UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Protocolo de Recuperação por Retorno, Coordenado, Não Determinístico

por

SÉRGIO LUIS CECHIN

Tese submetida à avaliação,
como requisito parcial para a obtenção do grau de
Doutor em Ciência da Computação

Profa. Dra. Ingrid Jansch-Pôrto
Orientadora

Porto Alegre, abril de 2002

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Cechin, Sérgio Luis

Protocolo de Recuperação por Retorno, Coordenado, Não Determinístico / por Sérgio Luis Cechin. — Porto Alegre: PPGC da UFRGS, 2002.

142 f.: il.

Tese (doutorado) — Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2002. Orientador: Jansch-Pôrto, Ingrid.

1. Tolerância a falhas. 2. Sistemas distribuídos. 3. Recuperação de processos. 4. Checkpoint. I. Jansch-Pôrto, Ingrid. II. Título.

Dedico este trabalho a minha esposa e meus filhos,
causas principais de minha dedicação.

Agradecimentos

Agradeço à ***Deus***, origem e destino de tudo.

Agradeço à minha esposa e meus filhos, pelo *amor* que me sustentou durante as tempestades do caminho, e pela *compreensão*, mesmo quando não podia ver este amor.

Agradeço a meus pais, por estar aqui, e porque sempre me *incentivaram* a buscar o saber, mesmo nas séries mais básicas.

Agradeço a minha orientadora, e agora também colega, a professora Ingrid, pela *compreensão* e *paciência* com minhas deficiências, e pelas *oportunidades* que me possibilitou, durante o doutorado.

Agradeço aos professores Luiz Eduardo Buzato, Taisy Weber e Cláudio Geyer, componentes da banca de avaliação da proposta de tese, pelas sugestões e redirecionamento do trabalho.

Agradeço ao colega Raul Ceretta, com quem estabeleci *amizade* e com quem muitos assuntos foram aprofundados, em discussões sempre construtivas.

Agradeço também ao colega e amigo Adriano Fontoura, com quem tive a *oportunidade* de interagir como co-orientador informal no seu trabalho de mestrado, concluído com o maior sucesso.

Agradeço a todos os colegas do curso, pelo *companheirismo* que nunca faltou.

Agradeço a UFRGS, ao Instituto de Informática e ao Programa de Pós-Graduação em Computação, que propiciaram-me cursar e concluir o doutorado.

Sumário

Lista de Abreviaturas

FIFO *First-In First-Out*

IP *Internet Protocol*

PWD *Piecewise Deterministic*

RDT *Rollback Dependency Trackability*

RLE *Recovery Line Events*

RLM *Recovery Line Messages*

RTLA *Raw Temporal Logic of Actions*

sse se e somente se

TCP *Transmission Control Protocol*

TLA *Temporal Logic of Actions*

TLC *TLA+ Model Checker*

Lista de Figuras

Lista de Tabelas

Resumo

O uso da recuperação de processos para obter sistemas computacionais tolerantes a falhas não é um assunto novo. Entretanto, a discussão de algoritmos para a recuperação em sistemas distribuídos, notadamente aqueles que se enquadram na categoria assíncrona, ainda encontra pontos em aberto. Este é o contexto do presente trabalho.

Este trabalho apresenta um novo algoritmo de recuperação por retorno, em sistemas distribuídos. O algoritmo proposto é do tipo coordenado, e seus mecanismos componentes determinam que seja classificado como um algoritmo baseado em índices (*index-based coordinated*). Desta forma, a tolerância a falhas é obtida através do estabelecimento de linhas de recuperação, o que possibilita um retorno consideravelmente rápido, em caso de falha.

Seu desenvolvimento foi feito com o objetivo de minimizar o impacto ao desempenho do sistema, tanto quando este estiver operando livre de falhas como quando ocorrerem as falhas. Além disso, os mecanismos componentes do algoritmo foram escolhidos visando facilitar a futura tarefa de implementação.

A satisfação dos objetivos decorre principalmente de uma importante característica assegurada pelos mecanismos propostos no algoritmo: o não bloqueio da aplicação, enquanto é estabelecida uma nova linha de recuperação. Esta característica, associada ao rápido retorno, oferece uma solução promissora, em termos de eficiência, para a recuperação, um vez que o impacto no desempenho tende a ser reduzido, quando o sistema encontra-se operando em ambas condições: livre de erros ou sob falha.

Diferentemente da maioria dos algoritmos coordenados encontrados na literatura, o algoritmo proposto neste trabalho trata as mensagens perdidas. A partir da análise das características das aplicações, bem como dos canais de comunicação, quando estes interagem com o algoritmo de recuperação, concluiu-se que os procedimentos usados para recuperação de processos devem prever o tratamento desta categoria de mensagens. Assim, o algoritmo proposto foi incrementado com um mecanismo para tratamento das mensagens que têm o potencial de tornarem-se perdidas, em caso de retorno, ou seja, evita a existência de mensagens perdidas.

Uma das decisões tomadas durante o desenvolvimento do algoritmo foi a de permitir um processamento não determinístico. Na realidade, esta escolha visou o aumento do espectro das falhas que poderiam ser tratadas pela recuperação. Tradicionalmente, a recuperação por retorno é empregada para tolerar falhas temporárias. Entretanto, a diversidade de ambiente, freqüente nos SDs, também pode ser usada para tolerar algumas falhas permanentes.

Para verificar a correção do algoritmo, decidiu-se empregar um formalismo existente. Assim, a lógica temporal de Lamport (TLA) foi usada na especificação dos mecanismos do algoritmo bem como em sua demonstração de correção.

O tratamento referente às mensagens perdidas, através do uso de mensagens de resposta, associado com o uso de uma lógica temporal, levou à necessidade de rever os critérios de consistência. Esta revisão gerou um conjunto de fórmulas de consistência ajustadas à existência de mensagens de diferentes classes: mensagens da aplicação e mensagens de resposta.

Palavras-chave: Tolerância a falhas, sistemas distribuídos, recuperação de pro-

cessos, checkpoint.

TITLE: "NON DETERMINISTIC AND COORDINATED ROLLBACK RECOVERY PROTOCOL"

Abstract

Rollback-recovering process is not a new approach to augment computational systems with fault tolerance. Nevertheless, solution space on distributed recovery algorithms, mainly asynchronous, is not totally covered.

This thesis presents a new distributed rollback recovery algorithm.

The algorithm is an index-based coordinated one. Thus, the fault tolerance is obtained by taking recovery lines, which enables a fast rollback, in case of failures.

We have decided to allow nondeterministic processing. In fact, this choice has aimed to enlarge rollback-recovery failure spectrum. Rollback-recovery was designed to tolerate transient faults. Nevertheless, we can use rollback-recovery added to environment diversity to tolerate some permanent faults.

It was developed to minimize performance overhead on failure-free and under fault occurrence. Furthermore, the mechanisms of the algorithm were devised to allow an easy implementation.

The proposed algorithm has an important characteristic: checkpointing operation does not need to block the application. Fast rollback plus non-blocking checkpointing give us a pretty efficient recovery as performance overhead is reduced at both failure-free and faulty operation.

Unlike most of the coordinated algorithms from the literature, we deal with lost messages category. We have analyzed application and communication channels interaction with rollback-recovery algorithms and conclude that rollback-recovery algorithms have to deal with this category of messages. Thus, we augment our algorithm to deal lost messages category (in fact, potentially lost messages).

We decide to use a pre-existent formalism to verify the correctness of the algorithm. Thus, we have chosen the temporal logic of Lamport (TLA), which was used to write both the specification and its correction proof.

The handling of the lost messages associated with the use of a temporal logic, caused to need to review the criterion of consistency. This review produces a consistency set of formulas well fitted to different classes of messages: application messages and answer messages.

Keywords: fault tolerance, distributed systems, processes rollback-recovery, checkpointing.

1 Introdução

Um sistema apresenta defeito quando seu comportamento não é coerente com o proposto por sua especificação. Sabe-se que defeitos são conseqüência de falhas. Assim, para tratar estes possíveis cenários de falha, pode-se recorrer a dois enfoques [AVI 76, LAP 85, JAL 94]: **prevenção de falhas** e **tolerância a falhas**. No enfoque da prevenção de falhas, tenta-se impedir que elas ocorram ou sejam introduzidas aos sistemas. Entretanto, não é possível antecipar e eliminar todas as falhas.

Com a tolerância a falhas, o objetivo é garantir a continuidade do serviço, mesmo na presença de falhas. Entretanto, para que os procedimentos de tolerância a falhas sejam efetivos, é necessário conhecer o comportamento das falhas, ou seja, é necessário que um modelo de falhas seja definido. Com isso em vista, Anderson e Lee [AND 81] propuseram uma taxonomia através da qual as falhas são classificadas como temporárias (*transient faults*), quando estiverem presentes por um período limitado de tempo, ou permanentes (*permanent faults*), quando estiverem presentes por um período de tempo maior que um valor especificado. Para cada tipo de falha, deve ser aplicado um tratamento adequado.

No caso das falhas temporárias, os meios de obtenção de segurança de funcionamento (ou dependabilidade) valem-se do fato que elas se manifestam e algum tempo depois desaparecem, permitindo que o sistema retorne à operação normal, sem exigir a intervenção direta sobre a origem do problema, mas obtendo um comportamento tolerante a falhas. Este é o caso da recuperação por retorno.

Na recuperação por retorno, o processamento é desviado para um estado anterior ao da ocorrência da falha. Um ponto de retorno óbvio corresponderia ao início do processamento, ou seja, a aplicação seria reiniciada, após a ocorrência da falha e suas conseqüências. Esta não é uma situação aceitável, principalmente em se tratando de aplicações que apresentam um tempo longo de processamento, como é o caso das aplicação científicas. Seria mais razoável um ponto de retorno anterior à falha, com pequena perda de processamento.

No caso das aplicações em que existe apenas um processo, um ponto de retorno poderá ser estabelecido em qualquer estado da computação anterior à ocorrência da falha. Neste tipo de aplicação, o conceito de estado de retorno confunde-se com o conceito de estado anterior.

Nas aplicações distribuídas, onde vários processos estão presentes, é necessária uma caracterização mais cuidadosa de qual estado servirá como ponto de reinício da computação. Este cuidado diz respeito, principalmente, à consistência de um ponto de recuperação global (conjunto de pontos de recuperação locais, um por processo). Para obter um ponto de recuperação global, cada nodo poderia ser programado para estabelecer pontos de recuperação local, de forma independente. Entretanto, conforme será discutido na seção sobre consistência, um conjunto de pontos estabelecidos desta forma pode não ser consistente. Na realidade, não existe um método direto para estabelecer pontos de recuperação, ao mesmo tempo, em todos os nodos [JAL 94]. Obviamente, se existisse, os pontos de recuperação obtidos por este mecanismo formariam um ponto de recuperação global e consistente. Assim, é necessário utilizar algum outro mecanismo para estabelecer um ponto de recuperação global.

Os métodos usados para garantir a consistência de um ponto de recuperação global podem ser subdivididos nos que buscam garantir a consistência no momento

do estabelecimento dos mesmos e naqueles que adiam a procura por um conjunto consistente, até que seja necessário (quando for necessário o retorno). Neste segundo caso, admitindo-se que a computação seja determinística, isto é, que o processamento seja feito sempre da mesma forma (desde que sejam mantidas as mesmas variáveis de entrada), o estado de retomada certamente deverá ser um que já tenha ocorrido. Entretanto, caso o processamento seja não-determinístico, é possível encontrar-se estados que não tenham existido em uma computação específica mas que poderiam ter ocorrido em outra computação válida [CHA 85], podendo ser utilizados para o reinício do processamento. Esta observação permite aumentar as possibilidades de que um conjunto de pontos locais seja consistente, podendo, desta forma, ser utilizado para o retorno do sistema.

As atividades de verificação e de salvamento de informações necessárias ao reinício do processamento são denominadas de **estabelecimento de pontos de recuperação** (*checkpointing*). Nestas atividades, os processos têm seus respectivos estados de processamento salvos de forma a poderem ser utilizados no caso da ocorrência de falhas. Quando o sistema possui um único usuário, basta que o estado do processamento seja salvo de tempos em tempos e, no caso de falha, este último estado seja restabelecido de maneira que o processamento possa ser retomado. Nos sistemas distribuídos, esta atividade não é tão simples. Deve-se determinar como os processos podem salvar os seus estados de forma a garantir que, em conjunto, formem um estado que possa ser utilizado para o retorno do sistema, em caso de falha (conjunto útil).

Uma forma de obter um conjunto útil é através de um mecanismo de coordenação entre os processos, de forma que, garantidamente, os estados locais salvos por eles corresponderão ao mesmo ponto da computação distribuída. Antes de salvar os seus estados, os processos devem trocar informações de maneira a assegurar que o conjunto de estados possa ser utilizado como ponto de reinício da computação, em caso de uma falha posterior.

Uma das formas de coordenar os processos que formam o sistema distribuído nas atividades de estabelecimento de um pontos de recuperação é através da sincronização dos mesmos. Neste caso, se um processo decidir estabelecer um ponto de recuperação, este fato deverá ser informado para todos os demais. Além disso, o sistema só poderá continuar o processamento da aplicação após todos terem completado a tarefa de estabelecimento de seus pontos de recuperação.

Em operação normal (sem falhas), o tempo gasto com as atividades de estabelecimento dos pontos de recuperação implica em uma redução no desempenho do sistema, uma vez que os processos estarão envolvidos com a tolerância a falhas e não com a computação da aplicação, durante estes períodos. Ainda, nesta mesma hipótese de não ocorrerem falhas, o sistema teve uma redução na produtividade resultante do processamento e não obteve um resultado efetivo. Entretanto, caso ocorram falhas, o tempo gasto com os salvamentos dos pontos de retorno será plenamente justificado, se tiver sido adequadamente planejado.

O impacto ao desempenho dos sistemas devido ao estabelecimento dos pontos de recuperação tende a ser maior quanto menor for a taxa de falhas. Desta forma, para um mesmo sistema, quanto mais **independente** for o mecanismo usado no estabelecimento dos pontos de recuperação, menor será o impacto sobre o desempenho, se comparado relativamente aos demais mecanismos.

No caso extremo de um mecanismo totalmente independente, não existe a

coordenação e os processos salvam seus pontos de retorno sem qualquer troca de informação entre si. Além disso, quando ocorrem as falhas, os processos devem procurar, dentre os pontos locais salvos, um conjunto coerente e que possibilite a menor perda possível de processamento. Caso não encontrem, resta ao sistema a necessidade de retorno para o início da computação. Este efeito é chamado de **efeito dominó**. Assim, apesar de ter evitado o custo da coordenação e ter ficado, apenas, com o de salvamento dos pontos de recuperação, o sistema não obteve nenhum ganho. Na realidade, o desempenho ficou pior do que se não fosse usado nenhum mecanismo de tolerância a falhas.

O desenvolvimento dos algoritmos tem levado à necessidade de que os programadores considerem um volume crescente de detalhes na sua implementação. Isso pode ser identificado quando se observam alguns pontos de referência na evolução desde o trabalho de Chandy e Lamport [CHA 85], passando pelo trabalho de Koo e Toueg [KOO 87] e finalizando em trabalhos atuais como o de Prakash e Singhal [PRA 96].

Entretanto, mesmo com esta complexidade, algumas premissas não condizentes com aspectos práticos de implementação dos sistemas distribuídos continuam a ser utilizadas. Na realidade, o uso destas premissas na definição dos protocolos não é incorreto. Entretanto, estas premissas podem causar sérias dificuldades àqueles que se propõem a implementar os algoritmos que as usam. São exemplos a premissa da detecção imediata de defeitos e o tratamento das mensagens perdidas pelos protocolos caracterizados como "de comunicação confiável".

Devido, principalmente, ao que foi discutido, este trabalho visa apresentar um novo algoritmo para a recuperação de processos, por retorno, em sistemas distribuídos. Este algoritmo deverá possibilitar uma implementação simples porém robusta, oferecendo um reduzido impacto ao desempenho do sistema, quando estiver operando em condição livre de erros. Quando ocorrerem erros, o algoritmo deverá propiciar a identificação rápida de uma linha de recuperação.

O algoritmo proposto tem, basicamente, três características: recuperação por retorno, estabelecimento coordenado dos pontos de recuperação e possibilidade de computação não determinística, por parte da aplicação.

A recuperação por **retorno**, diferentemente da recuperação por avanço, pode ser usada quando a natureza das falhas não é conhecida ou não é possível identificar a extensão de todos os erros por elas causados. Uma de suas características é o salvamento preventivo de estados de computação para uso posterior, em caso de ocorrência de falhas, como ponto de retomada da computação. Assim, a recuperação por retorno é uma técnica de tolerância a falhas bastante genérica, o que possibilita sua utilização em uma série de aplicações. Entretanto, a utilização desta técnica não é aconselhável quando, por exemplo, o sistema é do tipo tempo real ou de controle de um processo que não admite o retorno. Um exemplo clássico é o controle de trajetória de um avião.

O algoritmo proposto é do tipo **coordenado**. Entretanto, devido à forma como o algoritmo alcança a coordenação, não é necessário bloquear a aplicação. Assim, a computação pode continuar a ocorrer (exceto durante o salvamento do estado do processo em memória estável), mesmo durante a execução do mecanismo de estabelecimento dos pontos de recuperação.

A aplicação suportada pelo algoritmo proposto **não necessita ser determinística**. Ou seja, quando o sistema retornar para um estado previamente arma-

zenado e retomar o processamento, este pode não ser igual àquele que levou à falha. Com isso, flexibiliza-se a operação da aplicação, evitando premissas difíceis de serem garantidas, como o caso da repetição exata de processamento. Desta forma, um processamento diferente oferece um tipo de **diversidade de ambiente**, que pode permitir suplantar, além das falhas permanentes de *hardware*, até mesmo falhas de *software* [WAN 95].

1.1 Objetivo e metodologia definida

É objetivo deste trabalho que, ao final, tenha sido definido um novo algoritmo para recuperação de processos em ambiente distribuído. Para balizar o desenvolvimento deste algoritmo, foram estabelecidos alguns requisitos a serem alcançados. O algoritmo deve:

- ter assegurado formalmente o cumprimento de suas propriedades fundamentais;
- evitar recorrer a protocolos auxiliares ou de suporte;
- causar pequeno impacto ao desempenho;
- possibilitar uma implementação simplificada.

Dos requisitos apresentados, sobressaem duas preocupações principais: a prova formal e aspectos relacionados à implementação.

A proposta de um algoritmo deve ser subsidiada pela demonstração de sua correção funcional. A simples demonstração de casos de teste a partir de uma implementação não tem cobertura suficiente. Assim, uma maneira de se alcançar esta comprovação é a formalização e prova de correção do algoritmo. A formalização permite uma descrição precisa dos mecanismos definidos pelos algoritmos enquanto que a prova de correção oferece uma verificação das propriedades a serem oferecidas.

A proposta de um novo algoritmo deve ser feita de maneira a oferecer uma opção de implementação. Assim, um algoritmo definido com premissas difíceis de serem mantidas ou verificadas tem pouca utilidade prática.

Um dos fatores que devem ser considerados na escolha de um algoritmo é a simplicidade de implementação. Algoritmos que requerem o gerenciamento de várias listas em memória ou estruturas complexas associadas às mensagens ou ainda com mecanismos altamente sofisticados (e complexos) de suporte às decisões são difíceis de serem depurados e podem levar a erros de implementação. Para evitar estes problemas, é conveniente recorrer a algoritmos simples ou que utilizem mecanismos bem conhecidos.

Para formalizar e provar o algoritmo proposto, foi escolhida uma lógica temporal. Muito da decisão de especificar e verificar usando uma lógica estabelecida foi devido à observação de que, em geral, seja pelo desconhecimento de ferramentas formais, seja pelas limitações de espaço nos artigos, os algoritmos são especificados e verificados usando lógicas próprias. Este fato, por si, não apresenta problema algum. Entretanto, pode-se listar alguns pontos fracos deste enfoque:

- particularidade da especificação;

- heterogeneidade de representação de elementos semelhantes;
- maior probabilidade de escrever provas com erros;
- mistura da base lógica com a descrição do algoritmo.

Uma especificação onde toda a base deve ser construída determina uma dificuldade adicional para quem a analisa. Além do algoritmo proposto, o leitor deve estudar as bases da lógica usada na formalização. Quando o leitor conhece a linguagem, entretanto, pode concentrar-se no algoritmo descrito. Isso é muito semelhante a um programador que analisa um trecho de código escrito em uma linguagem que domine.

As especificações escritas em linguagens pré-estabelecidas têm a vantagem de gerar especificações relativamente uniformes: para representar elementos semelhantes são usadas expressões semelhantes. Isso, novamente, facilita a leitura da especificação.

Para construir as provas formais, são usadas regras de prova. Estas, por sua vez, e no que diz respeito a sua correção, são pouco sujeitas a dúvidas. Em contrapartida, as provas onde são usadas poucas regras sintáticas, ou seja, seguem uma seqüência de explicações textuais do tipo "*se* ocorrer isso, *então* aquilo", podem levar facilmente a erros, por terem sido esquecidas algumas possibilidades.

Finalmente, quando toda a base lógica deve ser construída, é provável que não exista uma separação clara entre a lógica e o algoritmo descrito. Além disso, esta base lógica, em geral, não é verificada, exceto talvez no contexto específico do algoritmo que está sendo verificado. Em geral, questões como, por exemplo, *soundness*[1] e *completeness*[2] [MAN 90] não são consideradas. Ao usar uma linguagem formal, a interface entre as construções básicas e as do algoritmo são claras. Adicionalmente, as construções básicas já foram verificadas.

Por todos os motivos acima, conclui-se que, no mínimo, o uso de uma linguagem formal pré-estabelecida favorece a especificação e a verificação, onde as chances de erros são bastante reduzidas.

Para obter um algoritmo simples, é comum utilizar-se a premissa de que algumas propriedades sejam garantidas por protocolos de suporte. Entretanto, a simplicidade alcançada com este enfoque tem seu preço. As propriedades requisitadas, em geral, são definidas de forma ideal. Para definir estas propriedades, são utilizados termos como *sempre*, *nunca*, etc, não necessariamente assegurados na prática. Um exemplo típico é a comunicação confiável, que pressupõe a garantia de entrega de todas as mensagens enviadas: as mensagens serão entregues **sempre**. Entretanto, opções disponíveis de implementação oferecem modalidades viáveis economicamente, cujas otimizações as vezes são obtidas à custa de perdas em casos pouco prováveis ou extremos. Desta forma, as implementações dos protocolos de suporte não conseguem responder às necessidades do algoritmo projetado, causando defeitos. Adicionalmente, mesmo que os protocolos de suporte satisfaçam, plenamente, as necessidades do algoritmo projetado, ainda pode-se citar uma interface (forma de acesso aos procedimentos do protocolo de suporte e documentação precisa dos

[1] no processo de dedução, toda fórmula que retornar VERDADEIRO ou FALSO será uma fórmula válida ou não, respectivamente.

[2] no processo de dedução, toda fórmula válida retornará VERDADEIRO.

mecanismos utilizados) mal definida como fator que contribui para o aparecimento de defeitos de operação.

Apesar dos requisitos anteriores oferecerem um algoritmo bastante robusto, é razoável supor-se que as aplicações busquem o melhor desempenho possível. Desta forma, deve-se considerar o custo da inclusão de tolerância a falhas quando comparado com o ganho final. Este ganho poderá ser expresso pela velocidade global de processamento, o que facilita a comparação dos custos. Entretanto, o ganho final esperado pode ser medido em termos de vidas humanas. Neste caso, a análise do impacto da tolerância a falhas pode não ser importante. De qualquer forma, mesmo que não seja o fator mais importante, critérios de desempenho, certamente, farão parte dos requisitos do sistema.

O presente texto descreve o resultado do trabalho de doutorado: a descrição de um novo algoritmo para recuperação de processos bem como a sua prova de correção.

Nas seções que seguem, dentro deste capítulo, serão discutidos o modelo de sistema distribuído, o critério de consistência e será apresentada uma classificação dos algoritmos de estabelecimento de pontos de recuperação e de recuperação.

No capítulo 2 serão discutidos os mecanismos e premissas usados na proposta do algoritmo de recuperação: coordenação, tipos de aplicação, tratamento de *livelocks* e de mensagens perdidas e a anexação de informação às mensagens.

O capítulo 3 apresenta uma introdução a Lógica Temporal de Ações – TLA – de Lamport. Este capítulo é especialmente interessante como introdução aos conceitos básicos da especificação e verificação formal, usando TLA.

A descrição informal do algoritmo é apresentada no capítulo 4 e uma comparação desta proposta com algoritmos semelhantes pode ser encontrada no capítulo 5.

Para fechamento da apresentação do algoritmo, no capítulo 6, é descrito formalmente o critério de consistência, a especificação do algoritmo e uma série de indicações de como foi desenvolvida a sua prova de correção. O conjunto completo das provas pode ser encontrado no relatório técnico [CEC 98].

Finalmente, no capítulo 7, estão resumidos os pontos mais importantes, as contribuições do trabalho e o rumo que se pretende seguir, para dar continuidade ao trabalho.

1.2 Modelo de sistema distribuído

O modelo de sistema distribuído que será utilizado neste trabalho é formado por um conjunto de **processadores** que efetuam uma **computação distribuída**. Para isso, trocam informações exclusivamente através de mensagens, enviadas através de uma **rede de comunicação**.

Os processos não possuem nenhuma outra forma para a troca de informações (por exemplo, memória compartilhada, relógios sincronizados, etc...) que não seja a troca de mensagens.

1.2.1 Modelo de processador

Por hipótese, os processadores são unidades de processamento da computação distribuída que executam **apenas um processo por vez**. Desta forma, será usado

o termo processador como sinônimo de processo. As velocidades relativas dos processos não são limitadas e estes processadores falham segundo um modelo de colapso (*crash*): estas falhas causam a parada total de processamento ou a perda do estado interno do processo. Além disso, os processos jamais executam uma transição de estado incorreta. Ainda, para que seja efetiva a recuperação por retorno, as falhas devem ser do tipo temporárias: seu efeito desaparece após algum tempo (o defeito foi reparado ou o componente foi substituído) [SCH 83].

Os processadores operam com dois tipos de memória: **estável** e **volátil**. Estas são caracterizadas segundo a garantia de seu conteúdo e quanto ao tempo de acesso.

A memória estável é aquela que, quando o efeito de uma falha tiver passado, não terá sido erroneamente alterada. Diz-se que as informações não são perdidas por efeito da falha. Em geral, apresenta um tempo de acesso relativamente longo.

Quanto à memória volátil, não há garantia de seu conteúdo, após a ocorrência de uma falha. Seu tempo de acesso é relativamente curto.

1.2.2 Modelo de rede de comunicação

A rede de comunicação é formada por canais que permitem a troca de informações (mensagens) entre processadores do sistema distribuído.

Apesar dos processadores, na prática, não estarem totalmente conectados pela rede física, será usado um modelo lógico [JAL 94] de canais **unidirecionais**, do tipo **assíncrono**, interligando cada par de processos do sistema. Assim, cada dupla de processos possui associados dois canais que permitem a troca bidirecional de mensagens. A utilização de canais unidirecionais possibilita, ao receptor de uma mensagem, identificar o transmissor.

Os canais são assíncronos no sentido que introduzem um atraso imprevisível porém finito às mensagens.

1.2.3 Modelo de computação distribuída

A computação é formada por uma seqüência de eventos não-determinísticos que iniciam estados determinísticos, nos quais pode ocorrer o envio de mensagens.

Cada processo será formado, ao menos, pelos seguintes módulos (vide figura 1.1):

- **aplicação**: módulo onde é executado o processamento da aplicação distribuída;
- **recuperação**: módulo responsável pelas atividades de tolerância a falhas;
- **detector de defeitos**: módulo responsável pela detecção e informação da ocorrência de defeitos;
- **comunicação**: módulo responsável pela troca de informações entre os processos: envio e recepção de mensagens através dos canais.

A troca de informações entre os processos é efetuada através de mensagens. Os processos podem assumir o papel de transmissor ou receptor de mensagens. Na figura 1.1, a um dos processos foi atribuído o papel de transmissor de uma mensagem e ao outro, o papel de receptor, apenas para simplificar a análise. Estas

mensagens originam-se no módulo da aplicação do transmissor, passam pelo módulo de recuperação e em seguida pelo de comunicação. Após passarem pela rede de comunicação, atingem o receptor e chegam ao respectivo módulo da aplicação. Nesta parte do trajeto, passam, em primeiro lugar, pelo módulo de comunicação e, em seguida, pelo módulo de recuperação.

A troca de mensagens e a operação de cada processo são monitoradas pelo módulo detector de defeitos.

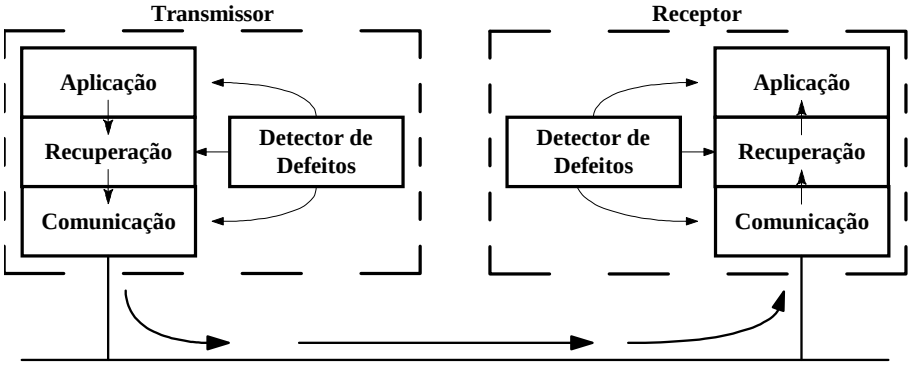

FIGURA 1.1 – Modelo de Computação Distribuída

O módulo da aplicação é aquele responsável por executar o processamento que implementa a especificação das funções planejadas para sistema. Este módulo origina e processa mensagens que serão denominadas de **mensagens de aplicação**.

No módulo de recuperação estará sendo executado o código de estabelecimento dos pontos de recuperação e, em caso de falhas, o código de retorno. Este módulo também pode originar e processar mensagens, específicas dos procedimentos de tolerância a falhas, e que serão chamadas de **mensagens de sistema**. Este módulo pode também utilizar-se das mensagens da aplicação para repassar informações aos outros processos. Isso é feito anexando-se as informações às mensagens da aplicação.

Do ponto de vista do módulo de recuperação, as ações relacionadas com a troca de mensagens podem gerar três tipos de eventos: transmissão, recepção e entrega.

Na **transmissão**, uma mensagem é passada para o módulo de comunicação do processo transmissor, de maneira a ser transmitida pela rede de comunicação.

Na **recepção**, uma mensagem recebida pelo módulo de comunicação do receptor é passada para o respectivo módulo de recuperação. Este, por sua vez, **entrega** a mensagem para o módulo da aplicação.

Supõe-se que o módulo detector de defeitos seja capaz de atuar tanto nos processos como na comunicação, informando o restante do sistema da ocorrência de falhas nestes componentes. Além disso, o módulo detector deve ser capaz de informar a restauração de um componente defeituoso, de forma que sejam iniciados os procedimentos de recuperação.

1.3 Consistência

O salvamento de estados, individualmente em cada processo de um sistema distribuído, a exemplo do que ocorre nos sistemas monoprocessados, é o primeiro

enfoque que pode ocorrer ao projetista de tolerância a falhas. Esta forma de salvar os estados é chamada de **estabelecimento independente dos pontos de recuperação**: cada processo pode decidir quando é mais conveniente efetuar esta tarefa. Este estado, salvo durante o estabelecimento de um ponto de recuperação, é chamado mais genericamente de um ***snapshot* local do processo**. O termo *snapshot* tem uso mais abrangente do que ponto de recuperação. Conforme Manivannan [MAN 97], o primeiro é utilizado na área de depuração (*debug*) e simulação distribuída, monitorização, estabelecimento de pontos de parada (*breakpoints*), etc; enquanto que o segundo é normalmente utilizado no contexto de tolerância a falhas.

Entretanto, o procedimento independente pode apresentar o problema de consistência discutido por Chandy e Lamport [CHA 85].

Caso seja necessário o retorno, após a ocorrência de uma falha, cada processo poderá escolher o seu ponto de retorno. Com o objetivo de reduzir a quantidade de processamento a ser repetido, provavelmente este ponto será o último estabelecido.

Levando-se em consideração que os processos de um sistema distribuído trocam informações através de mensagens e que o instante exato do estabelecimento dos pontos de recuperação só é conhecido localmente no processo, podem ocorrer duas situações de inconsistência:

- mensagens registradas como transmitidas por algum processo remetente podem não estar registradas como recebidas pelos processos destino. São chamadas de **mensagens perdidas**;

- ou mensagens registradas como recebidas por algum processo podem não estar registradas como transmitidas em qualquer dos processos do sistema. São chamadas de **mensagens órfãs**.

Estas inconsistências foram discutidas por Chandy [CHA 85], onde chega-se a conclusão que, em geral, para garantir a consistência, não deve ocorrer nenhum dos dois cenários descritos.

Desta forma, para uma retomada consistente do processamento no caso da recuperação, deve-se prover um tratamento que não permita a ocorrência de mensagens perdidas nem órfãs.

1.3.1 O conceito de consistência

Na realidade, a definição de consistência merece um pouco mais de detalhamento. Conforme discutido por Chandy, o conceito de consistência está associado com a impossibilidade de que um processo receba mais mensagens do que lhe foram enviadas. Este princípio leva à conclusão que um *snapshot* global será consistente se não houver mensagens não transmitidas registradas como recebidas. Desta forma, o conceito de consistência ficou associado ao tratamento das mensagens órfãs.

Apesar do exposto, não se pode usar de forma indiscriminada o conceito de consistência vinculado apenas às mensagens órfãs. Deve-se observar outros fatores.

Uma primeira observação diz respeito à aplicação do conceito de consistência. Chandy e Lamport estavam interessados em detecção de *deadlocks*. Nesta aplicação, o objetivo é determinar um estado global consistente, sem a intervenção de outros eventos que não os de comunicação. Naquela proposta de algoritmo, a coleta dos *snapshots* locais é feita de forma distribuída e evitando que ocorram mensagens

órfãs. Entretanto, após esta etapa, todos os processos devem enviar seus *snapshots* para um coordenador, que será responsável por calcular o estado dos canais, ou seja, determinar quais são as mensagens registradas como transmitidas mas não como recebidas (ou mensagens perdidas).

Em conseqüência, muitos trabalhos que utilizam o conceito de consistência apresentado por Chandy e Lamport, preocupam-se apenas com as mensagens órfãs. Um exemplo disso é o trabalho de Netzer e Xu [NET 95], onde os autores seguem o mesmo conceito de consistência de Chandy e Lamport. Muito pouco é mencionado sobre as mensagens perdidas. Entretanto, é referido um trabalho anterior [XU 93], onde os conceitos teóricos foram usados na implementação de um algoritmo de recuperação, e no qual é suposta a existência de um mecanismo para repetição de mensagens perdidas.

As implementações que utilizam a formalização de Chandy e Lamport, e depois estendidas por Netzer e Xu, requerem que exista um mecanismo de armazenamento das mensagens perdidas, para uma posterior repetição, em caso de falha. Esta necessidade está vinculada, principalmente, à premissa de que os canais de comunicação são confiáveis ou que, pelo menos, garantem que as mensagens não serão perdidas.

Do ponto de vista da formalização de consistência, pode-se continuar a usar os trabalhos já desenvolvidos, não esquecendo, entretanto, as premissas por eles usadas.

1.3.2 Tratamento das inconsistências

Os mecanismos que tratam as inconsistências ocasionadas por mensagens variam de algoritmo para algoritmo. Normalmente, o tratamento para mensagens perdidas é diferente daquele aplicado às mensagens órfãs. Isso é natural, uma vez que as duas categorias de mensagens apresentam características bem distintas. Entretanto, mesmo quando se trata de mensagens da mesma categoria, não há uniformidade de tratamento pelos diferentes algoritmos. Por exemplo, alguns [KOO 87, WAN 95] impedem a ocorrência das mensagens perdidas e/ou órfãs, enquanto que outros [STR 85, JUA 91, CHI 96] apresentam mecanismos que tratam a ocorrência das mesmas.

Da análise dos critérios de consistência usados pelos trabalhos na área de estabelecimento de pontos de recuperação globais e consistentes e da área de recuperação extraiu-se os conceitos que serão usados neste trabalho.

Em primeiro lugar, está a consistência no sentido apresentado por Chandy e Lamport: tratamento das mensagens órfãs.

Em segundo lugar, está o tratamento das mensagens perdidas. Nota-se que existem dois tipos de mensagens perdidas:

- mensagens perdidas no canal (*in-transit lost messages*);
- mensagens perdidas devido à recuperação.

As mensagens perdidas no canal são aquelas que não chegam ao seu destino devido a problemas no sistema de comunicação. Exemplos destas são mensagens que foram descartadas por erro ou que não puderam ser acomodadas em algum *buffer* existente no caminho entre transmissor e receptor.

Quanto às mensagens perdidas devido à recuperação, ocorrem quando uma mensagem foi registrada no ponto de recuperação do transmissor como transmitida

mas não foi registrada no ponto de recuperação do receptor como recebida, provavelmente por ter sido recebida após o estabelecimento deste ponto de recuperação.

Estes dois tipos de mensagens perdidas podem requerer tratamentos diferenciados, uma vez que possuem causas diferentes. Basicamente, as mensagens perdidas serão tratadas segundo o tipo de canal de comunicação disponível. Se o canal for do tipo que admite perda de mensagens, não há a necessidade de que a recuperação preocupe-se com elas; se o canal for do tipo que não perde mensagens por hipótese, o algoritmo de recuperação não pode causar mensagens perdidas [ELN 96].

1.3.3 Classificação geral dos algoritmos

O estabelecimento dos pontos de recuperação pode ser classificado de várias formas, dependendo dos tipos de informações que serão armazenadas. Desta forma, um conjunto de pontos de recuperação locais recebe uma dentre as seguintes designações [HÉL 99]:

- ponto de recuperação global;
- ponto de recuperação global consistente;
- ponto de recuperação global fortemente consistente.

Um **ponto de recuperação global** é formado por um conjunto de pontos de recuperação locais, um de cada processo componente do sistema.

Um **ponto de recuperação global consistente**, além de global, garante que não existam mensagens órfãs.

Finalmente, um **ponto de recuperação global fortemente consistente**, além de consistente, garante que não existam mensagens perdidas.

Um ponto de recuperação global e consistente (forte ou não, dependendo do contexto) é denominado de **linha de recuperação**.

Tomando como referência os tipos de pontos de recuperação consistentes, os algoritmos existentes na literatura podem ser classificados, no que diz respeito ao tratamento das mensagens órfãs e perdidas, nas seguintes classes:

- algoritmos fortemente consistentes;
- algoritmos consistentes.

Os algoritmos fortemente consistentes (como o proposto por Neves [NEV 98]) são aqueles que possibilitam o retorno para um ponto de recuperação, oferecendo um tratamento para as mensagens órfãs e as perdidas. Desta forma, estes algoritmos oferecem uma visão consistente para a aplicação e não necessitam suporte da comunicação.

No caso dos algoritmos consistentes, as mensagens perdidas não são tratadas. O algoritmo proposto por Koo e Toueg [KOO 87] passa, explicitamente, a responsabilidade do tratamento destas mensagens para o protocolo de comunicação. Outros [NET 95, MAN 97, JUA 91, VEN 97], definem o critério de consistência dos pontos de recuperação apenas pela garantia da inexistência de mensagens órfãs e seus mecanismos são modelados desta forma: as mensagens perdidas não são tratadas. Dentro desta categoria, tomando as premissas dos algoritmos como critério para classificação, podem ser incluídos ainda outros [PRA 96, CAO 98, ELN 92] que, apesar de não considerarem as mensagens perdidas quando justificados teoricamente, oferecem um mecanismo para seu tratamento, na descrição de sua implementação.

1.4 Requisitos de sistema

Muitas propostas de algoritmos de recuperação idealizam certas características dos sistemas distribuídos. Outros, usam como premissa a existência de protocolos auxiliares. Nestes trabalhos, as premissas são usadas como forma de particularizar o sistema alvo ou como forma de simplificar os problemas. A proposta deste novo algoritmo visa ser o menos restritivo possível, usando premissas mais próximas da realidade e evitando repassar os problemas para algoritmos auxiliares.

Um dos requisitos usuais diz respeito ao tratamento das **mensagens perdidas**: aquelas registradas como transmitidas mas não registradas como recebidas. Vários trabalhos [CHA 85, WAN 97, NET 95] consideram que um conjunto de pontos de recuperação, um por processo, será consistente se não houver uma relação de precedência entre qualquer par de pontos locais (*happened before*, conforme Lamport [LAM 78]). Ou seja, não existe um caminho causal entre dois pontos de recuperação locais: uma mensagem enviada após um ponto de recuperação e recebida antes de outro. Este critério de consistência considera que as únicas mensagens que tornam o sistema inconsistente são as **órfãs**: aquelas registradas como recebidas porém não registradas como transmitidas.

Por outro lado, Chandy e Lamport [CHA 85] mostraram que um estado global consistente será formado pelos estados dos processos e os estados dos canais. Coerentemente com isso, propuseram um algoritmo em que as mensagens órfãs não ocorrem (o algoritmo impede esta ocorrência) enquanto que as perdidas estarão armazenadas junto com o estado dos processos, nos pontos de recuperação (ou *snapshots*, na nomenclatura de Chandy e Lamport). Quando todos os processos tiverem salvo seus estados e o estado dos canais incidentes (aqueles através dos quais o processo pode receber mensagens), esta informação será enviada para um processo coletor que fará a sincronização dos estados dos canais, obtendo o estado global do sistema.

Percebe-se que o algoritmo proposto por Chandy e Lamport fornece um tratamento para as mensagens perdidas. Assim, pode-se perceber que os algoritmos, onde são tratadas apenas as mensagens órfãs, necessitam de um algoritmo de suporte para tratar as mensagens perdidas.

Koo e Toueg *et al.* [KOO 87] e Wang *et al.*[WAN 95a] propuseram o uso de protocolos de comunicação com suporte ao tratamento de mensagens perdidas. Assim, nos casos referidos, **o algoritmo de comunicação será responsável pela recuperação das mensagens que venham a se tornar perdidas, devido à recuperação**. Entretanto, não é válida a premissa de que as mensagens perdidas no canal durante a operação sem falhas apresentam o mesmo comportamento daquela perdidas devido à recuperação, haja visto o tratamento diferenciado que é dado às mensagens perdidas, quando são utilizados canais confiáveis, conforme apresentado por Elnozahy [ELN 96]. Desta forma, a premissa da existência de um algoritmo de comunicação que trate as mensagens perdidas não é suficiente. Deve-se conhecer a forma como são efetivamente realizadas as garantias à comunicação de maneira que os dois algoritmos, recuperação e comunicação, operem de forma harmônica.

Para evitar todos os problemas da adaptação da recuperação com a comunicação sem perdas de mensagens, o algoritmo proposto neste trabalho **não requer a existência de canais com garantia de entrega**. Entretanto, além de informar defeitos dos processos, o detector de defeitos deve ser capaz de **detectar e informar falhas do tipo mensagens perdidas** nos canais.

Outra premissa comum diz respeito à ordenação das mensagens. Muitos algoritmos [CHA 85, NET 95, BRZ 95] utilizam como premissa que os canais sejam do tipo FIFO. No caso dos canais de comunicação apresentarem este comportamento, a ordem em que as mensagens foram transmitidas seria a mesma percebida pelo processo destino, quando da recepção das mesmas.

Esta premissa só pode ser garantida se existir um algoritmo que realize a ordenação das mensagens. Este, deverá ser colocado entre os módulos da recuperação e da comunicação.

A ordenação das mensagens, apesar de ser um problema resolvido, implica em custos adicionais. Um dos custos está relacionado com o desempenho: a inclusão de mais um nível de processamento pode reduzir o desempenho do sistema. Outro custo diz respeito à forma como a ordenação é feita e sua interação com o protocolo de recuperação: é necessário uma interface muito bem documentada, visando sua integração ao restante do sistema, para garantir que, em caso de falha e recuperação, mensagens não sejam indevidamente descartadas ou repetidas.

Desta forma, o modelo de sistema do presente trabalho **não requer que as mensagens sejam ordenadas na recepção**.

Alguns algoritmos de recuperação, como aqueles que se utilizam de *log* de mensagens [ELN 92, JOH 87], necessitam que os processos tenham um comportamento do tipo PWD: efetuam processamento determinístico intercalado com eventos não-determinísticos (eventos de comunicação). Com isso, o retorno pode ser feito pela repetição exata dos eventos desde o último ponto de recuperação até o momento da falha: eventos de processamento, considerados determinísticos, e eventos de comunicação, considerados não-determinísticos. Em geral, são algoritmos que consideram que a única fonte de indeterminismo são os eventos gerados pela recepção das mensagens. Desta forma, armazenam estes eventos em memória estável, na forma de ***logs* de mensagens**.

Outros algoritmos [STR 85], apesar de registrarem as mensagens, fazem-no em memória volátil, sendo esta copiada para memória estável, quando do estabelecimento do próximo ponto de recuperação. Isso reduz o custo do salvamento destas informações.

O processamento PWD associado com o salvamento das mensagens para posterior repetição, pode levar a uma repetição da ocorrência da falha: causas idênticas (conteúdo da memória, no instante do estabelecimento do ponto de recuperação) associadas a processamentos idênticos (uma vez que a repetição das mensagens tornará totalmente determinístico) pode levar a conseqüências idênticas (repetição do erro). Desta forma, a falha poderá ser caracterizada como permanente, o que impedirá o uso da técnica de recuperação por retorno, útil quando as falhas são temporárias.

Para aumentar a cobertura de falhas, Wang [WAN 95] observou que algumas falhas permanentes poderiam apresentar comportamento temporário, quando alguns parâmetros do sistema distribuído fossem alterados. Parâmetros tais como a ordem de recepção das mensagens, o tempo de propagação ou o processador responsável por determinada computação, podem ser utilizados como exemplos. Esta alteração dinâmica nos parâmetros do sistema distribuído oferece um tipo de diversidade, que Wang denominou de **diversidade de ambiente**. Esta diversidade poderia levar o sistema a não falhar, na repetição do processamento, mesmo com falhas permanentes.

Desta forma, não é premissa do algoritmo proposto que o comportamento dos

processos tenha que ser PWD. Assim, espera-se permitir uma maior liberdade de processamento, visando alcançar a diversidade de ambiente proposta por Wang.

1.5 *Checkpointing* e recuperação

Os algoritmos utilizados para a recuperação de um sistema, em caso de falha temporária, baseiam-se na presença, em memória estável, de um conjunto de informações. Estas formam os pontos de recuperação que poderão ser recarregados na memória dos processos, permitindo a retomada do processamento. Isso significa que, de forma periódica, informações do processamento normal (estado do processo, registradores do processador, *buffers* de mensagens não respondidas, etc.) devem ser salvas em memória estável, visando reduzir a quantidade de processamento perdido, na eventual necessidade de uma recuperação.

No caso de falhas, as informações salvas durante o processamento normal serão utilizadas pela recuperação. Assim, existe uma forte correlação entre a forma como são salvas e a forma como serão utilizadas. A escolha descompromissada destas duas formas levará, certamente, a problemas de desempenho. Fatores como a estrutura de dados e a periodicidade de estabelecimento dos pontos de recuperação deverão ser ajustados aos parâmetros relacionados com a recuperação tais como, por exemplo, a forma como os dados são usados na recuperação e a taxa de falhas suposta para o sistema. Neste sentido, pode-se criticar os sistemas que propõem mecanismos de estabelecimento de pontos de recuperação e o indicam como uso potencial em tolerância a falhas, sem explicitar o mecanismo de recuperação que deve ser utilizado.

Além da eficiência, Prakash *et al.* [PRA 96] apresentam como característica importante dos mecanismos de recuperação a baixa interferência na aplicação: os mecanismos de recuperação não devem causar o bloqueio da aplicação, enquanto estão executando as suas tarefas.

A forma como são estabelecidos os pontos de recuperação é usada para classificar os algoritmos de recuperação. Este fator é importante na medida que influencia significativamente tanto o grau de interferência do mesmo na aplicação como a forma pela qual ocorre a recuperação.

Desta forma, a partir do estudo dos mecanismos utilizados para implementar o estabelecimento dos pontos de recuperação e o retorno, identificaram-se os mecanismos básicos empregados, os quais foram agrupados da seguinte forma:

- estabelecimento autônomo e opcional (pontos independentes);
- estabelecimento obrigatório por determinação externa (pontos forçados);
- armazenamento de mensagens (*logs*);
- bloqueio da aplicação.

Na realidade, o grupo dos pontos independentes, forçados e com *log* de mensagens corresponde à forma como são estabelecidos os pontos de recuperação locais, enquanto que o grupo dos bloqueantes indica, apenas, como a aplicação será afetada pelo estabelecimento dos pontos de recuperação.

Estes grupos correspondem a mecanismos que, em geral, não são encontrados isoladamente nos algoritmos, seja porque não caracterizam suficientemente o algoritmo (caso dos bloqueantes), seja porque apresentam limitações conhecidas (caso do independente).

No caso dos pontos de recuperação estabelecidos de forma **independente**, cada processo estabelece seus pontos de recuperação, sem fornecer informações de sua atividade para o restante do sistema. Este mecanismo está sujeito à ocorrência do **efeito dominó** (retorno ao início do processamento, em caso de falha).

Quando um algoritmo utiliza um mecanismo **forçado**, os processos estabelecem *snapshots* locais, seguindo um conjunto de critérios que visam garantir a existência de uma linha de recuperação. A utilização de critérios para forçar o estabelecimento dos pontos de recuperação pode ser visto, também, como uma forma de coordenação. Na realidade, é bastante difícil projetar um algoritmo que não faça uso de pontos de recuperação forçados e que garanta a ausência do efeito dominó.

No mecanismo de ***log* de mensagens**, estas são armazenadas de maneira a serem repetidas, no caso de falha. Assim, quando um erro for detectado, o processo retorna ao último ponto de recuperação e repete a execução até o momento da detecção do erro, simulando a recepção das mensagens armazenadas. Embora, somente o processo em falha necessite retornar, todos os outros devem ficar esperando o término da recuperação daquele que falhou.

A determinação dos instantes mais adequados para o estabelecimento dos pontos de recuperação, de maneira a garantir a consistência, é feita através da observação dos instantes de transmissão e recepção das mensagens. Desta forma, o projetista deve decidir como tratar aquelas mensagens trocadas pela aplicação, durante o estabelecimento destes pontos. Uma solução simples é suspender o processamento da aplicação, até que os novos pontos de recuperação tenham sido estabelecidos: são os algoritmos chamados de **bloqueantes**. Entretanto, o custo associado é muito elevado. Previamente ao presente trabalho, realizou-se um estudo comparativo entre um algoritmo bloqueante e outro não bloqueante, cujos resultados apresentados em [CEC 98a] confirmam numericamente esta noção intuitiva, através de cenários bastante variados. Outra solução é acrescentar mecanismos que possam identificar e ajustar as mensagens de maneira que seja garantida a consistência.

De forma semelhante aos mecanismos de estabelecimento dos pontos de recuperação, os usados no retorno, por ocasião da recuperação, também podem ser subdivididos. Estes são classificados de acordo com a forma como encontram uma linha de recuperação:

- identificação imediata;
- identificação mediante processamento adicional.

Conforme já explicado, o mecanismo de recuperação está fortemente relacionado com o mecanismo escolhido para o estabelecimento dos pontos de recuperação. A escolha de um mecanismo **imediato** para o retorno, implica na existência de uma linha de recuperação disponível e bem identificada. Desta forma, o mecanismo de estabelecimento dos pontos de recuperação deverá garantir a existência desta linha.

O contraponto aos algoritmos de retorno imediato são os algoritmos que necessitam de algum **processamento** para encontrarem uma linha de recuperação

ou sincronizarem as mensagens, de forma a garantir uma retomada consistente do processamento.

Entretanto, os autores de algoritmos de recuperação procuram enquadrar suas propostas utilizando uma nomenclatura diferente, que traz algumas informações sobre propriedades funcionais dos algoritmos. Têm sido usados os seguintes qualificativos:

- Síncronos;
- Assíncronos;
- Coordenados;
- Controlados pela comunicação (*communication-induced*).

Na realidade, os dois primeiros (síncronos e assíncronos) correspondem a uma classificação mais antiga, enquanto que os dois últimos têm sido utilizados mais recentemente, principalmente nos trabalhos que versam sobre os algoritmos controlados pela comunicação.

Uma descrição bastante detalhada dos tipos de algoritmos bem como uma extensa lista de referências bibliográficas foi publicada por Elnozahy *et al* [ELN 96].

Os termos síncrono e coordenado têm sido utilizados como sinônimos, no que diz respeito à caracterização da forma de estabelecimento dos pontos de recuperação. Entretanto, Baldoni [BAL 97] diferenciou-os considerando os mecanismos síncronos como formas de coordenação. Este enfoque se justifica na medida que alguns algoritmos utilizam a coordenação sem ter que bloquear o avanço da aplicação. Assim, de uma forma geral, o termo **síncrono** de estabelecimento dos pontos de recuperação ficou reservado para os mecanismos que bloqueiam a aplicação (bloqueantes), enquanto que a expressão **coordenado** é usada sempre que os processos trocam mensagens visando determinar uma linha de recuperação, sem a necessidade de bloquear a aplicação. Uma exceção é o trabalho de Do-Hyung e Chang-Soon [DOH 2000], onde o termo coordenado foi associado a bloqueio da aplicação.

Os **algoritmos síncronos** utilizam um mínimo de memória, pois necessitam manter apenas uma linha de recuperação. Entretanto, para o seu estabelecimento, necessitam sincronizar todo o sistema, o que tem um impacto bastante significativo no desempenho do sistema. Quando ocorrem falhas, o sistema identifica, de forma imediata, a linha de recuperação para o retorno.

Os **algoritmos coordenados** utilizam a mesma quantidade de memória que os síncronos. O desempenho do sistema, entretanto, não é tão afetado por eles, uma vez que não necessitam de sincronização. Necessitam apenas da troca de informações entre os processos para estabelecerem as linhas de recuperação. Da mesma forma que o síncrono, o sistema identifica, de forma imediata, a linha de recuperação para o retorno.

Tradicionalmente, os **algoritmos assíncronos** são tomados como contraponto dos síncronos, quando a característica analisada é o bloqueio da aplicação: os algoritmos assíncronos não são bloqueantes. Entretanto, os algoritmos classificados como assíncronos normalmente não buscam garantir a existência de uma linha de recuperação, durante o estabelecimento dos pontos de recuperação. São projetados para encontrá-la na recuperação. Este é o critério mais preciso para classificar os algoritmos: os coordenados trocam mensagens visando estabelecer uma linha

de recuperação enquanto que os assíncronos trocam mensagens visando limitar um possível efeito dominó, no caso de falha.

O termo **independentes** tem sido usado para identificar sistemas onde os pontos de recuperação são estabelecidos de forma totalmente individual: nenhum outro processo toma conhecimento do fato, nem o processo que decidiu por salvar um ponto de recuperação utiliza informações dos outros processos ou da computação em si (como, por exemplo, a dependência de computação). Esta forma de operar está sujeita, no caso de falha, ao efeito dominó.

Na realidade, um protocolo em que os pontos de recuperação são do tipo independentes, conforme descrito anteriormente, é de pouco uso prático, exceto talvez, em algum sistema onde o custo de comunicação seja tão elevado que qualquer processamento adicional não seria aceitável, mesmo com o risco da ocorrência do efeito dominó.

O caso **assíncrono** usa bastante memória, como o mecanismo independente. Entretanto, graças às trocas de informações, que degradam o desempenho, os processos podem proceder à coleta de lixo, reduzindo a quantidade de memória necessária. Além disso, as informações trocadas auxiliam na formação de uma linha de recuperação a partir dos pontos de recuperação locais já estabelecidos. Quando o procedimento de recuperação é necessário, o sistema deverá procurar por uma linha de recuperação que, de forma diversa do mecanismo independente, será sempre encontrada, limitando o efeito dominó.

Forçar um ponto de recuperação significa induzir um processo a suspender sua atividade fim para estabelecer um ponto de recuperação, não importando o custo envolvido. Com isso, pode-se garantir, por exemplo, a existência de uma linha de recuperação.

No caso dos **controlados pela comunicação**, são usados dois mecanismos: o independente e o forçado. Esta mistura busca extrair o melhor dos dois mecanismos. A expectativa é que o bom desempenho do mecanismo independente, durante o estabelecimento dos pontos de recuperação, associado ao bom desempenho do mecanismo forçado, durante a recuperação, leve a um resultado melhor do que cada um dos mecanismos, quando usados de forma isolada. São exemplos os trabalhos de Wang [WAN 97], de Alvisi [ALV 99] e de Baldoni [BAL 99, BAL 97].

A discussão anterior está resumida nas tabelas 1.1 e 1.2 onde estão assinalados os mecanismos usados, em geral, para caracterizar as categorias de algoritmos.

TABELA 1.1 – Tipos de *checkpointing*

Mecanismos	Categorias			
	Coordenados		Controlados pela Comunicação	Assíncronos
	Síncronos	Não Síncronos		
Independentes			X	X
forçados	X	X	X	
log de mensagens				X
bloqueantes	X			

Alguns algoritmos empregam apenas um mecanismo para estabelecer os pontos de recuperação. Este é o caso dos algoritmos coordenados do tipo não síncronos,

TABELA 1.2 – Tipos de recuperação

Mecanismos	Categorias			
	Coordenados		Controlados pela Comunicação	Assíncronos
	Síncronos	Não Síncronos		
imediato	X	X		
com processamento			X	X

como os propostos por Briatico *et al.*[BRI 84] e Chandy e Lamport [CHA 85]. Entretanto, além dos síncronos e dos controlados pela comunicação, pode-se encontrar outros algoritmos que utilizam mais de um mecanismo. Este é o caso dos trabalhos de Prakash [PRA 96] e Cao [CAO 98], onde é usado um algoritmo coordenado associado com pontos de recuperação estabelecidos de forma independente.

Conforme citado por Alvisi [ALV 99], provavelmente Briatico [BRI 84] teria sido o primeiro a propor um tipo de algoritmo onde pontos de recuperação independentes foram associados a critérios, controlados pelo padrão de comunicação, que levariam ao estabelecimento de pontos de recuperação forçados. Este tipo de protocolo foi designado de **estabelecimento de pontos de recuperação controlados pela comunicação (*Communication-Induced Checkpointing*)**.

Wang [WAN 97] apresentou um protocolo deste tipo, onde é utilizada a dependência entre os processos, devida a sua comunicação. Os algoritmos propostos são baseados nesta dependência. Além disso, foi desenvolvida a propriedade chamada de RDT – *Rollback Dependency Trackability* – a qual fornece um arcabouço teórico através do qual pode-se projetar algoritmos descentralizados eficientes para a construção da árvore de dependências entre processos.

Ambos, Alvisi e Wang, utilizam o mesmo mecanismo básico: são definidos critérios que forçam o estabelecimento de pontos de recuperação locais, adicionais aos independentes. A definição dos critérios que levam ao estabelecimento dos pontos forçados deve ser tal que funcionem como uma espécie de "cola" que liga os pontos independentes, formando linhas de recuperação.

Para que os processos possam minimizar o custo associado ao estabelecimento dos pontos de recuperação escolhendo os momentos mais adequados para a tarefa, normalmente devem ser acrescidas informações às mensagens da aplicação. Com este mecanismo, não serão necessárias mensagens adicionais para garantir a consistência.

Entretanto, o acréscimo de informações às mensagens apresenta os seguintes custos adicionais:

- quando for recebida uma mensagem e antes de entregá-la para a aplicação, os processos podem ser forçados a estabelecer um novo ponto de recuperação;
- as mensagens da aplicação serão aumentadas com informações adicionais;
- é necessário manter um conjunto relativamente grande de pontos de recuperação na memória estável.

Como o custo imposto por este mecanismo depende do padrão de processamento, a sua análise só poderá ser feita de forma estatística. Assim, em uma

tentativa de generalização, pode-se estimar que o custo deste mecanismo é menor do que o do mecanismo síncrono, sendo maior que o de estabelecimento de pontos independentes.

Entretanto, conforme medições efetuadas por Alvisi [ALV 99] e por Baldoni [BAL 97], o custo médio imposto pelos pontos forçados pode ser muito alto, fazendo com que "...existam evidências suficientes para suspeitar que muito do que se conhece sobre estes protocolos é questionável..."[ALV99].

Aparentemente, com os atuais critérios de estabelecimento dos pontos forçados, o desempenho apresentado não é muito satisfatório. Em uma primeira análise, poder-se-ia supor que a escolha de um critério mais eficiente levaria a um melhor desempenho. Entretanto, conforme exposto por Alvisi, um único critério estático não é suficiente. É necessária a aplicação dinâmica de vários critérios, adaptados às ocorrências dos pontos forçados e às variações do padrão de comunicação.

2 Mecanismos e premissas

Conforme visto no capítulo anterior, existem propostas de algoritmos para recuperação [KOO 87] em que, durante o estabelecimento dos pontos de recuperação, a aplicação deve suspender, totalmente, suas atividades. São chamados de algoritmos **bloqueantes**. Esta característica simplifica em muito o mecanismo de estabelecimento dos pontos de recuperação. Entretanto, impõe um alto custo de processamento, o que reduz o desempenho global do sistema. Na proposta deste novo algoritmo, permite-se que a aplicação continue a operar em paralelo com a operação de estabelecimento dos pontos de recuperação e, em certa medida, com o retorno.

Nem sempre os projetistas de algoritmos para recuperação preocupam-se com a chamada **coleta de lixo**, apesar de "... ser uma questão prática importante." [ELN 96]. Esta é uma das deficiências, por exemplo, do trabalho de Briático [BRI 84], onde um número infinito (teórico) de pontos de recuperação pode ser necessário.

Basicamente, a coleta de lixo visa remover aquelas informações que deixaram de ser úteis. Este é o caso de um ponto de recuperação, quando existe um outro mais recente que, juntamente com os demais, componha uma linha de recuperação consistente. No algoritmo proposto neste trabalho, a coleta de lixo é intrínseca ao algoritmo, ou seja, é usado um mínimo de pontos de recuperação. No momento em que uma nova linha de recuperação é completada, os pontos de recuperação que formavam uma linha de recuperação antiga, podem ser descartados.

Para o projeto do algoritmo de recuperação de processos proposto nesta tese foram considerados vários aspectos, cuja discussão e conclusões alcançadas encontram-se nas seções que seguem. Os aspectos discutidos são os seguintes:

- coordenação entre processos, para o estabelecimento dos pontos de recuperação;
- tipos de aplicação, no que diz respeito a perda de mensagens e ao uso dos canais de comunicação;
- detecção de defeitos, e seu relacionamento com o algoritmo de recuperação;
- *livelock*, que podem impedir o progresso do processamento;
- forma de tratamento de mensagens perdidas, principalmente o salvamento para possível repetição;
- anexação de dados às mensagens da aplicação e seu impacto ao desempenho;
- dependências de processamento e suas formas (direta e transitiva).

2.1 Protocolo coordenado

Um dos motivos que levou à escolha de um mecanismo coordenado para o estabelecimento dos pontos de recuperação foi a expectativa de resultados quanto ao desempenho. Há bons indícios de que este enfoque possibilite a otimização do desempenho em sistemas onde ocorram falhas.

Uma das argumentações contra o uso de mecanismos coordenados diz respeito ao custo das mensagens de coordenação. Entretanto, conforme Cao [CAO 98], esta argumentação só é válida para a última década, onde os custos da comunicação excederam em muito aquele do acesso à memória estável, favorecendo a algoritmos não coordenados.

Ainda, na argumentação de Cao, os sistemas modernos tiveram o custo de comunicação sensivelmente reduzido. Neste caso, os custos associados à coordenação passam a ser negligenciáveis, quando comparados ao do salvamento em memória estável.

Plank [PLA 98] também apresentou um algoritmo que não usa **memória estável local** para armazenar os pontos de recuperação. Estes são enviados para outros processadores e armazenados em memória comum: é utilizada a redundância natural dos sistemas distribuídos. Novamente, este sistema faz uso da alta velocidade dos canais de comunicação.

Hsu [HSU 99] implementou dois mecanismos de salvamento de pontos de recuperação: o salvamento remoto e o salvamento em memória estável local (sistema de discos). Com estas duas implementações e usando várias aplicações, efetuou medidas de desempenho. Os resultados foram bastante favoráveis ao uso de salvamento remoto. Estes resultados, conforme o próprio Hsu, são conseqüência do aumento da velocidade dos canais de comunicação.

Além das argumentações anteriores, pode-se citar ainda as medidas efetuadas por Elnozahy [ELN 94] e as equações de desempenho desenvolvidas em trabalho anterior [CEC 98b] para dois algoritmos específicos, um síncrono e outro assíncrono, onde, sob determinadas condições, o desempenho síncrono pode tornar-se melhor que o assíncrono.

Mesmo que os sistemas usados nas argumentações anteriores utilizem canais de comunicação de alta velocidade, não está claro o limite a partir do qual um mecanismo passa a ser melhor do que o outro. Além disso, conforme salientado por Cao [CAO 98], nos algoritmos não coordenados existem custos adicionais que nem sempre são considerados:

- complexidade em encontrar uma linha de recuperação, após a falha;
- susceptibilidade ao efeito dominó;
- necessidade de salvamento de múltiplos pontos de recuperação;
- processamento para coleta de lixo.

Muitas das argumentações anteriores baseiam-se na alta velocidade dos canais de comunicação, característica que é cada vez mais presente nas redes locais. Entretanto, a associação da coordenação com os outros mecanismos que formarão o algoritmo proposto, permitirá um bom desempenho, mesmo em redes comuns.

Finalmente, visando otimizar o desempenho, os pontos coordenados deverão ser estabelecidos com uma freqüência que minimize o custo médio associado às atividades de tolerância a falhas. Na realidade, esta necessidade de ajuste na periodicidade dos pontos de recuperação não deve ser considerada como mais uma dificuldade. Deve ser considerada como uma vantagem: como um parâmetro que permitirá adaptar o mecanismo de recuperação às características do sistema. Um exemplo disso é o ajuste deste parâmetro à taxa de falhas do sistema.

Outro ponto de crítica aos algoritmos coordenados está relacionado com a necessidade de um processo assumir a função de coordenador, o que pode implicar em um ponto único de falhas.

Para contornar este ponto fraco do algoritmo, propõe-se que seja implementado um mecanismo de rotação de coordenadores. Para isso, deve ser implementado um mecanismo específico.

Uma sugestão de mecanismo é a seguinte: os processos deverão ser numerados segundo a sua ordem de criação. Esta, por sua vez, determinará a ordem de rotação dos coordenadores, sendo que, após o último, retorna-se ao primeiro, formando uma lista circular.

Desta forma, a cada novo ponto de recuperação, apenas um processo será responsável pela tarefa de coordenação. Este, por sua vez, será inequivocamente identificado por todos os outros componentes.

2.2 Tipos de aplicações

Conforme apresentado na introdução, a consistência é alcançada através do tratamento das mensagens órfãs, quando o canal de comunicação é não confiável, ou através do tratamento das mensagens órfãs e perdidas, quando o canal de comunicação garante a entrega das mensagens.

Adicionalmente, alguns trabalhos propõem que o protocolo de comunicação trate as mensagens perdidas devido à comunicação e devido à recuperação.

Desta forma, percebe-se que o tipo de consistência pode ser determinado pelos canais de comunicação.

Então, para analisar o comportamento dos sistemas quando acrescidos de um protocolo de recuperação, decidiu-se por separar as **aplicações** e os **protocolos de comunicação** em classes. Estas, por sua vez, são combinadas para estabelecer as configurações de sistema.

A comunicação foi separada em duas classes, segundo a forma como trata as mensagens perdidas: se garante que não ocorram mensagens perdidas devido a ações embutidas em atividades de comunicação ou se não garante. Foram classificadas assim:

- Comunicação não confiável (nC);
- Comunicação confiável[1] (C).

As aplicações foram classificadas segundo dois critérios que podem ser combinados entre si. Os dois critérios são:

- quanto a perda de mensagens: aquelas que admitem a perda (PP) e aquelas que não admitem a perda de mensagens (nPP);
- quanto aos procedimentos relativos à perda de mensagens: aquelas que tratam as mensagens perdidas (TP) e aquelas que não tratam as mensagens perdidas (nTP).

[1] O termo *Comunicação confiável* está sendo utilizado como sinônimo de garantia de entrega de mensagens.

Estes dois critérios, quando combinados, fornecem quatro tipos diferentes de aplicações:

- PP e TP;
- PP e nTP;
- nPP e TP;
- nPP e nTP.

A classificação apresentada está resumida na figura 2.1.

Alguma dúvida poderia surgir sobre qual tipo de aplicação poderia perder mensagens. Pode-se citar as chamadas aplicações probabilísticas, que confiam na probabilidade de que suas mensagens chegarão aos destinos. Normalmente utilizam redundância de envio de mensagens, ou seja, enviam mais de uma cópia da mesma mensagem.

Outro tipo de aplicação é aquela em que não se está preocupado com tolerância a falhas, ou seja, pode perder mensagens, deixando para o usuário a responsabilidade de resolver estes casos. Um exemplo é o acesso a páginas na Internet.

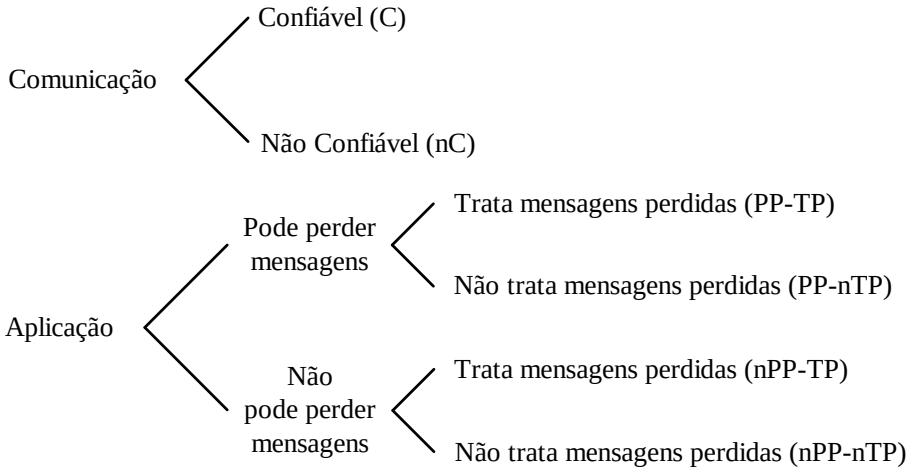

FIGURA 2.1 – Tipos de Comunicação e Aplicação

Cada tipo de aplicação pode ser combinado com um dos dois tipos de comunicação, formando um sistema. Tem-se, portanto, oito tipos diferentes de sistemas.

A categoria de aplicações PP-TP não faz sentido como objetivo de projeto, uma vez que, se a aplicação pode perder mensagens, não é razoável gastarem-se recursos para tratá-las[2]. Então, na realidade, existem seis classes de sistemas.

No sistema nPP-TP-C, também são desperdiçados recursos, pois as mensagens perdidas serão tratadas na aplicação e na comunicação. A redundância existente só será útil se alguma das classes não estiver cumprindo adequadamente seus requisitos.

No caso do PP-nTP-C, novamente são desperdiçados recursos, uma vez que as aplicações não necessitam da garantia de entrega das mensagens.

[2]Na prática, tais sistemas podem existir, no reaproveitamento de recursos de implementações existentes.

Finalmente, o sistema nPP-nTP-nC não pode ser usado uma vez que as aplicações não podem perder mensagens embora não as tratem. Assim, confiam no protocolo de comunicação para evitar que ocorram mensagens perdidas. Entretanto, a configuração do sistema prevê a utilização de canais não confiáveis.

Dentre as configurações possíveis sobram três: nPP-TP-nC, PP-nTP-nC e nPP-nTP-C. Estas combinações de aplicações são razoáveis uma vez que as premissas não são violadas nem os recursos são desperdiçados.

Uma vez definidos os sistemas, serão instrumentados com o módulo de recuperação, visando adicionar um mecanismo de tolerância a falhas. Este módulo será colocado entre o nível da aplicação e o da comunicação. A análise destas três configurações de sistemas tolerantes a falhas será exposta a seguir.

2.2.1 Configuração de sistema nPP-nTP-C

Nesta configuração, a aplicação não admite a perda de mensagens mas não provê meios que impeçam sua ocorrência. Portanto, confia em algum protocolo externo que garanta a entrega das mensagens. Para isso, a aplicação foi desenvolvida sobre um protocolo de comunicação confiável.

Com a inclusão do módulo de recuperação, a aplicação terá acesso à comunicação através do módulo de recuperação.

Quando em operação sem falhas, a recuperação não interfere com a operação dos outros dois módulos. Desta forma, mensagens perdidas devido à comunicação serão tratadas e a aplicação perceberá um canal que não perde mensagens.

Entretanto, quando ocorrer uma falha e o sistema for forçado a retornar, poderá haver mensagens perdidas devido à recuperação. Assim, ou o protocolo de comunicação consegue tratá-las ou o módulo de recuperação deve fazê-lo, de maneira que a aplicação continue a perceber um canal de comunicação confiável.

Desta forma, esta configuração de sistema leva à necessidade do tratamento das mensagens perdidas pela recuperação.

2.2.2 Configuração de sistema PP-nTP-nC

Nesta configuração a aplicação pode perder mensagens, portanto não as trata nem usa um canal confiável.

A configuração PP-nTP-nC é, no mínimo, interessante. Parece difícil imaginar uma aplicação que possa operar corretamente mesmo com mensagens perdidas e que faça uso de mecanismos de tolerância a falhas. Por exemplo, aplicações probabilísticas fazem *broadcast* enviando mensagens para alguns destinos, na expectativa que estes repitam a mensagem até que todos recebam-na. Note-se que não há problema se algumas mensagens forem perdidas. Entretanto, percebe-se que este mecanismo visa, exatamente, garantir que todos recebam a informação, mesmo que em termos probabilísticos. Desta forma, este tipo de aplicação fica melhor classificado como do tipo PP-TP-nC: trata as mensagens perdidas, mesmo que em termos probabilísticos.

Acesso a páginas da Internet é outro tipo de aplicação em que se pode perder mensagens, pois sempre é possível repetir a solicitação. Entretanto, estas aplicações não são do tipo que necessitem de mecanismos de tolerância a falhas.

Assim, não tendo sido identificado exemplo significativo de aplicações que se enquadrem no tipo PP-nTP-nC e que demandem o uso de tolerância a falhas, este

tipo de aplicação será descartada como uma possível configuração de sistema.

2.2.3 Configuração de sistema nPP-TP-nC

A categoria nPP-TP-nC é aquela em que a aplicação trata as mensagens perdidas. Portanto, nenhum dos protocolos de suporte necessita fazê-lo. Apesar disso, pode-se questionar esta afirmação.

Em primeiro lugar, considerando-se um sistema construído sem o algoritmo de recuperação, não parece haver problemas, pois a aplicação impede que as mensagens sejam perdidas nos canais.

Se o módulo de recuperação for adicionado, como o tratamento das mensagens perdidas existente na aplicação será capaz de tratar as mensagens devido à recuperação? Supondo-se que exista, na aplicação, um algoritmo para tratar as mensagens perdidas, qual será sua eficiência no tratamento daquelas geradas pela recuperação?

Pode-se obter algumas respostas através do estudo do sistema, buscando verificar se existe algum tipo de premissa associada a sua capacidade de tratar mensagens perdidas devido à recuperação.

Procedendo a esta análise, percebe-se duas possibilidades no que tange ao tipo de algoritmo. O algoritmo:

- trata explicitamente mensagens perdidas pela recuperação;
- não consegue distinguir os dois tipos de mensagens.

Se a aplicação trata especificamente as mensagens perdidas devido à recuperação, então os dois módulos podem estar fortemente acoplados. Assim, mesmo solucionando a questão, percebe-se que parte do protocolo de recuperação está misturado na aplicação. Isso pode gerar conseqüências imprevisíveis, caso seja necessário alterar um dos dois protocolos. Certamente, não é uma boa técnica de programação.

Se a aplicação implementa um algoritmo que não consegue distinguir um tipo de mensagem perdida do outro, o retorno de estado fará com que as mensagens perdidas devido à recuperação apareçam para a aplicação como mensagens perdidas pela comunicação, sendo então tratadas.

Mesmo neste caso, o projetista do sistema deve conhecer, além da forma como a recuperação pode gerar as mensagens perdidas, a forma como são tratadas pela aplicação, para garantir que a composição dos dois protocolos não leve à perda de mensagens da aplicação.

Como, em geral, os módulos são desenvolvidos de forma mais ou menos independente, é comum que não se tenha acesso às características de um dos dois algoritmos. Desta forma, mesmo com o uso de canais não confiáveis, é bastante razoável utilizar um algoritmo de recuperação que seja capaz de tratar as mensagens perdidas por ele causadas.

Finalmente, é comum que os algoritmos para a recuperação de mensagens (como o que a aplicação poderia estar usando) considere, apenas, as mensagens perdidas pela comunicação, não se preocupando se o mecanismo é capaz de tratar aquelas causadas pela recuperação.

2.2.4 Conclusão

Com base no que foi discutido, pode-se dizer que, se o algoritmo de recuperação é capaz de tratar as mensagens perdidas por ele causadas, então este algoritmo poderá ser usado com qualquer tipo de sistema, sem que seja necessário verificar o comportamento do conjunto aplicação/recuperação/comunicação.

Além disso, o uso de um algoritmo de recuperação que não trata este tipo de mensagem estará sujeito a falhas, as quais poderão demandar um tempo de depuração excessivamente longo.

Considerando os pressupostos acima, optou-se por introduzir o tratamento das mensagens perdidas devido à recuperação, no algoritmo proposto neste trabalho.

2.3 Detecção de defeitos

O enfoque deste trabalho é a recuperação de processos. Assim, a questão do detector de defeitos não será tratada, deixando para o restante do sistema a responsabilidade desta tarefa.

Mesmo assim, é necessário que sejam listadas as principais características a serem oferecidas pelo detector de defeitos, que se supõe está disponível no sistema.

O detector de defeitos deve ser capaz de informar ao sistema quando um dos processos falhou. Ou seja, de acordo com o modelo de falhas assumido, deve ser capaz de informar quando um processo está em colapso (*crash*, segundo [CRI 91]).

Entretanto, isso não é suficiente. É necessário que o detector de defeitos seja capaz de informar quando um processo em falha recuperou-se. Assim, poderá ser iniciado o procedimento de retorno. Desta forma, o modelo mais adequado para o detector de defeitos é o *crash-recovery* [OLI 97, DOL 97, AGU 98].

O detector de defeitos também deve informar, caso alguma mensagem seja efetivamente perdida nos canais. Desta forma, podem ser iniciados os procedimentos necessários à recuperação. As situações em que o detector informa este tipo de falha corresponde àquelas em que a comunicação não foi capaz de entregar as mensagens, sejam os canais confiáveis ou não.

Percebe-se que, a partir do momento em que foi detectada a ocorrência de um defeito, o processamento da aplicação deve ser suspenso, até que o detector informe a recuperação do processo que havia entrado em colapso.

Apesar da detecção de defeitos ser de responsabilidade do sistema de suporte, algumas situações de defeitos podem ser detectadas pelo protocolo de estabelecimento dos pontos de recuperação.

O protocolo proposto envolve o envio de mensagens de coordenação. A estas deverão corresponder mensagens de resposta. Se estas não chegarem, deverão ser repetidas até um limite de tentativas, quando se pode considerar que houve defeito.

O início do estabelecimento de uma nova linha de recuperação é de responsabilidade de um coordenador, definido entre os processos que formam o sistema. Entretanto, pode ocorrer defeito no processo que está atuando como coordenador. Assim, não será iniciado o estabelecimento de uma nova linha de recuperação, mesmo quando for satisfeito o critério de início da operação. Este defeito poderá ser detectado pelos outros processos do sistema, uma vez que foram informados do término do ponto anterior.

2.4 *Livelocks*

De forma simples, um sistema não consegue progredir em seu processamento quando está em *livelock*. No contexto da recuperação, isso significa que o sistema, após detectado um defeito, retorna, reexecuta o processamento e o defeito torna a se repetir, ficando preso neste ciclo.

Um cenário envolvendo *livelocks* foi descrito por Koo e Toueg [KOO 87]. Deste cenário, pode-se perceber que *livelocks* são causados por mensagens que estavam no canal, quando ocorreu a recuperação, e cujo registro de transmissão foi removido.

A solução mais direta para este problema é limpar o canal, antes de reiniciar o processamento. Desta forma, impede-se que uma mensagem que esteja no canal seja recebida. Esta solução apresenta o inconveniente de bloquear a aplicação, até que os canais estejam limpos. Adicionalmente, dependendo do tipo de rede de comunicação, pode não ser possível este procedimento.

Koo e Toueg propuseram que, no retorno, os processos sincronizassem sua operação usando um protocolo de duas fases (*two-phase-commit-protocol*). Esta solução só apresenta um comportamento consistente porque Koo e Toueg têm por premissa que os canais são FIFO.

Nos artigos de Strom e Yemini [STR 85] e de Silva e Silva [SIL 92], é utilizado um mecanismo denominado **reencarnações**. Cada mensagem carrega consigo um número que identifica o intervalo de recuperação em que foi gerada e, a cada nova recuperação, os processos incrementam este número. Quando uma mensagem recebida apresenta este número menor do que aquele registrado no processo receptor, a mensagem é descartada. Este procedimento não bloqueia a aplicação, não requer ordenamento das mensagens e obtém um resultado equivalente ao da limpeza dos canais.

2.5 Tratamento das mensagens perdidas

A forma como os protocolos tratam as mensagens perdidas foi discutido na seção sobre consistência dos pontos de recuperação. Naquela discussão, pode-se perceber que alguns dos algoritmos não tratam das mensagens perdidas, ignorando-as ou deixando-as a cargo da comunicação.

Entretanto, conforme Chandy e Lamport [CHA 85], um conjunto de *snapshots* forma um estado global consistente quando não ocorrem mensagens órfãs e as mensagens perdidas estão registradas. Desta forma, a definição de Chandy leva em consideração que o estado global do sistema é formado pelos *snapshots* dos estados dos processos e dos canais, sendo que o estado dos canais é formado pelas mensagens enviadas mas não recebidas: são as mensagens em trânsito.

As mensagens transmitidas mas ainda não recebidas podem vir a tornarem-se perdidas. Estas são as mensagens perdidas devido à comunicação, as quais poderão estar no estado do canal, dependendo do mecanismo usado para detectá-las.

Quando, entretanto, for introduzido um mecanismo de recuperação, pode ser gerado um segundo tipo de mensagens perdidas: aquelas geradas pela própria recuperação.

Os dois tipos de mensagens perdidas são diferenciados pela sua causa: devido à comunicação ou devido à recuperação, conforme já comentado na seção 1.3.2, reforçado a seguir.

As mensagens perdidas devido à comunicação são aquelas que não puderam chegar ao seu destino devido a problemas com o canal de comunicação. Por exemplo, mensagens que chegaram corrompidas ao destino ou que encontraram os *buffers* de recepção cheios, serão descartadas. Nestes casos, em geral, um mecanismo que usa mensagens de reconhecimento de recepção associado a temporizações no transmissor resolvem satisfatoriamente a situação.

O segundo tipo de mensagens perdidas – aquelas causadas pela recuperação – poderá ocorrer caso o algoritmo de estabelecimento dos pontos de recuperação desconsidere a possibilidade de causar inconsistências. Assim, este tipo de algoritmo permite que uma mensagem seja enviada antes de uma linha de recuperação e recebida após esta mesma linha. Caso a linha de recuperação não seja utilizada, então a mensagem, realmente, não causará nenhuma inconsistência. Por outro lado, se for detectado um defeito e o sistema tiver que retornar para aquela linha de recuperação, então o sistema será colocado em um estado onde uma mensagem estará registrada como transmitida mas não como recebida, característica das mensagens perdidas. Desta forma, levando-se em consideração a condicionalidade de que uma mensagem se torne perdida, pode-se dizer que esta mensagens são condicionalmente perdidas ou ainda **potencialmente perdidas**.

Para tratar as mensagens perdidas devido à recuperação, pode-se utilizar as seguintes formas:

- retransmissão (*resend*);
- *replay*;
- impedir sua ocorrência.

Na operação de retransmissão, as mensagens são armazenadas no transmissor e, quando necessário, são novamente enviadas para seus destinos.

No caso do *replay*, as mensagens estão armazenadas no receptor. Assim, na recuperação, os processos copiam as mensagens para os *buffers* de recepção, simulando a recepção das mesmas.

Finalmente, pode-se implementar um algoritmo em que não ocorram mensagens perdidas devido à recuperação. Este é o caso do algoritmo proposto por Gendelman [GEN 99].

2.5.1 Tratamento pelo protocolo de comunicação

Sem dúvida nenhuma, deixar o tratamento das mensagens perdidas a cargo do protocolo de comunicação é a forma mais simples de solucionar a questão. O protocolo de recuperação pode ser projetado para tratar somente as mensagens órfãs, na certeza que as mensagens perdidas serão tratadas no nível do protocolo de comunicação.

Entretanto, para que este enfoque funcione, os protocolos de comunicação devem ser implementados usando mecanismos que possibilitem tanto o tratamento de mensagens perdidas pela comunicação quanto aquelas perdidas devido à recuperação. Dois mecanismos serão analisados:

- retransmissão;
- numeração de mensagens.

Na **retransmissão**, o receptor da mensagem é responsável por responder sempre que receber uma mensagem. Caso o transmissor não responda ou responda com indicação de erro, a mensagem deverá ser retransmitida.

Um processo decide por retransmitir uma mensagem quando recebe uma resposta negativa ou quando não receber resposta alguma, após um limite de tempo pré-determinado. Caso a resposta seja positiva, o transmissor considera que a mensagem foi recebida corretamente. Na figura 2.2, estão representados: (a) o modelo abstrato da mensagem, (b) um cenário correspondente ao modelo abstrato, quando a mensagem foi recebida corretamente e (c) um segundo cenário, correspondente ao mesmo modelo, quando o receptor detectou algum tipo de falha e foi necessária a retransmissão (a mensagem ***m'*** é a repetição da mensagem ***m***).

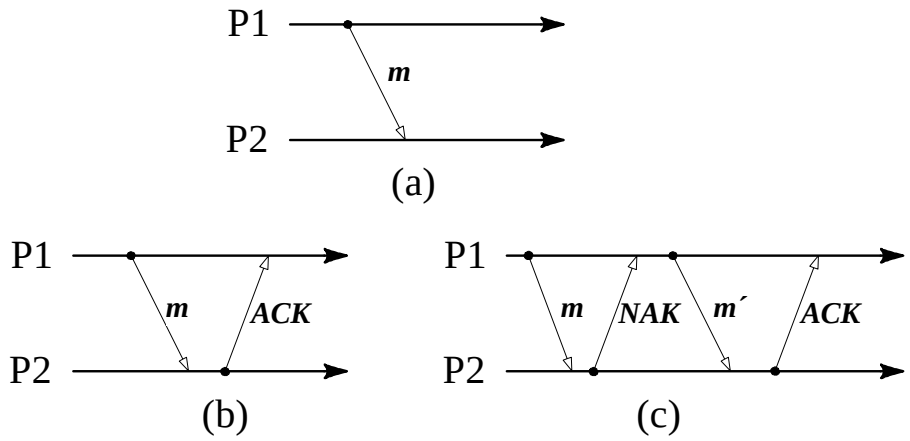

FIGURA 2.2 – Modelo abstrato das mensagens

No caso de uma mensagem ter-se perdido no canal (ou o comportamento do receptor é como se não tivesse recebido a mensagem), o transmissor deverá retransmitir a mensagem, conforme figura 2.3. Este cenário corresponde ao tratamento dado pelo protocolo de comunicação confiável, quando ocorre uma mensagem perdida, durante a operação normal (sem que tenha ocorrido retorno).

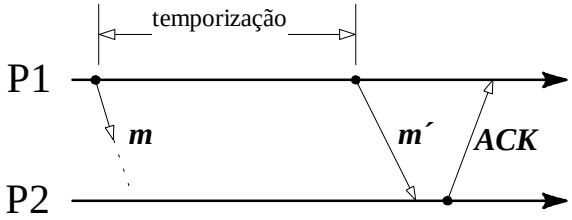

FIGURA 2.3 – Repetição de mensagens

Caso uma mensagem torne-se perdida quando ocorrer um retorno, então os *snapshots* locais foram estabelecidos conforme indicado na figura 2.4, ou seja, após o envio da mensagem, no processo P1, e antes da recepção da mesma, no processo P2.

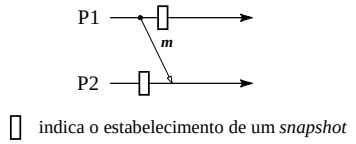

FIGURA 2.4 – Mensagem perdida

Entretanto, levando-se em consideração que as ações relacionadas às mensagens são, na realidade, formadas por um par de mensagens, e considerando-se a situação de retorno, existem várias possibilidades de estabelecimento dos *snapshots* locais de forma que transmissor e receptor não tenham a mesma informação sobre a mensagem. Estas formas de estabelecimento estão esquematizadas na figura 2.5.

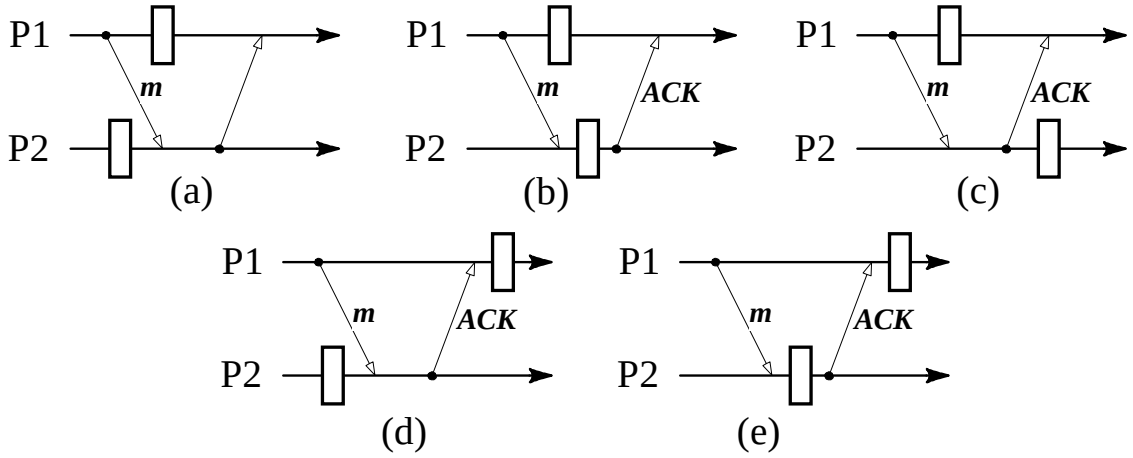

FIGURA 2.5 – Expansão das mensagens perdidas

Na figura 2.5, se houver o retorno aos últimos pontos de recuperação, nos cenários (a), (b) e (c), o processo P1 deverá repetir a mensagem, pois retornou para um estado em que a resposta não havia chegado. Nos casos (b) e (e), o processo P2 reenviará a resposta, pois retornou para um estado no qual a mensagem foi recebida porém a resposta não havia sido enviada. Por outro lado, no cenário (d), o processo P2 terá retornado para um estado em que a mensagem não havia sido recebida nem o será. Apesar disso, o processo P1 não repetirá a mensagem, uma vez que já recebeu a resposta. Desta forma, os dois processos estão inconsistentes, no que diz respeito à mensagem: P1 registrou o envio e a recepção da mensagem enquanto que P2 não possui mais informação relacionada à mensagem.

No caso de ser utilizada a **numeração de mensagens** como forma de tratar as mensagens perdidas, o transmissor deverá numerar seqüencialmente as mensagens enviadas para um determinado processo. Este, por sua vez, através da observação da seqüência numérica das mensagens, poderá detectar se houve perda. Caso ocorra quebra de seqüência, o processo solicitará a retransmissão da mensagem correspondente.

Quando o sistema retornar, após a ocorrência de um defeito, havendo mensagens perdidas, os processos poderão solicitar a retransmissão das mesmas. A

identificação de quais mensagens foram perdidas será feita quando o processo receptor verificar que o número associado à primeira mensagem recebida após a retomada do processamento está fora de ordem (haverá omissões na seqüência dos números de mensagens).

Para que este mecanismo possa operar, os processos devem manter o registro das mensagens enviadas. Entretanto, por questões práticas, este registro deve ser limitado. Os processos poderão manter uma quantidade determinada de mensagens, descartando as mais antigas, sempre que toda a memória disponível estiver sendo usada e houver a necessidade de mais espaço. A dificuldade relacionada com este mecanismo é determinar quanto de memória deve ser reservada para o registro das mensagens. Levando-se em consideração valores pré-determinados de freqüência e de tamanho médio das mensagens, quanto maior for o espaço reservado, maior será o tempo de retorno possível.

Em um mecanismo coordenado de estabelecimento dos pontos de recuperação, conhece-se antecipadamente o tempo máximo de retorno. Se o mecanismo for assíncrono, não se pode determinar, *a priori*, quanto será o tempo de retorno. No máximo, pode-se determinar o limite superior da quantidade de *snapshots* necessários para a recuperação. Wang [WAN 95a] demonstrou que este limite é $\frac{N(N-1)}{2}$, onde N é o número de processos.

2.5.2 Tratamento por *replay*

Uma forma alternativa para o tratamento das mensagens perdidas é o mecanismo de *log* associado ao *replay* (reprocessamento) das mensagens armazenadas [STR 85, JOH 90, LEE 91, ELN 92, WU 93]. Neste mecanismo, as mensagens enviadas ou recebidas são armazenadas de forma que possam ser reprocessadas quando necessário. Este armazenamento pode ser feito em memória estável ou volátil, dependendo do algoritmo.

Quando as mensagens enviadas são armazenadas no transmissor, não é necessário o seu salvamento em memória estável. Entretanto, caso um processo falhe, terá que solicitar aos outros (supostamente livres de erros) que reenviem as mensagens que possuem registradas [PRA 96], ocupando para isso o canal de comunicação (o problema da ocupação do canal na recuperação foi salientado por Kim [KIM 93]). Este mecanismo é utilizado por Elnozahy [ELN 92], apesar de salvar o *log* das mensagens transmitidas em memória estável.

Se, de forma diferente, as mensagens tiverem sido armazenadas no receptor, os processos que falharam podem simular a recepção das mensagens que devem ser reprocessadas, copiando o conteúdo do *log* para o *buffer* de recepção de mensagens. Uma vez que as mensagens devem estar no próprio processo que falhou, devem estar registradas em memória estável. Entretanto, não é necessária a retransmissão das mesmas pelos canais de comunicação [KIM 93].

2.5.3 Evitando as mensagens perdidas

Como terceira opção, pode-se impedir que as mensagens perdidas ocorram. Para isso, as mensagens devem ser verificadas quanto ao seu potencial de se tornarem perdidas, aplicando então um tratamento adequado prévio.

Os algoritmos podem ser projetados de maneira a impedir a sua ocorrência. Por exemplo, os processos podem esperar que todas as mensagens cheguem ao seu

destino para então estabelecerem uma linha de recuperação. Entretanto, o custo desta sincronização pode levar a um desempenho inaceitável. Este mecanismo é usado por Gendelman [GEN 99].

Uma alternativa é forçar a retransmissão das mensagens que apresentam potencial de se tornarem perdidas, no caso de retorno. Esta situação está exemplificada na figura 2.6. O processo P2 detectou que $\boldsymbol{m}$ é potencialmente uma mensagem perdida. Desta forma, respondeu negativamente, forçando a retransmissão da mensagem, e evitando a possibilidade de que $\boldsymbol{m}$ venha a se tornar perdida.

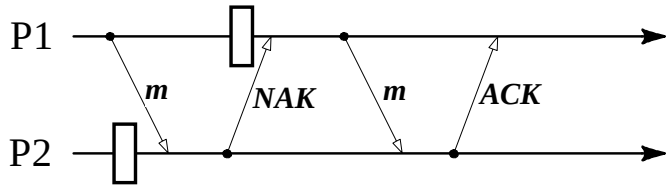

FIGURA 2.6 – Forçando a retransmissão

Este mecanismo apresenta um custo adicional relacionado com a resposta negativa e a conseqüente repetição de $\boldsymbol{m}$. Além disso, este custo deve ser multiplicado pelo número de mensagens com potencial de tornarem-se perdidas, quando forem estabelecidos os pontos de recuperação locais.

Como o número de mensagens com potencial de tornarem-se perdidas cresce com o número de pontos de recuperação, quanto maior for a freqüência destes, maior será o custo do mecanismo.

2.5.4 Discussão

Quando são comparados os mecanismos de tratamento das mensagens perdidas, percebe-se que nenhum deles fornece uma solução completa. Cada mecanismo apresenta um desempenho dependente das características do sistema onde será aplicado. Por exemplo, os mecanismos a base de *log* acarretam custos adicionais de salvamento em memória estável ou a utilização dos canais de comunicação. Por outro lado, estes custos não aparecem quando as mensagens perdidas são evitadas. Entretanto, neste último, são necessárias mais mensagens durante a operação livre de erros.

Os sistemas que utilizam mecanismos de *log* baseiam-se em um modelo de execução do tipo determinístico em partes (PWD – *Piece Wise Deterministic*). Neste modelo, a execução de um processo é formada por uma seqüência de estados, cada um iniciando por um evento não determinístico, como a recepção de uma mensagem. O processamento entre um evento e outro é determinístico. Desta forma, salvando os eventos não determinísticos (no caso, eventos de comunicação), um processo é capaz de reconstruir estados mais recentes a partir de estados anteriores e da lista de eventos não determinísticos [ELN 92].

De forma diferente, se o mecanismo utilizado para tratar as mensagens perdidas for o de evitá-las, o modelo de execução poderá ser totalmente não determinístico.

Os algoritmos que se utilizam de *log/replay* exigem, mesmo que implicitamente, um processamento do tipo PWD. Assim, dependendo da natureza da falha,

esta poderá ocorrer novamente, quando o processamento for repetido, impedindo o progresso do sistema.

Portanto, as falhas de projeto de *software* (ou simplesmente falhas de *software*) não podem ser tratadas por *log/replay*, uma vez que, quando o processamento é do tipo PWD, repetir-se-ão sempre que o processamento for refeito. Entretanto, conforme Wang [WAN 95] salientou, as falhas de *software* podem apresentar um comportamento temporário, quando em sistemas distribuídos, devido à natureza não determinística do processamento. Em um sistema distribuído, a repetição de um algoritmo pode não ocorrer exatamente da mesma forma: pode haver alterações nos processos, nos canais, no ambiente, etc, levando a algum tipo de diversidade.

Uma das conclusões de Huang [HUA 95] sobre o comportamento dos *logs* em sistemas de telecomunicações é que a reconstrução do estado do sistema, através da repetição do processamento das mensagens, é efetiva para tolerar falhas temporárias do tipo *fail-stop*. Se as falhas não forem do tipo *fail-stop* ou forem falhas do protocolo (falhas de *software*), alguns *logs* devem ser descartados (apesar de consistentes), para forçar uma execução alternativa. Novamente, observa-se a necessidade de uma repetição não determinística do processamento.

O uso de mecanismos que necessitam do modelo PWD deve ser feito com muito cuidado. Um exemplo disso é o trabalho de Borg *et al.*[BOR 89], onde foram necessários vários ajustes para garantir a execução determinística em ambiente UNIX.

Finalmente, Lin e Shin [LIN 98] salientaram o problema da contaminação dos pontos de recuperação devido à latência na detecção de erros, argumentando que a premissa de detecção imediata usada na maioria dos algoritmos não é razoável. Este problema afeta todos os mecanismos, mas é especialmente crítico quando associado à premissa de processamento PWD: uma falha cujos efeitos demorem a ser detectados poderá contaminar as mensagens armazenadas no *log* e, portanto, voltar a se manifestar a cada repetição de processamento, levando o sistema ao bloqueio.

2.6 Anexação de informações às mensagens

A implementação dos algoritmos de recuperação para sistemas distribuídos envolve a construção de módulos que serão executados em máquinas ou processadores separados e que se comunicam apenas através de mensagens. Estes módulos, exceto por alguns algoritmos (como é o caso do estabelecimento independente de pontos de recuperação), trocam informações de maneira que os pontos de recuperação locais estabelecidos sejam consistentes e possam ser utilizados no caso de um retorno.

Os mecanismos utilizados pelos algoritmos isoladamente, ou em conjunto, na troca de informações entre os módulos, podem ser divididos em três classes:

- os que enviam mensagens específicas com as informações de recuperação;
- os que acrescentam informações às mensagens da aplicação;
- os que não alteram a comunicação.

Na primeira classe, os módulos que implementam a recuperação enviam mensagens específicas que serão recebidas pelos outros módulos componentes. Estas

mensagens contêm informações que garantem a consistência entre os pontos de recuperação locais.

O mecanismo de troca de mensagens específicas (algumas vezes chamadas de mensagens do sistema) possibilita o envio de qualquer informação, em qualquer momento. Entretanto, apesar da aplicação não perceber a troca de mensagens entre os módulos de recuperação, o uso dos canais de comunicação causa uma degradação no desempenho do sistema.

No outro extremo, está o mecanismo em que a comunicação não é alterada. Os módulos de recuperação baseiam suas decisões apenas nas informações contidas nas mensagens da aplicação. São exemplos destas informações: a identificação do módulo transmissor da mensagem e a ordem na qual as mensagens foram recebidas. Apesar de fornecer pouca informação, estes mecanismos apresentam a vantagem de não interferir na comunicação e, em conseqüência, não degradam o desempenho.

O termo intermediário fica por conta do mecanismo de acréscimo de informações às mensagens da aplicação. Com este mecanismo, pode-se acrescentar qualquer tipo de informação. Entretanto, isso não pode ser feito a qualquer instante, uma vez que um módulo transmissor dependerá de uma mensagem da aplicação. Desta forma, o custo do mecanismo é menor do que enviar mensagens específicas porém maior do que não enviar informação. Este tipo de mecanismo é especialmente eficiente, quando as informações a serem acrescentadas às mensagens estão relacionadas com a transmissão ou a recepção das mesmas.

Para identificar qual o efeito da inclusão de informações às mensagens da aplicação, considere-se os seguintes dois mecanismos:

- as informações a serem trocadas são anexadas às mensagens da aplicação;
- as informações a serem trocadas são enviadas em mensagens específicas.

A relação entre o desempenho do sistema que utiliza a anexação de informações às mensagens e outro que utiliza mensagens específicas dependerá, basicamente, de dois fatores: da freqüência de troca de informações das mensagens específicas e da taxa de mensagens da aplicação.

Enquanto que a freqüência das mensagens específicas pode ser ajustada de forma mais ou menos independente, o mesmo não ocorre com a taxa de mensagens da aplicação. Assim, para cada freqüência escolhida de mensagens específicas, pode-se associar um limite na taxa de mensagens da aplicação que ainda fornecerá um desempenho melhor. Dito de outra forma, com o aumento da taxa de mensagens da aplicação, é possível encontrar uma freqüência razoável de troca de informações específicas que forneça um melhor desempenho.

Finalmente, para escolher como as mensagens serão trocadas, o projetista deverá utilizar critérios que levem ao melhor desempenho.

2.7 Dependências

O cálculo da dependência de comunicação entre os processos de um sistema é útil para determinar quais são os processos que devem estabelecer, de forma consistente, os seus pontos de recuperação locais.

Este cálculo pode ser feito de duas formas:

- através da dependência transitiva ou.
- através da dependência direta.

Na **dependência transitiva**, cada processo acrescenta às mensagens da aplicação dados que permitem identificar os processos dos quais recebeu mensagens. O processo receptor, a partir destas informações, é capaz de calcular toda a cadeia de dependências.

Na **dependência direta**, nenhuma informação é acrescentada. Cada processo tem registrada, somente, a identificação dos processos dos quais recebeu mensagens.

Supondo-se que um processo iniciador tenha que enviar solicitações de estabelecimento de pontos de recuperação para os processos de um sistema distribuído, não haverá necessidade de enviar mensagens para os processos dos quais não tenha dependência de comunicação. Assim, usando a informação de dependência, o iniciador poderá otimizar o número de mensagens a serem enviadas.

O iniciador dará andamento ao procedimento de solicitação de estabelecimento de novos pontos de recuperação, enviando mensagens aos processos dos quais depende. Se as dependências registradas forem do tipo transitiva, então serão enviadas mensagens de solicitação para todos os processos dos quais depende (direta ou indiretamente) e, portanto, cada processo receberá, no máximo, uma mensagem. Por outro lado, se as dependências registradas forem do tipo direto, então cada processo destino deverá repassar, recursivamente, a solicitação, para os processos dos quais depende. Assim, um processo que estiver na lista de dependências de dois processos diferentes, receberá duas mensagens.

Portanto, o uso de dependência direta tem um baixo custo associado às mensagens da aplicação, pois não é necessário anexar informações de dependência às mensagens. Por outro lado, o uso desta informação para o envio de mensagens de solicitação de estabelecimento de pontos de recuperação é caro, na medida que pode haver mensagens redundantes.

O uso da dependência transitiva envolve um custo associado às mensagens da aplicação maior do que aquele da dependência direta. Entretanto, quando usadas as informações de dependência para o envio de mensagens de solicitação de estabelecimento de pontos de recuperação, o custo é mínimo, envolvendo o envio de uma única mensagem apenas para os processos dos quais, efetivamente, depende.

Conforme discutido anteriormente, a periodicidade de estabelecimento dos pontos de recuperação e a taxa de mensagens ditarão o mecanismo mais eficiente. Caso seja escolhido um mecanismo de inclusão de informações às mensagens e levando-se em consideração que a periodicidade de estabelecimento dos pontos de recuperação pode ser ajustada, a taxa de mensagens terá de ser limitada, sob pena de um desempenho pior do que aquele apresentado por um mecanismo de dependência direta.

São exemplos de algoritmos que se utilizam de dependência os de Prakash [PRA 96], Cao [CAO 98] e o de Kim [KIM 93].

3 Lógica temporal de ações

Nas palavras de Clarke e Wing [CLA 96], "... o uso de métodos formais não garante, *a priori*, a correção. Entretanto, estes métodos podem aumentar o entendimento de um sistema, revelando inconsistências, ambiguidades e omissões que não seriam detectadas, caso não fossem usados".

O tipo de argumentação de Clarke e Wing, entre outros, e a percepção da informalidade (mesmo que com um verniz de formalismo) com que as provas de correção são tratadas, levou ao estudo de uma lógica formal que possibilitasse a especificação e a prova de correção do algoritmo proposto.

Foram analisadas três lógicas temporais. A lógica de Chandy e Misra [CHA 88] (UNIT), a de Manna e Pnueli [MAN 92] e a de Lamport [LAM 94] (TLA). Destas três escolheu-se a TLA de Lamport (mais informações sobre TLA podem ser encontradas em http://www.research.compaq.com/SRC/personal/lamport/home.html).

Neste capítulo, não se pretende ser exaustivo nem totalmente formal sobre os conceitos que serão discutidos. O objetivo é apresentar sucintamente as lógicas temporais, a TLA e os conceitos usados.

Uma ótima referência sobre TLA é o livro disponibilizado na página de Lamport, já citada. Além disso, encontram-se algumas ferramentas que podem auxiliar na tarefa. Infelizmente, a disponibilização destas ferramentas só ocorreu após a realização da especificação e prova formal apresentadas neste trabalho.

3.1 TLA - *Temporal Logic of Actions*

A primeira observação importante diz respeito às diferenças e semelhanças entre uma especificação de algoritmo no formato de um programa ou de uma lógica.

Da mesma forma que um programa, a TLA foi projetada para ser uma ferramenta prática. Entretanto, não foi abandonado o rigor do raciocínio formal, única forma de evitar erros sutis que aparecem nos algoritmos que exploram concorrência.

Como é de se esperar, programas e lógica têm objetivos diferentes. Enquanto que uma especificação formal visa capturar o raciocínio usado no projeto de um algoritmo, um programa descreve de forma precisa e exata quais e em que ordem devem ser executadas as operações.

O termo "execução" também diferencia as duas formas de descrição. Pode-se dizer que um programa é executado, no sentido que a cada comando corresponde uma seqüência de computação. No entanto, não faz sentido dizer que uma especificação formal será executada. Para entender como uma especificação se relaciona com um programa que está sendo executado, pode-se recorrer à figura 3.1. Na figura, um sistema é representado com suas entradas e saídas. Também é indicada a existência de um programa, responsável pelo comportamento do sistema. A este sistema, pode-se imaginar que tenha sido acoplado um módulo monitor, que recebe todas as informações de entrada, as saídas produzidas e algumas variáveis internas do programa.

O módulo monitor processa todas suas entradas e fornece, como saída, apenas a informação de verdadeiro ou falso. Se o relacionamento entre as entradas, saídas e as variáveis internas do programa estiverem de acordo com a descrição existente no módulo monitor, então a saída do monitor será *verdadeiro*. Caso contrário, será

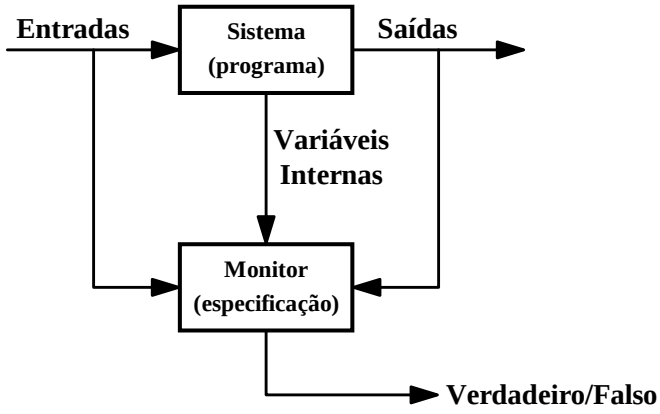

FIGURA 3.1 – Modelo de interação sistema–especificação

falso.

Obviamente, a imagem construída na figura 3.1 não ocorre na prática, pelo menos não de forma corriqueira, principalmente porque a especificação é escrita antes da implementação. Entretanto, a imagem construída serve de modelo para interpretar a afirmação de que **a especificação formal descreve como deve ser a operação do sistema**. Ou seja, se fosse construído um sistema completo como o modelo da figura 3.1 (implementação, ou programa, rodando no sistema, e especificação sendo usada como base de dados para um programa padrão de verificação, no monitor), então o bloco verificador poderia detectar qualquer desvio do comportamento pretendido da implementação contida no bloco **Sistema**.

Uma premissa importante sobressai na exposição anterior: o comportamento da implementação é verificado contra uma especificação. Ou seja, supõe-se que a especificação esteja correta: que corresponda ao comportamento pretendido. Desta forma, é necessário que se verifique a especificação. Para isso recorre-se às provas formais.

Para que seja possível provar uma descrição, é necessário um grau de formalismo que as linguagens de programação têm dificuldade em oferecer. Esta dificuldade é devida aos conceitos que as linguagens tratam. Por exemplo, as linguagens devem tratar conceitos como atribuição, a diferença entre endereço e dado e o conceito de passagem de parâmetros por valor ou por referência.

Uma característica interessante da TLA, e que teve um peso significativo na sua escolha, é o seu reduzido conjunto de novos operadores a serem entendidos, em relação a lógica proposicional comum. Para escrever uma expressão TLA, além dos operadores lógicos comuns, utiliza-se $'$ (linha), $\Box$ e $\exists$.

Apesar do modelo apresentado usar o conceito de um **programa verificador** que utiliza a especificação como referência, isso não é preciso. Na realidade, a especificação é uma fórmula lógica e, portanto, só pode ser verdadeira ou falsa. Esta fórmula diz quais são as relações aceitas para o comportamento do algoritmo.

Adicionalmente ao exposto, para que seja possível descrever o comportamento de um sistema ao longo do tempo, é necessária a noção de antes e de depois (ou seja, da passagem do tempo). Isso é obtido através da modelagem dos sistemas como seqüências de estados. Os estados, por sua vez, são identificados pelo valor

das variáveis de estado. Além disso, usam-se as variáveis para indicar o estado atual e as variáveis linha para indicar o estado futuro (ou estado para o qual haverá transição). Têm-se, então, uma lógica temporal.

Por exemplo, pode-se dizer que $x = y$, ou seja, que x e y são iguais, no estado atual. Pode-se, também, dizer que $x' = y$, significando que o valor de x no próximo estado deve ser igual ao valor de y no estado atual. Com a última forma, pode-se especificar, por exemplo, o progresso de uma variável.

3.2 Conceitos básicos

Nesta seção, serão apresentados alguns dos conceitos básicos necessários para o entendimento das especificações que usam TLA. Todos os conceitos apresentados nesta seção são originários da apresentação de Lamport sobre TLA [LAM 94].

3.2.1 Funções de estado, predicados e ações

Uma **função de estado** é uma expressão não booleana formada por variáveis e constantes. Informalmente, o significado de uma função de estado é um valor obtido da aplicação de outros valores a cada variável que forma a função.

Exemplo: $x^2 + y - 3$ é uma função de estado. Aplicando-se os valores 2 para x e 3 para y, o significado da função será o valor 4. Note-se que, do ponto de vista lógico, 2, 3 e 4 são apenas símbolos para os quais se atribui o significado das quantidades que representam.

Um **predicado de estado** é semelhante à função de estado exceto que é uma expressão booleana. Ou seja, são expressões em que aparecem operadores que relacionam duas partes da mesma. Por exemplo, são operadores: $=$ (igual) e $<$ (menor).

O significado de um predicado será sempre **verdadeiro** ou **falso**.

Exemplo: $x^2 = 3 - y$ é um predicado, pois será verdadeiro, no caso da igualdade verificar-se, ou falso, caso contrário.

Uma **ação** é semelhante a um predicado exceto que podem aparecer variáveis associadas ao estado atual e ao estado futuro (variáveis linha).

Neste caso, assim como no predicado, o significado será sempre **verdadeiro** ou **falso**.

Exemplo: $y = x' + 1$ é uma ação, pois será verdadeiro no caso da igualdade verificar-se ou falso, caso contrário. Além disso, a variável x da expressão deve ser usada com seu valor no estado futuro.

Note-se que um predicado é um caso particular de ação, onde não aparecem variáveis do estado futuro (variáveis linha).

3.2.2 Variáveis e constantes

Os termos **constante** e **variável** são entendidos de forma diferente por programadores e matemáticos. Para um programador, um valor que não se altera durante a execução do programa, é chamado de constante. Para um matemático, isso é uma variável, uma vez que pode-se não conhecer seu valor. Na terminologia da TLA, os símbolos com esta característica serão denominados de **variáveis rígidas**. Nas especificações, este tipo de variáveis serão identificadas pelos termos **constant** ou

constants.

Aquelas variáveis que não se enquadrarem à definição de variáveis rígidas, serão chamadas de **variáveis flexíveis** ou simplesmente variáveis. Nas especificações, este tipo de variável será identificado pelos termos **variable** ou **variables**.

3.2.3 Operadores temporais

Para representar a passagem de um estado para outro, ou seja, para representar o desenvolvimento do sistema ao longo do tempo, usam-se as ações (formadas por variáveis e variáveis linha).

O uso de variáveis com mais de uma linha (variáveis duas linhas, três linhas, etc...) não é permitido. Para descrever o comportamento de um sistema ao longo de vários estados é utilizado o conceito de **comportamento** (*behavior*): seqüência de estados que corresponde a uma possível execução do algoritmo descrito.

Com este conceito, pode-se definir o operador temporal $\Box$ (***always***). Seu significado é o de indicar algo que ocorre sempre, em todas as transições de um estado para outro. Assim, uma ação que sempre é verdadeira, ocorre em todas as transições do comportamento.

Por exemplo, para representar o comportamento de um contador, pode-se dizer que ou ele não é alterado ou será incrementado de uma unidade. Assim, pode-se dizer que $\Box(x' = x \vee x' = x + 1)$.

As lógicas temporais utilizam outros três operadores, que serão definidos em termos do operador $\Box$.

Supondo-se que F representa uma ação, o operador ***eventually*** indica uma condição que será verdadeira, certamente, em algum dos estados futuros. Representa-se este operador com o símbolo $\Diamond$ e sua definição formal é $\Diamond F \triangleq \neg\Box\neg F$.

Os outros dois operadores correspondem às combinações de $\Box$ e $\Diamond$.

O operador ***Infinitely Often*** – infinitamente freqüente – indica que a expressão sobre a qual é aplicado o operador ora é verdadeira ora é falsa. A notação é $\Box\Diamond F$. Na figura 3.2, é representado um exemplo de comportamento, chamado de F, que torna $\Box\Diamond F$ verdadeiro. Nos trechos hachurados, F é verdadeiro; nos demais, F é falso.

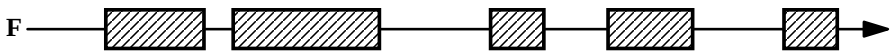

FIGURA 3.2 – Modelo de Computação – *Infinitely Often*

Finalmente, o operador ***Eventually Always*** – cuja notação é $\Diamond\Box F$ – indica que a expressão F, certamente, tornar-se-á verdadeira a partir de algum estado no futuro e, então, permanecerá desta forma.

Na figura 3.3 está representado este comportamento.

3.3 RTLA - *Raw* TLA

Antes de descrever a TLA propriamente dita, é interessante descrever uma RTLA, onde os conceitos são mais flexíveis.

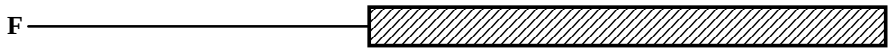

FIGURA 3.3 – Modelo de Computação – *Eventually Always*

O conceito mais importante introduzido com a RTLA é aquele em que se define o significado de uma ação ser verdadeira em um determinado comportamento.

Define-se que **uma ação é verdadeira em um comportamento** *sse* for verdadeira para os dois primeiros estados do comportamento.

As fórmulas da RTLA são formadas por ações e pelo operador $\Box$. Assim, se A for uma ação, $\Box A$ será uma fórmula da RTLA. Junto com a definição anterior, pode-se dizer que **um comportamento satisfaz** $\Box A$ *sse* cada passo do comportamento satisfaz a ação. Diz-se que cada passo do comportamento é um **passo** A.

Como, na RTLA, predicados são casos particulares de ações onde não existe referência ao estado futuro, pode-se definir um predicado de forma semelhante àquela usada para ações. Define-se que **um predicado é verdadeiro em um comportamento** *sse* for verdadeiro no primeiro estado do comportamento. Assim, se P for um predicado, **um comportamento satisfaz** $\Box P$ *sse* todos os estados do comportamento satisfizerem P.

3.3.1 Descrevendo programas com a RTLA

A forma como se usa a RTLA para descrever os sistemas (programas) será apresentada através de um exemplo.

Descrição Textual do Sistema: Deseja-se descrever um programa concorrente composto por dois processos onde, cada processo, possui um contador. Estes contadores são iniciados com zero e incrementados de tempos em tempos. Nesta descrição não é necessário modelar as causas que levam ao incremento dos contadores. O objetivo é descrever o comportamento das variáveis, quanto aos valores que podem assumir.

A descrição do programa será feita através de uma fórmula RTLA, a qual será denominada Φ.

Para descrever o programa, procura-se uma definição para Φ que seja verdadeira em todos os comportamentos compatíveis com uma computação permitida. Além disso, computações que não correspondem ao pretendido, devem tornar falsa a fórmula Φ.

Na descrição textual do comportamento pretendido pode-se identificar, pelo menos, dois requisitos de natureza diversa. O primeiro diz respeito aos valores de início das variáveis. O segundo informa como estas variáveis devem progredir ao longo do tempo. Estes dois requisitos originarão sub-fórmulas distintas.

O requisito de inicialização das variáveis é, pela sua própria natureza, um predicado. Deve-se garantir que seja verdadeiro no primeiro estado do comportamento, não importando o restante dos estados. Assim, a fórmula que modela a inicialização é um predicado simples, que será representado pelo símbolo $Init_{\Phi}$.

No caso do problema a ser modelado, os contadores (variáveis) devem ser inicializados com zero. Estes contadores serão chamados de x e y. Tem-se, então:

$$Init_\Phi \triangleq (x = 0) \wedge (y = 0) \tag{3.1}$$

A segunda parte da especificação deve informar como as variáveis progridem no tempo. Levando em consideração que os contadores estão em processos distintos, pode-se representar as ações do sistema através da disjunção das ações de cada processo. Assim, usando M para a ação do sistema e M_1 e M_2 para as ações de cada processo, pode-se escrever:

$$M \triangleq M_1 \vee M_2 \tag{3.2}$$

Esta fórmula informa que serão válidos todos os comportamentos em que ocorra ação no processo 1 ou no processo 2 ou, simultaneamente, nos dois processos.

Para completar a fórmula Φ, basta lembrar que a ação M deve ser válida para todos os pares de estados de todos os comportamentos aceitos para o sistema a ser especificado. O operador $\Box$ será usado para indicar este fato. Tem-se, então, a seguinte definição para Φ.

$$\Phi \triangleq Init_\Phi \wedge \Box M \tag{3.3}$$

A equação 3.3 restringe o conjunto de todos os comportamentos possíveis àqueles que correspondem às computações pretendidas: no primeiro estado, a fórmula $Init_\Phi$ deve ser verdadeira e o relacionamento entre dois estados seguidos quaisquer deve ser uma ação M.

Resta, agora, definir as ações M_1 e M_2. Conforme a descrição do sistema, cada ação deverá incrementar um dos contadores do sistema. Assim, pode-se escrever:

$$M_1 \triangleq x' = x + 1 \tag{3.4}$$

$$M_2 \triangleq y' = y + 1 \tag{3.5}$$

Entretanto, as fórmulas 3.4 e 3.5 são ambíguas. Se M_1 for verdadeira e M_2 não, então Φ será verdadeira, o que corresponde a um comportamento válido. Mas esta combinação e a definição das ações informa **apenas** que x foi incrementado. Não informa nada a respeito de y (exceto que não foi incrementado). Isso significa que y pode assumir qualquer valor, o que não é aceito como comportamento de um contador. Assim, as fórmulas 3.4 e 3.5 serão alteradas de maneira a evitar este comportamento. As fórmulas ficarão:

$$M_1 \triangleq x' = x + 1 \wedge y' = y \tag{3.6}$$

$$M_2 \triangleq y' = y + 1 \wedge x' = x \tag{3.7}$$

Os comportamentos permitidos por estas fórmulas estão mais próximos daquele pretendido.

Percebe-se, entretanto, que a forma como as ações foram definidas não permite que as variáveis x e y sejam incrementadas simultaneamente. Esta diferença é mais sutil, mas pode implicar em especificações erradas (que não refletem a computação pretendida). Estes erros sutis poderão não ser detectados em uma análise informal, sendo revelados quando a descrição for submetida à prova formal.

O resultado da exposição anterior pode ser observado na figura 3.4, onde está descrita a fórmula Φ.

$$\begin{aligned} Init_\Phi &\triangleq (x = 0) \wedge (y = 0) \\ M_1 &\triangleq x' = x + 1 \wedge y' = y \\ M_2 &\triangleq y' = y + 1 \wedge x' = x \\ M &\triangleq M_1 \vee M_2 \\ \Phi &\triangleq Init_\Phi \wedge \Box M \end{aligned}$$

FIGURA 3.4 – Descrição RTLA

3.4 TLA – *stuttering*, *safety*, *liveness* e *fairness*

Conceitualmente, a interpretação de uma descrição RTLA ou TLA é a mesma. Entretanto, é necessário que as fórmulas aceitas por uma lógica temporal sejam mais rigorosas, o que levará à TLA.

3.4.1 *Stuttering*

Em um sistema real, a ocorrência de ações pode ser intercalada por períodos em que o sistema não executa operação alguma. Os passos que ocorrem durante estes períodos de inatividade, nos quais as variáveis da descrição não são alteradas, são chamados de **stuttering steps**. Para modelar este comportamento, as fórmulas da TLA devem incluir a possibilidade de que nenhuma ação ocorra.

Para indicar esta ausência de ações, pode-se construir as fórmulas das ações de maneira que, ou as variáveis sejam alteradas (ocorra a ação) ou as variáveis não sejam alteradas. Se M for uma ação, deve-se representá-la por $M \vee v' = v$, onde v é o conjunto das variáveis flexíveis do sistema.

No caso do exemplo da figura 3.4, a fórmula Φ será alterada para:

$$\Phi \triangleq Init_\Phi \wedge \Box(M \vee ((x' = x) \wedge (y' = y))) \tag{3.8}$$

A TLA introduz notação para representar pares ordenados. Por exemplo, $< x, y >$ é um par ordenado. Desta forma, pode-se reescrever a fórmula 3.8:

$$\Phi \triangleq Init_\Phi \wedge \Box(M \vee < x, y >' =< x, y >) \tag{3.9}$$

Genericamente, se f for uma função de estado, pode-se escrever:

$$\Phi \triangleq Init_\Phi \wedge \Box(M \vee f' = f) \tag{3.10}$$

Finalmente, a TLA utiliza uma notação mais compacta para indicar o que está dito na fórmula 3.10:

$$\Phi \triangleq Init_\Phi \wedge \Box[M]_f \tag{3.11}$$

A fórmula 3.11 apresenta a estrutura aceita para fórmulas da TLA. São admitidos, apenas, predicados e fórmulas com o formato $\Box[M]_f$.

3.4.2 *Safety*, *liveness* e *fairness*

O objetivo de se escrever uma especificação é representar todos os comportamentos considerados corretos, para o sistema sob descrição. Para isso, busca-se representar as fórmulas de maneira a excluir qualquer comportamento incorreto. Este é o conceito básico de **segurança** (tradução livre de *safety*). As fórmulas escritas devem garantir que nada errado venha a ocorrer.

Entretanto, as fórmulas da TLA permitem que se descreva um sistema em que nada ocorra. Apesar de correto, no que diz respeito a evitar que algo errado ocorra, não parece razoável que este comportamento estenda-se *ad infinitum*.

Para evitar um comportamento em que nada ocorra, as fórmulas da TLA podem ser acrescidas com ações que garantam que algo ocorra. É a chamada propriedade de *liveness*: garantia de progresso do sistema. Assim, associando-se *liveness* e *safety*, garante-se que o sistema progredirá em acordo com um dos comportamentos especificados.

Pode-se demonstrar que a fórmula adicional, para prover *liveness*, tem a forma geral de fórmulas TLA ($\Box[M]_f$).

Para garantir que todas a ações que formam a especificação tenham chance de serem ativadas, é necessário garantir a propriedade de ***fairness***: a justiça na distribuição de processamento.

Um problema adicional das fórmulas de *liveness*, da maneira proposta por Lamport [LAM 94], é a possibilidade que sejam incluídas novas fórmulas de *safety*. Este efeito não é desejado, uma vez que introduz restrições aos comportamentos aceitos pela especificação, que não foram explicitados (estão implícitos nas fórmulas de *liveness*). Além disso, podem levar a uma especificação que não condiz com o sistema que se deseja modelar.

Entretanto, pode-se demonstrar que as fórmulas de *fairness* da TLA também garantem a propriedade de *liveness*. Além disso, estas fórmulas não acrescentam nenhuma propriedade de *safety*.

Então, mesmo em descrições onde é necessário garantir a propriedade de *liveness*, as fórmulas podem conter termos que representem a propriedade de *fairness* e não conter termos específicos para representar *liveness*. Uma discussão mais detalhada pode ser encontrada nos trabalhos de Lamport sobre TLA.

3.4.3 O que usar?

A discussão anterior leva a alguns resultados práticos. O primeiro é que, caso seja necessário representar *liveness* ou *fairness*, pode-se fazê-lo pela adição de uma expressão que é o resultado da aplicação de algumas transformações às fórmulas de *safety*.

O segundo resultado é que se pode demonstrar muitas das propriedades importantes dos sistemas, usando apenas as propriedades de *safety*. Isso acontece porque, em geral, é necessário garantir que não ocorra processamento errado.

Finalmente, a verificação de *liveness* é necessária para demonstrar que algo vai ocorrer como, por exemplo, que um sistema não entrará em *deadlock*.

3.5 Operadores da TLA

A definição de TLA apresentada até agora seria suficiente para especificar os sistemas. Entretanto, alguns operadores adicionais são definidos.

Não serão listados nem definidos todos os operadores da TLA. Serão apresentados, apenas, aqueles utilizados na especificação e prova do algoritmo proposto neste trabalho.

Os operadores podem ser agrupados da seguinte forma:

- operadores lógicos;
- operadores de conjuntos;
- operadores sobre funções;
- operadores sobre registros (*records*);
- operadores sobre n-uplas (*tuples*);
- operadores sobre ações;
- operadores temporais.

Os grupos de operadores serão apresentados e descritos nos itens que seguem.

3.5.1 Operadores lógicos

Os operadores lógicos correspondem àqueles usados na lógica proposicional comum. São os operadores $\wedge$, $\vee$, $\neg$ e $\Rightarrow$ (implicação).

É definido um conjunto chamado BOOLEAN, formado pelos valores TRUE e FALSE.

Também fazem parte do conjunto de operadores lógicos os quantificadores universal e existencial: $\forall x : p$, $\exists x : p$, $\forall x \in S : p$ e $\exists x \in S : p$.

Adicionalmente, foi definido o operador CHOOSE. Com este operador, pode-se definir um valor relacionando apenas as suas propriedades. Por exemplo, $UmNat =$ CHOOSE $n : n \in Nat$, permite que seja escolhido um número natural, sem dizer qual deles. Na realidade, não é importante o valor que *UmNat* vai receber mas sim a sua propriedade (neste caso, a de ser um número Natural).

3.5.2 Operadores de conjuntos

São os operadores que relacionam conjuntos e elementos a conjuntos.

Estes operadores são: $=$, $\neq$, $\in$, $\notin$, $\cup$, $\cap$, $\subseteq$ e $\setminus$ (diferença de conjuntos).

O operador SUBSET merece comentário. Escrever SUBSET S não significa um subconjunto específico de S, mas o conjunto de todos os subconjuntos possíveis de S.

3.5.3 Operadores sobre funções

Os operadores sobre funções são os mais usados nas especificações e provas. Sua notação é semelhante a de vetores e matrizes, usada, por exemplo, nas linguagens Pascal e C.

A aplicação de uma função f sobre um elemento e é notada por $f[e]$. Isso é diferente de, por exemplo, $f(e)$. O primeiro é uma função enquanto que o segundo é um operador.

Uma função possui um domínio bem definido enquanto que um operador indica uma substituição sintática. Além disso, usar f, de $f[e]$, isoladamente, tem significado: é uma função; enquanto que f, de $f(e)$, não tem significado. A diferença entre funções e operadores foi discutida por Lamport [LAM 99].

O operador DOMAIN f informa o conjunto domínio da função f.

A notação $[x \in S \mapsto e]$ representa uma função que tem domínio em S e que, a cada elemento x do domínio, associa um valor e. Assim, se esta função for denominada de f, $f[x] = e$, para todo $x \in S$.

Uma notação bastante importante é aquela que representa a alteração de um elemento da função, mantendo todos os outros inalterados. A notação $[f$ EXCEPT $![e_1] = e_2]$ representa uma função g, **diferente de** f, que tem o mesmo domínio e que associa os mesmo valores a todos os elementos deste domínio, exceto para o valor e_1.

O símbolo **!** é uma forma compacta de representar f. Além disso, pode-se usar o símbolo **@** na expressão e_2 para representar $f[e_1]$.

Por exemplo, para representar a ação de incremento de um dos contadores de um conjunto de contadores, sem alterar os outros, pode-se usar:

$$Cont' = [Cont \text{ EXCEPT } ![p] = @ + 1] \tag{3.12}$$

Se não fossem usados os símbolos **!** e **@**, a fórmula 3.12 poderia ser escrita da seguinte forma:

$$Cont' = [Cont \text{ EXCEPT } Cont[p] = Cont[p] + 1] \tag{3.13}$$

Pode-se usar a fórmula 3.13, entretanto é mais comum encontra-se a notação 3.12.

3.5.4 Operadores sobre *records*

Duas notações serão apresentadas: a definição de um conjunto de registros, que pode ser usada para definir todos os registros aceitos como válidos, e a forma de acesso aos elementos de um registro.

Para definir um registro, utiliza-se a notação $[h_1 : S_1, ..., h_n : S_n]$. Esta notação representa todos os registros, com campos $h_1, ..., h_n$, onde $h_i \in S_i$.

O acesso a um campo h de um registro r é representado pela notação $r.h$.

3.5.5 Operadores sobre *tuples*

São três as notações para n-uplas: a definição de um conjunto de n-uplas, a representação de uma n-upla e o acesso a um dos elementos da n-upla.

Para definir uma n-upla, usa-se a notação $S_1 \times ... \times S_n$, onde cada S_i representa o conjunto ao qual o elemento de ordem i ($i \in \{1, 2, ..., n\}$) pertence.

Como uma n-upla é ordenada, pode-se indicar os valores dos elementos componentes apenas pela sua posição na n-upla. Assim, para representar uma n-upla específica, utiliza-se a notação $< e_1, ..., e_n >$.

Finalmente, o acesso ao i-ésimo elemento da n-upla p é notado por $p[i]$.

Percebe-se que a notação de um elemento da n-upla é idêntica a de uma função. Isso não é uma coincidência: realmente, na TLA, um n-upla representa uma função.

Por exemplo, pode-se escrever $< a, b, c > [2]$, o que é equivalente ao elemento b. Ou seja, esta n-upla representa uma função, que tem como domínio o conjunto $\{1, 2, 3\}$, e que associa a cada valor do domínio os valores a, b e c, respectivamente.

3.5.6 Operadores sobre ações

Além dos operadores já descritos (caso do operador ' – linha – e do operador $[A]_f$), deve-se citar o operador UNCHANGED. Este operador, aplicado a uma função de estado f, indica que o valor de f não é alterado.

3.5.7 Operadores temporais

Vários são os operadores nessa modalidade. Entretanto, será utilizado, apenas, o operador $\Box$, descrito anteriormente.

3.6 Significado de prova formal

Antes de passar à utilização da TLA na especificação e prova do algoritmo proposto neste trabalho, é importante esclarecer o que se entende por **prova formal**, algumas vezes referida como prova de correção.

Uma prova formal nada mais é do que a demonstração da implicação de um conjunto de fórmulas (P) em outro conjunto (Q), ou seja, $P \Rightarrow Q$. O conjunto P é formado pelas fórmulas a serem demonstradas enquanto que o conjunto Q é formado por fórmulas que definem o comportamento pretendido.

Isso seria suficiente, não fosse a questão: se a referência para a prova é o conjunto de fórmulas Q, quem garante que este conjunto representa, de fato, o comportamento pretendido?

Na realidade, não existe uma resposta simples para esta questão. O conjunto Q deve representar, de forma simples, a operação pretendida. Esta simplicidade deve ser tal que permita a sua verificação pela simples inspeção das fórmulas. Uma discussão sobre o assunto pode ser encontrada em vários trabalhos do grupo de pesquisa do Prof. Peter Ladkin (http://www.rvs.uni-bielefeld.de/publications/Reports).

Por exemplo: seja um sistema formado por dois processos, P_1 e P_2, que trocam mensagens. No início, o processo P_1 envia uma mensagem com uma marca. Quando o processo P_2 recebe a marca, reenvia-a para o processo P_1. Este, por sua vez, envia, novamente, a marca para P_2, e esta seqüência se repete *ad infinitum*.

Este exemplo, usado por Chandy e Lamport [CHA 85], foi chamado de um sistema que conserva uma única marca (*single token conservation*).

A especificação das ações deste sistema envolvem o envio e recepção de mensagens, o gerenciamento dos canais de comunicação e o tratamento da marca. Entretanto, para descrever qual o comportamento pretendido, basta dizer que, durante todo o tempo, o sistema deve ter uma e apenas uma marca.

Desta forma, no caso do exemplo, tem-se um conjunto de fórmulas que representa todas as ações previstas para o sistema (conjunto P – modelagem do envio de mensagens e tratamento da marca), e uma fórmula que indica o comportamento pretendido (conjunto Q – a propriedade que só pode existir uma única marca).

Pode-se estabelecer duas conclusões: a primeira é que o conjunto de referência (conjunto Q – aquele contra o qual verifica-se se a especificação proposta está correta) expressa o que se deseja, enquanto que o primeiro conjunto (conjunto P – a especificação) diz como se pretende atingir o objetivo.

A segunda conclusão é que não existe uma prova de correção **absoluta**. Sempre é necessário descrever o comportamento pretendido, de maneira a estabelecer uma referência para a verificação formal.

4 Descrição informal do algoritmo

A primeira versão do algoritmo, cuja descrição inicia nesta seção, foi publicada anteriormente [CEC 2001]. Foram descritas as principais características e as premissas consideradas. Também foi apresentada a primeira versão da especificação sem a prova formal, que foi desenvolvida posteriormente.

A descrição do algoritmo será feita em duas partes. A primeira, assunto deste capítulo, é a apresentação informal do algoritmo. A segunda, a ser apresentada no capítulo 6, é onde o algoritmo será apresentado formalmente, através de sua especificação e prova de correção.

4.1 Descrição geral

Como o algoritmo é coordenado, periodicamente são estabelecidos pontos de recuperação de modo a compor linhas de recuperação. A seguir, são descritos os passos desta atividade.

Um coordenador inicia o estabelecimento de uma nova linha de recuperação, enviando uma solicitação a cada processo do sistema. Esta nova linha de recuperação será chamada de **CurrCP**.

Ao receberem esta solicitação, os processos estabelecem seus pontos de recuperação e respondem ao coordenador.

Como o sistema deve assegurar a existência de uma linha de recuperação para o caso de falha durante o estabelecimento de **CurrCP**, é mantida a linha de recuperação anterior. Esta linha de recuperação será chamada de **PrevCP**.

O coordenador, quando receber as respostas de todos os outros processos, informa-os que podem iniciar o procedimento de coleta de lixo.

Ao receberem a liberação para coleta de lixo, os processos copiam (ou alteram, convenientemente, os apontadores dos pontos de recuperação) o seu último ponto de recuperação estabelecido (armazenado em **CurrCP**) no ponto anterior (**PrevCP**). Em seguida, respondem ao coordenador.

Do ponto de vista da implementação, a coleta de lixo poderia ser realizada pela alteração de um ponteiro; do ponto de vista da especificação, a coleta de lixo será representada pela igualdade de **PrevCP'** com **CurrCP**.

Ao receber as respostas de todos os outros processos, o coordenador encerra a operação de estabelecimento desta linha de recuperação. Então, estará pronto para iniciar uma nova.

Na descrição anterior, não foram apresentados alguns problemas relacionados com o tipo de algoritmo usado e o modelo de sistema distribuído, para o qual foi projetado. Os seguintes problemas serão discutidos e terão apresentadas propostas de solução:

- como garantir uma operação correta quando os canais são **não-FIFO**;
- como é obtida a **coordenação entre processos**;
- como é garantida a **não intrusão na aplicação** (não bloqueio da aplicação);
- como é obtida uma **coleta de lixo** consistente e não bloqueante;

Deixar de **garantir a ordenação das mensagens** pode levar à troca de ordem de uma mensagem de solicitação de estabelecimento de ponto de recuperação e uma mensagem da aplicação. Isso acarretaria o estabelecimento inconsistente dos pontos de recuperação. Para resolver este caso, todas as mensagens têm associado um índice que identifica qual foi o último ponto de recuperação estabelecido pelo seu processo transmissor (este índice identifica um intervalo de ponto de recuperação). Ao receber qualquer mensagem, o receptor verifica este índice. Se for maior que o seu próprio índice, então um novo ponto de recuperação é estabelecido, antes da entrega da mensagem para a aplicação; caso contrário, a mensagem é entregue, sem realizar outros procedimentos.

Como já descrito, existem apenas dois conjuntos de pontos de recuperação: **CurrCP** e **PrevCP**. Para que o algoritmo esteja correto, deve ser garantido que sempre, pelo menos um deles seja consistente. Assim, duas etapas são críticas: aquela em que um novo ponto de recuperação está sendo estabelecido (**CurrCP** estará inconsistente e, portando, **PrevCP** deve ser consistente) e aquela em que um antigo ponto de recuperação está sendo descartado (**PrevCP** estará inconsistente e, portanto, **CurrCP** deve ser consistente). Para alcançar estes requisitos, as ações que formam o algoritmo devem **estar coordenadas adequadamente**.

O **não bloqueio da aplicação** impõe restrições adicionais ao algoritmo. Este deve estabelecer os pontos de recuperação, de forma consistente, mesmo com os processos trocando mensagens de aplicação e reconhecimento. Novamente, a solução encontrada foi através do uso dos índices dos intervalos dos pontos de recuperação.

A **coleta de lixo** foi projetada de maneira a não bloquear a aplicação. Em paralelo com a troca de mensagens, os processos comunicam-se, visando a coleta de lixo. Isso é feito pelos conjuntos de mensagens que, na descrição, são do tipo **CMT** e **ACMT** (vide ações **RxAckReq**, **RxAckCmt** e **RxCmt** na especificação). É interessante notar que o mecanismo de coleta de lixo não interfere com o estabelecimento da linha de recuperação, uma vez que não ocorrem de forma concorrente. A única interferência diz respeito à conservação dos recursos disponíveis (no caso, memória estável). Apesar de não ser necessário, o algoritmo proposto descarta os pontos que foram substituídos por outros mais novos.

4.2 Operação do algoritmo

Todo algoritmo que pretende ser aplicado na recuperação de processos deve ser descrito em suas duas possibilidades de operação: quando o sistema está livre de erros e quando não está.

Na operação livre de erros, o algoritmo de recuperação deve estabelecer, periodicamente, pontos de recuperação. Desta forma, quando ocorrerem falhas, o sistema poderá retornar à linha de recuperação mais conveniente. Assim, nas seções que seguem, o algoritmo proposto neste trabalho será apresentado dividido em duas etapas, correspondentes às possibilidades de operação mencionadas.

4.2.1 Estabelecimento de uma linha de recuperação

Conforme apresentado na seção 4.1, são quatro as etapas que levam ao estabelecimento de uma linha de recuperação consistente.

- Solicitação de estabelecimento;
- Resposta à solicitação;
- Confirmação do estabelecimento;
- Resposta à confirmação.

Para **iniciar uma nova linha de recuperação**, o coordenador (**Mngr**) estabelece seu ponto de recuperação local e, antes de qualquer outra mensagem ser transmitida ou recebida, envia mensagens de solicitação de estabelecimento de um novo ponto de recuperação (**REQ**) para todos os processos do sistema. Esta mensagem, além de seu significado intrínseco, carrega a identificação do último ponto de recuperação estabelecido no transmissor. Desta forma, os processos que receberem esta informação poderão verificar se já estabeleceram o seu ponto de recuperação local correspondente.

Se os canais fossem modelados como FIFO, não seria necessário mais nenhuma operação (conforme Chandy e Lamport [CHA 85]). Entretanto, para canais não FIFO, uma vez que não se pode garantir a ordem de recepção das mensagens, outras formas de controle devem ser acrescentadas. Para solucionar a questão da não ordenação das mensagens, será utilizado um mecanismo semelhante àquele proposto por Lai e Yang [LAI 87]. Assim, todas as mensagens da aplicação terão anexados o número de identificação do último ponto de recuperação estabelecido no transmissor. Desta forma, antes de processarem as mensagens recebidas do coordenador e, dependendo da identificação carregada pela mensagem e do índice do último ponto de recuperação do receptor, estes poderão ser levados a estabelecer novos pontos de recuperação locais.

Levando-se em consideração o mecanismo descrito, ao receber uma mensagem **REQ**, o processo destino (P2, P3 ou P4 na figura 4.1) pode responder de duas formas distintas: estabelecer um ponto de recuperação e responder ao processo coordenador ((a) na figura 4.1); apenas responder ao coordenador, pois o ponto solicitado já foi estabelecido ((b) na figura 4.1).

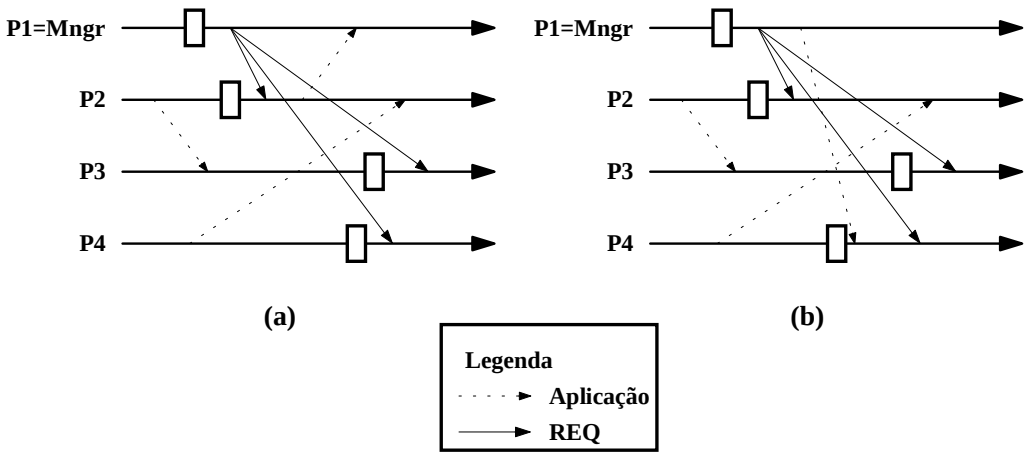

FIGURA 4.1 – Estabelecimento de ponto de recuperação no receptor das mensagens

Os processos só estabelecerão um novo ponto de recuperação se a solicitação recebida contiver um número de identificação maior do que aquele do último registrado no receptor. Assim, caso uma mensagem da aplicação com o número de identificação maior do que o do processo receptor chegue ao destino antes da mensagem **REQ**, o processo estabelecerá um novo ponto de recuperação. Quando a mensagem **REQ** chegar, o processo já terá estabelecido seu ponto de recuperação local, bastando apenas responder.

Caso ocorra de um processo receber uma mensagem com um número de identificação menor que o seu próprio, estará configurada uma mensagem perdida. Na realidade, esta é uma mensagem potencialmente perdida, uma vez que só se efetivará esta condição caso ocorra o retorno para os pontos de recuperação, em relação aos quais a mensagem é perdida. Um cenário correspondente a situação explicada está apresentado na figura 4.2, onde a mensagem é potencialmente perdida em relação aos pontos de recuperação C_1 e C_2. Estes, por sua vez, determinam o encerramento de um intervalo de ponto de recuperação (identificado pelo índice $n-1$) e o início de um novo intervalo de ponto de recuperação (identificado pelo índice n). No recebimento, o índice da mensagem $(n-1)$ será menor do que o índice do receptor (n).

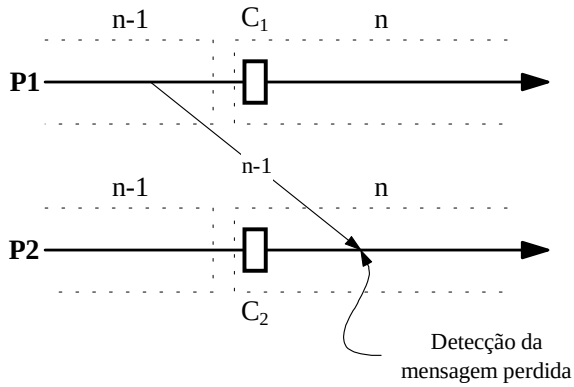

FIGURA 4.2 – Mensagem potencialmente perdida

No caso de ser detectada uma mensagem potencialmente perdida, quando o processo receptor enviar a mensagem de resposta, esta conterá um número de identificação que poderá forçar o estabelecimento de um novo ponto de recuperação, no processo transmissor da mensagem original. Isso acontecerá quando o transmissor não tiver recebido, ainda, a solicitação de estabelecimento de um novo ponto de recuperação.

O uso dos identificadores dos pontos de recuperação visa garantir que não ocorram mensagens órfãs. Para evitar as mensagens perdidas é utilizado o *log* de mensagens no transmissor. Desta forma, quando um processo salva um novo ponto de recuperação, estará salvando o *log* das mensagens transmitidas cujas respostas não chegaram ao transmissor.

Quando todos os processos tiverem recebido as mensagens **REQ**, ter-se-á registrada uma nova linha de recuperação. Apesar disso, o coordenador e os outros processos não estão informados de que todos os outros já estabeleceram seus pontos locais: uns não sabem do término da operação nos outros. Mesmo assim, caso ocor-

resse uma falha, estes pontos de recuperação poderiam ser utilizados para o retorno, uma vez que formam um conjunto consistente.

Os processos só responderão com mensagens de **reconhecimento à solicitação** de estabelecimento de um novo ponto de recuperação – **AREQ** – após receberem uma solicitação.

Entre o recebimento da solicitação e a resposta, os processos estabelecem novos pontos de recuperação. Desta forma, quanto todos tiverem respondido, cada processo terá registrado na memória dois pontos de recuperação: o novo e o anterior.

Note-se que todos os processos devem responder à solicitação do coordenador. Se isso não ocorrer, o coordenador não pode dar continuidade aos procedimentos de estabelecimento do ponto de recuperação e, portanto, não poderá encerrar sua tarefa. Isto inclui a impossibilidade de passar a coordenação para outro processo do sistema, caso tenha sido implementada uma rotação de coordenadores.

No caso de ausência de resposta de algum processo, o coordenador deve sinalizar a falha e iniciar o procedimento de recuperação. Adicionalmente, se o coordenador não encerrar o estabelecimento do ponto de recuperação nem iniciar um novo procedimento de recuperação em tempo hábil, os outros processos do sistema têm condições de detectar o fato e iniciar a recuperação (pois, provavelmente, o coordenador falhou).

Após ter recebido todas as respostas dos outros processos, o coordenador envia a **confirmação de estabelecimento** – mensagens **CMT**. Como já explicado, em caso de falha, o sistema poderá retornar para este último ponto. Entretanto, após receber todas as respostas, deve-se efetuar algumas tarefas auxiliares. Para isso, é necessário que o coordenador informe a todos os processos do sistema que a linha de recuperação foi estabelecida com sucesso.

Note-se que, do ponto de vista do estabelecimento de uma linha de recuperação, **a etapa de confirmação não é necessária**. O protocolo proposto não utiliza um protocolo bloqueante de duas fases (*two-phase commit*), diferentemente do que fazem, por exemplo, Koo e Toueg [KOO 87].

Os processos que formam o sistema, quando recebem a confirmação de estabelecimento do ponto de recuperação, podem proceder à troca do coordenador e à coleta de lixo, ou seja, como os pontos de recuperação mais recentes formam uma linha de recuperação, os pontos anteriores podem ser descartados.

Após terem efetuado a atualização de seus pontos de recuperação, todos os processos **respondem ao coordenador**, usando uma mensagem **ACMT**. Com isso, informam que procederam à atualização dos pontos de recuperação e a coleta de lixo.

Esta última etapa visa garantir que todos os processos tenham completado as atividades essenciais e auxiliares de estabelecimento do ponto de recuperação. Desta forma, o coordenador poderá encerrar o estabelecimento da linha de recuperação e proceder às atividades como a rotação de coordenadores ou o início de um novo ponto de recuperação. Esta etapa foi detectada como necessária durante a escrita das provas de correção do algoritmo. Entretanto, não foi varrido o espaço de soluções em busca de outras condições que possibilitassem detectar, de forma mais eficiente, o encerramento das atividades de estabelecimento de um ponto de recuperação.

Apesar de não ter sido provado, parece ser possível determinar condições que possibilitem operar, de forma consistente, sem a última etapa. Entretanto, por uma

questão de coerência com as provas desenvolvidas, esta afirmativa não será defendida neste trabalho, deixando esta possibilidade para um trabalho futuro.

4.2.2 Consistência com mensagens de reconhecimento

Na seção 1.3, foram discutidos os aspectos gerais da consistência de um conjunto de pontos de recuperação e sua relação com as mensagens trocadas entre os processos. As considerações usadas naquela seção não discriminavam tipos de mensagens. Assim, o critério de consistência era aplicado a todas as mensagens.

Entretanto, conforme descrito, o algoritmo necessita que todas as mensagens sejam respondidas. Desta forma, existem mensagens que carregam informação (mensagens da aplicação) enquanto que outras são as respostas (mensagens de reconhecimento). Além disso, a cada mensagem da aplicação, existe uma mensagem de reconhecimento associada. Assim, necessita-se reavaliar os critérios de consistência que vinham sendo usados, levando-se em consideração que a transmissão dos dados só é efetivada quando a mensagem de resposta é recebida.

A classificação das mensagens como órfãs e perdidas, suas causas e efeitos, continua válida. Entretanto, deve-se aprofundar o estudo da consistência, de maneira a considerar a associação das duas mensagens necessárias à transmissão das informações.

Para analisar a questão da consistência, será feita a análise da **vida de uma mensagem**. Nesta análise, as mensagens da aplicação são representadas por **APP** e as mensagens de resposta são representadas por **ACK**. Com esta análise, busca-se identificar as características que mensagens consistentes devem apresentar.

Associados a cada mensagem existem dois eventos: a transmissão e a recepção da mesma. Desta forma, a cada transmissão de informação (composta por uma mensagem **APP** e uma **ACK**), estarão associados quatro eventos:

- **TAPP**: transmissão da **APP**;
- **RAPP**: recepção da **APP**;
- **TACK**: transmissão da **ACK**;
- **RACK**: recepção da **ACK**.

Como o interesse presente situa-se em discutir a consistência entre pontos de recuperação e sabendo que a causa das inconsistências está vinculada às mensagens, serão verificados todos os cenários envolvendo uma transmissão de informação e dois pontos de recuperação. O ponto de recuperação no processo transmissor da mensagem (onde ocorrem os eventos **TAPP** e **RACK**) será indicado por C_1 e o ponto de recuperação no processo receptor da mensagem (onde ocorrem os eventos **RAPP** e **TACK**) por C_2 .

As mensagens serão caracterizadas por alguns parâmetros que permitirão a sua identificação. Estes parâmetros são:

- o tipo da mensagem: se é **APP** ou **ACK**;
- o índice do último ponto de recuperação estabelecido pelo processo transmissor, quando a mensagem foi transmitida pela aplicação;

- o índice do último ponto de recuperação estabelecido pelo processo receptor, quando a mensagem foi recebida pela aplicação;
- identificação de uma mensagem associada.

Se uma mensagem for chamada de m, então os parâmetros listados serão representados por $m.d$, $m.itx$, $m.irx$ e $m.msg$, respectivamente. Estes parâmetros são definidos da seguinte forma (uma definição mais rigorosa será apresentada na seção 6.2.1):

- $m.d$: identifica o tipo da mensagem. Pode ser uma mensagem da aplicação (**APP**) ou uma mensagem de resposta (**ACK**);
- $m.itx$: representa o instante de tempo em que a mensagem foi transmitida, em relação ao ponto de recuperação estabelecido no processo transmissor. Para isso, é utilizado o índice do último ponto de recuperação estabelecido no transmissor, quando a mensagem foi transmitida;
- $m.irx$: representa o instante de tempo em que a mensagem foi recebida, em relação ao ponto de recuperação estabelecido no processo receptor. Para isso, é utilizado o índice do último ponto de recuperação estabelecido no receptor, quando a mensagem foi recebida;
- $m.msg$: identifica a mensagem da aplicação que está sendo respondida pela mensagem m. Este parâmetro só tem sentido quando a mensagem m for de resposta, ou seja, $m.d =$**ACK**.

Quando uma mensagem é transmitida, o intervalo de recepção não é conhecido, uma vez que a mensagem ainda não foi recebida. Desta forma, o intervalo de recepção receberá o valor NN, indicando a sua situação de não recebida. Note-se que o valor de NN deverá ser, sempre, maior ou igual do que qualquer índice de ponto de recuperação em qualquer processo. Isso implica que a mensagem chegará ao destino no período atual ou em um período futuro.

Antes de discutir a seqüência de eventos associados às mensagens, serão explicadas as premissas usadas implicitamente na discussão. A primeira é que o processo transmissor da mensagem da aplicação é o mesmo receptor da mensagem de resposta, assim como o receptor da mensagem da aplicação é o mesmo transmissor da mensagem de resposta.

A segunda premissa é que só faz sentido receber uma resposta após ter sido transmitida a mensagem da aplicação correspondente.

Para indicar o relacionamento entre o instante de estabelecimento dos pontos de recuperação e os instantes de transmissão e recepção das mensagens, será utilizada a seguinte notação:

$Antes(x, C_i)$, para indicar que o índice x é menor que aquele do ponto de recuperação C_i;

$Apos(x, C_i)$, para indicar que o índice x é maior ou igual que aquele do ponto de recuperação C_i.

Caso o índice x usado nas duas fórmulas anteriores representar a transmissão e a recepção de uma mensagem (indicados por $m.itx$ e $m.irx$, respectivamente), estará caracterizado o relacionamento de antecedência entre os eventos da mensagem e o estabelecimento do ponto de recuperação C_i.

Assim, por exemplo, $Antes(m.itx, C_1)$ indica que a transmissão da mensagem m ocorreu antes do estabelecimento do ponto de recuperação C_1. De forma análoga, $Apos(m.irx, C_2)$ indica que a recepção da mensagem m ocorreu após o estabelecimento do ponto de recuperação C_2.

O **primeiro evento** da troca de informações é o **TAPP**. Com este evento, o processo transmissor envia uma mensagem do tipo **APP**. Como a análise baseia-se na relação entre mensagens e pontos de recuperação, pode-se considerar que este evento ocorra antes ou depois de C_1. Usando a notação *Antes* e *Apos*, são possíveis os seguintes cenários, representados na figura 4.3:

1. $(m.d = \mathbf{APP}) \wedge Antes(m.itx, C_1) \wedge (m.irx = NN) \wedge (m \in C_1)$

2. $(m.d = \mathbf{APP}) \wedge Apos(m.itx, C_1) \wedge (m.irx = NN)$

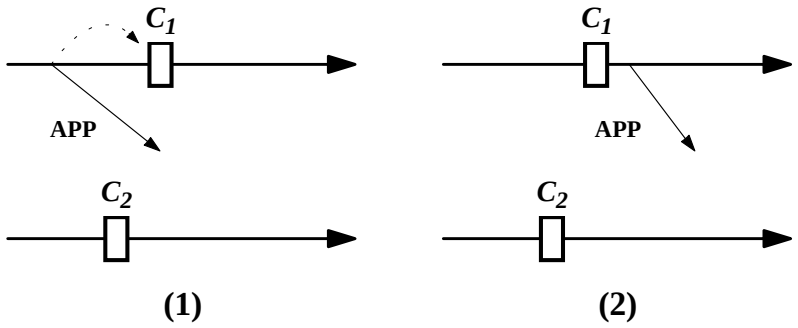

FIGURA 4.3 – Cenários de transmissão de uma mensagem da aplicação

Em ambos os cenários está representada uma mensagem do tipo **APP**. No primeiro cenário ((1) na figura 4.3), considera-se uma mensagem transmitida antes do estabelecimento de C_1, enquanto que no segundo ((2) na figura 4.3), considera-se uma mensagem transmitida após C_1. Como a mensagem não foi recebida, o intervalo de recepção deverá ser NN. Embora tenha sido representado hipoteticamente o estabelecimento do ponto de recuperação C_2 no processo receptor, é irrelevante o que ocorre neste processo para o cenário delineado, do ponto de vista do transmissor.

Em caso de falha e retorno para os dois pontos de recuperação da figura, a mensagem poderá estar armazenada em C_1, de maneira a poder ser repetida, no cenário (1) (o armazenamento está representado, na figura, através de uma seta tracejada). Entretanto, a sua repetição não será possível no cenário (2), uma vez que a mensagem foi criada (transmitida) após o estabelecimento de C_1.

O evento seguinte ao da transmissão da mensagem é o de recepção. Este é o **segundo evento** na vida da mensagem. Como uma mensagem pode ter dois cenários de transmissão e como a recepção pode ocorrer também de duas formas diferentes, no que diz respeito a sua posição em relação ao estabelecimento de C_2, tem-se quatro cenários de recepção.

No caso da transmissão ocorrer antes do estabelecimento do ponto de recuperação no transmissor (cenário (1) na figura 4.3), poderá ocorrer da mensagem ser recebida segundo duas possibilidades, representadas na figura 4.4:

1.1. $(m.d = \mathbf{APP}) \wedge Antes(m.itx, C_1) \wedge Antes(m.irx, C_2) \wedge (m \in C_1)$;

1.2. $(m.d = \mathbf{APP}) \wedge Antes(m.itx, C_1) \wedge Apos(m.irx, C_2) \wedge (m \in C_1)$.

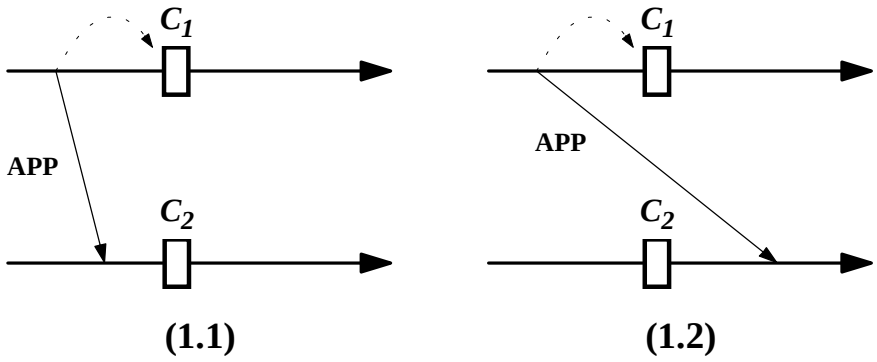

FIGURA 4.4 – Cenários 1.1 e 1.2 de recepção de uma mensagem da aplicação

Os dois cenários da figura 4.4 representam mensagens da aplicação. Desta forma, o tipo das mensagens é **APP**. No caso 1.1, a recepção ocorreu antes de C_2, enquanto que no caso 1.2 ocorreu após C_2.

Se a transmissão da mensagem da aplicação ocorrer após o estabelecimento do ponto de recuperação do transmissor (cenário (2) na figura 4.3), também poderá ocorrer da mensagem ser recebida segundo duas outras possibilidades, representadas na figura 4.5:

2.1. $(m.d = \mathbf{APP}) \wedge Apos(m.itx, C_1) \wedge Antes(m.irx, C_2)$;

2.2. $(m.d = \mathbf{APP}) \wedge Apos(m.itx, C_1) \wedge Apos(m.irx, C_2)$.

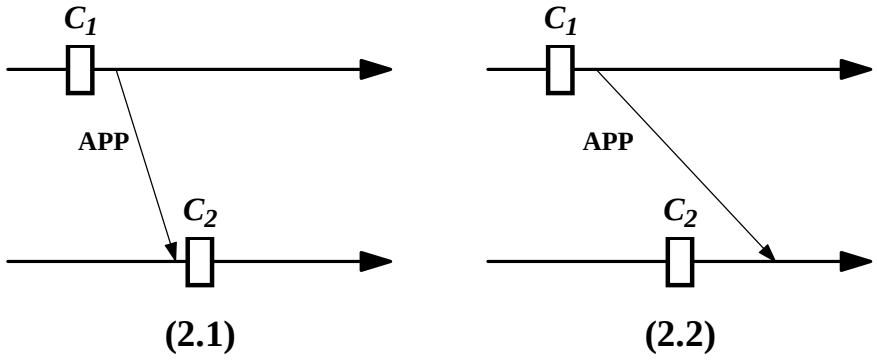

FIGURA 4.5 – Cenários 2.1 e 2.2 de recepção de uma mensagem da aplicação

Os dois cenários da figura 4.5 representam possibilidades relacionadas com as mensagens da aplicação. Desta forma, o tipo da mensagem será **APP**. Da mesma

forma que nos cenários 1.1 e 1.2 anteriores, no cenário 2.1 a recepção ocorreu antes de C_2 enquanto que no caso 2.2 ocorreu após C_2.

No caso 2.2, a mensagem da aplicação não pode ser salva em C_1, uma vez que foi gerada após aquele ponto de recuperação.

A mensagem da aplicação do cenário 2.1 torna os pontos de recuperação inconsistentes, uma vez que a mensagem não poderá estar registrada em C_1 (ponto de recuperação do transmissor) mas estará registrada em C_2 (ponto de recuperação do receptor). Esta é uma mensagem potencialmente órfã, a qual não pode ser aceita em estados consistentes. Desta forma, este caso e os seus derivados não podem ser aceitos como consistentes.

Até aqui, ou seja, no que diz respeito aos eventos associados com uma mensagem da aplicação, foram identificados três cenários consistentes: os cenários 1.1 e 1.2, da figura 4.4, e o cenário 2.2, da figura 4.5.

Na continuação dos eventos que formam uma mensagem, o processo receptor envia uma resposta à mensagem da aplicação recebida. Este é o **terceiro evento** na vida da mensagem.

Uma vez que foram identificados três cenários consistentes e que a transmissão da resposta pode ocorrer antes ou depois do estabelecimento de C_2, tem-se seis cenários possíveis, que serão analisados dois a dois.

O primeiro cenário analisado será o 1.1 da figura 4.4. As duas possibilidades de transmissão da resposta estão representadas na figura 4.6:

1.1.1. $(m.d = \mathbf{ACK}) \wedge Antes(m.itx, C_2) \wedge (m.irx = NN) \wedge (m.msg.d = \mathbf{APP}) \wedge Antes(m.msg.itx, C_1) \wedge Antes(m.msg.irx, C_2) \wedge (m.msg \in C_1)$;

1.1.2. $(m.d = \mathbf{ACK}) \wedge Apos(m.itx, C_2) \wedge (m.irx = NN) \wedge (m.msg.d = \mathbf{APP}) \wedge Antes(m.msg.itx, C_1) \wedge Antes(m.msg.irx, C_2) \wedge (m.msg \in C_1)$.

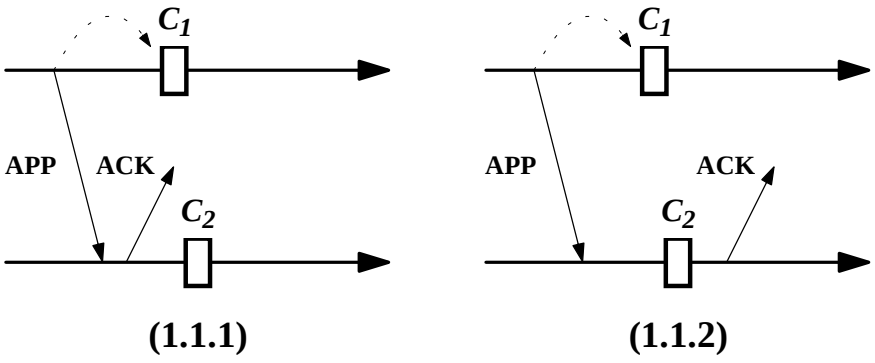

FIGURA 4.6 – Cenários 1.1.1 e 1.1.2 de transmissão de uma mensagem de resposta

Com o evento da transmissão da resposta, os cenários passam a representar duas mensagens componentes: a mensagem da aplicação (representada por **APP** nas figuras) e sua mensagem de resposta associada (representada por **ACK** nas figuras). Entretanto, as expressões passam a referenciar a mensagem de resposta com a notação m^1 enquanto que a mensagem da aplicação passa a ser referenciada

[1]nos cenários anteriores, a mensagem m era de aplicação, ou seja, $m.d = \mathbf{APP}$; nestes cenários e nos próximos, a mensagem m passará a ser de resposta, ou seja, $m.d =$**ACK**.

de forma indireta, através da propriedade $m.msg$. Ou seja, a mensagem de resposta carrega uma informação que permite identificar a mensagem da aplicação a qual corresponde.

Os dois cenários apresentados na figura 4.6 são consistentes. Não existem mensagens órfãs e a mensagem da aplicação está salva no ponto de recuperação do transmissor, o que possibilita sua repetição, caso venha a se tornar perdida.

O próximo cenário que será analisado corresponde ao 1.2 da figura 4.4. De forma semelhante à análise do cenário anterior, existem duas possibilidades de transmissão da resposta, que estão representadas na figura 4.7:

1.2.1. $(m.d = \mathbf{ACK}) \wedge Antes(m.itx, C_2) \wedge (m.irx = NN) \wedge (m.msg.d = \mathbf{APP}) \wedge Antes(m.msg.itx, C_1) \wedge Apos(m.msg.irx, C_2) \wedge (m.msg \in C_1)$;

1.2.2. $(m.d = \mathbf{ACK}) \wedge Apos(m.itx, C_2) \wedge (m.irx = NN) \wedge (m.msg.d = \mathbf{APP}) \wedge Antes(m.msg.itx, C_1) \wedge Apos(m.msg.irx, C_2)) \wedge (m.msg \in C_1)$.

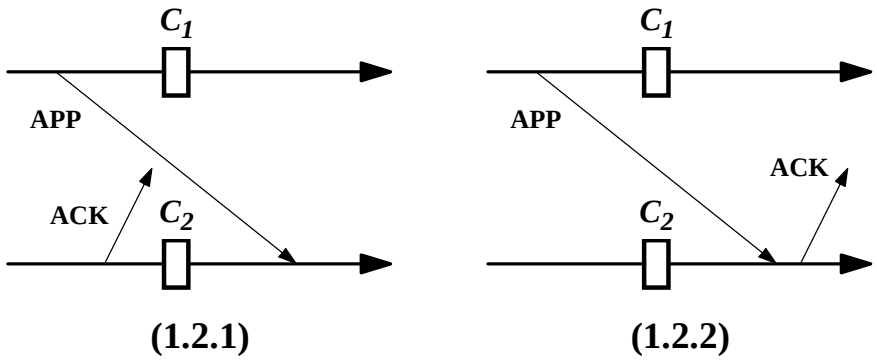

FIGURA 4.7 – Cenários 1.2.1 e 1.2.2 de transmissão de uma mensagem de resposta

Dos dois cenários apresentados na figura 4.7, apenas o segundo (cenário 1.2.2) é consistente, uma vez que, no primeiro (cenário 1.2.1), a transmissão da mensagem de resposta ocorre antes da recepção da mensagem da aplicação, o que é absurdo.

O último dos três cenários consistentes corresponde ao 2.2 da figura 4.5. Novamente, existem duas possibilidades de transmissão da resposta, representadas na figura 4.8:

2.2.1. $(m.d = \mathbf{ACK}) \wedge Antes(m.itx, C_2) \wedge (m.irx = NN) \wedge (m.msg.d = \mathbf{APP}) \wedge Apos(m.msg.itx, C_1) \wedge Apos(m.msg.irx, C_2)$;

2.2.2. $(m.d = \mathbf{ACK}) \wedge Apos(m.itx, C_2) \wedge (m.irx = NN) \wedge (m.msg.d = \mathbf{APP}) \wedge Apos(m.msg.itx, C_1) \wedge Apos(m.msg.irx, C_2)$.

Da mesma forma que anteriormente, o cenário 2.2.1 da figura 4.8 não é válido, um vez que a resposta foi transmitida antes da recepção da mensagem da aplicação.

A análise da vida das mensagens até o evento da transmissão da resposta permitiu identificar quatro cenários consistentes: 1.1.1 e 1.1.2 da figura 4.6, 1.2.2 da figura 4.7 e 2.2.2 da figura 4.8.

Na seqüência, o **quarto e último evento** na vida de uma mensagem é a recepção da resposta, por parte do processo transmissor da mensagem da aplicação.

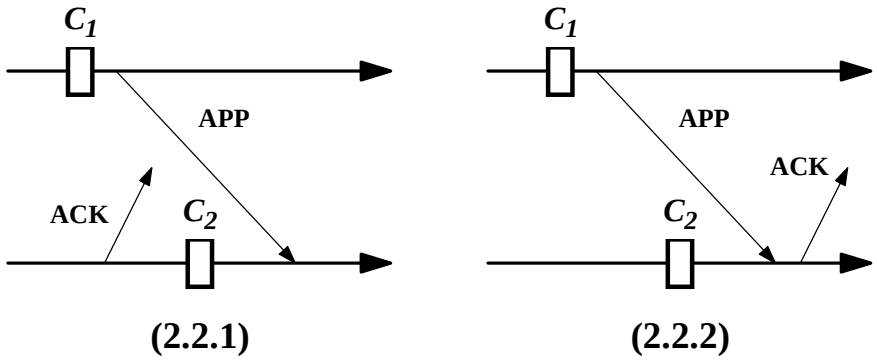

FIGURA 4.8 – Cenários 2.2.1 e 2.2.2 de transmissão de uma mensagem de resposta

Novamente, é possível que uma mensagem de resposta alcance o seu destino (processo transmissor da mensagem da aplicação associada) antes ou depois do estabelecimento de C_1. Assim, considerando-se os quatro cenários consistentes identificados até aqui, são oito os cenários a serem analisados.

A combinação do cenário 1.1.1 da figura 4.6 com as duas possibilidade de recepção da resposta, fornece os cenários representados na figura 4.9.

1.1.1.1. $(m.d = \mathbf{ACK}) \wedge Antes(m.itx, C_2) \wedge Antes(m.irx, C_1) \wedge (m.msg.d = \mathbf{APP}) \wedge Antes(m.msg.itx, C_1) \wedge Antes(m.msg.irx, C_2)$;

1.1.1.2. $(m.d = \mathbf{ACK}) \wedge Antes(m.itx, C_2) \wedge Apos(m.irx, C_1) \wedge (m.msg.d = \mathbf{APP}) \wedge Antes(m.msg.itx, C_1) \wedge Antes(m.msg.irx, C_2) \wedge (m.msg \in C_1)$.

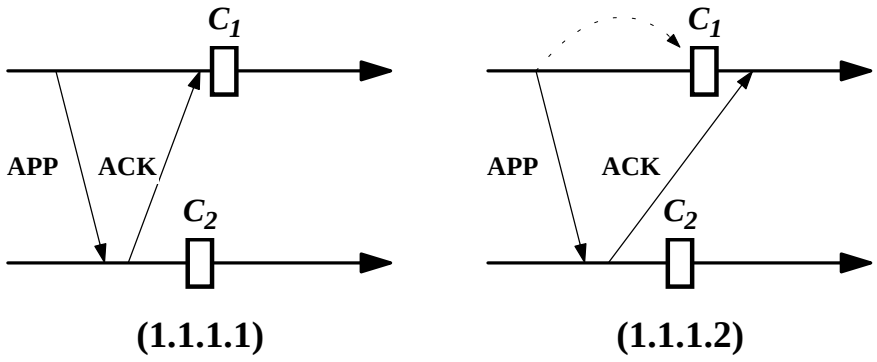

FIGURA 4.9 – Cenários 1.1.1.1 e 1.1.1.2 de recepção de uma mensagem de resposta

No cenário 1.1.1.1 da figura 4.9, todos os eventos da mensagem ocorrem à esquerda dos pontos de recuperação. Assim, esta cenário é consistente e não é necessário que a mensagem da aplicação esteja armazenada em C_1. No caso do cenário 1.1.1.2 da mesma figura, a mensagem de reconhecimento chega ao transmissor após o estabelecimento de C_1. Desta forma, para que o cenário seja consistente, a mensagem da aplicação deverá estar armazenada em C_1.

O cenário 1.1.2 da figura 4.6, quando consideradas as duas possibilidades de recepção da resposta, formam cenários que estão representados na figura 4.10.

1.1.2.1. $(m.d = \mathbf{ACK}) \wedge Apos(m.itx, C_2) \wedge Antes(m.irx, C_1) \wedge (m.msg.d = \mathbf{APP}) \wedge Antes(m.msg.itx, C_1) \wedge Antes(m.msg.irx, C_2) \wedge (m.msg \in C_1)$;

1.1.2.2. $(m.d = \mathbf{ACK}) \wedge Apos(m.itx, C_2) \wedge Apos(m.irx, C_1) \wedge (m.msg.d = \mathbf{APP}) \wedge Antes(m.msg.itx, C_1) \wedge Antes(m.msg.irx, C_2) \wedge (m.msg \in C_1)$.

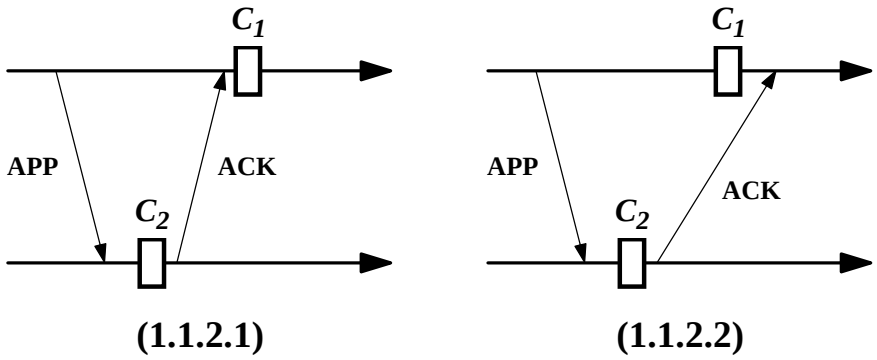

FIGURA 4.10 – Cenários 1.1.2.1 e 1.1.2.2 de recepção de uma mensagem de resposta

No cenário 1.1.2.1 da figura 4.10, a mensagem de resposta é potencialmente órfã. Desta forma, é inconsistente. Por outro lado, o cenário 1.1.2.2 é consistente pois nenhuma mensagem é potencialmente órfã e a mensagem da aplicação está salva no ponto de recuperação do transmissor, possibilitando a repetição, em caso de falha e retorno.

Quando consideradas as possibilidades de recepção da mensagem de resposta e o cenário 1.2.2 da figura 4.7, tem-se os cenários representados na figura 4.11.

1.2.2.1. $(m.d = \mathbf{ACK}) \wedge Apos(m.itx, C_2) \wedge Antes(m.irx, C_1) \wedge (m.msg.d = \mathbf{APP}) \wedge Antes(m.msg.itx, C_1) \wedge Apos(m.msg.irx, C_2) \wedge (m.msg \in C_1)$;

1.2.2.2. $(m.d = \mathbf{ACK}) \wedge Apos(m.itx, C_2) \wedge Apos(m.irx, C_1) \wedge (m.msg.d = \mathbf{APP}) \wedge Antes(m.msg.itx, C_1) \wedge Apos(m.msg.irx, C_2) \wedge (m.msg \in C_1)$.

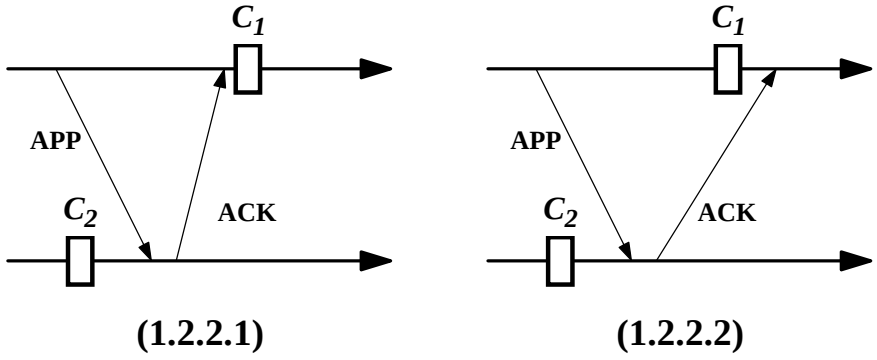

FIGURA 4.11 – Cenários 1.2.2.1 e 1.2.2.2 de recepção de uma mensagem de resposta

No cenário 1.2.2.1 da figura 4.11 a mensagem de resposta é potencialmente órfã, o que corresponde a um estado inconsistente.

Por outro lado, o cenário 1.2.2.2 é consistente, uma vez que a mensagem da aplicação está armazenada no ponto de recuperação do transmissor. Apesar de potencialmente perdida, esta mensagem poderá ser repetida, caso necessário.

Finalmente, o cenário 2.2.2 da figura 4.8 corresponde a dois outros cenários, representados na figura 4.12.

2.2.2.1. $(m.d = \mathbf{ACK}) \wedge Apos(m.itx, C_2) \wedge Antes(m.irx, C_1) \wedge (m.msg.d = \mathbf{APP}) \wedge Apos(m.msg.itx, C_1) \wedge Apos(m.msg.irx, C_2)$;

2.2.2.2. $(m.d = \mathbf{ACK}) \wedge Apos(m.itx, C_2) \wedge Apos(m.irx, C_1) \wedge (m.msg.d = \mathbf{APP}) \wedge Apos(m.msg.itx, C_1) \wedge Apos(m.msg.irx, C_2)$.

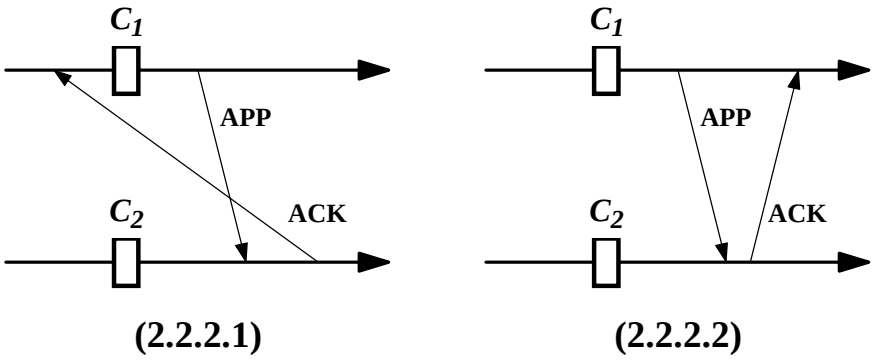

FIGURA 4.12 – Cenários 2.2.2.1 e 2.2.2.2 de recepção de uma mensagem de resposta

Na figura 4.12, o cenário 2.2.2.1 apresenta uma mensagem de resposta que retorna no tempo, o que é absurdo. Mesmo que não fosse considerado este fato, esta mensagem de resposta seria órfã, o que configura um estado inconsistente.

Por outro lado, as mensagens do cenário 2.2.2.2 desenvolvem-se à direita dos pontos de recuperação. Sendo assim, o retorno para estes pontos de recuperação não tornará as mensagens da aplicação e de resposta órfãs nem perdidas. Portanto, este é um cenário consistente.

Ao final da análise da vida de uma mensagem, composta por uma mensagem da aplicação e sua resposta, foram obtidos cinco cenários consistentes sendo que os outros quatro cenários ou correspondiam a um estado inconsistente ou configuravam situações absurdas.

O resultado alcançado foi obtido através da aplicação de critérios bastante conservadores: qualquer mensagem registrada como recebida mas não transmitida configura um estado inconsistente. Entretanto, a análise de consistência merece ser aprofundada, quando isso acontece com uma mensagem de resposta. Uma análise deste tipo poderia levar a conclusões que ampliariam os cenários considerados consistentes. Estes novos cenários poderiam propiciar um maior entendimento da consistência, quando usando mensagens de resposta.

Finalmente, os cenários identificados como consistentes serão formalizados, de maneira a serem usados nas provas de correção do algoritmo.

4.2.3 Retorno

Quando o detector de defeitos informar a recuperação de um processo que sofreu falha, o sistema pode iniciar o procedimento de retorno.

Este procedimento não foi formalizado, uma vez que a garantia de existência de uma linha de recuperação é suficiente para possibilitar o retorno correto do sistema. Entretanto, a especificação e prova de correção do mecanismo de retorno é trabalho interessante que poderá ser realizado no futuro. Junto com este trabalho, também pode-se incluir a modelagem formal das falhas e a interação entre o três: estabelecimento das linhas de recuperação, modelagem de falhas e o retorno.

Assim sendo, uma vez detectada a recuperação de um processo, todos os outros processos deverão ser informados do fato. Na seqüência, o sistema deve proceder à procura por uma linha de recuperação. Para isso, cada processo deverá informar a todos os outros os índices dos pontos de recuperação que estão armazenados em sua memória estável.

Caso **o sistema não esteja estabelecendo um ponto de recuperação**, os processos possuirão apenas um ponto válido (ou seja, dois pontos iguais). Por outro lado, se o algoritmo estiver estabelecendo um novo ponto, pode ocorrer que existam dois pontos distintos, os quais serão inequivocamente identificados pelos seus índices. No último caso, pode ocorrer que exista uma segunda linha de recuperação, ou seja, todos os processos possuem este segundo ponto de recuperação, ou não (alguns processos não estabeleceram este segundo ponto de recuperação).

De qualquer forma, existirá sempre uma linha de recuperação, conforme será provado no capítulo 6. Tudo o que o sistema tem que fazer é detectar a mais adequada.

Como todos os processos enviam os índices dos pontos de recuperação que têm disponível e sabendo que, pelo menos, um deles vai pertencer a uma linha de recuperação, basta identificar o maior índice disponível em todos os processos.

Para isso, pode-se recorrer a dois enfoques possíveis: um centralizado e outro distribuído.

No enfoque centralizado, o processo que detectou a falha solicita a todos os outros processos que informem quais os pontos de recuperação que têm registrados. De posse desta informação, este coordenador escolhe aquele ponto de recuperação que esteja presente em todos os processos. Então, informa a sua decisão a todos os outros. A centralização, apesar de simples, é uma desvantagem deste mecanismo, pois o coordenador torna-se um ponto único de falhas (*single point of failure*).

Outra opção é o enfoque distribuído, onde todos os processos informam a todos os outros quais são os pontos de recuperação que estão registrados em sua memória. Com este conjunto de informações, cada processo decide para onde retornar: para aquele ponto de recuperação que esteja presente em todos os processos. Como todos possuem a mesma informação, todos retornarão para o mesmo ponto. A desvantagem é a necessidade de troca de informações entre todos os processos, o que apresenta um custo de comunicação maior do que aquele do enfoque concentrado.

Após encerrada a recarga do estado dos processos, o índice dos pontos de recuperação deverá ser incrementado, em relação ao maior índice de ponto de recuperação já estabelecido, mesmo que incompleto. Assim, o novo índice de ponto de recuperação será igual ao máximo dentre os índices de recuperação recebidos mais um. Este mecanismo será utilizado para evitar o problema dos *livelocks*.

Uma vez efetuado o retorno do estado dos processos, as mensagens registradas

como em trânsito nos canais poderão ser repetidas. Para isso, estas serão colocadas nos canais, carregando o novo índice de ponto de recuperação. Como estas mensagens estão sendo transmitidas, serão colocadas na memória que representa o estado dos canais, à espera de respostas para sua remoção.

Adicionalmente, as mensagens recebidas cujo índice for menor do que o novo índice, serão descartadas. Com isso, consegue-se limpar dos canais as mensagens transmitidas anteriormente à detecção do erro, sem ter que bloquear a comunicação para fazê-lo.

A recarga do estado dos processos tem que ser feita com o bloqueio da aplicação. Não existe outra forma de fazê-lo. Entretanto, a repetição das mensagens pode ser feita com a aplicação já rodando. Este procedimento não deve acarretar problemas, na medida que os canais não têm a restrição de serem FIFO, possibilitando a recepção de mensagens em ordem diferente daquela da transmissão.

4.2.4 Etapas do estabelecimento dos pontos de recuperação

Para analisar o comportamento do sistema durante o estabelecimento de um novo ponto de recuperação, é necessário separar esta atividade em duas: antes e após ter sido estabelecida uma linha de recuperação. Aqui, entende-se o estabelecimento da linha de recuperação como o instante em que todos os processos salvaram seus pontos de recuperação local em memória estável ((1) na figura 4.13), mesmo que a informação ainda não tenha sido propagada para todos os envolvidos.

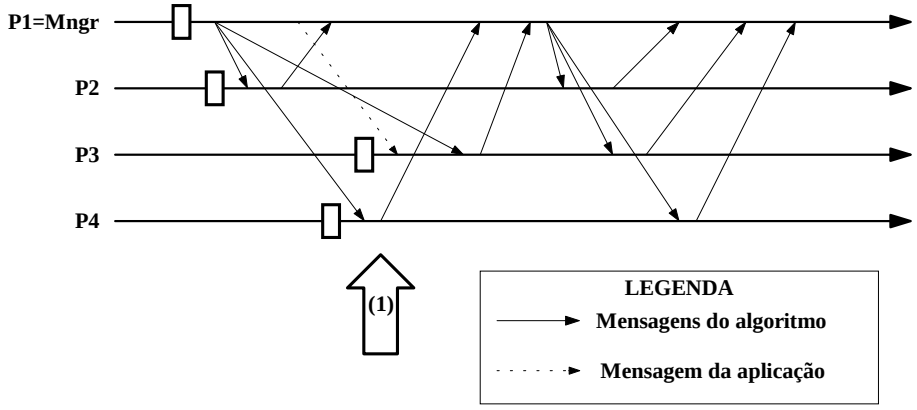

FIGURA 4.13 – Divisão do período de estabelecimento de um ponto de recuperação

Levando-se em consideração o exposto, a primeira etapa inicia quando o coordenador (**Mngr**) estabelece seu novo ponto de recuperação e termina quando todos os processos tiverem estabelecido seus pontos de recuperação locais. A segunda etapa inicia logo após o término da primeira e encerra quando o coordenador receber todas as mensagens de resposta à confirmação: mensagens **ACMT**.

Como a análise tem por objetivo identificar cenários de comportamento do sistema onde as falhas ocorrem durante o estabelecimento dos pontos de recuperação, esta será dividida em duas: quando a falha ocorrer na primeira etapa e quando ocorrer na segunda etapa, conforme divisão descrita anteriormente.

Se uma falha ocorrer durante a segunda etapa do estabelecimento de uma nova linha de recuperação, os processos trocarão informações de maneira a escolherem os

pontos de retorno mais adequados.

Por outro lado, se a falha ocorrer antes que todos os processos tenham efetuado a coleta de lixo (figura 4.14), então alguns deles terão apenas um ponto de recuperação (aqueles que já fizeram a coleta de lixo), enquanto que outros terão dois pontos de recuperação: o último e o anterior. Como o mecanismo de retorno prevê que os processos retornem para os pontos de recuperação de mesmo índice, presentes em todos os processos, o sistema retornará para o último ponto de recuperação que forma uma linha de recuperação.

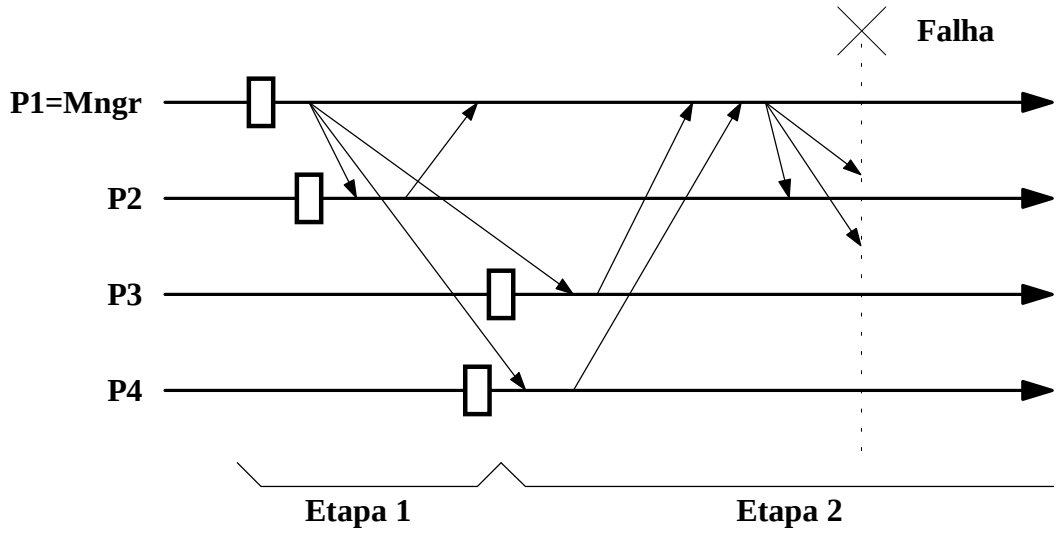

FIGURA 4.14 – Falha ocorrida na segunda etapa: antes do término da coleta de lixo

Ainda na segunda etapa, se a falha ocorrer após a coleta de lixo ter sido completada (figura 4.15), o mecanismo de retorno fará com que os processos retornem para os únicos pontos disponíveis, ou seja, aqueles que foram estabelecidos por último e que formam uma linha de recuperação.

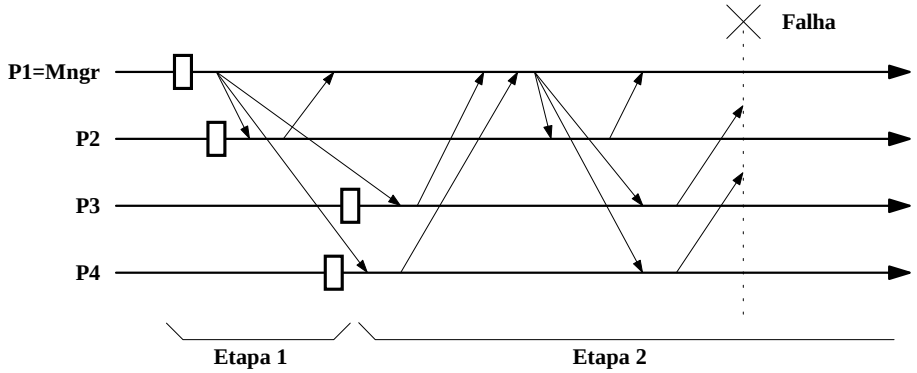

FIGURA 4.15 – Falha ocorrida na segunda etapa: após o término da coleta de lixo

Caso a falha tenha ocorrido durante a primeira etapa do estabelecimento de uma nova linha de recuperação, conforme aparece na figura 4.16, não terá sido

completado o último ponto de recuperação. O mecanismo de retorno, por sua vez, fará com que o sistema retorne para a linha de recuperação anterior, uma vez que a nova não foi completada. Desta forma, garante-se que o sistema será colocado em um estado consistente.

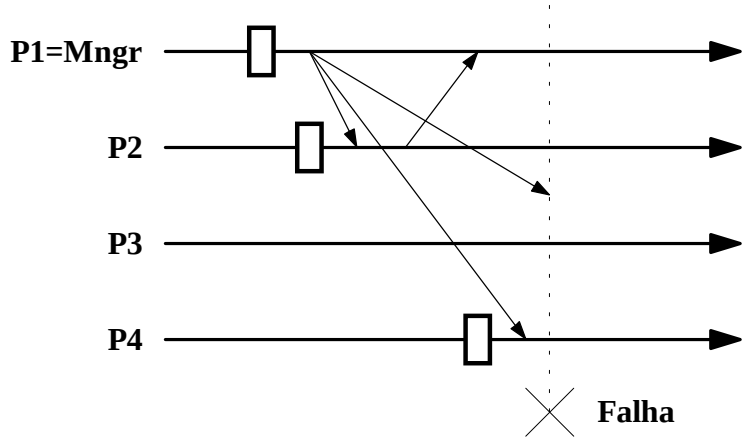

FIGURA 4.16 – Falha ocorrida na primeira etapa

Pode-se verificar que, pela análise anterior, uma falha ocorrida durante os procedimentos de estabelecimento de uma nova linha de recuperação não leva o sistema para um estado inconsistente.

4.2.5 Mecanismo para evitar *livelocks*

O mecanismo aqui proposto é semelhante ao de Silva e Silva [SIL 92]. Cada mensagem terá anexado um número que permitirá ao processo receptor identificar quando ocorreu a sua transmissão.

Entretanto, de forma diferente de Silva e Silva, será utilizado o mesmo número que identifica o intervalo do ponto de recuperação em que a mensagem foi transmitida. Para isso, quando ocorrer um retorno, o índice dos pontos de recuperação será incrementado, em relação ao maior índice encontrado (pode ocorrer dos processos possuírem índices diferentes de pontos de recuperação), dentre aqueles trocados pelos processos. Este novo índice será registrado em cada processo e uma mensagem só será entregue quando o seu índice for maior ou igual que aquele registrado no processo receptor.

Tendo em vista o mecanismo proposto e considerando mensagens que estejam no canal durante a recuperação, pode-se propor dois cenários: a mensagem será transmitida antes ou depois do ponto de recuperação usado no retorno. Em ambos os casos, a mensagem será descartada, quando alcançar o receptor. Entretanto, aquela mensagem transmitida antes do ponto de recuperação estará armazenada em memória estável e será repetida: agora com um novo índice de transmissão, o que garantirá a sua entrega.

O mecanismo proposto visa limpar dos canais aquelas mensagens que estavam em trânsito durante a recuperação. Ao serem recebidas, estas mensagens serão descartadas, não sendo entregues para a aplicação.

Com o mecanismo proposto, as mensagens não necessitam ter anexada mais uma informação de controle. Apesar desta informação adicional representar um

acréscimo médio pequeno em termos do tamanho total das mensagens, poderia implicar em uma queda significativa no desempenho, devido ao seu caráter multiplicativo, uma vez que o aumento incide sobre todas as mensagens da aplicação.

5 Comparação com outros algoritmos

Dentre os artigos sobre algoritmos de recuperação ou estabelecimento de pontos de recuperação, três merecem destaque devido as suas semelhanças com o proposto nesta tese. São os artigos de Briatico, Ciuffoletti e Simoncini [BRI 84], de Silva e Silva [SIL 92] e de Gendelman, Bic e Dillencourt [GEN 99].

O algoritmo de Briatico *et al.* pode ser classificado na categoria dos controlados pela comunicação. Este algoritmo utiliza os mecanismos forçado e independente para o estabelecimento dos pontos de recuperação. O critério usado para forçar o estabelecimento dos pontos de recuperação utiliza-se de informações anexadas às mensagens da aplicação, de forma semelhante a proposta neste trabalho.

No caso do algoritmo de Silva e Silva, pode-se enquadrá-lo na categoria dos coordenados não síncronos. São estabelecidos pontos forçados que determinam linhas de recuperação. De forma semelhante ao proposto nesta tese, é utilizado um coordenador para coordenar a atividade de estabelecimento de uma nova linha de recuperação. Ainda, este algoritmo permite processamento não determinístico, transparente (o algoritmo de recuperação é implementado no sistema operacional ou entre este e a aplicação. Além disso, a aplicação não necessita ter conhecimento da existência de um mecanismo para recuperação) e problemas de *livelocks* são tratados através do mecanismo de reencarnações [STR 85].

Da mesma forma que o algoritmo de Silva e Silva, o algoritmo proposto por Gendelman *et al.* é do tipo coordenado não síncrono. Este algoritmo também utiliza-se de informações anexadas as mensagens, de maneira a identificar os instantes mais apropriados para o estabelecimento forçado dos pontos de recuperação. Entretanto, foi desenvolvido baseado no tipo de canal disponível (canal confiável ou não) e através do impedimento da ocorrência de mensagens potencialmente perdidas.

5.1 Algoritmo de Briatico, Ciuffoletti e Simoncini

Os autores propõem um algoritmo de recuperação distribuído que não sofre do efeito dominó. Para isso, no artigo, é apresentada uma definição formal dos conceitos relacionados ao estabelecimento de pontos de recuperação coordenados. A base desta formalização é a de axiomas usados para provar o teorema principal, onde são estabelecidos os critérios que determinam as características de uma linha de recuperação. No final, é apresentada uma proposta de implementação e, nas conclusões, são apresentados pontos não cobertos pelo protocolo.

A formalização do estabelecimento dos pontos de recuperação é dividida da seguinte forma: modelo de computação, linha de recuperação e pontos de recuperação (chamados de pontos planejados).

No modelo de computação são apresentados os processos, como uma seqüência de eventos, os eventos propriamente ditos e uma função de interação, que corresponde à formalização das mensagens usadas pelos processos para a troca de informação.

A função de interação é definida através de quatro axiomas: o primeiro axioma caracteriza as mensagens como sendo formadas por um evento de transmissão e um de recepção que não ocorrem no mesmo processo; no segundo, é dito que uma mensagem pode ter apenas um transmissor e um receptor; o terceiro caracteriza a garantia de recepção das mensagens e o quarto fornece uma relação entre o número

de pontos de recuperação estabelecidos antes do evento de transmissão de uma mensagem e antes do evento de recepção da mesma mensagem. Este último axioma, quando associado com outros, visa impedir a ocorrência do efeito dominó.

Para ordenar os eventos, são definidas duas relações de ordem: uma relação de ordem total, que ordena os eventos ocorridos em um mesmo processo, e uma relação de ordem parcial, que ordena os eventos ocorridos em processos diferentes. Esta relação de ordem parcial está intimamente ligada à função de interação e, portanto, com as mensagens.

Uma linha de recuperação foi definida através de dois conjuntos: um conjunto de pontos de recuperação (chamado de RLE) e um conjunto de mensagens recuperáveis (chamado de RLM).

O conjunto RLE deverá ser formado de tal modo que contenha apenas um ponto de recuperação para cada processo. Além disso, todas as mensagens cujos processos transmissor e receptor tiverem representantes no RLE, devem ter os eventos formadores (transmissão e recepção) posteriores ao do estabelecimentos dos respectivos pontos de recuperação. Finalmente, se uma mensagem estiver registrada no RLM, então a sua recepção deverá ser posterior ao ponto de recuperação registrado no RLE, pelo receptor. Além disso, não poderá ocorrer da mensagem ter sido transmitida após o ponto de recuperação ter sido registrado no RLE, pelo transmissor. Com isso, é garantido que somente as mensagens potencialmente perdidas estarão registradas no RLM.

Um ponto de recuperação planejado (PRP) foi definido pela estrutura formada por um *timestamp* (**T**), um conjunto **PE**, subconjunto dos processos do sistema (**Q**) e um conjunto **MS**, subconjunto do conjunto de mensagens **M**.

Além da definição dos componentes de um ponto de recuperação, os autores propuseram duas funções: a função **PRPset**, que vincula a cada evento um conjunto de pontos de recuperação, e a função **K**, que associa um *timestamp* (local a cada processo) a cada evento.

Usando a notação proposta por Briatico *et al.*, **PRPset**(α)[n] indica o conjunto de pontos de recuperação associados ao evento α, no *timestamp* "n". De forma semelhante, **K**(α) representa o *timestamp* associado ao evento α, pelo processo onde α ocorreu.

Seguem, a definição dos PRPs e as funções associadas, um conjunto de quatro axiomas, que estabelecem:

- as condições iniciais das funções **K**;
- a forma como as funções **K** e **PRPset** são alteradas, quando do estabelecimento de um ponto de recuperação;
- a forma como as funções **K** e **PRPset** são alteradas, quando da transmissão e recepção de uma mensagem;
- quais outros eventos não alteram **K** nem **PRPset**.

Finalmente, é apresentado o teorema que une todas as definições anteriores na forma da definição de uma linha de recuperação. Para este teorema, os autores apresentam uma prova formal.

Com as definições apresentadas, é proposto um algoritmo que, segundo os autores, implementa a especificação formal. Alguns aspectos desta implementação devem ser mencionados.

Para implementar o mecanismo que impede a ocorrência do efeito dominó, é acrescentado a cada mensagem um *timestamp*. Quando uma mensagem é recebida, o processo receptor pode atuar no sentido de registrar o identificador do processo transmissor e a mensagem recebida. Além disso, pode ser levado ao estabelecimento de um novo ponto de recuperação.

O mecanismo proposto não envolve mensagens de resposta, uma vez que é assumida a garantia de recebimento das mensagens: é pressuposto que exista um protocolo básico para comunicação confiável.

A escolha do salvamento das mensagens no receptor, visando uma futura repetição quando do retorno, não foi ao acaso. A outra opção seria salvamento no transmissor, o que levaria à necessidade de algum tipo de mensagem de resposta, para determinar a eliminação daquelas mensagens já entregues.

Mais ainda, a escolha do salvamento das mensagens no receptor implica na premissa da entrega garantida. Isso acontece porque, caso não houvesse garantia de entrega, uma mensagem poderia ser perdida no canal. Desta forma, o transmissor teria informação da transmissão da mensagem enquanto que o receptor não a teria armazenado na sua memória.

A recuperação é feita com o auxílio da premissa segundo a qual um processo que está esperando pelo término do processamento de outro não sofrerá falhas. Além disso, para completar a recuperação, é necessário que os canais sejam duplicados e que os processos possuam reservas à quente.

Finalmente, os autores colocam como trabalho em aberto, a tarefa de coleta de lixo, uma vez que o algoritmo proposto mantém, em memória, todos os pontos estabelecidos.

Os três pontos em que o artigo de Briatico *et al.* não fornece uma resposta são respondidos pelo protocolo proposto nesta tese. Na proposta atual, não é necessária a existência de mecanismos de comunicação confiável, o que amplia o cenário dos sistemas onde este tipo de mecanismo pode ser utilizado; o salvamento das mensagens para uma possível repetição é feito no transmissor, o que resolve a questão das mensagens eventualmente perdidas no canal e, finalmente, faz parte do algoritmo proposto a coleta de lixo, ou seja, no pior caso são mantidos apenas dois conjuntos de pontos de recuperação. No algoritmo de Briatico *et al.* não existe previsão para coleta de lixo: na realidade, os autores reconhecem como limitação de seu algoritmo o fato que "...tende a considerar úteis todos os pontos de recuperação estabelecidos", o que representa um custo adicional de armazenamento estável.

5.2 Algoritmo de Silva e Silva

No artigo de L. Silva e João Gabriel Silva [SIL 92], é proposto um algoritmo transparente para a recuperação em sistemas distribuídos. A definição de transparência está relacionada com a implementação dos mecanismos de recuperação no sistema operacional e do fato de que a aplicação não necessita ser informada da existência destes mecanismos.

Os autores apresentam como motivação suficiente para o uso da transparência

do mecanismo de estabelecimento dos pontos de recuperação e da recuperação propriamente dita, a separação de interesses. Com isso, a tarefa de programação da aplicação não tem a sua complexidade aumentada devido aos requisitos de tolerância a falhas.

Entretanto, é colocada como limitação deste mecanismo o fato de ser útil, apenas, para falhas de *hardware*.

Para justificar os mecanismos usados em seu algoritmo, Silva e Silva propõem que os algoritmos usados na recuperação transparente sejam classificados em duas classes principais: independentes e consistentes. Os independentes sofrem do **efeito dominó**. Para evitar este efeito, foram incrementados com o mecanismo de **registro de mensagens** (*message log*). São citados como pontos fracos desta classe de algoritmos o determinismo de execução e o fato de não considerarem a possibilidade de propagação de erros, que pode existir entre a ocorrência da falha e a detecção dos erros subseqüentes.

Segundo os autores, os consistentes permitem o não determinismo da execução e resolvem o problema da propagação dos erros. Nesta classe de algoritmos, cada ponto de recuperação faz parte de um estado global, entendido conforme o conceito usado por Chandy e Lamport [CHA 85]: nenhuma mensagem pode ter sido registrada como recebida sem antes ter sido registrada como transmitida.

Duas características são citadas como desvantagens dessa segunda classe de algoritmos: são requeridas mensagens de controle e, em algumas propostas, são introduzidos mecanismos de sincronização, que bloqueiam a aplicação. No texto do artigo, são listadas algumas propostas a partir das quais as desvantagens citadas são analisadas.

O algoritmo proposto por Silva e Silva não requer relógios sincronizados, permite processamento não-determinístico e não estabelece limitações quanto a mensagens perdidas, duplicadas ou entregues fora de ordem.

Como critério de consistência, Silva e Silva estabelecem três regras: regra 1, a transmissão e a recepção de uma mensagem deve ocorrer antes dos pontos de recuperação dos respectivos processos; regra 2, a transmissão de uma mensagem deve ocorrer antes do ponto de recuperação, no processo transmissor, e a recepção pode ocorrer após o ponto de recuperação do processor receptor (o que corresponde a uma mensagem perdida); regra 3, um processo só pode estabelecer novo ponto de recuperação após ter encerrado o estabelecimento do ponto de recuperação global anterior.

Para o estabelecimento de um ponto de recuperação, é necessária a existência de um coordenador. Este, periodicamente, inicia a criação de novo ponto de recuperação, através do envio de solicitação específica para cada processo do sistema. Após estabelecidos os pontos de recuperação por cada processo, estes respondem ao coordenador, de forma que possa dar como encerrada a atividade daquele ponto de recuperação global.

Para garantir a consistência dos pontos de recuperação estabelecidos da forma indicada, foi incluído nas mensagens da aplicação o índice do último ponto de recuperação estabelecido. Desta forma, o processo receptor pode identificar quando a mensagem foi transmitida e, comparando esta informação com o estado atual dos seus pontos de recuperação, pode determinar se uma mensagem é potencialmente perdida. Se este for o caso, o receptor armazena a mensagem pois, em caso de retorno, esta mensagem deverá ser repetida para garantir a consistência.

Os algoritmos de recuperação podem ser divididos em duas fases distintas: o estabelecimento dos pontos de recuperação e a recuperação propriamente dita. No algoritmo de Silva e Silva, certamente, a fase mais importante para garantir a operação consistente do sistema é a recuperação. Faz parte da atividade de recuperação, além do retorno propriamente dito, impedir a ocorrência dos *livelocks* e eliminar a latência de detecção dos erros. Também são propostas algumas otimizações ortogonais ao problema da recuperação: só força o retorno dos processos que efetivamente se comunicaram (de forma direta ou indireta) com o processo que falhou; se um processo falhou mas não se comunicou, então este processo retorna sem informar aos outros. Entretanto, estas otimização não foram implementadas.

Para resolver a questão dos *livelocks*, é utilizado um contador de recuperações. A cada recuperação, este contador é incrementado. Como estes contadores são também anexados às mensagens da aplicação, o processo receptor pode saber se esta mensagem foi transmitida. Caso tenha sido em um período de recuperação anterior ao atual, esta mensagem é descartada. Este mecanismo fornece um modo de *limpar o canal* sem bloquear a aplicação. A desvantagem é a necessidade de acrescentar mais informação às mensagens da aplicação.

A questão da latência dos erros foi resolvida utilizando um injetor de falhas. Este foi usado para determinar uma matriz de latências que será usada pelo algoritmo de recuperação. Sempre que ocorre um erro, o mecanismo de retorno calcula, a partir da matriz de latências, quando a falha deve ter ocorrido. Se o instante calculado para a falha está no mesmo intervalo de recuperação de sua detecção, retorna-se para o último ponto estabelecido; se o instante calculado está no intervalo anterior, retorna-se para o ponto de recuperação anterior. É premissa que o intervalo entre pontos de recuperação seja maior do que a latência de detecção dos erros.

Silva e Silva argumentam que seu algoritmo pode ser usado em ambientes cujos canais podem perder mensagens e propõem um mecanismo que provê o armazenamento das mensagens que deverão ser repetidas, no caso do retorno, nos processos receptores. Como não existe premissa de comunicação confiável, não há garantia de entrega das mensagens. Desta forma, uma mensagem perdida no canal não será repetida, colocando o sistema em estado inconsistente.

Segundo os autores, é necessário manter apenas dois pontos de recuperação globais completos, para garantir a correta recuperação. Entretanto, não foi considerado que, para descartar um ponto de recuperação, é necessário que outro esteja pronto, para substituí-lo. Desta forma, será necessário reservar memória estável para três pontos de recuperação completos. Além disso, não é discutida nem mencionada, dentre os trabalhos futuros, a necessidade de propor um mecanismo de coleta de lixo: dos pontos de recuperação e dos registros das mensagens recebidas.

Apesar dos autores afirmarem que recuperação por retorno só pode ser usada para tolerar falhas de *hardware*, em trabalhos mais recentes, Wang [WAN 95] e Huang [HUA 95] apresentam argumentos que indicam como viável a utilização de recuperação por retorno para tratar as falhas de *software*. E, como premissa, os processos devem apresentar um comportamento não determinístico (mais precisamente, determinístico em partes – *piecewise deterministic* – abreviado por **PWD**).

Finalmente, a informalidade da apresentação pode não assegurar o cumprimento de todas as propriedades do algoritmo. Isso seria resolvido se houvesse uma especificação formal e uma indicação de como provar a correção da mesma.

5.3 Algoritmo de Gendelman, Bic e Dillencourt

A discussão apresentada aqui tem como base o trabalho apresentado por Gendelman *et al.*[GEN 99] e um relatório técnico mais detalhado [GEN 99a].

O trabalho apresenta um algoritmo para recuperação de processos onde é suposta a garantia de entrega das mensagens pelos canais.

Conforme os autores, o algoritmo apresenta as seguintes características:

- operação com canais confiáveis;
- é não bloqueante;
- causa pouco impacto no desempenho, quando em operação livre de erros;
- é escalável;
- não supõe características específicas dos sistema;
- simples, para facilitar a implementação e a prova de correção.

O algoritmo é apresentado em etapas. Na primeira etapa, é apresentado um algoritmo para ser usado com canais não confiáveis. Desta forma, o mecanismo usado para a recuperação não necessita tratar as mensagens perdidas em um eventual retorno: são tratadas apenas as mensagens órfãs.

Neste algoritmo, um coordenador envia mensagens de solicitação de estabelecimento de pontos de recuperação locais para todos os outros processos do sistema. Estas mensagens carregam um índice que identifica o último ponto de recuperação estabelecido.

Ao receberem estas mensagens, os outros processos estabelecem um ponto de recuperação tentativo e respondem ao coordenador.

Adicionalmente, em todas as mensagens da aplicação é anexado o índice do último ponto de recuperação estabelecido no processo transmissor. Assim, caso um processo receba uma mensagem com um índice de ponto de recuperação maior do que o seu próprio, um novo ponto de recuperação é estabelecido e é gerada uma resposta para o coordenador. Este mecanismo visa tratar as mensagens recebidas fora de ordem.

Quando o coordenador recebe todas as respostas, transmite uma mensagem de confirmação. A esta, os demais processos respondem com a transformação do ponto tentativo em permanente, descartando o ponto de recuperação permanente anterior.

Em seguida, são discutidas as mensagens perdidas e sua relação com os canais confiáveis. Então é apresentado um novo algoritmo para recuperação, que pode operar com canais confiáveis. Basicamente, é acrescentado o tratamento de mensagens perdidas, devido à recuperação.

A base do tratamento das mensagens perdidas é esperar que todas as mensagens potencialmente perdidas cheguem ao seu destino. Assim, uma mensagem registrada como transmitida mas não registrada como recebida fará com que o estabelecimento do estado global seja suspenso até que ela atinja seu destino.

Para tratar as mensagens perdidas, todos os outros processos acrescentam à mensagem enviada em resposta à solicitação de estabelecimento de um ponto

de recuperação, um contador que indica o balanço entre o número de mensagens transmitidas e recebidas.

Quando um processo detecta uma mensagem com o índice do ponto de recuperação menor do que o seu próprio, está caracterizada uma mensagem potencialmente perdida. Estas mensagens são armazenadas, em *log*, no receptor. Então, o processo informa ao coordenador o recebimento de uma mensagem potencialmente perdida. Com esta resposta, o coordenador pode decrementar o balanço de mensagens transmitidas e recebidas. Quando todos os balanços dos processos chegam a zero, o coordenador envia a mensagem de confirmação do ponto de recuperação, encerrando-o.

Alguns comentários podem ser tecidos a respeito do mecanismo de tratamento das mensagens perdidas.

Em primeiro lugar, o encerramento do estabelecimento dos pontos de recuperação deve ser adiado até que não existam mais mensagens potencialmente perdidas. Isso pode ser um problema, caso uma mensagem atrase mais do que o período entre o estabelecimento de pontos de recuperação.

Outro problema diz respeito à implementação da comunicação confiável. Sabe-se que algumas implementações caracterizadas com o qualificativo de comunicação confiável, por exemplo TCP, são do tipo *best-effort*, ou seja, pode ocorrer alguma perda de mensagens. Esta perda poderia levar ao bloqueio do mecanismo de estabelecimento de pontos de recuperação.

Um dos processamentos do algoritmo prevê que as mensagens potencialmente perdidas sejam anexadas ao ponto de recuperação cujo índice **corresponde àquele da mensagem**. Isso faz com que as mensagens potencialmente perdidas só possam ter índice igual ao do último ponto estabelecido. Se fosse diferente, haveria a necessidade de guardar mais pontos de recuperação do que apenas o último. Assim, é necessário garantir que um ponto de recuperação só possa ser iniciado quando o anterior tiver sido completado. Apesar disso, nenhuma discussão a respeito das condições necessárias para o início do estabelecimento de um novo ponto de recuperação é apresentada.

Mesmo sem a discussão do processo de retorno, supõe-se que seja utilizado o último ponto de recuperação global completo, que forma uma linha de recuperação. Assim, o algoritmo será capaz de operar corretamente, mesmo que ocorram falhas durante o estabelecimento de novos pontos de recuperação, uma vez que, neste caso, não estará estabelecido um ponto de recuperação completo.

Ainda no que diz respeito ao retorno, um cenário pode ser problemático. Suponha-se que uma falha ocorra após o coordenador ter tornado seu ponto de recuperação tentativo em permanente mas antes de poder enviar as mensagens de confirmação. Neste caso, o coordenador estará com o último ponto de recuperação enquanto que os outros processos ainda estarão com o ponto anterior.

O algoritmo proposto no presente trabalho resolve os problemas citados. Em primeiro lugar, o retorno é feito usando um mecanismo que procura por uma linha de recuperação dentre os últimos pontos de recuperação estabelecidos. Devido ao mecanismo usado para o estabelecimento dos pontos de recuperação, pode haver apenas dois pontos de recuperação locais por processo: o ponto atual e o anterior.

Em segundo lugar, o estabelecimento dos pontos de recuperação não fica esperando pela chegada das mensagens potencialmente perdidas: elas já estão na memória, quando são estabelecidos os pontos de recuperação, podendo ser gravadas

na memória estável, junto com o estado dos processos.

Finalmente, caso a implementação dos canais seja do tipo *best effort* e mensagens possam ser perdidas, o mecanismo de *log* na transmissão garante que mesmo aquelas estarão no estado do processo, quando for estabelecido um novo ponto de recuperação.

6 Descrição formal do algoritmo

O algoritmo proposto é do tipo que garante que sempre exista uma linha de recuperação. Desta forma, sempre existirá um ponto de recuperação consistente que poderá ser utilizado em caso de falha. Além disso, o procedimento de retorno é bastante simplificado, bastando que o sistema retorne para a última linha de recuperação estabelecida.

Sendo assim, a prova deste tipo de algoritmo reduz-se a demonstrar que sempre existe uma linha de recuperação. Portanto, o motivo da especificação e prova será a primeira etapa do algoritmo: o estabelecimento consistente dos pontos de recuperação.

6.1 Formalização do critério de consistência

Na seção 4.2.2, foram discutidos os cenários que garantem a consistência de dois pontos de recuperação quando em presença de mensagens da aplicação e suas respectivas respostas de confirmação da recepção. Isso foi feito através da análise da chamada **vida da mensagem**. Foram obtidos cinco cenários consistentes.

Nesta seção, os cenários consistentes serão formalizados. Assim, será possível usá-los para verificar a correção da especificação do algoritmo.

Define-se $Par(m, C_1, C_2)$ a partir das cinco condições de consistência identificadas na seção 4.2.2, que relacionam um par de pontos de recuperação C_1 e C_2 e a mensagem m. Formalmente, $Par(m, C_1, C_2)$ é definido da seguinte forma:

$$
\begin{aligned}
&Par(m, C_1, C_2) \triangleq \\
\vee\ &\wedge(m.d = \mathbf{ACK}) \wedge Antes(m.itx, C_2) \wedge Antes(m.irx, C_1) \\
&\wedge(m.msg.d = \mathbf{APP}) \wedge Antes(m.msg.itx, C_1) \wedge Antes(m.msg.irx, C_2) \\
\vee\ &\wedge(m.d = \mathbf{ACK}) \wedge Antes(m.itx, C_2) \wedge Apos(m.irx, C_1) \\
&\wedge(m.msg.d = \mathbf{APP}) \wedge Antes(m.msg.itx, C_1) \wedge Antes(m.msg.irx, C_2) \\
&\wedge(m.msg \in C_1) \\
\vee\ &\wedge(m.d = \mathbf{ACK}) \wedge Apos(m.itx, C_2) \wedge Apos(m.irx, C_1) \\
&\wedge(m.msg.d = \mathbf{APP}) \wedge Antes(m.msg.itx, C_1) \wedge Antes(m.msg.irx, C_2) \\
&\wedge(m.msg \in C_1) \\
\vee\ &\wedge(m.d = \mathbf{ACK}) \wedge Apos(m.itx, C_2) \wedge Apos(m.irx, C_1) \\
&\wedge(m.msg.d = \mathbf{APP}) \wedge Antes(m.msg.itx, C_1) \wedge Apos(m.msg.irx, C_2) \\
&\wedge(m.msg \in C_1) \\
\vee\ &\wedge(m.d = \mathbf{ACK}) \wedge Apos(m.itx, C_2) \wedge Apos(m.irx, C_1) \\
&\wedge(m.msg.d = \mathbf{APP}) \wedge Apos(m.msg.itx, C_1) \wedge Apos(m.msg.irx, C_2)
\end{aligned}
$$

Tomando como verdadeiro que $Antes(x, C) \wedge Apos(x, C) = FALSE$ e que $Antes(x, C) \vee Apos(x, C) = TRUE$, o terceiro termo da expressão anterior, se reunido ao segundo e ao quarto, pode ser simplificado, o que fornece os casos 2 e 3 da equação 6.1.

$$
\begin{array}{l}
Par(m, C_1, C_2) \triangleq \\
\quad Caso\ 1 \left\{ \begin{array}{ll} \vee & \wedge(m.d = \mathbf{ACK}) \wedge Antes(m.itx, C_2) \wedge Antes(m.irx, C_1) \\ & \wedge(m.msg.d = \mathbf{APP}) \\ & \wedge Antes(m.msg.itx, C_1) \wedge Antes(m.msg.irx, C_2) \end{array} \right. \\
\quad Caso\ 2 \left\{ \begin{array}{ll} \vee & \wedge(m.d = \mathbf{ACK}) \wedge Apos(m.irx, C_1) \\ & \wedge(m.msg.d = \mathbf{APP}) \\ & \wedge Antes(m.msg.itx, C_1) \wedge Antes(m.msg.irx, C_2) \\ & \wedge(m.msg \in C_1) \end{array} \right. \\
\quad Caso\ 3 \left\{ \begin{array}{ll} \vee & \wedge(m.d = \mathbf{ACK}) \wedge Apos(m.itx, C_2) \wedge Apos(m.irx, C_1) \\ & \wedge(m.msg.d = \mathbf{APP}) \\ & \wedge Antes(m.msg.itx, C_1) \wedge (m.msg \in C_1) \end{array} \right. \\
\quad Caso\ 4 \left\{ \begin{array}{ll} \vee & \wedge(m.d = \mathbf{ACK}) \wedge Apos(m.itx, C_2) \wedge Apos(m.irx, C_1) \\ & \wedge(m.msg.d = \mathbf{APP}) \\ & \wedge Apos(m.msg.itx, C_1) \wedge Apos(m.msg.irx, C_2) \end{array} \right.
\end{array}
\tag{6.1}
$$

Pode-se simplificar a expressão anterior, colocando em evidência a definição dos tipos das mensagens: $m.d = \mathbf{ACK}$ e $m.msg.d = \mathbf{APP}$. Com isso, obtém-se como parte da expressão resultante, quatro subexpressões que representam as combinações consistentes dos instantes de estabelecimento dos pontos de recuperação e os eventos de transmissão e recepção das mensagens.

Na figura 6.1 estão representados os quatro casos. Nessa figura, as mensagens estão identificadas pelos seus nomes. Quando há a necessidade de armazenamento da mensagem da aplicação no ponto de recuperação do processo transmissor, esta é indicada por uma linha tracejada.

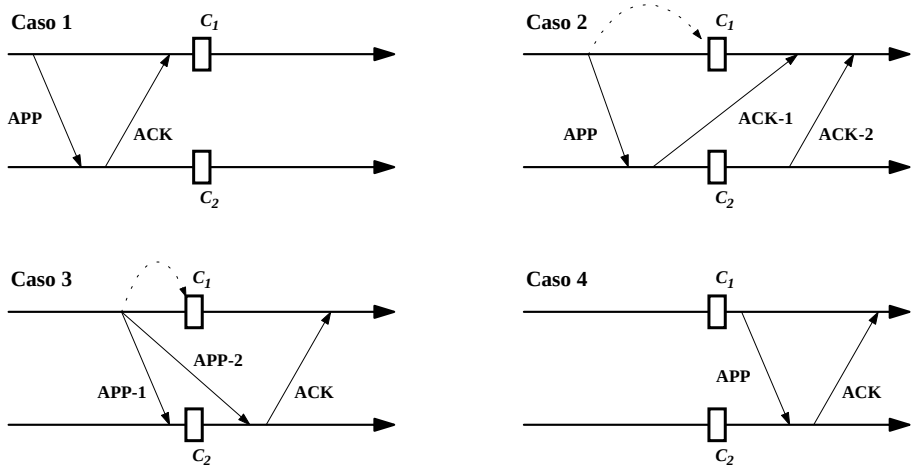

FIGURA 6.1 – Os quatro casos consistentes

Serão adotadas algumas simplificações para a expressão $Par(m, C_1, C_2)$.

Para indicar que todos os eventos da vida da mensagem ocorreram **antes** do estabelecimento dos pontos de recuperação nos processos transmissor e receptor da mesma, será usado:

$$
ParMB(m, C_1, C_2) \quad \triangleq \quad \wedge Antes(m.itx, C_2) \wedge Antes(m.irx, C_1) \tag{6.2}
$$

$$\wedge Antes(m.msg.itx, C_1) / \ Antes(m.msg.irx, C_2)$$

Quando todos os eventos da vida da mensagem ocorreram **após** o estabelecimento dos pontos de recuperação nos processos transmissor e receptor da mesma, será usado:

$$\begin{aligned} ParMA(m, C_1, C_2) \quad \triangleq \quad & \wedge Apos(m.itx, C_2) \wedge Apos(m.irx, C_1) \\ & \wedge Apos(m.msg.itx, C_1) \wedge Apos(m.msg.irx, C_2) \end{aligned} \tag{6.3}$$

Se os dois eventos que definem a mensagem de resposta estão após o estabelecimento dos pontos de recuperação; a transmissão da mensagem da aplicação ocorreu antes do ponto de recuperação; e a mensagem da aplicação está armazenada no ponto de recuperação do transmissor, tem-se a seguinte simplificação:

$$\begin{aligned} ParMK(m, C_1, C_2) \quad \triangleq \quad & \wedge Apos(m.itx, C_2) \wedge Apos(m.irx, C_1) \\ & \wedge Antes(m.msg.itx, C_1) \wedge (m.msg \in C_1) \end{aligned} \tag{6.4}$$

A última simplificação pode ser realizada quando a mensagem da aplicação tem seus eventos de definição antes do estabelecimento dos pontos de recuperação; a mensagem de resposta foi recebida após o ponto de recuperação do transmissor; e a mensagem da aplicação está armazenada no ponto de recuperação do transmissor. Tem-se o seguinte:

$$\begin{aligned} ParMP(m, C_1, C_2) \quad \triangleq \quad & \wedge Apos(m.irx, C_1) \wedge Antes(m.msg.itx, C_1) \\ & \wedge Antes(m.msg.irx, C_2) \wedge (m.msg \in C_1) \end{aligned} \tag{6.5}$$

Com as simplificações, a expressão completa para $Par(m, C_1, C_2)$ será:

$$\begin{aligned} Par(m, C_1, C_2) \triangleq \ & \wedge \ (m.d = \mathbf{ACK}) \wedge (m.msg.d = \mathbf{APP}) \\ & \wedge \ \vee ParMB(m, C_1, C_2) \\ & \quad \vee ParMP(m, C_1, C_2) \\ & \quad \vee ParMK(m, C_1, C_2) \\ & \quad \vee ParMA(m, C_1, C_2) \end{aligned} \tag{6.6}$$

O segundo termo da conjunção representa a condição de consistência no que diz respeito ao relacionamento entre os instantes dos eventos da mensagem e do estabelecimento dos pontos de recuperação. Este termo será representado pela fórmula:

$$\begin{aligned} ParMCC(m, C_1, C_2) \triangleq \ & \vee ParMB(m, C_1, C_2) \\ & \vee ParMP(m, C_1, C_2) \\ & \vee ParMK(m, C_1, C_2) \\ & \vee ParMA(m, C_1, C_2) \end{aligned} \tag{6.7}$$

Substituindo a expressão 6.7, na expressão 6.6, ter-se-á:

$$\begin{aligned} Par(m, C_1, C_2) \triangleq \ & \wedge (m.d = \mathbf{ACK}) \wedge (m.msg.d = \mathbf{APP}) \\ & \wedge ParMCC(m, C_1, C_2) \end{aligned} \tag{6.8}$$

A expressão 6.8 corresponde aos tipos de mensagens que afetam a consistência. Implicitamente, segundo a fórmula, todos os outros tipos levam a um estado inconsistente. Entretanto, isso não é necessariamente verdadeiro: é preciso definir se as mensagens diferentes de **APP** e **ACK** poderão afetar ou não a consistência. Para solucionar a questão, basta observar que as mensagens diferentes de **APP** e **ACK** não afetam o estado da aplicação: portanto não afetam a consistência. Desta forma, a conjunção pode ser transformada em uma implicação. Assim, ter-se-á:

$$Par(m, C_1, C_2) \triangleq \left(\begin{array}{ll} \wedge(m.d = \mathbf{ACK}) & \\ \wedge(m.msg.d = \mathbf{APP}) & \Rightarrow \quad ParMCC(m, C_1, C_2) \end{array} \right) \tag{6.9}$$

Como uma mensagem só interfere com a consistência de um par de pontos de recuperação estabelecidos nos processos transmissor e receptor da mesma, os índices dos pontos de recuperação C_1 e C_2 serão substituídos pela indicação do processo no qual foi estabelecido aquele ponto de recuperação específico. Desta forma, uma vez que $m.tx = m.msg.rx$ e $m.rx = m.msg.tx$, o ponto C_1 poderá ser substituído por $C_{m.msg.tx}$ ou $C_{m.rx}$ e o ponto C_2 poderá ser substituído por $C_{m.msg.rx}$ ou $C_{m.tx}$. Assim, pode-se substituir os parâmetros C_1 e C_2 por C, apenas. A nova expressão da consistência será:

$$Conj(m, C) \triangleq \left(\begin{array}{ll} \wedge(m.d = \mathbf{ACK}) & \\ \wedge(m.msg.d = \mathbf{APP}) & \\ \wedge(m.tx = m.msg.rx) & \Rightarrow \quad ConjCC(m, C) \\ \wedge(m.rx = m.msg.tx) & \end{array} \right) \tag{6.10}$$

Note-se que o novo operador *ConjCC* corresponde a *ParMCC* onde foram substituídos os índices dos pontos de recuperação. Além disso, os operadores *Conj* referem-se a conjuntos de pontos de recuperação, enquanto que os operadores *Par* referiam-se a pares de pontos de recuperação.

Entretanto, a fórmula $Conj(m, C)$ expressa, apenas, o critério pelo qual pode-se julgar se uma mensagem é consistente ou não, em relação ao conjunto de pontos de recuperação C. Isso **não é suficiente** para verificar a correção do algoritmo. Deve-se dizer que **apenas as mensagens geradas** pelo algoritmo necessitam apresentar esta propriedade. Ou seja, dentre os elementos do conjunto das mensagens, o algoritmo só permite gerar aqueles que satisfizerem a propriedade $Conj(m, C)$.

Assim sendo, seja H o conjunto das mensagens geradas, e usando a condição de consistência para qualquer mensagem, a formalização do critério de consistência a ser usado para o algoritmo será:

$$ConjMH(H, C) \triangleq$$
$$\forall m \in Message : \left(\begin{array}{ll} \wedge(m.d = \mathbf{ACK}) & \\ \wedge(m.msg.d = \mathbf{APP}) & \\ \wedge(m.tx = m.msg.rx) & \Rightarrow \quad ConjCC(m, C) \\ \wedge(m.rx = m.msg.tx) & \\ \wedge(m \in H) & \end{array} \right) \tag{6.11}$$

A expressão 6.11 descreve o conjunto de todas as mensagens geradas pelo algoritmo, de maneira que sejam consistentes. Note-se que, nos parâmetros do operador

ConjMH, aparecem não mais uma mensagem mas todo o conjunto das mensagens geradas pelo algoritmo. Entretanto, conforme a definição de $ConjMH(H, C)$, as mensagens contidas em H deverão ser todas do tipo **ACK**. Para que seja mais geral, a equação 6.11 será transformada de maneira a expressar a consistência de qualquer par de elementos do conjunto das mensagens. O resultado será:

$$MH(H, C) \triangleq$$
$$\forall x, y \in Message : \left(\begin{array}{l} \wedge(x \in H) \\ \wedge(y \in H) \\ \wedge(y.d = \mathbf{ACK}) \\ \wedge(x.d = \mathbf{APP}) \\ \wedge(y.tx = x.rx) \\ \wedge(y.rx = x.tx) \\ \wedge(y.msg = x) \end{array} \right. \Rightarrow \left. CC(x, y, C) \right) \tag{6.12}$$

A equação 6.12 expressa o relacionamento entre mensagens (x e y) e os pontos de recuperação (conjunto C), de maneira que sejam consistentes (formem uma linha de recuperação). A equação aplica-se às mensagens com as seguintes características:

- foram geradas ($x \in H$ e $y \in H$);
- são do tipo **APP** e **ACK** ($x.d = \mathbf{APP}$ e $y.d = \mathbf{ACK}$);
- estão relacionadas de maneira que uma é a resposta da outra ($y.msg = x$);
- e o transmissor de uma é o receptor da outra ($y.tx = x.rx$ e $y.rx = x.tx$).

As mensagens não caracterizadas são consideradas consistentes, uma vez que tornam verdadeiro o operador $MH(H, C)$.

Finalmente, a equação 6.12 pode ser mais adequadamente formalizada pelas seguintes expressões:

$$MH(H, C) \triangleq \forall x, y \in Message : \left(\begin{array}{l} \wedge(x \in H) \\ \wedge(y \in H) \\ \wedge M(x, y) \end{array} \right. \Rightarrow \left. CC(x, y, C) \right) \tag{6.13}$$

$$M(x, y) \triangleq \left(\begin{array}{l} \wedge(x.d = \mathbf{APP}) \\ \wedge(y.d = \mathbf{ACK}) \\ \wedge(y.tx = x.rx) \\ \wedge(y.rx = x.tx) \\ \wedge(y.msg = x) \end{array} \right) \tag{6.14}$$

$$CC(x, y, C) \triangleq \left(\begin{array}{l} \vee MB(x, y, C) \\ \vee MA(x, y, C) \\ \vee MK(x, y, C) \\ \vee MP(x, y, C) \end{array} \right) \tag{6.15}$$

$$MB(x, y, C) \triangleq \left(\begin{array}{l} \wedge Antes(x.itx, C_{x.tx}) \\ \wedge Antes(x.irx, C_{x.rx}) \\ \wedge Antes(y.itx, C_{x.rx}) \\ \wedge Antes(y.irx, C_{x.tx}) \end{array} \right) \tag{6.16}$$

$$MA(x, y, C) \quad \triangleq \quad \begin{pmatrix} \wedge Apos(x.itx, C_{x.tx}) \\ \wedge Apos(x.irx, C_{x.rx}) \\ \wedge Apos(y.itx, C_{x.rx}) \\ \wedge Apos(y.irx, C_{x.tx}) \end{pmatrix} \tag{6.17}$$

$$MK(x, y, C) \quad \triangleq \quad \begin{pmatrix} \wedge Antes(x.itx, C_{x.tx}) \\ \wedge Apos(y.itx, C_{x.rx}) \\ \wedge Apos(y.irx, C_{x.tx}) \\ \wedge x \in C_{x.tx} \end{pmatrix} \tag{6.18}$$

$$MP(x, y, C) \quad \triangleq \quad \begin{pmatrix} \wedge Antes(x.itx, C_{x.tx}) \\ \wedge Antes(x.irx, C_{x.rx}) \\ \wedge Apos(y.irx, C_{x.tx}) \\ \wedge x \in C_{x.tx} \end{pmatrix} \tag{6.19}$$

As expressões 6.13 até 6.19 são aquelas que descrevem, formalmente, o conceito de consistência. Levam em consideração mensagens órfãs e perdidas, além da existência de mensagens de resposta. Estas expressões serão usadas para provar que o algoritmo garante a consistência.

6.2 Descrição da especificação

Nesta seção serão apresentadas a especificação do algoritmo e a descrição de seus elementos. A especificação, escrita em TLA, é a seguinte:

```
MODULE Recovery

EXTENDS Naturals

CONSTANTS
  Proc,          Conjunto dos identificadores dos processos do sistema
  Mngr,          Identificador do manager (Coordenador)
  Deliver        Deliver(m) Entrega uma mensagem para a aplicacao

VARIABLES
  CurrCP         Ultimo CP estabelecido
  PrevCP         Penultimo CP estabelecido
  CS             Estado do canal - memoria volatil
  ME             Estado do Manager
  Canal          Canais de comunicacao entre os processos
  H              Message HISTORY

ASSUME
  w ≜ <CurrCP, PrevCP, CS, ME, Canal, H>
  MsgType ≜ { "APP", "ACK", "REQ", "AREQ", "CMT", "ACMT" }
  NN ≜ CHOOSE x: (∀n ∈ Nat: x > n)
  NoMsg ≜ CHOOSE x: (x.d ∉ MsgType)
  Message ≜ [tx: Proc, rx: Proc, itx: Nat, irx: Nat ∪ { NN },
                    d: MsgType, msg: Message ∪ { NoMsg }]
  ProcS ≜ Proc \ {Mngr}

TypeInvariant ≜
       ∧ CurrCP = [p ∈ Proc ↦ Nat x SUBSET(Message)]
       ∧ PrevCP = [p ∈ Proc ↦ Nat x SUBSET(Message)]
```

```
        ∧ CS = [p ∈ Proc ↦ SUBSET(Message)]
        ∧ ME ∈ {0,1}
        ∧ Canal = [p ∈ Proc ↦ [q ∈ Proc ↦ SUBSET(Message)]]
        ∧ H = SUBSET(Message)

Init ≜  ∧ CurrCP = [p ∈ Proc ↦ <0, {}>]
        ∧ PrevCP = [p ∈ Proc ↦ <0, {}>]
        ∧ CS = [p ∈ Proc ↦ {}]
        ∧ ME = 1
        ∧ Canal = [p ∈ Proc ↦ [q ∈ Proc ↦ {} ]]
        ∧ H = {}

StoreCS(p,m) ≜ CS' = [CS EXCEPT ![p] = @ ∪ { m }]

RemoveCS(p,m) ≜ ∧ m ≠ NoMsg
                ∧ CS' = [CS EXCEPT ![p] = @ \ { m }]

Checkpoint(p,k) ≜ CurrCP' = [CurrCP EXCEPT ![p] = <k, CS[p]> ]

MoveCP(p) ≜ PrevCP' = [PrevCP EXCEPT ![p] = CurrCP[p] ]

Verify(p,m) ≜ ∧ m.itx > CurrCP[p][1] ⇒ Checkpoint(p, m.itx)
              ∧ m.itx ≤ CurrCP[p][1] ⇒ CurrCP' = CurrCP

Send(p, q, m) ≜ ∧ m.tx = p ∧ m.rx = q ∧ p ≠ q
                ∧ m.itx = CurrCP'[p][1]
                ∧ m.irx = NN
                ∧ Canal' = [Canal EXCEPT ![q][p] = @ ∪ { m } ]
                ∧ H' = H ∪ { m }

Receive(p, q, m) ≜ ∧ m.tx = q ∧ m.rx = p ∧ p ≠ q
                   ∧ m ∈ Canal[p][q]
                   ∧ m.irx = CurrCP'[p][1]
                   ∧ Canal' = [Canal EXCEPT ![p][q] = @ \ { m } ]

Replay(p, q, m1, m2) ≜ ∧ m1.tx = q ∧ m1.rx = p
                       ∧ m2.tx = p ∧ m2.rx = q ∧ p ≠ q
                       ∧ m1 ∈ Canal[p][q]
                       ∧ m1.irx = CurrCP'[p][1]
                       ∧ m2.itx = CurrCP'[p][1]
                       ∧ m2.irx = NN
                       ∧ Canal' = [Canal EXCEPT
                                   ![p][q] = @ \ { m1 },
                                   ![q][p] = @ ∪ { m2 } ]
                       ∧ H' = H ∪ { m2 }
```

```
SendApp(p) ≜ ∧ ∃ m1 ∈ Message: ∃ q ∈ Proc:
                ∧ m1.d = "APP"
                ∧ m1.msg = NoMsg
                ∧ Send (p, q, m1)
                ∧ StoreCS(p,m1)
             ∧ UNCHANGED <CurrCP, PrevCP, ME>

SendAppMan ≜ ∧ ∃ m1 ∈ Message: ∃ q ∈ Proc:
                ∧ m1.d = "APP"
```

```
                    ∧ m1.msg = NoMsg
                    ∧ Send (Mngr, q, m1)
                    ∧ StoreCS(Mngr,m1)
              ∧ UNCHANGED <CurrCP, PrevCP, ME>

RxApp(p) ≜  ∧ ∃ m1 ∈ Message: ∃ q ∈ Proc:
                  ∧ Verify(p,m1)
                  ∧ m1.d = "APP"
                  ∧ Deliver(p,m1)
                  ∧ ∃ m2 ∈ Message:
                        ∧ m2.d = "ACK"
                        ∧ m2.msg = m1
                        ∧ Replay (p,q,m1,m2)
            ∧ UNCHANGED <PrevCP, CS, ME>

RxAppMan ≜  ∧ ∃ m1 ∈ Message: ∃ q ∈ Proc:
                  ∧ m1.itx ≤ CurrCP[Mngr][1]
                  ∧ m1.d = "APP"
                  ∧ Deliver(p,m1)
                  ∧ ∃ m2 ∈ Message:
                        ∧ m2.d = "ACK"
                        ∧ m2.msg = m1
                        ∧ Replay (Mngr,q,m1,m2)
            ∧ UNCHANGED <CurrCP, PrevCP, CS, ME>

RxAck(p) ≜  ∧ ∃ m1 ∈ Message: ∃ q ∈ Proc:
                  ∧ Receive (p,q,m1)
                  ∧ Verify(p,m1)
                  ∧ m1.d = "ACK"
                  ∧ m1.msg.tx = p ∧ m1.msg.rx = q
                  ∧ RemoveCS(p,m1.msg)
            ∧ UNCHANGED <PrevCP, ME, H>

RxAckMan ≜  ∧ ∃ m1 ∈ Message: ∃ q ∈ Proc:
                  ∧ Receive (Mngr,q,m1)
                  ∧ m1.itx ≤ CurrCP[Mngr][1]
                  ∧ m1.d = "ACK"
                  ∧ m1.msg.tx = Mngr ∧ m1.msg.rx = q
                  ∧ RemoveCS(Mngr,m1.msg)
            ∧ UNCHANGED <CurrCP, PrevCP, ME, H>

RxReq(p) ≜  ∧ ∃ m1 ∈ Message:
                  ∧ Verify(p,m1)
                  ∧ m1.d = "REQ"
                  ∧ ∃ m2 ∈ Message:
                        ∧ m2.d = "AREQ"
                        ∧ m2.msg = NoMsg
                        ∧ Replay(p, Mngr, m1, m2)
            ∧ UNCHANGED <PrevCP, CS, ME>

SendReq ≜  ∧ StartNewCP ∧ ME=1
           ∧ CurrCP[Mngr][1] = PrevCP[Mngr][1]
           ∧ ∀ q ∈ ProcS: ∀ m ∈ Message:
                 m ∈ Canal[q][Mngr] ⇒ m.d = "APP" ∨ m.d = "ACK"
```

```
            ∧ Checkpoint(Mngr, CurrCP[Mngr][1]+1)
            ∧ ∀ q ∈ ProcS: ∃ m1 ∈ Message:
                   ∧ m1.d = "REQ"
                   ∧ m1.msg = NoMsg
                   ∧ Send (Mngr, q, m1)
            ∧ ME′ = 0
            ∧ UNCHANGED <PrevCP, CS>

RxAckReq ≜  ∧ ME = 0
            ∧ CurrCP[Mngr][1] > PrevCP[Mngr][1]
            ∧ ∀ q ∈ ProcS: ∃ m1 ∈ Message:
                   ∧ m1.itx = CurrCP[Mngr][1]
                   ∧ m1.d = "AREQ"
                   ∧ ∃ m2 ∈ Message:
                          ∧ m2.d = "CMT"
                          ∧ m2.msg = NoMsg
                          ∧ Replay (Mngr, q, m1, m2)
            ∧ MoveCP(Mngr)
            ∧ UNCHANGED <CurrCP, CS, ME>

RxAckCmt ≜  ∧ ME = 0
            ∧ CurrCP[Mngr] = PrevCP[Mngr]
            ∧ ∀ q ∈ ProcS: ∃ m1 ∈ Message:
                   ∧ m1.itx = CurrCP[Mngr][1]
                   ∧ m1.d = "ACMT"
                   ∧ Receive (Mngr,q,m1)
            ∧ ME′ = 1
            ∧ UNCHANGED <CurrCP, PrevCP, CS, H>

RxCmt(p) ≜  ∧ ∃ m1 ∈ Message:
                   ∧ m1.itx = CurrCP[p][1]
                   ∧ m1.d = "CMT"
                   ∧ ∃ m2 ∈ Message:
                          ∧ m2.d = "ACMT"
                          ∧ m2.msg = NoMsg
                          ∧ Replay (p, Mngr, m1, m2)
            ∧ CurrCP[p][1] > PrevCP[p][1]
            ∧ MoveCP(p)
            ∧ UNCHANGED <CurrCP, CS, ME>

Manager ≜  ∨ SendAppMan ∨ RxAppMan ∨ RxAckMan
           ∨ SendReq ∨ RxAckReq ∨ RxAckCmt

Slave(p) ≜  ∨ SendApp(p) ∨ RxApp(p) ∨ RxAck(p)
            ∨ RxReq(p) ∨ RxCmt(p)

Next ≜ Manager ∨ ∃ p ∈ ProcS: Slave(p)

Recovery ≜ Init ∧ □ [Next]_w
```

```
LOCAL

E(S) ≜ ∀ p,q ∈ Proc: S[p][1] = S[q][1]

MB(x,y,S) = ∧ x.itx < S[x.tx][1]
```

```
                ∧ x.irx < S[x.rx][1]
                ∧ y.itx < S[x.rx][1]
                ∧ y.irx < S[x.tx][1]

MA(x,y,S) = ∧ x.itx ≥ S[x.tx][1]
            ∧ x.irx ≥ S[x.rx][1]
            ∧ y.itx ≥ S[x.rx][1]
            ∧ y.irx ≥ S[x.tx][1]

MK(x,y,S) = ∧ x.itx < S[x.tx][1]
            ∧ y.itx ≥ S[x.rx][1]
            ∧ y.irx ≥ S[x.tx][1]
            ∧ x ∈ S[x.tx][2]

MP(x,y,S) = ∧ x.itx < S[x.tx][1]
            ∧ x.irx < S[x.rx][1]
            ∧ y.irx ≥ S[x.tx][1]
            ∧ x ∈ S[x.tx][2]

M(x,y) = ∧ (x.d = "APP")
         ∧ (y.d = "ACK")
         ∧ (x.tx = y.rx)
         ∧ (x.rx = y.tx)
         ∧ (y.msg = x)

CC(x,y,S) = MB(x,y,S) ∨ MA(x,y,S) ∨ MK(x,y,S) ∨ MP(x,y,S)

MH(H,S) ≜ ∀ x,y ∈ Message: x ∈ H ∧ y ∈ H ∧ M(x,y) ⇒ CC(x,y,S)

Com(H,S) ≜ E(S) ∧ MH(H,S)

Consistent ≜ Com(H,CurrCP) ∨ Com(H,PrevCP)
```

```
THEOREM
       Recovery ⇒ □ Consistent
```

Toda especificação em TLA pode ser dividida em três partes: a definição das variáveis e premissas, a definição das ações e os teoremas a serem demonstrados. A seguir, é dedicada uma subseção para cada uma destes partes.

6.2.1 Variáveis e constantes da especificação

São definidas três constantes (variáveis rígidas):

- *Proc*: conjunto de todos os identificadores dos processos que formam o sistema distribuído;
- *Mngr*: identificador do processo chamado de coordenador, gerente ou iniciador;
- *Deliver*: função de entrega das mensagens.

O processo *Mngr* é responsável por iniciar o procedimento de estabelecimento dos pontos de recuperação. Este processo, por sua vez, faz parte do conjunto dos processos que formam o sistema distribuído (*Proc*).

O algoritmo proposto foi projetado para operar como suporte à aplicação. Desta forma, é necessário uma interface através da qual as mensagens possam ser entregues. A função *Deliver* modela esta interface. Entretanto, não é objetivo desta especificação detalhar como é feita a entrega.

São definidas seis variáveis (variáveis flexíveis):

- *CurrCP*: conjunto dos últimos pontos de recuperação estabelecidos;
- *PrevCP*: conjunto dos pontos de recuperação anteriores;
- *CS*: estado dos canais;
- *ME*: estado do coordenador;
- *Canal*: canais de comunicação;
- H: histórico de mensagens transmitidas.

No conjunto *CurrCP*, estão registrados os últimos pontos de recuperação estabelecidos por cada processo. Pela definição de *CurrCP*, cada processo tem associada uma dupla formada por um número natural (o índice do ponto de recuperação) e um conjunto de mensagens (onde será armazenado o estado do canal).

O conjunto *PrevCP* possui a mesma estrutura de *CurrCP*. Sua função é garantir a existência de uma linha de recuperação, enquanto uma nova está sendo estabelecida em *CurrCP*.

Faz parte do algoritmo o salvamento do estado do canal (ou mensagens potencialmente perdidas). Como não é possível ler o estado do canal, é necessário usar uma variável que represente este estado. Para isso, foi introduzida na especificação a variável *CS*. Esta variável guarda as mensagens transmitidas por um processo e cuja resposta não foi recebida. Cada processo possui a sua própria memória, não acessível pelos outros processos (conforme modelo de sistema distribuído). A cada processo corresponde um elemento da variável *CS*.

ME é uma variável auxiliar que indica o estado do coordenador (processo *Mngr*). Indica se o estabelecimento de um conjunto de pontos de recuperação (ou linha de recuperação) está em andamento ou não. Esta variável pode receber os valores 0 ou 1 (números Naturais) e sua manipulação é de responsabilidade única do coordenador. Esta variável faz parte da condição de habilitação de início do estabelecimento de um novo ponto de recuperação.

A variável *Canal* modela os canais de comunicação necessários à troca de mensagens entre processos. Os canais modelados são do tipo unidirecional: a cada par de processos estão associados dois conjuntos componentes de *Canal*. Portanto, esta variável corresponde a uma matriz de conjuntos onde as mensagens podem ser colocadas (na transmissão) ou removidas (na recepção). Apesar de existir um conjunto componente de *Canal* que possibilita um processo enviar mensagens para si próprio, a forma da especificação não o permite.

Como o algoritmo alcança seu objetivo através do controle das mensagens, é necessário representar aquelas que o algoritmo permite que sejam geradas. Para isso, foi introduzida a variável H. Esta variável é um conjunto no qual estão todas as mensagens geradas. Desta forma, para verificar se o algoritmo só permite mensagens consistentes, basta verificar se as mensagens contidas em H são consistentes. Para

que isso seja verdadeiro, mensagens colocadas em H não podem ser removidas. Isso pode ser facilmente verificado pois, na especificação, não existem ações que removam mensagens de H.

Um comentário adicional sobre a variável H é que ela não tem significado prático do ponto de vista da implementação, podendo ser eliminada posteriormente. Sua atuação é exclusiva nas provas de correção.

Na seqüência da especificação são estabelecidas algumas premissas (ASSUME, na especificação) :

- **w**: n-upla das variáveis da descrição, no formato de TLA;
- *MsgType*: conjunto dos tipos de mensagens necessários para a operação do algoritmo;
- *NN*: valor que indica o último ponto de recuperação estabelecido no processo receptor de uma mensagem que ainda não chegou ao seu destino. Este valor será, sempre, maior do que qualquer índice que possa ser atribuído a um ponto de recuperação;
- *NoMsg*: mensagens não tratadas pelo algoritmo;
- *Message*: definição do conjunto das mensagens válidas;
- *ProcS*: conjunto de todos os identificadores de processo exceto o coordenador (*Mngr*).

Uma mensagem que está sendo transmitida não tem identificador de intervalo de ponto de recuperação do receptor. Entretanto, para poder falar sobre a consistência destas mensagens, é necessário que exista alguma forma de comparação entre estas e o instante de estabelecimento dos pontos de recuperação. Assim, estas mensagens tiveram o seu instante potencial de recebimento modelado por *NN*. Sua definição visa capturar a idéia de tempo futuro: algo que poderá vir a ocorrer em um tempo no futuro, mas que não se pode determinar quando.

O algoritmo proposto garante a consistência do sistema para os tipos de mensagens definidos. Se ocorrer uma mensagem de tipo diferente, não é possível garantir a consistência. Estas mensagens não definidas são representadas por *NoMsg*. Sua função é dupla: na especificação, visa indicar valores de mensagens não válidos (e que, portanto, não necessitam de tratamento), e nas provas, visa garantir que determinadas propriedades são características de todas as mensagens válidas.

O Conjunto *Message* representa todas as mensagens que podem trafegar nos canais, não importando se satisfazem os critérios de consistência ou não. Os elementos deste conjunto são formados por seis campos, descritos a seguir:

- *tx*: identificador do processo transmissor da mensagem;
- *rx*: identificador do processo destino da mensagem;
- *itx*: índice do último ponto de recuperação estabelecido pelo processo transmissor da mensagem, no momento de sua transmissão;
- *irx*: índice do último ponto de recuperação estabelecido pelo processo receptor da mensagem, no momento de sua recepção;

- d: identifica o tipo da mensagem. Deve ser um elemento do conjunto *MsgType*;
- *msg*: mensagem que está sendo respondida.

Os identificadores dos processos transmissor e receptor das mensagens devem fazer parte do conjunto *Proc*.

Os índices dos pontos de recuperação devem pertencer ao conjunto dos *Naturais* (definido na TLA). Além disso, o índice de recepção pode receber o valor *NN*.

O campo *msg* deverá carregar a mensagem da aplicação (do tipo **APP**) que está sendo respondida. Assim, este campo receberá valores válidos somente quando a mensagem for do tipo **ACK**. Nos outros tipos de mensagens este campo deverá conter *NoMsg*.

6.2.2 Ações da especificação

A forma como as ações serão descritas segue uma estrutura *top-down*. Assim, a descrição inicia pela fórmula de mais alto nível, *Recovery*, seguindo as fórmulas componentes, as subcomponentes destas, e assim por diante.

A fórmula *Recovery* apresenta o formato padrão para fórmulas TLA. É composta por uma fórmula de inicialização, *Init*, e por uma fórmula que determina os próximos estados possíveis, $\Box[Next]_w$.

A fórmula *Init* inicializa todas as variáveis.

Como as ações do coordenador e dos outros processos apresentam particularidades, a fórmula *Next* indica que uma ação pode ocorrer no coordenador (fórmula *Manager*) ou nos outros processos (fórmula *Slave*).

Das fórmulas que representam o comportamento do coordenador, três delas possuem equivalente nos outros processos. São as fórmulas:

- que representa a transmissão de uma mensagem da aplicação;
- a que representa a recepção de uma mensagem da aplicação com a conseqüente transmissão da mensagem de reconhecimento;
- e a fórmula que representa a recepção de uma mensagem de reconhecimento.

Estas três fórmulas correspondem a *SendAppMan*, *RxAppMan* e *RxAckMan*, para o coordenador, e *SendApp*, *RxApp* e *RxAck*, para os outros processos.

As outras fórmulas correspondem a ações específicas de cada tipo de processo. São ações relacionadas com o estabelecimento dos pontos de recuperação e à coleta de lixo. No caso de coordenador, estas ações estão representadas pelas fórmulas *SendReq*, *RxAckReq* e *RxAckCmt*, que correspondem, respectivamente, às seguintes operações:

- envio de solicitação para o estabelecimento de um novo ponto de recuperação;
- recepção das respostas de estabelecimento de novo ponto de recuperação, de todos os outros processos do sistema, e envio da confirmação da operação, sinalizando que os processos podem iniciar a coleta de lixo;
- recepção das respostas de confirmação de operação e término do estabelecimento da linha de recuperação.

Para cada ação do coordenador existe uma ação de resposta nos outros processos, de maneira a completar o estabelecimento da nova linha de recuperação. Estas ações são representadas pelas fórmulas *RxReq* e *RxCmt*. Correspondem, respectivamente, às seguintes operações:

- recebimento da solicitação de novo ponto de recuperação e resposta a esta solicitação;
- recebimento da confirmação de operação e início da coleta de lixo e envio da resposta correspondente.

As ações descritas utilizam fórmulas específicas para modelar ações de uso comum. Estas ações estão relacionadas com a comunicação: *Send*, *Receive* e *Replay*; com o controle e estabelecimento dos pontos de recuperação: *Verify*, *Checkpoint* e *MoveCP*; e com a gerência da memória que contém o estado dos canais de saída: *StoreCS* e *RemoveCS*.

A ação de *Send* modela a transmissão de uma mensagem. A mensagem deve conter o identificador dos processos transmissor e receptor e estes não podem ser iguais. A mensagem deve conter os índices dos pontos de recuperação do transmissor e do receptor. Entretanto, como a mensagem ainda não foi recebida, o índice do receptor deverá ser *NN*. A mensagem, então, será colocada no *Canal* e no histórico (variável H) de processamento.

Na ação de *Receive*, tem-se a remoção de uma mensagem do *Canal*. Para isso, a mensagem deverá estar presente no mesmo. Da mesma forma que *Send*, esta mensagem deve conter o identificador dos processos transmissor e receptor. Os processos transmissor e receptor não podem ser os mesmos e o registro do intervalo de recepção deve ser aquele do processo receptor.

A ação de *Replay* é uma mescla das ações de *Send* e *Receive*. Corresponde ao recebimento de uma mensagem com a correspondente transmissão da resposta.

A primeira das ações de estabelecimento de pontos de recuperação é *Verify*. Esta ação tem por objetivo verificar se o recebimento de uma mensagem deve forçar o estabelecimento de novo ponto de recuperação local ou não. Esta decisão dependerá do índice do ponto de recuperação do processo transmissor (que a mensagem carrega) e do índice do ponto de recuperação do processo receptor da mesma.

As duas outras ações correspondem ao estabelecimento de um ponto de recuperação local (*Checkpoint*) e a cópia do ponto de recuperação atual, existente em *CurrCP*, para o ponto de recuperação anterior, *PrevCP*, de maneira a eliminar-se informações inúteis. Esta cópia é modelada na ação *MoveCP*.

Os processos devem manter o estado dos canais de saída para uma possível recuperação. Esta informação é gravada nos pontos de recuperação (vide ação *Checkpoint*). Desta forma, é necessário que as mensagens sejam colocadas no estado do canal, quando forem transmitidas, e removidas deste, quando for recebido o reconhecimento correspondente. Estas duas operações são modeladas, respectivamente, pelas ações *StoreCS* e *RemoveCS*.

Agora pode-se descrever as ações que modelam a operação do algoritmo. As primeiras ações descritas a seguir são aquelas relacionadas com o envio e recepção das mensagens da aplicação e sua resposta.

As ações *SendApp* e *SendAppMan* modelam a transmissão de uma mensagem do tipo **APP**. Esta mensagem é transmitida usando a ação *Send*. Além disso, é colocada no estado dos canais de saída do processo transmissor (*CS*).

As ações *RxApp* e *RxAppMan* modelam a recepção de uma mensagem do tipo **APP**. Fazem parte desta ação verificar a necessidade de estabelecimento de novo ponto de recuperação (através da ação *Verify*) e a entrega da mensagem recebida (função *Deliver*). A esta mensagem, o receptor responde com uma mensagem do tipo **ACK** que carrega a mensagem da aplicação. Na realidade, não haveria necessidade de que as mensagens de resposta carregassem toda a mensagem da aplicação correspondente. Poder-se-ia recorrer ao uso de identificadores de mensagens. Entretanto, para fins de especificação e prova, esta forma de trabalhar é mais simples.

No recebimento da resposta das mensagens da aplicação, o processo transmissor da aplicação verifica se deve estabelecer novo ponto de recuperação (*Verify*) e remove a mensagem da aplicação correspondente do estado dos canais.

As próximas ações descritas são aquelas relacionadas diretamente com o estabelecimento dos pontos de recuperação. Estas serão descritas na ordem que ocorrem, durante o estabelecimento de um novo ponto de recuperação.

Para iniciar um novo ponto de recuperação, o coordenador executa uma ação de *SendReq*. Para isso, algumas condições devem ser satisfeitas. Os pontos de recuperação não podem ser ativados a qualquer hora. É necessário uma disciplina de tempo, para evitar um consumo não planejado dos recursos computacionais. Assim, o início de um novo ponto de recuperação está sujeito à função *StartNewCP*, que modela o desejo do sistema iniciar um novo ponto de recuperação. Em geral, esta função tornar-se-á verdadeira periodicamente.

A segunda condição diz respeito à variável *ME*. Se o sistema já se encontrar em operação de estabelecimento de um ponto de recuperação $ME = 0$, então não poderá iniciar um novo.

Para garantir a consistência, o ponto de recuperação atual (*CurrCP*) e o anterior (*PrevCP*) devem ter o mesmo índice. Isso significa que são iguais e que, portanto, estão consistentes com a operação esperada do algoritmo. Esta afirmativa será provada no invariante de número 8: se dois pontos de recuperação, em um mesmo processo, possuírem o mesmo índice, então seus conteúdos serão iguais.

Finalmente, para que um novo ponto de recuperação possa ser iniciado, não pode haver mensagens relacionadas ao estabelecimento de pontos de recuperação (**REQ**, **AREQ**, **CMT**, **ACMT**) nos canais: só podem estar trafegando nos canais mensagens do tipo **APP** e **ACK**.

Se todas as condições forem satisfeitas, então o coordenador salva um novo ponto de recuperação (ação *Checkpoint*) e envia uma mensagem do tipo **REQ** para todos os outros processos. Adicionalmente, a variável *ME* recebe o valor zero, indicando que o estabelecimento de um novo ponto de recuperação está em andamento.

Todos os processos, exceto o coordenador, recebem a mensagem de requisição de novo ponto de recuperação (**REQ**). O processo receptor, então, verifica se deve estabelecer novo ponto de recuperação (através de *Verify*) e responde com uma mensagem do tipo **AREQ**.

Na ação *RxAckReq*, o coordenador receberá as respostas de todos os outros processos. Então, enviará mensagens de confirmação da operação (mensagens do tipo **CMT**) para todos os outros processos. Além disso, copiará o conteúdo do último ponto de recuperação estabelecido (*CurrCP*) no ponto de recuperação anterior (*PrevCP*). Com isso, inicia-se a fase de coleta de lixo.

Todos os processos que receberem a mensagem **CMT** (ação *RxCmt*) copiam seu último estado armazenado (*CurrCP*) no anterior (*PrevCP*), seguindo o coorde-

nador. Além disso, respondem com uma mensagem do tipo **ACMT**.

Quando o coordenador receber todas as respostas **ACMT**, saberá que a coleta de lixo foi completada e poderá encerrar a operação de estabelecimento do ponto de recuperação. Isso é detectado na ação *RxAckCmt*.

6.2.3 Teorema da consistência

Faz parte desta subseção a definição do que será usado como critério de consistência, a ser garantido pelo algoritmo. Este teorema está explicitado pela fórmula $Recovery \Rightarrow \Box Consistent$.

O estudo das mensagens e suas características, visando estabelecer se são ou não consistentes, está descrito nas seções 1.3 e 6.1. Aqui, serão utilizadas as fórmulas desenvolvidas.

A fórmula *Consistent* é definida através da consistência do conjunto *CurrCP*, no contexto das mensagens geradas (H), **ou** através da consistência do conjunto *PrevCP*, no mesmo contexto. Nesta fórmula aparece a condição para que o algoritmo garanta a consistência da operação: deve existir sempre, pelo menos, uma linha de recuperação.

A fórmula da consistência de um conjunto, no contexto das mensagens geradas, $Com(H, S)$, é definida pela conjunção de duas outras: $E(S)$ e $MH(H, S)$ (conforme definido na seção 6.1). Estas duas fórmulas expressam os conceitos de consistência, usando as variáveis da especificação.

A fórmula $E(S)$ representa a necessidade de que todos os pontos de recuperação que formam o conjunto S tenham o mesmo índice. O índice dos pontos de recuperação são números naturais gerados pelo algoritmo. Assim, esta característica deve ser garantida pelo próprio algoritmo.

A fórmula $MH(H, S)$ diz respeito à relação entre as mensagens e os pontos de recuperação estabelecidos. Além disso, relaciona dois tipos de mensagens: as mensagens enviadas pela aplicação e as mensagens de resposta (*acknowledgement*). A definição diz que, para todos os pares de mensagens geradas, se uma for mensagem da aplicação e a outra, a sua resposta (conforme definição de $M(x, y)$), então estas deverão obedecer aos critérios de consistência ($CC(x, y, S)$).

Os critérios de consistência indicam como deve estar relacionado um par de mensagens (x e y), com relação aos pontos de recuperação considerados (contidos em S). São quatro as possibilidades:

- $MB(x, y, S)$: a transmissão (itx) e a recepção(irx) do par de mensagens ocorreram **antes** do estabelecimento dos pontos de recuperação;

- $MA(x, y, S)$: a transmissão (itx) e a recepção(irx) do par de mensagens ocorreram **após** o estabelecimento dos pontos de recuperação;

- $MK(x, y, S)$: a transmissão da mensagem da aplicação ocorreu antes do ponto de recuperação, a mensagem de resposta foi transmitida e recebida após os pontos de recuperação e a mensagem da aplicação está armazenada no ponto de recuperação do seu transmissor;

- $MP(x, y, S)$: a transmissão e a recepção da mensagem da aplicação ocorreram antes dos pontos de recuperação, a recepção da mensagem de resposta ocorreu

após o ponto de recuperação e a mensagem da aplicação está armazenada no ponto de recuperação do seu transmissor.

Em suas definições, as fórmulas anteriores usam os operadores *Antes* e *Apos*. Estes, são empregados de maneira a estabelecer o relacionamento de antecedência entre os eventos associados às mensagens (transmissão e recepção) e o estabelecimento dos pontos de recuperação. É necessário, portanto, detalhar estes operadores.

O operador $Antes(x, C_i)$ indica que o índice x é menor do que o índice associado ao ponto de recuperação C_i, enquanto que o operador $Apos(x, C_i$ indica que o índice x é maior ou igual que aquele associado ao ponto de recuperação C_i. Assim, serão usadas as seguintes definições:

$$Antes(x, C_i) \quad \triangleq \quad x < Indice(C_i) \tag{6.20}$$

$$Apos(x, C_i \quad \triangleq \quad x \geq Indice(C_i) \tag{6.21}$$

onde $Indice(C_i)$ corresponde ao índice associado ao ponto de recuperação C_i. Estas definições foram transformadas para fórmulas TLA, de acordo com a definição das variáveis que correspondem a pontos de recuperação (vide *CurrCP* e *PrevCP* na seção 6.2.1 e as fórmulas do critério de consistência apresentadas na especificação.

Então, de acordo com o discutido, é necessário provar que sempre existirá uma linha de recuperação. Esta linha poderá estar registrada em *CurrCP* ou *PrevCP*. Assim, deve-se provar que a propriedade *Consistent* é um invariante.

A prova do invariante da consistência está descrita na seção 6.3.

6.3 Prova formal

Nas subseções que seguem, serão apresentadas algumas premissas, regras e conceitos utilizados para escrever a prova de correção do algoritmo, além da prova propriamente dita.

As provas não serão apresentadas por completo no texto, estando detalhadas no relatório técnico [CEC 98]. Entretanto, serão apresentados o início da prova bem como a discussão dos pontos mais importantes.

6.3.1 Formato da seqüência de induções da prova

Uma prova formal, em geral, é feita pela demonstração de uma seqüência de induções que partem das premissas e chegam à fórmula que se deseja demonstrar. Entretanto, durante o desenvolvimento das provas, é comum pensar de forma inversa. Ou seja, a partir da fórmula que se deseja provar, pensa-se quais as fórmulas que poderiam ser usadas como premissas para a sua obtenção.

Com base nesta observação prática de como as provas são desenvolvidas, Lamport propôs um refinamento da dedução natural, chamado de **prova inversa**, onde escreve-se as expressões que são necessárias para provar uma fórmula dada. Como resultado, obtém-se uma prova estruturada em árvore, uma vez que determinada expressão pode requerer mais de uma fórmula para a sua prova.

As provas desenvolvidas para o algoritmo proposto utilizam, em geral, a prova inversa. Entretanto, algumas vezes é mais interessante aplicar a dedução natural. Quando nada for dito, deverá ser entendido que a prova está sendo apresentada na

ordem inversa. Caso contrário, será explicitado que a prova está sendo apresentada como uma seqüência de induções (prova por dedução natural).

EXEMPLO 1: Da lógica, sabe-se que $A \Rightarrow A \vee B$. Esta fórmula é equivalente a um passo de prova onde, a partir da fórmula A, pode-se deduzir a fórmula $A \vee B$. No caso da prova inversa, para provar a fórmula $A \Rightarrow A \vee B$, deve-se provar a fórmula A. Representa-se as duas fórmulas em linhas separadas, da seguinte forma:

$$\begin{array}{c} A \vee B \\ A \end{array} \tag{6.22}$$

Duas passagens são muito comuns, quando representando provas na forma inversa. Estas passages serão chamadas de **regra *and*** e **regra *or***, e também serão apresentadas através de exemplos.

EXEMPLO 2 – Regra *and*: Este passo de prova inversa corresponde à indução $(A \Rightarrow C) \Rightarrow ((A \wedge B) \Rightarrow C)$. Quando representando-a na forma inversa, a prova será composta pelas seguintes linhas:

$$\begin{array}{c} (A \wedge B) \Rightarrow C \\ A \Rightarrow C \end{array} \tag{6.23}$$

Pode-se raciocinar da seguinte forma: para provar que $(A \wedge B) \Rightarrow C$ é verdadeiro, basta provar que $A \Rightarrow C$ é verdadeiro. Ou ainda, sabendo-se que $A \wedge B \Rightarrow A$, basta mostrar que $A \Rightarrow C$ para provar a seqüência $A \wedge B \Rightarrow A \Rightarrow C$. Note-se que a regra que permite este passo de prova é $A \wedge B \Rightarrow A$. Esta informação será acrescentada às provas através de comentários. Desta forma, tem-se o seguinte:

```
A ∧ B ⇒ C
```

Pela lógica

```
A ⇒ C
```

EXEMPLO 3 – Regra *or*: Este passo de prova inversa corresponde à indução $(A \Rightarrow B) \Rightarrow (A \Rightarrow (B \vee C))$, e será composta pelas seguintes linhas:

$$\begin{array}{c} A \Rightarrow (B \vee C) \\ A \Rightarrow B \end{array} \tag{6.24}$$

Da mesma forma que a regra do *and*, pode-se raciocinar da seguinte forma: sabendo-se que $B \Rightarrow (B \vee C)$, pode-se escrever a seqüência $A \Rightarrow B \Rightarrow (B \vee C)$. Assim, para provar a seqüência toda, basta provar que $A \Rightarrow B$. Pelo mesmo motivo anterior, pode-se escrever:

```
A ⇒ B ∨ C
```

Pela lógica

```
A ⇒ B
```

6.3.2 Utilização da regra INV1, da TLA

Para provar que uma determinada propriedade é invariante em uma descrição TLA, em geral, é necessário provar a seguinte indução:

$$Init \wedge \Box[Next]_w \Rightarrow \Box Ivar \tag{6.25}$$

Na expressão 6.25, *Ivar* representa a propriedade invariante a ser provada.

Usando a forma inversa de prova, pode-se dizer que, para provar a fórmula 6.25, devem ser provadas as seguintes duas fórmulas:

$$\begin{aligned} Init &\Rightarrow Ivar && (6.26)\\ Ivar \wedge \Box[Next]_w &\Rightarrow \Box Ivar && (6.27) \end{aligned}$$

Uma vez provada a fórmula 6.26, pode-se escrever a seguinte indução:

$$(Init \wedge \Box[Next]_w \Rightarrow \Box Ivar) \Rightarrow (Ivar \wedge \Box[Next]_w \Rightarrow \Box Ivar) \quad (6.28)$$

Uma vez provada a fórmula 6.26, restará provar a fórmula 6.27. Para isso, será utilizada a regra de prova INV1 da TLA. Esta regra é definida da seguinte forma:

$$\frac{I \wedge [N]_f \Rightarrow I'}{I \wedge \Box[N]_f \Rightarrow \Box I} \quad (6.29)$$

Assim, aplicando-se a regra de prova INV1 da TLA, a prova da expressão 6.27 será reduzida à prova da seguinte fórmula:

$$Ivar \wedge [Next]_w \Rightarrow Ivar' \quad (6.30)$$

Em resumo, a prova de um invariante de uma especificação TLA, ou seja, a prova da fórmula 6.25 reduz-se à prova das seguintes duas fórmulas:

$$\begin{aligned} Init &\Rightarrow Ivar\\ Ivar \wedge [Next]_w &\Rightarrow Ivar' \end{aligned} \quad (6.31)$$

Todas as provas de invariantes que serão apresentadas usam, no seu início, a redução representada pelas fórmulas 6.31.

6.3.3 Introdução à prova do invariante *Consistent*

Para provar a correção do algoritmo é necessário provar o teorema $Recovery \Rightarrow \Box Consistent$. Entretanto, conforme apresentado na prova, foi necessário o uso de outros invariantes, que tiveram de ser provados. Assim, o teorema a ser provado é:

$$Recovery \Rightarrow \Box Ivars, \quad (6.32)$$

onde *Ivars* é a conjunção de todos os invariantes.

Note-se que uma das fórmulas componentes da fórmula *Ivars* é o invariante *Consistent*, a qual expressa os critérios a serem alcançados de maneira que o sistema esteja, sempre, em estado consistente. Assim, o início da prova de correção é o seguinte:

PROOF Recovery ⇒ □ Ivars

```
LOCAL
  Ivars ≜ I1 ∧ I2 ∧ ... ∧ In
THEOREM
  Recovery ⇒ □ Ivars
```

```
<0>1. Recovery ⇒ □ Ivars

  Definição de Recovery
  Init ∧ □[Next]_w ⇒ □ Ivars

  TLA INV1
  1. Init ⇒ Ivars
  2. Ivars ∧ [Next]_w ⇒ Ivars'

<1>1. Init ⇒ Ivars

  Definição de Ivars
  Init ⇒ I1 ∧ I2 ∧ ... ∧ In

  Provar:
  1. Init ⇒ I1
  2. Init ⇒ I2
  ...
  n. Init ⇒ In

<1>2. Ivars ∧ [Next]_w ⇒ Ivars'

  Definição de Ivars
  I1 ∧ I2 ∧ ... ∧ In ∧ [Next]_w ⇒ I1' ∧ I2' ∧ ... ∧ In'

  Provar:
  1. I1 ∧ I2 ∧ ... ∧ In ∧ [Next]_w ⇒ I1'
  2. I1 ∧ I2 ∧ ... ∧ In ∧ [Next]_w ⇒ I2'
  ...
  n. I1 ∧ I2 ∧ ... ∧ In ∧ [Next]_w ⇒ In'
```

De acordo com o exposto na expressão 6.32, definiu-se *Ivars* como a conjunção dos vários invariantes a serem provados. Assim, para possibilitar a identificação dos invariantes, cada um recebeu um número, o que levou à fórmula 6.33.

$$Ivars \triangleq I1 \wedge I2 \wedge ... \wedge In \tag{6.33}$$

A forma inversa de prova apresenta outra característica interessante: a estrutura em árvore. Isso é devido a que, para provar uma fórmula, pode ser necessário provar mais de uma fórmula. Portanto, é preciso indicar esta estrutura, ou seja, deve-se indicar:

- o nível na hierarquia;
- o ramo da árvore.

O nível da hierarquia é representado entre os sinais $<$ e $>$, sendo seguido por um número que indica o ramo da árvore. A raiz da árvore tem nível 0. Assim, a primeira fórmula a ser provada recebe o identificador $< 0 > 1$, que corresponde ao teorema 6.32.

A prova do teorema 6.32 utiliza a definição de *Recovery* e, em seguida, a definição de INV1, conforme discutido no item 6.3.2. Assim, para provar o teorema 6.32 deve-se provar as duas fórmulas:

$$\begin{array}{lrcl} <1>1. & Init & \Rightarrow & Ivars \\ <1>2. & Ivars \wedge [Next]_{<w>} & \Rightarrow & Ivars' \end{array} \tag{6.34}$$

Estas duas fórmulas são os ramos da fórmula $< 0 > 1$. Conforme pode ser visto, estas fórmulas estão no nível 1 da árvore, e cada uma recebeu um número seqüencial.

Na prova da fórmula $Init \Rightarrow Ivars$, foi usada a definição de *Ivars*, de maneira que, para prová-la, é necessário provar que *Init* implica em cada um dos invariantes componentes de *Ivars*.

A prova de $Ivars \wedge [Next]_{<w>} \Rightarrow Ivars'$ também usou a definição de *Ivars*, o que levou à necessidade de provar que $Ivars \wedge [Next]_{<w>}$ implica, separadamente, em cada invariante-linha.

Desta forma, para provar que uma determinada fórmula, por exemplo I_i, é um invariante, basta provar as duas fórmulas:

$$\begin{array}{lrcl} 1. & Init & \Rightarrow & I_i \\ 2. & Ivars \wedge [Next]_{<w>} & \Rightarrow & I'_i \end{array} \tag{6.35}$$

Uma característica importante, usada nas provas, foi a possibilidade de criar invariantes mais ou menos à vontade, desde que sua prova também fosse realizada. Obviamente, a escolha das fórmulas invariantes não foi feita ao acaso. Utilizando-se o conhecimento do comportamento do algoritmo, pode-se escrever fórmulas que representam propriedades invariantes.

Finalmente, pode-se observar que o invariante *Consistent*, objetivo principal da prova, receberá um tratamento idêntico ao dos outros invariantes. Assim, exceto por ter sido o primeiro a ser provado, não há necessidade de nenhuma referência diferenciada a ele.

6.3.4 Utilização de outros invariantes

Na subseção 6.3.3, foi apresentado o início da prova. Nesta introdução, mostrou-se que é possível usar os invariantes em qualquer ponto da prova, uma vez que tenham sido provados.

Com esta total liberdade de uso dos invariantes, pode ocorrer de provar-se um invariante, por exemplo I_i, usando o invariante I_j na tarefa e vice-e-versa. Supondo-se que $Ivars \triangleq I_i \wedge I_j$, pode-se verificar que esta dependência circular não causa maiores problemas, uma vez que a prova de cada invariante se reduz a prova das seguintes fórmulas:

$$\begin{array}{rcl} Init & \Rightarrow & I_i \\ I_i \wedge I_j \wedge [Next]_{<w>} & \Rightarrow & I'_i \end{array} \tag{6.36}$$

$$\begin{array}{rcl} Init & \Rightarrow & I_j \\ I_i \wedge I_j \wedge [Next]_{<w>} & \Rightarrow & I'_j \end{array} \tag{6.37}$$

Ou seja, para provar qualquer um dos invariantes não se necessita de nenhuma prova adicional. São necessárias, apenas, as fórmulas dos próprios invariantes.

Entretanto, o mesmo não ocorre com os invariantes-linha. A questão é: pode-se utilizar os invariantes-linha em qualquer passo da prova? A resposta é afirmativa, desde que sejam observados alguns cuidados.

O primeiro cuidado é garantir que o invariante-linha usado já tenha sido provado. Assim, garante-se a sua validade em qualquer estado do comportamento do algoritmo.

O segundo cuidado é não permitir dependências circulares: a prova de invariância de I_j depende da fórmula I'_i e a prova de invariância de I_i depende de I'_j. A questão aqui é que, para provar I_j foi necessária a fórmula I'_i, que por sua vez usou I_i, que depende de I'_j, e que requer I_j, que era o objetivo inicial.

Em algumas provas dos invariantes são usados outros invariantes-linha. Entretanto, não existe dependência circular entre eles.

6.3.5 Prova do Invariante *Consistent*

Levando em consideração o início da prova apresentado na subseção 6.3.3, será apresentada a prova do invariante $I1$. Este invariante é o objetivo da prova de correção, ou seja, é aquele que estabelece que o algoritmo só permite combinações de mensagens e pontos de recuperação que sejam consistentes, no sentido discutido nas seções anteriores.

A prova do invariante inicia estabelecendo a necessidade de serem provadas duas outras fórmulas, conforme estabelecido na subseção 6.3.3.

A prova da **fórmula** $Init => I1$ (fórmula de **inicialização**) será subdividida em duas:

- a prova que todos os pontos de recuperação têm o mesmo índice ($E(CurrCP)$);
- e a prova que todas as mensagens geradas são consistentes ($MH(H, CurrCP)$).

Para provar que todos pontos de recuperação possuem o mesmo índice, basta observar que todos possuem, no início, o índice 0.

A prova da consistência baseia-se no fato que, no início, não existem mensagens geradas ($H = \{\}$). Portanto, todas as mensagens geradas são consistentes.

Para provar a **fórmula** $Ivars \wedge [Next]_{<w>} => I1'$ (fórmula de **progresso**) deve-se separar os *stuttering steps* (passos em que as variáveis relacionadas em w não são alteradas).

Se as variáveis não são alteradas ($w' = w$), então $CurrCP' = CurrCP$ e $PrevCP' = PrevCP$. Desta forma, a consistência é invariante.

Quando ocorrer alguma ação (alguma variável for alterada), a prova também pode ser separada em duas: uma delas, onde o conjunto *CurrCP* é consistente, e outra, em que o conjunto *PrevCP* é consistente. Para cada parte em que foi dividida a prova, deve-se provar que cada ação do coordenador e dos escravos garante a consistência.

Então, levando-se em consideração as ações do coordenador e dos escravos, a prova estrutura-se em uma árvore, onde as folhas correspondem às ações cuja correção deve ser demonstrada.

Em geral, as folhas da árvore de prova, listadas na tabela 6.1, são, novamente, subdivididas em duas: a prova de $E(S)$ e a prova de $MH(H, S)$, onde S corresponde a *CurrCP* ou *PrevCP*.

TABELA 6.1 – Estrutura da Prova do Invariante *Consistent*

$< 3 >$	$2.Consistent \wedge Ivars \wedge Next \Rightarrow Consistent'$			
$< 4 >$	$1.Com(H, CurrCP) \cdots$		$2.Com(H, PrevCP) \cdots$	
$< 5 >$	$1.Manager$	$2.Slave(p)$	$1.Manager$	$2.Slave(p)$
$< 6 >$	$1.SendAppMan$	$1.SendApp(p)$	$1.SendAppMan$	$1.SendApp(p)$
	$2.RxAppMan$	$2.RxApp(p)$	$2.RxAppMan$	$2.RxApp(p)$
	$3.RxAckMan$	$3.RxAck(p)$	$3.RxAckMan$	$3.RxAck(p)$
	$4.SendReq$	$4.RxReq(p)$	$4.SendReq$	$4.RxReq(p)$
	$5.RxAckReq$	$5.RxCmt(p)$	$5.RxAckReq$	$5.RxCmt(p)$
	$6.RxAckCmt$		$6.RxAckCmt$	

Nas ações onde $CurrCP' = CurrCP$, que em geral aparece em um termo $UNCHANGED$, a **prova de** $E(CurrCP)$ é trivial. Basta escrever:

$$E(CurrCP) \wedge CurrCP' = CurrCP \Rightarrow E(CurrCP') \tag{6.38}$$

Este caso ocorre nas ações do coordenador, exceto em $SendReq$. De forma semelhante, também ocorre nas ações dos escravos, exceto aquelas onde há a recepção de mensagens que podem causar o estabelecimento de um novo ponto de recuperação. São as ações $RxApp(p)$, $RxAck(p)$ e $RxReq(p)$.

De forma análoga, as provas envolvendo $E(PrevCP)$ (proveniente de $Com(H, PrevCP)$) podem ser tão simples quanto escrever:

$$E(PrevCP) \wedge PrevCP' = PrevCP \Rightarrow E(PrevCP') \tag{6.39}$$

Isso ocorre nas ações do coordenador, exceto $RxAckReq$, e nas ações dos escravos, exceto $RxCmt(p)$.

Para provar a ação $SendReq$, foi usado o invariante 7.

Para demonstrar a correção das ações $RxApp(p)$, $RxAck(p)$ e $RxReq(p)$, mostrou-se que levam a $\neg E(CurrCP)$. Como por premissa a expressão $E(CurrCP)$ é verdadeira, então o resultado será $E(CurrCP) \wedge \neg E(CurrCP)$, o que é sempre falso. Como de algo falso pode-se deduzir qualquer expressão, completou-se a prova.

Quando $E(PrevCP)$ e $RxAckReq$, buscou-se provar que a ação garante $E(PrevCP')$. Para isso foram usados os invariantes $I5$, $I6$ e $I12$.

Para provar $RxCmt(p)$, foram utilizados os invariantes $I5$, $I6$ e $I13$. A seqüência da prova levou a $\neg E(PrevCP)$. Novamente, pode-se deduzir qualquer fórmula, completando a prova.

Até agora, as provas estiveram relacionadas com a demonstração da fórmula $E(S)$. Para **provar** $M(H, S)$, é preciso separar o problema em duas partes. A primeira parte diz respeito à existência de duas variáveis na fórmula $MH(H, S)$, cujas alterações efetuadas nas ações devem ser verificadas: H e $CurrCP$, quando $S = CurrCP$, e H e $PrevCP$, quando $S = PrevCP$. Logo, não basta verificar apenas o comportamento de $CurrCP$ (ou $PrevCP$). É necessário verificar como a variável H é alterada pelas ações. Ou seja, deve-se garantir que as mensagens colocadas na variável H satisfaçam os critérios de consistência.

A segunda parte diz respeito à não alteração da variável $CurrCP$ (e $PrevCP$). Se esta variável não for alterada, mesmo com a alteração de H, as provas resultantes

são simples. Por outro lado, as provas adquirem maior complexidade, quando a variável H também sofre alterações.

Uma das formas usadas para provar a parte $MH(H, S)$ foi o desmembramento em suas fórmulas componentes:

- $x \notin H$, indica o fato da mensagem x não ter sido gerada;
- $y \notin H$, indica o fato da mensagem y não ter sido gerada;
- $\neg M(x, y)$, indica o fato das mensagens x e y não estarem sujeitas aos critérios de consistência;
- $CC(x, y, CurrCP)$, indica o fato das mensagens x e y estarem sujeitas aos critérios de consistência, em relação ao conjunto $CurrCP$ de pontos de recuperação.

Para cada caso, foi desenvolvida uma prova. Além disso, em vários desenvolvimentos foi necessário subdividir as provas com fórmulas que consideravam, por exemplo, o fato de uma mensagem ser de um tipo, ou não. No caso da ação *SendAppMan*, quando $x \notin H$, foi usada a fórmula:

$$(x = m1) \vee (x \neq m1) \tag{6.40}$$

onde $m1$ corresponde à mensagem enviada pela parte da aplicação localizada no processo coordenador.

Ainda, quando $x \notin H$, foi necessário considerar a geração ou não da mensagem de resposta, representada por y. Para isso, foi usada a fórmula:

$$(y \in H) \vee (y \notin H) \tag{6.41}$$

onde estão representadas as duas possibilidades: que a resposta tenha sido gerada ou não.

Os mesmos comentários válidos no caso da fórmula $x \notin H$ têm seus análogos para a fórmula $y \notin H$.

A fórmula $\neg M(x, y)$, por outro lado, é bastante simples, uma vez que não possui variáveis de estado. Desta forma, o valor $M(x, y)'$ é o mesmo $M(x, y)$, portanto $\neg M(x, y) \Rightarrow \neg M(x, y)'$.

Finalmente, a fórmula $CC(x, y, CurrCP)$ só utiliza a variável $CurrCP$. Desta forma, quando a ação possui $CurrCP' = CurrCP$, a sua prova é bastante simples. Isso ocorre, também, com a fórmula $CC(x, y, PrevCP)$ e $PrevCP' = PrevCP$.

Para completar a prova do invariante *Consistent*, foi necessário escrever **outras fórmulas** que descrevem comportamentos invariantes do sistema. Estas expressões, em geral, são a formalização de comportamentos **óbvios**. Entretanto, sem eles não seria possível completar as provas.

As fórmulas usadas como invariantes para auxiliar na prova do invariante principal estão listadas a seguir:

$$\begin{aligned} I2 \quad &\triangleq \quad \forall\, x, y \in Message : \\ &\qquad M(x, y) \wedge y \in H \Rightarrow y.irx \geq x.itx \wedge y.itx \geq x.irx \\ I3 \quad &\triangleq \quad \forall\, m \in Message : \end{aligned}$$

$$
\begin{array}{rcl}
 & & m \in H \Rightarrow m.itx \leq m.irx \\
I3.1 & \triangleq & \forall\, m \in Message : \\
 & & m \notin H \vee m.itx \leq m.irx \\
I3.2 & \triangleq & \forall\, m \in Message : \\
 & & m.itx > m.irx \Rightarrow m \notin H \\
I4 & \triangleq & \forall\, x, y \in Message : \\
 & & \wedge M(x, y) \wedge y \in H \wedge x.itx < CurrCP[x.tx][1] \\
 & & \wedge y.irx \geq CurrCP[x.tx][1] \\
 & & \Rightarrow x \in CurrCP[x.tx][2] \\
I5 & \triangleq & \forall\, m \in Message : \\
 & & m \in Canal[m.rx][m.tx] \Rightarrow m \in H \\
I6 & \triangleq & \forall\, m \in Message : \\
 & & m \in H \Rightarrow m.itx \leq CurrCP[m.tx][1] \\
I7 & \triangleq & (ME = 1) \Rightarrow E(PrevCP) \\
I8 & \triangleq & \forall\, p \in Proc : \\
 & & CurrCP[p][1] = PrevCP[p][1] \Rightarrow CurrCP[p] = PrevCP[p] \\
I10 & \triangleq & \forall\, p \in Proc : \\
 & & CurrCP[p][1] \geq PrevCP[p][1] \\
I11 & \triangleq & \forall\, m \in\!: Message : \\
 & & m \in CurrCP[m.tx][2] \wedge m.itx < PrevCP[m.tx][1] \\
 & & \Rightarrow m \in PrevCP[m.tx][2] \\
I12 & \triangleq & \forall\, p \in Proc : \\
 & & CurrCP[p][1] \leq CurrCP[Mngr][1] \\
I13 & \triangleq & \forall\, m \in Message : \\
 & & m \in\!: Canal[m.rx][m.tx] \wedge m.d = \text{“}\mathsf{CMT}\text{”} \\
 & & \Rightarrow CurrCP[m.tx][1] = PrevCP[m.tx][1] \\
I14 & \triangleq & \forall\, m \in Message : \\
 & & m.d = \text{“}\mathsf{APP}\text{”} \wedge m \in Canal[m.rx][m.tx] \\
 & & \Rightarrow m \in CS[m.tx] \\
I15 & \triangleq & \forall\, m \in Message : \\
 & & m.itx < CurrCP[m.tx][1] \wedge m \in CS[m.tx] \\
 & & \Rightarrow m \in CurrCP[m.tx][2] \\
I16 & \triangleq & \forall\, m \in Message : \\
 & & m \in H \wedge m.d = \text{“}\mathsf{ACK}\text{”} \wedge m \in Canal[m.rx][m.tx] \\
 & & \Rightarrow m.msg \in CS[m.msg.tx] \\
I17 & \triangleq & \forall\, m \in Message : \\
 & & m \in H \wedge m.d = \text{“}\mathsf{ACK}\text{”} \wedge m \notin Canal[m.rx][m.tx] \\
 & & \Rightarrow m.irx \leq CurrCP[m.rx][1] \\
I18 & \triangleq & \forall\, m \in\!: Message : \\
 & & m \in Canal[m.rx][m.tx] \wedge m.d = \text{“}\mathsf{ACMT}\text{”}
\end{array}
$$

$$
\begin{array}{lcl}
 & & \Rightarrow CurrCP[m.tx][1] = PrevCP[m.tx][1] \\
I21 & \triangleq & \forall\, m \in Message\ : \\
 & & m \in H \wedge m.d = \text{“}\mathsf{CMT}\text{”} \Rightarrow m.tx = Mngr \\
I22 & \triangleq & \forall\, x, y \in Message\ : \\
 & & M(x,y) \wedge y \in Canal[x.tx][x.rx] \\
 & & \Rightarrow x \notin Canal[x.rx][x.tx] \\
I23 & \triangleq & \forall\, x, y \in Message\ : \\
 & & \wedge x \in H \wedge x.d = \text{“}\mathsf{ACK}\text{”} \wedge x.msg \neq :\ NoMsg \\
 & & \wedge y \in H \wedge y.d = \text{“}\mathsf{ACK}\text{”} \wedge y.msg = x.msg \\
 & & \Rightarrow x = y \\
I24 & \triangleq & \forall\, m \in Message\ : \\
 & & m \in H \wedge m.d = \text{“}\mathsf{ACK}\text{”} \\
 & & \Rightarrow m.msg.irx \neq NN \\
I25 & \triangleq & \forall\, m \in Message\ : \\
 & & m \in H \wedge m.d = \text{“}\mathsf{ACK}\text{”} \\
 & & \Rightarrow \left(\begin{array}{l} \wedge m.msg \notin Canal[m.tx][m.rx] \\ \wedge m.msg.tx = m.rx \\ \wedge m.msg.rx = m.tx \end{array} \right) \\
I26 & \triangleq & \forall\, m \in Message\ : \\
 & & m.msg.d = \text{“}\mathsf{APP}\text{”} \wedge m.msg.irx = NN \\
 & & \Rightarrow m \notin H \\
I27 & \triangleq & \forall\, m \in Message\ : \\
 & & m.msg.d \neq \text{“}\mathsf{APP}\text{”} \wedge m.msg \neq NoMsg \\
 & & \Rightarrow m \notin H
\end{array}
$$

A numeração dos invariantes seguiu a ordem de aparecimento de sua necessidade, para a prova do invariante principal. Entretanto, após completada a prova, notou-se que alguns deles eram equivalentes. Por esse motivo, os invariantes de número 9, 19 e 20 foram removidos.

6.3.6 Prova do *framework* de invariantes

Durante o desenvolvimento das provas dos invariantes, observou-se a semelhança existente entre várias delas. Em vista deste fato, definiu-se dois *frameworks* de invariantes. Estes *frameworks* são, também, invariantes, onde aparece um operador que deverá ser definido ($Prop(x, y)$), quando os *frameworks* forem aplicados a um invariante real.

Estes *frameworks* tiveram sua correção comprovada para a maioria dos casos. Entretanto, para cada um, sobraram três fórmulas que não puderam ser verificadas, uma vez que dependem do operador não definido.

Os dois invariantes definidos e provados estão relacionados com as mensagens da aplicação e de resposta. No caso das mensagens da aplicação, o *framework* é chamado de **FAPP** (*framework* **APP**) enquanto que o *framework* para as mensagens de resposta é chamado de **FACK** (*framework* **ACK**).

Estes *frameworks* serão apresentados através de sua fórmula, significado e a prova de correção. Adicionalmente, será listado o conjunto de fórmulas pendentes e que devem ser provadas, quando tiver associado um invariante real.

Assim, se um invariante real tiver a mesma estrutura que uma das duas fórmulas básicas (FAPP ou FACK), bastará provar as fórmulas pendentes para que o invariante esteja provado.

FAPP

Fórmula:

$$\begin{array}{lcl} FAPP & \triangleq & \forall\, x, y \in Message : \\ & & (x \in H \wedge x.d = \text{“}\mathsf{APP}\text{”}) \Rightarrow Prop(x, y) \end{array} \tag{6.42}$$

Significado:

O *framework* estabelece uma disjunção de condições necessárias para concluir $Prop(x, y)$. Neste *framework*, as condições são que a mensagem x tenha sido gerada ($x \in H$) e que seja da aplicação ($x.d =$ “APP”).

Prova:

O desenvolvimento da prova foi feito através do desmembramento da fórmula inicial até alcançar às ações. Dentre estas, apenas aquelas em que a prova é dependente da propriedade $Prop(x, y)$, não puderam ser provadas. Estas três fórmulas pendentes estão listadas a seguir.

Fórmulas Pendentes:

1. $x \notin H \wedge Ivars \wedge SendAppMan$
   $\Rightarrow \quad (x \notin H' \vee x.d \neq \text{“}\mathsf{APP}\text{”} \vee Prop'(x, y))$
2. $x \notin :\ H \wedge Ivars \wedge \exists\, p \in ProcS\ :\ SendApp(p)$
   $\Rightarrow \quad (x \notin :\ H' \vee x.d \neq \text{“}\mathsf{APP}\text{”} \vee Prop'(x, y))$
3. $Prop(x, y) \wedge Ivars \wedge Next$
   $\Rightarrow \quad (x \notin :\ H' \vee x.d \neq \text{“}\mathsf{APP}\text{”} \vee Prop'(x, y))$

FACK

Fórmula:

$$\begin{array}{lcl} FACK & \triangleq & \forall\, x, y \in Message : \\ & & (y \in H \wedge y.d = \text{“}\mathsf{ACK}\text{”}) \Rightarrow Prop(x, y) \end{array} \tag{6.43}$$

Significado:

O *framework* estabelece uma disjunção de condições necessárias para concluir $Prop(x, y)$. Neste *framework*, as condições são que a mensagem y tenha sido gerada ($y \in H$) e que seja de resposta ($y.d =$ “ACK”).

Prova:

O desenvolvimento da prova foi feito através do desmembramento da fórmula inicial até alcançar às ações. Dentre estas, apenas aquelas em que a prova é dependente da propriedade $Prop(x, y)$, não puderam ser provadas. Estas três fórmulas pendentes estão listadas a seguir.

Fórmulas Pendentes:

1. $y \notin H \wedge Ivars \wedge RxAppMan$
   $\Rightarrow \quad (y \notin H' \vee y.d \neq \text{“ACK”} \vee Prop'(x, y))$
2. $y \notin H \wedge Ivars \wedge \exists\, p \in ProcS \,:\, RxApp(p)$
   $\Rightarrow \quad (y \notin H' \vee y.d \neq \text{“ACK”} \vee Prop'(x, y))$
3. $Prop(x, y) \wedge Ivars \wedge Next$
   $\Rightarrow \quad (y \notin H' \vee y.d \neq \text{“ACK”} \vee Prop'(x, y))$

6.3.7 Prova dos invariantes

Nesta seção, serão apresentados os invariantes listados anteriormente, seu significado no sistema e uma indicação de como foi feita sua prova. Conforme descrito na seção 6.3.3, a prova de cada invariante foi separada em duas: a prova da fórmula de inicialização e a prova da fórmula de progresso. Entretanto, as explicações da prova foram direcionadas para o ramo que apresenta maior dificuldade.

Invariante 2

Fórmula:

$$I2 \quad \triangleq \quad \forall\, x, y \in Message \,: \\ M(x, y) \wedge y \in H \Rightarrow y.irx \geq x.itx \wedge y.itx \geq x.irx \tag{6.44}$$

Significado:

Este invariante expressa a relação de causa e efeito existente entre uma mensagem da aplicação e sua resposta. Os eventos de uma mensagem de resposta estão relacionados aos eventos de sua mensagem da aplicação associadas de maneira que a recepção da resposta deve ocorrer no mesmo intervalo de ponto de recuperação ou em intervalo posterior que a transmissão da mensagem da aplicação. De forma semelhante, a transmissão da mensagem de resposta deve ocorrer no mesmo intervalo de ponto de recuperação ou em intervalo posterior que a recepção da mensagem da aplicação.

Na fórmula do invariante está expresso (nas premissas da indução) que a relação de causa e efeito só é mantida para as mensagens x e y que satisfizerem o operador $M(x, y)$[1] e que a mensagem y tenha sido gerada ($y \in H$)

Prova:

Para provar este invariante foi utilizada a prova do invariante genérico FACK, discutida na subseção 6.3.6, onde:

[1] As mensagens x e y satisfazem o operador $M(x, y)$ se as características das mensagens tornam verdadeira a fórmula do operador.

$$Prop(x,y) \quad \triangleq \quad \vee\neg : M(x,y) \\ \vee(y.irx \geq x.itx \wedge y.itx \geq x.irx)$$

Invariante 3

FÓRMULA:

$$I3 \quad \triangleq \quad \forall\, m \in Message : \\ m \in H \Rightarrow m.itx \leq m.irx \tag{6.45}$$

SIGNIFICADO:

O invariante expressa a necessidade que o evento de transmissão de todas as mensagens geradas deva ocorrer em um intervalo de ponto de recuperação igual ou anterior ao de sua recepção. Esta caracterização identifica as mensagens órfãs. Assim, o invariante expressa que não podem ser geradas mensagens órfãs.

PROVA:

A prova do invariante 3 é a mesma para os invariantes 3.1 e 3.2, uma vez que estes dois últimos correspondem a manipulações do primeiro.

Com o desmembramento da fórmula do invariante chega-se à necessidade de provar que uma mensagem não poderá ser gerada ($m \notin H'$) ou deverá ter o evento de transmissão no mesmo intervalo ou em intervalo anterior de ponto de recuperação ($m.itx \leq m.irx$).

Nas ações em que $H' = H$, se uma mensagem não foi gerada, continuará não tendo sido gerada. Este é o caso de $RxAppMan$, $RxAckMan$ e $RxAck(p)$.

Todas as outras ações geram mensagens, portanto alterações em H. Para estas, demonstrou-se que as mensagens geradas possuem $m.itx \leq m.irx$.

Invariante 4

FÓRMULA:

$$I4 \quad \triangleq \quad \forall\, x, y \in Message : \\ \wedge M(x,y) \wedge y \in H \wedge x.itx < CurrCP[x.tx][1] \\ \wedge y.irx \geq CurrCP[x.tx][1] \\ \Rightarrow x \in CurrCP[x.tx][2] \tag{6.46}$$

SIGNIFICADO:

O invariante expressa uma condição para que uma mensagem da aplicação esteja armazenada no ponto de recuperação do processo transmissor. Esta condição está representada pela termo $x \in CurrCP[x.tx][2]$. Além da mensagem y ter sido gerada ($y \in H$) e de, junto com a mensagem x, satisfazerem o operador $M(x,y)$, a mensagem da aplicação deve ter sido gerada antes do ponto de recuperação onde estará armazenada ($x.itx < CurrCP[x.tx][1]$) e a mensagem de resposta deverá ter sido recebida após este mesmo ponto de recuperação ($y.irx \geq CurrCP[x.tx][1]$).

Este invariante representa um dos cenários onde a mensagem da aplicação deve estar armazenada no ponto de recuperação, conforme apresentado na figura 6.2.

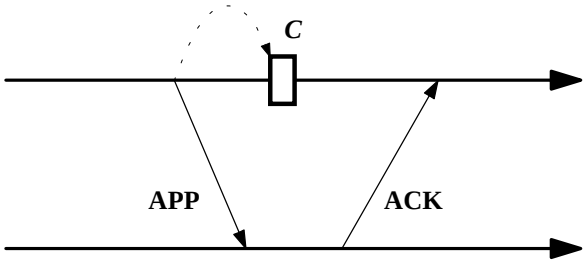

FIGURA 6.2 – Cenário correspondente ao Invariante 4

PROVA:

O invariante tem a mesma forma do invariante incompleto FACK. Assim, para prová-lo, basta provar as três fórmulas pendentes deste invariante.

A propriedade $Prop(x, y)$, neste caso, é a seguinte:

$$\begin{aligned} Prop(x,y) \quad \triangleq \quad & \vee\neg : M(x,y) \\ & \vee(y.irx \geq x.itx \wedge y.itx \geq x.irx) \end{aligned}$$

Invariante 5

FÓRMULA:

$$\begin{aligned} I5 \quad \triangleq \quad & \forall\, m \in Message : \\ & m \in Canal[m.rx][m.tx] \Rightarrow m \in H \end{aligned} \tag{6.47}$$

SIGNIFICADO:

Este invariante expressa o fato que toda mensagem que estiver no canal ($m \in Canal[m.rx][m.tx]$) deve ter sido gerada ($m \in H$).

PROVA:

A prova completa deste invariante pode ser encontrada no relatório técnico [CEC 98].

Invariante 6

FÓRMULA:

$$\begin{aligned} I6 \quad \triangleq \quad & \forall\, m \in Message : \\ & m \in H \Rightarrow m.itx \leq CurrCP[m.tx][1] \end{aligned} \tag{6.48}$$

SIGNIFICADO:

O invariante 6 informa que toda mensagem gerada ($m \in H$) deve ter sido transmitida ($m.itx$) no intervalo de ponto de recuperação atual ($m.itx = CurrCP[m.tx][1]$) ou em intervalo anterior ($m.itx < CurrCP[m.tx][1]$).

PROVA:

A prova completa deste invariante pode ser encontrada no relatório técnico [CEC 98].

Invariante 7

FÓRMULA:

$$I7 \quad \triangleq \quad (ME = 1) \Rightarrow E(PrevCP) \tag{6.49}$$

SIGNIFICADO:

A variável ME é de uso exclusivo do coordenador e sua função é indicar quando o sistema está estabelecendo um novo ponto de recuperação. Se $ME = 0$, então o sistema está estabelecendo um novo ponto; quando $ME = 1$, então o sistema está em operação normal. Neste último caso, $CurrCP = PrevCP$. Além disso, os pontos de recuperação são consistentes (formam uma linha de recuperação).

Assim ***se*** $ME = 1$, ou seja, os pontos de recuperação são consistentes, ***então*** $E(PrecCP)$ é verdadeiro.

PROVA:

Para demonstrar este invariante é necessário deduzir que $ME' = 0$ ou $E(PrevCP')$. Esta prova é relativamente simples, exceto no caso das ações $RxAckCmt$ (pois a ação faz $ME' = 1$) e $RxCmt(p)$ (pois a ação torna $E(PrevCP')$ falso).

Assim sendo, na prova de $RxAckCmt$, demonstra-se que a ação sempre gera $E(PrevCP')$, ou seja, $RxAckCmt \Rightarrow E(PrevCP')$, o que encerra a prova.

No caso de $RxCmt(p)$, demonstrou-se que as premissas da prova e a ação levam a duas fórmulas contraditórias, ou seja, levam a um valor falso, a partir do qual pode-se deduzir qualquer fórmula.

Invariante 8

FÓRMULA:

$$\begin{aligned} I8 \quad \triangleq \quad & \forall\, p \in Proc : \\ & CurrCP[p][1] = PrevCP[p][1] \Rightarrow CurrCP[p] = PrevCP[p] \end{aligned} \tag{6.50}$$

SIGNIFICADO:

O invariante 8 garante que, ***se*** dois pontos de recuperação em um processo qualquer tiverem o mesmo índice, ***então*** estes pontos de recuperação são iguais. É a garantia de que os índices dos pontos de recuperação são suficientes para identificá-los.

PROVA:

A prova completa deste invariante pode ser encontrada no relatório técnico [CEC 98].

Invariante 10

FÓRMULA:

$$I10 \triangleq \forall\, p \in Proc : \\ CurrCP[p][1] \geq PrevCP[p][1] \tag{6.51}$$

SIGNIFICADO:

Este invariante relaciona o índice do ponto de recuperação atual (*CurrCP*), de um processo, com o índice do ponto de recuperação anterior (*PrevCP*). Conforme o invariante, estes índices devem ser os mesmos ou o índice do ponto de recuperação anterior deve ser menor do que o do ponto de recuperação atual.

Apesar deste invariante apresentar o relacionamento entre os índices dos pontos de recuperação, o uso do invariante 8 (que permite a identificação dos pontos de recuperação, através de seus índices) possibilita estender este relacionamento aos pontos de recuperação como um todo e não só aos seus índices.

O relacionamento entre os índices dos pontos de recuperação, descrito neste invariante, expressa a forma como são atualizados os pontos de recuperação atual (*CurrCP*) e anterior (*PrevCP*). Um novo ponto de recuperação é estabelecido em condições tais que os pontos atual e anterior são os mesmos. Este novo ponto de recuperação é salvo em *CurrCP* (ponto de recuperação atual) e deve apresentar um índice que não havia sido usado anteriormente. Na prática, isso é garantido incrementando-se o índice do último ponto de recuperação estabelecido. Desta forma, ao final da operação, o índice do ponto de recuperação atual será maior do que o do ponto de recuperação anterior. No final do estabelecimento de uma nova linha de recuperação (todos os processos já estabeleceram seus pontos de recuperação locais), o conteúdo dos pontos atuais é copiado para os pontos anteriores, fazendo com que fiquem com os mesmos índices.

Desta forma, durante toda a operação, o índice do ponto de recuperação atual dos processos será maior ou igual do que o índice do ponto de recuperação anterior, conforme estabelecido pelo invariante.

PROVA:

As ações em que $CurrCP' = CurrCP$ e $PrevCP' = PrevCP$ terão o mesmo tipo de prova.

Nas ações restantes, ocorre que $CurrCP' = CurrCP$ ou $PrevCP' = PrevCP$. Nas ações em que $CurrCP' = CurrCP$, a prova desenvolve-se no sentido de verificar o relacionamento entre as variáveis *PrevCP* e *PrevCP'* e vice-e-versa.

Invariante 11

FÓRMULA:

$$I11 \triangleq \forall\, m \in : Message : \\ m \in CurrCP[m.tx][2] \wedge m.itx < PrevCP[m.tx][1] \\ \Rightarrow m \in PrevCP[m.tx][2] \tag{6.52}$$

Significado:

O invariante estabelece uma condição para que uma mensagem esteja armazenada em um ponto de recuperação anterior ($m \in PrevCP[m.tx][2]$). A mensagem deverá estar armazenada no ponto de recuperação atual ($m \in CurrCP[m.tx][2]$) e ter sido transmitida antes do estabelecimento do ponto de recuperação anterior ($m.itx < PrevCP[m.tx][1]$).

Este invariante usa, de forma implícita, o fato que os pontos de recuperação *PrevCP* são mais antigos ou iguais aos pontos de recuperação *CurrCP*.

Prova:

Em várias ações da especificação, as variáveis *CurrCP* ou *PrevCP* permanecem inalteradas. Nestes casos, as provas são tão simples quanto mostrar que se uma variável não foi alterada, então as premissas continuam válidas no estado futuro (variáveis-linha).

Nos outros casos, requer-se provas específicas. Esta situação acontece com as ações *SendReq*, *RxAckReq*, *SendApp*(p) e *RxCmt*(p), onde não ocorre $CurrCP' = CurrCP$, e as ações *RxAckReq* e *RxCmt*(p), onde não ocorre $PrevCP' = PrevCP$.

Invariante 12

Fórmula:

$$I12 \quad \triangleq \quad \forall\, p \in Proc : \\ CurrCP[p][1] \leq CurrCP[Mngr][1] \tag{6.53}$$

Significado:

Este invariante expressa o relacionamento entre o índice do ponto de recuperação atual do coordenador e o dos outros processos. O índice do coordenador é sempre maior ou igual ao dos outros processos.

Esta relação entre índices é razoável, uma vez que um novo ponto de recuperação é sempre iniciado pelo coordenador.

Prova:

Com exceção das ações *SendReq*, *RxApp*(p), *RxAck* e *RxReq*, todas as outras podem ser provadas com $CurrCP' = CurrCP$.

Para provar *SendReq*, a prova foi dividida em duas: caso o processo considerado seja o coordenador e caso contrário.

A prova para as três ações restantes é uma só, sendo que a prova completa deste invariante pode ser encontrada no relatório técnico [CEC 98].

Invariantes 13 e 18

Fórmulas:

$$
\begin{array}{lcl}
I13 & \triangleq & \forall\, m \in Message : \\
 & & m \in : Canal[m.rx][m.tx] \wedge m.d = \text{“CMT”} \\
 & & \Rightarrow CurrCP[m.tx][1] = PrevCP[m.tx][1] \\
I18 & \triangleq & \forall\, m \in : Message : \\
 & & m \in Canal[m.rx][m.tx] \wedge m.d = \text{“ACMT”} \\
 & & \Rightarrow CurrCP[m.tx][1] = PrevCP[m.tx][1]
\end{array}
\tag{6.54}
$$

Significado:

Os dois invariantes representam a mesma propriedade para dois tipos de mensagens diferentes: “CMT” e “ACMT”. Se uma destas mensagens estiver presente em algum canal de comunicação, então o índice dos pontos de recuperação *CurrCP* e *PrevCP* do processo transmissor serão idênticos.

Na realidade, levando-se em consideração outros invariantes, não só os índices serão iguais, mas também os próprios pontos de recuperação.

Prova:

Para provar este invariante, foi necessário acrescentar, na especificação, uma condição que garantisse o estabelecimento de um novo ponto de recuperação, somente após ter encerrado o estabelecimento de um ponto mais antigo. Esta condição não havia sido prevista no algoritmo nem na especificação do mesmo. Entretanto, com o desenvolvimento da prova de correção, detectou-se a sua necessidade. Este fato foi um dos exemplos mais importantes que justificaram a valia de uma especificação formal e de sua prova de correção. Sem estas, o algoritmo conteria uma falha, e sua implementação não poderia funcionar adequadamente.

Para expressar a condição detectada, foi acrescentada à ação *SendReq* a seguinte fórmula:

$$
\begin{array}{ll}
\forall\, q \in ProcS : \forall\, m \in Message : & m \in Canal[q][Mngr] \\
 & \Rightarrow m.d = \text{“APP”} \vee m.d = \text{“ACK”}
\end{array}
\tag{6.55}
$$

A fórmula acrescentada garante que, para ocorrer a ação *SendReq*, os canais só podem conter mensagens da aplicação e de resposta. Ou seja, eventuais mensagens de controle do algoritmo (como é o caso das mensagens “CMT” e “ACMT”) foram totalmente consumidas, o que significa que o sistema não está executando o estabelecimento de uma linha de recuperação.

Invariante 14

Fórmula:

$$
\begin{array}{lcl}
I14 & \triangleq & \forall\, m \in Message : \\
 & & m.d = \text{“APP”} \wedge m \in Canal[m.rx][m.tx] \\
 & & \Rightarrow m \in CS[m.tx]
\end{array}
\tag{6.56}
$$

Significado:

Este invariante apresenta uma condição para que uma mensagem esteja na memória volátil das mensagens transmitidas (que representa o estado dos canais

de comunicação) por um processo ($m \in CS[m.tx]$). Para uma mensagem estar armazenada nesta memória, ela deve ser da aplicação ($m.d =$ "APP") e estar no canal de comunicação ($m \in Canal[m.rx][m.tx]$), ou seja, não deve ter sido removida, pelo receptor.

PROVA:

A prova completa deste invariante pode ser encontrada no relatório técnico [CEC 98].

Invariante 15

FÓRMULA:

$$\begin{aligned} I15 \quad \triangleq \quad & \forall\, m \in Message : \\ & m.itx < CurrCP[m.tx][1] \wedge m \in CS[m.tx] \\ & \Rightarrow m \in CurrCP[m.tx][2] \end{aligned} \tag{6.57}$$

SIGNIFICADO:

Este invariante expressa as condições para que uma mensagem esteja armazenada no ponto de recuperação atual, de seu processo transmissor ($m \in CurrCP[m.tx][2]$). Esta mensagem deve ter sido transmitida antes do estabelecimento do ponto de recuperação atual ($m.itx < CurrCP[m.tx][1]$) e deve estar armazenada na memória volátil que representa o estado do canal ($m \in CS[m.tx]$).

Ou seja, toda mensagem que estiver no estado do canal, quando do estabelecimento do ponto de recuperação pelo processo transmissor da mensagens, será armazenada neste ponto de recuperação.

PROVA:

Novamente, várias provas são simplificadas pelo fato das ações usarem $CurrCP' = CurrCP$. Além desta, como o invariante envolve a variável CS, a fórmula $CS' = CS$ também gera algumas simplificações.

Invariante 16

FÓRMULA:

$$\begin{aligned} I16 \quad \triangleq \quad & \forall\, m \in Message : \\ & m \in H \wedge m.d = \text{"ACK"} \wedge m \in Canal[m.rx][m.tx] \\ & \Rightarrow m.msg \in CS[m.msg.tx] \end{aligned} \tag{6.58}$$

SIGNIFICADO:

De forma semelhante ao invariante 14, o invariante 16 apresenta outras condições para que uma mensagem esteja na memória volátil das mensagens transmitidas (que representa o estado dos canais de comunicação) por um processo ($m.msg \in CS[m.msg.tx]$). Por outro lado, enquanto que no invariante 14 essa indicação é feita diretamente com a mensagem da aplicação, no invariante 16 a indicação é feita de forma indireta, através da mensagem de resposta associada (m).

Para que a mensagem da aplicação ($m.msg$) associada com uma mensagem de resposta ($m.d =$ "ACK") esteja armazenada no estado dos canais, a mensagem

de resposta deve ter sido gerada ($m \in H$), o que, na especificação, é idêntico a transmitida, e deve estar no canal de comunicação ($m \in Canal[m.rx][m.tx]$), ou seja, não deve ter sido removida pelo receptor.

Dito de outra forma, uma mensagem da aplicação só poderá ser removida do estado do canal quando a sua resposta tiver sido recebida.

PROVA:

Exceto pelas ações *RxAckReq* e $RxAck(p)$, as provas são relativamente simples. A dificuldade da prova é que estas ações são responsáveis por remover as mensagens da aplicação do estado do canal.

Invariante 17

FÓRMULA:

$$\begin{aligned} I17 \quad \triangleq \quad & \forall\, m \in Message : \\ & m \in H \wedge m.d = \text{“ACK”} \wedge m \notin Canal[m.rx][m.tx] \\ & \Rightarrow m.irx \leq CurrCP[m.rx][1] \end{aligned} \tag{6.59}$$

SIGNIFICADO:

A premissa do invariante faz referência a uma mensagem válida do tipo resposta. Esta mensagem não está no canal. Como a mensagem está em H, então foi transmitida. Por outro lado, como a mensagem não está no canal, alguém a removeu: o processo receptor. Logo, a mensagem foi recebida. Ou seja, não existe a possibilidade de que a mensagem venha a ser recebida no futuro. Desta forma, a mensagem terá sido recebida durante o intervalo de ponto de recuperação atual ou anterior. Não poderá ocorrer de ser recebida em um intervalo futuro.

PROVA:

A prova completa deste invariante pode ser encontrada no relatório técnico [CEC 98].

Invariante 21

FÓRMULA:

$$\begin{aligned} I21 \quad \triangleq \quad & \forall\, m \in Message : \\ & m \in H \wedge m.d = \text{“CMT”} \Rightarrow m.tx = Mngr \end{aligned} \tag{6.60}$$

SIGNIFICADO:

Uma mensagem de controle do tipo “CMT” só pode ser gerada pelo processo coordenador. Assim, se uma mensagem tiver sido gerada ($m \in H$) e for do tipo (“CMT”), então o processo transmissor terá sido o coordenador.

PROVA:

A prova completa deste invariante pode ser encontrada no relatório técnico [CEC 98].

Invariante 22

FÓRMULA:

$$\begin{aligned} I22 \quad \triangleq \quad & \forall\, x, y \in \mathit{Message} : \\ & M(x, y) \wedge y \in \mathit{Canal}[x.tx][x.rx] \\ & \Rightarrow x \notin \mathit{Canal}[x.rx][x.tx] \end{aligned} \tag{6.61}$$

SIGNIFICADO:

Sejam duas mensagens, x e y, que satisfazem o operador $M(x, y)$, então uma condição para que x (mensagem da aplicação, conforme o operador $M(x, y)$) tenha sido removida do canal é que a mensagem y (mensagem de resposta associada com a mensagem x da aplicação, conforme o operador $M(x, y)$) ainda esteja no canal.

PROVA:

Para provar a parte $y \in \mathit{Canal}[x.tx][x.rx]$, foram desenvolvidas quatro provas distintas, segundo o tipo de ação. Foram desenvolvidas provas para:

- ações que envolvem *Send*;
- ações que envolvem o envio de mensagens do tipo "ACK";
- ações que envolvem *Receive*;
- ações que envolvem *Replay*.

De forma semelhante à anterior, para provar da parte $x \notin \mathit{Canal}[x.rx][x.tx]$, também foram desenvolvidas quatro provas. Dependendo do tipo da ação, foram desenvolvidas provas para:

- ações que envolvem *Send*;
- ações que envolvem o envio de mensagens do tipo "APP";
- ações que envolvem *Receive*;
- ações que envolvem *Replay*.

Invariante 23

FÓRMULA:

$$\begin{aligned} I23 \quad \triangleq \quad & \forall\, x, y \in \mathit{Message} : \\ & \wedge x \in H \wedge x.d = \text{"ACK"} \wedge x.msg \neq : \mathit{NoMsg} \\ & \wedge y \in H \wedge y.d = \text{"ACK"} \wedge y.msg = x.msg \\ & \Rightarrow x = y \end{aligned} \tag{6.62}$$

SIGNIFICADO:

Com este invariante é garantido que uma mensagem da aplicação tem associada apenas uma única mensagem de resposta. Ou seja, não são geradas mais de uma mensagem de resposta para cada mensagem da aplicação.

O invariante tem como premissa a existência de duas mensagens, x e y, que foram geradas ($x \in H$ e $y \in H$) e que são mensagens de resposta ($x.d =$ "ACK" e $y.d =$ "ACK"). Além disso, estas mensagens de resposta referem-se às mesmas mensagens da aplicação ($y.msg = x.msg$). A conclusão, expressa no invariante, é que a mensagem x deve ser a mesma y, garantindo que a cada mensagem da aplicação está associada apenas uma mensagem de resposta.

PROVA:

Para provar o invariante em todas as ações, foram desenvolvidas três provas.

A primeira aplica-se às ações que não alteram o histórico de mensagens geradas (a variável H). São as ações $RxAckMan$, $RxAckCmt$ e $RxAck(p)$.

As outras duas provas são aplicadas àquelas ações onde ocorre alteração no histórico. A diferença entre estas duas provas é que em uma delas usa-se a prova por absurdo.

Invariante 24

FÓRMULA:

$$\begin{aligned} I24 \quad \triangleq \quad & \forall\, m \in Message : \\ & m \in H \wedge m.d = \text{“ACK”} \\ & \Rightarrow m.msg.irx \neq NN \end{aligned} \tag{6.63}$$

SIGNIFICADO:

Se uma mensagem de resposta ($m.d =$ "ACK") foi gerada ($m \in H$), então a mensagem da aplicação deve ter sido recebida. Portanto, o seu índice do intervalo de recepção não poderá ser igual a NN ($m.msg.irx \neq NN$).

PROVA:

No desenvolvimento da prova chega-se a um resultado onde o índice de recepção da mensagem da aplicação é um número natural. Entretanto, pela definição de NN ($\forall\, n \in Nat : x > n$), conclui-se que NN é maior do que qualquer número natural e, portanto, maior do que o índice de recepção. Logo, o índice de recepção é diferente de NN.

Invariante 25

FÓRMULA:

$$\begin{aligned} I25 \quad \triangleq \quad & \forall\, m \in Message : \\ & m \in H \wedge m.d = \text{“ACK”} \\ & \Rightarrow \left(\begin{array}{l} \wedge m.msg \notin Canal[m.tx][m.rx] \\ \wedge m.msg.tx = m.rx \\ \wedge m.msg.rx = m.tx \end{array} \right) \end{aligned} \tag{6.64}$$

SIGNIFICADO:

Se uma mensagem válida de resposta ($m.d =$ "ACK") foi transmitida ($m \in H$), então a mensagem da aplicação associada não estará presente no canal ($m.msg \notin$

$Canal[m.tx][m.rx]$), ou seja, terá sido removida pelo receptor. Além disso, o transmissor da mensagem da aplicação será o mesmo receptor da mensagem de resposta ($m.msg.tx = m.rx$) e o receptor da mensagem da aplicação será o mesmo transmissor da mensagem de resposta ($m.msg.rx = m.tx$).

PROVA:

A prova completa deste invariante pode ser encontrada no relatório técnico [CEC 98].

Invariante 26

FÓRMULA:

$$\begin{aligned} I26 \quad \triangleq \quad & \forall\, m \in Message : \\ & m.msg.d = \text{“APP”} \wedge m.msg.irx = NN \\ & \Rightarrow m \notin H \end{aligned} \tag{6.65}$$

SIGNIFICADO:

Este invariante apresenta uma conjunção de condições para as quais uma mensagem não poderá ter sido gerada ($m \notin H$). Estas condições são a necessidade que a mensagem anexada seja da aplicação ($m.msg.d =$ “APP”) e que não tenha sido recebida ($m.msg.irx = NN$).

PROVA:

A prova completa deste invariante pode ser encontrada no relatório técnico [CEC 98].

Invariante 27

FÓRMULA:

$$\begin{aligned} I27 \quad \triangleq \quad & \forall\, m \in Message : \\ & m.msg.d \neq \text{“APP”} \wedge m.msg \neq NoMsg \\ & \Rightarrow m \notin H \end{aligned} \tag{6.66}$$

SIGNIFICADO:

Este invariante expressa uma conjunção de condições para que uma mensagem de resposta não seja gerada: se não tiver mensagem associada ($m.msg \neq NoMsg$) ou o tipo da mensagem associada for diferente de uma mensagem da aplicação ($m.msg.d \neq$ “APP”).

PROVA:

A prova completa deste invariante pode ser encontrada no relatório técnico [CEC 98].

7 Conclusões e trabalhos futuros

Nesta tese, foram apresentados os resultados alcançados durante o trabalho de doutorado, onde foi proposto um novo algoritmo para recuperação de processos em sistemas distribuídos.

O desenvolvimento do algoritmo proposto teve como balizador principal o conjunto de requisitos necessários à implementação. Além disso, buscou-se ferramentas que permitissem demonstrar a correção do algoritmo.

Neste capítulo, são analisadas as soluções apresentadas no trabalho, à luz dos objetivos e pressupostos estabelecidos a princípio. Também são apresentadas as conclusões e os (muitos) trabalhos futuros que se pode antever, como conseqüência desta tese.

7.1 Direcionamento para a implementação

A simplicidade de implementação foi uma das características usadas para direcionar o desenvolvimento do algoritmo proposto. Para isso, buscou-se utilizar mecanismos bem conhecidos, de maneira a facilitar a tarefa de implementação. Outro recurso utilizado foi o de evitar, ao máximo, o uso de algoritmos auxiliares. Com isso, a incorporação do algoritmo a uma aplicação existente não oferece maiores dificuldades de interface, uma vez que se pode trabalhar de maneira que não existam premissas a serem satisfeitas por protocolos adicionais.

Neste trabalho, aparece claramente a questão do compromisso entre os mecanismos utilizados e a característica de falhas do sistema: dependendo da freqüência de falhas, pode ser mais interessante utilizar um tipo de algoritmo ou outro.

Quando for necessário considerar a eficiência do algoritmo de recuperação, deve-se levar em conta o desempenho global dos mecanismos usados na sua implementação. Deve-se considerar o impacto do estabelecimento dos pontos de recuperação, quando em operação livre de erros, conforme estudado por Elnozahy [ELN 92a], bem como o tempo de processamento que deverá ser reprocessado, em caso de falha e retorno. Em geral, estas duas etapas estão intimamente relacionadas e a melhora no desempenho de uma implica em piora na outra. Esta relação de compromisso foi analisada pelo autor em trabalho anterior [CEC 98a] através da comparação de dois algoritmos de recuperação de processos: um síncrono e outro assíncrono. Deste estudo, foi concluído que o desempenho sofre influência, principalmente, de dois fatores: do bloqueio da aplicação, durante o estabelecimento dos pontos de recuperação e da necessidade de buscar uma linha de recuperação, dentre os pontos de recuperação locais registrados, em caso de falha. O primeiro fator afeta principalmente o algoritmo síncrono enquanto que o segundo, o assíncrono.

O algoritmo proposto neste trabalho foi desenvolvido de maneira a não bloquear a aplicação, durante o estabelecimento dos pontos de recuperação, nem buscar por uma linha de recuperação, em caso de falha, uma vez que estaria disponível. O algoritmo proposto foi classificado como coordenado, ou seja, os processos coordenam-se para o estabelecimento de linhas de recuperação, sem bloquear a aplicação. Além disso, em caso de falha, os processos podem usar a última linha de recuperação estabelecida, imediatamente, sem haver necessidade de busca.

Adicionalmente, dois outros ganhos foram alcançados: utilização mínima de

memória estável e rapidez no retorno. O volume de dados a serem salvos na memória estável é mínimo, uma vez que só há a necessidade de manter uma linha de recuperação. A retomada de processamento, após uma falha, é feita de forma bastante rápida, uma vez que as mensagens armazenadas nos *logs* podem ser retransmitidas imediatamente e em qualquer ordem. Além disso, mensagens perdidas no canal e que poderiam levar a ocorrência de *livelocks* são tratadas (descartadas) concomitantemente com o processamento normal. Desta forma, uma vez que o sistema tenha sido retornado para um estado livre de erros, pode-se retomar o processamento.

Para que o algoritmo proposto estabeleça linhas de recuperação, é necessário que a aplicação informe a periodicidade com que isso deve ocorrer. Esta necessidade pode ser considerada uma fraqueza, uma vez que a aplicação deverá suprir este parâmetro, para o algoritmo de recuperação. Entretanto, a adaptação de parâmetros dos algoritmos de suporte das aplicações pela própria aplicação é uma prática desejável, manifestada por vários autores. Este expediente permite um ajuste imediato dos algoritmos de suporte às necessidades da aplicação, fornecendo um sistema mais eficiente do que seria possível, caso não houvesse este tipo de ajuste.

O fornecimento de parâmetros, por parte da aplicação, para possibilitar que a recuperação seja mais eficiente, é apresentada no trabalho de Brézinsky [BRZ 95]. Neste trabalho, a aplicação classifica as mensagens de acordo com a possibilidade de tornarem-se órfãs ou perdidas, sem que isso afete a consistência da mesma. Assim, o algoritmo de recuperação é dirigido de maneira a tratar as mensagens não só baseado no seu relacionamento com os pontos de recuperação mas também usando informações das aplicações.

Entretanto, pode ser impossível obter, diretamente, os parâmetros das aplicações. Para superar esta dificuldade, pode ser utilizado um método indireto, como a captura de informações da aplicação. Em trabalho recente [FON 2001], foram apresentados o estudo dos mecanismos mais usados para a captura de mensagens bem como as medidas do impacto ao desempenho das aplicações. Visando associar a captura de mensagens com a recuperação, observou-se que o conjunto dos mecanismos de captura (integração, interceptação e serviço) fornece uma boa flexibilidade de implementação, uma vez que cada mecanismo foi projetado para diferentes níveis do sistema: desde o sistema operacional até a aplicação (usuário). Além disso, o impacto ao desempenho não se mostrou significativo, principalmente no caso dos mecanismos de integração e interceptação.

7.2 Prova de correção

Não basta propor um algoritmo. É necessário mostrar que todas as execuções permitidas pelo mesmo correspondem a execuções corretas. No caso do algoritmo proposto neste trabalho, isso significa mostrar que a **propriedade de consistência** é mantida durante qualquer das execuções permitidas.

Para mostrar que o algoritmo mantém a consistência, é necessário desenvolver uma prova de correção. Para isso, uma das possibilidades é descrever o algoritmo através de uma especificação formal e, então, mostrar que esta descrição garante a propriedade desejada (no caso, a consistência).

Escolheu-se especificar e provar a correção do algoritmo usando a linguagem de especificação TLA+ de Lamport [LAM 94]. Desta forma, escreveu-se uma es-

pecificação onde as ações correspondem às atividades a serem desenvolvidas pelos processos do sistema, com o objetivo de estabelecer pontos de recuperação locais. Estes, por sua vez, deverão formar linhas de recuperação.

Com a especificação do algoritmo e o critério de consistência formalizados, iniciou-se a escrita das provas de correção. Durante esta etapa, percebeu-se que a alteração de algumas cláusulas levaria a uma especificação mais robusta, além de facilitar a tarefa de prova. Ao final, várias foram as alterações incorporadas à especificação original. Em geral, estas alterações correspondiam a comportamentos que não haviam sido antecipados mas que foram revelados durante a escrita das provas de correção.

Um exemplo desta realimentação ocorreu durante a prova de correção dos invariantes 13 e 18 (vide seção 6.3.7). Estes invariantes expressam o fato que, enquanto houver mensagens do tipo "CMT" ou "ACMT" em alguns dos canais de comunicação, então os pontos de recuperação anterior e o atual, do processo transmissor da mensagem, serão os mesmos. Entretanto, estes invariantes não poderiam ser provados sem que fosse caracterizado o estado do sistema, quando do início do estabelecimento de uma nova linha de recuperação. Esta caracterização foi feita com o acréscimo de uma cláusula à ação que corresponde ao início de uma nova linha de recuperação: para que esta ação seja habilitada, todas as mensagens em trânsito nos canais que partem do coordenador devem ser do tipo "APP" (da aplicação) ou "ACK" (resposta às mensagens da aplicação). Ou seja, não pode haver mensagens de controle de estabelecimento de uma linha de recuperação. Desta forma, a especificação foi acrescida de uma cláusula que identifica quando pode iniciar o estabelecimento de um novo ponto de recuperação.

Como resultado final da tarefa de prova de correção, foi obtida uma especificação depurada, o que garante que não serão passados erros de algoritmo para uma futura etapa de implementação.

Apesar da tarefa de desenvolvimento das provas de correção ser bastante trabalhosa, o resultado final é compensador, na medida que se obtém uma especificação correta e mais precisa.

A prova formal oferece uma comprovação de que a especificação garante a propriedade buscada. Entretanto, pode-se cometer erros ao descrever a propriedade a ser alcançada, o que levaria a um comportamento inadequado, quando o algoritmo fosse implementado. Este problema não tem solução. Entretanto, mesmo neste caso, a prova formal auxilia na comparação entre o comportamento de uma implementação e o comportamento desejado, na medida que garante a equivalência entre a propriedade e a especificação. Desta forma, pode-se comparar os resultados da implementação, diretamente, com a propriedade descrita, o que é bem mais simples do que compará-los com a especificação completa.

7.3 Extensão dos critérios de consistência

Para desenvolver a prova formal, foi necessário descrever a especificação e a propriedade a ser verificada: a consistência. Entretanto, os critérios tradicionais de consistência apresentam as mensagens classificadas como potencialmente órfãs ou perdidas. As mensagens não são analisadas de acordo com a sua função (por exemplo, discriminar as mensagens da aplicação das de resposta), o que não é adequado

ao algoritmo proposto.

Então, teve que ser desenvolvida uma formalização da propriedade de consistência que levasse em consideração os dois tipos de mensagens existentes: as mensagens da aplicação, "APP", e as mensagens de resposta (reconhecimento), "ACK".

Para desenvolver o modelo de consistência, foram analisados os eventos significativos do comportamento das mensagens de aplicação e resposta, em relação aos instantes de estabelecimento dos pontos de recuperação locais. Foram listadas e analisadas todas as combinações possíveis dos eventos de transmissão e recepção das mensagens da aplicação e de sua resposta.

Ao final, obteve-se uma fórmula composta por quatro outras, que caracteriza o comportamento dos pares de mensagens (aplicação e resposta), em relação aos pontos de recuperação estabelecidos no transmissor e no receptor das mesmas, de maneira a garantir a consistência do sistema.

Este trabalho tem duas contribuições para a análise da consistência. A primeira foi a definição de consistência, que leva em consideração os diferentes tipos de mensagens. Ou seja, considera que os critérios de consistência não se aplicam igualmente a todos os tipos de mensagens. A segunda contribuição diz respeito a forma como a análise foi desenvolvida. Foram listadas todas as combinações dos eventos que caracterizam as mensagens em relação aos pontos de recuperação locais e, então, os cenários obtidos foram analisados. Como resultado, foi definida uma expressão formal que expressa o relacionamento entre mensagens e pontos de recuperação, de maneira que a computação seja consistente.

Cabe salientar que os critérios de consistência considerados na análise dos cenários obtidos foram bastante conservadores. Se premissas alternativas fossem usadas, a expressão obtida para a consistência poderia ser diferente. Uma revisão dos critérios usados na análise dos cenários e a obtenção de novas expressões de consistência é trabalho que poderá ser desenvolvido no futuro.

7.4 Tratamento das mensagens perdidas

Também é inovador o enfoque dado às mensagens perdidas. Vários são os aspectos que levam a necessidade de tratá-las. Estes aspectos dizem respeito ao tipo de algoritmo usado para o estabelecimento dos pontos de recuperação, o tipo de canal de comunicação, o uso de *log* de recepção e as causas que podem levar uma mensagem a tornar-se perdida.

No algoritmo proposto neste trabalho, as mensagens perdidas devido ao retorno, durante a recuperação, recebem um tratamento que garante a retomada consistente do processamento.

A seguir, os aspectos relacionados com o tratamento das mensagens perdidas serão discutidas. Nesta seção, considera-se "comunicação confiável" como sinônimo de "garantia de entrega".

7.4.1 Algoritmos apenas para *Checkpointing*

Quando se projeta algoritmos visando exclusivamente o estabelecimento de pontos de recuperação, é necessário levar em consideração apenas o ambiente onde esta operação será executada. Entretanto, quando os pontos de recuperação tiverem uma utilização específica, esta deve ser levada em consideração. Esse é o caso

do estabelecimento de pontos de recuperação, visando atividades de tolerância a falhas. Neste ambiente, ou seja, em um ambiente onde podem ocorrer falhas, é necessário que o algoritmo usado para o estabelecimento dos pontos de recuperação seja ajustado à possibilidade de recuperação do sistema.

Desta forma, algumas das premissas que podem ser usadas como base para o desenvolvimento de algoritmos de estabelecimento de pontos de recuperação (mais precisamente, de *snapshots*, uma vez que não necessariamente serão usados para recuperação) não são aceitáveis quando esses *snapshots* são usados como suporte à recuperação e ao retorno. Isso ocorre com as mensagens perdidas. Para que um algoritmo de estabelecimento de pontos de recuperação possa ser usado na recuperação de processos, é necessário que as mensagens perdidas sejam tratadas de alguma forma. Por exemplo, pode-se recorrer ao *log* de mensagens ou ao bloqueio da aplicação, até que não existam mais mensagens perdidas.

7.4.2 Canais confiáveis e não confiáveis

Visando tornar transparente, para a aplicação, a operação do algoritmo de recuperação, este é acrescentado entre a aplicação e a comunicação. Se a aplicação pressupõe que os canais de comunicação são confiáveis, então o algoritmo de recuperação não pode permitir que ocorram mensagens perdidas; se, por outro lado, a aplicação pressupõe que os canais de comunicação não são confiáveis, então a recuperação não necessita tratar as mensagens perdidas [ELN 96].

Entretanto, neste trabalho, buscou-se mostrar que nem sempre o tipo de canal determina a forma como a recuperação deve tratar as mensagens perdidas. É necessário considerar o tipo de aplicação, no que diz respeito ao seu comportamento frente as mensagens perdidas. Os detalhes desta análise estão descritos na seção 2.2 e levaram à conclusão da necessidade do tratamento das mensagens perdidas, mesmo quando os canais não são confiáveis.

Ainda com relação aos canais, encontra-se na literatura propostas de algoritmos de recuperação onde uma das premissas é a existência de canais capazes de garantir que as mensagens serão entregues nos destinos. Entretanto, a implementação de canais totalmente confiáveis não é possível sem o auxílio da aplicação. Ou seja, a aplicação deve prover procedimentos para tratar aquelas mensagens que não puderem ser entregues pela comunicação, com os recursos que lhe estão disponíveis (à comunicação).

Como o algoritmo para o estabelecimento dos pontos de recuperação foi instalado entre a aplicação e a comunicação, as mensagens que não puderam ser entregues serão passadas para a recuperação que deverá prover algum tipo de tratamento. Desta forma, o algoritmo de recuperação não pode supor a inexistência de mensagens perdidas. É necessário que os algoritmos especifiquem o tratamento para estas mensagens, mesmo que seja apenas repassá-las para a aplicação.

7.4.3 Utilização de *log* de recepção

O *log* de mensagens na recepção é um mecanismo encontrado em algoritmos para o estabelecimento dos pontos de recuperação. Este mecanismo é usado para tratar as mensagens potencialmente perdidas, armazenando-as de maneira a serem repetidas, em caso de falha e recuperação.

Entretanto, a utilização de *log* de recepção só poderá oferecer um tratamento

consistente das mensagens perdidas se estiver associado com uma comunicação totalmente confiável. Isso é necessário pois uma mensagem transmitida e ainda não recebida, quando da ocorrência de uma falha, pode levar a uma mensagem perdida que não estará registrada no *log* de recepção, o que impedirá a sua repetição, levando o sistema para um estado inconsistente.

Novamente, é necessário que os canais sejam totalmente confiáveis, o que não se mostrou ser viável.

Cabe salientar que alguns protocolos utilizam *log* de recepção sem a premissa de comunicação confiável. Estes algoritmos utilizam o *log* de recepção de forma diversa da descrita aqui e, em geral, levam em consideração que as mensagens podem ser perdidas nos canais.

7.4.4 Mensagens perdidas devido à comunicação e à recuperação

Uma mensagem é chamada de perdida quando existe o registro de sua transmissão mas não existe o correspondente registro de sua recepção. Esta definição caracteriza um estado do sistema, não informando nada sobre a forma como foi atingido.

Entretanto, no caso específico de recuperação de processos, as mensagens podem tornar-se perdidas devido a problemas na comunicação ou devido ao processo de retorno. Quando são consideradas medidas para o tratamento das mensagens perdidas, nem sempre esta diferenciação é levada em conta. Em geral, são consideradas, apenas, aquelas mensagens perdidas devido à comunicação.

As mensagens perdidas devido à comunicação são aquelas que, por algum motivo relacionado com os canais de comunicação, não atingiram seu destino. Estas podem ser detectadas pela ausência da mensagem de confirmação, após transcorrido um tempo limite de espera (*timeout*). Em geral, estas falhas são solucionadas pela repetição da mensagem, até que uma resposta seja recebida. A detecção e o tratamento destes defeitos dependem de fatores dinâmicos do sistema como temporizadores e mecanismos de controle de repetição.

No caso das mensagens perdidas devido a recuperação, não existem fatores dinâmicos. Na realidade, estas mensagens nem mesmo podem ser caracterizadas como realmente perdidas. Estas são melhor definidas como potencialmente perdidas. O cenário que leva ao seu aparecimento é o seguinte: quando ocorre uma falha e o sistema retorna, deve fazê-lo para um ponto de recuperação local a cada processo. Desta forma, o sistema é colocado em um estado que havia sido registrado anteriormente. Entretanto, neste estado, pode haver uma mensagem que tenha sido transmitida antes do ponto de recuperação usado para o retorno, no processo transmissor, e que tenha sido recebida após o ponto de recuperação usado para o retorno, no processo receptor. Desta forma, está caracterizada uma mensagem perdida para a qual não existem mecanismos de temporização associados que permitam a sua detecção. Pode-se argumentar que a solução é recuperar todos os recursos do sistema, inclusive os mecanismos de temporização. Entretanto, quando é proposto o algoritmo, isso deve estar explícito, o que, em geral, não é feito.

Adicionalmente, pode-se notar que, se não houvesse falha e a conseqüente necessidade de retorno ou se um ponto de recuperação posterior fosse usado, a mensagem em questão não se tornaria perdida. Desta forma, além de ficar bem justificada a nomenclatura de mensagem potencialmente perdida, também está bem identificado que a sua causa é o retorno, o que demanda um tratamento diferenciado

daquele empregado no caso das mensagens perdidas devido à comunicação.

7.5 Incorporação da coleta de lixo

Os algoritmos síncronos, devido a sua natureza, incorporam um mecanismo para a coleta de lixo: remoção dos pontos de recuperação que perderam a utilidade. Esta coleta de lixo é feita naturalmente, quando um novo ponto de recuperação é estabelecido, uma vez que pode ser salvo de forma a utilizar a área de memória estável ocupada por um ponto que perdeu a sua utilidade. Este ponto mais antigo pode ser eliminado pois faz parte de uma linha de recuperação que já foi substituída por uma mais recente.

Na realidade, os algoritmo síncronos de estabelecimento de pontos de recuperação requerem espaço em memória estável para dois pontos de recuperação. Isso é necessário pois, durante o estabelecimento de um ponto de recuperação (que só estará completo ao seu final), deve existir uma linha de recuperação intacta. No final do estabelecimento da nova linha de recuperação, a linha mais antiga poderá, então, ser descartada.

O algoritmo proposto, apesar de não ser síncrono, utiliza-se do mesmo mecanismo para conservar os recursos do sistema. Entretanto, como trabalho original, o mecanismo de coleta de lixo foi incorporado na especificação do algoritmo e provada a sua correção.

7.6 Eliminação da confirmação final

Do ponto de vista do coordenador, a seqüencia de ações a serem efetuadas para o estabelecimento de uma linha de recuperação, é formada por quatro etapas bem definidas e que ocorrem sem bloquear a aplicação: envio da solicitação de estabelecimento de uma nova linha de recuperação; recepção das confirmações de estabelecimento, de cada processo; envio da informação de que uma nova linha de recuperação já foi estabelecida e início da coleta de lixo; recepção das confirmações de execução da coleta de lixo, de cada processo.

As quatro etapas visam o estabelecimento de um conjunto de pontos de recuperação locais (uma linha de recuperação) e a coleta de lixo, sendo que a linha de recuperação é obtida no final da segunda etapa, ou seja, antes de iniciar a coleta de lixo. Desta forma, durante o estabelecimento da nova linha de recuperação, coexistirão dois conjuntos de pontos de recuperação: um mais antigo, que forma uma linha de recuperação, e um mais novo, que formará uma linha de recuperação. Assim, mesmo que todos os processos ainda não tenham sido informados, a nova linha de recuperação estará disponível, ao final da segunda etapa.

A partir do mecanismo proposto para o estabelecimento dos pontos de recuperação, o retorno foi definido de forma a identificar qual o conjunto que é consistente e que proporciona a menor perda de processamento (o último conjunto estabelecido).

Aparentemente, o conjunto dos mecanismos usados para o estabelecimento de uma linha de recuperação e para o retorno levam ao estabelecimento de uma linha de recuperação sem a necessidade da etapa final: o envio, por parte de todos os processos, da confirmação de execução da coleta de lixo. Entretanto, a existência desta etapa tornou menos complexa a prova de correção e por este motivo foi man-

tida. A avaliação da possibilidade de eliminar esta última etapa foi deixada como um possível trabalho futuro.

7.7 Acelerando o *Checkpointing*

O coordenador solicita o estabelecimento de pontos de recuperação locais através do envio de uma mensagens específica para cada processo do sistema. Quando estas mensagens chegam aos seus destinos, estes processos devem estabelecer um novo ponto de recuperação e então responder, informando que a tarefa foi concluída.

Como o algoritmo tem por premissa que as mensagens não necessitam ser ordenadas (não é necessário, por exemplo, usar canais FIFO), pode ocorrer que uma mensagem enviada após a transmissão da solicitação de um novo ponto de recuperação, chegue antes ao seu destino. Neste caso, o algoritmo prevê que o processo destino deverá estabelecer um novo ponto de recuperação, sem ter que esperar pelo recebimento da mensagem de solicitação. Entretanto, a resposta de que foi completada a tarefa, só será enviada após o recebimento da mensagem de solicitação.

Assim sendo, o envio da resposta imediatamente após ter sido estabelecido o ponto de recuperação local, sem esperar pela chegada da mensagem de solicitação, poderia reduzir o tempo gasto no tarefa. Esta otimização também foi deixada para um trabalho futuro, uma vez que envolveria alterações da especificação e, como conseqüência, na prova de correção.

7.8 Separação da especificação em módulos

Uma das formas de especificar os sistemas usando TLA é especificar módulos que serão usados na especificação de outros módulos.

Esta maneira de trabalhar oferece a vantagem da reutilização, ou seja, módulos que especificam determinados mecanismos podem ser usados na definição de vários outros módulos mais complexos. Além disso, os módulos básicos podem ter, em anexos, uma série de teoremas (propriedades) já provadas, o que pode simplificar as provas de correção dos módulos que os utilizam.

No caso da especificação do algoritmo proposto, estão incorporadas ações que dizem respeito à comunicação confiável, à coleta de lixo, ao *log* de transmissão de mensagens, à modelagem de canais não FIFO e, obviamente, ao salvamento de pontos de recuperação. Estes mecanismos estão entrelaçados na especificação. A análise da viabilidade de separá-los em módulos individuais e a sua utilização na composição de uma nova especificação do algoritmo proposto é, certamente, uma tarefa futura.

7.9 Extensões da especificação

Durante o trabalho de especificação e prova, foi considerada a possibilidade de acrescentar-se a modelagem da ocorrência de falhas, a modelagem da etapa de retorno, a modelagem das colisões na rede de comunicação e a modelagem do fator tempo. A inclusão destes modelos, certamente, forneceria uma especificação mais

rica em detalhes além de mais precisa. Entretanto, devido ao tempo disponível, optou-se por não utilizá-las, uma vez que não seriam necessárias para demonstrar a correção do algoritmo.

Entretanto, a inclusão destes fatores à especificação é tarefa que poderá ser usada como motivo para trabalhos futuros. Além disso, estes fatores poderão originar a especificação de módulos específicos que poderão ser anexados à especificação de algoritmos, da mesma forma que foi proposto na seção 7.8.

7.10 Implementação

Um dos principais objetivos de propor-se novos algoritmos é obter aplicações mais robustas e eficientes. Este também é o caso do algoritmo proposto neste trabalho. Portanto, a implementação do algoritmo está na lista de possíveis trabalhos futuros.

Entretanto, até que o algoritmo seja implementado como parte de uma aplicação, provavelmente deverá passar por várias etapas de desenvolvimento. Algumas destas etapas podem ser a simulação de modelos, a simulação de sistemas distribuídos, a implementação do algoritmo de forma isolada e a implementação como mecanismo componente de um sistema maior.

A **simulação de modelos** é uma tarefa que poderá ser efetivada a curto prazo. Para isso, é necessário uma ferramenta de simulação de modelos TLA. O simulador de modelos TLC, recentemente disponibilizado, poderá ser usado nesta tarefa.

A **simulação de sistemas distribuídos** também dependerá de ferramentas específicas, onde os sistemas distribuídos possam ser simulados. Estas ferramentas deverão permitir a escrita do código de implementação do algoritmo-alvo além de permitirem a modelagem de falhas. Atualmente, um dos trabalhos de dissertação do grupo de tolerância a falhas é sobre a adaptação do simulador NS2, do projeto VINT (http://www.isi.edu/nsnam/vint/), de maneira que possam ser simuladas falhas.

Finalmente, a implementação propriamente dita deverá ser feita de maneira isolada, para que sejam efetuadas medições de desempenho e analisado seu comportamento, quando em ambiente real e, então, o algoritmo será suficientemente conhecido para permitir a sua implementação, como um componente de um sistema completo.

7.11 Projetos futuros

Ao final do trabalho, haviam sido listadas uma série de atividades decorrentes da tese e que poderiam ser desenvolvidas no futuro. Alguns dos projetos identificados, adicionais aos já comentados, são listados a seguir:

- Separação da especificação TLA do algoritmo proposto em várias especificações correspondentes aos algoritmos componentes;
- Formalização, em TLA, da ocorrência de falhas, para utilização em especificações de sistemas tolerantes a falhas;

- Formalização, em TLA, da passagem do tempo, para permitir uma análise formal do desempenho;
- Estudo do comportamento da recuperação de processos diante da utilização da memória de outras máquinas do sistema distribuído, como memória estável;
- Estudo e proposta de expressões alternativas para a caracterização da propriedade de consistência, para aplicações específicas, usando o procedimento de análise apresentado nesta tese;
- Implementação de bibliotecas para salvamento do estado de processos, em ambientes Unix/Linux e Windows, para uso pelos algoritmos de recuperação distribuída;
- Estudo e extensão do conceito de consistência para canais de comunicação não confiáveis. Deste trabalho, resultaria a especificação de um módulo ao qual estariam associados teoremas (propriedades);
- Estudo e possível implementação de simuladores de sistemas distribuídos para facilitar a coleta de dados da execução de algoritmos distribuídos, quando em presença de falhas. Uma análise completa do desempenho de um algoritmo de recuperação só pode ser feita se os dois mecanismos componentes: estabelecimento dos pontos de recuperação e retorno, puderem ser exercitados em diferentes proporções, ou seja, com taxas variáveis de falhas;
- Estudo e desenvolvimento de ferramentas que possibilitem analisar o desempenho dos sistemas diante da ocorrência de falhas. Com isso, a recuperação poderia ser modelada, por exemplo, em um programa sintético, do qual poderiam ser extraídas medidas de desempenho;
- Desenvolvimento de *workloads* para *benchmarks* que evidenciem o efeito da recuperação de processos sobre o desempenho total do sistema;
- Continuação do aprimoramento do algoritmo proposto para a recuperação, incorporando aperfeiçoamentos aos mecanismos e otimizações que possibilitem um melhor desempenho. Este é o caso da eliminação da quarta etapa do algoritmo proposto (conforme seção 7.6) e a aceleração discutida na seção 7.7.

Bibliografia

[AGU 98] AGUILERA, M. K.; CHEN, W.; TOUEG, S. Failure detection and consensus in the crash-recovery model. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED COMPUTING, 12., 1998, Greece. **Proceedings...** [S.l.: s.n.], 1998. p.233–245.

[ALV 99] ALVISI, L. et al. **An analysis of communication-induced checkpointing**. Austin, Texas: Department of Computer Science, Univ. of Texas at Austin, 1999. (Technical Report, TR-99-01).

[AND 81] ANDERSON, T.; LEE, P. A. **Fault tolerance, principles and practice**. Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

[AVI 76] AVIZIENIS, A. Fault-tolerant systems. **IEEE Transactions on Computers**, New York, v.25, n.12, p.1304–1312, Dec. 1976.

[BAL 97] BALDONI, R. et al. A communication-induced checkpointing protocol that ensures rollback-dependency trackability. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 27., 1997, Seattle. **Digest of Papers...** Los Alamitos:IEEE Computer Society Press, 1997. p.68–77.

[BAL 99] BALDONI, R.; QUAGLIA, F.; FORNARA, P. An index-based checkpointing algorithm for autonomous distributed systems. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.10, n.2, p.181–192, Feb. 1999.

[BOR 89] BORG, A.; BLAU, W.; GRAETSCH, W. Fault tolerance under unix. **ACM Transactions on Computer Systems**, New York, v.7, n.1, p.1–24, Feb. 1989.

[BRI 84] BRIATICO, D.; CIUFFOLETTI, A.; SIMONCINI, L. A distributed domino-effect free recovery algorithm. In: SYMP. ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATA BASE SYSTEMS, 1984. **Proceedings...** [S.l.: s.n.], 1984. p.207–215.

[BRZ 95] BRZEZIŃSKI, J.; HÉLARY, J.-M.; RAYNAL, M. **Semantics of recovery lines for backward recovery in distributed systems**. Rennes: Institut National de Recherche en Informatique et en Automatique, 1995. (Rapport de Recherche, RR-2468). Disponível em: <http://www.inria.fr/rrrt/rr-2468.html>. Acesso em: set. 2001.

[CAO 98] CAO, G.; SINGHAL, M. On coordinated checkpointing in distributed systems. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.9, n.12, p.1213–1225, Dec. 1998.

[CEC 98] CECHIN, S. L. **Prova formal de protocolo de recuperaćão por retorno, em tla**. 2002. Relatório de Pesquisa (RP-319) - Instituto

de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[CEC 98a] CECHIN, S. L. **Avaliação teórica do desempenho de algoritmos de recuperaćão por retorno do tipo síncrono e assíncrono**. 1998. Trabalho Individual (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[CEC 98b] CECHIN, S. L.; JANSCH-PÔRTO, I. Performance evaluation of checkpointing and rollback-recovery algorithms for distributed systems. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES, 10., 1998. **Proceedings...** [S.l.: s.n.], 1998. p.137–146.

[CEC 2001] CECHIN, S. L.; JANSCH-PÔRTO, I. A new efficient coordinated checkpointing. In: IEEE LATIN AMERICAN TEST WORKSHOP, 2., 2001. **Proceedings...** [S.l.: s.n.], 2001. p.56–61.

[CHA 85] CHANDY, K. M.; LAMPORT, L. Distributed snapshot: determining global states of distributed systems. **ACM Transactions on Computer Systems**, New York, v.3, n.1, p.63–75, Feb. 1985.

[CHA 88] CHANDY, K. M. **Parallel program design**: a foundation. Massachusetts: Baddison-Wesley, 1988. 516p.

[CHI 96] CHIU, G.; YOUNG, C. Efficient roolback-recovery technique in distributed systems. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.7, n.6, p.565–577, June 1996.

[CLA 96] CLARKE, E. M.; WING., J. M. Formal methods: state of the art and future directions. **ACM Computing Surveys**, New York, v.28, n.4, p.626–643, Dec. 1996.

[CRI 91] CRISTIAN, F. Understanding fault-tolerant distributed systems. **Communications of the ACM**, New York, v.34, n.2, p.56–78, 1991.

[DOH 2000] DO-HYUNG, K.; CHANG-SOON, P. A communication-induced checkpointing algorithm using virtual checkpoint on distributed systems. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED SYSTEMS, 7., 2000, Iwate, Japan. **Proceedings...** [S.l.]:IEEE Computer Society, 2000. p.145–150.

[DOL 97] DOLEV, D. et al. Failure detectors in omission failure environments. In: ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING (PODC), 16., 1997, Santa Barbara, California. **Proceedings...** [S.l.: s.n.], 1997.

[ELN 92] ELNOZAHY, E. N.; ZWAENEPOEL, W. Manetho: transparent rollback-recovery with low overhead, limited rollback, and fast output

commit. **IEEE Transactions on Computers**, New York, v.41, n.5, p.526–531, May 1992.

[ELN 92a] ELNOZAHY, E. N.; JOHNSON, D. B.; ZWAENEPOEL, W. The performance of consistent checkpointing. In: SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, 11., 1992, Houston, Texas. **Proceedings...** [S.l.]:IEEE Computer Society Press, 1992. p.39–47.

[ELN 94] ELNOZAHY, E. N.; ZWAENEPOEL, W. On the use and implementation of message logging. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 24., 1994, Austin, Texas. **Digest of Papers...** Los Alamitos:IEEE Computer Society Press, 1994. p.298–309.

[ELN 96] ELNOZAHY, E. N.; JOHNSON, D. B.; WANG, Y. M. **A survey of rollback-recovery protocols in message-passing systems**. Pittsburgh: Department of CS, Carnegie Mellon University, 1996. (Technical Report, CMU-CS-96-181). Disponível em: <ftp://ftp.cs.cmu.edu/user/mootaz/papers/S.ps>. Acesso em: set. 2001.

[FON 2001] FONTOURA, A. B.; JANSCH-PÔRTO, I.; CECHIN, S. L. Evaluating approaches of the capturing of application information. In: IEEE LATIN AMERICAN TEST WORKSHOP, 2., 2001. **Proceedings...** [S.l.: s.n.], 2001. p.148–153.

[GEN 99] GENDELMAN, E.; BIC, L. F.; DILLENCOURT, M. B. An efficient checkpointing algorithm for distributed systems implementing reliable communication channels. In: SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, 18., 1999, Lausanne, Switzerland. **Proceedings...** Los Alamitos:IEEE Computer Society Press, 1999. p.1–2.

[GEN 99a] GENDELMAN, E.; BIC, L. F.; DILLENCOURT, M. B. **Efficient checkpointing algorithm for distributed systems with reliable communnication channels**. Irvine, CA: Department of Information and Computer Science - University of California, 1999. (TR #99-34). Disponível em: <http://www.ics.uci.edu/ egendelm/prof/publications.html>. Acesso em: set. 2001.

[HSU 99] HSU, S.-T.; CHANG, R.-C. An implementation of using remote memory to checkpoint processes. **Software - Practice and Experience**, Sussex, England, v.29, n.11, p.985–1004, Sept. 1999.

[HUA 95] HUANG, Y.; WANG, Y.-M. Why optimistic message logging has not been used in telecommunications systems. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 25., 1995. **Digest of Papers...** [S.l.: s.n.], 1995. v.25, p.459–463.

[HÉL 99] HÉLARY, J.-M.; NETZER, R. H. B.; RAYNAL, M. Consistency issues in distributed checkpoints. **IEEE Transactions on Software Engineering**, New York, v.25, n.2, p.274–281, Mar. 1999.

[JAL 94] JALOTE, P. **Fault tolerance in distributed systems**. New Jersey: Prentice-Hall, 1994. 432p.

[JOH 87] JOHNSON, D. B.; ZWAENEPOEL, W. Sender-based message logging. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 7., 1987. **Digest of Papers...** New York:IEEE, 1987. p.14–19.

[JOH 90] JOHNSON, D. B.; ZWAENEPOEL, W. Recovery in distributed systems using optimistic message logging and checkpointing. **Journal of Algorithms**, New York, n.11, p.462–491, Aug. 1990.

[JUA 91] JUANG, T.; VENKATESAN, S. Crash recovery with little overhead. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 11., 1991. **Proceedings...** [S.l.: s.n.], 1991. p.454–461.

[KIM 93] KIM, J. L.; PARK, T. An efficient protocol for checkpointing recovery in distributed systems. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.4, n.8, p.955–960, Aug. 1993.

[KOO 87] KOO, R.; TOUEG, S. Checkpoint and rollback-recovery for distributed systems. **IEEE Trans. on Software Engineering**, New York, v.SE-13, n.1, p.23–31, Jan. 1987.

[LAI 87] LAI, T. H.; YANG, T. H. **On distributed snapshots**. 25th. Amsterdam: Springer-Verlag, 1987. 153–158p.

[LAM 78] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. **Communications of the ACM**, New York, v.21, n.7, p.558–565, July 1978.

[LAM 94] LAMPORT, L. The temporal logic of actions. **ACM Transactions on Programming Languages and Systems**, New York, v.16, n.3, p.872–923, 1994.

[LAM 99] LAMPORT, L. Specifying concurrent systems with tla+. In: BROY, M.; STEINBRÜGGEN, R. (Ed.). **Calculational system design**. Amsterdam: IOS Press, 1999. p.183–247.

[LAP 85] LAPRIE, J. C. Dependable computing and fault tolerance: concepts and terminology. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 1985. **Proceedings...** Sylver Spring:IEEE Computer Society, 1985. v.15, p.2–11.

[LEE 91] LEE, I.; IYER, R. K.; TANG, D. Error/failure analysis using event logs from fault tolerant systems. In: INTERNATIONAL SYMPO-

SIUM ON FAULT-TOLERANT COMPUTING, 21., 1991. **Digest of Papers...** [S.l.: s.n.], 1991. p.10–17.

[LIN 98] LIN, T. H.; SHIN, K. G. Damage assessment for optimal rollback recovery. **IEEE Transactions on Computers**, New York, v.47, n.5, p.603–613, May 1998.

[MAN 90] MANNA, Z.; WALDINGER, R. **The logical basis for computer programming**: deductive systems. Reading, Massachusetts: Addison-Wesley Publishing Company, 1990. 642p. v.2.

[MAN 92] MANNA, Z. **The temporal logic of reactive and concurrent systems**: specification. New York: Springer-Verlag, 1992. 427p.

[MAN 97] MANIVANNAN, D.; NETZER, R. H. B.; SINGHAL, M. Finding consistent global checkpoints in a distributed computation. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.8, n.6, p.623–627, June 1997.

[NET 95] NETZER, R. H. B.; XU, J. Necessary and sufficient conditions for consistent global snapshot. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.6, n.2, p.165–169, Feb. 1995.

[NEV 98] NEVES, N. F. **Time-based coordinated checkpointing**. 1998. Thesis - Department of Computer Science, University of Illinois at Urbana-Champain, Urbana, Illinois.

[OLI 97] OLIVEIRA, R.; GUERRAOUI, R.; SCHIPER, A. **Consensus in the crash-recover model**. Lausanne, Switzerland: EPFL – Départment d'Informatique, 1997. (Technical-Report 97-239).

[PLA 98] PLANK, J. S.; LI, K.; PUENING, M. A. Diskless checkpointing. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.9, n.10, p.972–986, Oct. 1998.

[PRA 96] PRAKASH, R.; SINGHAL, M. Low-cost checkpointing and failure recovery in mobile computing systems. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.7, n.10, p.1035–1048, Oct. 1996.

[SCH 83] SCHLICHTING, R. D.; SCHNEIDER, F. B. Fail-stop processors: an approach to designing fault-tolerant computing systems. **ACM Transactions on Computer Systems**, New York, v.1, n.3, p.222–238, Aug. 1983.

[SIL 92] SILVA, L. M.; SILVA, J. G. Global checkpointing for distributed programs. In: SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, 11., 1992, Houston, Texas. **Proceedings...** [S.l.]:IEEE Computer Society Press, 1992. p.155–162.

[STR 85] STROM, R.; YEMINI, S. Optimistic recovery in distributed systems. **ACM Transactions on Computer Systems**, New York, v.3, n.3, p.204–226, Aug. 1985.

[VEN 97] VENKATESAN, S.; JUANG, T. T.-Y.; ALAGAR, S. Optimistic crash recovery without changing application messages. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.8, n.3, p.263–271, Mar. 1997.

[WAN 95] WANG, Y.-M. et al. Checkpointing and its applications. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 25., 1995. **Digest of Papers...** [S.l.: s.n.], 1995. p.22–31.

[WAN 95a] WANG, Y.-M. et al. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.6, n.5, p.546–554, May 1995.

[WAN 97] WANG, Y.-M. Consistent global checkpoints that contain a given set of local checkpoints. **IEEE Transactions on Computer**, New York, v.46, n.4, p.456–468, Apr. 1997.

[WU 93] WU, K.-L.; FUCHS, W. K. Rapid transaction-undo recovery using twin-page storage management. **IEEE Transactions on Software Engineering**, New York, v.19, n.2, p.155–164, Feb. 1993.

[XU 93] XU, J.; NETZER, R. H. B. Adaptative independent checkpointing for reducing rollback propagation. In: IEEE SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING, 5., 1993. **Proceedings...** [S.l.: s.n.], 1993. p.754–761.