Capítulo 4

Detectores de Falhas

Elias P. Duarte Jr.

Versão do dia 18 de abril de 2024

Pense comigo: qual o primeiro passo para tornar um sistema distribuído tolerante a falhas? Em outras palavras: você quer que o sistema continue funcionando mesmo que alguns dos seus processos falhem. Que tal começar *identificando* que processos falharam? Acredito que esta resposta é bastante intuitiva: para *tolerar* uma falha é preciso antes *saber* que a falha aconteceu. Apesar desta não ser a única alternativa para a construção de sistemas distribuídos tolerantes a falhas, o monitoramento do sistema para a descoberta de possíveis falhas é muito relevante. Pois este é o propósito dos *detectores de falhas*. Através de um detector de falhas, o sistema distribuído é monitorado, e podemos receber informações sobre quais processos falharam. É preciso lembrar que estamos falando nesse caso de falhas *crash*, isto é, consideramos que o único tipo de falha que um processo pode sofrer é a parada total, com perda do estado interno. No próximo capítulo, vamos também considerar o modelo de falhas por parada com recuperação (*crash-recovery*) voltada para a eleição de líder. A eleição de líder é, na verdade, um caso especial da detecção de falhas: na sua forma mais tradicional, os processos elegem como seu líder simplesmente o processo correto com menor identificador.

Neste capítulo vamos começar definindo os detectores de falhas. Para esta definição ser completa, é preciso explicar o contexto em que foram apresentados, que é o da impossibilidade do consenso em distribuídos assíncronos sujeitos a falhas crash. Os detectores foram originalmente propostos como uma forma de "contornar" esta impossibilidade. Vamos ver que tudo depende de duas propriedades: completude e precisão. A completude é a capacidade do detector de identificar processos que realmente falharam. A precisão é a capacidade do detector de não suspeitar incorretamente que um processo correto está falho. Com base nas duas propriedades, vamos então definir diversas classes de detectores. São então apresentados dois algoritmos básicos para a implementação dos detectores de falhas, um é baseado em testes explícitos (dito *pull-based*) enquanto que no outro os processos periodicamente enviam mensagens informando que estão vivos (dito *push-based*).

4.1 O Surgimento dos Detectores de Falhas: A Impossibilidade do Consenso

Curiosamente, os detectores de falhas não surgiram como uma estratégia de monitoramento de falhas propriamente dita. Eles surgiram no contexto do consenso, que é definido a seguir, mas tratado especificamente nos Capítulos 7, 8, 9 e 10. Para muitos, o problema do consenso é o problema "central", o mais importante em sistemas distribuídos. O consenso é também chamado de "acordo". Informalmente, os processos iniciam a execução do consenso, em que cada processo "propõe" um valor inicial, de um conjunto de valores possíveis.

O consenso entre processos não é muito diferente do consenso entre humanos. A Figura 4.1 ilustra o consenso sendo executado por 5 amigos que vão decidir seu programa de sábado a tarde. Um propõe irem ao cinema, outro propõe irem ao parque, ainda um terceiro propõe jogar truco. Ao final, todos decidem por ir ao parque.

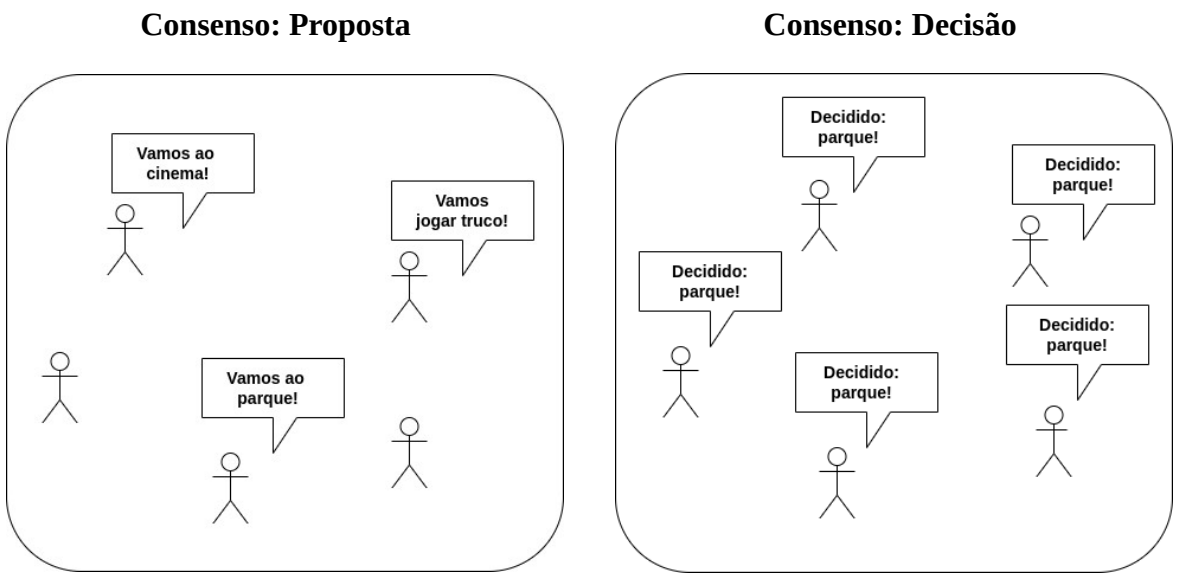

Figura 4.1: O consenso: inicialmente são *propostos* valores e, ao final, todos *decidem* por um único valor entre aqueles propostos.

Na forma mais básica o valor proposto/decidido é um único bit, que pode ser 0 ou 1. O consenso garante que todos os processos entram em acordo sobre o valor final, que é o mesmo para todos os processos. O valor final decidido deve estar entre os valores inicialmente propostos. Ou seja, no caso dos valores binários, os processos ao final do consenso decidem por 0 ou 1. Na verdade, o valor final decidido por um processo que executa o consenso pode ser uma ação. Por exemplo, considere uma usina nuclear. O reator é monitorado por múltiplos processos que, dependendo das medições realizadas, podem tomar a decisão de parar ou não parar o reator.

O consenso tem um número enorme de aplicações. Ele está na raiz da replicação distribuída, da execução de transações distribuídas, da alocação de recursos, das decisões que precisam ser tomadas em acordo por processos de diversos tipos de sistema distribuído. Veja o exemplo da replicação: é preciso garantir que todas as réplicas sempre armazenam exatemente o mesmo valor para que o sistema permaneça consistente. Se há dois ou mais comandos de alteração do valor replicado, é preciso garantir que todas as réplicas executam os comandos na mesma ordem. É importante garantir que todos os processos tomem a mesma decisão. Informalmente, as três propriedades que o consenso precisa garantir são:

● *Acordo*: todos os processos corretos decidem pelo mesmo valor
● *Validade*: se todos os processos propõem o mesmo valor inicialmente, a decisão é sempre por aquele único valor proposto
● *Terminação*: todo processo correto eventualmente decide por um valor

Sendo o consenso um problema tão importante, e tão usado o que vem agora é surpreendente. Em 1985, Fisher, Lynch e Paterson [1] provaram que é impossível garantir a execução correta do consenso em sistemas distribuídos assíncronos, sujeitos a falhas *crash*. Este é um dos resultados mais importantes da área e é chamado “Impossibilidade FLP”, das iniciais dos autores. A prova em si é surpreendente: ela é calcada no fato de que basta a falha de 1 único processo pode impedir que os demais processos cheguem a qualquer decisão. Este fato se torna ainda mais significativo quando pensamos que em um sistema assíncrono é impossível distinguir um processo lento de um processo falho, e a decisão final pode justamente depender de um único processo que está ou falho ou irremediavelmente lento.

Sendo o consenso tão importante, um número enorme de autores concentraram seus esforços na tentativa de "contornar" a impossibilidade. Veja que "contorná-la" significa apenas lidar com ela de alguma maneira, pois ela existe e não tem como ser "eliminada". Uma das alternativas neste sentido é adotar um modelo diferente do puramente assíncrono, em que não há qualquer premissa temporal [2]. Outra alternativa é alterar a própria definição do problema, para que se encaixe no que é possível.

Chandra e Toueg [3] tiveram uma ideia brilhante: eles resolveram investigar como fica o problema do consenso e sua impossibilidade se os processos tiverem a sua disposição informações sobre falhas dos demais processos. Ou seja, o sistema fica sendo monitorado e um processo pode acessar um módulo monitor que informa a lista dos processos falhos. Esses "monitores" do sistema são os detectores de falhas não confiáveis, tópico deste capítulo e descritos na próxima seção.

4.2 Os Detectores de Falhas Não Confiáveis: Completude e Precisão

Considere um sistema distribuído assíncrono, que consiste de um conjunto de N processos, ou seja $\pi = \{0, 1, 2, 3, ..., N-1\}$. Considere que os processos podem estar em um de dois estados possíveis: correto ou falho. O modelo de falhas é o da parada total (*crash*). Um detector de falhas produz como saída a lista dos processos que *suspeita* de estarem falhos. Veja que a palavra "suspeita" foi inclusive escrita em itálico na frase anterior. Este é o ponto central dos detectores de falhas para sistemas assíncronos. Lembre-se que neste tipo de sistema, um processo correto pode demorar um tempo não conhecido para executar uma tarefa, bem como uma mensagem pode levar um tempo arbitrário para ser entregue no destino. Assim, é absolutamente impossível distinguir um processo lento de um processo falho. Por isso os detectores de falhas para sistemas assíncronos podem cometer *enganos*: considerar erroneamente que um processo correto (mas lento) está falho.

O fato de que os detectores de falhas nem sempre produzem um resultado correto está refletido até mesmo no nome que Chandra e Toueg deram a eles: *unreliable failure detectors*. Isto é: detectores de falhas não confiáveis. Eles são não confiáveis porque podem reportar como falho um processo correto, e vice versa. Os franceses captaram bem a ideia quando começaram a chamar os detectores de falhas de "oráculos". A inspiração está na teoria de linguagens e no Oráculo de Delfos, na Grécia Antiga, onde era possível fazer consultas a sacerdotizas que faziam previsões. Outro ponto era se a previsão estava correta ou não, ou se a interpretação do que era falado era correta ou não.

Voltando à motivação original para a definição dos detectores de falhas, será que um detector permite o consenso? Levando em conta que os detectores cometem enganos, para saber se o detector permite -- ou não -- o consenso precisamos investigar suas propriedades. Desta forma, a pergunta chave aqui é: quais propriedades um detector de falhas deve ter para viabilizar o consenso?

Foram definidas duas propriedades básicas dos detectores de falhas:

→ Completude (*Completeness*): reflete o fato do detector de falhas suspeitar de processos que efetivamente falharam

→ Precisão (*Accuracy*): propriedade de acordo com a qual o detector de falhas não suspeita de processos que estão efetivamente corretos (também traduzida no Brasil como "acurácia", e poderia ser como "acuidade", mas vamos preferir a palavra "precisão").

Na realidade, uma destas propriedades é bem fácil de atingir na prática, enquanto a outra não. Qual é qual? Pense antes de seguir a leitura...

.... suspense...

Levando em conta que o modelo de falhas é o da parada (*crash*) e portanto um processo falho não responde a qualquer estímulo, a completude é fácil de obter. Um processo falho não produz qualquer saída e vai acabar sendo suspeitado de ter falhado. Observe que mesmo assim há um período entre a falha ocorrer e o detector efetificamente identificar aquela falha. Durante este intervalo de tempo o detector ainda vai dizer que o processo está correto. Mas após um tempo é garantido que o detector vai suspeitar do processo falho. Exatamente quanto tempo depende da frequência do monitoramente, mas o importante é que um processo falho vai acabar sempre suspeitado.

Há uma palavra em inglês que reflete que algo vai acontecer após um tempo finito mas não conhecido: *eventually.* Se uma coisa ocorre com 100% de certeza após "algum tempo" mas este tempo não é conhecido, dizemos que ocorre "eventually". Veja que a palavra correspondente em português não tem este significado! Em português, se dizemos que algo ocorre "eventualmente", aquele algo pode ou não ocorrer. Assim traduzimos o eventually para após um intervalo de tempo finito mas não conhecido ou, quando o contexto permite, simplesmente como "após um tempo".

Voltando às propriedades dos detectores de falhas. Vimos que na prática é fácil conseguir a completude. Por outro lado, a precisão não é nada fácil de obter. O problema é que em um sistema assíncrono não há limite de tempo conhecido para a execução de tarefas, nem a transmissão de mensagens. Assim um processo correto, mas lento, pode ser incorretamente diagnosticado como falho. Veja que isso ocorre no nosso dia-a-dia de uso da Internet, por exemplo quando acessamos uma página Web. Após clicar em um determinado *link*, o navegador pode informar que o site não está disponível. Pode ser que tenha falhado mesmo. Mas pode ser também uma demora para responder. Não há como garantir a precisão. Claro, pode ser que em um determinado dia, todas as respostas cheguem dentro de um intervalo de tempo esperado, e que sua ausência efetivamente indique uma falha do destinatário. Quer dizer, pode ser que eu jamais suspeite que um processo correto está falho, mas não há como garantir que isso vai ocorrer. Neste caso a propriedade é atendida, mas, de forma circunstancial, por assim dizer.

4.3 Classificação dos Detectores de Falhas

Como mencionado acima, que mesmo que a completude seja viável em sistemas reais, é preciso dar ao detector um tempo desde que um processo falha até ser detectado. Ou seja, a propriedade é atendida após um intervalo de tempo, finito mas não conhecido, isto é, *eventually*! Demora um tempo após o detector suspeitar que um processo falho está realmente falho, mas vai acontecer, com 100% de certeza.

As duas propriedades, tanto a completude como a precisão são classificadas como fortes e fracas. Vamos começar pela completude:

- Completude Forte: *eventually*, todo processo que falha é suspeitado por todo processo correto
- Completude Fraca: *eventually*, todo processo que falha é suspeitado por pelo menos 1 processo correto

A diferença entre a completude forte e a fraca está no número de processos que suspeitam daquele que falhou. Se todos suspeitam, a completude é forte. Se pelo menos 1 único processo suspeita do que falhou, então é fraca. Veja que há uma estratégia para transformar a completude fraca em forte. Basta fazer com que aquele único processo que detectou a falha informe os demais. Mais adiante, vamos estudar estratégias de comunicação confiáveis que podem ser usadas neste contexto.

A precisão também pode ser classificada como forte ou fraca:

- Precisão Forte: nenhum processo correto é suspeitado antes de falhar
- Precisão Fraca: pelo menos um processo correto jamais é suspeitado antes de falhar

Observe que no caso da precisão, o foco está no processo que não é suspeitado, e não nos processos executando o detector, como no caso da completude. Além disso, no caso da precisão, as versões *eventually* são definidas a parte:

- Precisão Forte Eventual: existe um tempo após o qual nenhum processo é suspeitado antes de falhar
- Precisão Forte Eventual: existe um tempo após o qual pelo menos 1 processo correto não é suspeitado de ter falhado

Com base nas duas classes de completude e nas quatro classes de precisão os detectores de falha são então organizados em oito classes distintas, como mostra a Tabela 4.1. Como é possível transformar a completude fraca em forte, bastaria apresentar as classes correspondentes à completude forte, a primeira linha.

Detectores de Falhas: Classes	Precisão Forte	Precisão Fraca	Precisão Forte *Eventual*	Precisão Fraca *Eventual*
Completude Forte	*Perfect* **P**	*Strong* **S**	*Eventually Perfect* **◊P**	**◊S**
Completude Fraca	**Q**	*Weak* **W**	**◊Q**	*Eventually Weak* **◊W**

Tabela 4.1: Classes de detectores de falhas.

A Tabela 4.1 pode parecer incompleta a princípio, pois alguns detectores têm nome e outros não -- mas é assim mesmo, alguns detectores tem nome outros são apenas conhecidos pela sigla. Optamos por deixar os nomes em inglês, por causa da palavrinha *eventually*, que é sempre um problema na língua portuguesa. Os detectores mais importantes são, sem sombra de dúvidas: o Perfeito (P), o Perfeito após um Tempo (◊P), o Forte após um Tempo (◊S) e o mais fraco de todos, o ◊W. Eu costumo dizer que Chandra e Toueg, os inventores dos detectores de falhas, eram muito bons de marketing. Eles organizam os detectores em 8 classes distintas, para depois mostrar que até mesmo um detector da classe mais fraca de todas é capaz de permitir o consenso.

Este resultado merece atenção. Opa, quer dizer que se tivermos um detector de falhas ◊W então o consenso é possível em sistemas assíncronos sujeitos a falhas *crash*? Maravilha, vamos implementar este detector e então problema resolvido! Antes fosse. O ponto é que não existe implementação prática de um detector para obedecer a uma determinada classe. As propriedades vão ser atendidas dependendo do que acontecer no sistema. Podemos pensar: puxa, mas basta esperar tempo suficiente que não cometeremos falsas suspeitas. Certo? Certo, mas ficar esperando por um tempo longo demais não é solução prática. Podemos até mesmo aumentar o tempo de espera na medida em que concluímos que houve uma falsa suspeita: cada vez que o detector identifica que cometeu um engano, identificando como correto um processo falho, ele incrementa o intervalo de tempo para suspeitar. É certo que ao longo do tempo, ele vai acabar não suspeitando mais. Mas pode acabar também com intervalos de espera inviáveis.

Na próxima seção vamos ver implementações genéricas e práticas de detectores. Se você usar um destes detectores em um sistema real, pode ter, na verdade, todas as classes de detectores, dependendo do que ocorrer durante a execução (processos ficarem mais lentos, rede congestionada atrasar mensagens, etc).

4.4 Implementação dos Detectores de Falhas

Pronto, agora que já definimos os detectores de fallhas, e vimos sua classificação de acordo com as propriedades de completude e precisão, podemos partir para a implementação. Veja que nesta seção vamos estudar algoritmos para implementar detectores, não programas concretos em linguagens de programação específicas.

Existem duas estratégias básicas para a implementação de detectores de falhas: *pull* e *push*. Na estratégia *pull* os processos são configurados para se testarem uns aos outros para, através dos resultados dos testes, definirem os estados de todos os processos do sistema. Enquanto isso, na estratégia *push*, os processos são pré-configurados para enviar periodicamente mensagens

informando que estão corretos, "vivos". Estas mensagens costumam ser chamadas de *heartbeats*, "batidas do coração" em inglês. Nesta estratégia a ausência de um (ou mais) *heartbeats* pode causar um timeout que leva à suspeita do processo silencioso. Na estratégia *pull*, o testes podem ser consistir de procedimentos que façam uma avaliação completa do estado do processo testado. Em ambos os casos, quando o sistema é assíncrono, torna-se impossível diferenciar um processo lento de um processo falho.

Em seguida vamos ver os algoritmos que denominamos de "força-bruta" para implementar detectores de falhas *pull* e *push*. Estes algoritmos são ditos de força-bruta porque: (i) no caso push, todos os processos mandam mensagens de heartbeat para todos os demais e (ii) no caso pull, todos os processos testam todos os demais. Mais à frente vamos ver alternativas *pull* ao algoritmo da força bruta, são os algoritmos vRing (Capítulo 6) e vCube (Capítulo 7).

O Algoritmo 4.1 mostra em pseudo-código um detector de falhas força-bruta *push*, executado pelo processo i. Periodicamente, o processo i envia seu $heartbeat_i$ para os demais processos. Cada processo que executa o algoritmo mantém um conjunto com aqueles processos que considera Suspeitos. Quando o detector é invocado, a saída que produz é justamente o conteúdo deste conjunto. Inicialmente, o conjunto Suspeitos é inicializado como vazio. Para cada processo j, o processo i calcula um valor de tempo denominado $timeout_j$ -- quando um $heartbeat_j$ não é recebido durante um intervalo de tempo igual a $timeout_j$, o processo i passa a suspeitar que o processo j está falho. É possível usar diversas estratégias para calcular o *timeout*, o importante é manter o timeout atualizado na medida em que vão chegando os *heartbeats*, para evitar enganos. Uma das estratégias mais utilizadas hoje para cálculo de *timeout* é o algoritmo de Van Jacobson usado pelo TCP [4].

Algoritmo Detector de Falhas Força-Bruta Push executado pelo processo i;

Init: Suspeitos ← S - {i};

for all j ∈ S - {i}: compute $timeout_j(heartbeat_j)$;
send($heartbeat_i$) to j;

Upon $heartbeat_i$ interval expires: for all j ∈ S - {i}: send($heartbeat_i$) to j;

Upon receive($heartbeat_j$): Suspeitos ← Suspeitos - {j}
update $timeout_j(heartbeat_j)$;

Upon $timeout_j$ expires: Suspeitos ← Suspeitos + {j}

Upon Detector de Falhas invocado: return Suspeitos;

Algoritmo 4.1: Detector de falhas força-bruta *push*.

Em seguida, o Algoritmo 4.2 é do algoritmo do detector de falhas força-bruta *pull*. O processo i mantém o mesmo conjunto Suspeitos, que é retornado quando o detector é invocado. O monitoramento agora é feito através de um teste explícito, assim o processo i envia uma requisição de test ($TestRequest_i$) para o processo j e aguarda a reposta correspondente ($TestReply_j$). Caso a resposta não chegue dentro do intervalo de tempo $timeout_j$, o processo i passa a suspeitar que o processo j está falho.

Algoritmo Detector de Falhas Força-Bruta Pull executado pelo processo i;

Init: Suspeitos ← S - {i};

for all j ∈ S - {i}: compute $timeout_j(TestReply_j)$;
send($TestRequest_i$) to j;

Upon heartbeat$_i$ interval expires: for all j ∈ S - {i}: send($heartbeat_i$) to j;

Upon receive(TestReply j): Suspeitos ← Suspeitos - {j}
update $timeout_j(heartbeat_j)$;

Upon timeout$_j$ expires: Suspeitos ← Suspeitos + {j}

Upon Detector de Falhas invocado: return Suspeitos;

Algoritmo 4.2: Detector de falhas força-bruta *pull*.

Algumas palavras finais para terminar o capítulo. Os detectores de falhas podem ser vistos hoje como ferramentas para o monitoramento de falhas em sistemas distribuídos. Entretanto, tiveram um impacto gigantesco na teoria de sistemas distribuídos. Tudo partiu da sua proposta original, que visou compreender quais as propriedades um detector de falhas deveria garantir para permitir o consenso em sistemas distribuídos assíncronos sujeitos a falhas *crash*. Entretanto, o impacto dos detectores foi muito além: virtualmente *todos* os problemas de sistemas distribuídos foram repensados com a adição de detectores. Eles permitiram mesmo uma classificação dos problemas em si, tendo em vista a classe de detector necessária para resolver o problema. Assim, há problemas, por exemplo, que demandam um detector perfeito, e assim por diante. Nos capítulos a frente em vários casos vamos mencionar a classe de detector que o problema em questão demanda para ser resolvido.

Referências Bibliográficas

[1] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," Journal of the ACM (JACM), Vol. 32, No. 2, pp. 374-382, 1985.

[2] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the Presence of Partial Synchrony," Journal of the ACM (JACM), Vol. 35, No. 2, pp. 288-323, 1988.

[3] Tushar D. Chandra, Sam Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," Journal of the ACM (JACM), Vol. 43, No. 2, pp. 225-267, 1996.

[4] Vern Paxson, et al, "Computing TCP's retransmission timer," RFC 6298, 2011.