Um Framework para Prototipagem e Simulação de Detectores de Defeitos na Construção de Sistemas de Tempo Real

Alirio Santos de Sá, Raimundo José de Araújo Macêdo

Laboratório de Sistemas Distribuídos (LaSiD)
Programa de Pós-Graduação em Mecatrônica
Universidade Federal da Bahia,
Campus de Ondina, 40170-110, Salvador-BA, Brasil
{aliriosa, macedo}@ufba.br

Abstract. *Modern Supervision and Control Systems (S&C) are characterized by the use of the Commercial Off-The-Shelf Components (COTS Components), such as computer networks and operating systems. Though such COTS components can optimize costs and reuse, they compromise dependability. Thus, evaluation of dependability mechanisms for such environments has been a strong focus of research. In this paper is described a framework for rapid prototyping and testing of failure detectors for S&C systems. This framework not only allows to implement and simulate existing detection strategies, but it also allows for the construction of new strategies, or even the composition of new strategies from existing ones, in order to reach the desired failure detection quality of service in the simulated environment.*

Resumo. *Sistemas modernos de controle e supervisão (S&C, Supervision and Control Systems) são caracterizados pela utilização de componentes de prateleira (COTS, Commercial off-the-shelf), como redes e sistemas operacionais, os quais otimizam custo e reutilização, mas podem comprometer a confiabilidade. Assim, verificar mecanismos adequados de confiabilidade para esses ambientes tem sido alvo de estudo. Neste artigo, é descrito um framework para prototipagem e teste de detectores de defeitos sobre S&C, que permite não só implementar e simular diferentes estratégias de detecção existentes na literatura, mas também propor novas estratégias, ou mesmo conceber composições de estratégias, de modo a atingir a qualidade de serviço de detecção desejada no ambiente simulado.*

1. Introdução

Controle e supervisão são mecanismos utilizados na indústria para não só otimizar a utilização e produção das plantas[1] ativas, mas também incrementar a qualidade e a confiabilidade dos produtos fabricados. 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 robustas 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 e tolerância a falhas.

[1]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 forno, um robô etc.

Acomodar todos os requisitos demandados pelos sistemas industriais modernos é um desafio que tem motivado fabricantes e acadêmicos na busca de novas soluções. Nesse sentido, a indústria moderna e a academia têm se valido da utilização de componentes de prateleira, como redes e sistemas operacionais, para implementar e propor soluções que: 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; diminuem os custos operacionais; e possibilitam a integração entre os diferentes níveis[2] que compõem o processo de manufatura [Lian et al. 2001].

Apesar de todos os benefícios apontados, novos desafios são encontrados quando se realiza a construção de sistemas modernos de controle e supervisão usando *COTS* – o sistema está sujeito a incertezas relacionadas às variações no tempo de computação dos algoritmos (atraso de computação) e transmissão das mensagens pelo subsistema de comunicação (atraso de comunicação). Esses aspectos podem comprometer a confiança no funcionamento, fazendo com que tais sistemas apresentem um comportamento imprevisível e operem em regimes não confiáveis. O problema se torna ainda mais grave em cenários em que aplicações críticas precisam ser suportadas. As aplicações de tempo real críticas necessitam de mecanismos de tolerância a falhas para garantir a correção na execução das operações e permitir que modos de funcionamento previsíveis sejam alcançados mesmo após a ocorrência de falhas de alguns dos componentes do sistema [Jalote 1994].

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 tornam o projeto dos sistemas de controle e supervisão ainda mais difícil, visto que os aumentos do tráfego na rede e do processamento podem implicar, em alguns casos, em atrasos não determinísticos quando se utiliza componentes de prateleira. Para a implementação de mecanismos de tolerância a falhas, um serviço de detecção de defeitos é fundamental, seja para ativar procedimentos de recuperação, seja para permitir a reconfiguração do sistema [Jalote 1994]. Para aplicações de tempo real críticas 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 com tráfego na rede ou tempo de computação variados, os detectores adaptativos de defeitos podem produzir informações mais precisas sobre o estado dos dispositivos do sistema, evitando que os mecanismos de tolerância a falhas tomem decisões que venham a prejudicar o desempenho do sistema. Informações errôneas produzidas pelos detectores de defeitos podem iniciar procedimentos de recuperação ou de reconfiguração, os quais consomem recursos computacionais e podem implicar em regimes de operação indesejáveis para o sistema de controle ou de supervisão. Assim, para a implementação de sistemas de tempo real confiáveis, usando componentes de prateleira, é importante que, durante o projeto do sistema, o projetista possa avaliar diferentes implementações de mecanismos de tolerância a falhas e diferentes estratégias de detecção adaptativa de defeitos, de modo que possa, a priori, não só verificar o impacto da implementação das diferentes abordagens de mecanismos de tolerância a falhas na construção do sistema de controle, mas também observar a qualidade de serviço de diferentes estratégias de detecção de defeitos, podendo assim utilizar a estratégia que possua um nível de qualidade de serviço (*QoS*) de detecção conveniente para o bom fun-

[2] Aquisição, controle e supervisão.

cionamento da aplicação. Portanto, o desenvolvimento de ferramentas e implementações que facilitem a prototipagem, simulação e análise de sistemas de controle e supervisão confiáveis são extremamente importantes para tornar o projeto do sistema mais confiável e minimizar a ocorrência de falhas durante o funcionamento do sistema por conta de situações que não foram previstas antecipadamente durante o projeto.

Nesse contexto, o presente artigo apresenta um *framework* para prototipagem e teste de detectores de defeitos sobre sistemas distribuídos de controle e supervisão. O *framework* proposto permite implementar e testar através de simulações diferentes estratégias de detecção existentes na literatura. Além disso, com as facilidades disponibilizadas pelo *framework*, pode-se propor novas estratégias, ou mesmo conceber estratégias que representem composições de estratégias existentes, de modo a atingir os resultados desejados de qualidade de serviço de detecção no ambiente simulado. O *framework* foi desenvolvido sobre o ambiente *Matlab/Simulink* que é amplamente utilizado por projetistas de sistemas dinâmicos das mais diversas áreas de conhecimento, como tempo real, controle e supervisão, visualização e processamento de imagens, inteligência artificial, entre outros. Desse modo, a implementação permite ainda utilizar as facilidades disponíveis no *Matlab/Simulink* de maneira que o processo de prototipagem, simulação e análise possa cobrir o maior número de cenários possíveis.

Este artigo está organizado da forma a seguir. Na seção 2 é feita uma breve apresentação de alguns trabalhos correlatos. Na seção 3 são discutidos aspectos relacionados a adaptabilidade de detectores de defeitos. A seção 4 descreve o ambiente *Matlab/Simulink*. Na seção 5 é descrito o *framework* proposto e, por fim, na seção 6 são apresentadas as considerações finais e sugestões para trabalhos futuros.

2. Trabalhos Relacionados

Na busca por soluções eficientes para a implementação de sistemas modernos de controle e supervisão, diversos trabalhos como [Wittenmark and 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 and Macêdo 2005] podem ser encontradas discussões focadas no desenvolvimento de uma arquitetura confiável para o desenvolvimento de *S&C*. Questões como confiança no funcionamento e tolerância a falhas são abordadas em trabalhos como [Piuri 1994], [Kim and Shin 1994], [Elks et al. 2000] e [Sá and Macêdo 2005]. Além disso, técnicas de projeto e arquiteturas de suporte têm sido sugeridas. Em paralelo, ferramentas e mecanismos de apoio ao projeto também têm sido estudados, por exemplo: em [Lincoln and Cervin 2002] é proposta a ferramenta *Jitterbug*, que dá suporte, durante o projeto, à análise de desempenho de sistemas de controle sobre variações temporais; em [Henriksson and Cervin 2003] é discutido o *TrueTime*, uma ferramenta para o projeto de sistemas de *S&C* de tempo real distribuídos. Ambas as ferramentas são construídas usando o ambiente de simulação *Matlab/Simulink*, todavia nenhuma dessas ferramentas possui suporte direto a implementação de *S&C* confiáveis. Em [Branicky et al. 2003] é encontrada uma revisão mais aprofundada sobre diversos *frameworks* usados durante o projeto para prototipagem, análise e simulação aplicáveis aos sistemas de controle sobre rede. Entretanto, nenhum dos *frameworks* discutidos pelos autores abordam aspectos relacionados à implementação de mecanismos de tolerância a falhas para sistemas de *S&C* de tempo real confiáveis.

3. Detectores 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 remoto do estado do componente pode ser feito usando dois modelos básicos: *Pull* e *Push* [Felber 1998]. No modelo *Pull*, uma vez recebida uma mensagem de *Are you Alive?* do componente monitor do detector de defeitos, o componente monitorado deve responder com seu atual estado (mensagem de *I am alive!* ou *heartbeat*). A cada mensagem de monitoramento enviada, o monitor 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. No modelo *Push*, o componente monitorado espontaneamente envia o seu estado atual. Baseado no intervalo entre chegada das mensagens, o componente monitor do detector de defeitos 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 número de mensagens de detecção trocadas entre o componente monitorado e detector de defeitos é menor e, portanto, consome menos recursos do canal de comunicação.

3.1. Adaptabilidade em Detecção de Defeitos

As abordagens de detecção de defeitos apresentadas nesta seção utilizam o modelo de monitoramento *Push*, baseado em mensagens de *heartbeats* [Aguilera et al. 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, sequencialmente assinalada e denotada por *heartbeat* (m^{hb}), 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. Todavia, para simplificar a notação usada, a referência ao tempo global será omitida.

A cada *heartbeat* assinalado por k recebido (m_k^{hb}), q calcula o intervalo de tempo (Δ_{k+1}^{to}) necessário para a chegada do próximo *heartbeat* (m_{k+1}^{hb}). Se m_{k+1}^{hb} não chega dentro de Δ_{k+1}^{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. Para quaisquer dois instantes consecutivos σ_k e σ_{k+1}, tem-se: $\sigma_{k+1} - \sigma_k = \Delta^i$. 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.

Em um ambiente onde não há variação no atraso e nem perda de mensagens no subsistema de comunicação, os *heartbeats* chegam em q espaçados entre de 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 $delay + \Delta^i$, em que $delay$ é o intervalo de tempo necessário para viagem da mensagem.

Se variações no atraso são consideradas, cada mensagem m_k^{hb} terá um $delay_k$ associado. Como $A_k = \sigma_k + delay_k$, observa-se que $A_{k+1} - A_k = \Delta^i + (delay_{k+1} - delay_k)$. 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. Assim, sendo EA_k a estimativa realizada pelo detector para o instante de chegada de um *heartbeat* m_k^{hb}, 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.

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]. Dessa forma, o marco estimado no tempo FP (*Freshness Point* [Chen et al. 2002]) para chegada do próximo *heartbeat* é definido por: $FP_{k+1} = EA_{k+1} + \alpha_{k+1}$. Portanto, a adaptabilidade do detector consiste em ajustar EA e α [Nunes and Jansch-Pôrto 2004]. Assim, uma vez recebido m_k^{hb} 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}. O *Freshness Point* de m_{k+1}^{hb} pode ser reescrito em função de Δ_{k+1}^{to} [Chen et al. 2002]: $FP_{k+1} = FP_k + \Delta_{k+1}^{to}$ e, consequentemente, $\Delta_{k+1}^{to} = (EA_{k+1} - EA_k) + (\alpha_{k+1} - \alpha_k)$. Desse modo, pode-se afirmar que, a adaptabilidade está diretamente ligada a precisão de Δ^{to}.

Diversos estudos têm sido realizados no sentido de avaliar diferentes abordagens de detectores adaptativos de defeitos. Trabalhos como [Müller 2004], [Nunes and Jansch-Pôrto 2004], [Macêdo and Lima 2004], [Sá and Macêdo 2005] e [Falai and Bondavalli 2005] propõem, avaliam (usando as métricas de *QoS* para detecção de [Chen et al. 2002]) e realizam comparações entre diferentes abordagens de detecção adaptativa de defeitos.

4. *Matlab*, *Simulink* e *TrueTime*

Matlab [The Mathworks 2002] é um acrônimo 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. Computação e matemática, desenvolvimento de algoritmos, aquisição de dados, modelagem, simulação e prototipagem entre outros, são casos típicos de uso de tal ambiente. O *Matlab* é composto por cinco partes básicas, dentre as quais se destacam as seguintes: *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 autovalores, transformadas de *fourier* etc.); *linguagem Matlab*, que permite a manipulação de matrizes, contém declarações para fluxo de controle, funções, estruturas de dados, entrada/saída (E/S) e possui suporte para programação orientada a objetos.

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, controle, aquisição de dados, inteligência artificial, tempo real etc.

4.1. O Pacote Simulink

O *Simulink* [The Mathworks 2004] é um pacote de *software*, integrado ao *Matlab* que possibilita a modelagem, simulação e análise de sistemas dinâmicos[3]. 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, as saídas e os 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.

No *Simulink*, um diagrama em blocos é usado como uma representação gráfica do modelo matemático de um sistema dinâmico. Os diagramas 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 etc. Esses blocos podem ser agrupados para representar subsistemas maiores (blocos virtuais) ou compor diagramas. Blocos virtuais, por sua vez, representam uma composição de blocos e sinais, visualizados no modelo como um único bloco. Dessa forma, blocos virtuais podem novamente ser agrupados, em diagramas com blocos virtuais e/ou não virtuais, para compor um subsistema ainda maior e assim por diante.

4.2. O ToolBox TrueTime

O *TrueTime* [Henriksson and Cervin 2003] é um *toolbox* que, associado ao *Simulink*, permite simular sistemas de tempo real distribuídos de controle e supervisão, além de possibilitar o estudo dos efeitos do escalonamento de tarefas e da comunicação sobre o sistema modelado. Para tanto, o mesmo oferece quatro tipos de blocos: *Kernel*; *Network*; *Battery*; e *Wireless Network*.

O bloco *Kernel* 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. Além disso, o bloco mantém diversas estruturas de dados (como filas, registros, monitores etc.) naturalmente encontradas em sistemas operacionais com *kernels* de tempo real. O escalonamento das tarefas pode ser realizado seguindo uma política baseada em prioridades fixas ou dinâmicas. O bloco *Network* simula um meio de acesso e transmissão em uma rede local e 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 sobre as estações (transmissora e receptora), os dados do usuário, o instante da transmissão e, opcionalmente, atributos de tempo real (tais como *prazo* ou prioridade). Diferentes implementações de protocolos de comunicação são suportadas por esse bloco, exemplo: *CSMA/CD*, *CSMA/AMP*, *Round Robin*, *FDMA*, *TDMA* e *Switched Ethernet*. O bloco *Wireless Network* é uma extensão do bloco *Network* para suportar protocolos usados em redes sem fio (como: *IEEE 802.11b/g* e *802.15.4*) e que permite simular características peculiares a esse tipo de rede (como perda de sinal e interferência entre nós). O bloco *Battery* é usado em conjunto com o bloco *Kernel* para simular dispositivos que possuam restrições de

[3]Sistemas que mudam suas saídas em função do tempo, como sistemas elétricos, mecânicos etc.

consumo de energia. Para tanto, pode-se configurar nesse bloco a quantidade inicial de energia em *Watts*. Durante a simulação, uma tarefa pode ser criada para verificar ou administrar o consumo de energia.

5. O Framework Proposto

Um *framework* é um conjunto de classes que constitui um projeto abstrato para a solução de um problema, de modo a permitir maior modularidade, reusabilidade, extensibilidade e inversão do fluxo de controle [Schmidt and Fayad 1997]. De modo geral, um *framework* representa uma aplicação semi-completa de software que pode ser especializada para produzir aplicações específicas dentro de um determinado domínio de problema. A idéia básica para o *framework* proposto neste artigo foi suportar a simulação de *S&C* confiáveis, focando inicialmente na construção de detectores adaptativos de defeitos com suporte a *QoS* de detecção. As premissas observadas para a concepção de tal *framework* foi prover uma infra-estrutura flexível e reusável que pudesse acomodar a avaliação de diferentes estratégias e modelos de mecanismos de tolerância a falhas de forma rápida e simples.

5.1. Visão Geral

O *framework* proposto foi construído usando as facilidades existentes no *Toolbox TrueTime* e aproveitando os recursos existentes no ambiente *Matlab*/*Simulink*. Dessa forma, algumas das entidades contidas no *framework* encapsulam chamadas de sistemas disponíveis no *TrueTime*[4] e abstraem alguns detalhes de implementação para o programador.

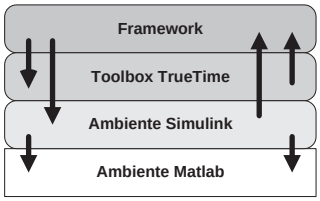


Figura 1. Arquitetura do *framework* proposto

Além de comunicar com o *TrueTime*, usando algumas das chamadas de sistemas disponíveis, entidades do *framework* interagem com o *Simulink* para controlar eventos de início e término de simulação, consultar informações de blocos existentes no modelo que está sendo simulado. A figura 1 apresenta um esquema exemplo de como a implementação do *framework* está estruturada. As setas indicam possíveis interações entre os elementos que compõem a base sobre a qual a implementação do *framework* foi realizada.

5.2. Composição da Infraestrutura

O conjunto de classes do *framework*, ver figura 2, está divido em duas infra-estruturas: a *infra-estrutura de ambiente* e a *infra-estrutura de confiabilidade*.

A *infraestrutura de ambiente* acomoda os módulos de classes e o modelo de interação necessário para descrever e simular um ambiente de sistemas de controle distribuídos, tais classes são: *Device*, *Task*, *Subsystem*, *Simulation* e *SimulationHistory*.

A classe *Device* representa um dispositivo em um sistema de controle. No *framework*, está classe foi especializada em *Controller*, *Sensor* e *Actuator*. Um *Device* encapsula funções

[4]Como *ttKernel, ttCreateTask, ttCreatePeriodicTask, ttInitNetwork, ttCurrentTime* etc.

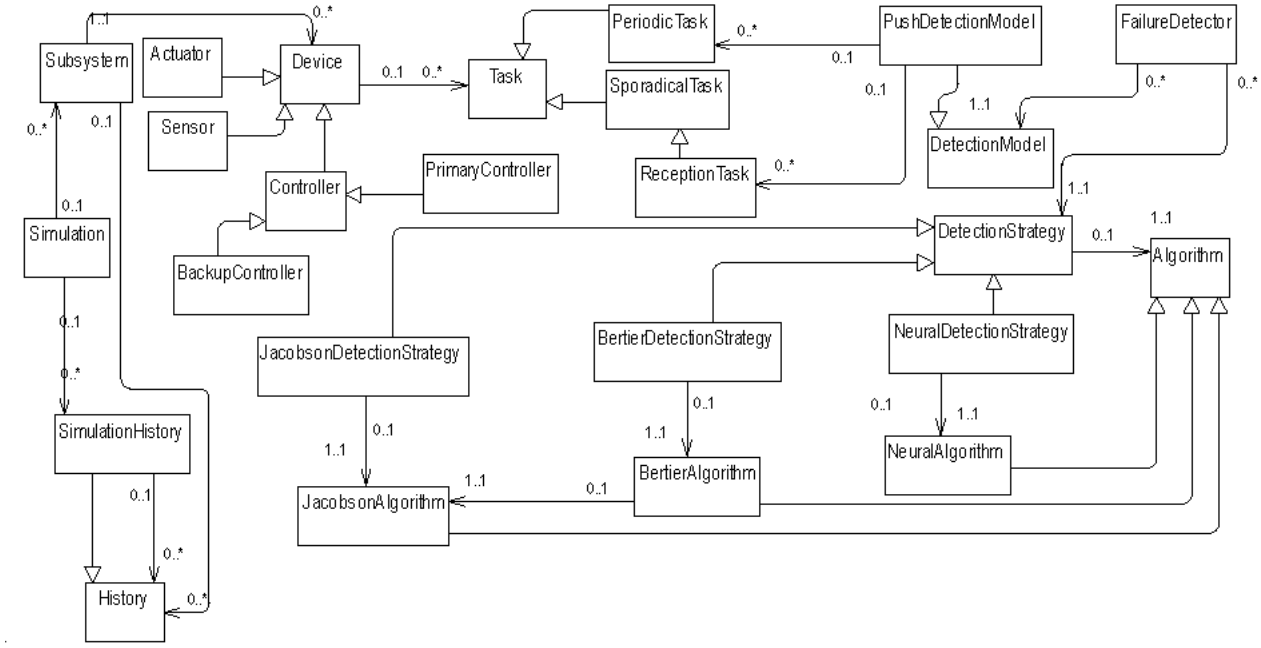


Figura 2. Diagrama das principais classes do *framework*

básicas do bloco *Kernel* do *TrueTime*, controlando a criação de tarefas, interação com outros *Devices* através da rede e permitindo que cada dispositivo específico possa acomodar as tarefas necessárias a sua categoria de atividade. A classe *Task* representa uma abstração para tarefas que podem ser executadas em um dispositivo. No *framework*, está classe foi especializada em *PeriodicTask* e *SporadicalTask*. A classe *Subsystem* acomoda a interação entre dispositivos de um mesmo sistema. Em um ambiente no qual diversos sistemas de controle podem compartilhar o mesmo meio de comunicação, um *Subsystem* representa o conjunto de dispositivos (*Devices*) de um mesmo sistema. Um *Subsystem* conhece o histórico de eventos relevantes de um dado dispositivo[5]. Um conjunto de subsistemas de controle que compartilham um ou mais canais de comunicação compõem um cenário de simulação. Uma instância da classe *Simulation* tem por objetivo comandar as ações de subsistemas (*SubSystem*s) e dispositivos (*Device*s). É responsabilidade da classe *Simulation* inicializar operações nos dispositivos, preparar a execução de tarefas e agrupar o histórico de eventos dos diversos subsistemas que compõem um cenário. A classe *SimulationHistory* representa o histórico de eventos armazenados em uma simulação. Todo *SimulationHistory* é um histórico de eventos (*History*) que pode ser composto por vários outros históricos de eventos associados a diferentes subsistemas.

A *infra-estrutura de confiabilidade* diz respeito ao conjunto de classes associadas aos mecanismos de tolerância a falhas, suas principais entidades são: *FailureDetector*, *DetectionModel*, *DetectionStrategy* e *Algorithm*.

A classe *FailureDetector* representa a implementação de um detector de defeitos, responsável por usar um modelo de monitoramento e uma estratégia de detecção de falhas para averiguar falhas em dispositivos do sistema e adaptar-se a variações temporais impostas pela velocidade dos dispositivos ou pelo tráfego no canal de comunicação. A classe *DetectionModel*, por sua vez, representa um modelo de monitoramento de falhas. Esse modelo pode ser

[5]É papel do programador apresentar ao *Subsystem* os eventos que precisam ter seu histórico armazenado.

especializado em *PushDetectionModel* e *PullDetectionModel*, os quais representam respectivamente os modelos *Push* e *Pull* (ver seção 3). A classe *DetectionStrategy* é uma abstração que representa uma estratégia para a adaptação a ser usada pelo mecanismo de detecção de falhas. No *framework* proposto neste artigo, toda estratégia de detecção usa um algoritmo de adaptação, sendo assim, é de sua responsabilidade coordenar a passagem dos parâmetros e dados necessários para a devida execução de tal algoritmo. *DetectionStrategy* foi especializada para implementar a abordagem de [Bertier et al. 2003] (*BertierDetectionStrategy*), [Jacobson 1988] (*JacobsonDetectionStrategy*) e [Sá and Macêdo 2005] (*NeuralDetectionStrategy* e *RPROPNeuralDetectionStrategy*). A classe *Algoritm* é a representação de um algoritmo, provendo as facilidades para definição de entradas, consulta a resultados de uma dada execução de algoritmo e possibilita chamada a outros algoritmos. Dentro da infra-estrutura de confiabilidade, essa entidade foi especializada em *BertierAlgoritm*, *JacobsonAlgoritm*, *NeuralAlgoritm* e *RPROPAlgoritm*.

5.3. Interação entre as Entidades

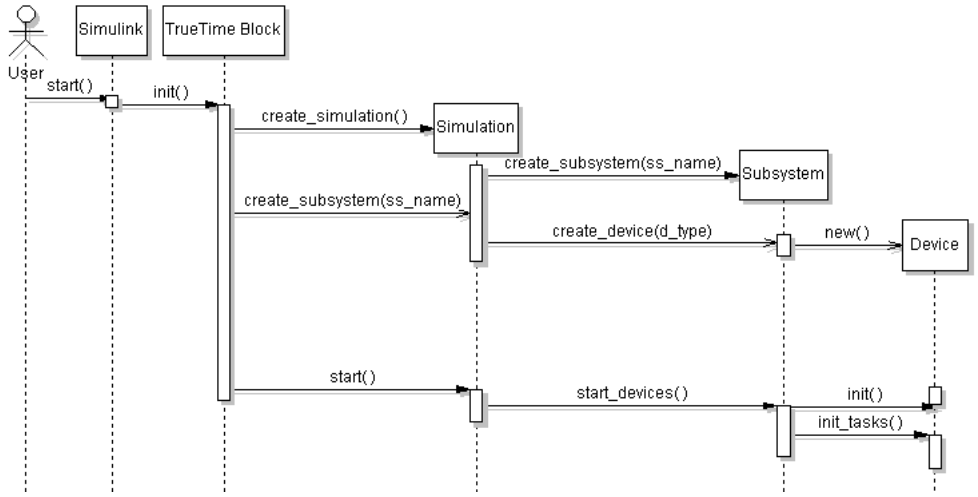


Figura 3. Diagrama de seqüência para inicialização da simulação do sistema

Ao modelar um cenário descrevendo um sistema de tempo real distribuído, o projetista deve especificar as entidades e como essas entidades irão interagir. Através dos *Device*s e *Subsystem*s implementados no *framework* um modelo natural de interação entre dispositivos sensores, atuadores e controladores já está previamente implementado. Para determinar detalhes de sensoriamento, atuação ou controle, o projetista deve estender a respectiva classe responsável por cada uma dessas ações e sobrescrever o método de execução ($run()$). Quando o modelo é posto em execução no *Simulink*, o mesmo executa a função de inicialização de cada um dos blocos *Kernel* que representam os dispositivos do sistema. A partir daí, cada bloco *Kernel* tentará gerar uma instância da entidade de simulação do *framework* proposto. A instância de simulação é única, assim o bloco que primeiro fizer a chamada à função *create_simulation()* criará a instância e os demais apenas receberão uma referência para a mesma (ver figura 3). Da mesma forma, cada bloco Kernel enviará uma mensagem à instância de *Simulation*, através do método *create_device()*, solicitando a criação de um dispositivo (*Device*) e informando o nome do subsistema (*Subsystem*), no qual tal dispositivo deve ser inserido. Assim, se o subsistema indicado já existe, o objeto *Simulation* captura a instância do subsistema, cria uma instância do dispositivo solicitado e em seguida insere o mesmo no subsistema. Caso o subsistema ainda não exista, este é criado para que em seguida a instância do

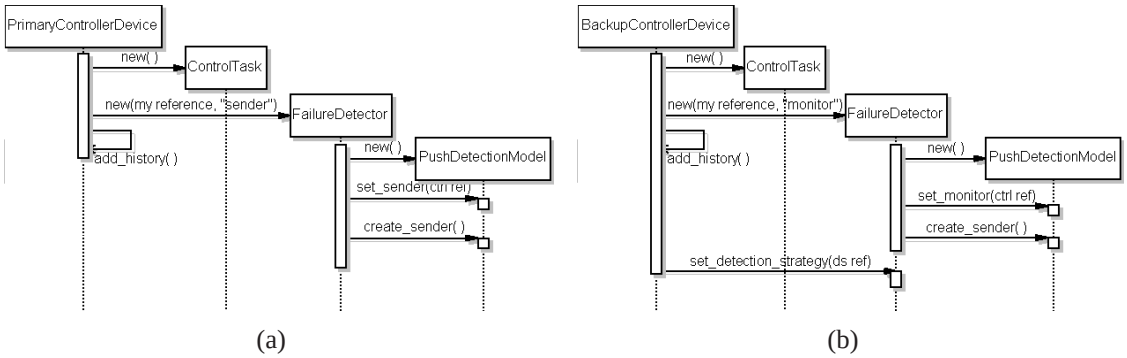
dispositivo seja criada e associada ao mesmo. Toda solicitação de criação de dispositivo é avaliada e, a depender do tipo do dispositivo, uma subclasse diferente será criada através de uma chamada de método específica (por exemplo, *create_controller_device*, *create_sensor_device*, *create_actuator_device* ou *create_backup_controller_device*). O próximo passo é solicitar a inicialização das tarefas em cada dispositivo, enviando uma mensagem à instância de *Simulation*, através do método *start()*. A inicialização de uma tarefa, em geral, envolve alocação de interfaces de *E/S*, definição e configuração das interfaces de rede e definição do tipo de escalonamento de tarefas a ser adotado.

Se o projetista deseja embutir um detector de defeitos em algum dos dispositivos do sistema, este deve criar o *Device* e adicionar antes da inicialização das tarefas do dispositivo a chamada para criação do módulo do detector que deseja instalar (módulo monitor ou modulo transmissor de *i am alive!*), usando o modelo de monitoramento e a estratégia de detecção que achar conveniente. A listagem de código 1 apresenta um exemplo de como instalar um módulo monitor de *heartbeats*, seguindo um modelo de monitoramento *Push*. Os passos para instalação do emissor de *heartbeat* são idênticos, diferindo apenas na definição do papel do módulo (ver linha 9).

Listagem de Código 1. Instalação do monitor de heartbeats em um dispositivo

```
1  my_device = new_backup_controller_device(my_subsystem, device_id);
2  my_history   = history('heartbeats', nhistories);
3  my_subsystem = add_history(my_subsystem, my_history);
4
5  %adicionar monitor failure detector task
6  pdm = push_detection_model('heartbeat', my_device);
7
8  %create a failure detector monitor
9  pdm                 = set(pdm, 'monitor', my_device);
10 [pdm, my_device]    = create_monitor(pdm);
11
12 set(my_failure_detector, 'detection_model', pdm);
13
14 %init device tasks
```

De modo geral, ao instanciar um dispositivo a ser monitorado por um detector de defeitos deve-se instalar no mesmo o módulo correspondente ao modelo de monitoramento adotado. No caso do modelo *Pull*, o dispositivo monitorado deve possuir um módulo *ecoador* de *i am alive*. No caso do modelo de monitoramento *Push*, tal dispositivo deve possuir um emissor de *i am alive* ou *heartbeats*. Da mesma forma, se cada dispositivo que necessita monitorar falhas de outro dispositivo, deve instalar o módulo de monitor de *i am alive* associado ao modelo de monitoramento adotado. No modelo *Pull*, diferente do modelo *Push*, além do processo de recepção de *i am alive* tem-se um mecanismo de emissão de *"are you alive?* associado.



(a) (b)

Figura 4. Detector distribuído. (a) emissor e (b) monitor de *heartbeats*.

As figuras 4(a) e 4(b) apresentam a seqüência de passos usadas para criar e associar um mecanismo de detecção de defeitos baseado no modelo de monitoramento *push* a dispositivos controladores primário e secundário, respectivamente.

5.4. Exemplos de Uso

5.4.1. Sistemas de Controle Distribuído com Controlador Replicado

A figura 5 apresenta um exemplo de diagrama em blocos construído no *Simulink* para a simulação de um sistema de controle confiável sobre rede. Em tal exemplo considera-se a existência um dispositivo sensor, um atuador e dois controladores (primário e secundário). Cada dispositivo é implementado usando um bloco *Kernel* do *TrueTime* e são conectados através do bloco *Network*.

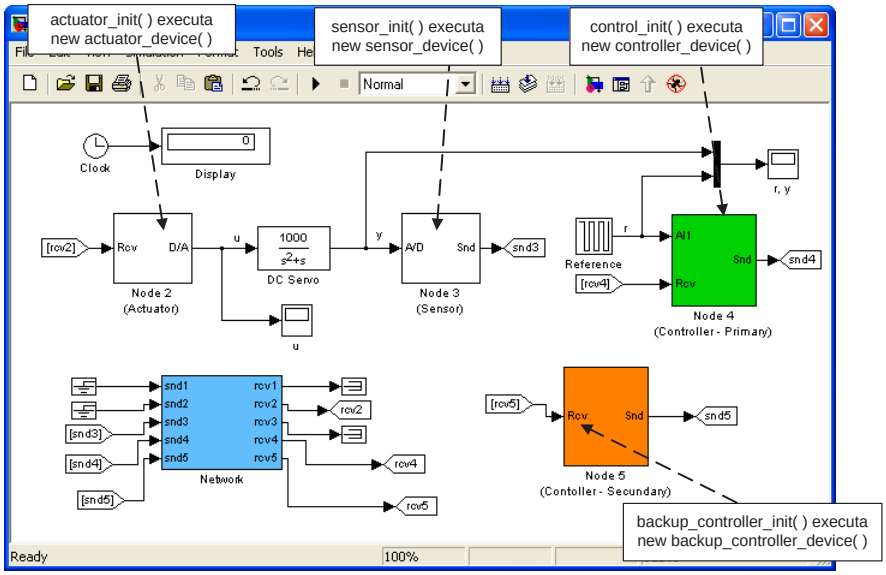


Figura 5. Modelo para sistema de controle confiável sobre rede

O controlador secundário possui um detector de defeitos embutido para verificar falhas por *crash* [Jalote 1994] do controlador primário, caso este último falhe, o controlador secundário assume e então passa a enviar as ações de controle para o atuador. A listagem 2 apresenta o trecho de código no qual o controlador secundário consulta, na linha 2, o detector de defeitos para verificar falhas do controlador primário. Nas linhas $5-7$, funções disponíveis no *framework* são utilizadas para obter o endereço do dispositivo atuador.

Listagem de Código 2. Trecho de código embutido no controlador secundário para enviar ação de controle após a falha do controlador primário

```
1  %consulta ao detector de defeitos para saber se o controlador primario esta OK
2   if is_fail(my_failure_detector, 'primary-controller')
3
4       %nao esta ok entao resolve endereco do atuador e envia acao de controle
5       the_subsystem_ref           = get_subsystem_ref(get_subsystem_name);
6       the_actuator_device_ref     = get_device_reference_by_name('actuator');
7       the_actuator_device_address = get_device_address(the_subsystem_ref, the_actuator_device_ref);
8
9       ttSendMsg(the_actuator_device_address, control_action, 8);
10  end
```

5.4.2. Usando o Framework para criar um nova estratégia de adaptação

Nesse exemplo, um novo detector de defeitos é implementado a partir da abordagem de detecção de [Bertier et al. 2003]. Contudo, o novo detector utiliza uma abordagem diferente para o cálculo da margem de segurança. A listagem de código *3* apresenta a implementação do construtor do novo detector de defeitos. Nas linhas de *3–7* a especialização do detector de [Bertier et al. 2003] é realizada, seguindo o modelo de programação orientada a objetos do *Matlab Script*. Das linhas *9–18*, o algoritmo de [Bertier et al. 2003] é modificado para adicionar uma referência a um algoritmo baseado em rede neural, usado em [Sá and Macêdo 2005], como parâmetro.

Listagem de Código 3. Nova abordagem de detecção de defeitos

```
1  function [my_detector] = new_failure_detector_approach(interrogation_time, gamma, beta, phi, window_size, A_history)
2
3  %cria um detector de defeitos baseado no algoritmo de bertier
4  the_super = bertier_failure_detector(interrogation_time, gamma, beta, phi, window_size, A_history);
5
6  %define que o novo detector é uma subclasse do detector de bertier
7  my_detector = class(detector, '_super', the_super);
8
9  %captura a estratégia de detecção
10 b_strategy  = get(my_detector, 'bertier_failure_detection_strategy');
11
12 %com a estratégia de detecção captura-se o algoritmo de detecção de bertier
13 b_algorithm = get(b_strategy, 'algorithm');
14
15 %adiciona o algoritmo de predição baseado na ultima estimativa como parametro ao algoritmo de bertier
16 the_rprop_neural_net_algorithm = new_rprop_neural_net_algorithm();
17
18 b_algorithm = add_parameter(b_algorithm, parameter('rprop_neural_net_algorithm', the_rprop_neural_net_algorithm));
19
20 %atribui o novo algoritmo a estrategia de bertier
21 b_strategy  = set(b_strategy, 'algorithm', b_algorithm);
22
23 %atualiza a estrategia de detecção no novo detector de defeitos.
24 my_detector = set(my_detector, 'bertier_failure_detection_strategy', b_strategy);
```

A abordagem para adaptação da margem de segurança usada pelo novo detector é apresentada na listagem de código *4*. Como pode ser observado nessa listagem, se a diferença entre os valores estimado e real para o instante de chegada de um *heartbeat* for maior que 10% do valor real (linha 3), então a abordagem usa uma rede neural para estimar a margem de segurança (linhas *4–6*), caso contrário a estratégia original de [Bertier et al. 2003] é utilizada (linha 18).

Listagem de Código 4. Exemplo do uso da facilidade da composição de algoritmos

```
1  function [my_algorithm, the_safety_margin] = compute_safety_margin(my_algorithm, current_A, last_EA)
2  %verifica o percentual
3  if(abs(current_A - last_EA)/current_A > 0.1)
4          %captura o algoritmo baseado em RPROP RNA
5          the_rprop_neural_net_algorithm = get(my_algorithm, 'rprop_neural_net_algorithm');
6
7          %define o padrao de entrada
8          pattern = [current_A last_EA];
9
10         the_rprop_neural_net_algorithm = set(the_rprop_neural_net_algorithm, 'pattern', pattern);
11
12         %calcula a margem de segurança
13         [the_rprop_neural_net_algorithm, the_safety_margin] = run(the_rprop_neural_net_algorithm);
14
15         %atualiza o estado do algoritmo de predição
16         my_algorithm    = set(my_algorithm, 'rprop_neural_net_algorithm', the_rprop_neural_net_algorithm);
17 else
18         [my_algorithm, the_safety_margin] = compute_safety_margin(my_algorithm._super, current_A, last_EA)
19 end
```

6. Considerações Finais e Sugestões de Trabalhos Futuros

Este artigo descreveu detalhes de implementação e exemplos de uso de um *framework* aplicável à prototipagem, simulação e análise de sistemas de *S&C* confiáveis. Tal *framework*, usando as facilidades existentes na ambiente *Matlab/Simulink* e *Toolbox TrueTime*, permite

de forma simples e flexível a construção de protótipos de sistemas de *S&C* dotados de mecanismos de detecção adaptativa de defeitos. Além disso, conforme apresentado, o *framework* dispõe de diferentes estratégias de detecção e permite que novas estratégias sejam criadas a partir da composição de estratégias existentes. Os procedimentos para o cálculo da *QoS* de detecção foram implementados usando *Matlab Script*. Toda avaliação da *QoS* dos detectores simulados foi feita *off-line*, usando o histórico de eventos armazenados durante a simulação. Tal decisão foi tomada para melhorar o desempenho durante a execução da simulação. Uma descrição dos procedimentos para o cálculo da *QoS* de detecção pode ser encontrada em [Sá 2006]. Como trabalho futuro, sugere-se que sejam adicionadas ao *framework* entidades que facilitem a prototipagem e teste de outros mecanismos de tolerância a falhas. Avaliações usando o *framework* proposto neste artigo podem ser encontradas em [Sá and Macêdo 2005] e [Sá 2006]. Além disso, é interessante que o framework atualmente implementado usando *Matlab Script* seja implementado em *C++* e integrado ao *toolbox TrueTime* para que se obtenha um custo computacional menor na execução do modelo simulado.

Referências

Aguilera, M. K., Chen, W., and Toueg, S. (1997). Heartbeat: A timeout-free failure detector for quiescent reliable communication. In Mavronicolas, M. and Tsigas, P., editors, *Proc. of the 11th Intern. Workshop on Distributed Algorithms*, volume 1320 of *LNCS*, pages 126–140. Springer-Verlag, Saarbrücken, Germany.

Andrade, S. and Macêdo, R. (2005). A component-based real-time architecture for distributed supervision and control applications. In *10th IEEE Intern. Conf. on Emerging Technologies and Factory Automation*, volume I, pages 19–22, Italy. ETFA2005.

Bertier, M., Marin, O., and Sens, P. (2003). Performance analysis of hierarchical failure detector. In *Proc. Of The Intern. Conf. On Dependable Systems And Networks*, pages 635–644, San-francisco,Usa. IEEE Society Press.

Branicky, M., Liberatore, V., and Phillips, S. (2003). Networked control system co-simulation for co-design. *Proc. of the 2003 American Control Conf.*, 4.

Chen, W., Toueg, S., and Aguilera, M. K. (2002). On the quality of service of failure detectores. *IEEE Trans. On Computer*, 51(2):561–580.

Elks, C. R., Dugan, J. B., and Johnson, B. W. (2000). Reliability analysis of hard real-time systems in the presence of controller malfunctions. In *Proc. of the XVIII Reliability and Maintainability Symp.*, pages 58–64, Los Angeles, CA. IEEE Computer Society Press.

Falai, L. and Bondavalli, A. (2005). Experimental evaluation of the qos failure detectors on wide area network. In *Intern. Conf. On Dependable Systems And Networks*.

Felber, P. (1998). 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.

Henriksson, D. and Cervin, A. (2003). Truetime 1.13 - reference manual. Tech. Report Isrn Lutfd2/Tfrt--7605--se, Dep. Of Automatic Control, Lund Institute Of Technology.

Jacobson, V. (1988). Congestion avoidance and control. *ACM Computer Comm. Review; Proc. Of The Sigcomm '88 Symp. In Stanford, Ca, August, 1988*, 18, 4:314–329.

Jalote, P. (1994). *Fault Tolerance In Distributed Systems*. Prentice Hall, New Jersey.

Kim, H. and Shin, K. G. (1994). On the maximum feedback delay in a linear/nonlinear control system with input disturbances caused by controller-computer failures. *IEEE Trans. on Control Systems Technology*, 2(2):110–122.

Lian, F., Moyne, J. R., and Tilbury, D. M. (2001). Performance evaluation of control networks: ethernet, controlnet, and devicenet. *IEEE Control Systems Mag.*, 21:66–93.

Lincoln, B. and Cervin, A. (2002). JITTERBUG: a tool for analysis of real-time control performance. *Proc. of the 41st IEEE Conf. on Decision and Control*, 2.

Lynch, N. A. (1996). *Distributed Algorithms*. M. Kaufmann, San Francisco, California.

Macêdo, R. J. A. and Lima, F. (2004). Improving the quality of service of failure detectors. *Simpósio Brasileiro de Redes de Computadores*.

Müller, M. (2004). Performance evaluation of a failure detector using SNMP. Semester project, École Polytechnique Fédérale de Lausanne, Switzerland.

Nunes, R. C. and Jansch-Pôrto, I. (2004). Qos of timeout-based self-tuned failure detectors: the effects of the communication delay predictor and the safety margin. In *Intern. Conf. On Dependable Systems And Networks*.

Ogata, K. (1990). *Modern Control engineering*. PH, Englewood Cliffs, 2nd edition.

Piuri, V. (1994). Design of fault-tolerant distributed control systems. *IEEE Trans. On Instrumentation and Measurement*, 43(2):257–264.

Sá, A. S. (2006). Detectores adaptativos de defeitos para sistemas de controle de tempo real críticos. Mestrado em mecatrônica, Programa de Pós Graduação de Mestrado em Mecatrônica, Departamentos de Engenharia Mecânica e Ciência da Computação, Universidade Federal da Bahia.

Sá, A. S. and Macêdo, R. J. A. (2005). An adaptive failure detection approach for real-time distributed control systems over shared ethernet. In *Proc. of 18th Intern. Congress of Mechanical Engineering – Symposium Series in Mechatronics*, volume 2, pages 43–50, Ouro Preto, Brazil. COBEM2005.

Schmidt, D. and Fayad, M. (1997). Object-oriented application frameworks. *Communications of the Association for Computing Machinery*, 40:32–38.

The Mathworks (2002). *Matlab:The Language of Technical Computing*. The Mathworks Inc., Nantick, USA.

The Mathworks (2004). *Simulink Reference:Simulation and Model-Based Design*. The Mathworks Inc., Nantick, USA.

Törngren, M. (1998). Fundamentals of implementing real-time control applications in distributed computer systems. *Real-Time Systems*, 14(3):219–250.

Wittenmark, B. and Törngren, M. (1994). Timing problems in real-time control systems: Problem formulation. Report, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.