Capítulo 3

Os Canais de Comunicação

Elias P. Duarte Jr.

Versão do dia 16 de abril de 2024

Um sistema distribuído é definido como um conjunto de processos que se comunicam trocando mensagens. Esta troca de mensagens ocorre através de um canal de comunicação entre os processos. Na verdade, para viabilizar a troca de mensagens é necessária uma rede -- e todos os protocolos que a constituem -- que são abstraídos como o canal de comunicação. Sendo uma "abstração", o canal de comunicação nos permite ignorar todos os detalhes a rede, começando pela parte física, incluindo os protocolos das diversas camadas. Podemos imaginar apenas dois processos quaisquer conectados por um enlace virtual, que é o canal de comunicação. Neste capítulo vamos começar definindo as mensagens que são a base da comunicação e em seguida vamos definir os dois modelos tradicionais de canais de comunicação, o justo e o confiável.

2.1 A Mensagem

Se os processos de um sistema distribuído se comunicam utilizando troca de mensagens, precisamos o quanto antes definir o que é uma mensagem, exatamente. Antes, vale relembrar: uma mensagem que vamos chamar de *msg* é transmitida utilizando a primitiva *send(msg)*. No destino, a mensagem é recebida com a primitiva *receive(msg)* e possivelmente entregue à aplicação com a primitiva *deliver(msg)*. Uma mensagem é um registro (tal qual definido como uma *struct* da linguagem C ou lista heterogênea da linguagem Python) e consiste de uma sequência de campos. A mensagem é serializada para ser transmitida pelo canal de comunicação. Cada mensagem carrega tanto informações de controle, como os dados (*payload* - carga paga). A parte da mensagem que tem campos referentes às informações de controle é o cabeçalho, ou *header*. Entre as informações mais comuns que aparecem no cabeçalho estão os identificadores dos dois processos que estão comunicando: origem (*source-id*) e destino (*destination-id*). Veja que o identificador em si pode ser constituído de vários campos. Por exemplo, na Internet o identificador de um processo consiste do par *(endereço IP, porta)*, que aqui é tratado simplesmente como *source-id* ou *destination-id.*

É surpreendentemente fácil **identificar cada mensagem de forma única** no sistema distribuído. Para isso cada processo deve manter localmente um contador das mensagens que transmite. Surpreendentemente, este contador é muitas vezes chamado de *timestamp*, apesar de não ter nada a ver diretamente com "tempo". Lembre-se: o *timestamp* é um contador simples, tendo valor 1 para a 1a mensagem, valor 2 para a 2a mensagem, e assim por diante. O identificador da mensagem (*msg-id*) é então constituído do identificador do processo origem, mais o valor do contador: (*source-id*, *timestamp*). Ter uma forma de identificar cada mensagem do sistema, mesmo que consista de milhares de processos e zilhões de mensagens traz grandes vantagens. É possível, por exemplo, confirmar o recebimento de uma mensagem específica. Também é possível saber se uma mensagem que acaba de ser recebida do canal de comunicação já foi entregue, ou seja, se é uma mensagem duplicada. É possível também usar os identificadores para estabelecer uma ordem de entrega, como veremos a frente. Vale a pena não esquecer das diversas vantagens de ser capaz de identificar cada mensagem de forma única no sistema distribuído.

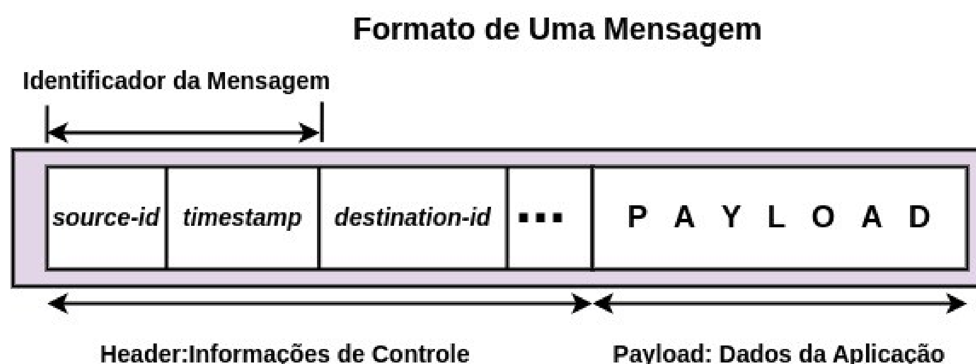

Figura 3.1: Estrutura típica de uma mensagem.

A Figura 3.1 mostra uma mensagem típica que os sistemas distribuídos utilizam. Os campos da mensagem daquela figura são: identificador da origem (*source-id*) e *timestamp* (contador de mensagens da origem) formando o identificador da mensagem; identificador do destino (*destination-id*). O *header* pode ter diversos outros campos, como indicado pelas reticências. O payload é onde estão dados específicos da aplicação, para a qual a mensagem é entregue.

Vale destacar que quando se especifica uma primitiva de envio de uma mensagem de um algoritmo distribuído específico podem ser explicitados campos do cabeçalho fora da mensagem. O objetivo é destacar o que é importante no algoritmo em questão. Por exemplo:

send(*source-id*, *dest-id*, *msg*)

Neste exemplo, os endereços de origem e destino são destados fora da mensagem, e *msg* indica os demais campos inclusive o *payload*. Qualquer campo do *header* pode ser destacado desta forma.

2.2 O Canal de Comunicação: Uma Abstração da Rede

A representação gráfica mais comum de um canal de comunicação no contexto de sistemas distribuídos é como um segmento de reta conectando dois processos, como mostra a Figura 3.2. Esta figura acaba levando a uma confusão bastante comum: fica a impressão que existe um *link* físico conectando os dois processos. Na verdade até pode ser o caso, mas não necessariamente.

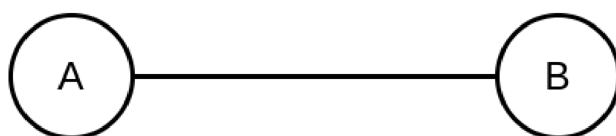

Figura 3.2: A representação gráfica usual para um canal de comunicação entre os processos A e B.

O canal de comunicação é uma abstração para o meio de transmissão de mensagens entre dois processos, seja qual for ele. Assim, é possível que os dois processos estejam em dois países dos mais distantes do planeta, conectados pela Internet mundial. O exemplo da Figura 3.3 mostra justo o processo A no Brasil e o processo B na China, conectados através da Internet. O canal de comunicação neste caso consiste de dezenas de roteadores intermediários, além de enlaces das mais diversas tecnologias, e usa série de protocolos de rede que permitem aos dois processos o envio e a recepção de mensagens. Nesse caso, toda essa complexidade é abstraída: consideramos simplesmente que há um canal de comunicação entre os processos, como o desenhado como o segmento de reta da Figura 3.2.

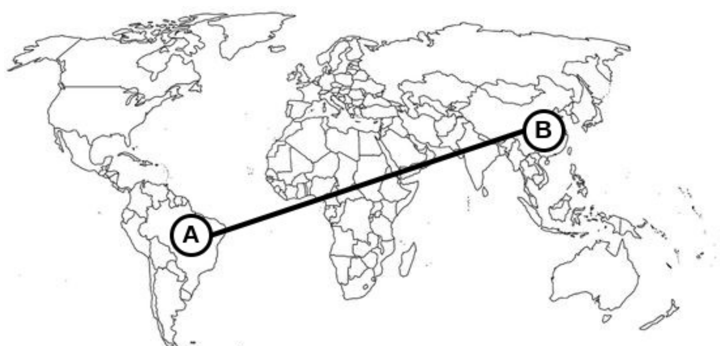

Figura 3.3: Os processos A e B estão em dois países dos mais distantes entre si do mundo.

Na verdade, em casos excepcionais, pode ser que nem sequer exista uma "rede" propriamente dita entre os processos, eles podem estar sendo executados na mesma máquina. Alternativamente podem estar na mesma rede local ou, como mencionado, conectados por uma ou mais redes formadas por enlaces de quaisquer tecnologias (comunicação sem-fio, um enlace de fibra ótica, cabo coaxial, etc.). A Figura 3.4 mostra estes três casos e insiste que todos são representados pelo mesmo segmento de reta entre os dois processos.

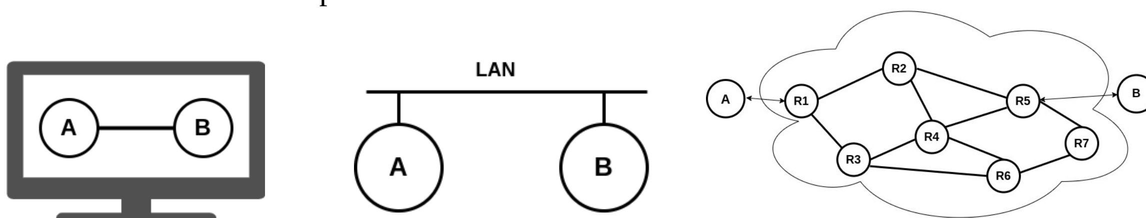

Figura 3.4: Os processos A e B podem estar executando na mesma máquina, em duas máquinas de uma rede local, ou podem estar conectados por uma ou mais redes através de diversos roteadores. Em todos esses casos a representação do canal de comunicação é a mesma, mostrada na **Fig. 3.2**.

Vale destacar que sempre há uma pilha de protocolos de rede que viabiliza a transmissão e recepção de mensagens sobre o canal de comunição. Mas todos aqueles protocolos também ficam "invisíveis". Só enxergamos as primitivas *send(msg)* e *receive(msg)* para transmitir mensagens através do canal de comunicação.

Existem dois modelos principais de canal de comunicação, que refletem o seu funcionamento: o canal justo e o canal perfeito, que são descritos nas próximas seções. Para antecipar para os ansiosos: o canal justo é aquele que, ocasionalmente, pode perder mensagens, enquanto que em um canal perfeito, como o nome implica, nunca há perda de mensagens. Vamos examinar os dois modelos em mais detalhes. Veja que estes modelos são muito importantes: toda vez que se especifica ou descreve um sistema ou algoritmo distribuído é fundamental explicitar o tipo de canal de comunicação considerado.

2.3 O Canal de Comunicação Justo

O primeiro tipo de canal de comunicação é o "justo", em tradução livre do inglês *fair-loss*. Este tipo de canal de comunicação pode perder mensagens. Assim, nem toda mensagem transmitida através de um canal de comunicação justo chega ao destino. Entretanto, fique atento: o canal *justo* não tem este nome a toa. Existe uma **boa** probabilidade de sucesso na transmissão da mensagem. A

propriedade da justiça de um canal reflete este fato. A forma como esta propriedade é especificada na teoria de sistemas distribuídos é curiosa:

- *Se uma mensagem é transmitida infinitas vezes pelo processo A para o processo B, e ambos os processos não falham, então a mensagem é recebida infinitas vezes pelo processo B.*

Veja que não se espera que nenhum processo transmita uma mensagem infinitas vezes, nem que o destinatário receba a mensagem infinitas vezes. O ponto é que se a probabilidade de sucesso da transmissão de uma mensagem sobre um canal justo é maior que zero. Outras duas propriedades comuns de um canal justo são mais intuitivas. A primeira propriedade diz respeito ao fato do canal não gerar mensagens espúrias, se uma mensagem foi recebida então ela foi propriamente transmitida. Por fim, pode haver duplicação de mensagens neste canal, mas é de pequena monta, uma mensagem não é duplicada infinitamente.

Na prática, o canal de comunicação justo representa bem o serviço fornecido pela Internet e as redes IP em geral: o protocolo IP perde mensagens ocasionalmente. Pode haver duplicação de mensagens. Mas não gera mensagens "espúrias", não gera nenhuma mensagem na verdade, se um pacote IP está trafegando na rede é porque algum processo transmitiu aquele pacote. Por consequência, se constuímos o sistema distribuído usando o protocolo de transporte UDP os canais de comunicação também são justos. O TCP, ao contrário, produz canais de comunicação perfeitos, descritos na próxima seção.

2.3 O Canal de Comunicação Perfeito

O segundo tipo de canal de comunicação é o perfeito, também chamado de confiável. Diferente do canal de comunicação justo, o canal perfeito deve ser construído explicitamente, e justamente sobre um canal justo. Os enlaces físicos de comunicação das mais diversas tecnologias podem ser considerados canais justos. Como mencionado acima, a própria rede IP -- que oferece a comunicação através da Internet -- pode ser vista como um canal justo. Relembre que o canal justo pode perder mensagens ocasionalmente. Além disso, as mensagens também podem ser duplicadas. Estes são os dois pontos resolvidos pelo canal perfeito, que também garante que não são geradas mensagens espúrias, e mais duas propriedades: não-duplicação e entrega confiável, descritas a seguir.

A primeira propriedade sobre a qual é necessária uma ação explícita é a não-duplicação: o destino só entrega cada mensagem uma única vez. Veja que o verbo que se usou aqui é "entrega", que remete à nossa primitiva *deliver()*. Desta forma, mesmo que uma mensagem seja recebida múltiplas vezes -- isto é, através da primitiva *receive()* -- ela só é entregue uma única vez. Como garantir a não duplicação? Na verdade, lembrando que conseguimos identificar cada mensagem de forma única no sistema distribuído, não é difícil garantir que cada mensagem é entregue uma única vez. Basta manter informações sobre todas as mensagens já entregues. Desta forma, quando chega uma nova mensagem *msg* recebida com *receive(msg)*, em seguida checamos se ela já foi entregue. Isso pode ser feito por exemplo definindo o conjunto *Delivered* com todas as mensagens já entregues. Se a nova mensagem msg não estiver no conjunto *Delivered*, então ela pode ser entregue e é adicionada ao conjunto *Delivered*.

Há formas eficientes de armazenar as informações sobre quais mensagens já foram entregues. Se por exemplo as mensagens tem *timestamps* sequenciais (1, 2, 3, ...) então basta armazenar o *timestamp* da última mensagem recebida. Assim, por exemplo se o conjunto *Delivered* contiver o *timestamp* 5, isso significa que as mensagens 1, 2, 3, 4 e 5 foram entregues. Claro que se mensagens faltaram para garantir a sequência exata, múltiplos *timestamps* devem ser adicionados ao conjunto *Delivered*. Considere por exemplo que *Delivered* = {5,7}. Isso significa que todas as mensagens até a 5a, além da mensagem com *timestamp* 7 já foram entregues, mas a mensagem 6 ainda não foi entregue.

A segunda propriedade que precisa ser explicitamente trabalhada para conseguirmos um canal de comunicação perfeito é a entrega confiável. Ela pode ser definida como a seguir:

- *Se o processo A transmite uma mensagem para o processo B, e ambos os processos permanecem corretos (i.e., não falham), então após um intervalo de tempo finito o processo B recebe aquela mensagem.*

O que esta propriedade da entrega confiável garante é que se uma mensagem é transmitida, então ela é recebida. Veja que esta propriedade deve ser garantida 100% das vezes, não adianta garantir 99,99999%, pois daí o canal é justo, não perfeito.

Na teoria de sistemas distribuídos, que não tem preocupações práticas, uma forma trivial de implementar a entrega confiável em um canal perfeito construído sobre um canal justo é simplesmente ficar transmitindo *todas* as mensagens infinitamente. Por causa da propriedade da justiça (teórica) de que se uma mensagem for transmitida infinitas vezes, ela é recebida infinitas vezes, as infinitas transmissões garantem o recebimento no destino. Claro que isso não é feito na prática: usar recursos caros de uma rede para mandar as mesmas mensagens infinitas vezes não faz

qualquer sentido. Mas veja que, apesar dos fortes indícios de loucura, os teóricos de sistemas distribuídos, na verdade, adotam esta solução simplesmente para terem um atalho que lhes permite ter canais perfeitos sem maiores preocupações. Assim eles podem focar em outros algoritmos distribuídos mais complexos, considerando que têm canais de comunicação perfeitos à sua disposição.

Muito bem: como vamos implementar a entrega confiável para conseguirmos na prática um canal de comunicação perfeito sobre um canal de comunicação justo? A ideia é usar confirmações de recebimento, um tipo de mensagem popularmente conhecida como um ACK (de *ACKnowledgment*). O ACK é uma mensagem transmitida de um processo a outro informando do recebimento de uma determinada mensagem. Desta forma se o processo A transmite uma mensagem para o processo B, logo após esta mensagem ser recebida por B, este envia um ACK para A, informando que a mensagem foi recebida. Fácil, né? Pois nem tanto. O problema é que agora temos que garantir que o ACK também vai chegar ao destino. Como ele é transmitido sobre um canal justo, ele pode se perder! Isso está no "contrato" do canal justo. A Figura 3.5 ilustra o ACK se perdendo.

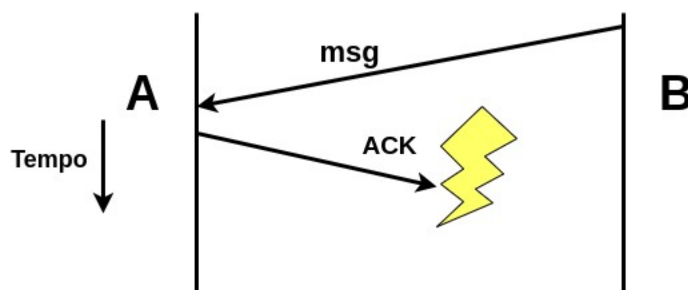

Figura 3.5: O ACK que está confirmando o recebimento da mensagem msg se perde.

Como resolver o problema de garantir que o ACK chegou ao destino? Que tal confirmar o recebimento do ACK, com um ACK-ACK. Problema resolvido? Não! O ACK-ACK também pode se perder, como ilustra a Figura 3.6. Precisamos também confirmar o recebimento do ACK-ACK, com um ACK-ACK-ACK, e assim por diante... Este é o conhecido problema dos dois exércitos, descrito magistralmente por Tanenbaum quando explica o encerramento de uma conexão TCP [Tanenbaum, 2021]. Para surpresa geral, não há solução para o problema. Não existe protocolo para confirmar em um número finito de mensagens o recebimento de uma mensagem específica, transmitida sobre um canal de comunicação que pode perder mensagens (isto é, justo).

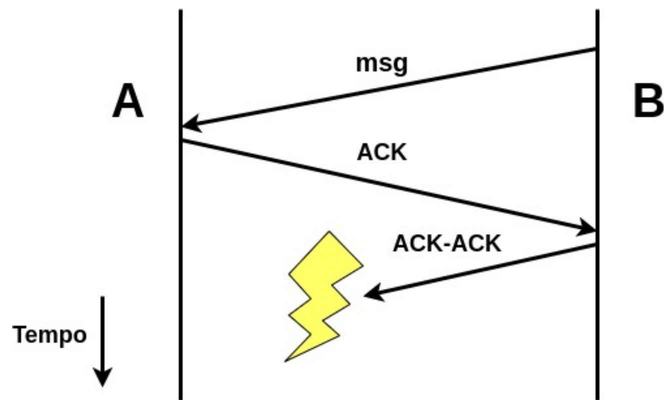

Figura 3.5: Confirmar o ACK não ajuda: o ACK-ACK também pode se perder.

Qual o impacto desta impossibilidade na prática? Em primeiro lugar, é importante deixar claro que não se trata de uma curiosidade teórica, vivemos este fato todos os dias da nossa vida, pois ele está presente por exemplo no encerramento de uma conexão TCP, protocolo utilizado mundialmente por bilhões de seres humanos na Internet. Voltando ao nosso problema: como vamos fazer para contruir um canal perfeito sobre um canal justo? A ideia é a seguinte: vamos manter o uso de ACKs. Mas precisamos manter um conjunto de mensagens transmitidas e não confirmadas, que vamos chamar de *Sent*. Periodicamente todas as mensagens no conjunto *Sent* são retransmitidas. Quando chega um ACK para uma mensagem específica, esta é retirada do conjunto *Sent*. O Algoritmo 3.1 especifica um enlace perfeito construído sobre um enlace justo usando esta estratégia.

Algoritmo do Canal de Comunicação Perfeito com ACKs

//Construído sobre um **Canal Justo**

Variavéis: Sent, Delivered Conjuntos de Mensagens; Periodo: Intervalo de Tempo;

Init: Sent ← vazio; Delivered ← vazio; Start-Timer(Periodo);

UPON timeout do Periodo: for all msg in Sent do send(msg); Start-Timer(Periodo);

UPON there is a new msg to transmit: Sent ← Sent U {msg}; send(msg);

UPON receive(msg): if msg is an ACK
then remove confirmed msg from Sent;
else if msg not in Delivered
then deliver(msg);
Delivered ← Delivered U {msg};
send(ACK for the received msg);

Algoritmo 3.1: Canal de comunicação perfeito construído sobre canal justo.

O Algoritmo 3.1 mantém os conjuntos *Sent* e *Delivered*, já descritos. O conjunto *Sent* contém as mensagens transmitidas e que ainda não tiveram seu recebimento confirmado pelo destinatário. O conjunto *Delivered* contém as mensagens recebidas do destinatário que já foram entregues à aplicação. A variável *Período* determina o intervalo de tempo entre transmissões de todas as mensagens do conjunto Sent. Se uma mensagem é recebida e não é um ACK, então há o envio do ACK para aquela mensagem -- veja que ACKs já podem ter sido transmitidos anteriormente para aquela mensagem, mas podem ter se perdido, assim *toda* mensagem recebida é confirmada (exceto ACKs, que logicamente não são confirmados). Mensagens recebidas que não são ACKs e não estão no conjunto *Delivered* (portanto ainda não foram entregues à aplicação) são entregues e colocadas no conjunto *Delivered*.

Vamos pensar nesta solução. Qual o número máximo de vezes que uma mensagem pode ser transmitida até que tenha finalmente seu recebimento confirmado? A resposta é: infinitas vezes. Lembre-se que acima chamamos os teóricos da computação distribuída de loucos por definirem um algoritmo em que todas as mensagens são transmitidas infinitas vezes. Mas veja que isso pode acontecer com nosso algoritmo baseado em ACKs também. Basta os ACKs se perderem sobre o canal de comunicação justo. Claro que a probabilidade de isso acontecer em um canal justo das nossas redes de hoje é muito pequena. Claro que, mesmo que um número enorme de ACKs se perca, não vamos esperar infinitamente por uma confirmação: provavelmente vamos considerar que o canal de comunicação se rompeu, se a confirmação não chegar depois de várias tentativas. No caso do encerramento da conexão TCP é feito um único envio do último ACK-FIN que encerra a conexão, mas esta permanece aberta por um intervalo de tempo, para o caso daquele ACK-FIN se perder. Entretanto o intervalo pode ser insuficiente: pode chegar nova mensagem, e daí a conexão não existe mais, a resposta é um RESET enviado pelo próprio sistema operacional.

Existe um teorema clássico da teoria de sistemas distribuídos que prova essa impossibilidade que acabamos de estudar na prática. Este livro não tem teoremas, que podem ser fartamente encontrados na literatura da área. Mas vale a pena descrever como esta prova é feita, para meus leitores e leitoras ficarem a par deste resultado importante. O teorema diz que é impossível que dois processos que não falham jamais mas estão conectados por canal justo decidam pela execução de uma entre duas ações possíveis. A prova é feita com base na impossibilidade. Considere que sim, há diversas soluções para o problema. Agora pegue a solução que precisa do menos trocas de mensagens para resolver o problema. A última mensagem desta solução pode se perder, pois o canal de comunicação é justo. Assim, veja que nossa solução não pode depender dessa última mensagem, ela tem que funcionar com ou sem aquela mensagem. Assim, existe uma solução com menor número de mensagens! Mas tudo começou justo com a solução com menos mensagens entre todas.

Absurdo! Não há solução que resolve o problema com número finito de mensagens. Na prática, se houver muitas retransmissões, não vamos esperar até o infinito: após um tempo desistimos por considerar o canal rompido. Mas quase sempre tudo vai funcionar bem com grande probabilidade: vamos ter as confirmações recebidas corretamente em uma ou algumas poucas tentativas, no máximo.