Determinação de Estados Globais Consistentes em Sistemas Distribuídos

Gustavo Maciel Dias Vieira
Luiz Eduardo Buzato

Relatório Técnico IC–99-09

Março de 1999

Determinação de Estados Globais Consistentes em Sistemas Distribuídos

Gustavo Maciel Dias Vieira Luiz Eduardo Buzato

Resumo

Algoritmos para a determinação de estados globais consistentes em um sistema distribuído têm grande importância para tarefas que envolvam monitorização e reconfiguração, como detecção de *deadlocks*, detecção de perda de *token*, depuração, coleta de lixo, entre outros. Este documento aborda aspectos teóricos da determinação de estados globais consistentes e descreve um ambiente de programação projetado e implementado para permitir a experimentação com algoritmos para obtenção de estados globais consistentes.

1 Introdução

Em um sistema distribuído por muitas vezes é necessário avaliar um predicado sobre o estado global do sistema. Porém, devido às incertezas dos atrasos das mensagens e da dificuldade de se manter um tempo global único, é difícil manter uma visão global consistente de uma aplicação sobre a qual um predicado possa ser corretamente avaliado [1]. Algoritmos para a determinação de estados globais consistentes (EGC) em um sistema distribuído têm grande importância para tarefas que envolvam monitorização e reconfiguração, como detecção de *deadlocks*, detecção de perda de *token*, depuração, coleta de lixo, entre outros.

O desenvolvimento destes algoritmos tem então uma grande importância, sendo uma parte importante deste processo: (i) uma etapa de estudo e aquisição de conhecimento teórico sobre esses algoritmos e (ii) etapa de projeto, implementação, testes e avaliação dos algoritmos estudados e/ou propostos durante a fase (i). Este documento está estruturado em função dessas etapas. A Seção 2 aborda aspectos teóricos da determinação de estados globais consistentes. A Seção 3 descreve o ambiente de programação projetado e implementado para permitir a experimentação com algoritmos para obtenção de EGC.

2 Estados Globais Consistentes

Nesta seção apresentamos aspectos teóricos relevantes ao problema de determinação de estados globais consistentes. Inicialmente o modelo computacional é apresentado, em seguida o conceito fundamental de precedência causal entre eventos e mecanismos para manter e disseminar a informação de precedência são discutidos. Definimos então o que caracteriza um estado global consistente e apresentamos algoritmos para a sua construção.

2.1 Modelo Computacional

Uma aplicação distribuída consiste em um conjunto de n processos seqüenciais ($p_0, p_1, \ldots, p_{n-1}$) que trabalham em conjunto para alcançar um objetivo comum. Estes processos são independentes, não existindo mecanismos de memória compartilhada, de acesso a um relógio global ou de sincronização de relógios, sendo a comunicação entre os processos realizada inteiramente via troca de mensagens sobre uma rede de comunicação.

O mecanismo de troca de mensagens garante que mensagens não são corrompidas, mas o tempo de entrega não é limitado e as mensagens podem ser entregues fora de ordem. A comunicação é feita por canais unidirecionais entre processos e nenhum processo está isolado dos demais.

Os processos executam de maneira estritamente seqüencial e a sua execução é modelada por uma seqüência de eventos, onde e_i^k representa o k-ésimo evento executado pelo processo p_i. Os processos mantêm um conjunto de variáveis locais que constituem o seu estado, sendo σ_i^k o estado do processo p_i após a execução do evento e_i^k.

A maneira usual de se representar graficamente a execução de uma aplicação distribuída é através de um diagrama espaço-tempo. Linhas horizontais representam os processos ao longo do tempo, que progride da esquerda para a direita. Eventos são representados como pontos sobre estas retas e mensagens trocadas por processos são representadas por setas ligando o evento de envio e o evento de recebimento da mensagem. Um exemplo de diagrama espaço-tempo é mostrado na Figura 1.

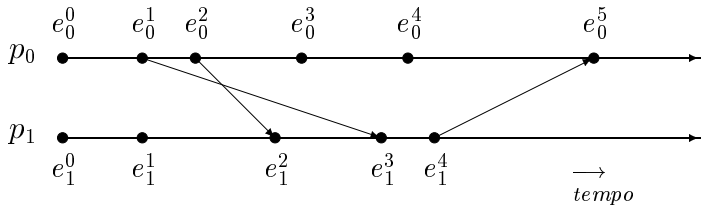

Figura 1: Diagrama espaço-tempo.

2.2 Ordenação de Eventos

Para se construir um estado global consistente é necessário estabelecer uma ordenação entre os eventos observados no sistema. A relação de precedência causal é suficiente para estabelecer uma ordenação consistente de eventos.

A precedência causal captura a dependência entre eventos. Dois eventos ligados por uma relação de causa e efeito tem uma ordem intrínseca que deve ser respeitada em uma ordenação consistente dos eventos do sistema. É dito então que um evento e precede causalmente outro evento e' se e pode ter influenciado a ocorrência de e'. A relação de precedência causal ($\rightarrow$) é definida da seguinte maneira [6]:

1. Se e_i^k e e_i^l são eventos de p_i e $k < l$, então $e_i^k \rightarrow e_i^l$;

2. Se para uma mensagem m, $e_i = \text{envio}(m)$ e $e_j = \text{recepção}(m)$, então $e_i \rightarrow e_j$;

3. Se $e \rightarrow e'$ e $e' \rightarrow e''$, então $e \rightarrow e''$.

Se dois eventos ocorrem em um mesmo processo, então a ordem de execução destes eventos neste processo indica a ordem que estes eventos devem ser observados em uma ordenação consistente de todos os eventos do sistema. Esta é uma regra natural e é proveniente da natureza seqüencial da execução dos processos. Considerando que o tempo de transmissão de uma mensagem é sempre maior que zero, então o evento de envio de uma mensagem deve preceder o evento de recepção desta mesma mensagem. Por último, a relação de precedência causal é transitiva. Se o evento de recebimento de uma mensagem é precedido pelo evento de envio, então todos os eventos subseqüentes ao evento de recebimento também devem ser precedidos pelo evento de envio da mensagem.

A relação de precedência causal é não reflexiva, não existe sentido físico em afirmar que um evento precede a si mesmo. Dois eventos e e e', tais que $e \not\rightarrow e'$ e $e' \not\rightarrow e$, são chamados eventos concorrentes e são denotados por $e \,||\, e'$. Isto implica que não existe relação causal entre estes dois eventos e que não existe uma ordem implícita na qual estes eventos devam aparecer em uma ordenação consistente de todos os eventos do sistema.

Uma maneira de se conseguir ordenar os eventos de uma aplicação consiste em associar para cada evento um valor, que representa o instante de 'tempo' em que o evento aconteceu. Caso fosse possível ter um relógio físico que registrasse exatamente o mesmo tempo em todos os processos, então o tempo real poderia ser usado para ordenar os eventos, respeitando a relação de precedência causal. No entanto, é muito difícil manter relógios físicos sincronizados com o nível de precisão necessário para ordenar eventos que ocorrem milhares de vezes por segundo. Define-se então um relógio lógico como sendo uma função $R(e)$ que atribui um valor a um evento e, sem relação direta com o tempo real, mas que respeite a relação de precedência causal. Ou seja, a função deve respeitar a condição fraca de consistência de relógio:

$$e \rightarrow e' \Rightarrow R(e) < R(e').$$

Desta forma a função é consistente com a relação de precedência causal mas não a caracteriza. Caso se deseje caracterizar a relação de precedência então o relógio deve respeitar a condição forte de consistência de relógio:

$$e \rightarrow e' \Leftrightarrow R(e) < R(e').$$

Relógios lógicos podem ser construídos com apenas um contador ou com um vetor de contadores (este último com n contadores, um para cada processo da aplicação). O primeiro respeita a condição fraca de consistência enquanto que o segundo respeita a condição forte de consistência.

2.2.1 Relógio Lógico Escalar

Lamport[6] definiu um relógio lógico que usa um contador e que respeita a condição fraca de consistência de relógio. Os valores de relógio correspondem a inteiros não negativos e cada processo p_i mantém uma variável local r_i que mantém o valor de relógio corrente. Para

um dado evento e_i^k o valor de $R(e_i^k)$ corresponde ao valor da variável r_i no estado σ_i^k. A manutenção do relógio é feita de acordo com as seguintes regras:

1. Antes de executar um evento, p_i executa a seguinte operação:

$$r_i := r_i + 1$$

2. Cada mensagem enviada por p_i leva consigo o valor de r_i no momento de seu envio. Quando p_i recebe uma mensagem de p_j com o valor r_j associada a mesma, as seguintes operações são executadas antes de qualquer outra:

$$\begin{aligned} r_i &:= \max(r_i, r_j) \\ r_i &:= r_i + 1 \end{aligned}$$

Desta forma um evento tem associado a ele um relógio sempre maior que o relógio de eventos que o precederam, sem no entanto manter informações sobre quais são efetivamente estes eventos. Este comportamento respeita a condição fraca de consistência de relógio, o que não necessariamente caracteriza a precedência.

Um exemplo da evolução dos relógios em uma execução é mostrado na Figura 2.

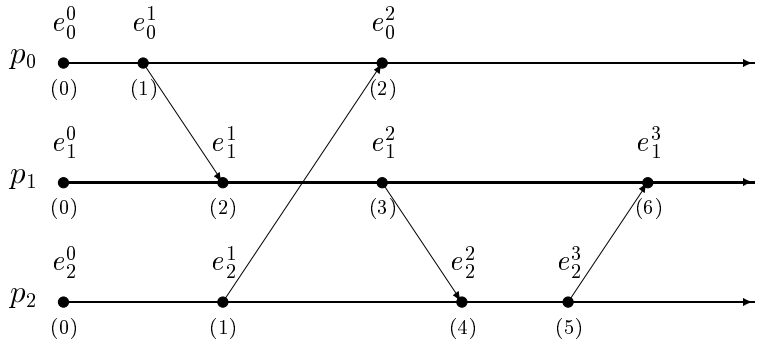

Figura 2: Relógios lógicos escalares.

2.2.2 Vetores de Relógios

Uma outra alternativa para manter um relógio lógico e ainda caracterizar a relação de precedência causal consiste em usar um vetor de relógios lógicos escalares [1, 7]. Desta forma é possível guardar a informação de todos os eventos que precedem causalmente o evento atual. Já que os eventos são executados seqüencialmente em cada processo, então basta guardar o relógio do último evento que precede causalmente o evento atual em cada um dos processos. O relógio lógico é então implementado como um vetor de n relógios escalares.

Cada processo p_i mantém um vetor v_i de n inteiros não negativos. Este vetor mantém o valor de relógio local corrente ($v_i[i]$) e o relógio global que este processo conhece. Para um dado evento e_i^k o valor de $R(e_i^k)[j]$ corresponde ao valor do relógio do último evento do processo p_j que precede causalmente o evento e_i^k. A manutenção do relógio é feita de acordo com as seguintes regras:

1. Antes de executar um evento, p_i executa a seguinte operação:

$$v_i[i] := v_i[i] + 1$$

2. Cada mensagem enviada por p_i leva consigo o vetor v_i no momento de seu envio. Quando p_i recebe uma mensagem de p_j com o vetor v_j associada a mesma, as seguintes operações são executadas antes de qualquer outra:

$$\begin{aligned} \forall k : 0 \leq k \leq n-1 : v_i[k] &:= \max(v_i[k], v_j[k]) \\ v_i[i] &:= v_i[i] + 1 \end{aligned}$$

Definindo-se a relação $<$ entre vetores de relógios da seguinte forma:

$$\begin{aligned} v_i < v_j \quad \Leftrightarrow \quad & \forall k : 0 \leq k \leq n-1 : v_i[k] \leq v_j[k] \text{ e} \\ & \exists k : 0 \leq k \leq n-1 : v_i[k] \neq v_j[k], \end{aligned}$$

observa-se que vetores de relógio respeitam a condição forte de consistência de relógio. Ainda mais, sabendo-se em que processos foram executados os eventos é possível determinar se um precede o outro com uma simples comparação de inteiros da seguinte forma:

$$\begin{aligned} e_i^k \rightarrow e_j^l \quad &\Leftrightarrow \quad R(e_i^k)[i] < R(e_j^l)[i] \\ e_i^k \,||\, e_j^l \quad &\Leftrightarrow \quad R(e_i^k)[i] > R(e_j^l)[i] \text{ e } R(e_i^k)[j] < R(e_j^l)[j]. \end{aligned}$$

Uma possível execução na qual os eventos estão associados a um vetor de relógios é mostrada na Figura 3.

2.3 Observando Estados Globais Consistentes

Supondo uma arquitetura reativa em que um monitor observa os eventos gerados pelos processos da aplicação, existem pares de eventos que não podem participar de um mesmo estado global consistente. Não faz sentido um estado global onde os estados de dois processos p_i e p_j, mostram que p_i recebeu uma mensagem de p_j, mas p_j não enviou esta mensagem. Deve-se estabelecer critérios para caracterizar quais estados formam um estado global consistente. Define-se então os conceitos de histórico, cortes, estados globais e consistência [1].

O histórico local de um processo p_i (h_i) corresponde ao conjunto de todos os eventos executados por este processo, isto é $h_i = \{e_i^0, e_i^1, \ldots\}$. Um prefixo do histórico local de um processo p_i (h_i^k) corresponde aos $k+1$ primeiros eventos executados pelo processo p_i, isto é $h_i^k = \{e_i^0, e_i^1, \ldots, e_i^k\}$. Um corte ($C$) é um subconjunto de todos os eventos executados pela

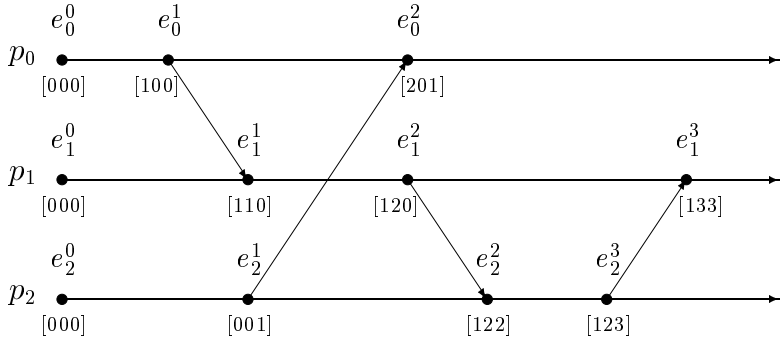

Figura 3: Vetores de relógios.

aplicação distribuída formado por prefixos dos históricos locais de cada um dos processos da aplicação. Um corte pode ser especificado por um conjunto de n números naturais $\{c_0, c_1, \ldots, c_{n-1}\}$, correspondendo aos índices dos prefixos de cada um dos processos, isto é $C = \{h_0^{c_0}, h_1^{c_1}, \ldots, h_{n-1}^{c_{n-1}}\}$. O conjunto dos últimos eventos de cada processo em um corte $\{e_0^{c_0}, e_1^{c_1}, \ldots, e_{n-1}^{c_{n-1}}\}$ corresponde à fronteira do corte.

Um estado global (Σ) é um conjunto dos estados locais de cada um dos processos, sendo determinado pelos estados dos processos na fronteira de um corte, logo $\Sigma = \{\sigma_0^{c_0}, \sigma_1^{c_1}, \ldots, \sigma_{n-1}^{c_{n-1}}\}$. Um corte é consistente se para quaisquer dois eventos e e e' pertencentes ao corte a seguinte relação é válida:

$$\text{Se } (e \in C) \text{ e } (e' \rightarrow e) \Rightarrow e' \in C.$$

Esta condição é coerente com a relação de precedência causal. Se um evento faz parte de um corte, então todos os eventos que o precederam causalmente também devem fazer parte do corte. A Figura 4 mostra dois cortes C e C' como partições do diagrama espaço-tempo. Os eventos que se situam antes da partição fazem parte do corte, sendo os últimos eventos antes da partição a fronteira do corte. Por exemplo, a fronteira do corte C é $\{e_0^2, e_1^2, e_2^2\}$. Graficamente o critério de consistência pode ser visto da seguinte forma: se as setas que representam as mensagens possuem sua origem do lado esquerdo do corte e seu fim do lado direito, então o corte é consistente, caso contrário o corte é inconsistente. Por exemplo, o corte C é consistente enquanto que o corte C' é inconsistente.

Um estado global é consistente se for gerado pela fronteira de um corte consistente. Desta forma um estado global consistente pode ser visto como uma possível observação do sistema que respeite a relação de precedência causal. A sucessão de estados globais consistentes tem paralelo com a passagem de tempo, sucessivos EGC representam 'instantes' de tempo de uma possível execução da aplicação distribuída. Considerando que os estados σ_i^k de um processo p_i estão associados aos eventos e_i^k que os geraram, define-se que dois estados σ_i^k e σ_j^l são inconsistentes entre si se e somente se:

$$(e_i^{k+1} \rightarrow e_j^l) \text{ ou } (e_j^{l+1} \rightarrow e_i^k).$$

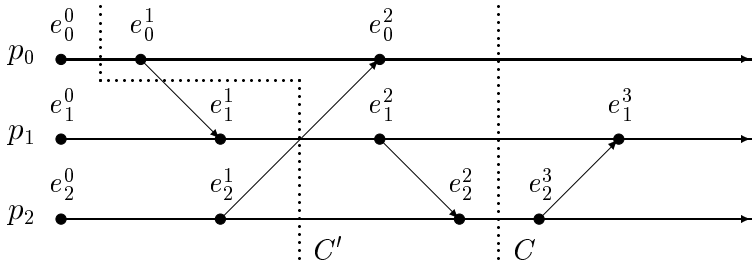

Figura 4: Exemplo de cortes.

Ou seja, podem existir relações de precedência entre dois eventos na fronteira do corte e ainda os estados gerados por estes eventos continuam consistentes entre si. Por exemplo, na Figura 4 os eventos e_1^2 e e_2^2 são consistentes. O critério de consistência do corte deve ser respeitado, tornando inconsistentes estados gerados por eventos que dependam causalmente de eventos ainda não incorporados ao corte. Como exemplo, observa-se que os eventos e_0^1 e e_1^1 são inconsistentes. Estendendo-se esta definição, pode-se afirmar que um estado global $\Sigma = \{\sigma_0^{c_0}, \sigma_1^{c_1}, \ldots, \sigma_{n-1}^{c_{n-1}}\}$ é consistente se e somente se:

$$\forall i, j : 0 \leq i, j \leq n-1 : e_i^{c_i+1} \not\rightarrow e_j^{c_j}.$$

É possível então decidir quais estados podem fazer parte de EGC se for possível extrair dos estados observados informações de precedência. Isto pode ser feito utilizando-se relógios lógicos, considerando-se que $R(\sigma_i^k) = R(e_i^k)$. Define-se então os critérios que podem ser usados para construir um EGC usando-se relógios lógicos escalares e vetores de relógios [4].

Relógios lógicos escalares não caracterizam a relação de precedência, mas é possível implicar que para um estado global $\Sigma = \{\sigma_0^{c_0}, \sigma_1^{c_1}, \ldots, \sigma_{n-1}^{c_{n-1}}\}$ a relação:

$$\forall i, j : 0 \leq i, j \leq n-1 : R(\sigma_i^{c_i+1}) \geq R(\sigma_j^{c_j})$$

é verdadeira se este estado global for consistente. Deve-se notar que a relação é verdadeira se o estado for consistente, mas um estado pode ser consistente mesmo caso esta relação seja falsa.

Vetores de relógios por sua vez caracterizam a relação de precedência causal, então um estado global $\Sigma = \{\sigma_0^{c_0}, \sigma_1^{c_1}, \ldots, \sigma_{n-1}^{c_{n-1}}\}$ é consistente se e somente se:

$$\forall i, j : 0 \leq i, j \leq n-1 : R(\sigma_i^{c_i})[i] \geq R(\sigma_j^{c_j})[i].$$

2.4 *Checkpoints* Globais Consistentes

Observar todos os eventos de uma computação para construir EGC pode ser muito dispendioso, portanto é interessante considerar apenas alguns estados de interesse, chamados *checkpoints* [4]. No entanto, construir um *checkpoint* global consistente a partir de *checkpoints* locais não é tão simples como construir um estado global consistente a partir dos estados dos processos [3].

Checkpoints são estados de um processo que formam uma abstração da execução do processo. *Checkpoints* são eventos internos do processo, isto é eventos que não estão associados ao envio ou recebimento de mensagens. Denota-se $\hat{\sigma}_i^k$ um *checkpoint* do processo p_i, que corresponde a um estado $\sigma_i^{k'}$, onde $k \leq k'$. Entre dois *checkpoints* locais de um processo vários eventos podem ocorrer, ou seja mensagens podem ser enviadas ou recebidas. Existe uma relação de precedência causal entre dois *checkpoints* $\hat{\sigma}_i^k \rightarrow \hat{\sigma}_j^l$, se para os eventos que geraram os estados $\sigma_i^{k'}$ e $\sigma_j^{l'}$ a relação $e_i^{k'} \rightarrow e_j^{l'}$ for verdadeira.

Desta forma dois *checkpoints* são consistentes entre si se forem concorrentes, pois a existência de uma relação de precedência entre dois *checkpoints* $\hat{\sigma}_i^k \rightarrow \hat{\sigma}_j^l$ implica na existência de um evento e_i^α posterior a $\hat{\sigma}_i^k$ que precede causalmente um evento e_i^β anterior a $\hat{\sigma}_j^l$. Percebe-se que um corte contendo os eventos que geraram os estados $\hat{\sigma}_i^k$ e $\hat{\sigma}_j^l$ em sua fronteira não é consistente. A Figura 5 mostra um padrão de *checkpoints* onde pode-se observar que $\hat{\sigma}_0^1$ e $\hat{\sigma}_1^1$ são consistentes e $\hat{\sigma}_0^0$ e $\hat{\sigma}_2^1$ são inconsistentes.

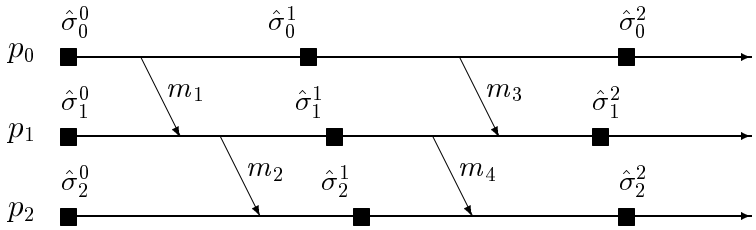

Figura 5: *Checkpoints* em uma execução.

Um *checkpoint* global ($\hat{\Sigma}$) consiste em um conjunto de n *checkpoints*, um para cada processo, especificado por um conjunto de inteiros $\{c_0, c_1, \ldots, c_{n-1}\}$, isto é $\hat{\Sigma} = \{\hat{\sigma}_0^{c_0}, \hat{\sigma}_1^{c_1}, \ldots, \hat{\sigma}_{n-1}^{c_{n-1}}\}$. Um *checkpoint* global é consistente se e somente se:

$$\forall i, j : 0 \leq i, j \leq n-1 : \hat{\sigma}_i^{c_i} \not\rightarrow \hat{\sigma}_j^{c_j}.$$

Porém, não basta que dois *checkpoints* sejam consistentes entre si para que eles possam participar de um *checkpoint* global consistente. Por exemplo, na Figura 5 dois *checkpoints* concorrentes $\hat{\sigma}_0^1$ e $\hat{\sigma}_2^2$ não podem participar de nenhum *checkpoint* global consistente, porque $\hat{\sigma}_0^1 \rightarrow \hat{\sigma}_1^2$ e $\hat{\sigma}_1^1 \rightarrow \hat{\sigma}_2^2$. Ou seja, existe uma relação de precedência causal entre estes *checkpoints* e todos os *checkpoints* do processo p_1.

Define-se a relação de Z-precedência ($\rightsquigarrow$) entre *checkpoints* da seguinte forma [5]:

$$\hat{\sigma}_i^k \rightsquigarrow \hat{\sigma}_j^l \Leftrightarrow \begin{cases} \hat{\sigma}_i^k \rightarrow \hat{\sigma}_j^l, \text{ ou} \\ \exists \hat{\sigma}_\alpha^\beta : (\hat{\sigma}_i^k \rightsquigarrow \hat{\sigma}_\alpha^\beta) \text{ e } (\hat{\sigma}_\alpha^{\beta-1} \rightsquigarrow \hat{\sigma}_j^l). \end{cases}$$

A condição necessária e suficiente para que um conjunto de *checkpoints* possa ser estendido para um *checkpoint* global consistente implica na não existência de Z-precedência entre os *checkpoints* do conjunto.

Caso o conjunto possua apenas um *checkpoint* $\hat{\sigma}$ como elemento e não possa ser estendido para formar um *checkpoint* global consistente, isto é $\hat{\sigma} \rightsquigarrow \hat{\sigma}$, então $\hat{\sigma}$ é chamado de *checkpoint* inútil e não pode fazer parte de nenhum *checkpoint* global consistente. É dito também que o *checkpoint* $\hat{\sigma}$ está envolvido em um ciclo-Z.

Devido a possibilidade de existirem *checkpoints* inúteis, o processo de seleção dos estados que devem ser escolhidos como *checkpoints* é uma parte importante de processo de obtenção de *checkpoints* globais consistentes. Existem três abordagens para resolver este problema: síncrona, assíncrona e quase-síncrona.

A abordagem síncrona suspende a operação da aplicação enquanto os processos se sincronizam, garantindo que o conjunto de *checkpoints* seja consistente. A abordagem assíncrona não impõe restrições sobre quais estados são escolhidos como *checkpoints*, possivelmente gerando *checkpoints* inúteis, que devem ser detectados e descartados pelo algoritmo que constrói *checkpoints* globais consistentes. A abordagem quase-síncrona também não restringe quais estados devem ser *checkpoints*, mas força os processos a tirarem *checkpoints* seguindo um protocolo, de modo a evitar ou minimizar a ocorrência de *checkpoints* inúteis

Os protocolos quase-síncronos são classificados de acordo com o padrão de *checkpoints* que eles geram [4]. Caso não existam relações de Z-precedência entre os *checkpoints* sem que exista uma relação de precedência causal, o protocolo é chamado ZPF. Caso não existam ciclos-Z, o protocolo é chamado ZCF. O último tipo de protocolo tenta minimizar a ocorrência de ciclos-Z mas não os evita, são chamados PZCF.

3 Arquitetura de Software do Ambiente de Programação

Nosso objetivo principal ao implementar algoritmos para a construção de estados globais consistentes foi por em prática os vários mecanismos estudados, aprofundando o entendimento dos mesmos além de abrir a possibilidade de se testar algumas otimizações. Para tanto foi necessário construir um substrato mínimo onde fosse possível construir implementações modulares e com grande nível de reaproveitamento de código. Decidimos então construir um suporte de comunicação bem simples, mas que permitisse que os algoritmos e mecanismos de controle (relógios lógicos, etc.) fossem implementados independentemente e depois combinados formando uma aplicação distribuída. Nos concentramos em algoritmos quase-síncronos, mas tendo a vista a implementação de algoritmos baseados em outras abordagens do problema. Utilizamos a linguagem Java™ [2, 8] pois já possuíamos implementações em alto nível dos algoritmos estudados nesta linguagem [4, 5]. Além disso, Java incentiva a modularização, facilita a construção de interfaces de alto nível e possui uma grande portabilidade.

Adotamos uma política de separação de tarefas, com dois componentes distintos: o sistema a ser observado, composto por vários processos, e o monitor que fará as observações. A Figura 6 mostra a organização do sistema. O ambiente de programação foi construído como um conjunto de classes e interfaces a partir dos quais um sistema completo (processos mais monitor) pudesse ser construído. É interessante que o programador possa construir estes dois componentes de forma independente, sendo este conjunto de classes a peça de ligação entre os mesmos. Fica possível então observar o comportamento de vários algoritmos

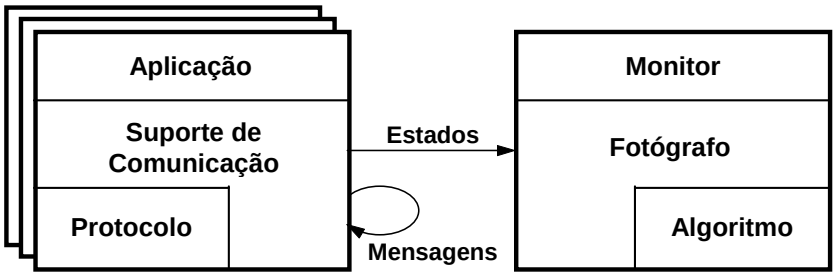

Figura 6: Arquitetura de monitorização.

ao monitorar a mesma aplicação ou de um algoritmo monitorando aplicações com diferentes padrões de troca de mensagens, simplesmente recombinando os componentes.

O objetivo central do ambiente de programação é permitir que algoritmos possam ser implementados rapidamente, sem a necessidade do programador se preocupar com todos os detalhes de implementação. Especificamente, é desejável que aspectos relativos ao subsistema de comunicação estejam já implementados e possam ser acessados por meio de uma interface simples, como por exemplo a localização dos processos e do monitor, o mecanismo de troca de mensagens entre os componentes da aplicação, a propagação de informação de relógio, o envio de *checkpoints* ao monitor, entre outros. Decidimos implementar a comunicação a partir de primitivas básicas de UDP/IP, pois queríamos ter um grande controle sobre como as mensagens seria processadas. A ênfase foi na personalização e facilidade de uso, deixando um pouco de lado a eficiência.

Descrevemos a seguir os dois componentes básicos do sistema.

3.1 A Aplicação Distribuída

A computação a ser monitorizada é uma parte muito importante de qualquer implementação de algoritmos para construção de EGC. Sem uma aplicação que tenha um padrão de troca de mensagens e estados de processos interessantes, fica difícil avaliar se o algoritmo está realmente funcionando como esperado ou perceber vantagens e desvantagens de vários algoritmos. Ao projetarmos o ambiente de programação que dá suporte a construção da aplicação distribuída, o objetivo não foi construir um substrato de comunicação genérico, mas sim algo simples e flexível que pudesse acomodar várias aplicações com comportamentos diversos que demonstrem o funcionamento dos algoritmos sendo avaliados.

Construímos então uma abstração da rede de comunicação com apenas três primitivas:

- `sendMessage`
- `receiveMessage`
- `sendCheckpoint`

As duas primeiras servem para troca de mensagens entre os processos da aplicação e a última serve para mandar um estado ao monitor. As primitivas de troca de mensagens possuem semântica semelhante a das primitivas `send` e `receive` de UDP/IP, permitindo troca de mensagens sem garantia de entrega. A mensagem é um vetor de *bytes*, ficando a aplicação responsável por linearizar os dados a serem transmitidos. Chegamos a conclusão que implementar um mecanismo genérico de linearização seria muito dispendioso e desnecessário já que as aplicações deveriam trocar dados muito simples.

Por outro lado, implementar algum mecanismo para prover transparência de localização aos processos mostrou-se necessário. Todos os algoritmos estudados supõem que processos possuem um identificador inteiro que é usado entre outras coisas como o índice do processo em um vetor de relógios. Se o ambiente já provesse esta transparência, o programador poderia construir suas implementações em um nível de abstração bem mais próximo daquele empregado para a exposição dos algoritmos. Desta forma, os processos da aplicação possuem inteiros como identificação, sendo estes usados como parâmetros de origem e destino das primitivas `sendMessage` e `receiveMessage`. O mapeamento entre inteiros e ênuplas (host:porta) é feito transparentemente pela abstração de rede de comunicação, por meio de um arquivo de configuração. A aplicação deve apenas informar à rede de comunicação qual é o seu identificador e qual arquivo de configuração contém os mapeamentos.

Independente da aplicação, o substrato de comunicação deve prover uma maneira de manter e disseminar informação de relógio entre os processos. Com este objetivo a abstração de rede de comunicação trabalha em conjunto com um protocolo que é responsável por manter informação de relógio e, no caso de protocolos quase-síncronos, forçar a aplicação a mandar o seu estado para o monitor. O protocolo é notificado a cada mensagem que é enviada ou recebida pelo processo, devendo gerar um *timestamp* para cada mensagem que sai e observando o *timestamp* das mensagens que chegam, antes da entrega à aplicação. O protocolo também pode executar outras ações antes de fornecer/processar um *timestamp*, isto facilita a implementação de algoritmos quase-síncronos e também permite outros comportamentos, como por exemplo uma aplicação que manda seu estado ao monitor a cada evento.

Os processos devem cooperar com o monitor enviando seu estado para o mesmo, expontaneamente ou seguindo um protocolo. A aplicação então deve prover uma primitiva `takeCheckpoint` que coleta o estado relevante à aplicação e envia o mesmo ao monitor. Esta primitiva pode ser usada pela própria aplicação ou pelo protocolo associado a rede de comunicação. A aplicação fica responsável por escolher qual protocolo deve ser usado e iniciá-lo corretamente. Para enviar o estado ao monitor a primitiva `sendCheckpoint` deve ser usada, o protocolo é notificado que um estado está sendo enviado ao monitor, permitindo que sejam tomadas ações possivelmente diferentes das ações executadas quando do envio de mensagens comuns entre processos. Por exemplo, em protocolos de *checkpoint* o relógio lógico só deve ser avançado quando um *checkpoint* é enviado ao monitor. A rede de comunicação garante que entre os processos e o monitor o canal de comunicação é FIFO, e que os estados dos processos serão entregues com esta garantia de ordenação. A primitiva `takeCheckpoint` não precisa necessariamente mandar todo o estado relevante ao monitor, basta que a aplicação guarde localmente este estado e envie somente um identificador do mesmo. O monitor nunca toma decisões baseado no estado em si, somente em seu *timestamp*.

3.2 O Monitor

O monitor é responsável por receber os estados dos processos, construir um estado global consistente, realizar alguma computação sobre este estado global e possivelmente agir sobre a aplicação. Separamos as duas primeiras tarefas em um módulo separado chamado fotógrafo, que deve prover ao monitor apenas uma sucessão de estados globais consistentes. É este módulo que o ambiente provê ao programador, permitindo a partir dele construir o monitor completo.

O fotógrafo está organizado como um servidor de estados. Ao se construir um novo fotógrafo o mesmo extrai do arquivo de configuração da aplicação o endereço em que deve ouvir e passa a receber os estados enviados pelos processos. O fotógrafo possui a primitiva `getGlobalCheckpoint` que retorna um estado global consistente. A primitiva `getGlobalCheckpoint` possui vários comportamentos, ela pode retornar sucessivamente cada um dos estados globais construídos ou apenas o mais recente, ignorando os estados intermediários, ou então ela pode bloquear caso não haja um estado global novo ou apenas retornar o mesmo estado, sem bloquear. O monitor escolhe qual tipo de fotógrafo vai utilizar de acordo com que visão da aplicação que ele quer ter. Por exemplo, um monitor interessado em detectar perda de *token* não precisa observar todos os estados globais consistentes, apenas o mais recente. Por outro lado um monitor que está depurando uma aplicação distribuída pode querer ver todos os estados globais que foram construídos e deve ficar bloqueado enquanto um novo estado não estiver disponível.

O fotógrafo constrói um estado global consistente a partir dos estados dos processos que recebeu usando um algoritmo que lhe é passado em sua inicialização. Em certas ocasiões, respeitando a sua semântica, o fotógrafo pede ao algoritmo para construir um novo estado a partir do estado global atual e dos estados que foram recebidos e ainda não foram incorporados a nenhum estado global consistente. Este algoritmo é selecionado pelo monitor ao construir o fotografo e deve ser compatível com o protocolo usado pela rede de comunicação para colocar *timestamps* nas mensagens.

4 Descrição do Ambiente de Programação

Nesta seção descreveremos os aspectos mais importantes das classes e interfaces que compõem o ambiente de programação e como elas se relacionam. A especificação completa do ambiente de programação pode ser encontrada em:

http://www.dcc.unicamp.br/~gdvieira/mc030/packages.html

4.1 Classe `Network`

A classe `Network` implementa a abstração de rede de comunicação descrita na Seção 3.1. Cada um dos processos da aplicação deve instanciar um objeto desta classe e usá-lo para se comunicar com os demais componentes da aplicação.

O construtor desta classe possui a seguinte assinatura:

- `Network(int pid, String configFile, CheckpointProtocol ckptProt)`

O processo deve informar qual o seu identificador, qual arquivo de configuração possui os mapeamentos identificador–(host:porta) e qual protocolo vai cuidar da manutenção de *timestamps* nas mensagens. A construção do protocolo é descrita na Seção 4.2

Os métodos que possibilitam enviar e receber mensagens tem as seguintes assinaturas:

- `void sendMessage(int dest, byte clientMessage[])`
- `int receiveMessage(byte clientMessage[])`
- `void sendCheckpoint(Serializable processCheckpoint)`

O método `sendMessage` recebe como parâmetros o identificador do destino e uma referência ao vetor que contém a mensagem, retornando imediatamente. Analogamente, `receiveMessage` preenche um vetor com uma mensagem recebida e retorna o identificador da origem, bloqueando enquanto não for recebida alguma mensagem. O método `sendCheckpoint` possui uma semântica diferente. Ele estabelece uma conexão com o monitor e envia o estado do processo passado como parâmetro, só retornando uma vez que o monitor recebeu corretamente este estado. Utilizamos os mecanismos de linearização automática da linguagem Java, logo o estado do processo pode ser qualquer objeto serializável Java.

Esta classe é o ponto de contato entre os processos da aplicação e o monitor. Os processos só devem se comunicar entre si e com o monitor usando os métodos desta classe, que fornece a abstração da localização destes componentes. O formato do arquivo de configuração que efetivamente conhece a localização dos processos e monitor esta descrito na documentação da classe `NodeConfiguration`.

4.2 Classe `CheckpointProtocol` e Interface `Checkpointable`

A classe abstrata `CheckpointProtocol` caracteriza um protocolo que cuida da informação de relógio e funciona em conjunto com uma instância da classe `Network`. Do ponto de vista do programador da aplicação, deve ser selecionado o protocolo mais apropriado, dentre as subclasses concretas desta classe, e uma instância do mesmo deve ser passada a classe `Network`. Do ponto de vista do programador dos protocolos, esta classe deve ser derivada, implementando o comportamento dos diferentes protocolos. Um exemplo de implementação para protocolos quase-síncronos está na Seção 5.1. A descrição completa dos métodos que devem ser (re)definidos em uma implementação concreta desta classes está na documentação da mesma.

O construtor da classe `CheckpointProtocol` tem a seguinte assinatura:

- `CheckpointProtocol(Checkpointable process)`

Este método recebe uma referência a um objeto que implemente a interface `Checkpointable`. Este objeto deve ser a classe que é responsável por coletar o estado da aplicação e mandá-lo ao monitor. Por exemplo, se o objeto que possui a referência a instância da classe `Network` implementar a interface `Checkpointable`, então a construção de novos objetos `Network` e `CheckpointProtocol` pode ser:

```
Network net= new Network(1, "file", new AnyCheckpointProtocol(this));
```

A interface `Checkpointable` possui apenas um método:

- `void takeCheckpoint()`

que deve ser implementado por alguma das classes do processo de modo a cooperar com o protocolo de construção de EGC, como descrito na Seção 3.1.

4.3 Classe `Photographer`

A classe `Photographer` implementa um fotógrafo genérico de aplicações distribuídas. Esta é uma classe abstrata que fornece os serviços básicos de um servidor de estados. Classes derivadas concretas fornecem implementações de semânticas específicas, como descrito na Seção 3.2. Para o momento, o ambiente de programação possui dois fotógrafos prontos para uso: `AllStatesPhotographer` e `AllStatesSyncPhotographer`. Ambos fornecem ao monitor todos os estados globais consistentes construídos, sendo que o primeiro não bloqueia a espera de um novo estado, enquanto o segundo bloqueia.

Os construtores de todos os fotógrafos possuem a seguinte assinatura:

- `Photographer(String configFile, PhotographerAlgorithm pa)`

O monitor deve passar ao fotógrafo o arquivo de configuração, de onde este extrai informações sobre em qual porta ele deve esperar conexões e receber estados de processos. Deve ser passado também o algoritmo que será usado para construir estados globais consistentes. A classe `PhotographerAlgorithm` é descrita na Seção 4.4.

Quando o monitor deseja observar um EGC, ele deve chamar o seguinte método:

- `Checkpoint[] getGlobalCheckpoint()`

Este método retorna um vetor de estados locais, correspondendo a um estado global consistente. Qual estado é retornado depende do tipo de fotógrafo e a consistência é garantida pelo algoritmo associado ao fotógrafo. A descrição da classe `Checkpoint` pode ser encontrada na documentação do ambiente.

4.4 Interface `PhotographerAlgorithm`

A interface `PhotographerAlgortihm` deve ser implementada por uma classe que queira implementar um algoritmo do fotógrafo. Esta interface possui apenas um método:

- `void newConsistentGlobalState(StateVector state)`

Este método deve construir um novo EGC a partir do EGC atual e dos estados recebidos pelo fotógrafo. A classe `StateVector` implementa uma estrutura de dados para guardar EGC e estados ainda não incorporados ao EGC. Esta classe possui métodos para manipular estes estados e está descrita em detalhes na documentação do ambiente.

O programador que estiver construindo o monitor deve instanciar a classe `Photographer` como nesse exemplo:

```
Photographer view=
    new AllStatesPhotographer("file", new AnyPhotographer());
```

onde `AnyPhotographer` é uma classe que implementa a interface `PhotographerAlgortihm`.

Um exemplo de implementação desta interface pode ser visto na Seção 5.2.

4.5 Uma aplicação exemplo

Como ilustração de uma pequena aplicação que pode ser construída sobre o ambiente de programação, temos os Programas 4.1 e 4.2.

O Programa 4.1 é um processo de uma aplicação que troca um *token* entre seus componentes, não executando outra coisa. O padrão de mensagens e estados deste processo não é muito interessante, mas ele serve como um exemplo da simplicidade da interface da classe `Network`. O Programa 4.2 é um monitor que pode observar o Programa 4.1 em execução.

Na confecção destes pequenos programas não nos preocupamos em fazem um tratamento de erros e exceções adequado em nome da brevidade. Versões mais extensas destes dois programas foram implementadas, servindo como nossa base de testes dos algoritmos.

5 Implementação de Protocolos Quase-Síncronos

Nesta seção apresentamos os algoritmos para obtenção de EGC que foram implementados e como o ambiente de programação introduzido nas seções anteriores foi utilizado na implementação. Implementamos os protocolos quase-síncronos e os algoritmos para construção progressiva de *checkpoints* globais consistentes apresentados em [4].

5.1 Relógios e Protocolos Quase-Síncronos

Um protocolo quase-síncrono observa a troca de mensagens entre os processos da aplicação, trocando informação de controle (relógios) e forçando um padrão de *checkpoints* sobre a aplicação [4]. Todos os protocolos quase-síncronos apresentam duas características em comum: eles mantêm um relógio e forçam a aplicação a tirar *checkpoints* em certas situações. Para cada mensagem que chega é avaliado um predicado sobre o relógio da mensagem recebida e o relógio da aplicação, caso este predicado seja verdadeiro então a aplicação deve tirar um *checkpoint* forçado.

Definimos então uma classe `QSCkptProtocol`, derivada de `CheckpointProtocol`, que implementa um protocolo quase-síncrono abstraindo o relógio a ser usado. Uma interface (`QSClock`) foi definida e possui métodos para manter e propagar informações de relógio além do método `mustTakeForcedCkpt(QSClock)`, que retorna `true` caso o processo deva tirar um *checkpoint* forçado. A classe `QSCkptProtocol` mantém uma referência para um objeto que implementa esta interface, usando este objeto como relógio.

A classe `QSCkptProtocol` implementa os métodos abstratos de `CheckpointProtocol` da seguinte maneira. Os métodos que tratam do envio e manutenção de informações de controle associadas a mensagens são encaminhadas diretamente ao objeto que implementa o relógio. O método que trata o recebimento de uma mensagem primeiro verifica se um *checkpoint* deve

Programa 4.1 Exemplo de um processo da aplicação

```
import java.io.*;
import java.net.*;
import java.util.Random;
import chap.*;

public class TokenProcess implements Runnable, Checkpointable
{
  private int pid;
  private boolean hasToken;
  private Random bag;
  private Network net;

  public TokenProcess(int pid, String configFile)
  {
    this.pid= pid; bag= new Random(666);
    if (pid == 0) hasToken= true; else hasToken= false;
    try { net= new Network(pid, configFile, new VC_FDI_Protocol(this));
    } catch(Exception e) {}
  }
  public void run()
  {
    byte[] data= new byte[1]; int rand;
    data[0]= 23; takeCheckpoint();
    while (true) {
      if (hasToken) {
       rand= (int)Math.round(bag.nextDouble() * 2000);
       try{Thread.sleep(rand);}catch(InterruptedException e){}
       rand= (int)Math.round(bag.nextDouble() * (net.getNPeers()-1));
       try { net.sendMessage(rand, data); } catch(IOException e) {}
       hasToken= false;
      } else {
       try{ net.receiveMessage(data); } catch(IOException e) {}
       hasToken= true;
      }
    }
  }
  public void takeCheckpoint()
  {
    try { net.sendCheckpoint(new Boolean(hasToken));
    } catch(IOException e) {}
  }
  public static void main(String[] args)
  {
    try{ new TokenProcess(Integer.parseInt(args[0]), args[1]).run();
    } catch(NumberFormatException e) {}
  }
}
```

Programa 4.2 Exemplo de monitor

```
import java.net.*;
import java.io.*;
import chap.*;

public class Monitor
{
  public static void main(String[] args)
  {
    try {
      Photographer foto=
        new AllStatesSyncPhotographer("TokenConfig", new VC_ZPFPhotoAlgorithm());
      while (true) {
        Checkpoint[] state= foto.getGlobalCheckpoint();
        for (int i=0; i<state.length; i++)
          System.out.println(state[i]);
        System.out.println();
      }
    } catch(Exception e) {e.printStackTrace();}
  }
}
```

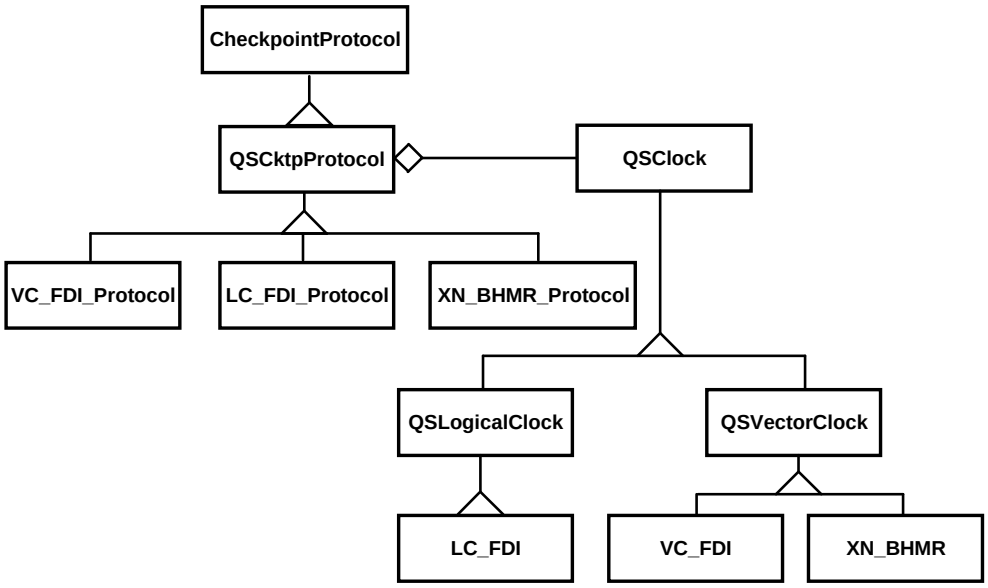

Figura 7: Hierarquia de classes para protocolos quase-síncronos.

ser tirado, chamando `mustTakeForcedCkpt`, e depois chama o método correspondente no relógio. Desta forma, para se definir um novo protocolo quase-síncrono, basta construir um relógio que respeite os critérios do protocolo e derivar a classe `QSCkptProtocol`, instruindo-a a usar este relógio.

Existem dois tipos de relógios lógicos usados pelos protocolos implementados: relógios lógicos escalares e vetores de relógios. Construímos a classe `QSLogicalClock` e a classe `QSVectorClock`, que implementam parte da interface `QSClock`. O método `mustTakeForcedCkpt` não foi definido e é a sua implementação que caracteriza um protocolo específico. A Figura 7 mostra a hierarquia das classes que compõem protocolos quase-síncronos.

Com estas classes já definidas a implementação dos protocolos consistiu em escolher o tipo de relógio a ser usado e implementar o método `mustTakeForcedCkpt`. Implementamos então um relógio para protocolos de cada uma das classificações: ZPF, ZCF, PZCF.

O protocolo VC_FDI propaga vetores de relógios e é ZPF. Este protocolo induz *checkpoints* sempre que o vetor de relógios de uma mensagem recebida possuir algum valor maior que o valor correspondente no vetor de relógios do processo. Este protocolo força um número não limitado de *checkpoints* forçados para cada *checkpoint* espontâneo.

O protocolo LC_FDI propaga relógios lógicos escalares e produz um padrão de *checkpoints* ZCF. Este protocolo força o processo a tirar um *checkpoint* quando é recebida uma mensagem com um valor de relógio maior que o valor de relógio do processo. Este protocolo induz um número menor de *checkpoints* do que o protocolo VC_FDI, gerando no máximo $n-1$ *checkpoints* forçados para cada *checkpoint* espontâneo.

O protocolo XN_BHMR propaga vetores de relógios e é PZCF. Este protocolo tenta detectar um tipo especial de ciclo-Z. Um *checkpoint* forçado é induzido sempre que o processo recebe um relógio que contenha, para a posição correspondente a este processo, o mesmo valor que o seu relógio.

Implementamos três classes `VC_FDI_Protocol`, `LC_FDI_Protocol` e `XN_BHMR_Protocol`, incorporando estas implementações de relógios. Estas classes podem ser passadas diretamente a rede de comunicação, colocando o protocolo em funcionamento.

5.2 Visão Progressiva

Gerar um padrão de *checkpoints* utilizando um algoritmo quase-síncrono é apenas uma parte do problema de se determinar *checkpoints* globais consistentes. O algoritmo que rege o funcionamento do fotógrafo também deve ser construído, devendo ser compatível com o padrão de *checkpoints* gerado pelo algoritmo quase-síncrono.

A implementação dos algoritmos do fotógrafo consistiu em construir classes que implementem a interface `PhotographerAlgorithm`. Implementamos três algoritmos, um para cada protocolo quase-síncrono implementado.

O algoritmo VC_ZPFPhotoAlgorithm constrói *checkpoints* globais consistentes a partir de um padrão de *checkpoints* ZPF e utiliza vetores de relógios. A cada passo deste algoritmo, são incorporados ao *checkpoint* global consistente todos os *checkpoints* que não possuam relação de precedência com outros estados ainda não incorporados.

O algoritmo LCPhotographerAlgorithm constrói *checkpoints* globais consistentes a partir de um padrão de *checkpoints* ZCF e utiliza relógios lógicos escalares. Este algoritmo incorpora no *checkpoint* global consistente todos os *checkpoints* que tenham o valor mínimo de relógio.

O algoritmo VC_PZCFPhotoAlgorithm constrói *checkpoints* globais consistentes a partir de um padrão de *checkpoints* PZCF e utiliza vetores de relógios. Este protocolo progride a cada passo ou incorporando *checkpoints* úteis ou descartando *checkpoints* inúteis.

6 Conclusão

O ambiente de programação desenvolvido mostrou-se bastante adequado à implementação de algoritmos para construção de EGC. Implementamos sem grandes dificuldades alguns protocolos quase-síncronos e conseguimos um conjunto de classes limpo e que possibilitou grande reutilização de código. Implementamos também uma aplicação de teste, composta por processos que trocam um *token*, verificando que a interface da abstração de comunicação é poderosa o suficiente servir como base para a experimentação com estes algoritmos.

Algumas melhorias poderiam ser incorporadas ao estágio atual do ambiente de programação. Um mecanismo de linearização de mensagens se mostrou necessário, facilitando a construção da aplicação e principalmente simplificando o processo de propagação de relógios. Uma arquitetura que levasse em conta grupo de processos, eliminaria a necessidade de se manter um arquivo de configuração e possibilitaria a construção de monitores mais flexíveis, abrindo a possibilidade de um monitor replicado, aumentando a confiabilidade do sistema de monitorização.

Referências

[1] Ozalp Babaoglu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. Technical Report UBLCS-93-1, University of Bologna, January 1993.

[2] Mary Campione and Kathy Walrath. *The Java Tutorial: Object-Oriented Programming for the Internet*. Addison-Wesley, second edition, March 1998.

[3] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, October 1996.

[4] Islene Calciolari Garcia. Estados globais consistentes em sistemas distribuídos. Master's thesis, Instituto de Computação–UNICAMP, July 1998.

[5] Islene Calciolari Garcia and Luiz Eduardo Buzato. Progressive construction of consistent global checkpoints. Technical Report IC-98-36, Instituto de Computação–UNICAMP, October 1998.

[6] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[7] Michel Raynal and Mukesh Singhal. Logical time: Capturing causality in distributed systems. *IEEE Computer*, pages 49–56, February 1996.

[8] Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA. *Java API Documentation*, 1997. Version 1.1.