadical_task( ...
    my_push_detection_model.monitor, ...
    'failure_detector_monitor', ...
    'failure_detector_monitor', ...
    'failure_monitor', 1, it, {} ...
);

monitor = my_push_detection_model.monitor;
```

– *create_sender*. Cria uma tarefa de indicação de estado e associa ao dispositivo que deve ser monitorado. Abaixo é apresentada a implementação desse método:

Algoritmo A.7. Implementação do modelo de emissão de *heartbeats Push*

```
function [my_push_detection_model, sender] = create_sender(my_push_detection_model)

it = get_interrogation_time;

message.id = 0;

my_push_detection_model.sender = create_periodic_task( ...
    my_push_detection_model.sender, ...
    'failure_detector_emissor', ...
    'failure_detector_sender', ...
    'detector_sniffer', 2, it, 0.0, ...
    message ...
);

sender = my_push_detection_model.sender;
```

– *init*. Sobrecarrega o método *init* da classe *Detection_Model*, de modo que se possa colocar em execução as tarefas associadas ao modelo de monitoramento em questão. A implementação desse método é apresentada a seguir.

Algoritmo A.8. Implementação do método *init* da classe *Push_Detection_Model*

```
function my_push_detection_model = init( ...
        my_push_detection_model, my_failure_detector, type ...
)

message.id = -1;

it = get(my_failure_detector, 'interrogation_time');

switch lower(type)
    case {'monitor'}
        ttCreateTask( ...
```

```
12                  'failure_detector_monitor', ...
13                  it, 1, 'failure_detector_monitor' ...
14          );
15      case {'emissor'}
16          ttCreatePeriodicTask( ...
17                  'failure_detector_emissor', 0.0, it, 1, ...
18                  'failure_detector_sender', message ...
19          );
20  end
21
22  my_push_detection_model.initted = 1;
```

- *Detection_Strategy*. Representa uma estratégia de detecção. Toda estratégia de detecção usa um algoritmo para detecção de falhas.
  - *execute*. Executa a estratégia de detecção de defeitos. Esse método é responsável por executar o algoritmo de detecção e verificar os resultados seguindo a abordagem de detecção de defeitos implementada pelo detector.
  - *hibernate_algorithm*. Usado para salvar em meio persistente o estado atual do algoritmo usado pela estratégia de detecção.
  - *restore_algorithm*. Usado para recuperar de um meio persistente o estado de um algoritmo anteriormente utilizado por estratégia de detecção.
- *Jacobson_Detection_Strategy*. É uma estratégia de detecção que usa o algoritmo de (JACOBSON, 1988). Essa estratégia sobrecarrega o método *execute* da classe *Detection_Strategy*. Esse método é apresentado a seguir.

Algoritmo A.9. Implementação do método usado para a execução da estratégia de detecção de (JACOBSON, 1988)

```
1   function [my_strategy, the_FP] = execute(my_strategy, heartbeat)
2
3   %get Arrival time (A) and Expected Arrival time (EA) for last estimative
4   last_A = get(my_strategy, 'ArrivalTime');
5
6   the_A  = heartbeat.arrival;
7
8   %get the last Freshness Point (FP) and Interrogation Time (IT)
9   the_FP = get(my_strategy,     'FreshnessPoint');
10  the_IT = get(my_strategy, 'InterrogationTime');
11
12  delay = the_IT;
13
14  if (last_A ~= -1)
```

```
15          delay = the_A - last_A;
16  end
17
18  %get the jacobson algorithm and prepare the input
19  the_algorithm = get(my_strategy   ,          'Algorithm');
20  the_algorithm = set(the_algorithm, 'Input', delay);
21
22  %run jacobson algorithm to get the safety margin (alpha)
23  [the_algorithm, the_delay] = run(the_algorithm);
24
25  the_FP = the_A + the_delay;
26
27  %save the strategy state
28  my_strategy = set(my_strategy, 'Algorithm'      , the_algorithm);
29  my_strategy = set(my_strategy, 'FreshnessPoint',        the_FP);
30  my_strategy = set(my_strategy, 'ArrivalTime'    ,         the_A);
```

- *Bertier_Detection_Strategy.* É uma estratégia de detecção que usa o algoritmo de (BERTIER; MARIN; SENS, 2002). Essa estratégia sobrecarrega o método *execute* da classe *Detection_Strategy.* Esse método é apresentado a seguir.

Algoritmo A.10. Implementação do método usado para a execução da estratégia de detecção de (BERTIER; MARIN; SENS, 2002)

```
1   function [my_strategy, the_FP] = execute(my_strategy, heartbeat)
2
3   %get the bertier algorithm
4   the_algorithm = get(my_strategy,    'algorithm');
5   A_history     = get(the_algorithm, 'A_history');
6
7   A_history     = add_sample(A_history,       heartbeat.arrival);
8
9   the_algorithm = set(the_algorithm    , 'A_history', A_history);
10
11  %run bertier algorithm to get the freshness point (FP)
12  [the_algorithm, the_FP] = run(the_algorithm);
13
14  %save the strategy state
15  my_strategy = set(my_strategy, 'algorithm', the_algorithm);
```

- *Neural_Net_Detection_Strategy.*É uma estratégia de detecção que usa uma rede neural como algoritmo. Essa estratégia sobrecarrega o método *execute* da classe *Detection_Strategy.* Esse método é implementado como segue.

Algoritmo A.11. Implementação do método usado para a execução da estratégia de detecção baseada em RNA

```
1   function [my_strategy, the_FP] = execute(my_strategy, heartbeat)
2
3   the_FP    = -1;
4
5   the_queue = get(my_strategy, 'queue');
```

```
6  i_time     = get(my_strategy, 'interrogationtime');
7  the_queue = offer(the_queue, heartbeat);
8  queue_size = get(the_queue, 'size');
9
10 if (is_full(the_queue))
11
12    the_pattern   = create_neural_pattern(the_queue, i_time);
13
14    the_algorithm   = get(my_strategy   , 'algorithm');
15    mapper          = get(the_algorithm,     'mapper');
16
17    if isobject(mapper)
18       %calcula o numero de elementos no padrao de entrada
19       p_size = size(the_pattern);
20       p_max  = 1;
21       for p = 1 : length(p_size)
22          p_max  = p_max * p_size(p);
23       end
24
25       %normaliza entrada
26       for p = 1 : p_max
27          [mapper, the_pattern(p)] = run(mapper, the_pattern(p));
28       end
29    end
30
31    %atualiza padrao de entrada
32    the_algorithm = set(the_algorithm, 'pattern'   ,    the_pattern);
33
34    %atualiza a estrategia de deteccao
35    my_strategy   = set(my_strategy   , 'algorithm', the_algorithm);
36
37    %verifica o ultimo valor calculado pela RNA
38    last_delay    = get(the_algorithm, 'return');
39
40    %captura o penultimo elemento da pilha
41    last_heartbeat= get(the_queue, queue_size - 1);
42
43    A    = heartbeat.arrival;
44
45    %checa o ultimo atraso
46    if isempty(last_delay)
47       the_algorithm   = set(the_algorithm, 'return', i_time);
48       last_delay                = i_time;
49    end
50
51    last_EA                      = last_delay + last_heartbeat.arrival;
52
53    [my_strategy.detection_strategy, ann_output] = execute(my_strategy.detection_strategy);
54
55    if isobject(mapper)
56       %calcular o atraso esperado
57       [mapper, ann_output(1)] = reverse(mapper, ann_output(1));
58
59       %calcular o jitter esperado
60       [mapper, ann_output(2)] = reverse(mapper, ann_output(2));
61    end
62
63    the_FP = heartbeat.arrival + (i_time + ann_output(1)) + ann_output(2);
64
65 end
66
67 my_strategy = set(my_strategy, 'queue', the_queue);
```

- *Jacobson_Algorithm*. Representa a implementação do algoritmo de (JACOBSON,

1988). *Jacobson_Algorithm* é um subtipo de *Algorithm*. Essa classe sobrescreve o método *run* da classe *Algorithm*, resultando na implementação abaixo:escreve o método *run* da classe *Algorithm*, resultando na implementação abaixo:

Algoritmo A.12. Implementação do método usado para a predição dos instantes de chegadas dos *heartbeats* usando abordagem de (JACOBSON, 1988)

```
1  function [my_algorithm, the_output] = run(my_algorithm)
2
3  %get jacobson algorithm parameters
4  the_input    = get(my_algorithm,    'input');
5  the_delay    = get(my_algorithm,    'delay');
6  the_variance = get(my_algorithm, 'variance');
7  gamma        = get(my_algorithm,    'gamma');
8  beta         = get(my_algorithm,     'beta');
9  phi          = get(my_algorithm,      'phi');
10
11 the_error    = the_input    - the_delay;
12 the_delay    = the_delay    + gamma     * the_error;
13 the_variance = the_variance + gamma     * (abs(the_error) - the_variance);
14 the_output   = beta         * the_delay + phi * the_variance;
15
16 my_algorithm = set(my_algorithm, 'variance', the_variance);
17 my_algorithm = set(my_algorithm, 'delay'   ,    the_delay);
18 my_algorithm = set(my_algorithm, 'return'   ,  the_output);
```

- *Bertier_Algorithm*. Representa a implementação do algoritmo de (BERTIER; MARIN; SENS, 2002). *Bertier_Algorithm* é um subtipo de *Algorithm*.

  Os principais métodos dessa classe são descritos a seguir:

  – *compute_safety_margin*. Calcula a margem de segurança. A implementação desse método é apresentada a seguir:

  Algoritmo A.13. Implementação do método usado para o cálculo da margem de segurança no algoritmo de (BERTIER; MARIN; SENS, 2002)

```
1  function [my_algorithm, the_safety_margin] = compute_safety_margin(my_algorithm, current_A,
2
3  %prepare jacobson algorithm
4  the_jacobson_algorithm = get(my_algorithm, 'the_jacobson_algorithm');
5
6  the_jacobson_algorithm = set(the_jacobson_algorithm, 'input', current_A - last_EA);
7
8  %run algorithm and get a safety margin
9  [the_jacobson_algorithm, the_safety_margin] = run(the_jacobson_algorithm);
10
11 %update jacobson algorithm state
12 my_algorithm    = set(my_algorithm, 'the_jacobson_algorithm', the_jacobson_algorithm);
```

– *get_safety_margin*. Captura o último valor calculado para a margem de segurança.

– *run*. Sobrecarga o método *run* da classe *Algorithm*, com o intuito de implementar a abordagem de (BERTIER; MARIN; SENS, 2002). A implementação desse método pode ser vista abaixo:

Algoritmo A.14. Implementação do método usado para a predição dos instantes de chegadas dos *heartbeats* usando abordagem de (BERTIER; MARIN; SENS, 2002)

```
1  function [my_algorithm, the_freshness_point] = run(my_algorithm)
2
3  the_freshness_point = -1;
4
5  A_history = get(my_algorithm, 'A_history');
6
7  if ~is_empty(A_history)
8     %get the arrival time history
9     current                 = get(A_history,  'current');
10    [A_history, A] = get_next_sample(A_history);
11
12    %get initialization window size and heartbeat rate
13    window_size        = get(my_algorithm,        'window_size');
14    interrogation_time = get(my_algorithm, 'interrogation_time');
15
16    if current < window_size
17       U       = A;          %U(1) = A(0)
18       last_EA = A;
19
20       if current >= 0
21          U  = get(my_algorithm, 'U');
22          U  = (A  / (current + 1)) * ((current * U) / (current + 1));
23
24          %get last EA
25          last_EA = get(my_algorithm, 'EA');
26       end
27
28       EA = U + ((current  + 1) * interrogation_time) / 2;
29       my_algorithm    = set(my_algorithm, 'U', U);
30
31    else
32       last_EA = get(my_algorithm, 'EA');
33       past_A  = get_sample(A_history, current - window_size + 1);
34
35       %get a new estimation for arrival time
36       EA      = last_EA + (A - past_A) / window_size;
37    end
38
39    %get the safety margin
40    [my_algorithm, the_safety_margin] = ...
41       compute_safety_margin(my_algorithm, A, last_EA);
42
43    %compute the freshness point
44    the_freshness_point = EA + the_safety_margin;
45
46    %update the bertier estimation variables
47    my_algorithm    = set(my_algorithm, 'A_history',             A_history);
48    my_algorithm    = set(my_algorithm, 'EA'       ,                    EA);
49    my_algorithm    = set(my_algorithm, 'return'   , the_freshness_point);
50 end
```

- *Neural_Net_Algorithm*. Representa a implementação do algoritmo de um algoritmo baseado em *RNA*. O principal método dessa classe é:

  - *run*. Sobrecarga o método *run* da classe *Algorithm*, com o intuito de executar o modelo matemático proposto pela *RNA*. A implementação desse método pode ser vista abaixo:

Algoritmo A.15. Implementação do método usado para a predição dos instantes de chegadas dos *heartbeats* usando abordagem baseada em *RNA*

```
1 function [my_algorithm, the_output] = run(my_algorithm)
2
3 the_net     = get(my_algorithm,    'network'); %captura a rede neural
4 the_pattern = get(my_algorithm,    'pattern'); %captura o padrão
5 the_output  = sim(the_net      , the_pattern); %solicita o processamento do padrão na rede
6
7 my_algorithm = set(my_algorithm, 'return', the_output);%devolve resultado do processamento
```

  - *train*. Usado para proceder o treinamento da *RNA*. Esse método dependerá do modelo de treinamento a ser utilizado. Dessa forma, cada subtipo dessa classe deverá sobrecarregar esse método.

- *RPROP_Neural_Net_Algorithm*. Representa a implementação do algoritmo de um algoritmo baseado em *RNA* com treinamento *RPROP*.

  - *is_undefined_network*. Verifica se a *RNA* usada pelo algoritmo já foi definida.
  - *save*. Salva o estado atual do algoritmo em meio persistente. Isso inclui os parâmetros relacionados e a *RNA* utilizada.
  - *restore*. Restaura o estado de um algoritmo a partir de um persistente.
  - *train*. Usado para proceder o treinamento da *RNA* usando o algoritmo *RPROP*.

A.4.1.4 Classes Utilitárias

- *Algorithm*. Classe abstrata que representa um algoritmo. Todo algoritmo recebe uma coleção de parâmetros, realiza um processamento e produz um resultado.
  - *add_parameter*. Adiciona um parâmetro ao conjunto de parâmetros necessários à execução do algoritmo.
  - *get_parameter*. Captura um parâmetro do conjunto de parâmetros utilizados pelo algoritmo.
  - *run*. Põe o algoritmo em execução.
  - *update_parameter*. Atualiza o valor de um parâmetro do conjunto de parâmetros utilizados por um algoritmo.
- *Parameter*. Classe que representa um parâmetro usado em um algoritmo.
- *Queue*. Representação de uma estrutura de fila. É um parâmetro para um algoritmo. Na simulação é usada para enfileirar os *heartbeats*. Os principais métodos dessa classe são descritos a seguir:
  - *is_empty*. Verifica se à fila está vazia.
  - *is_full*. Verifica se a fila está cheia.
  - *offer*. Enfileira um elemento oferecido a fila.
  - *peek*. Captura o primeiro elemento da fila.

A.4.2 Principais Funções Implementadas

A.4.2.1 Funções para Inicialização dos Dispositivos .

Algoritmo A.16. Código utilizado na inicialização do controlador primário

```
% Função controller_init
% * funcão utilitária usada para inicializar o controlador primário.
%
% Entradas:
% * deviceID - determina o identificador do dispositivo no subsistema.
% Veja Também:
%
% * backup_controller_init,
% * sensor_init,
% * actuator_init

function controller_init(deviceID)

global my_simulation; %variável contendo uma instância da simulação
global assigner; %variável que armazena a ordem dos dispositivos

%inicializa assigner
initialize_assigner;

%associa nome do dispositivo ao seu id no subsistema
assigner.devices{deviceID} = 'controller';

%inicializa variaveis globais pertinentes ao subsistema e a simulação
[my_simulation, my_system] = ...
        initialize_global_variables(assigner.devices{deviceID});

%inicia o dispositivo no subsistema
[my_system, my_device] = start_device(my_system, deviceID);
```

Algoritmo A.17. Código utilizado na inicialização do controlador secundário

```
% Função backup_controller_init
% * funcão utilitária usada para inicializar o controlador secundário.
%
% Entradas:
% * deviceID -
%       determina o identificador do dispositivo no subsistema.
%
% Veja Também:
% * controller_init,
% * sensor_init,
% * actuator_init

function backup_controller_init(deviceID)

global my_simulation; %variável contendo uma instância da simulação
global assigner; %variável que armazena a ordem dos dispositivos

%inicializa assigner
initialize_assigner;

%associa nome do dispositivo ao seu id no subsistema
assigner.devices{deviceID} = 'backup-controller';

%inicializa variaveis globais pertinentes ao subsistema e a simulação
[my_simulation, my_system] = ...
        initialize_global_variables(assigner.devices{deviceID});

%inicia o dispositivo no subsistema
[my_system, my_device] = ...
        start_device(my_system, deviceID);
```

Algoritmo A.18. Código utilizado na inicialização do atuador

```
% Função actuator_init
%  * função utilitária usada para inicializar o dispositivo atuador
%
% Entradas:
%  * deviceID - determina o identificador do dispositivo no subsistema
%
% Veja Também:
%  * controller_init,
%  * backup_controller_init,
%  * sensor_init

function actuator_init(deviceID)

global my_simulation;    %variável contendo uma instância da simulação
global assigner;         %variável q/ armazena a ordem dos dispositivos
initialize_assigner;     %inicializa assigner

%associa nome do dispositivo ao seu id no subsistema
assigner.devices{deviceID} = 'actuator';

%inicializa variaveis globais pertinentes ao subsistema e a simulação
[my_simulation, my_system] = ...
        initialize_global_variables(assigner.devices{deviceID});

%inicia o dispositivo no subsistema
[my_system, my_device] = start_device(my_system, deviceID);
```

Algoritmo A.19. Código utilizado na inicialização do sensor

```
% Função sensor_init
% * função utilitária usada para inicializar o dispositivo sensor
% Entradas:
% * deviceID, determina o id do dispositivo no subsistema
% Veja Também:
% * backup_controller_init,
% * controller_init,
% * actuator_init

function sensor_init(deviceID)
global my_simulation;    %variável contendo uma instância da simulação
global assigner;         %variável q/ armazena a ordem dos dispositivos
initialize_assigner;     %inicializa assigner

%associa nome do dispositivo ao seu id no subsistema
assigner.devices{deviceID} = 'sensor';

%inicializa variaveis globais pertinentes ao subsistema e a simulação
[my_simulation, my_system] = ...
        initialize_global_variables(assigner.devices{deviceID});

%inicia o dispositivo no subsistema
[my_system, my_device] = start_device(my_system, deviceID);
```

A.4.2.2 Funções para Instanciar e Parametrizar Dispositivos .

Algoritmo A.20. Função usada para instanciar e configurar um controlador primário

```
% Função new_controller_device: função utilitária instanciar e confi-
% gurar um dispositivo controlador primário.
%
```

```
4  % Entradas:
5  % * subsystem_index - indice do subsistema que contém o dispositivo
6  % * dev_index - indice do dispositivo no subsistema.
7  %
8  % Saídas: a instância do controlador
9  % Veja Também:
10 % * new_backup_controller_device,
11 % * new_sensor_device,
12 % * new_actuator_device
13
14 function my_controller = ...
15         new_controller_device(subsystem_index, dev_index)
16
17 %captura o endereço do dispositivo na rede
18 dev_address     = get_device_address(subsystem_index, dev_index);
19 ad_conversors = 1;
20
21 %instancia um controlador e atribui endereço de rede, indice e a po-
22 %litica de prioridades a ser usada no kernel do dispositivo
23 my_controller   = device( ...
24         'controller', dev_address, dev_index, ad_conversors, 0, 'prioFP'...
25 );
26
27 %define atributos da tarefa
28 task_name       = 'pid_task';
29 task_code       = 'ctrlcode1';
30 task_desc       = 'control';
31 task_priority   = 1;
32 task_deadline   = 0.010;
33 task_period     = 0.010;
34 task_release    = 0.000;
35 task_data       = PID_data;
36 %uses to distributed controller and sensor
37 my_controller   = create_sporadical_task(my_controller, task_name, ...
38         task_code, task_desc, task_priority, task_deadline, task_data ...
39 );
40
41 %prepara tarefa para manipulação de recepção de mensagens
42 task_name       = 'nw_handler';
43 task_code       = 'msgRcvCtrl';
44 task_desc       = 'receive';
45 task_priority   = 2;
46 my_controller   = create_reception_task( ...
47   my_controller, task_name, task_code, task_desc, task_priority ...
48 );
```

Algoritmo A.21. Função usada para instanciar e configurar um controlador secundário

```
1  % Função new_backup_controller_device: funcão utilitária instanciar
2  % e configurar um dispositivo controlador primário.
3  %
4  % Entradas:
5  % * subsystem_index - indice do subsistema que contém o dispositivo
6  % * dev_index - indice do dispositivo no subsistema.
7  %
8  % Saídas: a instância do controlador
9  % Veja Também:
10 % * new_controller_device,
11 % * new_sensor_device,
12 % * new_actuator_device
13
14 function my_controller = ...
15         new_backup_controller_device(subsystem_index, dev_index)
16 %captura o endereço do dispositivo na rede
17 dev_address     = get_device_address(subsystem_index, dev_index);
18 ad_conversors = 1;
```

```
19
20 %instancia um controlador e atribui endereço de rede, indice e a po-
21 %litica de prioridades a ser usada no kernel do dispositivo
22 my_controller   = device( ...
23         'backup-controller', dev_address, dev_index, ...
24         ad_conversors, 0, 'prioFP'...
25 );
26
27 %define atributos da tarefa
28 task_name       = 'pid_task';
29 task_code       = 'ctrlcode2';
30 task_desc       = 'control';
31 task_priority   = 1;
32 task_deadline   = 0.010;
33 task_period     = 0.010;
34 task_release    = 0.000;
35 task_data       = PID_data;
36
37 %uses to distributed controller and sensor
38 my_controller   = create_sporadical_task( ...
39         my_controller, task_name, task_code, task_desc, ...
40         task_priority, task_deadline, task_data ...
41 );
42
43 %prepare controller device receive task
44 task_name       = 'nw_handler';
45 task_code       = 'msgRcvBackupCtrl';
46 task_desc       = 'receive';
47 task_priority   = 2;
48 my_controller   = create_reception_task( ...
49   my_controller, task_name, task_code, task_desc, task_priority ...
50 );
```

Algoritmo A.22. Função usada para instanciar e configurar um dispositivo sensor

```
1  % Função new_sensor_device: funcão utilitária instanciar e confi-
2  % gurar um dispositivo sensor.
3  %
4  % Entradas:
5  % * subsystem_index - indice do subsistema que contém o dispositivo
6  % * dev_index - indice do dispositivo no subsistema.
7  %
8  % Saídas: a instância do controlador
9  % Veja Também:
10 % * new_controller_device,
11 % * new_backup_controller_device,
12 % * new_actuator_device,
13
14 function my_sensor = new_sensor_device(subsystem_index, dev_index)
15
16 %captura o endereço do dispositivo na rede
17 dev_address     = get_device_address(subsystem_index, dev_index);
18
19 %instancia um sensor e atribui endereço de rede, indice e a po-
20 %litica de prioridades a ser usada no kernel do dispositivo
21 my_sensor = device('sensor', dev_address, dev_index, 1, 0, 'prioFP');
22
23 %define atributos da tarefa
24 task_name       = 'sens_task';
25 task_code       = 'senscode';
26 task_desc       = 'aquisition';
27 task_priority   = 1;
28 task_period     = 0.010;
29 task_realease   = 0;
30 task_data.y     = 0;
31
```

```
32 %cria dispositivo sensor
33 my_sensor = create_periodic_task(...
34         my_sensor, task_name, task_code, ...
35         task_desc, task_priority, task_period, ...
36         task_realease, task_data ...
37 );
38
39 %cria tarefa de recepção do sensor
40 task_name       = 'nw_handler';
41 task_code       = 'msgRcvSensor';
42 task_desc       = 'receive';
43 task_priority   = 1;
44 my_sensor       = create_reception_task( ...
45   my_sensor, task_name, task_code, task_desc, task_priority ...
46 );
```

Algoritmo A.23. Função usada para instanciar e configurar um dispositivo atuador

```
1  % Função new_actuator_device: funcão utilitária instanciar e confi-
2  % gurar um dispositivo atuador.
3  %
4  % Entradas:
5  % * subsystem_index - indice do subsistema que contém o dispositivo
6  % * dev_index - indice do dispositivo no subsistema.
7  %
8  % Saídas: a instância do controlador
9  %
10 % Veja Também:
11 % * new_controller_device, new_backup_controller_device,
12 % * new_sensor_device,
13
14 function my_actuator = new_actuator_device(subsystem_index, dev_index)
15
16 %captura o endereço do dispositivo na rede
17 dev_address     = get_device_address(subsystem_index, dev_index);
18
19 %instancia um atuador e atribui endereço de rede, indice e a po-
20 %litica de prioridades a ser usada no kernel do dispositivo
21 my_actuator  = device('actuator', dev_address, dev_index, 0, 1, 'prioFP');
22
23 %define atributos da tarefa
24 task_name       = 'act_task';
25 task_code       = 'actcode';
26 task_desc       = 'actuation';
27 task_priority   = 1;
28 task_deadline   = 100;
29
30 %cria dispositivo atuador
31 my_actuator     = create_sporadical_task( ...
32         my_actuator, task_name, task_code, ...
33         task_desc, task_priority, task_deadline, {}...
34 );
35
36 %prepare actuator device receive task
37 task_name       = 'nw_handler';
38 task_code       = 'msgRcvActuator';
39 task_desc       = 'receive';
40 task_priority   = 2;
41 my_actuator     = create_reception_task( ...
42  my_actuator, task_name, task_code, task_desc, task_priority ...
43 );
```

Algoritmo A.24. Função usada para instanciar e configurar um subsistema

```
1 % Função new_subsystem: funcão utilitária instanciar e configurar um
```

```
2  % dispositivo padrão do ambiente de simulação.
3  %
4  % Entradas:
5  % * my_simulation, instância da simulação atual;
6  % * subsystem_name, nome do subsistema;
7  % * subsystem_index - indice do subsistema
8
9  function my_subsystem = ...
10         new_subsystem(my_simulation, subsystem_name, subsystem_index)
11
12 %captura os indices dos controladores primario e secundário e do atuador
13 my_controller1_device_index = get_device_index_by_name('controller');
14 my_controller2_device_index = get_device_index_by_name('backup-controller');
15 my_actuator_device_index    = get_device_index_by_name('actuator');
16
17 %cria uma instancia do subsistema
18 my_subsystem     = subsystem(subsystem_name, subsystem_index);
19
20 %instancia o controlador primario
21 my_controller1  = new_controller_device( ...
22         subsystem_index, my_controller1_device_index ...
23 );
24
25 %instancia o controlador secundário
26 my_controller2  = new_backup_controller_device( ...
27         subsystem_index, my_controller2_device_index ...
28 );
29
30 %cria o modelo de detecção de falhas
31 pdm = push_detection_model( ...
32         'heartbeat', my_controller2, my_controller1 ...
33 );
34
35 %cria modulo monitor e emissor de heartbeats e associa ao modelo
36 [pdm, my_controller2] = create_monitor(pdm);
37 [pdm, my_controller1] = create_sender(pdm);
38
39 %associa os controladores primario e secundários ao sistema
40 my_subsystem = set_device( ...
41         my_subsystem, my_controller1_device_index, my_controller1 ...
42 );
43 my_subsystem = set_device( ...
44         my_subsystem, my_controller2_device_index, my_controller2 ...
45 );
46
47 %instancia um dispositivo atuador
48 my_actuator  = new_actuator_device( ...
49         subsystem_index, my_actuator_device_index ...
50 );
51 my_subsystem = set_device( ...
52         my_subsystem, my_actuator_device_index, my_actuator ...
53 );
54
```

```
55 %adiciona historicos
56 nhistories = get(my_simulation, 'maxsamples');
57 my_history    = history('heartbeats', nhistories);
58 my_subsystem = add_history(my_subsystem, my_history);
59 my_history    = history('control_actuations',    nhistories);
60 my_subsystem = add_history(my_subsystem, my_history);
61 my_history    = history('control_samples',    nhistories);
62 my_subsystem = add_history(my_subsystem, my_history);
```

A.4.2.3 Funções Utilitárias .

Algoritmo A.25. Código utilizado na inicialização das variáveis globais da simulação

```
1  % Função initialize_global_variables: funcão utilitária para inicia-
2  % * lizar as variáveis globais usadas na simulação.
3  %
4  % Entradas:
5  % * device_name - nome do dispositivo.
6  %
7  % Saídas:
8  % * my_simulation - objeto que representa a simulação atual.
9  % * my_system - objeto que representa um subsistema após a
10 % * inicialização do dispositivo.
11
12 function [my_simulation, my_system] = ...
13          initialize_global_variables(device_name)
14
15 global my_simulation;    %instância da simulação
16 config_simulation;       %configura a simulação
17
18 %se a simulação ainda não foi inicializada, então inicializa
19 if ~isa(my_simulation, 'simulation')
20     %instância uma nova simulação, determinando o nome associado
21     %a simulação e o número de amostradas a serem simuladas.
22      my_simulation    = ...
23          simulation(get_simulation_name, get_simulation_size);
24 end
25
26 subsystem_name  = get_subsystem_name; %captura o nome global
27
28 %captura o indice do sistema pelo nome
29 subsystem_index = get_subsystem_index(get_subsystem_name);
30
31 %verifica se o sistema já foi definido na simulação
32 %se sim cria, senão recupera da instância da simulação
33 if is_undefined_subsystem(my_simulation, subsystem_index)
34   my_system = create_subsystem(subsystem_name, subsystem_index);
35 else
36   my_system = get_subsystem(my_simulation, subsystem_index);
```

```
37 end
38
39 %insere o dispositivo no subsistema
40 my_system = insert_device(my_system, device_name);
41
42 %atribui o subsistema a simulação
43 my_simulation = ...
44         set_subsystem(my_simulation, my_system, subsystem_index);
```

Algoritmo A.26. Código utilizado na definição das configurações da simulação

```
1  % script config_simulation: script usado para definir configurações
2  % globais da simulação.
3
4  global simulation_durarion; %define duração da simulação
5
6  global simulationsize; %define num de mensagens da simulação
7
8  global ndevice; %define o número de dispositivos
9
10 global inttime; %define periodo de heartbeat do detector
11
12 simulation_durarion =      10;
13 simulationsize       =     880;
14 ndevice              =      8;
15 inttime              =  0.0009;
```

Algoritmo A.27. Código utilizado na captura do nome do subsistema na simulação

```
1  % função get_subsystem_name: função que captura o nome do subsistema
2  % no cenário usado na simulação.
3  %
4  % Saída:
5  % * name: o nome do subsistema na simulação
6
7  function name = get_subsystem_name
8
9  %captura o nome do dispositivo no cenário
10 full_path = gcb;
11
12 %remove separadores de contexto de cenário
13 positions = strfind(full_path, '/');
14 lposition = length(positions);
15
16 name = '';
17
18 %se existe separador então remove
19 if(lposition > 1)
20     bindex = positions(1) + 1;
21     eindex = positions(2) - 1;
22
23     name = full_path(bindex: eindex);
24 end
```

Algoritmo A.28. Código utilizado para a captura do índice do subsistema na simulação

```
% função get_subsystem_index: função que captura o indice do
% subsistema no cenário usado na simulação.
%
% Entrada:
% * name: nome do subsistema no cenário. O nome do subsistema deve
% ter um número sequencial que identifique o subsistema no cenário
%
% Saída:
% * index: indice do subsistema na simulação

function index = get_subsystem_index(name)

index = [];

for c = length(name):-1:1
    if isempty(str2num(name(c)))
        break;
    end
    index = strcat(index, name(c));
end

if ~isempty(index)
    index = str2num(index);
else
    index = 1;
end
```

Algoritmo A.29. Código utilizado para instanciar um subsistema da simulação

```
% Função create_subsystem: função que instancia um subsistema no ce-
% nário.
%
% Entrada:
% * subsystem_name: nome do subsistema.
% * subsystem_index: indice do subsistema no cenário.
%
% Saída:
% * my_subsystem : instância de um subsistema

function my_subsystem = ...
        create_subsystem(subsystem_name, subsystem_index)

my_subsystem     = ...
        subsystem(subsystem_name, subsystem_index);
```

Algoritmo A.30. Código utilizado para iniciar a execução das tarefas em um dispositivo

```
% Função start_device: ponhe um dispositivo em funcionamento
%
% Entradas:
% * theSystem – uma instância do subsistema ao qual pertence
% o dispositivo
```

```
6  % * deviceID - identificador do dispositivo no subsistema.
7  %
8  % Saídas:
9  % * A instância do subsistema com o dispositivo em execução
10 % * a instância do dispositivo.
11
12 function [theSystem, my_device] = start_device(theSystem, deviceID)
13
14 %inicializa um dispositivo e ponhe suas tarefas em execucao
15 my_device = get_device(theSystem, deviceID);
16 my_device = init(my_device);
17 my_device = init_tasks(my_device);
```

A.4.2.4 Funções para Criação dos Detectores de Defeitos .

Algoritmo A.31. Código utilizado para instanciar a estratégia de detecção de (BERTIER; MARIN; SENS, 2002)

```
1  % Função create_bertier_detection_strategy: cria uma instância
2  %da estratégia de detecção de (Bertier,2002)
3  %
4  % Entradas:
5  % * interrogation_time - período de emissão de heartbeats
6  % * mi - confiança a ser atribuida a estimativa do erro.
7  % * beta - confiança na estimativa do atraso
8  % * phi - confiança na estimativa da variação do atraso
9  % * window_size - janela de heartbeats a ser usada na ini-
10 %cialização do algoritmo.
11 %
12 % Saídas:
13 % * A instância da estrategia de detecção de (Bertier, 2002).
14
15 function the_strategy = create_bertier_detection_strategy( ...
16         interrogation_time, mi, beta, phi, window_size ...
17 )
18
19 %prepare the bertier parameters
20 A_history = history('A_history', window_size);
21
22 %create the bertier strategy
23 the_strategy  = bertier_detection_strategy( ...
24         interrogation_time, mi, beta, phi, window_size, A_history ...
25 );
```

Algoritmo A.32. Código utilizado para instanciar a estratégia de detecção de (JACOBSON, 1988)

```
1  % Função create_jacobson_detection_strategy: cria uma instância
2  %da estratégia de detecção de (Jacobson,1988)
```

```
3  %
4  % Entradas:
5  % * interrogation_time - período de emissão de heartbeats
6  % * mi - confiança a ser atribuida a estimativa do erro.
7  % * beta - confiança na estimativa do atraso
8  % * phi - confiança na estimativa da variação do atraso
9  %
10 % Saídas:
11 % * A instância da estrategia de detecção de (Jacobson, 1988).
12
13 function the_strategy = create_jacobson_detection_strategy(...
14         interrogation_time, gamma, beta, phi ...
15 )
16
17 %prepare the jacobson parameters
18 gamma = 0.1; beta = 1.0; phi = 2.0;
19
20 %create the jacobson strategy
21 the_strategy = jacobson_detection_strategy( ...
22         gamma, beta, phi, interrogation_time ...
23 );
```

Algoritmo A.33. Código utilizado para instanciar a estratégia de detecção baseada em *RNA*

```
1  % Função create_jacobson_detection_strategy: cria uma instância
2  %da estratégia de detecção baseada em RNA
3  %
4  % Entradas:
5  % * interrogation_time - período de emissão de heartbeats
6  % * queue_size - fila de heartbeats a serem utilizados.
7  %
8  % Saídas:
9  % * A instância da estrategia de detecção baseada em RNA
10
11 function the_strategy = create_neural_network_strategy( ...
12         interrogation_time, queue_size, varargin ...
13 )
14 input_len = queue_size -1;
15
16 %preapre the rprop neural network algorithm parameters
17 %A1) the structural parameters
18 input_rangers = zeros(input_len, 2);
19
20 mmin = -1; mmax = 1;
21 if length(varargin) > 0
22     mmin = cell2mat(varargin(1));
23 end
24 if length(varargin) > 1
25     mmax = cell2mat(varargin(2));
26 end
27 mapper = [];
28 if length(varargin) > 2
29         mapper = varargin{3};
```

```
30 end
31 minimum_gradient = 0;
32 if length(varargin) > 3
33         minimum_gradient = cell2mat(varargin(4));
34 end
35
36 %normaliza os valores máximo e minimo
37 [mapper, mmin] = run(mapper, mmin);
38 [mapper, mmax] = run(mapper, mmax);
39
40 for (k=1:input_len),
41     input_rangers(k, 1) = mmin;
42     input_rangers(k, 2) = mmax;
43 end
44
45 layers_sizes  = [30 10 2];
46
47 %A2) the functional parameters
48 act_functions = {};
49 for a = 1: length(layers_sizes)
50     act_functions{a} = 'tansig';
51 end
52
53 %A3) the training parameters
54 training_algorithm          = 'trainrp';
55 training_epochs             = 100000;
56 training_learning_rate      = 0.1;
57 training_momentum           = 0.8;
58 training_minimum_gradient = minimum_gradient;
59 training_goal               = 0.0;
60 training_delta_increment    = 1.2;
61 training_delta_decrement    = 0.5;
62 training_initial_delta      = 0.08;
63 training_maximum_delta      = 50.0;
64
65 %prepare and create the artificial neural network strategy
66 %with the resilient propagation training algorithm (rprop)
67 %B1) create the neural net algorithm
68
69 the_rprop_algorithm = ...
70         rprop_neural_net_algorithm( ...
71                 'input_rangers'             , input_rangers , ...
72                 'layers_sizes'              , layers_sizes , ...
73                 'activation_functions'      , act_functions , ...
74                 'training_algorithm'        , training_algorithm , ...
75                 'training_epochs'           , training_epochs , ...
76                 'training_learning_rate'    , training_learning_rate , ...
77                 'training_momentum'         , training_momentum , ...
78                 'training_minimum_gradient', training_minimum_gradient , ...
79                 'training_goal'             , training_goal , ...
80                 'training_delta_increment'  , training_delta_increment , ...
81                 'training_delta_decrement'  , training_delta_decrement , ...
82                 'training_initial_delta'    , training_initial_delta , ...
```

```
83                  'training_maximum_delta'    , training_maximum_delta , ...
84                  'pattern', [], ...
85                  'mapper', mapper ...
86          );
87
88
89  %B2) create the neural strategy
90  the_strategy = neural_net_detection_strategy( ...
91          interrogation_time , the_rprop_algorithm , queue(queue_size) ...
92  );
```

A.4.2.5 Função para Cálculo do Desempenho do Sistema de Controle .

Algoritmo A.34. Código utilizado para o cálculo do desempenho do controle realizado por um grupo de subsistemas seguindo o conceito de *margem de jitter*

```
1  % Função calcQoPOfSubsystems: função calcula o indice de desempenho
2  % de cada um dos subsistemas de controle envolvidos em uma simulação
3  %
4  % Entrada:
5  % * rfile: arquivo que contém uma instancia da simulação realizada.
6  % * cPlant: o modelo da planta utilizada pelos subsistemas.
7  % * cController: o modelo do controlador continuo utilizado por cada
8  % subsistema.
9  % * taxa de amostragem que deve ser usada na discretização do controlador
10 %
11 % Saída:
12 % * QoPs: vetor com os indices de desempenho de cada um dos subsistemas de
13 % controle
14 % * apMargins: Margem de fase aparente de cada sistema de controle, observada
15 % durante o calculo do índice de desempenho.
16 % * pMargins: Margem de fase de cada sistema de controle, observada
17 % durante o calculo do índice de desempenho.
18 % * jitters: Jitter observado por cada subsistema durante a simulação
19 % * mJitters: Margem de Jitter calculada para cada um dos subsistemas
20 % * delayMargins: Margem de atraso calculada para cada um dos subsistemas
21
22 function [QoPs, apMargins, pMargins, jitters, mJitters, delayMargins] = ...
23         calcQoPOfSubsystems(rfile, cPlant, cController, h)
24
25 %carrega o arquivo com os dados da simulação
26 load(rfile, 'my_simulation');
27
28 %Verifica o número de subsistemas do cenário simulado;
29 n_systems = get(my_simulation, 'subsystems_count');
30
31 %inicializa vetores de retorno
32 QoPs            = zeros(1, n_systems);
33 apMargins       = zeros(1, n_systems);
34 pMargins        = zeros(1, n_systems);
35 jitters         = zeros(1, n_systems);
```

```
36 mJitters          = zeros(1, n_systems);
37 delayMargins      = zeros(1, n_systems);
38
39 %para cada subsistema procede com o calculo do indice de desempenho
40 for i_system = 1:n_systems
41
42         %captura o subsistema
43         my_system = get_subsystem(my_simulation, i_system);
44
45     %captura o historico dos estados das variaveis do subsistema.
46         %apenas variáveis relacionadas a atividade de controle.
47     ctrl_history     = get_history(my_system, 'control_samples');
48     sensor_history   = get_history(my_system, 'sensor_aquisition');
49     actd_history     = get_history(my_system, 'control_actuations');
50
51     %captura os valores das amostras contidas nos historicos
52     sensor_values = get(sensor_history, 'history');
53     ctrl_values = get(ctrl_history, 'history');
54     actd_values = get(actd_history, 'history');
55
56     %transforma as amostras em valores númericos
57     sensor_vector = [];
58     for i_sensor = 2 : length(sensor_values)
59         if(isempty(sensor_values{i_sensor}))
60             break;
61         end
62         sensor_vector(i_sensor-1) = sensor_values{i_sensor};
63     end
64
65     actd_vector = [];
66
67     for i_actd = 2 : length(actd_values)
68         if(isempty(actd_values{i_actd}))
69             break;
70         end
71         actd_vector(i_actd - 1) = actd_values{i_actd};
72     end
73
74     %now, we create the actd delay vector
75     actd_count = length(actd_vector);
76
77     sensor_count = length(sensor_vector);
78
79     smpls_size = min([actd_count sensor_count]);
80
81         %calcula os atrasos nos instantes de atuação
82     actd_delays = actd_vector(1:smpls_size) - sensor_vector(1:smpls_size);
83
84     jitters(i_system) = -1;
85
86         %se algum atraso pode ser calculado então procede
87         %com o calculo do desemepenho.
88     if(~isempty(actd_delays))
```

```
89          %calcula o jitter de comunicação e computação observado pelo subsitema
90          jitters(i_system) = max(actd_delays);
91
92                  %calcula a margem de fase aparente
93          [...
94                    delayMargins(i_system), ...
95                    apMargins(i_system), ...
96                    pMargins(i_system), ...
97                    mJitters(i_system)...
98                   ] = calcPisA(cPlant, cController, h, jitters(i_system), 0);
99
100                 %calcula o indice de desempenho para o sistema em questão.
101         QoPs(i_system) = apMargins(i_system) / pMargins(i_system);
102     end
103 end
```

REFERÊNCIAS BIBLIOGRÁFICAS

AGUILERA, M. K.; CHEN, W.; TOUEG, S. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In: MAVRONICOLAS, M.; TSIGAS, P. (Ed.). *Proceedings of the 11th International Workshop on Distributed Algorithms, WDAG'97 (Saarbrücken, Germany, September 24-26, 1997)*. Berlin-Heidelberg-New York-Barcelona-Budapest-Hong Kong-London-Milan-Paris-Santa Clara-Singapore-Tokyo: Springer-Verlag, 1997, (LNCS, v. 1320). p. 126–140. Disponível em: <http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=1320&spage=126>.

ALMEIDA, C.; Veríssimo, P.; CASIMIRO, A. *The quasi-synchronous approach to fault-tolerant and real-time communication and processing*. Lisboa, Portugal, jul 1998.

ANDRADE, S.; MACêDO, R. A component-based real-time architecture for distributed supervision and control applications. In: *10th IEEE International Conference on Emerging Technologies and Factory Automation*. Catania, Italy: IEEE/ETFA2005, 2005. I, p. 19–22.

ÅSTRÖM, K. J. Limitations on control system performance. *European Journal of Control*, v. 6, n. 1, p. 2–20, jan 2000.

AUDSLEY, A. N. et al. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, p. 284–292, jul 1993.

BERTIER, M.; MARIN, O.; SENS, P. Implementation and performance evaluation of an adaptable failure detector. In: IEEE COMPUTER SOCIETY. *Proceedings Of The International Conference On Dependable Systems And Networks*. Washington, Dc, 2002. p. 354–363. ISBN 0-7695-1597-5. Disponível em: <http://portal.acm.org/citation.cfm?id=738261>.

BERTIER, M.; MARIN, O.; SENS, P. Performance analysis of hierarchical failure detector. In: *Proceedings Of The International Conference On Dependable Systems And Networks*. San-francisco,Usa: IEEE Society Press, 2003. p. 635–644. ISBN 0-7695-1952-0. Disponível em: <http://citeseer.ist.psu.edu/674456.html>.

BIRKHOFF, G. D. *Dynamical Systems*. Rhode Island: American Mathematical Society., 1927. 305 p. ISBN 0-8218-3394-4.

BITTENCOURT, G. *Inteligência Artificial: Ferramentas e Teorias*. 2. ed. Florianópolis: Editora da UFSC, 2001. 362 p.

BUTTAZZO, G. et al. Managing quality-of-control performance under overload conditions. In: *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*. Washington, DC, USA: IEEE Computer Society, 2004. p. 53–60. ISBN 0-7695-2176-2.

CERQUEIRA, J. J. F. *Identificação de Sistemas Dinâmicos Usando Redes Neurais Artificiais: Uma Aplicação a Manipuladores Robóticos*. Tese — Departamento de Sistemas e Controle da Energia, Laboratório de Sistemas Modulares Robóticos, Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, São Paulo: Campinas, 2003.

CERVIN, A. *Merging Real-Time and Control Theory for Improving the Performance of Embedded Control Systems*. Pavia, Italy, Set 2004.

CERVIN, A. et al. How does control timing affect performance? *IEEE Control Systems Magazine*, n. 3, p. 16–30, Jun 2003.

CERVIN, A. et al. The jitter margin and its application in the design of real-time control systems. In: *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA)*. Göteborg, Sweden: [s.n.], 2004.

CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. *Journal Of The ACM*, v. 43, n. 2, p. 225–267, mar. 1996.

CHEN, W. *On Quality Of Service Of Failure Detectors*. PHD — Faculty of the Graduate School of Cornel University, 2000.

CHEN, W.; TOUEG, S.; AGUILERA, M. K. On the quality of service of failure detectores. *IEEE Transactions On Computer*, v. 51, n. 2, p. 561–580, 2002.

CHILLAREGE, R. What is software failure? *IEEE Transactions On Reliability*, v. 45, n. 3, Set 1986.

COLOM, P. M. *Analysis and Design of Real-Time Control Systems with Varying Control Timing Constraints*. Programa De Doctorat: Enginyeria En Informàtica Industrial / Tecnologies Avançades De La Producció — Département D'Enginyeria De Sistemes, Automàtica I Informàtica Industrial, Barcelona, jun 2002.

CRISTIAN, F. Understanding fault-tolerant distributed systems. *ACM*, v. 34, n. 2, p. 56–78, Feb 1991.

CRISTIAN, F.; FETZER, C. The timed asynchronous distributed system model. *IEEE Transactions On Paralell and Distributed Systems*, v. 10, n. 6, p. 603–638, Jun 1999.

DEMUTH, H.; BEALE, M. *Neural Network Toolbox: For Use with Matlab*. 5th. ed. Nantick, USA, jan 1998.

DESWARTE, Y.; KANOUN, K.; LAPRIE, J. Diversity against accidental and deliberate faults. *IEEE Computer Security, Dependability And Assurance: From Needs To Solutions*, p. 171–181, Jul 1998.

ELDER, M. C. *Fault Tolerance In Critical Information System*. Dissertação (Mestrado em Ciência da Computação) — Faculdade da Escola de Engenharia E Ciência Aplicada da Universidade de Virginia, 2001.

ELKS, C. R.; DUGAN, J. B.; JOHNSON, B. W. Reliability analysis of hard real-time systems in the presence of controller malfunctions. In: *Proceedings of the XVIII Reliability and Maintainability Symposium*. Los Angeles, CA: IEEE Computer Society Press, 2000. p. 58–64.

FARINES, J.; FRAGA, J. S.; OLIVEIRA, R. S. *Sistema de Tempo-Real*. 1st. ed. Santa Catarina: Departamento de Automação E Sistemas da Universidade Federal de Santa Catarina, 2000.

FELBER, P. *The Corba Object Group Service : A Service Approach to Object Groups in CORBA*. Tese de Doutorado em Informática — Département D'Informatique, École Polytechnique Fédérale De Lausanne, 1998.

FISCHER, M. J.; LYNCH, N. A.; PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, v. 32, n. 2, p. 374–382, apr 1985.

FREITAS, A. E. S. *Um modelo de Classificação de Excessões Baseado em Redes Kohonen Ajustadas por Algoritmos Genéticos*. Dissertação (Mestrado em Engenharia Eletrica) — Laboratório de Sistemas Inteligentes, Departamento de Engenharia Elétrica, Escola Politécnica, Universidade Federal da Bahia, 2004.

GAMBIER, A. Real-time control systems: A Tutorial. In: *Proceedings of the V Asian Control Conference 2004*. Melbourne, Australien: [s.n.], 2004. p. 1023–1030.

GÄRTNER, F. C. Fundamentals of fault tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, v. 31, n. 1, p. 1–26, Mar 1999.

GLASER, M.; KORDECKI, C.; REMBOLD, U. A concept for distributed control systems. In: *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*. Hawaii, USA: IEEE, 1989. v. 2, p. 667–672. ISBN 0-8186-1912-0.

GOODWIN, G. C.; GRAEBE, S. F.; SALGADO, M. E. *Control System Design*. Valparaíso: [s.n.], 2000.

GORENDER, S.; MâCEDO, R. J.; RAYNAL, M. A hybrid and adaptative model for fault-toleran distributed computing. In: *Proceedings of the International Conference on Dependable Systems and Networks*. Yokohama, Japan: IEEE Computer Society, 2005. p. 412–421. ISBN 0-7695-2282-3.

GUERRAOUI, R.; SCHIPER, A. Consensus: the big misunderstanding. In: *Proceedings of the 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-6)*. Tunis, Tunisia: IEEE Computer Society Press, 1997. p. 183–188. Disponível em: <http://lsewww.epfl.ch/Publications/ById/73.html>.

HAYKIN, S. *Neural Networks; A Comprehensive Foundation*. 1st. ed. New York: MACMillan, 1994.

HECHT-NIELSEN, R. Theory of the backpropagation neural network. In: *Proceedings of the International Joint Conference on Neural Networks*. [S.l.: s.n.], 1989. v. 1, p. 593–606.

HENRIKSSON, D.; CERVIN, A. *Truetime 1.13 - Reference Manual*. [S.l.], Oct 2003.

HENRIKSSON, D.; CERVIN, A.; ARZEN, K.-E. Truetime: Simulation of control loops under shared computer resources. In: *Proceedings of the 17th IFAC World Congress on Automatic Control*. Barcelona, Spain: [s.n.], 2002. p. 1–7.

HERMANT, J.; LANN, L. Fast asynchronous uniform consensus in real-time distributed systems. *IEEETC: IEEE Transactions on Computers*, v. 51, 2002.

HILMER, H.; KOCHS, H.-D.; DITTMAR, E. A fault-tolerant communication architecture for real-time control systems. In: *IEEE International Workshop on Factory Communication Systems*. Barcelona, Spain: [s.n.], 1997. p. 111–118. ISBN 0-7803-4182-1.

HSU, H. P. *Teoria e Problemas de Sinais e Sistemas*. Porto Alegre, RS: Bookman, 2004. ISBN 85-363-0360-3.

IRIE, B.; MIYAKE, S. Capabilities of three–layered perceptrons. In: *Proceedings of the IEEE International Conference on Neural Networks*. San Diego, CA: IEEE, 1988. v. 1, p. 641–648.

JACOBSON, V. Congestion avoidance and control. *ACM Computer Communication Review; Proceedings Of The Sigcomm '88 Symposium In Stanford, Ca, August, 1988*, v. 18, 4, p. 314–329, 1988.

JAIN, A. K.; MAO, J.; MOHIUDDIN, K. M. Artificial neural networks: A tutorial. *IEEE Computer*, v. 29, n. 3, p. 31–44, 1996.

JALOTE, P. *Fault Tolerance In Distributed Systems*. New Jersey: Prentice Hall, 1994.

JANG, J.-S. R.; SUN, C.-T.; MIZUTANI, E. *Neural-Fuzzy and Soft Computing:A Computational Approach to Learning and Machine Intelligent*. Upper Saddle River, NJ: Prentice Hall, 1997.

JI, K.; KIM, W.-J. Real-time control of networked control systems via ethernet. *International Journal of Control, Automation, and Systems*, v. 3, n. 4, p. 591–600, dec 1997.

JOSEPH, M.; PANDYA, P. K. Finding response times in a real-time system. *Computer Journal*, v. 29, n. 5, p. 390–395, 1986.

KAO, C.-Y.; LINCOLN, B. *Simple Stability Criteria for Systems with Time-Varying Delays*. mar.18 2004.

KIM, H.; SHIN, K. G. On the maximum feedback delay in a linear/nonlinear control system with input disturbances caused by controller-computer failures. *IEEE Transactions on Control Systems Technology*, v. 2, n. 2, p. 110–122, 1994.

KOPETZ, H. *Real-Time Systems: Design Principles For Distributed Embebed Applications*. [S.l.]: Kluwer Academic Publishers, 1997.

LAMPORT, L.; LYNCH, N. *Chapter on Distributed Computing: Methods and Models*. fev. 3 1989. Disponível em: <http://research.microsoft.com/users/lamport/pubs/lamport-chapter.pdf>.

LANN, G. L.; SCHMID, U. *How to Maximize Computing Systems Coverage*. [S.l.], April 2003. 26 pages p. Disponível em: <http://www.ecs.tuwien.ac.at/projects/Theta/papers/>.

LAPRIE, J.-C. Dependable computing and fault tolerance: concepts and terminology. *XXV Symposium International On Faulting-tolerant Computing*, p. 2–11, 1985.

LIAN, F.; MOYNE, J. R.; TILBURY, D. M. Performance evaluation of control networks: ethernet, controlnet, and devicenet. *IEEE Control Systems Magazine*, v. 21, p. 66–93, Fev 2001.

LIAN, F.; MOYNE, J. R.; TILBURY, D. M. Network design consideration for distributed control systems. *IEEE Transactions On Control Systems Technology*, v. 10, p. 297–307, Mar 2002.

LINCOLN, B. A simple stability criterion for control systems with varying delays. In: *15th IFAC World Congress*. [S.l.: s.n.], 2002. p. 2354–2359.

LINCOLN, B. *Dynamic Programming and Time-Varying Delay Systems*. PHD — Department of Automatic Control, Lund Institute of Technology, Lund, 2003.

LYNCH, N. A. *Distributed Algorithms*. San Francisco, California: Morgan Kaufmann, 1996.

MACÊDO, R. J. A.; LIMA, F. Improving the quality of service of failure detectors. *Simpósio Brasileiro de Redes de Computadores*, 2004.

MACÊDO, R. J. A. et al. Tratando a previsibilidade em sistemas de tempo-real distribuídos: especificação. linguagens, middleware e mecanismos básicos. *22º Simpósio Brasileiro de Redes de Computadores*, p. 105–163, 2004.

MACKAY, S. *Practical Industrial Data Networks: Design, Installation and Troubleshooting*. [S.l.]: Morgan Kaufmann Publishers, 2004. 421 p. ISBN 0-7506-5807-X.

MARTí, P. et al. A java-based framework for distributed supervision and control of industrial processes. *7th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA99)*, v. 1, p. 33–41, 1999.

MARTí, P. et al. Improving quality-of-control using flexible timing constraints: Metric and scheduling issues. In: *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*. Washington, DC, USA: IEEE Computer Society, 2002. p. 91. ISBN 0-7695-1851-6.

MARTí, P. et al. Jitter compensation for real-time control systems. In: *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*. Washington, DC, USA: IEEE Computer Society, 2001. p. 39. ISBN 0-7695-1420-0.

NUNES, R. C.; JANSCH-PôRTO, I. Qos of timeout-based self-tuned failure detectors: the effects of the communication delay predictor and the safety margin. In: *International Conference On Dependable Systems And Networks*. [S.l.: s.n.], 2004.

NYQUIST, H. Regeneration theory. *Bell System Techical Journal*, Bells Lab., v. 11, p. 126–147, jan 1932.

OGATA, K. *Modern Control engineering*. 2nd. ed. Englewood Cliffs: Prentice Hall, 1990. 778–784 p.

OGATA, K. *Discrete-Time Control Systems*. 2nd. ed. Upper Saddle River, NJ 07458, USA: Prentice-Hall, 1995. ISBN 0-13-034281-5.

OTANEZ, P. G. et al. The implications of ethernet as a control network. In: *Global Powertrain Conference*. Ann Arbor, Michigan: [s.n.], 2002. p. 1–9.

PAGANINI, F. *Robust H2 Analysis for Continuos Time Systems*. [S.l.], 1996 1996.

PINKUS, A. *Approximation theory of the MLP model in neural networks*. 1999. Disponível em: <citeseer.ist.psu.edu/pinkus99approximation.html>.

PIURI, V. Design of fault-tolerant distributed control systems. *IEEE Transactions On Instrumentation and Measurement*, v. 43, n. 2, p. 257–264, apr 1994.

RIEDMILLER, M.; BRAUN, H. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In: *Proceedings of 1993 IEEE International Conference on Neural Networks*. San Francisco, California: IEEE/INNS, 1993. v. 1, p. 586–591. U. Karlsruhe.

Sá, A. S. de; MACêDO, R. J. A. Detectores de defeitos adaptáveis para sistemas de controle distribuídos de tempo-real sobre redes ethernet. In: *Proceedings of VII Brazilian Workshop of Real-Time Systems*. Fortaleza, Brazil: SBRC/WTR2005, 2005. p. 65–68.

SAMPAIO, L. M. R. et al. How bad are wrong suspicions ? Towards adaptive distributed protocols. In: *Proceedings of 2003 International Conference on Dependable Systems and Networks*. [S.l.]: IEEE Computer Society, 2003.

SANTOS, R. T. et al. Extração de regras de redes neurais via algoritmos genéticos. In: SãO JOSé DOS CAMPS - SP, BRAZIL. *Proceedings IV Brazilian Conference on Neural Networks*. 1999. p. 158–163. Disponível em: <http://www.cs.kent.ac.uk/pubs/1999/1435>.

SCHIFFMANN, W.; JOOST, M.; WERNER, R. *Optimization of the Back-propagation Algorithm for Training Multilayer Perceptrons*. nov 1994. Disponível em: <http://citeseer.ist.psu.edu/12456.html; http://www.uni-koblenz.de/ schiff/TR16.92.ps.gz>.

SCHNEIDER, S.; PARDO-CASTOLLOTE, G.; HAMILTON, M. Can be ethernet real-time? Real-Time Innovations, p. 1–21, jun 1999. Disponível em: <http://www.rti.com, acessado em: 19 de maio de 2005>.

SETO, D.; LEHOCZKY, J. P.; SHIN, K. G. Trade-off analysis of real-time control performance and schedulability. In: *Real-Time Systems*. [S.l.]: Kluwer Academic Publishers, 2001. v. 21, p. 199–217.

SIMON, D. Analyzing control system robustness. *IEEE Potentials*, v. 21, n. 1, p. 16–19, mar 2002. ISSN 0278-6648.

SONG, Y.; KOUBAA, A.; SIMONOT, F. Shwitched ethernet for real-time industrial communication: Modeling and message buffering delay evaluation. In: *Proceedings of the IV IEEE International Workshop on Factory Communication Systems*. Västera, Sweden: IEEE, 2002. p. 27–35.

STALLINGS, W. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. 3. ed. Massachusetts: Addison-Wesley, 1999. 619 p. ISBN 0-201-48534-6.

TANENBAUM, A. S. *Computer Networks*. 4th. ed. Upper Saddle River, NJ 07458, USA: Prentice-Hall, 2003. ISBN 0-13-066102-3.

The Mathworks. *Matlab:The Language of Technical Computing*. Nantick, USA, jul 2002.

The Mathworks. *Simulink Reference:Simulation and Model-Based Design*. Nantick, USA, oct 2004.

The Mathworks. *Getting Started with Simulink*. Nantick, USA, mar 2006.

TIKK, D.; KÓCZY, L. T.; GEDEON, T. D. A survey on universal approximation and its limits in soft computing techniques. *Int. J. Approx. Reasoning*, v. 33, n. 2, p. 185–202, 2003. Disponível em: <http://dx.doi.org/10.1016/S0888-613X(03)00021-5>.

TINDELL, K.; BURNS, A.; WELLINGS, A. *Calculating Controller Area Network (CAN) Message Response Times*. aug 1995. 1163–1169 p.

TINDELL, K.; HANSSMON, H.; WELLINGS, A. J. Analysing real-time communications: Controller area network (CAN). In: *IEEE Real-Time Systems Symposium*. [S.l.]: IEEE Computer Society, 1994. p. 259–263. ISBN 0-8186-6600-5.

TÖRNGREN, M. Fundamentals of implementing real-time control applications in distributed computer systems. *Real-Time Systems*, v. 14, n. 3, p. 219–250, 1998.

VERÍSSIMO, P.; RODRIGUES, L. *Distributed Systems For Systems Architects*. Usa: Kluwer Academic Publishers, 2000. 623 p.

WITTENMARK, B.; TÖRNGREN, M. *Timing Problems in Real-Time Control Systems: Problem Formulation*. [S.l.], may 1994.

YOOK, J. K.; TILBURY, D. M.; SOPARKART, N. R. A design methodology for distributed control systems to optimize performance in the presence of time delays. In: *Proceedings of the American Control Conference*. Chicago, Lllinois: [s.n.], 2000. p. 1959–1964.