Tratando a Previsibilidade em Sistemas de Tempo-Real Distribuídos: Especificação, Linguagens, *Middleware* e Mecanismos Básicos

Raimundo Macêdo, George Lima, Luciano Barreto, Aline Andrade,
Alírio Sá, Frederico Barboza, Rodrigo Albuquerque, Sandro Andrade

Laboratório de Sistemas Distribuídos (LaSiD)
Departamento de Ciência da Computação (DCC)
Universidade Federal da Bahia (UFBA)
Av. Adhemar de Barros S/N, Ondina,
Salvador - BA, CEP: 40.170-110

{macedo,gmlima,lportoba,aline}@ufba.br

{aliriosa,fredjrb,albuquerque,sandros}@ufba.br

Abstract

Achieving predictability in a distributed real-time system involves a series of complementary techniques: from the specification and formal verification of the system, through the utilization of proper middleware and languages, up to the usage of resource management techniques that guarantee a predictable behaviour of the target run-time environment. The aim of this chapter is to provide an integrated view of the many facets of real-time systems design, showing how they relate to each other and interfere in the development process of such systems.

Resumo

Garantir a previsibilidade de um sistema de tempo-real distribuído envolve uma série de técnicas complementares, que vão desde a especificação e verificação formal do sistema, passando pelo uso de *middleware* e linguagens de programação, até o gerenciamento de recursos que garantam um comportamento previsível do ambiente computacional (hardware e software). O objetivo deste capitulo é dar ao leitor uma visão integrada dos diversos aspectos de projeto de sistemas de tempo-real, mostrando como esses aspectos se inter-relacionam e interferem no processo de desenvolvimento de tais sistemas.

3.1. Introdução

O avanço tecnológico das últimas décadas colocou de forma definitiva os computadores no cotidiano da vida moderna, seja num sistema bancário, de transporte, de saúde, de comunicação, de manufatura, entre tantos outros. Se de um lado tal inserção traz inúmeros benefícios às nossas vidas, de outro coloca novos desafios tecnológicos para a construção de sistemas computacionais cada vez mais confiáveis e seguros, sem os quais prejuízos e desastres tornar-se-ão inevitáveis.

Os sistemas de tempo-real são um caso especial de sistemas computacionais em que o computador interage diretamente com o ambiente, não somente por meio de uma interface homem-máquina (*e.g.*, vídeo e teclado), mas sobretudo através de dispositivos que captam informações e que interferem no ambiente, chamados de sensores e atuadores, respectivamente. Exemplos desses sistemas vão desde simples fornos de microondas até sistemas de controle automático de vôo. Nesse sentido, o termo previsibilidade que acompanha o título deste curso refere-se à capacidade do sistema em executar ações desejadas (usualmente, através dos atuadores) dentro de intervalos de tempos determinados *a priori* e, via de regra, ditados pelo próprio ambiente no qual o sistema computacional está imerso. Por exemplo, o controle de pouso de um avião tem que acionar o trem de pouso tão logo seja detectada a proximidade do solo através de um sensor de altitude.

A possibilidade dos vários componentes de um sistema de tempo-real estarem distribuídos em diferentes computadores e se comunicarem para a realização de tarefas, coloca dificuldade adicional na garantia da previsibilidade, uma vez que o mau funcionamento de um simples componente ou do meio de comunicação pode causar violação de prazos ou mesmo a interrupção total do serviço.

Dentro desse contexto, pode-se definir um sistema como previsível se o seu comportamento (funcional e temporal) pode ser de algum modo antecipado ou previsto. O comportamento funcional refere-se ao resultado de uma computação, como o acionamento do trem de pouso do exemplo citado. Já o comportamento temporal refere-se ao instante em que determinada ação é produzida (observe que o trem de pouso tem que ser acionado antes da aterrissagem). Para que o sistema funcione de forma previsível é necessário, em primeiro lugar, que haja um gerenciamento adequado dos recursos disponíveis a fim de que ações de tempo-real não sofram atrasos imprevistos, por exemplo, devido à indisponibilidade de memória, CPU, banda passante na rede de comunicação ou mesmo a execução simultânea de diferentes tarefas (acionadas a partir de estímulos diversos do ambiente).

Havendo a garantia de que os recursos necessários para a execução do sistema estão disponíveis, o único impedimento que pode suceder é a ocorrência de defeitos nos componentes, sejam eles no software ou no hardware. De fato, uma das poucas certezas que existe na área de computação é sobre a impossibilidade de se construir sistemas totalmente imunes à defeitos, ou seja, totalmente previsíveis. O que pode ser feito, entretanto, é minimizar a ocorrência desses defeitos através de mecanismos e técnicas diversas. Por exemplo, existem técnicas que diminuem a possibilidade de se introduzir falhas já na etapa de especificação do sistema (*e.g.*, especificação e verificação formal do software e hardware dos componentes). Outras técnicas introduzem robustez no sistema de tal forma

que falhas, mesmo que ocorram em tempo de execução, podem ser toleradas sem comprometer a previsibilidade (*e.g.*, replicação de componentes do sistema). Vale ressaltar que o esforço e recursos depreendidos para se verificar e colocar redundância no sistema deve estar diretamente relacionado ao custo advindo da violação de sua previsibilidade. Os sistemas de segurança críticos, como controle de plantas nucleares e de vôo de aeronaves tripuladas, requerem uma previsibilidade próxima de 100 por cento. Por outro lado, ações de sistemas convencionais (fora do escopo de tempo-real), como, por exemplo, consultas num cadastro de uma empresa, podem ser realizadas num intervalo de tempo muito menos rígido. Sendo assim, não há a necessidade de se garantir a previsibilidade em tais sistemas, apesar de ser imprescindível que sua correção (*e.g.*, o cadastro deve sempre estar consistente) seja assegurada.

Resumindo, garantir a previsibilidade de um sistema de tempo-real não é tarefa simples. De fato, isso envolve: (a) especificar as funções do sistema de tal forma que tanto seu comportamento funcional quanto o temporal estejam precisamente definidos; (b) usar ferramentas adequadas (linguagens de programação e *middleware*) para facilitar a implementação dessas especificações; (c) prover o gerenciamento dos recursos computacionais disponíveis a fim de que o sistema cumpra sua especificação; e (d) quando necessário, prover redundância dos componentes do sistema para aumentar a confiança no seu funcionamento. Cada um desses temas, geralmente, são tratados independentemente, o que pode fazer parecer que os itens (a)-(d) sejam assuntos desconexos. Um dos objetivos desse curso é mostrar a ligação entre os mesmos no que diz respeito à construção de sistemas de tempo-real previsíveis.

O restante do presente capítulo está estruturado da seguinte forma. Inicialmente, a seção 3.2 indroduz algumas definições e conceitos que serão usados ao longo do capítulo. A seção 3.3 apresenta uma discussão sobre a utilização de métodos formais para especificação de tempo-real. As seções 3.4 e 3.5 cobrem, respectivamente, aspectos de linguagens de programação e *middleware* relacionados aos sistemas de tempo-real. A seção 3.6 trata dos aspectos de suporte básico à previsibilidade, com ênfase nos métodos de escalonamento. A seção 3.7 discute o uso de técnicas de tolerância a falhas para aumentar a confiabilidade dos sistemas de tempo-real distribuídos. Finalmente, a seção 3.8 apresenta nossas considerações finais.

3.2. Conceitos Básicos de Sistemas de Tempo-Real

Alguns conceitos básicos sobre o tema abordado nesse curso serão aqui apresentados. Isto inlcui uma discussão informal sobre tempo e previsibilidade, caracterização de sistemas de tempo-real e considerações sobre a ocorrência de falhas em tais sistemas. Os conceitos apresentados ao longo do capítulo serão ilustrados através de uma aplicação-exemplo, que será descrita ao final desta seção. Obviamente, não é a intenção aqui detalhar um sistema real e sim apresentar, de forma simplificada, um sistema que seja suficientemente representativo aos propósitos do curso. A finalidade é usar um exemplo único ao longo do curso, de tal forma que os diferentes conceitos apresentados possam ser integrados, facilitando assim a assimilação dos mesmos por parte do leitor.

3.2.1. Tempo e Previsibilidade

Aplicações de tempo-real precisam expressar seus requisitos em função do tempo, o que inclui momentos para disparar determinados eventos, determinação dos momentos em que eventos ocorreram, medir a duração de operações, ordenar temporalmente mensagens recebidas etc. Portanto, é imprescindível a existência de uma referência de tempo comum para todos os elementos envolvidos na consecução das tarefas e isso exige a implementação de um sistema de relógios sincronizados entre si (sincronização interna) e em relação a uma referência externa (sincronização externa). A sincronização interna é suficiente se os requisitos das aplicações envolvem apenas medidas de duração de operações. Contudo, se as medidas precisam ser relacionadas às horas do dia, então a sincronização externa precisa ser realizada.

Para se realizar a sincronização dos relógios existem algumas dificuldades inerentes aos sistemas físicos. Isoladamente, em cada computador, um relógio é implementado a partir de um contador que recebe estímulos periódicos de um oscilador (*e.g.*, quartzo). Uma dificuldade inicial é que tais osciladores não são perfeitos, gerando um desvio durante determinados períodos que exigem calibrações periódicas dos relógios. Num sistema de tempo-real distribuído, que envolve vários relógios de vários computadores, dificuldades adicionais surgem devido aos tempos de propagação das mensagens, sendo que todos os relógios distribuídos precisam ser re-sincronizados periodicamente. Além dessas dificuldades, os relógios ou processos (tarefas) que os controlam podem falhar, gerando incertezas. Todas essas limitações implicam em relógios onde se pode atingir, dependendo das características físicas dos osciladores, computadores e redes de comunicação, uma dada precisão (*precision*), que define a diferença máxima entre medidas de relógios distintos, exatidão (*accuracy*) que define a diferença máxima entre um relógio e uma referência externa, e, finalmente, granularidade, que define a duração de uma batida do relógio (que, por sua vez, representa um número múltiplo de batidas no relógio individual). O problema de sincronização de relógio foi bastante estudado, havendo vários algoritmos que atendem aos variados requisitos das aplicações de tempo-real [53, 67, 66, 82]. Neste curso, quando nos referirmos ao tempo, assumiremos a existência de um sistema de relógios devidamente sincronizados com a precisão, exatidão e granularidade requeridas pelas aplicações em discussão, e por questões didáticas, omitiremos os detalhes de granularidade na referência aos momentos de tempo expressos.

Como a especificação de sistemas de tempo-real envolve aspectos temporais, resultados corretos em tais sistemas significam valores corretos produzidos nos momentos corretos. Obviamente, o momento correto depende da aplicação e/ou do contexto que ela está inserida. Por exemplo, suponha que o tempo de transmissão de um sinal de rádio da Terra para Júpiter é de uma hora. Então, qualquer comando enviado para uma nave orbitando Júpiter não pode levar menos do que uma hora para ser executado. Esse tempo de espera, contudo, é inaceitável para um usuário de uma empresa de telefonia. Portanto, uma aplicação que precise de uma hora para realizar conexões telefonicas será facilmente considerada incorreta, mesmo se, ao final, as conexões sejam feitas com sucesso.

A seguir introduziremos outros aspectos de projetos relacionados à garantia de previsibilidade de sistemas de tempo-real. Esses conceitos serão aprofundados nas seções que seguirão.

3.2.2. Tarefas de Tempo-real e seus Atributos

Um sistema computacional de tempo-real típico controla ou monitora elementos do mundo real (também denominado de ambiente), representados pelos sensores, atuadores e operadores do sistema. O sistema computacional de tempo-real deve reagir a eventos oriundos das entidades que pertencem ao ambiente, como mudança na temperatura de um forno, comandos do operador (através da interface homem-máquina) ou, até mesmo, a própria passagem do tempo [50]. Eventos que são relevantes ao sistema são mapeados de forma que eles gerem estímulos (*e.g.,* sinais eletrônicos) para processamento pelo sistema computacional. Por exemplo, a temperatura atmosférica pode ser irrelevante para determinado sistema, mas a temperatura de um forno que está sendo controlado deve ser considerada relevante, e, portanto, percebida pelo sistema computacional. Para tanto, existem interfaces (*e.g.,* sensores, conversores de sinal) entre o sistema computacional e o ambiente, responsáveis por mapear os eventos relevantes ao sistema em estímulos que ativam sua computação - geralmente estruturada em um conjunto de *tarefas*. Portanto, uma tarefa pode ser informalmente definida como uma seqüência de ações executadas pelo sistema, e acionada pela ocorrência de eventos do ambiente. Como outro exemplo, considere a temperatura de uma caldeira que passou de seu limite operacional. A interface da caldeira, representada por um sensor de temperatura, faz chegar ao sistema computacional essa informação (evento) e, a partir daí, uma ou mais tarefas corretivas são ativadas. Cada uma dessas tarefas, por sua vez, ativa outras tarefas ou responde ao ambiente através da interface (*e.g.,* atuadores, válvulas) para que haja a devida resposta ao aumento de temperatura.

Neste capítulo, um determinado sistema ou aplicação será representado por um conjunto de n tarefas $\Gamma = \{\tau_1, \tau_2, ...\tau_n\}$. Cada uma dessas tarefas é descrita em termos de atributos. Alguns dos atributos mais comumente usados são [22, 23, 50, 34]:

- *Deadline*. As especificações temporais de um sistema de tempo-real são, na maioria das vezes, expressas em função do intervalo máximo de tempo que cada uma das suas tarefas deve executar. Esse intervalo de tempo é relativo ao início da execução da tarefa e é conhecido como *deadline*. Para cada tarefa τ_i, seu *deadline* relativo será representado por D_i.
  Existem algumas tarefas que não podem violar seus *deadlines*, pois gerariam resultados incorretos. Essas tarefas são consideradas de tempo-real rígido (*hard real-time*). Se essas tarefas de *deadline* rígido estão associadas a serviços críticos, que envolvem vidas humanas ou grandes quantidades de recursos financeiros, então são chamadas de tarefas críticas. Sistemas que possuem algum serviço composto de uma ou mais tarefas críticas são chamados de sistemas de tempo-real críticos (*safety-critical systems*). Sistemas críticos, como controle de vôo, controle ambiental etc., devem possuir alto nível de previsibilidade.
  Quando não há tarefas com *deadline* rígido no sistema, ele é conhecido como sistema de tempo-real brando (soft real-time systems). Descumprir *deadlines* nesse caso é aceitável em termos de custos para a aplicação ou para o usuário, embora não seja desejável. Sistemas multimídia são exemplos de tais sistemas.
- Período. Este atributo, representado aqui por T_i para cada tarefa τ_i, diz respeito às ativações das tarefas no sistema. Se há um período de ativação da tarefa conhecido, a tarefa é dita periódica. Algumas tarefas são ditas esporádicas. Para

essas, geralmente, o período significa o mínimo tempo entre duas ativações consecutivas. Quando nada se sabe sobre as ativações de uma tarefa, diz-se que ela é aperiódica.

- *Jitter*. Esse atributo representa a variação máxima de duas ativações consecutivas que uma tarefa periódica ou esporádica pode apresentar. Por exemplo, se uma tarefa periódica τ_i possui *jitter* $J_i = 0$, então ela é ativada no sistema a cada T_i unidades de tempo. Considerar esse atributo é importante, dado que variações no intervalo entre ativações de uma tarefa pode afetar o comportamento temporal do sistema.
- Tempo máximo de execução. Esse atributo representa o tempo de execução da tarefa no pior caso, considerando uma dada arquitetura. Essa informação é fundamental para prever o comportamento temporal do sistema (*e.g.*, todas as tarefas terminam antes de seus *deadlines*?). Esse tempo é derivado por técnicas especiais [76] que levam em consideração detalhes do sistema computacional (hardware e software) onde as tarefas serão executadas. Representa-se por C_i o tempo máximo de execução da tarefa τ_i.

Dado que os atributos acima são conhecidos, os desenvolvedores do sistema devem decidir como as tarefas serão ativadas no sistema. Há duas abordagens conhecidas. A primeira consiste em ativar as tarefas tão logo os eventos a elas associados sejam percebidos pelo sistema [19]. Nesse caso, diz-se que a ativação das tarefas é *baseada em* ou *orientada à* eventos (*event-triggered*). A segunda alternativa é fazer com que as tarefas sejam ativadas em instantes pré-determinados (*time-triggered*), ou seja, suas ativações ocorrerão em momentos específicos no tempo. Para comparar ambas as abordagens, suponha que uma tarefa está associada ao evento "temperatura da fornalha maior que 1.000 graus". Na primeira abordagem, tal tarefa será ativada tão logo o sistema computacional perceba o evento através da sua interface. Na segunda, contudo, o sistema somente acionará a tarefa em um momento pré-definido, independente da ocorrência do evento. Há uma discussão na comunidade científica a respeito de qual abordagem é a mais apropriada. Argüi-se que a primeira é mais flexível enquanto que a segunda favorece à previsibilidade em detrimento da flexibilidade. Como será visto ao longo do texto, essa argumentação não deveria focalizar somente esses extremos. De fato, há várias situações onde é possível garantir previsibilidade com um grau de flexibilidade razoável.

Como visto, um sistema de tempo-real é constituído por um conjunto de tarefas, que possuem requisitos temporais próprios. Suponha, por exemplo, que devido ao aumento repentino da temperatura de uma fornalha, uma determinada tarefa seja ativada. Essa tarefa pode, em princípio, usar recursos computacionais necessários para a execução de outras tarefas, podendo assim causar a violação de algum *deadline*. Tal cenário deve ser evitado através de técnicas e mecanismos que forneçam o gerenciamento adequado dos recursos do sistema. Um dos problemas principais que surge nesse gerenciamento é o escalonamento de tarefas, o qual sofre influência de vários fatores, tais como relações de precedência entre as tarefas, suas interações etc. A Seção 3.6 tratará desse problema.

3.2.3. Obstáculos à Previsibilidade

Quando uma ação computacional se desvia do que foi especificado ou esperado, o sistema se mostra defeituoso. Um defeito é assim a externalização de um erro. Erros, definidos

como estados incorretos do sistema, por sua vez, são causados por falhas [57].[1]

Um defeito ocorre na dimensão funcional se o sistema produz resultados (valores) fora de sua especificação. Na dimensão temporal um sistema pode produzir resultados *antecipadamente* ou em *atraso*. Se algum resultado nunca é produzido, diz-se que o sistema apresenta defeitos por *omissão*. Esse tipo de defeito pode ser considerado pertencente tanto à dimensão funcional (resultados nulos são produzidos) quanto à temporal (o sistema está infinitamente atrasado). Omissões persistentes são chamadas de defeitos tipo *crash*. O tipo mais genérico de defeito que um sistema pode apresentar é chamado de *bizantino* ou arbitrário. Nesse caso, o sistema pode apresentar comportamento completamente imprevisível em ambas as dimensões.

Construir um sistema de tempo-real 100% previsível é, portanto, evitar seus possíveis defeitos. Como é impossível construir-se tal sistema perfeito, na prática, busca-se minimizar a probabilidade da ocorrência de defeitos, tanto funcionais quanto temporais. Isso pode ser feito através de diversas técnicas complementares, seja durante a especificação do sistema, da sua programação ou através da implementação de mecanismos que, em tempo de execução, aumentam a confiança no funcionamento do sistema.

Erros durante a fase de concepção do sistema tendem a se propagar durante sua construção ou implementação e execução. Deseja-se, portanto, detectar tais erros antes mesmo do sistema ser construído. Algumas técnicas, tais como prototipação e testes, auxiliam nessa tarefa. No caso de sistemas de tempo-real críticos, contudo, apenas essa abordagem não é o bastante. Outra alternativa é o uso de ferramentas baseadas em formalismos através das quais os comportamentos funcional e temporal do sistema possam ser *provados*. Só então é possível certificar-se de que a especificação do sistema está correta de acordo com critérios de correção identificados. Na seção 3.3, uma das ferramentas usadas para esse fim será brevemente descrita.

Mesmo que a especificação, programação, implementação e o gerenciamento dos recursos estejam corretos, o sistema pode ainda apresentar comportamento imprevisível. Por exemplo, se uma tarefa no sistema apresenta algum defeito na sua execução devido ao mau funcionamento do hardware causado, por exemplo, por interferência eletromagnética , nem seu comportamento temporal nem o funcional podem ser mais assegurados. Existem várias técnicas que lidam com esse tipo de possibilidade [57], visando assegurar a confiança no funcionamento do sistema (*dependability*), mesmo na ocorrência de defeitos. Como será visto, elas estão baseadas em prover um nível adequado de redundância no sistema de tal forma que a falha seja mascarada (*e.g.*, através de uma réplica da tarefa) ou o sistema seja recuperado (*e.g.*, outra tarefa é ativada com tal objetivo). Nesse sentido, sistemas distribuídos são uma maneira natural de conceber redundância na computação do sistema. Esse tópico será abordado na seção 3.7.

3.2.4. Suporte ao Desenvolvimento

Dado que a especificação do sistema esteja correta e todos os mecanismos não funcionais (*e.g.*, confiabilidade) devidamente planejados, o próximo passo é a utilização de ferramentas de desenvolvimento que forneçam abstrações adequadas aos desenvolvedores do

[1] Essa nomenclatura, apresentada por Laprie [57], é a mais comunmente aceita pela comunidade científi ca. Neste texto, os termos defeitos e falhas foram traduzidos do inglês *failure* e *faults*, respectivamente.

sistema. Em particular, tais ferramentas devem ser capazes de expressar os requisitos temporais, além dos funcionais presentes em aplicações convencionais.

Encaixa-se nessa categoria de ferramenta, as linguagens de programação e as arquiteturas de *middleware*. As linguagens são responsáveis por fornecer a devida interface para o desenvolvedor de sistemas, mapeando os seus requisitos para os ambientes de execução (por exemplo, máquina virtual, *middleware* e sistema operacional). Há diversas linguagens destinadas à construção de sistemas de tempo-real, a exemplo de Pearl e Esterel. Já o *middleware* fornece um nível de abstração intermediário entre o ambiente de execução e a aplicação, "escondendo" do desenvolvedor da aplicação detalhes do ambiente, como de localização de serviços, heterogeneidade de plataformas usadas etc. As seções 3.4 e 3.5 abordam, respectivamente, linguagens de programação e *middleware*, enfatizando aspectos relacionados com previsibilidade.

3.2.5. Exemplo Ilustrativo

Essa seção apresenta uma aplicação ilustrativa, que representa um UAV (*Unmanned Air Vehicle*), o qual será utilizado durante o decorrer do texto. Os dados e a modelagem usados são fictícios e têm o propósito didático.

Um UAV é um veículo aéreo não tripulado que realiza atividades através de controle autônomo ou remoto. Esta categoria de aeronave vem sendo cada vez mais utilizada, tanto em aplicações civis (mapeamento de regiões, estudos climáticos etc.) quanto em aplicações militares (coleta de informações sobre forças inimigas e suas instalações, reconhecimento de armas químicas e biológicas, lançamento de mísseis etc.) [14, 41]. A figura 3.1 fornece uma ilustração de um UAV típico [93].

Figura 3.1: Ilustração de um UAV.

O tipo de UAV modelado no exemplo aqui descrito é conhecido como pré-programado [14], pois é completamente autônomo, capaz de realizar vôos completos entre dois pontos previamente conhecidos e reagir adequadamente em situações não previstas sem necessidade de intervenção humana. De forma simplificada, dois sistemas computacionais básicos fazem parte do exemplo: o Sistema de Controle de Navegação (SCN) e o Sistema de Controle de Atitude (SCA). O primeiro é responsável por criar/gerenciar um conjunto de metas que viabilizem a partida e a chegada ao destino. Um exemplo de meta pode ser sobrevoar uma certa área, a certa altura, seguindo uma determinada rota de vôo. O SCN, portanto, necessita de informações sobre a posição relativa da aeronave a cada

instante. Já o SCA interage com sensores e/ou atuadores para cumprir a meta independentemente das perturbações externas (*e.g.*, mudança de velocidade do vento) que possam ocorrer.

Para efeito de ilustração, o sistema SCN é composto de três tarefas (τ_{gps}, τ_{vrf} e τ_{ctl}) enquanto que o SCA contém apenas uma tarefa (τ_{atd}). Os atributos dessas tarefas estão descritos na tabela 3.1. A tarefa τ_{gps} é periódica e responsável pela leitura por GPS dos dados de localização. A tarefa τ_{vrf}, também periódica, se encarraga de verificar se a rota que está sendo feita corresponde à rota estabelecida. O procedimento de verificação pode assim requisitar a ativação da tarefa de controle (τ_{ctl}) caso seja necessário. Quando acionada, τ_{ctl} tem por finalidade estabelecer nova meta que deverá ser obedecida pelo SCA. Por exemplo, τ_{ctl} pode alterar ou ajustar alguns parâmetros de definição de rota de vôo como, por exemplo, direção e altura. Note que τ_{ctl} é esporádica e o tempo mínimo entre duas ativações consecutivas é igual ao período de τ_{vrf}. Em outras palavras $T_{\text{ctl}} = T_{\text{vrf}}$, pois pode haver a necessidade de ativar τ_{ctl} a cada instância de τ_{vrf}. Os parâmetros de definição de rota de vôo são lidos pelo sistema SCA, através da tarefa τ_{atd}. Isso significa que a comunicação entre τ_{ctl} e τ_{atd} é aqui modelada por compartilhamento de recursos. Com base nestes parâmetros e nas informações dos sensores do UAV, o SCA deve ser capaz de determinar os posicionamentos adequados dos atuadores para que a meta determinada pelo SCN seja cumprida. É importante observar que, mesmo que não haja alteração dos parâmetros de rota de vôo, pode haver a necessidade de se executar τ_{atd} (outros eventos, tais como mudança na velocidade ou direção do vento, por exemplo, podem requerer seu processamento). Por esse motivo, modelamos τ_{atd} como uma tarefa periódica.

Tabela 3.1: Tarefas dos Sistemas SCN e SCA.

Sistema	Tarefa	T_i	C_i	D_i	Tipo
SCN	τ_{gps}	100	20	100	Periódica
	τ_{vrf}	150	40	120	Periódica
	τ_{ctl}	150	60	140	Esporádica
SCA	τ_{atd}	200	50	200	Periódica

Há diversos componentes de hardware que fazem parte do UAV. Entre eles, encontram-se sensor(es) de posição baseados em GPS, sensor(es) de direção baseados em antena de rádio freqüência, altímetro, tacômetro etc. No decorrer do curso, alguns aspectos de hardware serão mencionados quando forem necessários ao entendimento dos conceitos apresentados. Contudo, na maior parte do texto, mais ênfase será dada às tarefas descritas na tabela 3.1.

3.3. Especificação Formal

Na seção anterior, a previsibilidade foi apresentada como um conceito essencial no projeto de sistemas de tempo-real. Uma alternativa para a garantia de previsibilidade do comportamento dos sistemas é o uso de métodos formais. Métodos formais compõem o conjunto de linguagens, técnicas e ferramentas baseadas em modelos matemáticos e destinadas a fornecer um padrão rigoroso de modelagem e análise de sistemas computacionais

[25, 29]. Estes métodos podem ser utilizados na descrição do sistema e na especificação e verificação de propriedades, ainda durante a fase de projeto, possibilitando eventuais correções e a certificação do seu comportamento.

A verificação de modelos (*model checking*) é uma técnica que permite a verificação de propriedades de sistemas de modo automático [26]. Esta é uma técnica aplicável aos sistemas computacionais que podem ser modelados por autômatos finitos ou variações deste estilo de representação, e consiste em três passos: a representação de uma aplicação através de um modelo $\mathcal{A}$ representado como um sistema de transição de estados finitos (autômatos); a representação de uma propriedade ϕ em uma linguagem lógica de especificação de propriedades e, finalmente, a execução de um algoritmo de verificação que responda a questão: "o modelo $\mathcal{A}$ satisfaz a propriedade ϕ, ou seja, $\mathcal{A} \models \phi$?" [8]. Diversas são as variações de algoritmos de verificação implementadas nas muitas ferramentas de verificação de modelos para sistemas de tempo-real disponíveis [39, 42, 73, 86, 96].

Esta seção apresenta uma ferramenta de verificação de modelos largamente utilizada, chamada UPPAAL [7]. O UPPAAL é uma ferramenta para modelagem, simulação e verificação de sistemas de tempo-real. Esta ferramenta está disponível gratuitamente para fins não comerciais. Para a apresentação do UPPAAL, serão modeladas algumas tarefas do exemplo do UAV definido na seção anterior.

3.3.1. Os Autômatos com Tempo

Um sistema é modelado no UPPAAL através de uma variante dos autômatos com tempo (*timed automata*) [2]. Um autômato com tempo é um autômato finito incrementado com um conjunto finito de relógios contínuos [26]. Neste modelo, as transições são assumidas como instantâneas, podendo, entretanto, o tempo evoluir enquanto o autômato permanece em um de seus estados. Assim, para modelar um evento α de grande duração é necessário dividi-lo em duas transições distintas: o início de α e o final de α.

A cada transição é possível associar três elementos:

- um *guarda*, que é uma condição de valores de relógio. A transição somente pode ser realizada se os valores atuais dos relógios satisfizerem a condição descrita;
- um *rótulo* ou *nome de ação* similar aos existentes nos autômatos tradicionais. Condições de sincronização podem ser representadas por ações que descrevam leitura (`nomeCanal?`) e escrita (`nomeCanal!`) em canais;
- um conjunto *reset* composto por uma coleção de relógios, que serão reiniciados quando a transição for executada.

A cada estado pode também ser associado um conjunto de condições de valores de relógios que deve sempre ser respeitado pelo sistema, chamado de invariantes de estados.

Na figura 3.2, apresentada a seguir, a expressão `local >= T_VRF` representa um guarda (onde `>=` significa $\geq$ na sintaxe de UPPAAL), a variável `controle!` representa um rótulo de ação e a expressão `tmpExec := 0` representa um elemento de um conjunto *reset*. A condição `tmpExec <= C_VRF` é um invariante para o estado `Calc`.

Em qualquer instante do tempo, pode-se descrever o sistema através de um par (q, v), composto pelo estado atual q do autômato e por um conjunto valoração v que

atribui um número real não negativo para cada um dos relógios do autômato. O sistema pode evoluir de duas formas:

- pela ativação do autômato através de uma transição ação, que leva o sistema da configuração (q, v) para (q', v'), considerando que a condição guarda para esta transição é satisfeita, que a ação associada à transição será executada, que todos os relógios do conjunto *reset* da transição serão zerados e que os demais relógios do autômato permanecerão com os valores atuais;
- pela passagem de tempo através de uma transição *delay*, quando todos os relógios são incrementados de d ($d \in \mathbb{R}^+$), levando o autômato da configuração (q, v) para a configuração $(q, v + d)$, considerando que as condições invariantes do estado q continuam observadas.

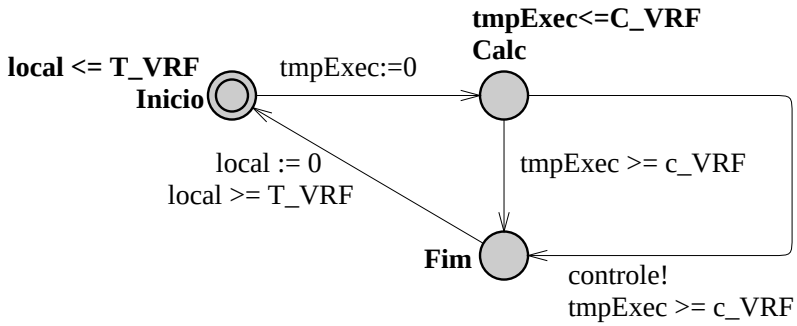

Figura 3.2: Modelagem da Tarefa τ_{vrf} do UAV.

A figura 3.2, gerada pelo UPPAAL, apresenta um modelo da tarefa τ_{vrf} do UAV. Para este modelo foram declarados dois relógios, $local$ e $tmpExec$, que representam respectivamente o período e o tempo de execução da tarefa, e foram identificados três estados. O primeiro estado (`Inicio`) descreve o fato da tarefa se encontrar pronta para iniciar sua execução. Como a tarefa é periódica, ela necessita deixar este estado antes do final do seu período (`T_VRF`). Este fato é descrito através da associação de uma condição invariante associada ao estado `Inicio` (`local <= T_VRF`). Durante a permanência da tarefa no estado (transição *delay*) todos os relógios do modelo são incrementados do mesmo valor. A ativação da tarefa é representada por uma transição para o seu segundo estado (`Calc`). A esta transição, foi associada um conjunto *reset* cujo único elemento é o relógio $tmpExec$. O relógio $tmpExec$ irá contabilizar o tempo de execução da tarefa e poderá ser usado para a verificação do cumprimento de *deadlines*. O estado `Calc` representa o momento em que a tarefa verifica se a rota corrente está de acordo com a rota estabelecida para o cumprimento do objetivo passado ao UAV. A tarefa permanecerá neste estado, no máximo, o tempo especificado como seu tempo de execução no pior caso (`C_VRF`). Este requisito é descrito através da associação de uma invariante ao estado `Calc` que relaciona o tempo de execução, contabilizado pelo relógio $tmpExec$, ao atributo C_{vrf} (`tmpExec <= C_VRF`). Para evoluir do estado `Calc` para o estado `Fim`, a tarefa dispõe de duas transições. Na primeira delas não existe nenhuma ação associada, representando o fato da tarefa τ_{vrf} ter verificado que a rota em curso está correta e que nenhum ajuste precisa ser feito. Já a segunda transição possui, como ação associada, uma

operação de escrita no canal `controle`. Esta ação de sincronização descreve o fato de que a tarefa verificou que a rota atual difere do especificado e disparou uma mensagem de ativação para a tarefa τ_{ctl}, de forma que esta última possa estabelecer as novas metas para o SCA. Nas duas transições está presente uma condição guarda (`tmpExec >= c_VRF`). Esta condição guarda impede que o autômato evolua para o estado `Fim` antes que seja decorrido um tempo mínimo de execução (`c_VRF`). Isto visa impedir que a computação da tarefa possa ser interpretada como ocorrendo, por exemplo, em 0 (zero) unidades de tempo. O estado `Fim` representa o momento em que a tarefa já concluiu sua execução, mas ainda não está pronta para executar novamente, porque seu novo período ainda não iniciou. Esta condição é descrita pelo guarda associado à transição do estado `Fim` para o estado `Inicio` (`local >= T_VRF`). Ao ocorrer a transição para o estado `Inicio` o relógio $local$ é reiniciado e o ciclo recomeça.

Assim, considerando que o conjunto valoração dos relógios seja representado através da tupla $\langle c_{local}, c_{tmpExec} \rangle$, que os valores de período e de tempo máximo de execução da tarefa são aqueles definidos na seção anterior, e que o tempo mínimo de execução é igual a metade do tempo máximo, pode-se definir uma possível execução parcial da tarefa τ_{vrf}, como descrito abaixo:

$$(\texttt{Inicio}, <0,0>) \rightarrow (\texttt{Inicio}, <15,15>) \rightarrow (\texttt{Calc}, <15,0>) \rightarrow$$
$$(\texttt{Calc}, <45,30>) \overset{controle!}{\rightarrow} (\texttt{Fim}, <45,30>) \rightarrow (\texttt{Fim}, <165,150>) \rightarrow$$
$$(\texttt{Inicio}, <0,150>)$$

Esta execução descreve um ciclo completo da tarefa τ_{vrf}. Neste ciclo, a tarefa verificou a necessidade de ajuste de rota e ativou, através do canal `controle`, a tarefa τ_{ctl} para que esta proceda as ações necessárias.

3.3.2. Rede de Autômatos com Tempo

Uma rede de autômatos com tempo (*network of timed automata*) é a composição paralela de vários autômatos com tempo sincronizados entre si. O modelo de execução de uma rede de autômatos com tempo define dois modos de evolução: caso ocorra uma transição *delay*, todos os relógios de todos os autômatos são incrementados do mesmo valor, considerando que as condições invariantes dos estados atuais de cada autômato devem ser respeitadas; caso ocorra uma transição ação, apenas o autômato envolvido muda de estado e os relógios evoluem como descrito na seção 3.3.1. Caso esta ação envolva a sincronização de dois autômatos através de uma operação sobre canais, ambos os autômatos evoluirão para um novo estado. O conjunto *reset* será formado pela união do conjunto *reset* de cada transição envolvida e a transição só poderá acontecer caso as expressões guarda das duas transições sejam satisfeitas.

Como exemplo, considera-se os autômatos com tempo das tarefas τ_{vrf} e τ_{ctl} apresentados nas figuras 3.2 e 3.3. De acordo com a semântica descrita acima e considerando que o modelo do sistema é composto pela rede de autômatos das duas tarefas, uma execução parcial possível para o sistema (τ_{vrf},τ_{ctl}) é apresentada abaixo:

$$(\texttt{Inicio}, \texttt{Inicio}) \rightarrow (\texttt{Calc}, \texttt{Inicio}) \rightarrow (\texttt{Fim}, \texttt{Ajuste}) \rightarrow (\texttt{Fim}, \texttt{Fim})$$

Deve-se destacar nesta execução, a existência de uma condição de sincronização entre as duas tarefas, representada pela evolução simultânea dos seus estados ($(\texttt{Calc}, \texttt{Inicio}) \rightarrow (\texttt{Fim}, \texttt{Ajuste})$).

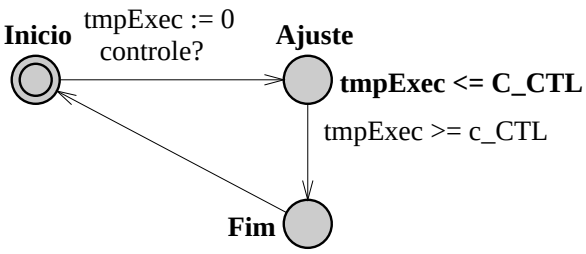

Figura 3.3: Modelagem da Tarefa τ_{ctl} do UAV.

3.3.3. Lógicas com Tempo

Uma vez que o sistema tenha sido modelado através de uma rede de autômatos com tempo, é possível especificar as propriedades desejadas para o sistema, de forma que a ferramenta de verificação possa verificar se o modelo descrito satisfaz estas propriedades. No UPPAAL, a linguagem de especificação é uma lógica de tempo denso denominada *Logic for Safety and Bounded Liveness Properties* ($\mathcal{L}_s$) [7], definida como um fragmento da lógica de tempo denso $\mathcal{L}_v$[58].

Em termos gerais, esta linguagem é composta pelos operadores:

- lógicos: `and`, `or`, `imply`, `not`
- temporais: $\Box$ (sempre), $\Diamond$ (futuramente), $\rightarrow$ (leva a)
- de caminho: `A` (para todo caminho), `E` (existe um caminho)

A sintaxe e semântica de ($\mathcal{L}_s$) estão definidas de acordo com o descrito na tabela 3.2.

Tabela 3.2: Sintaxe e semântica dos operadores temporais e de caminho de $\mathcal{L}_s$.

Prop.	Descrição
E$\Diamond$p	Existe ao menos um caminho onde p futuramente acontece (possivelmente p)
A$\Diamond$p	Para todo caminho, p futuramente acontece (futuramente p)
E$\Box$p	Existe ao menos um caminho onde p sempre acontece (potencialmente p)
A$\Box$p	Para todo caminho, p sempre acontece (invariavelmente p)
p$\rightarrow$q	Desde que p seja verdade q será futuramente verdade (p leva a q)

As variáveis `p` e `q` podem assumir os valores: o estado de algum dos autômatos (`nomeProcesso.estado`), uma operação de comparação de um número real com o valor de um dos relógios do autômato (`relogio` $\sim$ `valor`), uma fórmula da lógica proposicional (`p and q`, `p or q`, `not p` e `p imply q`) ou a palavra reservada `deadlock`, que representa o fato da rede de autômatos não mais poder evoluir sem que alguma restrição modelada seja violada. O símbolo $\sim$ representa um dos seguintes operadores: $=, !=, <, \leq, >, \geq$.

Abaixo apresenta-se a formalização de algumas propriedades em ($\mathcal{L}_s$):

- Em algum instante futuro, a tarefa τ_{vrf} será ativada (entrará no estado `Calc`): A$\Diamond$`VRF.Calc`

- O sistema do UAV nunca entra em *deadlock*:
  `A□not deadlock`
- Invariavelmente, caso a tarefa τ_{ctl} esteja no estado `Fim`, então o seu relógio de tempo de execução é no mínimo igual a constante `c_CTL`:
  `A□(CTL.Fim imply CTL.tmpExec≥c_CTL)`
- Em algum instante futuro, a tarefa τ_{ctl} será ativada (entrará no estado `Ajuste`):
  `A◊CTL.Ajuste`

O UPPAAL verifica as três primeiras propriedades como verdadeiras, enquanto que a última propriedade é verificada como falsa. Esta última propriedade não é satisfeita pelo modelo por existir um caminho computacional em que a mensagem de ativação `controle` nunca é enviada pela tarefa τ_{vrf}.

É importante observar que na modelagem apresentada foram considerados os aspetos temporais e de sincronização das tarefas, enquanto que outros aspectos foram abstraídos. Estes últimos aspectos podem ser considerados utilizando-se outros formalismos. Embora o modelo apresentado tenha sido bastante simples, os verificadores de modelos são utilizados na prática em aplicações reais [6, 38, 63]. Em muitos casos, no entanto, simplificações necessitam ser feitas para não haver uma explosão de estados e, conseqüentemente, tempos de resposta inaceitáveis.

3.4. Linguagens de Programação para Sistemas de Tempo-real

Um dos principais problemas na implementação de sistemas de tempo-real consiste em mapear os requisitos temporais das aplicações para o suporte de execução (*e.g.*, máquina virtual, *middleware*, sistema operacional). Um dos meios para facilitar essa tarefa consiste na adoção de linguagens de programação e ambientes de execução com características especiais capazes de traduzir e atender a esses requisitos.

Considerando linguagens de programação, é importante diferenciar os aspectos funcionais dos aspectos não-funcionais associados ao desenvolvimento de aplicações de tempo-real. Os aspectos funcionais comportam as abstrações fornecidas pela linguagem de programação para exprimir requisitos das aplicações. Por exemplo, algumas linguagens fornecem tipos de dados específicos, tais como filas de espera com diferentes semânticas de uso (bloqueantes, não-bloqueantes), blocos de código com restrições temporais de execução e mecanismos para tratamento de inversão de prioridades (vide seção 3.6.4 sobre o problema de inversão de prioridades). Nesta seção, descreveremos abstrações específicas para sistemas de tempo-real disponíveis em linguagens de programação dedicadas como Timber [11] e Alert [74], e extensões propostas para linguagens de programação de propósito geral, tais como Ada e Real-Time Java [12, 13]. Utilizaremos fragmentos de código para ilustrar o uso dessas abstrações.

Os aspectos não-funcionais relacionados às linguagens de programação envolvem elementos do suporte de execução, a exemplo do mecanismo de tratamento de exceções, que pode ser utilizado para o tratamento de falhas e os coletores de lixo (*garbage collectors*), cuja execução pode prejudicar o desempenho de aplicações de tempo-real. Apresentaremos uma visão geral das soluções existentes nessa área, explicando tais mecanismos a partir do exemplo do UAV (descrito na seção 3.2.5).

3.4.1. Abordagens de Desenvolvimento para Aplicações de Tempo-Real

Aplicações de tempo-real possuem requisitos de execução (notadamente, requisitos temporais) que são difíceis de serem expressos ou capturados por linguagens de programação de propósito geral, tais como C ou Pascal. Por exemplo, é comum em aplicações de tempo-real que certos trechos de código devam iniciar ou terminar sua execução em instantes precisos de tempo (conhecidos por tempo de ativação e *deadline*, respectivamente). Um método corriqueiro de implementação dessa situação consiste em associar um temporizador a esse trecho de código para ativar a aplicação no seu tempo de início, bem como verificar a possível violação do *deadline*.

A implementação de uma aplicação de tempo-real através dos mecanismos tradicionais fornecidos pelas linguagens de propósito geral é, via de regra, uma tarefa complexa devido à falta de abstrações adequadas, a exemplo de abstrações para manipulação de atributos temporais. Além disso, o código resultante tende a utilizar mecanismos específicos e de baixo nível do sistema operacional ou ambiente de execução. Estes problemas tendem a aumentar a complexidade de desenvolvimento e de manutenção do código. O desenvolvimento de abstrações mais expressivas é portanto um aspecto fundamental para o desenvolvimento de sistemas de tempo-real mais robustos, legíveis e de fácil manutenção.

Dentre os trabalhos que propõem novas abstrações voltadas à atender aos requisitos de aplicações de tempo-real, podemos destacar duas abordagens principais. A primeira abordagem consiste na extensão de uma linguagem de programação de propósito geral (*e.g.*, Java ou C) com novas abstrações e tipos de dados que permitam abranger o escopo de aplicações de tempo-real. Dois exemplos dessa abordagem são as linguagens Ada e Real-Time Java. A principal vantagem de utilizar uma linguagem com tais extensões é a existência de programadores capacitados na linguagem, o que dispensa investimentos adicionais na capacitação da equipe de desenvolvimento. Por outro lado, a linguagem permanece com suas características originais o que, em geral, dificulta a verificação de programas, por exemplo, quanto à coerência do uso de abstrações temporais. A segunda abordagem utiliza linguagens específicas que propõem novas abstrações para facilitar o desenvolvimento. Em geral, linguagens específicas são menos expressivas, mais simples e possuem menos abstrações do que linguagens de propósito geral. Tal abordagem facilita a verificação de propriedades relativas às restrições temporais e tende a reduzir a quantidade de código a ser desenvolvido, porém requer que os programadores aprendam uma nova linguagem. A decisão entre o uso de uma linguagem de programação específica ou uma linguagem de propósito geral com extensões depende ainda de fatores técnicos como a disponibilidade da linguagem para o ambiente em questão e compabatilidade com o hardware.

Nesta seção apresentaremos linguagens específicas e de propósito geral com extensões voltadas para o desenvolvimento de aplicações de tempo-real. Devido ao largo espectro de linguagens de programação disponíveis nesse âmbito, nos concentraremos nos aspectos mais importantes, tais como abstrações para a gestão de tempo e de recursos compartilhados - sem detalhar, contudo, aspectos sintáticos das linguagens. Descreveremos abstrações e exemplos de código referentes ao sistema de tempo-real do UAV (apresentado na seção 3.2.5) em Ada, Real-Time Java [12, 13], Timber [11], Pearl [1, 35]

e Alert [74]. Outras linguagens relevantes não apresentadas nessa seção incluem Esterel [15], Giotto [40], Lustre [36], Real-Time Euclid [49], HUME [37], Flex [46] e Cleopatra [9, 10]. Outra fonte importante de informações sobre linguagens de programação para sistemas de tempo-real é o livro de Burns e Wellings [22].

3.4.2. Gestão de Tempo

Um dos aspectos fundamentais na implementação de aplicações de tempo-real consiste na definição e manipulação de atributos temporais associados ao serviço fornecido pela aplicação como, por exemplo, *deadline*, tempo de ativação e periodicidade de execução. Para isso, é necessário que as aplicações possam consultar uma fonte precisa e confiável de tempo, que forneça diferentes níveis de granularidade (de horas até milisegundos). Além disso, aplicações de tempo-real devem poder retardar sua execução por um intervalo de tempo ou até um instante de tempo no futuro de modo a produzir resultados corretos na dimensão temporal.

O código da figura 3.4 ilustra a implementação deste tipo de situação em Ada por meio da instrução `delay until`. A instrução `delay until` *t* interrompe a execução do programa até o limite superior de tempo estabelecido pelo instante *t*, definindo, dessa forma, um atraso absoluto de tempo. No exemplo da figura 3.4, a seqüência de recolhimento do trem de pouso do UAV é executada, no mínimo, cinco segundos após o término da sua seqüência de decolagem. Caso o valor do relógio do sistema no instante da execução do comando `delay until` seja maior do `Hora_atual + 5.0`, a aplicação não é interrompida. Note que é possível que o trem de pouso seja recolhido muito tempo após os cinco segundos estabelecidos em virtude da concorrência com outros processos pelo processador ou por recursos compartilhados.

```
Hora_atual := Clock;  -- A função Clock retorna a hora atual
-- Procedimentos de decolagem
delay until Hora_atual + 5.0  -- Retardo de tempo absoluto
-- Recolhimento do trem de pouso
```

Figura 3.4: Disparo temporizado de comandos em Ada.

Outra situação recorrente em aplicações de tempo-real consiste em verificar se uma seqüência de comandos foi executada dentro de um limite de tempo. Isto permite evitar que um processo exceda o tempo de processamento que lhe foi concedido ou permaneça excessivamente em espera por um recurso ou mensagem vinda de outro processo. Em caso de estouro no limite de tempo, a aplicação pode efetuar medidas corretivas como, por exemplo, reenviar a mensagem, ou ir para um estado seguro, o que corresponde ao modo de operação *fail safe* discutido na seção 3.7.2, em caso de aplicações críticas. Em Ada, o programador pode facilmente associar limites de tempo de execução para o término de uma seqüência de comandos. No exemplo da figura 3.5 referente à tarefa τ_{gps}, caso a leitura dos dados do GPS exceda o valor da variável `deadline` (cujo valor inicial é de 100 milisegundos), a leitura será abortada e os comandos de tratamento relativo ao estouro de tempo (especificados pela cláusula `abort`) serão executados. A instrução `select` define opções de execução, podendo ser combinada com a instrução `accept`

(recebimento de mensagens) para definir *timeouts* associados ao envio de mensagens ou a uma chamada de procedimento.

```
select
        delay deadline
        -- Comandos a serem executados após estouro do limite de tempo
then abort
        -- Comandos com limite de tempo
            -- Leitura de dados do GPS
end select
```

Figura 3.5: Associação de limites de tempo para comandos da tarefa τ_{gps} em Ada.

Boa parte das aplicações de tempo-real possuem ao menos uma tarefa executando o mesmo conjunto de instruções periodicamente. Neste aspecto, existem diversas formas de se codificar uma tarefa periódica nas linguagens de programação existentes. O exemplo da figura 3.6 ilustra uma estratégia, apresentando o esqueleto da tarefa periódica τ_{gps} escrita em Ada. Neste exemplo, utilizamos alguns tipos de dados temporais da biblioteca `Ada.Real_Time` para definir a periodicidade (variável `Periodo`) e o tempo da próxima leitura dos dados do GPS (variável `TLeitura`). O corpo do programa se resume em um laço infinito (`loop`) que lê periodicamente os dados do GPS.

```
task body TGPS is
  TLeitura: Ada.Real_Time.Time := ...; -- tempo de início da leitura
  Periodo:    -- periodicidade de execução
     Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(100);
begin
    loop
        delay until TLeitura;
        -- Lê dados do GPS
       TLeitura := TLeitura + Periodo;
    end loop;
end TGPS;
```

Figura 3.6: Especificação da tarefa τ_{gps} em Ada.

Em Real-Time Java, a definição de atributos temporais para uma tarefa requer a utilização das classes `PeriodicParameters`, `SporadicParameters` e `AperiodicParameters` que definem parâmetros temporais para tarefas periódicas, esporádicas e aperiódicas, respectivamente. Após a definição do tipo de tarefa, o programador pode especificar outros parâmetros, a exemplo da prioridade de execução associada à tarefa, definida pela classe `PriorityParameters`. O trecho de código da figura 3.7 apresenta a definição da tarefa de verificação τ_{vrf} (objeto `Tvrf`) em Real-Time Java. Esta tarefa possui a maior prioridade de execução permitida, definida pela constante `MAX_PRIORITY`. Os parâmetros temporais, representados pelo objeto `pp`, definem a tarefa τ_{vrf} com uma periodicidade de 150 milisegundos, tempo de execução no pior caso de 40 milisegundos e *deadline* relativo de 120 milisegundos, entre outros. Esta tarefa

cria uma instância de um objeto da classe `TPeriodica`, que por sua vez é uma extensão da classe `RealtimeThread`. Pode-se ainda associar tratadores de exceção para eventos temporais, tais como, o estouro do tempo de execução no pior caso (`overrunHandler`) e para a violação de um *deadline* (`deadlineMissHandler`).

```
public class TPeriodica extends RealtimeThread {
 public TPeriodica (PriorityParameters p1,
                    PeriodicParameters p2) {...};
 public void run(){
   while (true) {
       // código da tarefa de verificação
      waitForNextPeriod();
   }
}
{PriorityParameters prio = new PriorityParameters (MAX_PRIORITY);
  PeriodicParameters pp = new PeriodicParameters (
     new RelativeTime (0,0),     // tempo de início (0 indica início imediato)
     new RelativeTime (150,0),   // período
     new RelativeTime (40,0),    // tempo de execução no pior caso
     new RelativeTime (120,0),   // deadline
    overrunHandler, deadlineMissHandler);  // tratadores de exceção
  TPeriodica Tvrf = new TPeriodica(prio, pp, ...);
  Tvrf.start();  // início da execução da tarefa
}
```

Figura 3.7: Especificação da tarefa τ_{vrf} em Real-Time Java.

A linguagem Timber fornece dois tipos de dados específicos para definição de atributos temporais : `TimeInstant` e `TimeDuration`. O tipo `TimeInstant` define um momento absoluto de tempo enquanto que o tipo `TimeDuration` especifica um intervalo de tempo entre duas variáveis `TimeInstant`. Para manipular uma variável definida a partir desses tipos de dados, o programador dispõe dos operadores `until`, `from`, `lasting` e `ending`. O trecho de código da figura 3.8 ilustra a utilização desses operadores. As variáveis `t1` e `t2` são instantes de tempo (cuja diferença é de cinqüenta milisegundos) enquanto que as variáveis `I1`, `I2`, `I3` e `I4` representam intervalos de tempo cujos valores são iguais após a execução do código. Outras abstrações que facilitam a manipulação de atributos temporais são os operadores `deadline`, que permite especificar o tempo máximo de término de uma ação; `baseline`, que determina o menor tempo em que uma tarefa pode ser iniciada e `timeline`, que denota o intervalo de tempo limitado entre um `baseline` e um `deadline`. Considerando o exemplo da figura 3.8, utilizamos os operadores `baseline` e `deadline` para definir novos instantes de tempo, definidos pelas variáveis `t5` e `t6`, a partir dos intervalos de tempo `I1` e `I2`.

Em Timber, é trivial implementar uma tarefa periódica. O trecho de código da figura 3.9 mostra o esqueleto da tarefa de controle de atitude τ_{atd} (`Tatd`), que se auto escalona com uma periodicidade de duzentos milisegundos. Convém notar que o comando `after` da figura 3.9 não especifica que a tarefa será bloqueada e novamente ativada *após* duzentos milisegundos e, sim, define que duzentos milisegundos após o início da execução do código da tarefa, esta será reativada. Na verdade, o comando

```
t2 = t1 + (50* milliseconds)
// os seguintes comandos são equivalentes
I1 = t1 ' until' t2
I2 = (50* milliseconds) ' from' t1
I3 =  t1 ' lasting' (50* milliseconds)
I4 = (50* milliseconds) ' ending' t2
// utilização de outros operadores temporais
t5 = ' baseline' I1    // t5 possui o mesmo valor que t1
t6 = ' deadline' I2    // t6 possui o mesmo valor que t2
```

Figura 3.8: Especificação e manipulação de atributos temporais em Timber.

`after` poderia aparecer em qualquer parte do código da tarefa (comandos após a instrução `action`) sem que houvesse alteração na semântica de execução do programa.

```
Tatd =  action
      // código do sistema de controle de atitude
      after (200* milliseconds)  Tatd  // reativação da tarefa
```

Figura 3.9: Especificação da tarefa periódica τ_{atd} em Timber.

A linguagem Alert possui o tipo de dado `time` que suporta operações tais como soma, diferenca, divisão e produto. Além disso, a variável de sistema `now` armazena o tempo atual. Uma tarefa em Alert contém uma parte de declarações e de código. As declarações definem variáveis globais, recursos utilizados (cláusula **uses**) e uma lista de propriedades (`properties`) associadas à tarefa. As propriedades de uma tarefa incluem os seguintes atributos: `criticality`, que define a importância da tarefa (*hard*, *soft* ou *non-realtime*), `activation` que define se a tarefa é periódica (*timedriven*) ou aperiódica (*eventdriven*), `interarrival` que especifica o intervalo mínimo entre duas ativações sucessivas da tarefa, `deadline` que denota um *deadline* relativo e `bandwidth` que especifica a fração de tempo do processador dedicada à execução da tarefa. A figura 3.10 apresenta a especificação em Alert da tarefa τ_{gps} (`Tgps`). Esta tarefa é periódica, de alta prioridade (*hard*) e utiliza um sensor de GPS, especificado pelo recurso `g` (definido mais adiante na figura 3.12), e cujo valor é periodicamente obtido através do método `le_gps`.

Em Alert, uma tarefa é iniciada ou finalizada através das instruções `activate` e `kill`, respectivamente. Outro ponto importante da linguagem é que o programador pode agrupar tarefas através da abstração `group` com o intuito, por exemplo, de começa-las em um tempo específico ou em instantes distintos. Neste último caso, o programador deve especificar para cada tarefa em questão o valor do deslocamento temporal relativo ao instante de ativação do grupo. Assim, em um grupo com cinco tarefas, pode-se ter três tarefas começando no tempo $t1$ e as outras duas tarefas iniciando nos tempos $t1 + 2$ segundos e $t1 + 15$ segundos, respectivamente.

A linguagem Pearl (*Process and Experiment Automation Realtime Language*) fornece duas instruções temporais básicas para o disparo de tarefas: `EVERY` e `WHEN`. Na figura 3.11, são apresentadas as condições de ativação das tarefas de verificação τ_{vrf} e

```
task Tgps {
    uses: sensor_gps g;
    properties:
         criticality : hard;      activation : timedriven;
         interarrival : 100ms;  deadline : 100ms;
         bandwidth : 0.1;
    code {
       int i = 1;
       body while (true) {
            g.le_gps(&i);
            ...
       }
    }
};
```

Figura 3.10: Especificação da tarefa τ_{gps} em Alert.

de controle τ_{ctl} do UAV, disparadas pelo comando ACTIVATE. A tarefa τ_{vrf} é ativada periodicamente a cada 150 milisegundos por meio da instrução EVERY, enquanto a tarefa τ_{ctl} é ativada esporadicamente através da instrução WHEN quando da detecção de um desvio de rota. No exemplo, considera-se que int_desvio_rota é uma interrupção gerada pela tarefa τ_{vrf}).

```
// tarefa periódica
 EVERY 0.15 SEC ACTIVATE tvrf
  ...
// tarefa esporádica ativada por uma interrupção
 WHEN int_desvio_rota ACTIVATE tctl
```

Figura 3.11: Condições de disparo das tarefas τ_{vrf} e τ_{ctl} em Pearl.

3.4.3. Gestão de Recursos Compartilhados

A previsibilidade de aplicações de tempo-real depende da gestão adequada dos recursos utilizados por tais aplicações (*e.g.,* arquivos, variáveis), principalmente no que concerne seu compartilhamento com aplicações não-críticas. Um exemplo de situação indesejável é conhecido como problema de inversão de prioridades (apresentado na seção 3.6.4). Este problema ocorre quando uma aplicação de maior importância têm sua execução excessivamente postergada ou impedida por uma aplicação de menor importância que detém a gestão de um recurso requerido por ambas as aplicações. Para solucionar ou minorar este tipo de anomalia, é importante que o desenvolvedor disponha de abstrações para definir ou selecionar políticas adequadas de gestão de recursos.

Em Alert, um recurso pode ser de dois tipos: crítico e não-crítico. Um recurso crítico, denominado rtresource, é gerenciado por um protocolo de concorrência implícito, enquanto que um recurso não-crítico requer o uso de mecanismos de sincronização. A sincronização de processos que utilizam um recurso crítico é implícita, ou seja, dispensa o programador do uso de semáforos, monitores ou variáveis de condição. Os

atributos associados a um recurso podem ser manipulados através de funções rotuladas com o tipo `entry`. Garante-se exclusão mútua na execução dessas funções caso elas tenham sido definidas como `protected`. O exemplo da figura 3.12 declara um recurso crítico para o controlador do sensor de velocidade descrito na figura 3.10. A linguagem define certas restrições semânticas quanto ao uso de recursos. Por exemplo, aplicações críticas só podem fazer uso de recursos do tipo `rtresource`. Da mesma forma, sugere-se que aplicações não-críticas não utilizem recursos do tipo `rtresource`.

```
rtresource sensor_gps {
    double valor_atual;
    void atualizaValor( double);
    entry protected double le_gps() {
          double d;
          d = valor_atual;
          return d;
    };
}
```

Figura 3.12: Definição de um sensor GPS como recurso crítico em Alert.

Filas de Espera

Uma das formas de reduzir a latência gerada pela espera de recursos compartilhados consiste em utilizar filas de espera com diferentes semânticas de bloqueio quando da execução de operações de leitura e escrita. Isso permite, por exemplo, que a aplicação saiba imediatamente se um recurso está disponível. Dessa forma, uma aplicação crítica pode decidir em continuar sua execução sem o recurso requisitado (caso este não seja estritamente necessário) ao invés de esperar até que o mesmo seja liberado por outra aplicação. Real-Time Java, por exemplo, define três tipos de filas. A fila `WaitFreeWriteQueue` permite que as tarefas críticas efetuem uma operação de escrita na fila de forma não-bloqueante através do método `write()`, sendo que as tarefas não-críticas podem ser bloqueadas ao efetuar uma operação de leitura através do método `read()` na mesma fila. A fila `WaitFreeReadQueue` permite que as tarefas críticas efetuem uma operação de leitura na fila de forma não-bloqueante através do método `read()` e as tarefas não-críticas podem ser bloqueadas ao efetuar uma operação de escrita utilizando o método `write()` na mesma fila. A fila `WaitFreeDequeue`, por sua vez, encapsula as características das duas filas descritas anteriores.

Protocolos de Gestão de Recursos

Uma abordagem comum para evitar anomalias relativas à gestão de recursos compartilhados, a exemplo do problema de inversão de prioridades, consiste em utilizar um protocolo de gestão de recursos. Duas abordagens largamente utilizadas para minorar o problema de inversão de prioridades são o protocolo de prioridade teto (*priority-ceiling*) e o protocolo

de herança de prioridade [84](vide seção 3.6.4). Nesta seção, descreveremos algumas linguagens que permitem especificar o uso de tais estratégias.

Em Ada, a utilização de recursos é regida por uma política de prioridades que estabelece algumas regras básicas de herança de prioridades relativas à tarefas e recursos, descritas a seguir. Quando da criação de uma nova tarefa, a nova tarefa herda a prioridade da tarefa que a criou. Ao receber uma mensagem (*e.g.*, através primitiva `accept`), uma tarefa servidora recebe imediatamente a prioridade do cliente.

Quanto à utilização de recursos, Ada permite associar uma prioridade a um recurso rotulado como protegido (`Protected`). Ao utilizar um recurso protegido, uma tarefa herda a prioridade deste recurso. Para ilustrar esse último caso, a figura 3.13 descreve a especificação em Ada da variável `altitude` como um recurso protegido com valor de prioridade 10.

```
Protected altitude is
    ...
  pragma Priority(10);
end altitude;
```

Figura 3.13: Definição de altitude como recurso protegido em Ada.

Os parâmetros de execução associados a uma tarefa Ada são especificados através de uma declaração do tipo `pragma`. O exemplo da figura 3.14 especifica que as tarefas são gerenciadas por uma política de escalonamento do tipo FIFO (1) e a utilização de um protocolo de prioridade teto (2) para a gestão de recursos.

```
pragma Task_Dispatching_Policy (FIFO_Within_Priority); --(1)
pragma Locking_Policy (Ceiling_Locking);   --(2)
```

Figura 3.14: Parâmetros de gestão de recursos em Ada.

Real-Time Java também permite a utilização e configuração de protocolos de prioridade teto e de herança de prioridade. A classe `MonitorControl` é uma super-classe abstrata de todas as classes que implementam uma política de controle de inversão de prioridades e fornece métodos que permitem parametrizar a política *default*. A classe `PriorityInheritance` implementa o algoritmo de herança de prioridades enquanto a classe `PriorityCeilingEmulation` implementa o algoritmo de prioridade teto.

3.4.4. Gestão de Memória

Alguns aspectos de implementação do ambiente de execução de linguagens de programação interferem diretamente na previsibilidade de aplicações críticas. Por exemplo, em ambientes baseados na máquina virtual Java, duas fontes consideráveis de imprevisibilidade devem-se principalmente à carga dinâmica de classes e à gestão automática de memória. De fato, em grande parte das implementações de linguagens orientadas a objeto atuais, não é necessário que as aplicações gerenciem a memória explicitamente. Assim, o programador é dispensado de utilizar funções para solicitar e liberar memória

para objetos. Tal abordagem facilita o desenvolvimento, pois libera o programador desta tarefa deixando-a a cargo do sistema de execução da linguagem. O componente responsável por esse gerenciamento de memória é conhecido como "coletor de lixo" (do inglês, *garbage collector*). Entretanto, a execução esporádica do coletor de lixo pode comprometer a previsibilidade do sistema, pois introduz uma latência de processamento que pode ser extremamente danosa para as aplicações críticas. Uma das soluções possíveis consiste em limitar a execução do coletor de lixo, restringindo sua execução à períodos de inatividade ou então à intervalos de tempo em que inexista risco de violação de *deadlines*. Real-Time Java permite a definição de escopos de memória (*scoped memory*) cuja gestão não sofre interferência do coletor de lixo. Tal funcionalidade é implementada pela classe `HeapMemory`. Além disso, o programador pode utilizar uma *thread* do tipo `NoHeapRealTimeThread` cuja importância é sempre maior do que a importância da tarefa coletora de lixo.

O *swapping* freqüente de páginas de memória de um processo é outra causa importante de imprevisibilidade para aplicações de tempo-real. Uma forma de evitar esta situação consiste em bloquear as páginas dessas aplicações na memória principal para evitar que as mesmas sejam substituídas e, em seguida, armazenadas em disco. Outra estratégia, menos ortodoxa, consiste em desligar o sistema de memória virtual do sistema operacional, deixando o gerenciamento de memória sob a responsabilidade das aplicações. Esta técnica é utilizada por certas aplicações críticas desenvolvidas para sistemas embarcados.

3.4.5. Verificação de Código

Devido a criticidade associada às tarefas que compõem um sistema de tempo-real, é essencial garantir a correção das mesmas antes do seu uso efetivo, por exemplo, através das técnicas de verificação formal apresentadas na seção 3.3. Uma forma complementar de verificação consiste em especificar propriedades associadas às aplicações de tempo-real através da sintaxe e semântica das abstrações fornecidas pela linguagem. Por exemplo, na linguagem Flex é possível definir blocos com restrições temporais, chamados de *constraint blocks*. Tais blocos podem ser especificados por uma expressão booleana definindo, neste caso, um invariante que deve ser preservado durante a execução do bloco ou através de uma restrição temporal estabelecendo o início e fim da execução do bloco. Dado um programa com blocos com tais restrições, o compilador verifica se existem incoerências entre as definições dos blocos (por exemplo, dois blocos de código que devem ser executados um após o outro).

Por fim, é desejável que os programas de tempo-real sejam passíveis de verificação em tempo de compilação para certificar que suas restrições temporais serão cumpridas antes de sua execução. Duas restrições que auxiliam nessa tarefa são a eliminação de laços infinitos e funções recursivas da sintaxe da linguagem. Uma linguagem específica para o desenvolvimento de aplicações de tempo-real poderia igualmente facilitar a análise de escalonabilidade (objeto da seção 3.6.2) e o cálculo ou estimativa do tempo de execução no pior caso.

3.5. *Middleware* para Sistemas de Tempo-Real

Em geral, o termo *middleware* representa uma camada de software que estende as funcionalidades de um sistema operacional, situando-se entre este e aplicação. Nesta seção serão apresentados alguns conceitos e objetivos de *middleware*, notadamente relativos ao seu uso como suporte à computação distribuída e à especificação de parâmetros temporais e garantia da previsibilidade em sistemas de tempo-real.

Em tempos passados, os sistemas de tempo-real eram construídos visando essencialmente o atendimento dos requisitos funcionais e temporais da aplicação, em ambientes convencionais. À medida em que estes sistemas se tornam mais complexos, ao incorporarem características envolvendo distribuição, modularidade e tecnologias embarcadas, a utilização de metodologias e técnicas para gerência da complexidade do processo de desenvolvimento se torna indispensável.

Conforme apresentado a seguir, o uso de *middleware* é adequado quando buscam-se padrões para inter-operabilidade, escalabilidade e gerência da complexidade através da utilização de abstrações e serviços. O objetivo desta seção é apresentar como o *middleware* pode ser aplicado na construção de sistemas distribuídos de tempo-real e como esta abordagem favorece o desenvolvimento de aplicações flexíveis, modulares e confiáveis.

3.5.1. Conceitos e Categorias de *Middleware*

A palavra *middleware* é atualmente utilizada em uma série de situações, nos mais diversos contextos e com os mais diversos objetivos. O que pode-se notar é que, desde o final da década de 80 até os dias atuais, a idéia do *middleware* passou por algumas mudanças, de modo a refletir os objetivos e idéias emergentes.

Um tipo de *middleware* freqüentemente utilizado é o orientado a mensagens (*Message-Oriented Middleware* - MOM). Nesta situação, o papel do *middleware* se aproxima ao trabalho de um repositório, onde mensagens podem ser armazenadas e recuperadas. Este *middleware* tem a função básica de prover um mecanismo para comunicação assíncrona entre aplicações. Os principais produtos que seguem a linha dos MOM's são o MQSeries (IBM) [71], o MSMQ (Microsoft) [72] e o SmartSockets (Talarian) [85].

A chamada de procedimentos remotos (*Remote Procedure Call* - RPC) foi uma das primeiras tentativas de definição de um *middleware* que oferecesse mecanismos de abstração de modo a gerenciar a complexidade do desenvolvimento de sistemas distribuídos. Divulgado em 1982 pela Sun Microsystems [69], esse tipo de *middleware* introduz uma série de ferramentas e definições que possibilitam a execução de procedimentos remotos com a mesma simplicidade e facilidade da execução de procedimentos locais. Uma camada de software disponibilizada pelo *middleware* realiza todo o procedimento de empacotamento, envio e desempacotamento de requisições via canal de comunicação.

Uma evolução natural do *middleware* baseado em RPC foi a utilização do ORB (*Object Request Broker*), introduzindo as vantagens da orientação a objetos na chamada de execuções remotas. Neste ponto, o conceito de *middleware* muda bastante, saindo da idéia de camada de tradução ou comunicação para a concepção de uma biblioteca de suporte a execuções remotas. Além disso, não somente as questões de comunicação em

rede foram abordadas, mas passou-se a incluir também a disponibilização de serviços tais como segurança, controle de acesso, transações distribuídas e balanceamento de carga.

A partir dos exemplos acima pode-se concluir que o *middleware* é uma camada de software que possui como objetivo a disponibilização de serviços que facilitam o trabalho do desenvolvedor. Esses serviços podem estar relacionados a diversos requisitos de aplicações: distribuição, armazenamento, tolerância a falhas, balanceamento de carga, dentre outras. Vale ressaltar que, enquanto as linguagens de programação para tempo-real , apresentadas na seção 3.4, enfocam mecanismos para melhorar a expressividade da linguagem em relação ao caráter temporal, os *middlewares* buscam também uma melhor gerência do ambiente de execução através da disponibilização de serviços.

3.5.2. Abordagens de *Middleware* para Sistemas de Tempo-Real

O desenvolvimento de sistemas de tempo-real tem sido realizado, em sua maioria, através de técnicas *ad-hoc*, com uma fundamentação incipiente quanto à metodologias e padrões. Isso ocorre devido às constantes restrições em relação ao desempenho, consumo de energia e uso de memória [78, 79]. Consequentemente, os produtos obtidos são sistemas difíceis de evoluir e manter, devido ao grau de especialização empregado no decorrer do processo de desenvolvimento, características estas indesejáveis em uma área onde um mercado altamente competitivo demanda soluções cada vez mais rápidas e de qualidade.

Além dos problemas causados pelas restrições acima citadas, os sistemas de tempo-real vêm cada vez mais adquirindo características distribuídas, o que contribui para a complexidade do sistema [48]. A existência de um suporte conceitual já consolidado e validado para sistemas distribuídos convencionais fez com que o conceito de *middleware* passasse também a ser aplicado no desenvolvimento de sistemas de tempo-real.

A partir da seção seguinte serão apresentadas três arquiteturas de *middleware* para suporte ao desenvolvimento de sistemas de tempo-real. Serão analisados os objetivos, serviços oferecidos e aplicações utilizando o exemplo ilustrativo apresentado na seção 3.2.5.

3.5.3. OSA+

O OSA+ (**O***pen* **S***ystem* **A***rchitecture* - **Pl***atform for* **U***niversal* **S***ervices*) [17] é um *middleware* projetado para o desenvolvimento de sistemas de tempo-real embarcados, facilitando o desenvolvimento de aplicações distribuídas de tempo-real.

A plataforma OSA+ é formada por um *kernel* bastante reduzido que oferece funcionalidades básicas. Esse *kernel* não contém partes dependentes do hardware ou do sistema operacional. Ao invés, o *kernel* faz uso de serviços especiais para construir sua própria funcionalidade. Conforme ilustrado na figura 3.15, estes serviços são chamados de serviços básicos e servem para realizar a adaptação do *kernel* a um ambiente específico de hardware e sistema operacional. Além dos serviços básicos, o *kernel* possibilita a definição de serviços da aplicação (indicados na figura como Serviço 1 e Serviço 2) e disponibiliza uma interface de usuário, utilizada para monitoramento dos serviços.

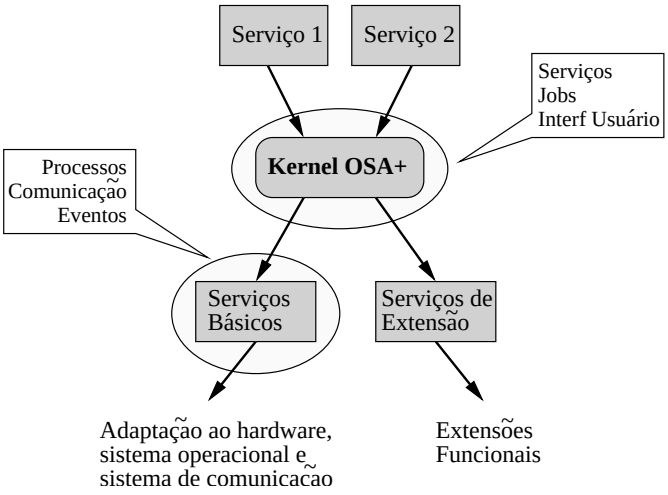

Figura 3.15: Arquitetura do OSA+.

Os principais conceitos da arquitetura OSA+ estão relacionados a serviços que se comunicam através de tarefas que executam ações e retornam valores. [16, 18].

Serviços são criados através da seguinte função:

```
osaStartService ( void (* Proc)(), osaProcType type,
                  osaConfigType* config,  char* name);
```

O parâmetro `type` define o modelo de execução utilizado cujos valores possíveis são: procedimento (função executada no mesmo espaço de endereçamento do processo inicial), processo leve (função executada em uma nova *thread*) ou processo convencional (função executada em um novo processo). O valor de `config` ajusta parâmetros adicionais de configuração, tais como as restrições temporais.

A principal idéia por trás do OSA+ é utilizar a troca de tarefas e valores de retorno como uma maneira padrão para informar condições temporais de serviços, tais como *deadline* relativo, periodicidade, etc. Se nenhuma condição temporal é especificada, todas as tarefas são tratadas com a mesma prioridade, usando uma abordagem FIFO. Os parâmetros temporais do serviço são representados por uma estrutura com os seguintes campos:

```
typedef struct {
      struct {
            osaTime deliver_at, deliver_tolerance, deliver_period;
            osaLong deliver_mult;
      } order_mode;
      union {
            struct {
                  osaTime deadline;
                  osaShort critically;
            } realtime;
            struct {
                  osaShort priority;
                  osaTime timeslice;
            } realtime_priority;
            struct {
                  osaShort cpu_load;
            } realtime_load;
      } service_mode;
} osaMode;
```

A estrutura `order_mode` define as restrições do canal de comunicação ao passo que a estrutura `service_mode` define as restrições da execução do serviço. `deliver_at` e `deliver_tolerance` especificam o intervalo no qual a mensagem deve ser entregue. Se este tempo é excedido, a mensagem é retornada para o emissor juntamente com uma mensagem de erro. O atributo `deliver_period` especifica o intervalo de tempo depois do qual a mensagem deve ser enviada novamente e `deadline` contém o último instante de tempo no qual a resposta da mensagem deve ser entregue. `critically` informa o que fazer se um *deadline* expira (*soft*, *firm* ou *hard*). `priority` é uma forma alternativa de especificação da restrição temporal atraves de prioridades. `cpu_load` contém a porcentagem de utilização da CPU requerida pelo serviço, permitindo que a execução do serviço seja independente da carga atual da CPU. Com este mecanismo, serviços podem ser controlados através de tarefas e resultados, permitindo uma maior flexibilidade e configuração em tempo de execução.

O OSA+ pode garantir desempenho em tempo-real de suas operações somente se todas as camadas subjacentes são capazes de exercer suas funções em tempo-real (ver seção 3.6 para discussão sobre suporte básico). O hardware deve garantir tempos de execução no pior caso para todas as operações. O sistema operacional também deve ter suporte de tempo-real, disponibilizando alguma política de escalonamento como prioridades fixas ou *earliest deadline first* (EDF - vide seção 3.6). E, por fim, os protocolos e meios de comunicação devem garantir entrega de mensagens também no pior caso.

3.5.4. TMOSM

Nos últimos anos várias tentativas foram realizadas no intuito de estender o *middleware* convencional para programação distribuída de modo a permitir a especificação de requisitos de ação temporal. Uma dessas iniciativas foi a abordagem denominada *Time-Triggered Message-Triggered Object* (TMO) [47], desenvolvido a partir de 1994. O TMO estende o modelo convencional de objetos, definindo uma estruturação sintática e semântica para a especificação dos requisitos temporais.

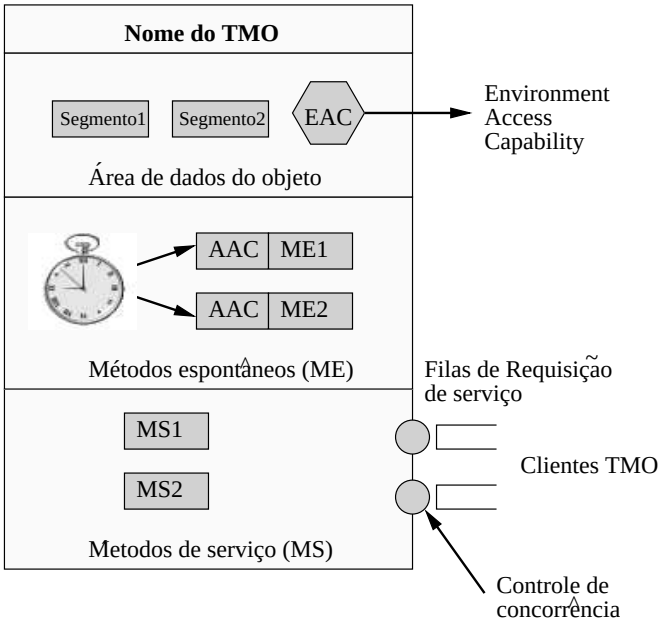

Figura 3.16: Estrutura básica de um TMO.

O *TMO Support Middleware* (TMOSM) é a implementação que suporta o conceito do TMO. As primeiras implementações do TMOSM foram realizadas utilizando um sistema operacional de tempo-real denominado *DREAM Kernel*, porém atualmente os autores têm direcionado os trabalhos para a construção de um *middleware* de tempo-real que funcione sobre plataformas convencionais ou *commercial-off-the-shelf* (COTS). Atualmente existe uma versão disponível para a plataforma Windows NT [47].

As principais características desse *middleware* são: comunicação baseada em mensagens UDP de *broadcast*, concorrência e controle realizados através de técnicas não-bloqueantes e escalonamento de tarefas realizado em dois níveis.

A figura 3.16 ilustra a estrutura básica de um TMO. Algumas das suas características são:

- **Definição de atributos e recursos para comunicação**: cada TMO possui uma área de dados onde atributos podem ser criados sob a forma de segmentos. O *Environment Access Capability* (EAC) oferece mecanismos para acessar o ambiente de comunicação em rede, dispositivos de I/O e outros TMO's.
- **Definição de métodos espontâneos**: o TMO contém um tipo de método chamado *time-triggered method* (TM) ou **métodos espontâneos**, em contrapartida aos métodos convencionais, aqui chamados de **métodos de serviço**. A execução de métodos espontâneos é disparada em tempos pré-definidos durante o projeto, ao passo que métodos de serviço são executados a partir de requisições dos clientes.
- **Restrição básica de concorrência (***Basic Concurrency Constraint* **- BCC)**: métodos de serviço não podem "perturbar" a execução de métodos espontâneos. Basicamente, a ativação de um método de serviço soliticitado por algum cliente é realizada somente quando não existem execuções potenciais de métodos espontâneos.

- **Definição e garantia de restrições temporais**: para cada método de um TMO é especificada uma janela de tempo na qual a execução deve iniciar e terminar, indicado na figura 3.16 com a sigla ACC, a ser descrita a seguir.

Os tempos de execução de métodos espontâneos devem ser especificados como constantes durante o projeto do sistema. Eles aparecem na primeira cláusula da especificação dos métodos e são chamados de **condição de ativação autônoma** (*Autonomous Activation Condition* - AAC):

```
for <time-var> =
from <activation-time> to <deactivation-time> [ every <period> ]
     start-during ( <earliest-start-time>, <latest-start-time>)
                  finish-by <deadline>
```

Por exemplo, no sistema do *Unmanned Air Vehicle* (UAV), poderia-se definir a condição de ativação autônoma da tarefa τ_{vrf} como:

```
for t = from 0am to 11:59pm every 150ms
     start-during (t, t+5ms) finish-by t+150ms
```

A expressão acima especifica que a tarefa τ_{vrf} deve ser executada a cada 150 milisegundos, initerruptamente (das 0am as 11:59pm). Cada execução deve iniciar em qualquer tempo dentro do intervalo (t, t+5*ms*) e deve ser completada até t+150 milisegundos.

3.5.5. TAO

Muitas aplicações de tempo-real podem se beneficiar de arquiteturas flexíveis e abertas para computação distribuída, tais como aquelas definidas pela especificação CORBA (*Common Object Request Broker Architecture*) [70]. O CORBA é um padrão emergente para computação distribuída baseada em objetos. Este tipo de *middleware* reside entre o cliente e o servidor promovendo uma visão uniforme da heterogeneidade da rede, das camadas do sistema operacional e da linguagem de programação.

Os componentes principais desse *middleware* são os ORB's (*Object Request Broker*). Os ORB's têm a função de eliminar tarefas tediosas e sujeitas a erros durante o desenvolvimento e manutenção de aplicações distribuídas. Em particular, ORB's automatizam tarefas comuns tais como localização e ativação de objetos, empacotamento e desempacotamento de parâmetros, tolerância a falhas (vide seção 3.7) e segurança.

Embora o CORBA seja bem adequado para aplicações convencionais, as suas implementações não são adequadas para aplicações de tempo-real devido à falta de características e garantias para qualidade de serviço. O TAO (The Ace ORB) [80] é uma implementação baseada em CORBA, com suporte de tempo-real e desenvolvido a partir de 1997 na Universidade de Washington.

Dentre os requisitos e características de uma implementação de um ORB, necessários para aplicações de tempo-real, destacam-se: políticas e mecanismos para especificação de requisitos de qualidade de serviço, garantia desta qualidade de serviço através de recursos do sistema operacional e do sub-sistema de comunicação, protocolos de comunicação eficientes e previsíveis, demultiplexação e *dispatching* eficientes e previsíveis, *stubs* e *skeletons* eficientes e previsíveis e gerenciamento de memória eficiente e previsível através da minimização de cópias de dados e alocações dinâmicas.

Estes objetivos são alcançados no TAO através de uma análise detalhada de cada componente da arquitetura CORBA. Nos testes de desempenho realizados [80], o subsistema de comunicação foi composto basicamente de interfaces ATM de alta velocidade (2.4 Gbps) e desenvolvido em camadas de modo a ser portável para outras arquiteturas de rede. A previsibilidade em relação aos ORB's foi atingida através da substituição do protocolo IIOP (*Internet Inter-ORB Protocol*) por um protocolo que permite a especificação de parâmetros de qualidade de serviço. Este protocolo chamou-se RIOP (*Real-Time Inter-Orb Protocol*) e faz uso de campos já existentes e não utilizados do IIOP para enviar informações de controle da previsibilidade.

A obtenção de *stubs* e *skeletons* previsíveis se torna possível através da otimização em relação às cópias de dados, alocações dinâmicas e substituição de chamadas excessivas de funções por execuções *in line*.

A especificação e garantia das restrições temporais no TAO é suportada através do *Scheduling Service* (ver seção 3.6.1 para detalhes sobre escalonamento). Este serviço é responsável pela alocação de recursos de CPU de modo a satisfazer as necessidades temporais das aplicações. O *Scheduling Service* é definido como um objeto CORBA, ou seja, como uma implementação de uma interface IDL. Esta abordagem permite que o serviço seja acessado ou local ou remotamente sem implicações para os clientes.

O *Scheduling Service* tem duas principais atividades, uma realizada *off-line* e outra *on-line*:

- análise *off-line* da escalonabilidade: o *Scheduling Service* realiza um teste *off-line* de escalonabilidade para toda operação de IDL registrada no serviço. Este teste avalia se existem recursos de CPU suficientes para realizar todas as operações críticas (ver detalhes na seção 3.6.2).
- consulta *on-line* de prioridades de requisições: em tempo de execução, o *Scheduling Service* disponibiliza uma interface que permite ao ORB a consulta de prioridades atribuidas pelo teste de escalonabilidade.

Para a especificação dos requisitos de qualidade de serviço, a linguagem de definição de interfaces original do CORBA foi estendida, dando origem à RIDL (*Real-Time Interface Definition Language*). Através desta nova linguagem, programadores podem especificar os requisitos temporais de suas aplicações sob a forma de *deadlines*, periodicidade das tarefas, tempos de execução no pior caso etc. A especificação dos requisitos temporais é realizada através da utilização da interface *RT_Task* e da estrutura *RT_Info*, ambas definidas pelo TAO, conforme a seguir:

```
interface RT_Task {
     typedef Time::TimeT TimeT;
     typedef Time::TimeT PeriodT;
     enum Task_Priority { INTERRUPT, IO_SERVICE, CRITICAL,
          HARD_DEADLINE, BACKGROUND}; // From highest to lowest.
     struct RT_Info {
          TimeT worst_case_execution_time_
          TimeT typical_execution_time_;
          PeriodT period_;
          Task_Priority priority;
          TimeT quantum;
          sequence < RT_Info > task_dependencies_;
     };
};
```

Componentes que requerem garantias temporais podem especificar o QoS desejado herdando a interface *RT_Task* e especificando os atributos da estrutura *RT_Info*. Por exemplo, poderia-se especificar a tarefa τ_{vrf} do sistema do UAV da seguinte maneira:

```
module UAV {
     struct GPS_position {
          float latitude;
          float longitude;
     };
     interface SCN : RT_Task {
          boolean vrf ( in GPS_position gpsPos);
          ...
     };
};
```

Na implementação do objeto distribuído, os campos da estrutura *RT_Info* são preenchidos informando os requisitos temporais da operação.

3.6. Suporte Básico para Sistemas Computacionais de Tempo-Real

O suporte básico para sistemas de tempo-real está concentrado no nível do sistema operacional. Aspectos que vão desde a definição da arquitetura (*e.g., micro-kernel*) até questões de baixo nível (*e.g.,* rotinas de acesso ao relógio, gerenciamento de interrupções) são cobertos por esse tema. A ênfase desta seção está concentrada naqueles aspectos mais relacionados à determinação da previsibilidade temporal em sistemas de tempo-real. Mais especificamente, trataremos do problema de *escalonamento* de tarefas.

Escalonar tarefas significa distribuir os recursos computacionais necessários para que o sistema funcione respeitando suas restrições temporais. De forma geral, isso envolve *alocar* tarefas nos processadores disponíveis (caso o sistema seja multiprocessado) e, em cada processador, escalonar as tarefas neles alocadas de forma que seus *deadlines* sejam observados. Nota-se então dois sub-problemas em questão: a *alocação* e o *escalonamento* propriamente dito.

Devido a complexidade envolvida, os problemas de escalonamento e alocação são tratados preferivelmente de forma separada nos sistemas de tempo-real críticos. Inicialmente, decisões de alocações de tarefas são tomadas estaticamente [65], ou seja, durante

o projeto do sistema. Então o problema de escalonamento é considerado. Essa estratégia fornece várias vantagens. Primeiro, a complexidade do problema é reduzida. Segundo, algoritmos de otimização podem ser usados para resolver o problema de alocação de forma a considerar diversos fatores, tais como minimização dos custos de comunicação entre as tarefas, balanceamento (estático) de carga etc. [88]. Terceiro, o problema de escalonamento pode ser tratado tal como num sistema uniprocessado, onde heurísticas ótimas e eficientes podem ser usadas. Por esses motivos, adotaremos também essa abordagem aqui. Em outras palavras, consideraremos que o problema de alocação é resolvido estaticamente e nos concentraremos no problema de escalonamento considerando apenas um processador (seção 3.6.1).

Uma vez que a política de escalonamento é conhecida, deve-se analisar o sistema para saber se suas restrições temporais serão respeitadas. Isso é feito durante a análise de escalonabilidade do sistema. Por exemplo, se o sistema é crítico, ao analisar sua escalonabilidade, deseja-se extrair garantias de que todas as suas tarefas cumprirão seus *deadlines*. Tal tema será tratado na seção 3.6.2.

Um dos fatores complicadores, tanto para o escalonamento quanto para a análise de escalonabilidade, é a interação entre tarefas. De fato, tarefas podem requerer recursos compartilhados ou se comunicar através de trocas de mensagens, por exemplo. Devido às suas interações, tarefas podem ficar esperando pelo processamento de outras, o que pode influenciar na sua previsibilidade. Na seção 3.6.4 esse problema será considerado.

Apesar de tratar a alocação de tarefas estaticamente, quando o sistema é distribuído, o sub-sistema de comunicação pode interferir no comportamento temporal das tarefas, visto que a rede de comunicação é um recurso compartilhado, onde cada tarefa deve ter acesso exclusivo durante a transmissão de suas mensagens. Seção 3.6.5 abordará esse aspecto de forma suscinta.

3.6.1. Escalonamento de Tarefas e Garantias de Previsibilidade

As abordagens de escalonamento preocupam-se em definir tanto as políticas de escolha de qual tarefa será executada em função do tempo (geração do escalonamento) quanto as técnicas para analisar o comportamento temporal do sistema (análise de escalonabilidade). A geração do escalonamento e a análise podem ser realizadas tanto em tempo de projeto quanto em tempo de execução. Pode-se, assim, dividir as abordagens de escalonamento em quatro grupos [60] (ver tabela 3.3), a depender do *momento* em que as decisões de escalonamento são tomadas e do momento em que a análise de escalonabilidade é realizada.

Tabela 3.3: Abordagens para escalonamento de tarefas.

Análise de Escalonabilidade	Escalonamento	
	Off-line	*On-line*
Off-line	(1) Escalonabilidade garantida Baixíssima flexibilidade	(2) Escalonabilidade garantida Média flexibilidade
On-line	(3) Escalonabilidade garantida Baixa flexibilidade	(4) Escalonabilidade não garantida Alta Flexibilidade

Na abordagem *off-line*, célula (1) da tabela, tanto as decisões de escalonamento quanto a análise são realizadas durante o projeto do sistema. Durante a execução o escalonador simplesmente segue a tabela de escalonamento gerada. Esse tipo de escalonamento é geralmente empregado em sistemas nos quais a ativação das tarefas é baseada no tempo (*time-triggered*), pois pode-se controlar o momento em que os eventos são percebidos pelo sistema. O principal argumento a favor desta abordagem é seu alto nível de previsibilidade [50, 51]. No entanto, tal argumentação é questionável [55], pois essa previsibilidade é conseguida através de um simplificado modelo de tarefas e de fortes hipóteses temporais, o que torna o sistema inflexível. Essa inflexibilidade pode ser reduzida se for possível fazer alguns ajustes em tempo de execução do escalonamento gerado *off-line*. Tal abordagem é pouco comum, mas pode ser encontrada [45].

As abordagens *on-line*, representadas na célula (4) da tabela, são as mais flexíveis, pois tanto as decisões sobre o escalonamento quanto a análise de escalonabilidade são realizadas em tempo de execução. O sistema pode assim se adaptar, durante sua execução, à ocorrência de eventos do ambiente. Na verdade, a análise nesse caso é conhecida como *teste de aceitação*. Esse teste verifica, para cada tarefa que é ativada no sistema, se o cumprimento de algum *deadline* pode ser comprometido. Se esse for o caso, a tarefa é rejeitada pelo escalonador. Caso contrário, ela é admitida. Como pode ser notado, políticas de escalonamento pertencentes a essa classe não podem garantir que todos os *deadlines* sejam cumpridos, pois, ao rejeitar tarefas, pode-se violar seus *deadlines*. Por essa razão, geralmente, são usadas para escalonar tarefas não-críticas.

Uma solução de compromisso entre flexibilidade e previsibilidade é conseguida pelas abordagens pertencentes à célula (3) da tabela, onde as mais aceitas e usadas são baseadas no uso de prioridades. Prioridade passa a representar urgência na execução e fornece um critério para o escalonador, em tempo de execução, escolher qual tarefa executar. Se as prioridades são determinadas em tempo de execução, diz-se que o escalonamento é dinâmico, ou baseado em prioridades variáveis. Caso as prioridades sejam definidas durante a fase de projeto, o escalonamento é dito estático ou baseado em prioridades fixas. Como será visto nas seções a seguir, escalonadores baseados em prioridades fixas são vantajosos em relação aos baseados em prioridades variáveis quando o critério de comparação é previsibilidade. Por essa razão, tal abordagem será a que esse curso cobrirá em mais detalhes.

Escalonamento Baseado em Prioridades Variáveis

Exemplos clássicos de políticas que variam a prioridade dinamicamente e que são usadas em sistemas de tempo-real são *Least-Slack-Time-First* (LST) [65, §4] e *Earlier-Deadline-First* (EDF)[64]. Algoritmos de escalonamento baseados nessas políticas são conhecidamente *ótimos* se as tarefas são escalonadas em apenas um processador, não compartilham recursos e podem sofrer preempção. Ser ótimo aqui significa que se existe algum escalonamento do conjunto de tarefas considerado, tal que nenhuma tarefa viola seu *deadline*, então os escalonamentos produzidos tanto por LST quanto por EDF também cumprirão todos os *deadlines*.

De acordo com LST, a tarefa com maior prioridade num instante t é aquela que

possui a menor folga de tempo para ser executada de forma que seu *deadline* não seja violado. Ordena-se assim as prioridades das tarefas com base nos valores das respectivas folgas. Essa folga de tempo (*slack*) é definida como $d_i - t - c_i(t)$, onde d_i é o *deadline* absoluto de τ_i, ou seja o tempo de término máximo para a tarefa ser executada com sucesso; e $c_i(t)$ é o que resta para ser executado da tarefa τ_i a partir do instante t. Tal folga de tempo é monitorada em tempo de execução para que as prioridades relativas entre as tarefas em execução sejam ajustadas.

Para seguir a política LST estritamente, deve-se ter o conhecimento sobre o tempo real de execução das tarefas (a fim de derivar $c_i(t)$). Dificilmente tem-se esse conhecimento. Geralmente, é o tempo de execução máximo C_i que é conhecido e, portanto, usado. Assim, na prática, aproxima-se a política LST para lidar com tal atributo. Outra limitação dessa política está relacionada com o instante do monitoramento da folga de execução das tarefas em tempo de execução. Tal parâmetro deve ser atualizado constantemente a fim de ajustar as prioridades relativas das tarefas. Devido ao alto custo computacional que tal abordagem iria impor, implementações de LST monitoram o tempo de folga apenas quando tarefas são ativadas ou têm suas execuções completadas.

O escalonador EDF, por sua vez, atribui, em tempo de execução, prioridade máxima à tarefa τ_i que possui menor *deadline* d_i. Assim, não há a necessidade de monitoramento dos tempos de execução nem das folgas de tempo das tarefas. Isso torna, do ponto de vista de implementação, o EDF mais atrativo.

O exemplo da figura 3.17, com duas tarefas periódicas, ilustra a diferença básica entre ambas as abordagens (considerando a versão aproximada do LST). Os atributos de τ_1 e τ_2 são: $T_1 = 30$ms, $C_1 = 10$ms e $D_1 = 20$ms; $T_2 = 80$ms, $C_2 = 45$ms e $D_2 = 79$ms. Ambas as tarefas são ativadas no instante 0. Nesse momento a prioridade τ_1 é maior que a de τ_2 para ambos os escalonadores, pois $d_1 = 30$ms é menor que $d_2 = 79$ms (critério do EDF) e $20 - 0 - 10 = 10$ms é menor que $79 - 0 - 45 = 34$ms (critério do LST). Quando τ_1 termina sua execução, τ_2 inicia a sua até sofrer preempção, aos 30ms, pois, na sua segunda ativação, τ_1, mais uma vez, tem maior prioridade para ambos os escalonadores. Apenas na sua terceira ativação, aos 60ms, é que as decisões de escalonamento do EDF e do LST diferem. O EDF atribui maior prioridade a τ_2, pois nesse instante $d_2 = 79$ms é menor que $d_1 = 80$ms. Para o LST, no entanto, τ_1 é mais prioritária, pois seu tempo de folga ($80 - 60 - 10 = 10$ms) é menor que o de τ_2 ($79 - 60 - 5 = 14$ms).

Escalonamento Baseado em Prioridades Fixas

A teoria de escalonamento usando prioridades fixas iniciou-se com a publicação do algoritmo *Rate-Monotonic* (RM) [64], uma simples heurística de atribuição de prioridades, que se dá em ordem inversa do período das tarefas. Quanto maior o período da tarefa, menor é sua prioridade. Atribuir prioridades fixas às tarefas tomando como parâmetro seus *deadlines* é uma outra política clássica de escalonamento, chamada de *Deadline-Monotonic* (DM) [4]. De acordo com DM, prioridades são atribuídas em ordem inversa de *deadlines*.

Como ilustração, considere o seguinte exemplo composto por duas tarefas peri-

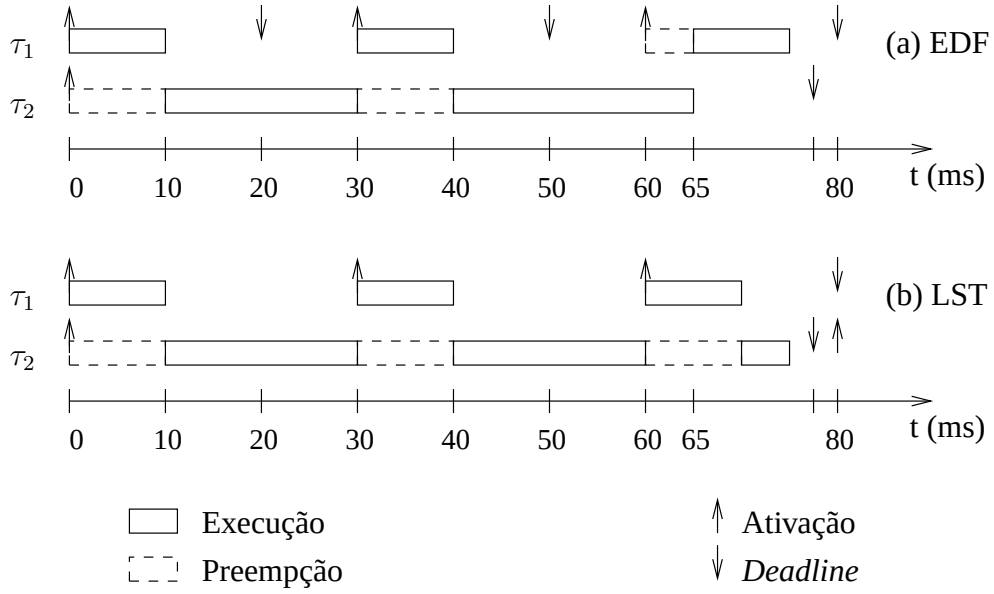

Figura 3.17: Ilustração de escalonamentos: (a) EDF e (b) LST.

ódicas, τ_1 e τ_2. O tempo máximo de execução de cada uma dessas tarefas é 30ms. Tarefa τ_1 tem período 80ms e *deadline* relativo 70ms, enquanto que esses atributos da tarefa τ_2 são 120ms e 50ms, respectivamente. Pelo critério RM, a tarefa τ_1 é a mais prioritária, enquanto que o escalonador DM atribui maior prioridade a τ_2. Como pode ser notado, esse conjunto de tarefas é escalonável pelo DM, mas não pelo RM. De fato, a tarefa τ_2 viola seu *deadline* na primeira e terceira ativações, momento em que τ_1 é mais prioritária. Esse exemplo ilustra o fato de que escalonadores DM possuem melhor desempenho que o RM [65, §6]. Note, contudo, que se as relações de monotonicidade entre períodos e *deadlines* relativos das tarefas são as mesmas, RM e DM se tornam equivalentes.

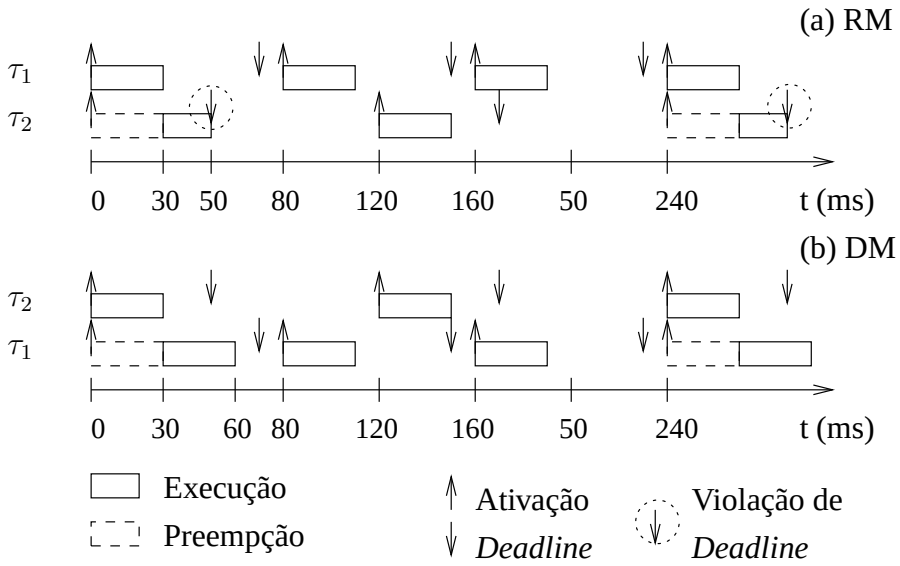

Figura 3.18: Ilustração de escalonamentos: (a) RM e (b) DM.

Como dito anteriormente, escalonadores baseados em prioridades fixas geralmente apresentam vantagens quando comparados com aqueles baseados em prioridades variáveis com relação a previsibilidade [65]. A intuição por atrás dessa afirmativa é a seguinte. Como as prioridades são fixas, uma tarefa sofre interferência apenas daquelas tarefas que possuem prioridades mais altas. Assim, pode-se, por exemplo, determinar, em situações de sobrecarga, quais as tarefas que não cumprirão seus *deadlines* analisando o nível de interferência que elas podem sofrer. Com prioridades variáveis, isso nem sempre é verdade, pois a atribuição de prioridades é também função do estado do sistema. Ou seja, é difícil prever quais tarefas podem violar seus *deadlines*, visto que suas prioridades variam em função do tempo. Por essa razão, algoritmos e análises de escalonabilidade baseados em prioridades fixas são preferíveis para sistemas de tempo-real críticos e, portanto, serão o foco de atenção das seções seguintes.

3.6.2. Análise de Escalonamento

Analisar (ou testar) a escalonabilidade de um sistema é verificar se todas as suas tarefas irão cumprir seus *deadlines*. Nem sempre é possível usar simulação como ferramenta para tal verificação devido ao alto número de possibilidades de escalonamento. Geralmente, usa-se funções matemáticas que exprimem o comportamento temporal do sistema no pior caso.

A análise de escalonamento é dita *suficiente* caso forneça um limite superior de escalonabilidade do sistema. Após esse limite nada pode ser dito com relação a escalonabilidade do conjunto de tarefas analisado. Assim, nem todos os conjuntos de tarefas escalonáveis são considerados como tal por análises apenas suficientes. Se a análise de escalonamento é apenas *necessária*, ela fornece limites de não-escalonabilidade do sistema. Em outras palavras, se o conjunto de tarefas não é escalonável, a análise assim o acusa. Caso contrário, nada pode ser afirmado. Obviamente, análises apenas necessárias são de pouca utilidade para o projetista do sistema. O ideal é usar uma análise *exata*, que é ao mesmo tempo necessária e suficiente. Desta forma, tanto os limites de escalonabilidade quanto os de não-escalonabilidade podem ser conhecidos.

Duas abordagens são geralmente usadas para se analisar a escalonabilidade de sistemas de tempo-real. Elas podem estar baseadas no cálculo da utilização máxima do processador ou podem verificar se os tempos de resposta das tarefas no pior caso ultrapassam seus *deadlines*.

Análise Baseada em Utilização do Processador

Suponha um conjunto de n tarefas periódicas e preemptíveis $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$, que não compartilham recursos e são escalonadas num sistema com único processador. Cada tarefa τ_i é ativada no sistema a uma taxa de $1/T_i$ e executa no máximo C_i. Em outras palavras, cada tarefa *utiliza* no máximo $u_i = C_i/T_i$ do potencial do processador. Assim, o fator de utilização máximo desse conjunto de tarefas é dado por

$$U = \sum_{\forall \tau_i \in \Gamma} \frac{C_i}{T_i} \tag{3.1}$$

A equação (3.1) é a forma geral da análise de escalonamento baseada na utilização do processador. Como pode ser notado, tal função matemática é simples, podendo inclusive ser calculada em tempo de execução. Por essa razão, muitas vezes, esse tipo de teste de escalonabilidade é usado na abordagem *on-line* (célula (4) da tabela 3.3) [65, 23].

O uso da equação (3.1) é bastante comum tanto para sistemas que usam escalonadores baseados em prioridades variáveis quanto para aqueles que usam prioridades fixas. Por exemplo, para o EDF, quando suas condições de otimalidade são verificadas e quando os *deadlines* das tarefas são iguais a seus respectivos períodos, basta verificar se $U \leq 1$ para determinar se o sistema é escalonável [64]. Assumindo o mesmo modelo de tarefas e o escalonador RM, usa-se o teste $U \leq n(2^{1/n} - 1)$. No primeiro caso o teste é exato enquanto que no segundo ele é apenas suficiente. Por exemplo, suponha duas tarefas periódicas que não compartilham recursos e cujos períodos são iguais aos seus respectivos *deadlines*. Seus atributos são: $T_1 = 10$, $C_1 = 5$, $T_2 = 20$, $C_2 = 10$. Usando equação (3.1), $U = 1$. Isso significa que o conjunto de tarefas é escalonável pelo escalonador EDF. Contudo, como $2(2^{1/2} - 1) < 1$, a análise falha em verificar a escalonabilidade desse conjunto de tarefas para o RM (lembrando que nesse caso a análise é apenas suficiente). É interessante notar que ambas as tarefas são de fato escalonáveis segundo RM, como ilustra a figura 3.19.

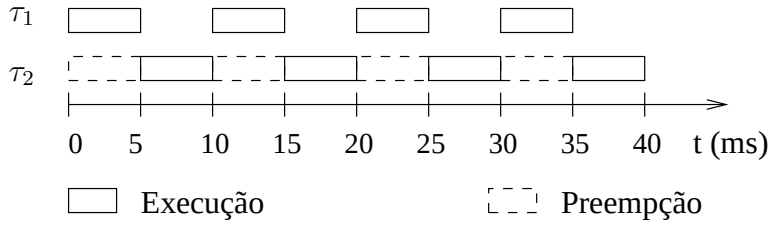

Figura 3.19: Escalonamento possível para o RM quando $U > n(2^{1/n} - 1)$.

A principal desvantagem dos testes baseados na equação 3.1 é que, geralmente, eles impõem restrições ao modelo de tarefas do sistema, o que, muitas vezes, inviabiliza seu uso na prática. Por exemplo, impor que os *deadlines* das tarefas sejam iguais aos seus respectivos períodos pode ser muito restritivo para sistemas de tempo-real em geral. Além disso, no caso particular do RM, o teste oferece um baixo nível de utilização de processador. De fato, quando o número n de tarefas cresce, a expressão $n(2^{1/n} - 1)$ tende a 0.69 [64]. Isso significa que um grande número de conjunto de tarefas escalonáveis (tal como o ilustrado na figura 3.19), mas que possuem taxa de uso de processador mais alta, é descartado pela análise. A abordagem descrita a seguir, usada para escalonadores baseados em prioridades fixas, possui a vantagem de modelar conjuntos de tarefas mais complexos mais facilmente e não traz consigo as desvantagens descritas acima.

Análise Baseada em Tempo Máximo de Resposta

Ao invés de medir a razão de utilização do processador, a abordagem descrita nessa seção calcula, para cada tarefa $\tau_i \in \Gamma$, seu pior tempo de resposta, R_i. Se $R_i \leq D_i$, a tarefa é escalonável. A implementação do procedimento para o cálculo de R_i, como poderá ser

observado, é bastante simples. É um procedimento iterativo, que converge para um valor finito caso $U \leq 1$ [22, §13]. A abordagem descrita aqui localiza-se na célula (2) da tabela 3.3 visto que a análise de escalonabilidade é realizada em tempo de projeto e as decisões de escalonamento são tomadas em tempo de execução.

Assume-se aqui um conjunto Γ de tarefas que são escalonadas por algum critério baseado em prioridades fixas (*e.g.*, RM ou DM). Em outras palavras, o critério segundo o qual as prioridades são atribuídas não é relevante para a validade da análise (apesar de ser para a escalonabilidade do sistema). Além disso, o modelo de tarefas inicialmente assumido nessa seção contempla apenas tarefas periódicas, que não compartilham recursos, que podem sofrer preempção e que possuem *deadlines* menores ou iguais aos seus respectivos períodos. Essas hipóteses simplificadoras do modelo serão removidas ao longo da descrição. Note, contudo, que esse modelo já é menos restritivo que o assumido na seção anterior, pois *deadlines* podem ser diferentes dos períodos.

Intuitivamente, pode-se verificar que o pior tempo de resposta de cada tarefa $\tau_i \in \Gamma$ ocorre quando é necessário C_i unidades de tempo para executar τ_i e que τ_i sofre interferência máxima (através de preempção) das tarefas que possuem priodidades mais elevadas que τ_i. Seja A_i essa interferência. Então, o tempo máximo de resposta de τ_i é dado por

$$R_i = C_i + A_i, \tag{3.2}$$

A interferência A_i ocorre quando todas as tarefas $\tau_j \in \Gamma$, que têm prioridades maiores que τ_i, são ativadas no sistema no mesmo instante que τ_i. Nesse cenário, note que cada τ_j pode ocorrer $\left\lceil \frac{R_i}{T_j} \right\rceil$ vezes durante o tempo de resposta de τ_i.[2] No pior caso, para cada uma dessas vezes, a execução de τ_j interfere C_j na execução de τ_i. Assim, o termo A_i é dado por:

$$A_i = \sum_{j \in \mathsf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j, \tag{3.3}$$

onde $\mathsf{hp}(i)$ é o conjunto de tarefas mais prioritárias que τ_i. Pelas equações (3.2) e (3.3), têm-se que [4]:

$$R_i = C_i + \sum_{j \in \mathsf{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j, \tag{3.4}$$

Note que o termo R_i aparece em ambos os lados da equação (3.4). Para resolvê-la aplica-se a relação de recorrência dada pela equação (3.5) [4] num procedimento iterativo. A iteração pode se iniciar com $r_i^0 = C_i$, onde r_i^k é a k-ésima aproximação de R_i. O procedimento se encerra quando $r_i^{k+1} > D_i$ ou se $r_i^{k+1} = r_i^k$ para algum k. No primeiro caso, τ_i não é escalonável, enquanto que o segundo significa que $R_i = r_i^k$.

$$r_i^{n+1} = C_i + \sum_{j \in \mathsf{hp}(i)} \left\lceil \frac{r_i^n}{T_j} \right\rceil C_j \tag{3.5}$$

[2] A operação $\lceil x \rceil$ retorna o menor inteiro que é maior ou igual a x.

Para exemplificar o uso da análise, considere o conjunto das tarefas dado pela aplicação UAV, descrita na seção 3.2.5. Assuma que as três tarefas do sistema de controle de navegação, τ_{gps}, τ_{vrf} e τ_{ctl} estejam alocadas num processador. Assuma também que não haja compartilhamento de recursos entre essas tarefas tal que uma possa causar o bloqueio da outra. Essas hipóteses simplificadoras serão removidas em breve.

Aplicando a análise de escalonamento descrita pela equação (3.4), encontra-se os valores 20, 60 e 140 para os tempos de respostas das tarefas τ_{gps}, τ_{vrf} e τ_{ctl}, respectivamente. O processo iterativo para tarefa τ_{ctl} é ilustrado a seguir.

$$
\begin{aligned}
r^0_{\text{ctl}} &= 60 \\
r^1_{\text{ctl}} &= 60 + \left\lceil \frac{60}{100} \right\rceil 20 + \left\lceil \frac{60}{150} \right\rceil 40 = 120 \\
r^2_{\text{ctl}} &= 60 + \left\lceil \frac{120}{100} \right\rceil 20 + \left\lceil \frac{120}{150} \right\rceil 40 = 140 \\
r^3_{\text{ctl}} &= 60 + \left\lceil \frac{140}{100} \right\rceil 20 + \left\lceil \frac{140}{150} \right\rceil 40 = 140 \\
R_{\text{ctl}} &= 140
\end{aligned}
$$

É interessante observar que esse teste de escalonabilidade é exato para o modelo de tarefas aqui assumido. Outra observação que deve ser mencionada é com relação à flexibilidade da análise. Por exemplo, a equação (3.4) também pode ser aplicada quando existem tarefas esporádicas, visto que essas, no pior caso, comportam-se como tarefas periódicas, como é o caso do exemplo acima, pois τ_{ctl} é esporádica.

A equação (3.4) pode ser modificada para lidar com modelos de tarefas que contemplam *jitter*, *deadlines* arbitrários, compartilhamento de recursos [89, 3, 22] ou incorporam a possibilidade da ocorrência de erros durante a execução das tarefas [21, 20, 61, 60, 62]. Na próxima seção algumas dessas adaptações serão apresentadas.

3.6.3. Lidando com *Jitter*

Tomando o exemplo da seção 3.2.5, note que a ativação de τ_{ctl}, tarefa esporádica, depende da execução de τ_{vrf}. Por exemplo, a depender do resultado do processamento de τ_{vrf}, τ_{ctl} pode ser acionada para realizar algum ajuste da meta. Assuma que τ_{vrf} ativa τ_{ctl} ao final do seu processamento. Assim, o período de ativação de τ_{ctl} é igual ao período de τ_{vrf}, caso τ_{ctl} seja ativada a cada instância de τ_{vrf}. A figura 3.20 ilustra esse cenário. Porém, observe que quando τ_{vrf} termina mais cedo (terceira ativação na figura), ocorrerá uma diminuição do intervalo de tempo de duas ativações consecutivas de τ_{ctl}. Essa diferença é o *jitter* associado à ativação de τ_{ctl}. Nesse exemplo, considerando que τ_{vrf} pode terminar entre 0ms (melhor caso) R_{vrf} (pior caso), esse *jitter* valerá 60ms.

Uma conseqüência direta do *jitter* é que poderá haver um aumento da interferência que a execução de τ_{ctl} provoca em tarefas menos prioritárias. Por exemplo, seja τ_i uma tarefa menos proritária. Do ponto de vista de τ_i, poderá haver ativações extras de τ_{ctl} (*e.g.*, entre a segunda e terceira ativações na figura 3.20). Assim, o número máximo de

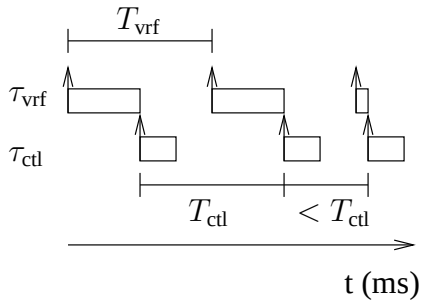

Figura 3.20: Ilustração de possível *Jitter* da tarefa τ_{ctl}.

ocorrências de τ_{ctl} durante o tempo de resposta de τ_i é dado por $\left\lceil \frac{R_i + J_{\text{ctl}}}{T_{\text{ctl}}} \right\rceil$ ao invés de $\left\lceil \frac{R_i}{T_{\text{ctl}}} \right\rceil$. Generalizando, qualquer tarefa que experimenta *jitter*, pode aumentar a interferência na execução de tarefas menos prioritárias. Modificando a equação (3.4) para considerar esse efeito, obtém-se a equação (3.6) [4].

$$W_i = C_i + \sum_{j \in \mathsf{hp}(i)} \left\lceil \frac{W_i + J_j}{T_j} \right\rceil C_j, \tag{3.6}$$

onde W_i é o tempo de resposta máximo de τ_i sem considerar J_i. Se τ_i também sofre *jitter*, deve-se somá-lo ao seu tempo de resposta. Assim, o valor final de R_i é dado por:

$$R_i = W_i + J_i \tag{3.7}$$

É interessante observar que o *jitter* pode ser usado para modelar uma das possibilidades de comunicação entre tarefas, como a figura 3.20 ilustra. Portanto, com a equação (3.6), o modelo de tarefas agora pode contemplar, além de tarefas esporádicas, dependência entre tarefas. Com essa alteração, porém, a análise passa a ser apenas suficiente. A seguir, um outro tipo de dependência entre tarefas será modelado.

3.6.4. Comunicação entre Tarefas: Compartilhamento de Recursos

Até então foi suposto que as tarefas não compartilham recursos. Tal modelo de tarefas restringe bastante o modo de se construir um sistema concorrente. O principal problema que surge quando há compartilhamento de recursos num sistema baseado em prioridades é a possibilidade de tarefas mais prioritárias ficarem esperando (bloqueadas) por recursos que estão sendo usados por tarefas menos prioritárias. A descrição desse problema, sua solução e suas conseqüências para a análise da escalonabilidade do sistema são discutidas nesta seção.

Inversão de Prioridades

Seja $\Gamma = \{\tau_1, \tau_2, \tau_3\}$ um conjunto de tarefas e suponha que suas prioridades fixas são atribuídas tal que τ_1 tem a mais alta prioridade enquanto que τ_3 possui a prioridade mais baixa. Suponha um recurso compartilhado por τ_1 e τ_3 e considere o seguinte cenário (figura 3.21). A tarefa τ_3 começa executar e aloca o recurso. Em seguida, τ_1 inicia sua

execução no instante a, ficando bloqueada a partir do momento em que requisita o recurso (instante b). No instante c, τ_2 inicia sua execução, causando a preempção de τ_3. A partir desse instante até o tempo d a execução de τ_1 sofre interferência não apenas da execução de τ_3, mas também da tarefa τ_2, com a qual τ_1 não compartilha recurso algum. Esse cenário caracteriza o problema de inversão de prioridades. Note que quaisquer tarefas que tenham prioridades intermediárias (entre τ_1 e τ_3) poderão interferir na execução da tarefa mais prioritária. Conseqüentemente, o comportamento temporal de τ_1 pode se tornar imprevisível.

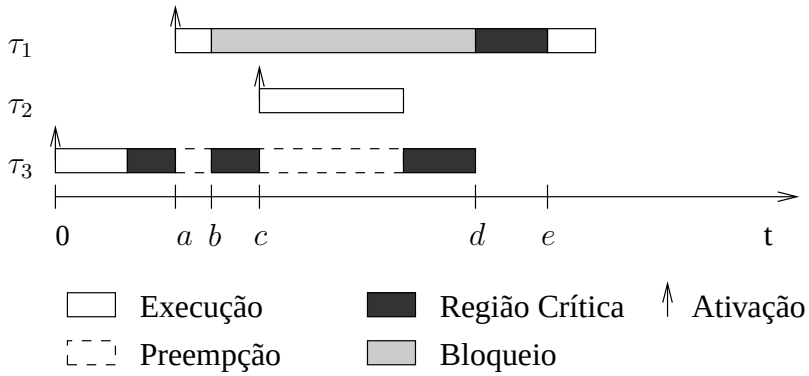

Figura 3.21: Inversão de prioridades.

Obviamente, a inversão de prioridades é um fenômeno que não pode ser completamente evitado quando há recursos compartilhados. Pode-se, contudo, determinar seus efeitos e minimizá-los. Uma simples solução para o problema é proibir a preempção de tarefas, quando essas estejam executando numa região crítica. Dessa forma, a execução de τ_3 não seria interrompida pela chegada de τ_2 e τ_1 esperaria um tempo máximo conhecido (determinado pelo tempo que τ_3 usa o recurso). Esse protocolo, apesar de resolver o problema, não é recomendado. De fato, nesse caso a execução de τ_3 (menos prioritária) causaria interferências em todas as tarefas do sistema, independentemente dessas estarem ou não compartilhando recursos. Existem várias possíveis soluções para esse problema. As seções abaixo discutem uma das mais usadas e eficazes, conhecida como protocolo de prioridade teto (*priority ceiling protocol*) [84].

Protocolo de Prioridade Teto

Esse protocolo assume que tarefas são escalonadas baseadas em prioridades fixas e que os recursos por elas requisitados são conhecidos. A cada recurso compartilhado é atribuída uma *prioridade teto*. Essa prioridade é igual a prioridade da tarefa mais prioritária que aloca tal recurso.

O princípio de funcionamento do protocolo é bastante simples. Para entendê-lo reconsidere o exemplo da figura 3.21. Quando τ_3 está executando na sua região crítica, sua prioridade é aumendada para a prioridade teto (mesma prioridade de τ_1). A tarefa τ_3 terá sua prioridade restaurada tão logo acabe de usar o recurso. A partir desse momento, τ_1 volta a ser mais prioritária e, assim, será escolhida para execução, causando a preempção

de τ_2. Esse cenário é ilustrado na figura 3.22. Na verdade, o protocolo aqui descrito é uma versão melhorada do protocolo de prioridade teto original [22, §13], pois as tarefas têm suas prioridades elevadas para a prioridade teto logo no início da região crítica.

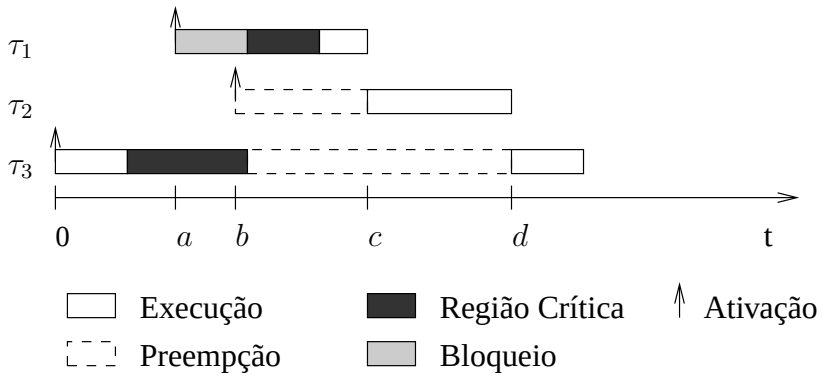

Figura 3.22: Inversão de prioridades.

Note que apesar de τ_3 ainda poder causar o aumento do tempo de resposta de τ_1 (o que é inevitável devido ao compartilhamento de recursos), as seguintes propriedades fazem o protocolo descrito muito interessante [22, 65]:

- o tempo que tarefas menos prioritárias bloqueiam tarefas mais prioritárias é minimizado;
- cada tarefa sofre no máximo um bloqueio por ativação causado por tarefas menos prioritárias;
- o tempo máximo de bloqueio é fixo e pode ser calculado;
- cenários de *deadlock* são evitados;
- conflitos de acesso a recursos compartilhados são resolvidos sem a necessidade do uso de trancas (*locks*) explícitas. A simples manipulação de prioridades funciona como trancas implícitas.

Tempo de Resposta das Tarefas

Seja K o conjunto de recursos compartilhados. Uma tarefa que usa um recurso compartilhado $k \in K$ somente poderá ser bloqueada por uma tarefa menos prioritária τ_j se [84]: (a) esta também usa o recurso k; e (b) a prioridade teto de k é maior ou igual à prioridade de τ_i. Defina então a função uso(k, i) que retorna 1, caso as condições (a) e (b) sejam satisfeitas, e que retorne 0 caso contrário. Lembrando que uma das características do protocolo de prioridade teto é que τ_i fica bloqueada por tarefas menos prioritárias apenas uma vez por ativação, para calcular o tempo máximo de bloqueio basta calcular o tempo máximo que o recurso pode estar sendo usado por tarefas menos prioritárias. Seja B_i esse tempo e assuma que RC(k) representa o tempo máximo de uso do recurso k. Então

$$B_i = \max_{k=1}^{K} \text{uso}(k, i)\text{RC}(k), \tag{3.8}$$

Incorporar o tempo B_i ao cálculo do tempo máximo de resposta de τ_i é bastante simples. Basta acrescentar o termo B_i na equação (3.7). Assim,

$$W_i = C_i + B_i + \sum_{j \in \mathsf{hp}(i)} \left\lceil \frac{W_i + J_j}{T_j} \right\rceil C_j, \tag{3.9}$$

A determinação de R_i é então dada pela equação (3.7).

3.6.5. Comunicação entre Tarefas: Ambiente Distribuído

Na seção anterior foi visto que a comunicação entre tarefas através do compartilhamento de recursos em memória comum pode afetar a previsibilidade do sistema, visto a possibilidade de occorrência do problema de inversão de prioridades. Num sistema distribuído, onde não há compartilhamento de memória, tarefas também podem interferir no comportamento temporal umas das outras. De fato, a rede de comunicação pode ser vista como um recurso compartilhado, diferindo, entretanto, em vários aspectos de ambientes uniprocessados. Por exemplo, tarefas podem tentar enviar mensagens através da rede de comunicação simultaneamente, exigindo que o subsistema de comunicação possua um protocolo elaborado para o controle de acesso ao meio. Além disso, preempção, usada como mecanismo de controle em sistemas centralizados, não está disponível no que se refere a transmissão de mensagens numa rede de comunicação.

Existe uma vasta gama de tecnologias de redes de comunicação que são usadas em sistemas de tempo-real, tanto no que se refere a redes locais (LAN) quanto a redes de longa distância (WAN), tornando impossível resumi-las aqui. Tomando apenas as LAN, por exemplo, a tecnologia TDMA (*Time Division Multiple Access*), usada pelo protocolo TTP (*Time-Triggered Protocol*) [50] estabelece intervalos de tempos de transmissão para cada nodo da rede. Outros tipos são [87, 65] *Token-Ring*, *Token-Bus* ou FDDI. Essas possuem um protocolo de acesso ao meio mais elaborado de forma que seja possível *escalonar* a transmissão de mensagens na rede, garantindo a previsibilidade da comunicação. Dado essa característica, pode-se derivar a análise do sistema como um todo considerando tanto o comportamento das tarefas quanto o das operações de transmissão e recebimento de mensagens [65, 91]. Devido a sua abrangência, a análise de escalonabilidade do sub-sistema de comunicação não será abordada aqui.

É desejável que protocolos de redes usados em sistemas de tempo-real possam prover o conceito de prioridades de mensagens. Desta forma, o sistema pode fornecer suporte ao escalonamento de transmissão de mensagens baseado em prioridades, de modo similar aos mecanismos usados no escalonamento de tarefas. A implementação desse conceito na maioria das redes, contudo, é restritiva. Por exemplo, o protocolo *Token-Ring* possui apenas três bits para representar prioridades. Nas redes FDDI existem dois tipos de mensagens, as síncronas e as assíncronas. Mensagens síncronas são periódicas, possuem *deadlines* e são transmitidas preferencialmente em relação às assíncronas, que seguem a política melhor-esforço.

Do ponto de vista do uso de prioridades de mensagens, a melhor representante talvez seja a rede CAN (*Controller Area Network*) [43], uma rede tipo barramento desenvolvida pela BOSCH originalmente para sistemas automotivos. Por essa estar estritamente baseada no conceito de prioridades de mensagens é interessante salientar alguns dos seus aspectos de funcionamento.

As mensagens CAN são unicamente identificadas por um campo de 11 bits (suas prioridades). Há portanto um total de 2048 diferentes níveis de prioridades disponíveis. Mensagens são transmitidas no barramento, um bit de cada vez, iniciando pela parte mais significativa do identificador. O bit pode ser dominante (0) ou recessivo (1). Se dois bits são transmitidos simultaneamente por dois diferentes nodos, o bit resultante é o dominante. Desta forma, a rede de comunicação se comporta como uma porta lógica 'E' e as mensagens mais prioritárias são aquelas que possuem menor identificador. Os nodos transmissores entram em sincronia para iniciar uma transmissão. Enquanto uma mensagem está sendo transmitida, tanto o nodo transmissor quanto os nodos receptores monitoram o barramento. Assim, se o bit transmitido é recessivo e um bit dominante é monitorado, o nodo transmissor suspende a transmissão e inicia o recebimento da mensagem. Quando o nodo transmissor pára de transmitir uma mensagem, sua mensagem é automaticamente escalonada para retransmissão.

Uma das grandes vantagens de CAN sobre outras redes de comunicação usadas para sistemas de tempo-real críticos é que ela se comporta fielmente ao modelo de sistema baseado em eventos. Mensagens associadas aos eventos cujo tratamento é mais prioritário terão maior prioridade e serão escalonadas antes daquelas menos prioritárias. Além disso, como o controle de acesso ao barramento é baseado nas prioridades, a análise de escalonamento para tarefas de prioridades fixas pode ser facilmente adaptada para lidar com transmissão de mensagens [90]. Sua principal desvantagem está relacionada ao baixo desempenho. De fato, a taxa de transmissão máxima em redes CAN é 1 Mbps.

3.7. Técnicas de Tolerância a Falhas para Sistemas Críticos

Os aspectos de previsibilidade necessários ao bom funcionamento dos sistemas de tempo-real críticos, especialmente os distribuídos, dependem do grau de confiança que podemos atribuir aos seus componentes, pois o defeito de qualquer desses componentes - caso não seja devidamente tratado - pode levar a uma catástrofe ou grandes prejuízos. Um sistema em que se pode confiar, ou *dependable*, pode ser qualificado de acordo com vários atributos, como *reliability*, *availability*, *maintainability*, *safety* e *security*, que tratam de serviço continuado, disponibilidade do serviço, tempo de reparo, confiabilidade em modos de falhas catastróficos e segurança (confidencialidade, autenticação etc.), respectivamente [56]. Uma vez que estamos interessados em sistemas críticos que não devam sofrer interrupções em seus serviços, então *reliability* ou confiabilidade é o atributo que nos interessará mais de perto. Em termos mais precisos, dado que determinado sistema (ou componente) está operacional no tempo t_0, uma medida de *reliability* define a probabilidade desse mesmo sistema estar operacional no tempo t, $t > t_0$. Para tal cálculo usa-se, por exemplo, uma razão de falhas constante λ definida em função de uma unidade de tempo (*e.g.*, λ = X falhas a cada hora). Assumindo que a função que determina o tempo de vida do sistema (ou componente) é exponencial, então a *reliability* R num dado tempo t é definido por $R(t) = e^{-(t-t_0)}$. Uma medida comumente usada para quantificar essa confiabilidade é o tempo médio para falhas ou MTTF (*Mean-Time-To-Failure*), definido como $1/\lambda$ [92, 44, 52].

Prevenção e tolerância a falhas (*fault-avoidance* e *fault-tolerance*) são os mecanismos fundamentais para atingirmos Confiança no Funcionamento (*dependability*), em

especial *reliability*. Na seção 3.3 discutimos como a especificação e verificação formal podem ser usadas para implementar prevenção de falhas. Entretanto, por mais exaustiva que seja a verificação e testes dos programas de tempo-real, estes não poderão excluir a possibilidade de eventuais falhas (por exemplo, causadas por problemas ambientais ou envelhecimento dos circuitos). Por conseguinte, mecanismos de tolerância a falhas são imprescindíveis, especialmente nos sistemas críticos onde interrupções do serviços podem levar à catástrofes. Nesta seção, iremos explorar os meios de se projetar tais sistemas de modo que estes possam funcionar corretamente mesmo na ocorrência de falhas.

Inicialmente serão apresentados os principais conceitos relacionados à tolerância a falhas, nos domínios de valor e temporal. Em seguida, serão introduzidas as técnicas fundamentais de tolerância a falhas, dando especial ênfase à técnica de replicação ativa usualmente empregadas em sistemas críticos. Por fim, o exemplo do UAV (*Unmanned Air Vehicles*) será visitado, indicando como utilizar as técnicas descritas para aumentar a confiabilidade de tais aeronaves não tripuladas.

3.7.1. O Modelo do Sistema

O modelo de um sistema constitui-se de uma representação abstrata do ambiente computacional e de comunicação onde os serviços projetados serão, em última instância, postos em funcionamento. Num sistema distribuído, a visão mais simplificada do modelo corresponde a um conjunto de processos que se comunicam apenas por troca de mensagens, através de canais de comunicação. Os processos em geral representam unidades de escalonamento (tarefas) e os canais de comunicação representam as redes de comunicação e protocolos correlatos. A especificação do modelo deve ser detalhada à medida que novos aspectos do mundo real precisarem ser representados para a construção de soluções e serviços. Via de regra, os sistemas distribuídos de tempo-real são caracterizados por modelos de sistemas onde há limites temporais finitos e conhecidos para a computação de ações internas nos componentes e para o tempo de transferência de mensagens entre os mesmos. Esse conhecimento prévio é necessário para que os recursos do sistema e escalonamento de tarefas possam ser realizados com vistas a atender os limites de tempo requisitados pela aplicação. Esses modelos são denominados de síncronos. Nos sistemas de tempo-real não-críticos, entretanto, admitem-se modelos onde tais limites de tempo possam se dar apenas em momentos de estabilidade, os chamados modelos parcialmente síncronos [28, 30, 33]. Nesta seção nos concentraremos nos aspectos de confiabilidade relacionados aos sistemas de tempo-real críticos, portanto modelados por sistemas síncronos. Como visto na introdução (seção 3.1), nenhum sistema está isento de ocorrências de falhas (de software ou hardware). Assim sendo, os tempos conhecidos no modelo síncrono não se verificarão caso ocorram defeitos no sistema. Consequentemente, é necessario que mecanismos de tolerância a falhas sejam introduzidos para reduzir a probabilidade de a ocorrência de falhas levar à interrupção do serviço.

3.7.2. O Modelo de Falhas

Um passo importante para se projetar um sistema tolerante a falhas é procurar descrever quais tipos de falhas (ou defeitos) podem acometer os componentes ou canais de comunicação. Uma vez definido o modelo de falhas e dotado o sistema de mecanismos

adequados, podemos afirmar que o sistema passa a ser tolerante a falhas dentro do modelo previsto (i.e., tipos, freqüência e quantidades máximas especificadas). É importante notar aqui que os mecanismos de tolerância a falhas introduzidos devem ser projetados de modo a não causar violações nos requisitos temporais (*deadlines*) definidos para o serviço. Falhas podem ter diversas origens, que vão desde causas físicas, como rompimento de uma trilha em circuito eletrônico ou ruídos na rede elétrica, até falhas de origem humana, como erro de projeto ou mesmo erro causado pelo mau uso dos sistemas (intencional ou não). Independentemente da origem da falha, num sistema de tempo-real distribuído, ela pode ser classificada de acordo com o comportamento do componente após a falha. Ou seja, como se comporta o componente quando apresenta defeito? Esse comportamento (e correspondente classificação do tipo de defeito/falha) pode variar de acordo com o nível de informação disponível após a falha. No extremo mais simples ou mais benigno, encontra-se o tipo de falha chamado de *fail-stop* [81] onde o componente falho pára de gerar qualquer resultado externo e, além disso, essa falha é garantidamente percebida (ou detectada) pelos componentes corretos do sistema. Observe que esse tipo de falha não afeta a computação de valores - a falha é apenas sentida na dimensão temporal. No outro extremo encontra-se a falha generalizada ou bizantina, onde pode ocorrer de tudo: valores gerados de forma errada, requisitos temporais (*deadlines*) podem ser violados e, pior ainda, componentes distintos podem perceber a falha do mesmo componente de forma inconsistente (*e.g.*, um componente recebe um valor errado, um outro componente, um valor errado diferente do primeiro, e um terceiro componente recebe o valor correto, mas fora do limite temporal especificado). Outros dois modelos intermediários frequentemente considerados nos sistemas síncronos são falhas de omissão, onde o componente deixa de produzir alguma saída esperada, e falhas temporais, onde saídas são produzidas, mas fora do limite de tempo especificado.

As hipóteses dos tipos de falhas que podem acometer o sistema, definem o que chamamos de *Modelo de Falhas*. A especificação de tal modelo é fundamental para que os mecanismos de tolerância à falhas possam ser testados e validados. Um cuidado que se deve ter é que o mapeamento feito pelas hipóteses de falhas deve ser o mais preciso possível de modo a permitir a implementação de mecanismos de tolerância para falhas mais frequentes. Idealmente, as falhas não mapeadas pelas hipóteses não devem ocorrer na prática. Entretanto, caso ocorram, essas devem ser eventos raros. Nesse caso, para garantir a confiabilidade do sistema, estratégias alternativas, que levem o sistema a um estado seguro na presença de falhas não previstas nas hipóteses, devem ser implementadas. Para isto, é preciso que sejam desenvolvidos algoritmos que detectem a violação das hipóteses e que ativem as estratégias alternativas. Sistemas com essas catacteristicas são chamados de *fail-safe*, pois assumem um estado seguro quando não mais podem continuar operando (*e.g.*, num sistema de sinalização de linhas de trens, um estado seguro seria parar todos os trens, evitando, portanto, possíveis colisões devido a defeitos no sistema de sinalização).

Quanto mais maligna a falha, mais díficil será seu tratamento em termos da complexidade computacional utilizada e do número máximo de falhas admitido. Para perceber essa implicação, considere o problema de consenso, que tem sido muito utilizado para avaliar modelos de sistemas distribuídos quanto a tolerância a falhas [54, 68, 24]. Colocado de uma forma simples, o problema de consenso consiste em um grupo de pro-

cessos concordar com um determinado valor, dado que cada processo possa ter proposto um valor diferente [75]. Estudos teóricos dos modelos de falha relacionados ao consenso permitiram a determinação do número mínimo de componentes necessários para poder tolerar um determinado número f de falhas [75]. Por exemplo, no modelo de falhas *fail-stop* são necessários $f + 1$ componentes corretos para tolerar f falhas, mas no modelo bizantino são necessários $3f + 1$ componentes corretos. Por último, o leitor deve estar consciente de que apenas defeitos de componentes são tolerados, pois, obviamente, o sistema como um todo não pode ser tolerante a sua própria falha. O que precisa ser feito é evitar que um defeito em um componente ou parte do sistema, causado por uma falha, não leve ao defeito do sistema, garantindo, portanto, a execução do serviço na ocorrência de falhas.

3.7.3. Tratamento de Falhas

O conceito básico para se implementar tolerância a falhas é redundância, de hardware ou software. Em termos gerais, a redundância pode ser de duas formas: temporal ou espacial. A redudância temporal implica que após a detectção do defeito, uma ação corretiva seja feita como, por exemplo, a repetição do procedimento que gerou o defeito. Esse tipo de redundância é em geral menos aplicável a sistemas críticos, já que o procedimento de recuperação pode violar as especificações rígidas de tempo (*hard deadlines*). No caso da redundância espacial, uma mesma ação crítica é executada por vários componentes replicados. Nesse caso, existe a possibilidade da falha ser mascarada no sentido de que a falha de um componente é compensada pela ação correta de outros componentes replicados. Redundância espacial é adequado para sistemas críticos por não gerar atrasos extras nos processos de detecção e recuperação de erros. Caso a falha não seja *mascarada*, esta pode levar o sistema a um estado de erro que precisa ser tratado para não gerar defeito no sistema. Portanto, para evitar que o defeito aconteça é necessário que um processo para detecção de erro seja utilizado. Após detectar-se o erro, deve-se evitar que seus efeitos se propaguem pelo sistema e então 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*). Realizado o processamento do erro, é preciso que o componente defeituoso seja identificado e o sistema reconfigurado de modo a prevenir que tal componente venha a ocasionar novas falhas para o sistema. Os mecanismos e as fases utilizadas na tolerância a falhas estão diretamente ligados aos requisitos funcionais e temporais aos quais o sistema deve atender, podendo variar de projeto para projeto. Entretanto, pode-se identificar quatro fases básicas que, em geral, estão presentes na implementação dos mecanismos de tolerância a falhas [59]: detecção do erro (*error detection*), confinamento dos efeitos de propagação do erro (*damage confinement*), processamento do erro (*error processing*) e tratamento da falha (*fault treatment*), detalhadas a seguir. Uma boa discussão sobre essas fases pode ser encontrada também em [44].

Detecção de Erros

A detecção de erros consiste em verificar o estado do sistema a fim de detectar estados errôneos. Observe que um dado componente pode apresentar defeito, que no nível interno do sistema representa uma falha, que, por sua vez, pode gerar um erro. Uma vez que a

presença de falhas e defeitos é verificada através dos erros, a detecção de erros também pode ser referenciada por ambos, detecção de defeitos ou detecção de falhas. Nem sempre é possivel se implementar um detector de defeitos perfeito, que consiga identificar todas as falhas ocorridas, sem gerar qualquer informação errônea (*e.g.,* indicar um falha inexistente). No entanto, na prática, existem limitações, dependendo do projeto e ambientes de implementação, para o desenvolvimento de tais detectores perfeitos. A seguir listamos algumas propriedades desejadas na implementação dos detectores:

- o detector deve observar se o serviço entregue pelos componentes do sistema está de acordo com os requisitos impostos pela especificação, abstraindo a implementação interna de tais componentes. Isto impede que erros ocorridos na implementação possam ocasionar defeitos também no detector;
- o detector de erros deve ser preciso o bastante para evitar falsas suspeitas, prevenindo que componentes corretos sejam indevidamente suspeitos de erros;
- o processo de detecção de erros deve ser completo o suficiente para garantir que possíveis erros no sistema serão detectados.

Uma forma alternativa de medir a eficiência dos detectores de defeitos foi introduzina por Chen, Toueg, e Aguilera para os modelos parcialmente síncronos [95]. Eles propuseram um conjunto de métricas para a especificação de qualidade de serviço (textitQoS) de um detector de defeitos, que definem a velocidade do detector (quão rápido detecta falhas) e sua precisão (quão bem evita erros ou falsas detecções). Foram definidas três métricas principais, a saber:

- ***detection time***- define o tempo entre a falha do componente e momento da detecção;
- ***mistake recurrence time***- define o tempo entre dois erros consecutivos;
- ***mistake duration*** - define o tempo para correção de um suspeita errada do detector.

Para implementar a detecção de defeitos há várias técnicas que podem ser usadas, dependendo do projeto do sistema [59] e dos requisitos temporais de QoS. Alguns testes comumente usados são:

- **testes temporais** - os testes temporais são utilizados para verificar se os componentes do sistema estão cumprindo os prazos (*deadlines*) para realização de uma determinada tarefa. Normalmente, esta categoria de teste é realizada através de temporizadores que atuam de acordo com a especificação de tempo;
- **testes estruturais** - esse tipo de teste é utilizado quando se deseja fazer a verificação da validade da estrutura utilizada para armazenar um dado. De modo geral, esta categoria de teste faz uso de informações redundantes que possam indicar erros na estrutura do dado. Um caso especial desse teste muito usado em hardware é a adição de bits extras na estrutura de um código. Caso o código seja corrompido não haverá correspondência entre sua estrutura e a informação do conjunto de bits de checagem adicionais, e, desta forma, o erro poderá ser detectado (*e.g.,* usando bits de paridade);
- **testes de coerência** - Nessa categoria de teste é verificado se as informaçes sobre o estado do sistema estão coerentes com as faixas de consistência estabelecidas pela especificação.

- **auto-teste** - nesste tipo de teste de erro, o módulo detector de erro faz parte do próprio sistema. Através deste módulo o sistema realiza uma série de testes para verificar se algum de seus componentes está defeituoso. Via de regra, o auto-teste utiliza valores fixos de entrada e seus respectivos valores de saída armazenados em uma memória do sistema. Os valores de entrada são submetidos aos seus respectivos componentes, e a saída gerada é comparada com a saída armazenada no detector do sistema. Caso qualquer um dos componentes não produza valores iguais ao esperados para a saída, o sistema detecta que o componente em questão está com defeito. Auto-teste é muito importante para detectar defeitos do tipo *fail-silent*, mais fáceis de serem tratados. Nesse caso, ao detectar um erro (não recuperável) o componente pára de enviar qualquer saída para o resto do sistema.

Confinamento do Risco

A detecção de um erro não é instantânea, uma vez que os testes dos componentes do sistema podem consumir uma certa quantidade de tempo e, ainda, o processo de detecção pode não estar sendo ativado de forma contínua, mas em ciclos periódicos. Por estas razões pode existir um intervalo de tempo entre a ocorrência da falha e a detecção do erro. Neste intervalo, é possível que o defeito se propague do componente defeituoso para os demais componentes com os quais este interage. Portanto, é importante determinar a extensão de atuação do erro e realizar o confinamento das possíveis regiões de risco.

As suposições quanto aos limites de disseminação do erro pode ser feita de forma dinâmica, utilizando informaçes sobre o fluxo de comunicação entre os componentes,ou de forma estática, inserindo barreiras (*fire walls*) que evitem a propagação do erro antes do próximo ciclo de detecção [94, 32].

Processamento do Erro

Os mecanismos de tolerância a falhas devem processar os erros originados na ativação de uma falha. A tolerância a falhas pode ser efetivada em duas fases básicas: na primeira fase, previne-se que um erro efetivo venha a provocar um defeito e, na segunda fase, remove-se um erro latente antes que este se torne um erro efetivo (ou que este venha a se efetivar novamente) [56]. O processamento de erros pode ser feito através da recuperação (*error recovery*) ou compensação (*error compensation* ou *error masking*) do estado ao qual foi remetido o sistema na presença do erro. A recuperação do erro consiste em substituir o estado errôneo por um estado livre de erros, podendo ser feita de duas formas básicas:

- **recuperação por retrocesso** (*backward error recovery*), consiste em trazer o sistema para um estado consistente antes da ocorrência do erro.
- **recuperação por avanço** (*forward error recovery*), consiste em, na presença do erro, encontrar um estado consistente que nunca tenha ocorrido antes (ou que não tenha ocorrido após a presença do erro).

Para o processamento do erro por recuperação, um detector de erros é fundamental. Tal detector deve garantir que a detecção do erro terá a latência necessária para que o prazo

especificado para a computação seja cumprido. O processamento do erro baseado em recuperação por retrocesso tem um complicador maior em sistemas de tempo-real, uma vez que o comportamento do sistema é ditado pelas mudanças no ambiente. Dessa forma a consistência de um estado está amarrada à validade da imagem do objeto controlado. Portanto, o uso de tal técnica pode levar a um estado que não é mais consistente devido ao mapeamento da imagem do ambiente no qual o sistema de tempo-real está inserido. O processamento do erro latente baseado em compensação, por outro lado, é feito através do uso de redundância de forma a permitir a entrega do serviço mesmo na presença do estado errôneo.

Tratamento da Falha

Uma vez que o erro foi completamente processado e o sistema encontra-se em um estado livre de erros, é preciso assegurar que uma mesma falha não irá ocasionar o mesmo erro de forma recorrente. Se a falha motriz do erro for uma falha transiente, em alguns casos o simples processamento do erro pode ser o bastante para que o sistema possa ser colocado novamente em atividade. Entretanto, se a falha for do tipo intermitente ou permanente, será necessário algum procedimento para tratamento da falha antes que se possa dar continuidade ao serviço. O tratamento da falha exige que o componente defeituoso seja identificado e possivelmente reparado. Caso o reparo do componente defeituoso não seja possível deve existir um mecanismo que permita a reconfiguração do sistema de modo que tal componente defeituoso seja excluído da configuração, ou ainda que o mesmo seja colocado ativo em uma configuração na qual não possa mais provocar o defeito.

3.7.4. Mecanismos de Tolerância a Falhas

Como mencionado anteriormente, redundância é o mecanismo básico de se implementar tolerância a falhas. Nos sistemas críticos, podemos destacar dois mecanismos básicos, TMR (*Triple Modular Redundance*) no nível do hardware, e replicação ativa - onde componentes de software são replicados em processadores distintos.

O conceito de TMR foi inicialmente sugerido por Von Neumann e consiste na triplicação de unidades de hardware que trabalham em paralelo. As saídas das unidades são passadas em paralelo para uma unidade de votação que repassa a saída majoritária. Observem que nenhuma medida de detecção do erro é necessária para mascarar a falha de um dos componentes. Contudo, se duas unidades falharem, não será possível o mascaramento das falhas. O modelo TMR foi depois generalizado para N unidades e chamado de NMR. O modelo TMR é especialmente adequado para falhas transientes. No entanto, caso a falha seja permanente, o procedimento mais adequado é a substituição da unidade com defeito.

Replicação ativa é muito usada em sistemas distribuídos de tempo-real críticos. Para que replicação ativa funcione, as réplicas precisam apresentar um comportamento chamado *determinismo de réplica*, em que todas as réplicas apresentam o mesmo estado em momentos aproximadamente iguais no tempo-real [83]. Especial atenção deve ser dada para garantir a consistência na presença de falhas de um dos membros do grupo de réplicas ou, ainda, na presença de falhas nos canais de comunicação. O defeito em

uma réplica deve ser manipulado de modo que o processamento realizado pelas demais réplicas compense o defeito e possibilite o mascaramento da falha. Para o defeito no canal de comunicação, entretanto, o algoritmo de controle deve estar apto para gerenciar falhas que levem ao particionamento da rede. Uma rede particionada pode gerar grupos de réplicas isolados de maneira que os grupos não possam se comunicar. Partições podem ser danosas, uma vez que na ausência de comunicação entre grupos de réplicas não se pode assegurar a consistência das ações ou dos estados.

Uma abordagem chamada de máquinas de estato (*state machine approach*) é apresentada por Schneider em [83], que descreve os mecanismos básicos para implementação de replicação ativa. Nessa abordagem, as requisições realizadas nas réplicas são processadas de forma seqüêncial, obedecendo a uma relação de causalidade, de modo que as saídas produzidas pelas réplicas são completamente determinadas pela seqüência de requisições.

O número de réplicas utilizada para tolerar um dado número f de falhas, dependerá do Modelo de Falhas apresentado pelos componentes replicados. Basicamente, existem três possibilidades distintas: os componetes apresentam um comportamento silencioso (*fail-silent*), podem apresentar falhas de valor, mas com determinismo de réplica e, por último, falhas bizantinas. No primeiro caso, $f + 1$ réplicas são suficientes para tolerar f falhas. No segundo é necessário uma votação pela maioria para tolerar a falha de $2f + 1$ componentes. Nesse caso, NMR pode ser usado para interligar as réplicas. Para falhas bizantinas, são necessárias $3f + 1$ [54]. Além disso, no caso de falhas bizantinas, cada componente tem que estar ligado a todos os outros componentes por $f + 1$ canais separados. Um caso típico de aplicação de replicação bizantina é na implementação de sincronização de relógios, visto que relógios costumam apresentar um comportamento arbitrário de falhas [31].

Uma forma de se desenvolver um sistema distribuído de tempo-real é usar a replicação ativa para implementar processadores distribuídos com semântica de *fail-silent*. Assim, um processador seria replicado e quando falhas não puderem ser mascaradas, a saída do conjunto de réplicas que representa um dado processador é forçada a ficar silenciosa. Ou seja, nenhuma saída será mais gerada. Essa estratégia foi usada, por exemplo, no sistema TTP/MARS [52]. Um problema que surge em decorrência dessa abordagem é a inserção e remoção de novas unidades de processadores replicados. Para isso se utiliza de um outro mecanismo chamado de *processor member* [27]. Com esse mecanismo, chamado de serviço de *membership*, defeitos das unidades tolerantes a falhas são reportados de forma mutuamente consistente entre as unidades operacionais. Garantir um pequeno atraso entre o momento de inserção ou remoção de uma unidade e o pronto conhecimento desse fato pelas demais unidades é um problema fundamental para garantir o correto funcionamento de aplicações de tempo-real críticas. No processo de inserção de uma nova unidade ou reintegração de uma unidade que falhou, um problema fundamental é garantir que o estado da unidade a ser inserida esteja consistente com as demais unidades operacionais.

Deve-se observar-se também que falhas podem ser casuadas por erros no software, de projeto ou implementação. Nesse caso, as técnicas discutidas anteriormente não podem ser diretamente aplicadas. Por exemplo, considere o esquema de replicação. Se

replicamos o código de uma tarefa que tem um erro de projeto, o mesmo erro se manifestará nas diversas réplicas de modo que nenhuma falha desse tipo será tolerada. No entando, existem alternativas para o tratamento de falhas de software. Em [77] é descrito o conceito de blocos de recuperação (*recovety blocks*), onde se utiliza pré-condições para verificar se o resultado de determinada computação está consistente com algum critério previamente conhecido, oferecendo alternativas de execução caso erros sejam detectados. Uma outra alternativa é utilizar um esquema de replicação onde cada réplica leva uma versão distinta do software, projetada e implementada por equipes de desenvolvimento distintas [5].

3.7.5. Introduzindo Tolerância a Falhas no UAV

Para garantir que o UAV cumprirá seu objetivo, é importante atribuir confiança ao seu funcionamento (*dependability*). Isto significa que tal veículo deverá alcançar a meta final mesmo na presença de falhas durante a operação. Falhas podem se manifestar tanto no hardware quanto no software. As informações providas pelo GPS, por exemplo, são importantes para a navegação da aeronave. Caso este sensor passe a produzir informações inconsistentes, a navegação do UAV poderá ser comprometida. Assumindo que as falhas dos sensores de posição baseados em GPS não são arbitrárias, pode-se utilizar um esquema TMR com três sensores para tolerar uma falha e garantir a navegação da aeronave.

A informação produzida pelo GPS não é precisa, podendo existir desvios entre as informações geradas por cada GPS. Portanto, o procedimento de seleção escolhe um valor dentro de uma faixa de valores possíveis, realizando, desta forma, uma seleção inexata. Tal seleção calcula, por exemplo, uma média dos dois valores mais próximos dentre os valores gerados pelo conjunto de sensores.

O mesmo mecanismo de redundância utilizado para os sensores de posição, pode ser aplicado para os demais componentes de hardware, com um maior ou menor número de componentes redundantes dependendo do risco de falhas. Todavia, existe uma relação de compromisso entre risco, custo e dinâmica do UAV, uma vez que a adição de novos componentes de hardware implica em novos custos e tem impactos no peso, consumo de energia e no grau de liberdade da aeronave. Para garantir o nível de tolerância a falhas são usados mecanismos de detecção de erros para informar os componentes de hardware defeituosos e possibilitar o reparo ou substituição.

Uma tarefa pode falhar devido a interferências físicas ou falhas em sua implementação. Uma falha no sistema de controle de atitude, realizado pela tarefa τ_{atd}, por exemplo, pode levar o veículo a executar ações adversas não programadas, que podem comprometer o objetivo final. Considerando que τ_{atd} falhe de forma arbitrária (bizantina), para esta tarefa pode-se utilizar um mecanismo com redundância ativa e quatro diferentes versões de τ_{atd}, de modo que falhas de software possam também ser toleradas. Cada versão da tarefa operaria em determinismo de réplica e seria executada em processadores distintos. Cada processador manteria um canal de comunicação com os diversos dispositivos sensores e atuadores da aeronave. Além disto, os processadores possuiriam, para a troca de informações entre si, canais de comunicação redundantes, de modo que as tarefas pudessem chegar a um consenso *bizantino* [54] sobre a seqüência e a ordem de cada resultado gerado. Cada versão de τ_{atd} capturaria os valores dos mesmos conjuntos

de sensores. Nesse esquema hipotético, o resultado gerado por uma τ_{atd} representa um conjunto de ações de controle, em que cada ação é identificada por um número de seqüência. As ações geradas são enviadas para as respectivas controladoras dos elementos atuadores. Cada versão de τ_{atd} envia suas ações de controle às controladoras. Assim, em condições normais, cada controladora receberia quatro cópias de uma mesma ação de controle. Entretanto, uma vez que apenas uma falha arbitrária é tolerada, cada controladora deve esperar por ao menos três ações de controle antes de decidir qual a ação a ser executada.

3.8. Considerações Finais

Garantir a previsibilidade de um sistema de tempo-real distribuído envolve uma série de técnicas e temas complementares, que vão desde a especificação e verificação formal do sistema, uso de *middleware* e linguagens de programação adequadas à tempo-real, até o de gerenciamento de recursos que garantam um comportamento previsível do ambiente computacional (hardware e software), como técnicas e análise de escalonamento e tolerância a falhas.

Este capítulo procurou apresentar uma visão integrada desses diversos aspectos de projeto de sistemas de tempo-real, mostrando como eles se inter-relacionam e interferem no processo de desenvolvimento. Para tanto, procuramos, quando possível, relacionar os diversos temas através de indicações no texto e, além disso, utilizamos um exemplo comum para ilustrar a aplicação dos conceitos cobertos nas diferentes seções. O objetivo do capítulo não foi dar uma visão profunda de cada tema, mesmo porque restrições de espaço impossibilitariam qualquer tentativa nesse sentido. Ao invés, procuramos cobrir os aspectos principais, motivando o leitor com uma abordagem suficientemente detalhada que permitisse um entendimento dos principais problemas envolvidos, deixando, no entanto, uma lista de referências bibliográfica que permitem um aprofundamento do temas pelos interessados.

Referências

[1] GI-Fachgruppe 4.4.2. PEARL 90 - Language Report - Version 2.2. Technical report, GI Gesellschaft für Informatik, Bonn, Germany, September 1998.

[2] R. Alur, C. Courcoubetis, and D. Dill. Model-Checking for Real Time Systems. In *5th Symp. on Logics in Computer Science*, pages 414–425. IEEE Computer Society Press, 1990.

[3] N. C. Audsley, A. Burns, R. Davis, K. Tindell, and A. J. Wellings. Fixed Priority Pre-Emptive Scheduling: An historical Perspective. *Real Time Systems*, 8(2):173–198, 1995.

[4] N. C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying New Scheduling Theory to Static Priority Pre-Emptive Scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[5] Algirdas Avižienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, December 1985. Special Issue on Software Reliability—Part I.

[6] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In *8th International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 244–256, July/August 1996.

[7] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL – A Tool Suite for Symbolic and Compositional Verification of Real Time Systems. In *1st Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, May 1995. 1019 of Lectures Notes in Computer Science.

[8] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer-Verlag New York, September 2001.

[9] A. Bestavros. TRA-Based Real-Time Executable Specification Using CLEOPATRA. In *10th Annual Rochester Forth Conference on Embedded Systems*, Rochester, NY, June 1990.

[10] A. Bestavros, D. Reich, and R. Popp. Cleopatra Compiler Design and Implementation. Technical Report TR-19-92, Computer Science Department, Boston University, Boston, MA, August 1992.

[11] A. P. Black, M. Carlsson, M. P. Jones, R. Kieburtz, and J. Nordlander. Timber: A Programming Language for Real-Time Embedded Systems. Technical report, OGI School of Science & Engineering, Beaverton, Oregon, April 2002.

[12] G. Bollella and J. Gosling. The Real-Time Specification for Java. *IEEE Computer*, 33(6):47–54, June 2000.

[13] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2001.

[14] J. M. Borky. Payload Technologies and Applications for Uninhabited Air Vehicles (UAVs). *IEEE*, pages 267–283, 1997.

[15] F. Boussinot and R. De Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[16] U. Brinkschulte, E. Schneider, F. Picioroaga, and A. Béchina. Distributed Real-Time Computing for Microcontrollers - The OSA+ Approach. *Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. April 29 - May 01. Washington D.C.*, 2002.

[17] U. Brinkschulte and T. Ungerer. A Microkernel Middleware Architecture for Distributed Embedded Real-Time Systems. *20th IEEE Symposium on Reliable Distributed Systems*, 2000.

[18] U. Brinkschulte and H. Vogelsang. A Distributed Scaleable Real-time Add-on for Operating Systems. *RTAS '98, 4th IEEE Real-time Technology and Application Symposium, Denver*, 1998.

[19] A. Burns. Real-Time Systems. In *Encyclopedia of Physical Science and Technology*, volume 14, pages 45–54. Academic Press, 2002.

[20] A. Burns, R. I. Davis, and S. Punnekkat. Feasibility Analysis of Fault-Tolerant Real-Time Task Sets. In *Proc. of the Euromicro Real-Time Systems Workshop*, pages 29–33. IEEE Computer Society Press, 1996.

[21] A. Burns, S. Punnekkat, L. Stringini, and D. Wright. Probabilistic Scheduling Guarantees for Fault-Tolerant Real-Time Systems. In *Proc. of the 7th Int'l Working Conference on Dependable Computing for Critical Application*, pages 339–356, 1999.

[22] A. Burns and A. J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Addison-Wesley, Reading, MA, USA, third edition, 2001.

[23] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[24] T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. Technical Report TR94-1426, Cornell University, Computer Science Department, May 1994.

[25] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, December 1996.

[26] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, January 2000.

[27] F. Cristian. Agreeing on Who Is Present and Who Is Absent in a Synchronous Distributed System. *IEEE*, pages 206–211, 1988.

[28] F. Cristian and C. Fetzer. The Timed Asynchronous Distributed System Model. In *Proceedings of the 28th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-28)*, pages 140–149, jun 1998.

[29] D. Déharbe, A. M. Moreira, L. Ribeiro, and V. M. Rodrigues. Introdução a Métodos Formais: Especifi cação, Semântica e Verifi cação de Sistemas Concorrentes, September 2000.

[30] D. Dolev, C. Dwork, and L. Stockmeyer. On the Minimal Synchronism Needed for Distributed Consensus. In *24th Annual Symposium on Foundations of Computer Science*, pages 393–402, Tucson, Arizona, 7–9 November 1983. IEEE.

[31] D. Dolev, J. Halpern, and R. Strong. On the Possibility and Impossibility of Achieving Clock Synchronization. In *Proceedings of the 16th ACM Symposium on Theory of Computation*, 1984.

[32] W. R. Dunn. Designing Safety Critical Computer Systems. *IEEE Computer Society*, pages 40–46, november 2003.

[33] C. Dwork, C. Lynch, and L. Stockmeyer. Consensus in The Presence of Partial Synchrony (Preliminary Version). Technical Memo MIT/LCS/TM-270, MIT, July 1984.

[34] J. M. Farines, J. S. Fraga, and R. S. Oliveira. *Sistemas de Tempo-Real*. Escola de Computação, 2000.

[35] W. Halang and R. Henn. Additional PEARL Language Structures for the Implementation of Reliable and Inherently Safe Real-Time Systems. In *International Federation for Real-Time Control*, 1988.

[36] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[37] K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *GPCE'2003*, 2003.

[38] K. Havelund, A. Skou, K. Larsen, and K. Lund. Formal Modeling and Analysis of An Audio/Video Protocol: An Industrial Case Study Using UPPAAL. In *The 18^{th} IEEE Real-Time Systems Symposium*, pages 2 – 13, 1997.

[39] T. A. Henziger, P. H. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. Software Tools for Technology Transfer, 1997. Disponível em http://www-cad.eecs.berkeley.edu/∼tah/HyTech/.

[40] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Embedded Control Systems Development with Giotto. In *Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES-01)*, volume 36, 8, pages 64–72, New York, June 22–23 2001. ACM Press.

[41] R. M. Howard and I. Kaminer. Survey of Unmanned Air Vehicles. *IEEE*, pages 2950–2953, 1995.

[42] P-A. Hsiung and F. Wang. A State-Graph Manipulator Tool for Real-Time System Specification and Verification. In *5^{th} International Conference on Real-Time Computing Systems and Applications - RTCSA'98*, October 1998.

[43] Int'l Standards Organisation. *ISO 11898. Road Vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High Speed Communication*, 1993.

[44] P. Jalote. *Fault Tolerance In Distributed Systems*. Prentice Hall, New Jersey, 1994.

[45] N. Kandasamy, J. P. Hayes, and B. T. Murray. Tolerating Transient Faults in Statically Scheduled Safety-Critical Embedded Systems. In *Proc. of the 18th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 212–221, 1999.

[46] K. B. Kenny and K. J. Lin. Building flexible real-time systems using the Flex language. *Computer*, 24(5):70–78, May 1991.

[47] K. H. Kim. An Efficient Middleware Architecture Supporting Time-Triggered Message-Triggered Objects and an NT-Based Implementation. *Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 1999.

[48] K. H. Kim. Object-Oriented Real-Time Distributed Programming and Support Middleware. In *Seventh International Conference on Parallel and Distributed Systems (ICPADS'00)*, pages 10–22, 2000.

[49] E. Kligerman and A. D. Stoyenko. Real-Time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12(9):941–949, September 1986.

[50] H. Kopetz. *Real-Time Systems Design for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[51] H. Kopetz. The Time-Triggered Model of Computation. In *Proc. of the 19th Real-Time Systems Symposium (RTSS)*, pages 168–177. IEEE Computer Society Press, 1998.

[52] H. Koptez. *Real-Time Systems: Design Principles for Distributed Embebed Applications*. Kluwer Academic Publishers, 1997.

[53] L. Lamport and P. M. Melliar-Smith. Synchronizing Clocks in the Presence of Faults. *Journal of the ACM*, 32(1):52–78, 1985.

[54] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. In C. J. Walter, M. M. Hugue, and Neeraj Suri, editors, *Advances in Ultra-Dependable Distributed Systems*. IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720, January 1995.

[55] G. L. Lann. What Are The Key Paradigms in The Integration of Timeliness and Availability (contribution to the panel discussion). In H. Kopetz and Y. Kakuda, editors, *Responsive Computer Systems*, volume 7 of *Dependable Computing and Fault-Tolerant Systems*, pages 327–330. Springer-Verlag Wien New York, 1993.

[56] J-C. Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. *XXV Symposium International on Faulting-Tolerant Computing*, pages 2–11, 1985.

[57] J. C. Laprie. *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag, 1992.

[58] F. Laroussinie, K.G. Larsen, and C. Weise. From Timed Automata to Logic and Back. In *MFCS 95*, 1995. Lectures Notes in Computer Science.

[59] P. A. Lee and T. Anderson. *Fault Tolerance : Principles and Practice*. Dependable Computing and Fault-Tolerant Systems. Springer-Verlag, Berlin ; New York, 1990. 2., rev. ed.

[60] G. M. A. Lima. *Fault Tolerance in Fixed-Priority Hard Real-Time Distributed Systems*. PhD thesis, Deptartment of Computer Science, University of York, 2003.

[61] G. M. A. Lima and A. Burns. An Effective Schedulability Analysis for Fault-Tolerant Hard Real-Time Systems. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 209–216. IEEE Computer Society Press, 2001.

[62] G. M. A. Lima and A. Burns. An Optimal Fixed-Priority Assignment Algorithm for Supporting Fault Tolerant Hard Real-Time Systems. *IEEE Transaction on Computers*, 52(10):1332–1346, 2003.

[63] M. Lindahl, P. Pettersson, and W. Yi. Formal Design and Analysis of a Gear Controller. In *4^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, pages 281 – 297, March 1998.

[64] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogram in a Hard Real-Time Environment. *Journal of ACM*, 20(1):40–61, 1973.

[65] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

[66] J. Lundelius and N. Lynch. An Upper and Lower Bound for Clock Synchronization. *Information and Control*, 62(2/3):190–204, August/September 1984.

[67] J. Lundelius-Welch and N. Lynch. A New Fault-Tolerant Algorithm for Clock Synchronization. *Information and Computation*, 77 (A01)(1):1–36, 1988.

[68] N. Lynch M. Fisher and M. Paterson. Impossibility of Distributed Consensus with one Faulty Process. *Journal of ACM*, pages 374–382, april 1985.

[69] Sun Microsystems. RPC: Remote Procedure Call Protocol Specifi cation. Version 2. *RFC. 1957, Sun Microsystems, Inc.*, June 1988.

[70] T. J. Mowbray and W. Ruh. *Inside CORBA: Distributed Object Atandards and Applications*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[71] http://www-3.ibm.com/software/ts/mqseries/messaging/ (data de acesso: 20/01/2004).

[72] http://www.microsoft.com/msmq/default.htm (data de acesso: 20/01/2004).

[73] P. C. Ölveczky. *Real-Time Maude 2.0 Manual*. Department of Computer Science, University of Illinois and Department of Informatics, University of Oslo, October 2003.

[74] L. Palopoli, G. Buttazo, and P. Ancilotti. ALERT: A C Language Extension for Real-Time Systems. In *IEEE RTCSA Conference*, December 1999.

[75] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. In C. J. Walter, M. M. Hugue, and Neeraj Suri, editors, *Advances in Ultra-Dependable Distributed Systems*. IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720, January 1995.

[76] P. Puschner and A. Burns. A Review of Worst-Case Execution-Time Analysis (editorial). *Real-Time Systems*, 18(2/3):115–128, 2000.

[77] B. Randell. System Structure for Software Fault Tolerance. In *Proceedings of the 1975 International Conference on Reliable Software*, pages 437–449, 1975.

[78] R. Sanz and M. Alonso. Corba for Control Systems. *Annual Reviews in Control*, vol. 25 (no. 1): pp. 169–181, 2001.

[79] D. C. Schmidt. Middleware Techniques and Optimizations for Real-Time, Embedded Systems. In *International Symposium on System Synthesis*, pages 12–17, 1999.

[80] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design of the TAO Real-Time Object Request Broker. *Computer Communications*, 21(4), 1998.

[81] F. B. Schneider. Fail-Stop Processors. In *Digest of Papers Spring Compcon 83, San Francisco*, pages 66–70. IEEE, March 1983.

[82] F. B. Schneider. A Paradigm for Reliable Clock Synchronization. Technical Report TR86-735, Cornell University, Computer Science Department, February 1986.

[83] F. B. Schneider. Implementing Fault-Tolerant Services Using The State Machine Approach: Tutorial. *ACM*, 22(4):299–319, set 1990.

[84] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-time Synchronisation. *IEEE Transaction on Computers*, 39(9):1175–1185, 1990.

[85] http://www.tibco.com/solutions/products/active_enterprise/smart/default.jsp (data de acesso: 20/01/2004).

[86] Y. Tachi and S. Yamane. Real-time symbolic Model Checking for Hard Real-Time Systems. In *Real-Time Computing Systems and Applications, Sixth International Conference on*, pages 496–499, December 1999.

[87] Andrew S. Tanenbaum. *Computer Networks (Fourth Edition)*. Prentice Hall, Englewood Cliffs, NJ, 2002.

[88] K. Tindell, A. Burns, and A. J. Wellings. Allocating Real-Time Tasks (An NP-Hard Problem Made Easy). *Real Time Systems*, 4(2):145–165, 1992.

[89] K. Tindell, A. Burns, and A. J. Wellings. An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks. *Real-Time Systems*, 4(2):145–165, 1994.

[90] K. Tindell, A. Burns, and A. J. Wellings. Calculating Controller Area Network (CAN) Message Response Times. *Control Engineering Practice*, 3(8):1163–1169, 1995.

[91] K. Tindell and J. Clark. Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. *Microprocessing and Microprogramming, Euromicro Journal*, 40:117–134, 1994.

[92] K. S. Trivedi. *Probability & Statistics with Reliability, Queuing and Computer Science Applications*. Prentice Hall, Englewood Cliffs, 1982.

[93] http://www.uavforum.com (data de acesso: 26/02/2004).

[94] P. Veríssimo and L. Rodrigues. *Distributed Systems For Systems Architects*. Kluwer Academic Publishers, USA, 2000.

[95] S. Toueg W. Chen and M. K. Aguilera. On the Quality of Service of Failure Detectores. *IEEE Transactions On Computer*, 51(1):13–32, jan 2002.

[96] S. Yovine. Kronos: A Verifi cation tool for Real-Time Systems. Software Tools for Technology Transfer, 1997. Disponível em http://www-verimag.imag.fr/TEMPORISE/kronos/.