

Capítulo 5

Eleição de Líder

Elias P. Duarte Jr.

Versão do dia 19 de março de 2026

Um número muito significativo de sistemas e aplicações distribuídas fazem uso de um *líder*. O líder é um dos processos do sistema que tem alguma função especial, em geral relacionada à coordenação dos processos, definindo ações ou papéis e auxiliando na tomada de decisões. O líder pode ainda ser uma referência para os demais processos, mantendo informações centralizadas com o objetivo de garantir a consistência, confiabilidade ou o desempenho do sistema.

Se os processos do sistema distribuído estão sujeitos a falhas, o mesmo pode acontecer com o líder. Assim, é normal que sistemas que precisem de um mecanismo dinâmico para a eleição de líder. No caso de falha do líder atual, ele deve ser substituído por outro líder. Na verdade, todo algoritmo para eleição de líder deve ser acompanhado de um detector de falhas. Na sua forma mais trivial podemos simplesmente usar um detector para identificar processos falhos e elegemos como líder o processo correto que atende a um determinado critério, por exemplo aquele com menor identificador.

Neste capítulo vamos primeiro estudar o detector de falhas $\Diamond P$ (Diamante P) para falhas *crash* no modelo GST, para em seguida implementar a eleição de líder usando aquele detector. Em seguida vamos definir um algoritmo para eleição de líder no modelo *crash-recovery*, reforçando as premissas deste modelo e o que ele permite, tendo em vista que sua característica mais importante é que os processos têm memória secundária.

5.1 O Detector de Falhas $\Diamond P$

Entre as implementações existentes para as diversas classes de detectores de falhas vistas no capítulo anterior, uma das mais importantes é a que vamos estudar agora para o detector $\Diamond P$, dito "perfeito após um tempo" (*eventually perfect*). Informalmente, as duas propriedades que o detector $\Diamond P$ precisa garantir são:

- *Completeness Forte*: após um intervalo de tempo finito mas desconhecido, todo processo falho é efetivamente suspeitado de ter falhado por todos os processos corretos.
- *Precisão Forte Após um Tempo*: após um intervalo de tempo finito mas desconhecido, nenhum processo correto é suspeitado por nenhum processo correto.

Veja que a própria definição deste detector de falhas é muito adequada para o modelo parcialmente síncrono GST. Vamos lembrar daquele modelo: o sistema inicia sem nenhuma garantia temporal (assíncrono) mas se torna síncrono após um intervalo de tempo finito mas desconhecido, obedecendo, a partir de então, limites de tempo para a execução de tarefas e transmissão de mensagens. O detector de falhas $\Diamond P$ vai cometer enganos enquanto o sistema estiver instável: processos corretos serão confundidos como falhos. Porém precisamos garantir que estes enganos sejam corrigidos quando o sistema se tornar síncrono.

O Algoritmo 5.1 mostra uma implementação clássica do detector de falhas $\Diamond P$. Veja que quando o detector identifica um evento. Neste caso um evento pode ser a falha de um processo ou o recebimento de um heartbeat de um processo suspeitado. O motivo dessas notificações explícitas é que a seguir vamos usar esse detector para implementar um algoritmo de eleição de líder.

Algoritmo Detector de Falhas $\Diamond P$ executado pelo processo i ;
Init: Corretos $\leftarrow \pi$;
Suspeitos $\leftarrow \emptyset$;
IntervaloMonit $\leftarrow$ estimativa de *timeout* inicial;
Repita periodicamente no IntervaloMonit:
se Corretos $\cap$ Suspeitos $\neq \emptyset$ // Cometeu engano!
então Timeout $\leftarrow$ Timeout + Δ ;
para todo $p \in \pi$ faça
se ($p \notin$ Corretos) E ($p \notin$ Suspeitos)
então Suspeitos $\leftarrow$ Suspeitos $\cup \{p\}$;
Notifica: p suspeito!
senão se ($p \in$ Corretos) E ($p \in$ Suspeitos)
então Suspeitos $\leftarrow$ Suspeitos - $\{p\}$;
Notifica: p correto!
send(p , Requisteste);
fim-para;
Corretos $\leftarrow \emptyset$;
Upon receive(q , Requisteste): send(q , RespostaTeste);
Upon receive(q , RespostaTeste): Corretos $\leftarrow$ Corretos $\cup \{q\}$;
Upon Detector de Falhas chamado: return Suspeitos;
Fim Algoritmo.
Algoritmo 5.1: Implementação clássica do detector $\Diamond P$.

O Algoritmo 5.1 descreve o detector $\Diamond P$ com uma implementação do tipo *pull*, baseada em testes. O teste é realizado da seguinte maneira: periodicamente, a cada intervalo de monitoramento,

o testador envia uma requisição de teste *RequisTeste* para cada um dos demais processos, esperando receber de cada um uma resposta ao teste *RespostaTeste*, antes de estourar o intervalo de monitoramento. O intervalo de monitoramento funciona como um intervalo de *timeout* neste caso. Se em um intervalo não chegou a *RespostaTeste* de um determinado processo, então o processo que não respondeu é considerado falho. O conjunto de processos Corretos é esvaziado a cada intervalo, quando é enviada a *RequisTeste*. Na medida em que chega a resposta de cada processo correto, ele é incluído no conjunto Corretos. Quando um processo está tanto no conjunto de Corretos quanto no conjunto de Suspeitos isso significa que houve um engano: ele foi suspeitado e posteriormente foi recebida sua resposta ao teste. Como o modelo de falhas é o *crash* (i.e., sem recuperação) só pode ter havido uma falsa suspeita. Neste caso, o processo é retirado do conjunto Suspeitos e o intervalo de monitoramento é incrementado de Δ unidades. O objetivo é fazer com que o intervalo fique grande o suficiente para acomodar os tempos de resposta de todos os processos monitorados.

Vale a pena discutir esta abordagem: até que ponto o intervalo de monitoramento vai crescer? A resposta é: para sempre! Sim, o tempo de resposta pode crescer indefinidamente. Ou seja, esta estratégia infelizmente não pode ser considerada prática. Além disso, como um único intervalo está acomodando o recebimento das respostas de *todos* os processos monitorados, a detecção pode ficar desnecessariamente demorada para alguns dos processos. De qualquer forma, esta é uma "implementação" clássica dos detectores $\diamond P$ que precisa ser conhecida por todos que estudam Sistemas Distribuídos.

Vamos pensar por que o Algoritmo 5.1 é $\diamond P$ -- *eventually perfect*. Em primeiro lugar vamos investigar se garante a propriedade da Completude Forte. Ela é atendida porque um processo falho (*crash*) não transmite mais mensagens de *RespostaTeste* e portanto vai ser suspeitado por todos os processos corretos a partir do intervalo de monitoramento seguinte à falha. A propriedade da Precisão Forte Após um Tempo só pode ser garantida após o GST (lembrando: *Global Stabilization Time*). O sistema fica síncrono, com um limite máximo para a troca de mensagens entre os processos, inclusive a *RequisTeste* e a *RespostaTeste*. Como o intervalo de monitoramento vai sendo ajustado (incrementado) a cada falsa suspeita, é garantido que em algum momento vai ser grande o suficiente para acomodar os tempos suficientes para todas as mensagens do tipo *RespostaTeste* serem recebidos e não haver *timeouts* prematuros.

5.2 Um Detector de Falhas Ω para o Modelo Crash

Como mencionado no início do capítulo, a eleição de líder pode ser feita com um detector de falhas. Entretanto, existe uma diferença importante: enquanto um detector tradicional retorna como saída a lista de processos suspeitos de terem falhado, a eleição de líder retorna *um único processo correto*, que é o líder. Na verdade, é possível chamar tal algoritmo de detector de falhas mesmo, e ele tem um nome grego: detector Ω (ômega). A seguir vamos estudar um detector de falhas Ω construído sobre o detector de falhas $\Diamond P$ descrito na seção anterior.

As duas propriedades que o detector de falhas Ω deve atender são as seguintes:

- Precisão após um tempo: após um intervalo de tempo finito mas desconhecido, todo processo correto elege como líder um processo correto.
- Acordo após um tempo: após um intervalo de tempo finito mas desconhecido, não há nenhum par de processos corretos que elegend como líderes dois processos distintos. Ou seja: todos os processos corretos elegend como líder o mesmo processo.

O Algoritmo 5.2 que vamos estudar a seguir é para eleição de líder no modelo temporal GST e modelo de falhas *crash*. Precisamos observar que antes do instante de tempo em que o sistema fica síncrono, diferentes processos corretos podem eleger líderes distintos. Podemos ter no sistema múltiplos líderes, inclusive simultaneamente. Mas depois que o sistema estabiliza, é garantido que um único líder será eleito, como veremos a seguir. Veja que deve ser definido um critério *determinístico* para eleição do líder sobre o conjunto de processos Corretos. Um critério determinístico retorna sempre o mesmo resultado quando aplicado sobre o mesmo conjunto Corretos. O mais popular dos critérios é o que elege como líder o processo correto de menor identificador. Assim, se o processo 0 (zero) está correto, ele é o líder. Caso contrário, se o processo 0 estiver falho mas o processo 1 estiver correto, ele será o líder. E assim por diante.

Algoritmo Eleição de Líder Detector de Falhas Ω

```
// Usa o Detector de Falhas  $\Diamond P$ ; modelo GST/falhas crash
// Executado pelo processo i;
Init: Suspeitos  $\leftarrow \emptyset$ ;
      Líder  $\leftarrow 1$ ;
Upon notificação de evento sobre processo p:
    se o evento indica que p é suspeito
    então Suspeitos  $\leftarrow$  Suspeitos  $\cup \{p\}$ ;
    senão // só pode ter havido um engano anteriormente
        Suspeitos  $\leftarrow$  Suspeitos  $- \{p\}$ ;
Upon Líder  $\neq$  CritérioLíder( $\Pi -$  Suspeitos):
    Líder  $\leftarrow$  CritérioLíder( $\Pi -$  Suspeitos)
Upon Eleição de líder chamada: return Líder;
Fim Algoritmo.
```

Algoritmo 5.2: Eleição de Líder (Detector Ω) usa detector $\Diamond P$.

Vamos agora entender porque o Algoritmo 5.2 para eleição de líder atende às duas propriedades: precisão após um tempo e acordo após um tempo. Segundo a *precisão após um tempo* um processo correto elege como líder outro processo correto (não um processo falho). Isso só pode ser verdade, pois o modelo de falhas é o *crash*, e se um processo realmente falha, ele não transmite qualquer mensagem e vai ser suspeitado e é removido do conjunto sobre o qual o líder é eleito. Desta forma, depois que um processo falha, após um tempo ele nunca mais é eleito líder.

A propriedade do acordo após um tempo só pode ser atingida depois do GST. Se o sistema fica síncrono, sabemos que, depois de um tempo suficiente, todos os processos corretos não cometem falsas suspeitas, todos os processos corretos são efetivamente considerados corretos. Assim, a função determinística *CritérioLíder()* será aplicada sobre o mesmo conjunto por todos os processos corretos, que portanto elegem o mesmo líder. Esta propriedade vem da precisão forte após um tempo mais a completude forte do detector $\Diamond P$.

5.3 Eleição de Líder no Modelo *Crash-Recovery*

A seguir vamos trabalhar em outro modelo de falhas, o *crash-recovery* que implicitamente permite a recuperação sem perda completa de estado do processo que falhou. Para permitir que o estado seja salvo entre "encarnações" do processo que falha e recupera, não há outra alternativa senão armazenar as informações em memória secundária. Esta é a principal característica do modelo *crash-recovery*: cada processo tem à sua disposição memória não volátil. Esta pode ser de qualquer tecnologia, como disco HDD (*Hard Disk Drive*) ou SSD (*Solid State Drive*), o que importa é que as informações não desapareçam após uma falha do processo.

Para acessar a memória secundária, o processo deve utilizar duas primitivas:

- *store*(info): utilizada para armazenar a info em memória secundária
- *retrieve*(info): utilizada para ler a info da memória secundária

Os algoritmos que definidos para o modelo *crash-recovery* sempre utilizam o recurso adicional da memória não-volátil para prover algum tipo de funcionalidade/serviço que não pode ser oferecido no modelo *crash* simples. No caso da eleição de líder vamos utilizar informações sobre o número de vezes que cada processo falhou/recuperou, isto é seu número de encarnações. Podemos imaginar que um processo que fica falhando e recuperando é muito menos "estável" do

que um processo que permanece correto. Esta informação pode ser então utilizada para eleger um líder mais estável.

No Algoritmo 5.3 cada processo mantém uma variável denominada "encarnação" em que guarda o número de vezes que falhou e recuperou. A variável é inicializada com 1, indicando primeira encarnação. O processo que inicializa imediatamente armazena a variável em memória secundária. Na recuperação, o processo primeiro faz a leitura do número da última *encarnação* na memória secundária, para em seguida incrementar seu valor em uma unidade e armazenar novamente o valor atualizado.

O Algoritmo 5.3 é uma versão baseada em *heartbeats (push)* do detector Ω para eleição de líder no modelo GST/*crash-recovery*. O conjunto Candidatos mantém informações sobre os processos corretos, inclusive suas *encarnações*. A cada intervalo, todo processo correto envia mensagens de *heartbeat* para todos os demais processos, informando seu número de *encarnações* atual. Ao receber tal mensagem, um processo participante verifica se o número de *encarnações* atual é maior do que aquele indicado no conjunto Candidatos. Em caso afirmativo, o conjunto é atualizado.

Para eleger o líder mais estável o possível entre os processos corretos, o *CritérioLíder()* determinístico de escolha é o seguinte: entre os processos corretos com menor número de encarnações, é eleito como líder aquele de menor identificador. Toda vez que um novo líder é eleito, é feito um incremento do intervalo de *heartbeats*, o motivo é para ter o intervalo grande o suficiente para evitar falsas suspeitas após o GST.

Algoritmo Eleição de Líder Modelo Crash-Recovery

```
// Implementação do Detector de Falhas  $\Omega$ 
// Modelo temporal GST e Modelo de falhas: crash-recovery
// Executado pelo processo i;
Init: Líder  $\leftarrow \perp$ ;
      encarnação  $\leftarrow 1$ ;
      store(encarnação)
      Candidatos  $\leftarrow \emptyset$ ;
      para todo processo p  $\in \pi$  faça
        send(p, heartbeat_i, encarnação_i;
      StartTimer(IntervaloHeartbeats);
Upon Recovery:
  Líder  $\leftarrow \perp$ ;
  retrieve(encarnação);
  encarnação++;
  store(encarnação)
  Candidatos  $\leftarrow \emptyset$ ;
  para todo processo p  $\in \pi$  faça
```

```

        send(p, heartbeat_i, encarnação_i;
Repita periodicamente no IntervaloHeartbeats:
    NovoLíder ← CritérioLíder(Candidatos);
    se NovoLíder ≠ Líder
    então IntervaloHeartbeats++;
    Líder ← NovoLíder;
    Candidatos ← ∅;
    para todo processo p ∈ π faça
        send(p, heartbeat_i, encarnação_i;
Upon receive(q, heartbeat_q, encarnação_recebida_q):
    se ∃ (q, enc) ∈ Candidatos | encarnação_recebida_q > enc
    então Candidatos ← Candidatos - (q, enc);
        Candidatos ← Candidatos ∪ (q, encarnação_recebida_q);
Upon Eleição de líder chamada: return Líder;
Fim Algoritmo.

```

Algoritmo 5.3: Eleição de Líder no modelo crash-recovery.

Para entender que o Algoritmo 5.3 atende às propriedades desejadas, vamos pensar que há pelo menos um processo que não falha (e portanto não recupera) a partir de um determinado instante de tempo igual ou posterior ao GST. A primeira propriedade que vamos examinar é a precisão após um tempo, que nos diz que um processo correto elege um processo correto. No modelo *crash-recovery* esta propriedade deve ser vista no contexto de processos instáveis, que ficam falhando e recuperando. Se um processo desses for eleito, ele pode falhar a qualquer momento, violando a propriedade. Entretanto, se houver pelo menos um processo que permanece correto, seu número de encarnações vai ser menor do que o do instável, portanto o processo que permanece correto é eleito.

Uma observação final: é possível que os *heartbeats* de processos instáveis nunca cheguem ao destino, mesmo considerando canais de comunicação perfeitos. Veja que o canal perfeito é construído sobre o canal justo, como vimos no Capítulo 3. Só há garantia de entrega da mensagem se ambos os processos permanecem corretos.

Referências

O detector de falhas Ω foi originalmente proposto em [1]. Em [2] os detectores são considerados no modelo crash-recovery. A implementação do detector $\Diamond P$ é apresentada em [3] e do detector Ω em [4].

[1] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. "The Weakest Failure Detector for Solving Consensus," Journal of the ACM, Vol. 43, No. 4, pp. 685-722, 1996.

- [2] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. "Failure Detection and Consensus in the Crash-Recovery Model," *Distributed Computing*, Vol. 13, No. 2, pp. 99-125, 2000.
- [3] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. "On the implementation of unreliable failure detectors in partially synchronous systems," *IEEE Transactions on Computers*, Vol. 53, No. 7, pp. 815-828, 2004.
- [4] Cristian Martín, Mikel Larrea, and Ernesto Jiménez. "Implementing the omega failure detector in the crash-recovery failure model," *Journal of Computer and System Sciences*, Vol. 75, No. 3 pp. 178-189, 2009.