SBA: Controle & Automação, Vol. 2, Nº 1, pp. 47-55, (TC)

REPLICAÇÃO AUTOMÁTICA E TRANSPARENTE DE MÓDULOS EM SISTEMAS DISTRIBUÍDOS

Orlando G. Loques Filho

Grupo de Sistemas de Computação
Departamento de Engenharia Elétrica - PUC/RJ
Cx.P. 38063 - CEP 22452 - Rio de Janeiro - Brasil

Resumo

A redundância de recursos (computadores) inerente em sistemas distribuídos de computação pode ser usada para prover tolerância a falhas. Usualmente isto requer que os módulos do sistema aplicativo sejam especialmente projetados, o que dificulta a re-utilização dos mesmos e aumenta o custo total do sistema. Este trabalho discute os problemas associados ao suporte automático de técnicas de replicação de módulos, transparentes ao nível de programação, em um ambiente distribuído de computação. As hipóteses e opções de projeto que têm que ser consideradas na concepção de qualquer sistema distribuído tolerante a falhas são analisadas. Uma técnica para suporte de replicação passiva de módulos selecionados é apresentada. Esta técnica permite que os módulos sejam projetados sem se ter tolerância a falhas em mente; em um segundo estágio do desenvolvimento do sistema eles podem ser transformados de modo a se obter esta capacidade. Finalmente, algumas considerações relativas a técnicas de replicação ativa de módulos são também apresentadas.

Automatic and Transparent Module Replication in Distributed Systems

Abstract

The inherent redundancy of resources (computers) existing in distributed computer systems can be used to achieve fault tolerance. This usually requires that the modules be specially designed, which makes it difficult to reuse them and raises the total cost of a system. This work discusses the issues related to the support of automatic and transparent module replication at the software level in a distributed computer environment. The assumptions and design options that must be considered in any fault tolerant distributed system design are analysed. An approach which supports passive replication of selected modules is presented. This approach allows the modules to be designed without fault tolerance in mind; at a second stage of system development they can be transformed to achieve that capability. Finally, some considerations concerning active replication based approaches are also presented.

1. INTRODUÇÃO

Os sistemas distribuídos possuem características naturais que facilitam a obtenção de tolerância a falhas. O fraco acoplamento entre os processadores do sistema ajuda a isolar o efeito das falhas. Além disso, a redundância de recursos (computadores) existente permite a implementação de estratégias de reconfiguração para a recuperação de falhas. Os componentes críticos podem ser replicados e alocados em estações diferentes, e técnicas de tolerância podem ser usadas para o mascaramento de falhas dos mesmos.

O nosso objetivo é o suporte de aplicações de tempo real, tais como as de controle e automação, as quais podem exigir garantia de operação contínua mesmo em presença de falhas, desta forma, requerendo o uso de técnicas de tolerância a falhas; exemplos práticos de sistemas tolerantes a falhas podem ser encontrados em (Kirrman 87, Turner 87). Em geral, o suporte de técnicas de tolerância a falhas é caro em termos dos recursos adicionais necessários e do esforço de programação requerido, e o uso das mesmas pode causar uma queda acentuada do desempenho em relação ao normalmente obtido em um sistema convencional. Portanto torna-se atrativo usá-las de maneira **seletiva** e **localizada**:

(i) somente os módulos que necessitam tolerância devem pagar por ela;

(ii) técnicas especiais de programação ou restrições à mesma não devem ser impostas;

(iii) técnicas distintas de tolerância a falhas devem ser disponíveis para suprir diferentes requisitos de segurança de funcionamento ("dependability").

Em um primeiro estágio, o projeto e o teste do comportamento lógico do sistema da aplicação devem ser independentes de considerações sobre tolerância a falhas. Em um segundo estágio de desenvolvimento deve ser possível adicionar esta capacidade, aonde necessário, porém sem necessidade de reprojetar os módulos de aplicação. Isto permite o uso de técnicas e ferramentas padronizadas no desenvolvimento de sistemas que tenham requisistos de segurança de funcionamento diversos. Além disso, esta transparência permite a re-utilização de módulos de programa, facilitando a construção modular de sistemas.

A disponibilidade de diferentes técnicas de tolerância a falhas padronizadas e independentes das aplicações em um mesmo ambiente de desenvolvimento e suporte de operação, é bastante atrativa. Os mecanismos de suporte associados podem ser desenvolvidos uma única vez, assegurando que seu projeto e implementação sejam corretos. Se for possível o seu reaproveitamento em outras aplicações, o custo total de desenvolvimento de cada sistema será reduzido. O uso de replicação ao nível dos módulos de software facilita a obtenção das referidas qualidades.

Neste artigo, após delinearmos o ambiente de software que serve de base para a construção de sistemas aplicativos (seção 2), discutimos as hipóteses fundamentais que têm que ser consideradas no projeto de qualquer sistema distribuído tolerante a falhas (seção 3). Uma técnica transparente de replicação passiva é apresentada na seção 4. Na seção 5 é discutido o relaxamento das hipóteses de erro previamente assumidas, sendo também tecidas considerações sobre o suporte de replicação ativa. Comentários e conclusões finais são apresentados na seção 6.

2. AMBIENTE DE SOFTWARE

A metodologia de projeto de software adotada permite que um sistema seja composto por um conjunto de módulos (Kramer 83), cada qual contendo variáveis próprias (locais) e encapsulando uma tarefa (ou processo). O uso de variáveis globais é restringido, os módulos comunicam-se com outros módulos através de uma interface definida por portas, usadas para a troca de mensagens. A comunicação é usualmente feita por meio de transações síncronas do tipo pedido-resposta ("request-reply"), similares a chamadas remotas de procedimento (Birrel 84). Contudo, uma transação assíncrona também é disponível oferecendo flexibilidade para a programação de situações especiais.

Os módulos de software podem ser desenvolvidos e testados independentemente da configuração de sistema aonde serão usados. Por exemplo, (a) a especificação de comunicações locais e remotas é transparente ao nível da programação; (b) as portas de comunicação são objetos encapsulados localmente, visíveis apenas no interior de cada módulo, assim evitando a interdependência entre módulos. Uma linguagem especial de configuração permite: a especificação de tipos de módulos (contexto) a partir dos quais o sistema será construído; a declaração das instâncias de módulos que serão criadas no sistema; a interligação de suas portas de comunicação, bem como o mapeamento das instâncias às estações. Os mecanismos de suporte de operação permitem que o conjunto de módulos de um sistema seja reconfigurado durante o funcionamento do mesmo (Kramer 85). Estas características permitem a especificação e o suporte de diferentes políticas de replicação de módulos e facilitam a obtenção da tolerância a falhas.

3. HIPÓTESES E OPÇÕES DE PROJETO

No projeto de qualquer sistema tolerante a falhas deve-se decidir o grau de cobertura a ser considerado. Isto acarreta, na prática, que um mecanismo de provisão de tolerância a falhas possa ser simplificado se for assumido que outros mecanismos são disponíveis e responsáveis pelo tratamento de outros tipos de falhas. Por exemplo, o sistema de comunicação é projetado de modo a tolerar falhas de comunicação.

Nesta seção são analisadas algumas hipóteses que têm que ser consideradas em qualquer projeto. Elas se apresentam em pares opostos de opções, sendo uma decisão de projeto prover algum meio independente para assegurar a opção, ou usar alguma técnica de tolerância específica para o tratamento da classe de erros associada à mesma.

i) (a) Sistema de Comunicação Confiável Vs. (b) Não-Confiável

A opção (i-a) requer que a confiabilidade do sistema de comunicação seja suficiente de modo a não comprometer a confiabilidade do sistema aplicativo. Em geral, é assumido que mensagens em trânsito não são corrompidas ou perdidas e que o sistema de comunicação não se torna particionado. A corrupção e perda de mensagens pode ser tratada através do uso de códigos e protocolos, embora existam sérias discussões

quanto ao nível de uso dos mesmos, vide (Saltzer 81); o particionamento pode ser prevenido pelo uso de canais redundantes e algoritmos de roteamento adequados. O requisito de operação contínua faz com que a hipótese (i-a) seja obrigatória em nossa área de aplicação. Foi demostrado em (Fisher 83) que o problema de consenso confiável em um ambiente distribuído não tem solução em sistemas nos quais o tempo de transporte das mensagens não é limitado. O projeto de algoritmos que funcionam sob condições de particionamento da rede de comunicação é bastante complexo, veja (Davidson 85), para uma revisão do assunto.

ii) (a) Projeto Correto Vs. (b) Erros de Projeto

A opção (ii-a) implica na inexistência de erros de projeto seja nas facilidades fornecidas pelo sistema ou nos módulos da aplicação. Desta forma, falhas de módulos não ocorrem devido a erros de projeto, por exemplo, mudanças de estado e mensagens erradas não são geradas. A falha de módulos pode ocorrer devido a defeitos de hardware. A validade desta opção depende do uso de técnicas e ferramentas de projeto de software adequadas, (Adrion 82). Embora uma prova inteiramente formal seja trabalhosa e cara, ela pode ser realizada se for essencial, (Melliar-Smith 82). Alternativamente, técnicas tais como a de afirmações executáveis ("executable assertions") podem ser usadas para detectar e confinar erros de projeto dentro de cada módulo, (Andrews 79). A natureza estática dos sistemas de controle facilita os testes dos mesmos também ajudando a validar a opção (ii-a).

iii) (a) Confinamento de Erros Vs. (b) Propagação de Erros

Os módulos do sistema se comunicam exclusivamente pela troca de mensagens. A opção (iii-a) implica em que a falha de um módulo não causa a falha de outro módulo através da propagação de erros, i.e., erros não se propagam através de mensagens. Se um módulo falha, ele pára suas atividades, comportamento "fail-stop", conforme definido em (Scheneider 83). Note que esta hipótese não impede a recuperação transparente de falhas por mecanismos localizados em cada estação. A validade da opção (iii-a) depende dos mecanismos de detecção de erros disponíveis em cada estação, várias técnicas existentes podem ser usadas, e.g., (Andrews 79, Chavade 82, Rennels 78). Um módulo testador especial, que executa periodicamente ou sempre que o processador estiver desocupado, pode ajudar a reduzir a latência de detecção de erros. O custo atual do "hardware", faz com que seja economicamente viável considerar a replicação de processadores, os quais são operados em paralelo formando pares de comparação mútua (Hendrie 83); isto provê um alto grau de confinamento de erros.

O conjunto de hipóteses escolhido possui enorme influência no projeto dos mecanismos de suporte. Como mencionado, um sistema de comunicação confiável, opção (i-a), é essencial. É interessante notar que em princípio a opção (iii-a) engloba as opções da hipótese (ii), ou seja, qualquer que seja o tipo de erro causador da falha, a propagação de mensagens erradas não será tolerada. A separação explícita dos dois conjuntos de hipóteses facilita a apresentação e concepção de sistemas tolerantes a falhas.

Na próxima seção será apresentada uma técnica de replicação passiva que funciona sob o conjunto de opções definido por (ii-a, iii-a). Na seção 6 será discutida a relaxação destas hipóteses e os efeitos decorrentes no projeto. São também analisadas algumas técnicas, baseadas em redundância ativa, que podem suprir os requisitos sob o conjunto de opções mais difíceis: (ii-b) e (iii-b).

4. REDUNDÂNCIA PASSIVA DE MÓDULOS

A já bem estabelecida técnica de recuperação de falhas por retorno ("backward error recovery"), (Anderson 81), pode ser usada em diferentes aplicações. Contudo, seu uso dificulta a obtenção de transparência ao nível de programação: a definição explícita de ações e esquemas de conversação são normalmente requeridos de maneira a evitar o efeito "dominó". Este efeito é basicamente causado pela possibilidade de propagação de erros entre módulos. O problema pode ser evitado se o confinamento de erros for assegurado. Isto permite grande simplificação nos mecanismos de suporte normalmente requeridos. Além disso, uma técnica automática de salvamento de estado pode ser usada para a recuperação transparente (ou mascaramento) de falhas.

A técnica para o mascaramento de falhas de módulos baseia-se em uma extensão do conceito de reserva quente ("hot stand-by"), (Bartlet 81), e usa o método de recuperação de falhas por retorno. Duas ou mais instâncias de cada módulo são necessárias: uma ativa, processando as mensagens da aplicação; outra(s) passiva(s) fornecendo uma memória estável (à prova de falhas) para manter uma cópia atualizada do estado da aplicação produzido pela instância ativa. Sem perda de generalidade, assumiremos que somente uma instância passiva é usada (figura 1). Durante a operação o sistema pode dinamicamente criar uma nova instância para substituir uma instância em falha. A instância passiva está completamente preparada para assumir o processamento. Assim, o tempo de recuperação é limitado pelo tempo de detecção da falha da instância ativa. A informação de estado é transferida da instância ativa para a instância passiva em determinados pontos da operação da primeira. A detecção de falhas, reintegração

e seleção do modo de operação (ativo ou passivo) de instâncias e a transferência de estado entre instâncias são da responsabilidade dos módulos de gerenciamento associados a cada instância.

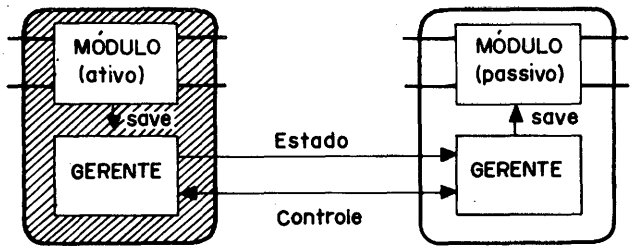

Figura 1

De acordo com a hipótese (iii-a), a primeira consequência da falha de um módulo será a interrupção de sua atividade. Portanto, considerando (i-a) e (iii-a) a falha de um módulo pode ser detectada por meio de medidas de tempo tomadas localmente por outro módulo. Nesta técnica a hipótese (ii-a) assegura que a falha de um módulo não se repita também em seu reserva devido a um erro de projeto. Ela também garante que mensagens erradas não serão geradas na ausência de outros erros, embora isto seja também assegurado pela hipótese (iii-a). Embora nem sempre explicitamente mencionado, hipóteses similares são adotadas em várias propostas contemporâneas, e.g., (Bartlet 81, Schneider 81, Borg 83, Cooper 86).

4.1 Primitivas para Tolerância a Falhas

Duas primitivas especiais são disponíveis para o suporte da técnica de mascaramento de falhas:

i) **Save:** é usada para a transferência de estado para a memória estável. Esta operação transfere informação suficiente para assegurar que todo o processamento do módulo efetuado após a mesma possa ser repetido em caso de re-execução decorrente de falha; isto pode incluir o re-envio de mensagens. A transferência de estado é efetuada por um mecanismo auxiliar, que copia o estado corrente do módulo e o transfere através de uma mensagem para a memória estável implementada pela instância passiva. Uma mensagem de reconhecimento é retornada para indicar a execução correta da primitiva **save**. O fato de que mensagens ou são recebidas completamente ou não são recebidas de forma alguma assegura a atomicidade da atualização. Desta forma, a falha da instância ativa durante a execução de uma primitiva **save** não deixa a instância passiva em um estado inconsistente. Se uma instância passiva não for disponível, o controle é imediatamente retornado à instância que invocou a transferência.

Na seção 4.3 é apresentado um conjunto de regras que podem ser usadas para identificar os pontos nos quais a primitiva **save** deve ser obrigatoriamente invocada de modo a assegurar a recuperação transparente. É importante notar que com o uso destas regras as chamadas à primitiva **save** podem ser automaticamente introduzidas no código do módulo.

ii) **Portas Confiáveis:** também é necessário mascarar a falha e a recuperação subsequente de um módulo de outros módulos com os quais ele se comunica. Por exemplo, considere um módulo que salva seu estado e em seguida falhe durante ou após uma transação com outros módulos. Se este era o módulo iniciante da transação, então esta poderá ser repetida pelo módulo que estava em reserva. Se este era o módulo destinatário, a transação poderá ser perdida. O mascaramento de transações repetidas ou perdidas é garantido através do uso de uma transação especial do tipo pedido-resposta confiável. Este serviço é selecionado pelo uso de portas de comunicação confiáveis. Os nomes das portas confiáveis são declarados separadamente em um arquivo associado a especificação da configuração do sistema.

Deve ser notado que os mecanismos que fornecem suporte para transações confiáveis e mascaramento de falhas podem ser introduzidos após o estágio convencional de programação. Isto é feito por meio de transformações no programa do módulo de modo a incluir as chamadas à primitiva "save" e pela seleção das portas confiáveis no estágio de configuração do sistema. Além da identificação das portas confiáveis, nossa técnica é completamente transparente ao nível de programação da aplicação. Assim, restrições ao estilo de programação não são compulsoriamente introduzidas.

4.2 Transação Pedido-Resposta Confiável

A transação confiável é similar à transação "pedido-resposta" padrão, sendo confiável no sentido de que se ela completa, ela assegura uma semântica "exactly-once", (Birrell 84), mesmo no caso de uma falha de qualquer instância dela participante. Este comportamento garante exatamente uma execução do pedido pelo módulo servidor e um recebimento da resposta correspondente pelo módulo cliente. Assim, um sistema de módulos comunicando-se puramente por transações do tipo pedido-resposta pode ser automaticamente feito tolerante a falhas. A figura 2, generaliza a execução de uma transação. Qualquer das instâncias envolvidas pode falhar durante a transação. No caso da falha de uma instância ativa, a instância passiva assume o controle e completa a transação. A recuperação pode requerer a retransmissão de mensagens. Dois casos devem ser considerados: (i) a falha do cliente seguida de sua recuperação pode gerar um pedido duplicado, este pedido deve

ser filtrado no lado do servidor. Se a resposta associada já tiver sido gerada pela tarefa servidora, esta mesma resposta deve ser retransmitida para o cliente; (ii) a falha do servidor seguida de sua recuperação pode resultar no reprocessamento do mesmo pedido. Se a resposta correspondente já tiver sido gerada, então uma resposta duplicada será gerada tendo que ser filtrada pelo cliente.

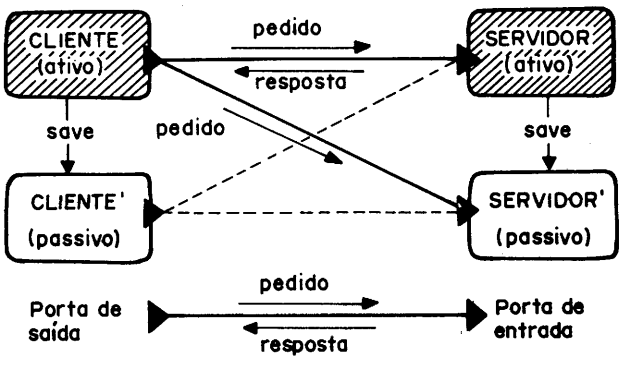

Figura 2

4.3 Controle Automático da Primitiva Save

A técnica de recuperação se baseia na repetição das saídas (pedidos ou respostas) produzidas pelos módulos durante suas re-execuções. O consumo de mensagens altera o estado do módulo. Portanto é necessário assegurar que uma mensagem processada por uma instância ativa seja também processada por sua réplica passiva durante uma fase de recuperação, de outra forma resultados diferentes podem ser produzidos. Duas regras são suficientes para assegurar a consistência da recuperação:

R1: **Uma transferência (save) tem que ser executada após uma mensagem ser consumida por uma porta de entrada confiável, antes que qualquer resultado decorrente do processamento desta mensagem seja enviado para fora do módulo.**

R2: **Uma transferência tem que ser realizada entre dois envios de mensagens sucessivos através da mesma porta de saída confiável.**

A regra R1 assegura a repetição de respostas produzidas como resultado do consumo e processamento de mensagens tipo pedido. A regra R2 assegura que não mais que uma transação confiável será repetida através da mesma porta de saída no caso em que o módulo cliente re-executa; isto impediria a recuperação das respostas mais antigas. Caso requerido, a regra R2 pode ser generalizada para uma execução de save após n transações. Neste caso as cópias das respostas correspondentes às n prévias transações devem ser armazenadas pelo módulo servidor. A regra R1 também foi projetada para o caso de transações aninhadas, como exemplificado na figura 3. Pode ser visto que em qualquer caso de falha, a ação iniciada por ped-1 completa-se consistentemente sem qualquer save adicional. Por exemplo, considere que o módulo-2 falhou algum tempo depois de executar a primitiva save. Durante a recuperação, o módulo-2 executa novamente re-generando ped-3. Se ped-3 já tiver sido gerada durante a execução prévia, ela será filtrada. Se o módulo-3 já havia produzido a resposta associada, resp-3, então o mecanismo de suporte recupera esta resposta retornando-a ao módulo-2. Se, antes da falha, a resp-2 já tiver sido gerada e consumida pelo módulo-1, então sua duplicata será descartada pelo mesmo. Um exemplo de programa completo justificando as regras para uso da save está disponível em (Loques 86).

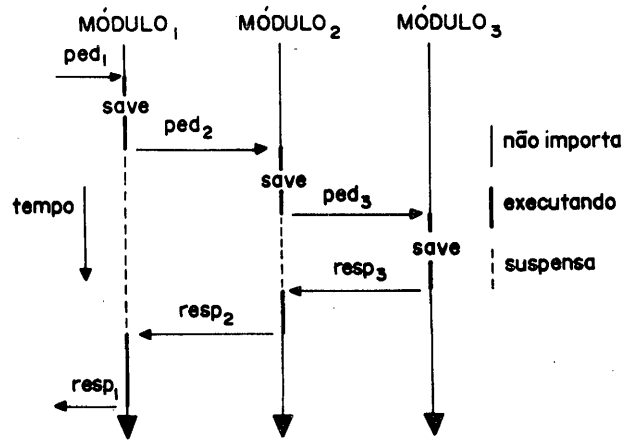

Figura 3

O mecanismo de suporte de execução para a primitiva save, é implementado através de módulos padronizados de gerenciamento (ver figura 1), os quais são automaticamente associados aos módulos replicados no estágio de especificação do sistema. As chamadas a primitiva save são transformadas em transações pedido-resposta normais (não tolerantes) com o módulo de gerenciamento. Este módulo adquire a informação a ser transferida na área de dados do módulo da aplicação e a envia para o módulo de gerenciamento da instância passiva. A repetição de saídas geradas em ativações sucessivas de um mesmo módulo é obtida através da transferência do conteúdo da área de trabalho do módulo invocador da primitiva "save". Alguma redução na quantidade de informação a ser transferida

pode ser obtida. Por exemplo, somente as variáveis que tiverem sido alteradas desde a última invocação da **save** precisam ser transferidas. Uma outra opção seria perder alguma transparência e requerer do programador a especificação do estado a ser transferido. Na prática, pode-se notar que a maioria dos módulos em sistemas embutidos possuem pouca informação de estado.

4.4 Comentários

A capacidade de tolerância a falhas, é obtida para um sistema de módulos comunicando-se através de transações pedido-resposta síncronas. Este tipo de transação é similar à invocação de um procedimento, no sentido de que o pedido especifica os argumentos de entrada e a resposta especifica os resultados da execução do procedimento. A programação de sistemas distribuídos usando o paradigma pedido-resposta/chamada-remota-de-procedimento é certamente não restritiva e tornou-se largamente adotada, (Birrel 84).

A metodologia de software utilizada permite que módulos sejam projetados independentemente. Um sistema de módulos é então especificado através de uma linguagem de configuração separada. Desta forma, um sistema pode ser programado sem qualquer preocupação com tolerância a falhas e transformado em seguida de modo a obter esta capacidade. Isto também possibilita a re-utilização de módulos.

De modo a permitir maior flexibilidade, a primitiva **save** é também disponível para uso explícito ao nível da linguagem de programação. Por exemplo: (i) o programador pode fazer uso de portas normais e confiáveis no mesmo módulo, possibilitando assim comunicação com módulos não replicados; (ii) um módulo pode salvar resultados parciais para minimizar a possibilidade de repetir computações longas em tempo. Nestes casos, o uso explícito da primitiva **save** pode ser útil.

O sistema Tandem (Bartlet 81), foi pioneiro no uso de replicação passiva de módulos para a obtenção de tolerância a falhas. Contudo, ele não provê uma técnica padronizada para recuperação, desta forma requerendo a intervenção do programador. Além disso, a interface de um módulo é definida por uma única fila de mensagens. Isto impede uma seleção não determinística das mensagens de entrada e também a seleção condicional do recebimento das mesmas através do uso de guardas lógicos, como suportado por nossa metodologia. Esta falta de flexibilidade pode tornar difícil a programação de algumas aplicações. Em adição, as interligações para comunicação são especificadas pela nomeação direta do módulo servidor pelo módulo cliente: isto restringe a re-utilização de módulos, (Kramer 85).

O suporte deste esquema transparente de replicação passiva está integrado na arquitetura de um sistema distribuído tolerante a falhas, descrita em (Loques 86). Esta referência também descreve os mecanismos de suporte de recuperação e provê uma comparação com outros trabalhos na área. O sistema também suporta um serviço de replicação do tipo reserva fria ("cold stand-by"). Este tipo de replicação não preserva estado entre sucessivas encarnações de instâncias, garantindo apenas a criação de uma nova instância, sendo no entanto adequado para algumas aplicações. Um gerente de configuração tolerante a falhas pode realizar atividades de reconfiguração durante a operação do sistema; ele é responsável pelo controle dos diferentes serviços de replicação. Dado que a replicação usa recursos de espaço e tempo, isto também permite um balanceamento entre segurança de funcionamento e desempenho: a replicação de módulos pode ser usada somente no estágio de operação em que for requerida.

5. OPÇÕES ALTERNATIVAS

O esquema anterior de replicação passiva funciona sob duas hipóteses implícitas: (ii-a) projeto correto e (iii-a) confinamento de erros. O uso de técnicas de tolerância baseadas na diversidade de projeto (Horning 74, Chen 78, Anderson 81), dentro de um módulo, pode permitir nossa proposta funcionar sob a opção (ii-b), erros de projeto. Contudo, neste caso, as mensagens geradas na interface de um módulo devem ser obrigatoriamente repetidas em caso de re-execução. A obtenção desta propriedade requer considerações especiais para a programação do módulo.

A opção (iii-b), i.e., propagação de erros, significa que um módulo pode ter vários modos de falha: as mensagens podem ser atrasadas, adiantadas, perdidas, claramente erradas, ou aparentemente corretas mais de fato erradas, i.e., bizantinas como nomeadas em (Lamport 82). O esquema de replicação passiva pode funcionar sob todos estes modos com exceção do bizantino; para isso seriam requeridos protocolos mais especializados de comunicação.

Replicação Ativa

Existem outras propostas que podem funcionar sob hipóteses de falha mais gerais. Elas se baseiam em extensões do esquema clássico TMR ("triple modular redundancy"), no qual todos os módulos replicados são ativos consumindo mensagens e realizando processamento (figura 4). Algum mecanismo de votação (ou filtragem) de mensagens é requerido para a obtenção de um único resultado livre de erros, a partir de mensagens replicadas de um mesmo conjunto de saída. Este resultado correto, deve ser

independentemente consumido e processado por um conjunto de réplicas destinatárias. Uma serialização idêntica das mensagens a serem consumidas e processadas por cada réplica destinatária é também necessária para manter seus estados consistentes.

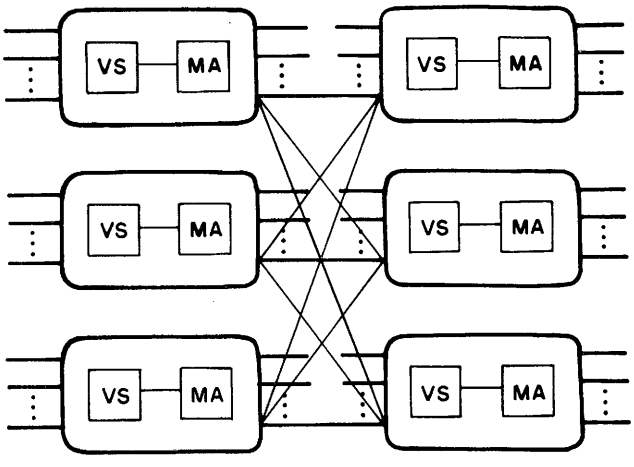

Figura 4

A obtenção dos requisitos acima não é trivial em um ambiente distribuído de computação. Deve ser notado que as soluções conhecidas implicam em retardos no processamento das mensagens e reduzem o não-determinismo normalmente desejado. Uma vantagem do esquema de replicação ativa, em relação ao de replicação passiva, é que a recuperação de uma falha não acarreta a degradação do desempenho em tempo do sistema.

A tolerância de erros de projeto, opção (ii-b), requer versões dissimilares dos algoritmos dos módulos ativos replicados. Neste caso cada réplica pode produzir valores aproximadamente idênticos. Isto pode acontecer devido aos diferentes algoritmos e/ou características inerentes aos computadores, como por exemplo: (a) resultados de computações em ponto flutuante dependem dos algoritmos usados; (b) leituras de sensores podem normalmente diferir. Aqui, um algoritmo especial de votação pode ser necessário (Pease 80), embora dependendo da aplicação soluções "ad hoc" sejam convenientes (Frison 82). A propagação de erros comuns pode ser tratada através de replicação e algoritmos de votação simples. Em princípio, a pior hipótese seria o tratamento de erros de projeto combinada com a propagação de erros no modo bizantino. Na prática, a técnica de filtragem adotada depende também das características e modos de falha do sistema de comunicação, (Powel 87).

Um exemplo bem conhecido do uso de redundância ativa pode ser encontrado no sistema SIFT, (Wensley 78). Os requisitos para a obtenção de consistência são cumpridos através da rigorosa sincronização da votação e consumo das mensagens com uma referência global de tempo. Isto resulta em baixa modularidade causada pela forte interação temporal existente entre todos os módulos do sistema (similar à existente em circuitos sequenciais construídos em hardware).

Uma opção atraente é usar protocolos de disseminação confiável ("reliable broadcast protocols"), (Chang 84, Babaoglu 85, Cristian 85). As duas últimas propostas podem garantir tempo de transporte de mensagem limitado, mais a atomicidade e ordenação do recebimento das mesmas. A primeira proposta (Chang 84) não garante qualquer limite de tempo para a finalização do protocolo (Cristian 87), sendo assim inadequada para aplicações de tempo real.

As propriedades dos protocolos de disseminação confiável asseguram que uma mensagem transmitida por um módulo em bom funcionamento será disponível na mesma ordem para todos os módulos replicados destinatários. Isto facilita o suporte de técnicas transparentes de replicação de módulos. Contudo, para que a implementação seja eficiente, uma forte interação com o projeto do sistema de comunicação e suas propriedades se torna necessária. Nesta etapa de nosso trabalho, estamos planejando a introdução no sistema de uma técnica transparente de replicação ativa de módulos, baseada em disseminação confiável. Isto permitirá a especificação, no nível de configuração, da técnica de replicação a ser usada por um módulo, assim facilitando o alcance das metas mencionadas na introdução.

6. COMENTÁRIOS FINAIS

O esquema de replicação passiva permite o uso de mecanismos bastante simples para a obtenção da tolerância a falhas sob hipóteses de falha restritas mais aceitáveis. Estações construídas com circuitos autotestáveis podem assegurar a hipótese de confinamento de erros. O custo atual do hardware e as vantagens resultantes do confinamento de erros tornam a opção muito atrativa. Uma desvantagem deste esquema decorre da degradação limitada do desempenho em tempo que pode ocorrer devido a re-execução do processamento dos módulos durante a recuperação. Estas características fazem que esta técnica seja indicada para sistemas requerendo alta disponibilidade mas sem requisitos muito rigorosos de resposta em tempo.

A replicação ativa de módulos pode ser empregada em aplicações possuindo requisitos de alta confiabilidade e restrições de resposta em tempo mais severas. No projeto

destes sistemas hipóteses mais gerais de falha são geralmente assumidas. Contudo, os recursos adicionais e os custos das técnicas de tolerância que têm que ser usadas para assegurar a consistência de estado devem ser também considerados.

As hipóteses de falha assumidas para os módulos e as propriedades específicas do sistema de comunicação influem diretamente nas soluções e custo de implementação da replicação de módulos. Mais trabalhos de modelagem, implementação, e experimentação são necessários para a obtenção de medidas efetivas da capacidade de cada uma das técnicas. A disponibilidade de ambas em um ambiente uniforme deverá facilitar a realização deste objetivo. Isto também deverá permitir que um projetista selecione a técnica mais apropriada para a obtenção da tolerância a falhas para o sistema da aplicação. Um resumo do esforço realizado na PUC/RJ neste sentido está disponível em (Loques 87).

7. REFERÊNCIAS

(Adrion 82) Adrion, W.R., Branstad, M.A., and Cherniavsky, J.C., Validation, Verificaton, and Testing of Computer Software, ACM Computing Surveys, Vol. 14, No. 2, junho 1982, p. 159-193.

(Anderson 81) Anderson, W., and Lee, P., Fault Tolerance Principles and Practice, Prentice Hall International, 1981.

(Andrews 79) Andrews, D.M., Using Executable Assertions for Testing and Fault-Tolerance, 9th Symposium on Fault-Tolerant Computing, junho 1979, p. 102-105.

(Babaoglu 85) Babaoglu, O., Drummond, R., Streets of Byzantium: Network Architectures for Fast Reliable Broadcasts, IEEE Trans. on Software Engineering, Vol. SE-11, N. 6, junho 1985, p. 546-554.

(Bartlet 81) Bartlet, J., A Non-Stop Kernel, 8th Symposium on Operating Systems Principles, dezembro 1981, p. 22-29.

(Birrell 84) Birrell, A.D., and Nelson, B.J., Implementing Remote Procedure Calls, ACM Trans. on Computer Systems, Vol. 2, N. 1, fevereiro 1984, p. 39-59.

(Borg 83) Borg, A., Baubach, J., and Glazer,S., A Message System Supporting Fault Tolerance, Proceedings of the Ninth Symposium on Operating Systems Principles, Bretton Woods, New Hampshire, outubro 83, p. 90-99.

(Chang 84) Chang, J. M., and Maxemchunmk N. F., Reliable Broadcast Protocols, ACM Transactions on Computer Systems, Vol. 2, N. 3, 1984, p. 251-273.

(Chavade 82) Chavade, J., and Crouzet, Y., The P.A.D: A Self-Checking Circuit for Fault-Detection in Microcomputers, 12th Symposium on Fault-Tolerant Computing, junho 1982, p. 55-62.

(Chen 78) Chen, L., and Avizienis, A., N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation, 8th Symposium on Fault-Tolerant Computing, junho 1978, p. 3-9.

(Cooper 85) Cooper, E.C., Replicated Distributed Programs, Proceedings of the Tenth ACM Symp. on Operating Systems Principles, Washington, U.S.A., dezembro de 1985, p. 63-78.

(Cristian 85) Cristian F., Aghili H., Strong R., e Dolev D., Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement, 15th International Symposium on Fault-Tolerant Computing, Ann Arbor, Michigan, USA, junnho de 1985, p. 200-206.

(Cristian 87) Cristian, F., Issues in the Design of Highly Available Computing Systems, Research Report, IBM Almaden Research Center, San Jose, California, USA, outubro de 1987.

(Davidson 85) Davidson, S. B., Garcia-Molina, H., Skeen, D., Consistency in Partitioned Networks, ACM, Computing Surveys, Vol. 17, N. 3, setembro 1985, p.341-370.

(Fisher 83) Fisher, M. J., Lynch, N. A., and Paterson, M., Impossibility of distributed consensus with one faulty process, in Proc. 2nd ACM Symp. Principles of Database Syst., março 1983.

(Frison 82) Frison, S., Wensley J., Interactive Consistency and its Impact on the Design of TMR Systems, 12th Symposium on Fault-Tolerant Computing, junho 1982, p. 228-233.

(Hendrie 83) Hendrie, G., A Hardware Solution to Part Failures Tottaly Insulates Programs, Eletronics, janeiro 27, 1983, p. 103-105.

(Horning 74) Horning, J.J., Lauer, H.C., Melliar-Smith, P.M., and Randell, B., A Program Structure for Software Fault-Tolerance, in Lecture Notes in Computer Science 16, (ed. Gelenbe and C. Kaiser), Springer-Verlag, Berlin (1974).

(Kirrmann 87) Kirrmann, H.D., Fault Tolerance in Process Control: An Overview and Examples of European Products, IEEE Micro, Vol. 7, N. 5, outubro 1987, p. 27-50.

(Kramer 83) Kramer, J., Magee, J., Sloman, M., and Lister,A., Conic: An Integrated Approach to Distributed Control Systems, IEE Proc., Vol. 130, Pt. E, N. 1, janeiro 1983, p. 1-10.

(Kramer 85) Kramer, J., and Magee, J., Dynamic Configuration for Distributed Systems, IEEE Trans. on Software Engineering, Vol. SE-11, N. 4, abril 1985, p. 424-435.

(Lamport 82) Lamport, L., Shostak, R., and Pease, M., The Byzantine Generals Problem, ACM Transactions on Programming Languages and Systems, Vol. 4, N. 3, julho 1982, p. 382-401.

(Loques 86) Loques Filho, O., Kramer, J., Flexible Fault-Tolerance in Distributed Systems, IEE Proceedings, V.133, Pt.E, N.6, novembro 1986, p. 319-331.

(Loques 87) Loques Filho, O., Leite, Julius C.B., Após: Ambiente para o Projeto e Operação de Sistemas, Io. Simpósio Brasileiro de Engenharia de Software, Petrópolis, outubro 1987, p. 154-156.

(Melliar-Smith 82) Melliar-Smith, P.M., and Schwartz, R.L., The Proof of SIFT, ACM SIGSOFT, Software Eng. Notes, Vol. 7, No. 1, janeiro 1982, p. 2-5.

(Pease 80) Pease, M., Lamport L., and Shostak, R., Reaching Agreement in Presence of Faults, Journal of ACM, V.27, N.2, 1980, p. 228-234.

(Powell 87) Powell, D. R., Fault-Tolerance in Delta-4, Research Report no. 87064, Laboratoire d'Automatique et d'Analyse des Systemes, Toulouse, fevereiro de 1987.

(Rennels 78) Rennels , D.A. , Avizienis, A., and Ercegovac, M., A Study of Standard Building Blocks for the Design of Fault-Tolerant Distributed Computing Control Systems, 8th Symp. on Fault-Tolerant Computing, junho 1978, p. 144-149.

(Saltzer 81) Saltzer, J. H., et. alli, End-to-End Arguments in System Design, Proc. Int. Conference on Distributed Computing Systems, Paris, Abril 1981, p. 509-512.

(Schneider 81) Schneider, F.B., and Schlichting, R., Towards Fault-Tolerant Process Control Software, 11th Symposium on Fault-Tolerant Computing, junho 1981, p. 48-55.

(Schneider 83) Schneider, F.B., Fail-Stop Processors, Digest of Papers, Spring COMPCON 83: 26th IEEE Int. Conf., fevereiro 83, p. 66-70.

(Turner 87) Turner, D.B., Burns, R.D., Hecht, H., Designing Micro-Based Systems for Fail-Safe Travel, IEEE Spectrum, fevereiro 1987, p. 58-63.

(Wensley 78) Wensley, J.H., Lamport, L., Goldberg, J., Green, M.W., Levitt, K.N., Melliar-Smith, P.M., Shostak, R.E., e Weinstock, C.B., The Design and Analysis of a Fault Tolerant Computer for Aircraft Control, Proceedings of the IEEE, outubro 1978, p. 1240-1254.