Capítulo 8 – Problemas com Sistemas Distribuídos

Hugo Paulino Bonfim Takiuchi
Curitiba – UFPR
15/05/2018

Introdução

- Lado Pessimista.
- O que pode dar errado vai dar errado.
- Em um sistema distribuído há mais possibilidades de erros do que em um único computador.
- Apesar de tudo dar errado o sistema tem que funcionar como esperado (garantias que o usuário espera).

Faults and Partial Failures

- Um computador apresenta comportamentos normalmente previsíveis: funcionam ou não. Não tem meio termo (*fuzzy*).
- Hardware funcionando corretamente apresenta resultados *determinísticos* (mesma saída para a mesma entrada).
- Se existe algum problema no hardware (memória corrompida) acontece uma falha total no sistema (tela azul).

Faults and Partial Failures

- *Fault*: um componente interno do sistema deixa de funcionar como esperado.
- *Failure*: O sistema como um todo não produz a saída esperada.
- Uma um mais *faults* podem ou não implicar em uma *failure*.
- Em caso de uma *fault* é preferível causar uma *failure* em todo o sistema do que retornar um resultado errado (mais complicado de lidar).

Faults and Partial Failures

- Em um sistema distribuído é necessário lidar com os problemas fuzzy (mais possibilidade de falhas).
- *Partial Failure:* Parte dos componentes do sistema estão falhos (de maneira imprevisível) e outras partes ainda continuam funcionando.
- *Partial Failure* é não determinísticos. Um sistema distribuído podem funcionar ou falhar imprevisivelmente em alguns casos.
- O tempo de envio de mensagens também é não determinístico.

Cloud Computing and Supercomputing

- Sistemas de computação em larga escala:
- *Supercomputing*:
  - → *Cluster* com pequeno número de computadores com alto desempenho (High Performance Computing).
  - → Possui milhares de CPUs.
  - → Utilizado para realização de tarefas científicas (previsão do tempo, dinâmica molecular).

Cloud Computing and Supercomputing

- Sistemas de computação em larga escala:
- *Cloud Computing*:
  - → Geralmente associado ao processamento e armazenamento de dados distribuídos entre diversas máquinas (*multi-tenant datacenter*).
  - → *Cluster* com mais computadores e menor desempenho.
  - → Os computadores são ligados pela internet (IP/ethernet)
  - → Flexibilidade na alocação de recursos.

Cloud Computing and Supercomputing

- Tratamento de falha em *supercomputing*:
  - ➔ Máquina(nodo) verifica o estado interno periodicamente.
  - ➔ Cria um *checkpoint (estado seguro)*.
  - ➔ Caso o nodo estiver falho a solução mais comum é parar todo o cluster.
  - ➔ recuperar o nodo falho.
  - ➔ retornar a execução para algum *checkpoint*.
- Escala uma partial failure para uma total failure.

Cloud Computing and Supercomputing

- Foco sobre *serviços de internet*:
  - ➔ Sistema disponível para o usuário durante todo o tempo e com baixa latência (não pode parar o cluster inteiro para recuperar um nodo).
  - ➔ *Supercomputing* utilizam hardware especializados e se comunicam por memória compartilhada e RDMA.
  - ➔ *Cloud Servide* hardware mais simples e apresentam um desempenho equivalente ao baixo custo. Apresentam mais falhas.

Cloud Computing and Supercomputing

- Foco sobre *serviços de internet*:
  - → *Datacenters* são baseados em IP/ethernet usando a topologia *Clos* (melhor escalabilidade e divisão de recursos).
  - → *Supercomputing* usam topologias especializadas como *Mesh* e *Torus* (multi-dimensionais). Melhor desempenho para quantidade de dados processados, porém difícil de escalar.
  - → Sistema com milhares de nodos podem haver vários nodos falhos. Ignora os nodos falhos (tempo e recursos otimizados).

Cloud Computing and Supercomputing

- Foco sobre *serviços de internet*:
  - Sistema tolerante a falhas continua funcionando mesmo com nodos falhos. Útil para operações e manutenção (*Rolling upgrade* e *cloud computing*).
  - Sistema distribuído geograficamente se comunicam pela internet (sobrecarregada e não confiável). Supercomputing as máquinas geralmente estão próximas (mais próximo menor latência).

Cloud Computing and Supercomputing

- Sistemas distribuídos permitem *partial failure* e *tolerância a falhas* (sistema confíável com componentes não-confiáveis).

- *Confiável (Reliable):* Sistema funciona como esperado.

Construindo Sistemas confiáveis (reliable) com componentes não confiáveis (unreliable)

- Um sistema é tão confiável quanto o seu componente menos confiável (falso). Sistemas confiáveis construídos sobre bases não confiáveis:
  - Um corretor de erros permite dados digitais serem transmitidos precisamente sobre um canal não confiável. Algum bit transmitido errado.
  - IP não confiável (perder, duplicar, atrasar e reordenar os pacotes). TCP é mais confiável pois retransmite pacotes, elimina duplicação e reordena.

Construindo Sistemas confiáveis (reliable) com componentes não confiáveis (unreliable)

- Sistema possa ser mais confiável do que seus componentes, porém há limites:
  - O corretor de erros de transmissão possui um limite de dados no canal.
  - O TCP não lida com atrasos.

Redes não confiáveis (Unreliable)

- *Shared-nothing*:
  - Máquinas comunicam através da troca de mensagens pela rede.
  - Cada uma tem seu próprio disco e memória.
  - Acesso aos dados de outra máquina através de um request (pedido).
- O sistema distribuído descrito no livro utiliza shared-nothing.

Redes não confiáveis (Unreliable)

- *Shared-nothing* é uma abordagem utilizada pela internet devido a 3 fatores:
  - → Relativamente barato pois não requer um hardware especial.
  - → Utiliza um serviço de computação em nuvem comum.
  - → Confiável pela redundância que os vários datacenters geograficamente espalhados proporcionam. Caso algum falhe será utilizado outro para resolver a tarefa.

Redes não confiáveis (Unreliable)

- A internet é *assíncrona:*
  - ➔ Um nodo A envia uma mensagem para outro nodo B.
  - ➔ O nodo A não sabe quando a mensagem chegou ou se ela chego para o nodo B.
- *Request*: mensagem enviada pela origem (pedido).
- *Response*: mensagem enviada pelo destino em resposta a um pedido.

Redes não confiáveis (Unreliable)

- Exemplos de erros possíveis quando se espera uma resposta:

  1) Request foi perdido (cabo de rede desconectado).

  2) Request aguardando na fila de envio (rede sobrecarregada).

  3) Nodo remoto falhou (aconteceu um crash).

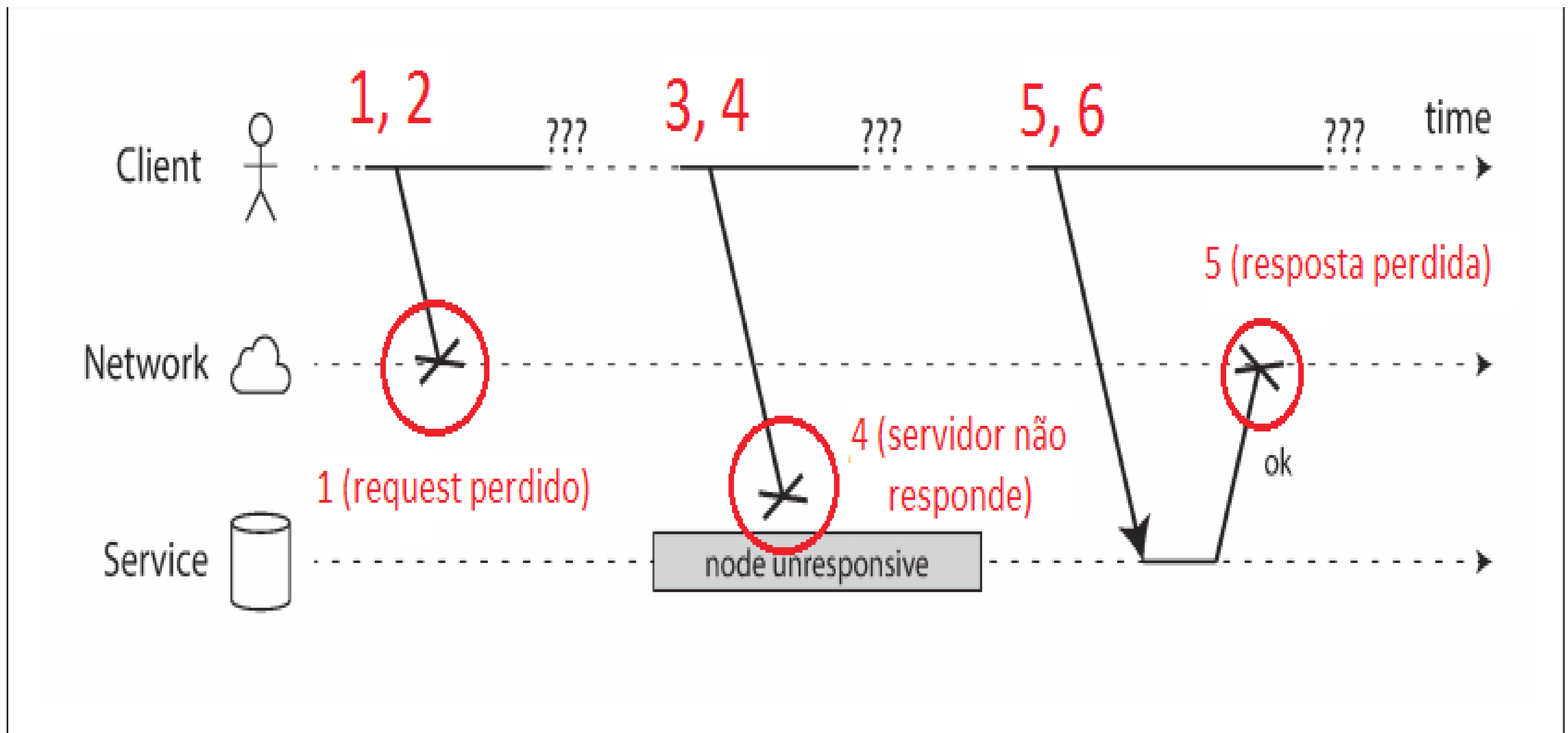
  4) Nodo remoto temporariamente parou de responder (pausa no nodo remoto)

  5) Resposta foi perdida (switch da rede está desconfigurado).

  6) Resposta aguardando na fila (máquina sobrecarregada).

Redes não confiáveis (Unreliable)

- Exemplo de possíveis erros :



Redes não confiáveis (Unreliable)

- Única informação é se a resposta chegou ou não.
- Sistema não sabe o que aconteceu durante a transmissão (nodo receptor falhou, rede sobrecarregada).
- *Timeout*: Após um período de tempo o nodo emissor constata que não haverá uma resposta.

Falhas de rede na prática

- Exemplos na prática:
  - Datacenter de porte médio possui 12 faults de redes por mês. 6 faults desconectaram um computador e 6 desconectaram um conjunto de computadores.
  - Adicionar redundância a rede não proporcionou a melhora esperada (falhas humanas).
  - Durante a atualização do Switch da rede pode haver uma reconfiguração, ocasionado atrasos na entrega de pacotes.

Falhas de rede na prática

- Exemplos na prática:
  - → Tubarões podem comer os cabos submarinos.
  - → Perca de pacotes dentro dos limites de tempo e aceita os que estão fora do limite.
- Software tem que ser capaz de lidar com as faults.

Detecção de falhas

- Sistemas podem detectar automaticamente falhas:
  - ➔ Um nodo para de enviar mensagens para outro nodo que está *morto* (falho).
  - ➔ Banco de dados distribuído com replicação de líder único. Caso o líder falhe outro nodo é escolhido como líder.

Detecção de falhas

- Exemplos de feedback na detecção de falhas:
  - → Se na máquina em que o nodo está executando é detectado que o processo está falho (não responde), então a conexão com o nodo é cortada e este é declarado falho.
  - → Se um processo de um node falha o seu sistema operacional avisa sobre a falha e outro nodo assume a função (antes do timeout).
  - → Se é possível acessar o switch da rede também é possível detectar links falhos em nível de hardware.

Detecção de falhas

- Exemplos de feedback na detecção de falhas:
  - ➔ Se o roteador verifica um endereço IP inválido, este pode avisar a falha.

- Durante o *timeout* o nodo espera a resposta, caso não recebe nenhuma mensagem do outro nodo, este é declarado *morto*.

Timeouts e Atrasos Sem Limite (Unbounded)

- Tamanho do *timeout*:
  - *Timeout* grande: tempo para descobrir que o nodo está morto será maior (Usuário tem que esperar ou recebe uma mensagem de erro).
  - *Timeout* pequeno: um nodo sem falha pode ser declarado falho, porém estava temporariamente parado (sobrecarga na rede ou no nodo).

Timeouts e Atrasos Sem Limite (Unbounded)

- Problemas:

  ➔ Um nodo é declarado prematuramente falho e outro nodo irá assumir suas tarefas (por exemplo, enviar um e-mail).

  ➔ O nodo declarado morto recupera e continuar a realizar suas tarefas causando duplicação (envio do mesmo e-mail 2 vezes).

Timeouts e Atrasos Sem Limite (Unbounded)

- Problemas:
  - → Um nodo declarado morto sobrecarrega a rede e no nodo.
  - → A tarefa passada para outro nodo.
  - → Nodo fica sobrecarregado e será declarado morto
  - → Irá sobrecarregar outro nodo (desencadeamento).
  - → No final todos os nodos podem ser declarados mortos e o sistema para de funcionar.

Timeouts e Atrasos Sem Limite (Unbounded)

- Exemplo de cálculo de um timeout:
  - Tempo máximo para enviar uma mensagem é *d* (ou é enviada ou é perdida).
  - Um nodo sem-falha responde em, no máximo, tempo *r*.
  - Transmissão de uma mensagem e sua confirmação de recebimento demora, no máximo, o tempo $2d + r$ (*timeout*).
  - Resposta fora deste limite sabe-se que a rede ou o nodo não estão funcionando.

Timeouts e Atrasos Sem Limite (Unbounded)

- Sistemas assíncronos *não tem limites conhecidos* (a mensagem será enviada o mais rápido possível porem não há um limite de tempo).
- A maioria dos servidores não garante um limite de tempo no tratamento de um request.

Congestionamento de Rede e Enfileiramento

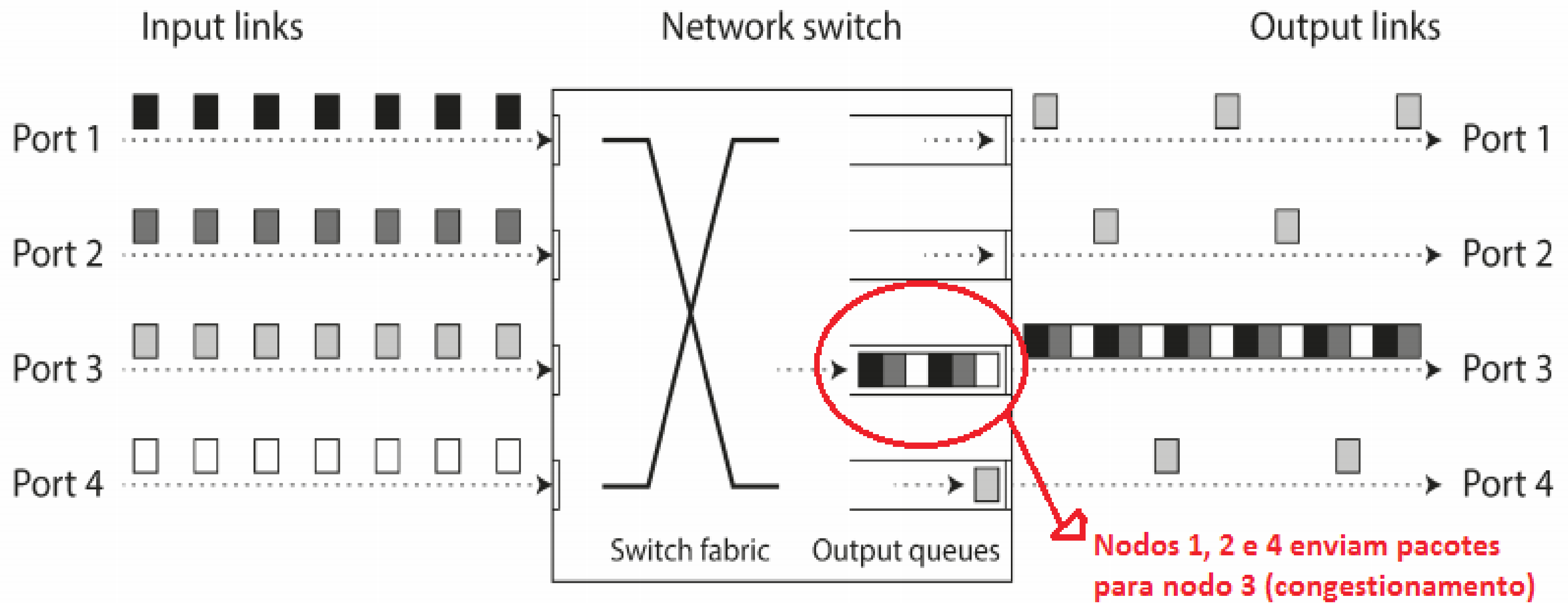
- Em uma rede de computadores o enfileiramento de pacotes acaba atrasando os seus envios. Exemplos:
  - Pacotes são enfileirados pelo switch no link destino um por um (na ordem em que chegaram).
    - Em um link cheio um pacote tem que esperar para ser transmitido (*network congestion*).
    - Pacotes perdidos e retransmitidos.
  - Pacote transmitido para uma máquina utilizando todos os CPUs é posto na fila de processamento pelo SO.

Congestionamento de Rede e Enfileiramento

- Em uma rede de computadores o enfileiramento de pacotes acaba atrasando os seus envios. Exemplos:
  - Em um ambiente virtualizado o sistema operacional cede uso do CPU por uma máquina virtual.
    - Máquina virtual não pode acessar os dados da rede.
    - Dados colocados na fila aumentando assim o atraso na rede.
  - TCP usa o controle de fluxo (*flow control*): limita a taxa de mensagens enviadas diminuindo a carga na rede ou no link destino. Fila de espera no nodo que envio a mensagem.

Congestionamento de Rede e Enfileiramento

- Exemplo congestionamento:



Congestionamento de Rede e Enfileiramento

- Sobre o TCP:
  - Considera um pacote como perdido se ele não é reconhecido durante o *timeout* (tempo de envio e recebimento da mensagem).
  - Caso um pacote seja perdido ele é automaticamente retransmitido.
  - A aplicação não sabe da perda do pacote nem da retransmissão, porém sabe se o *timeout* foi atingido e a mensagem retransmitida.

Congestionamento de Rede e Enfileiramento

- Sistemas mais preparados consegue absorver melhor a carga extra.
- Sistemas muito utilizados tendem a criar filas mais facilmente.
- Nuvens públicas e datacenters com muito usuários compartilham recursos.
- Caso um vizinho próximo consuma muitos recursos (mapReduce) pode ocorrer atrasos na rede (*noisy neighbor*).

Congestionamento de Rede e Enfileiramento

- Como não há controle no uso de recursos por parte dos usuários o *timeout* é calculado experimentalmente (um vizinho barulhento pode colocar atrasos na rede).
- Calcula o tempo de envio e resposta (*round-trip times*) de uma mensagem durante um período em diversas máquinas.
- Assim o *timeout* mantém o equilibrio entre um timeout precoce (devido ao vizinho barulhento) e um bom detector de falhas.

Congestionamento de Rede e Enfileiramento

- Como não há controle no uso de recursos por parte dos usuários o *timeout* é calculado experimentalmente (um vizinho barulhento pode colocar atrasos na rede).
- Calcula o tempo de envio e resposta (*round-trip times*) de uma mensagem durante um período em diversas máquinas.
- Assim o *timeout* mantém o equilibrio entre um timeout precoce (devido ao vizinho barulhento) e um bom detector de falhas.

TCP vs. UDP

- Confiabilidade (*reliability*) vs. variabilidade (*variability*).
- UDP valoriza mais a constância de envio do que a confiabilidade (Fluxo constante de dados).
- O protocolo UDP é utilizado por aplicativos sensíveis a latência da rede (videoconferências e streams pela internet) .
- O protocolo UDP não utiliza o controle de fluxo e também não retransmite pacotes perdidos (atrasam a rede).

TCP vs. UDP

- Em uma ligação voice over IP (voIP) como o Skype não há tempo para retransmitir um pacote antes do próximo pacote ser transmitido (pode criar ruído).
- Pacote é substituído por um pacote vazio.
- Pequeno silêncio durante a ligação.

TCP vs. UDP

- Em uma ligação voice over IP (voIP) como o Skype não há tempo para retransmitir um pacote antes do próximo pacote ser transmitido (pode criar ruído).
- Pacote é substituído por um pacote vazio.
- Pequeno silêncio durante a ligação.

Síncrono vs. Assíncrono

- Conseguir impor limites nos atrasos e reduzir a perda de pacotes em nível de hardware.
- Exemplo: linhas telefônicas são mais confiáveis.
  - Atrasos e perda de pacotes são raros.
  - Requer uma baixa latência constante
  - *Bandwidth* (largura da banda, taxa máxima de dados transferidos) suficiente para executar a ligação.

Síncrono vs. Assíncrono

- Em uma ligação e primeiro estabelecido um *circuito* (uma quantidade fixa e garantida de banda é alocada).
- Exemplo: redes ISDN usam 4000 frames por segundo, 1 frame a cada 250 microsegundos e 16 bits de banda são alocados para cada frame (pra cada usuário).
- Então durante a ligação é garantido que 16 bits de áudio serão transferidos a cada 250 microsegundos.

Síncrono vs. Assíncrono

- A rede telefônica é síncrona.
- Uma rede *síncrona* *tem os limites de tempo conhecidos* (bounded delays).
- No exemplo é garantido uma banda máxima (limite) de 16 bits durante todo o tempo, evitando enfileiramento (queueing).
- Consequentemente a latência máxima também e garantida pois não há atrasos e nem perda de pacotes.
- Uma rede *assíncrona* *não tem limite de tempo conhecido*, tanto para o envio de mensagem quanto para a execução das tarefas.

Atrasos de rede Previsíveis

- Rede telefônica utiliza uma banda máxima fixa para transferências
- TCP utiliza uma banda variável (envia maior quantidade de dados possíveis) para tentar enviar no menor tempo.
- Datacenters utilizam o *packet-switch protocol* (Máximo de banda disponível para alcançar o menor tempo de envio). Ao invés do *circuit-switch protocol* (banda de dados fixa).

Atrasos de rede Previsíveis

- Datacenters usa packet-switch, mais apropriado para o burst traffic (necessitam que os dados sejam enviados o mais rápido possível).

- Circuit-switch os dados podem utilizar uma banda com limite conhecido, inadequado para burst (grande quantidade de dados em curto tempo).

Atrasos de rede Previsíveis

- Datacenter com circuit-switch: tem que escolher quanto de banda alocar:
  - Bandwidth pequena: demora mais para ser entregue e uma parte da banda (aquela que não foi alocada) será desperdiçada.
  - Bandwidth grande: Circuit não poderá ser construído pois não garante a bandwidth durante o tempo de envio.

Latência e utilização de recursos

- Exemplo: uma conexão entre dois switch de telefone pode conter até 10,000 ligações simultaneamente.
  - Utiliza circuit-switch.
  - se ocorre apenas uma ligação ou 10,000 ligações simultâneas é alocada sempre 10,000 ligações.
  - Esse recurso foi alocado de maneira *estática* (mesma quantidade de banda sempre).

Latência e utilização de recursos

- A internet utiliza uma alocação *dinâmica* (quantidade de banda alocada é variável):
  - Os nodos tentam enviar seus pacotes o mais rápido possível
  - Switch é quem decide qual será enviado.
  - Enfileiramento para envio de pacotes.
  - Quantidade de espaço usado é otimizado (maior quantidade de banda possível para transmissão).

Latência e utilização de recursos

- Garantia da latência é conseguida através da alocação estática de recursos porém é mais caro.
- Para uma alocação dinâmica podem ocorrer atrasos variáveis porém é mais barato de implementar.
- Uma balança entre custo e atraso.

Relógios Não Confiáveis (Unreliable)

- Relógios e tempo são importantes para:
  - *Atingiu o Timeout.*
  - Tempo de resposta com 99 percentile.
  - Quantidade de tarefas executadas em um período de tempo.
  - Tempo gasto no site.
  - Data publicação do artigo.
  - Data de envio do lembrete.
  - Qual o tempo limite para expirar.
  - Timestamp da mensagem.

Relógios Não Confiáveis (Unreliable)

- 1 a 4 são *durações* (medem a quantidade de tempo entre o pedido e a resposta).
- De 5 a 8 são *pontos no tempo* (eventos que ocorrem em um certo tempo e data).
- Em sistemas distribuídos o tempo entre o envio de uma mensagem e o recebimento pode variar entre as máquinas, dificultando assim saber qual enviou primeiro.
- Cada máquina possui um relógio local.

Relógios Não Confiáveis (Unreliable)

- Sincronização por NTP (*Network Time Protocol*).
- Sincroniza um relógio local com base no tempo dado por um grupo de servidores.
- Esses servidores sincronizam com um medidor de tempo mais preciso como o GPS.

Monotonic vs. Time-of-Day

- *Time-of-Day:* como o nome diz mede a data e hora.
  - Esses valores são obtidos a partir do cálculo entre um tempo pré-determinado até a hora e data atuas.
  - Como exemplo, em Java e Linux exitem funções que calculam o tempo desde a meia-noite do dia 1 de Janeiro de 1970 até a data atual.
  - Desse resultado deduz a data e hora.

Monotonic vs. Time-of-Day

- *Time-of-Day:* como o nome diz mede a data e hora.
  - Sincronizados com o NTP,
  - Em teoria os relógios locais tem o mesmo valor.
  - Problemas se um relógio local está longe do servidor NTP: relógio ter que reiniciar parecendo que voltou no tempo.
  - Esses pulos de tempo para trás, além dos *leap seconds* atrapalham a sincronização.
  - *Leap seconds*: evento que ocorre no final de cada ano no qual é atrasado ou adiantado um segundo)

Monotonic vs. Time-of-Day

- *Monotonic*: São usados para medir intervalo de tempo (timeout).
  - → Esse relógio tem a propriedade de sempre mover para frente no tempo (diferente do Time-of-Day).
  - → A medição acontece calculando a diferença entre o tempo inicial e final definidos.
  - → 2 relógios locais podem possuir valores absolutos diferentes.
  - → Não faz sentido comparar os resultados (Significados diferentes).

Monotonic vs. Time-of-Day

- NTP é usado para adiantar ou retardar a velocidade no qual o relógio está contando o tempo (*slewing*).
- NTP pode retardar ou adiantar mas não causar um pulo temporal.
- Em sistemas distribuídos é utilizado para verificar o timeout:
  - Não precisa estar sincronizado com os outros relógios
  - Não e sensível a pequenas imprecisões na medição (NTP adianta ou atrasa a velocidade da contagem para readequar a medição).

Sincronização de Relógios e Precisão

- Time-of-Day utilizam NTP para sincronizar. Neste processo podem ocorrer vários erros como:
  - relógios físicos de quartz (hardware) não são muito preciso pois podem ocorrer *drifts* (executar mais rápido ou mais lento do que deveriam).
  - Relógio local que difere muito do valor dos servidores NTP pode ser forçado a reinicializar ou o relógio local se recusa a sincronizar. Tempo pode parecer que avançou ou voltou.

Sincronização de Relógios e Precisão

- Time-of-Day utilizam NTP para sincronizar. Neste processo podem ocorrer vários erros como:
  - Um nodo pode ser acidentalmente expulso dos servidores NTP. Pode passar despercebido por um tempo.
  - A sincronização NTP só pode ser tão boa quanto o atraso na rede, logo tem limites na precisão enquanto a rede está congestionada.

Sincronização de Relógios e Precisão

- Time-of-Day utilizam NTP para sincronizar. Neste processo podem ocorrer vários erros como:
  - Servidores NTP podem estar errados ou desconfigurados. Os clientes NTP buscam em vários servidores.
  - Leap seconds podem fazer um minuto durar 59 ou 61 segundos que pode resultar falhas em vários sistemas.
    - Resolve "mentindo". Adaptação gradativamente até o sistema possuir o valor correto (*Smearing*).

Sincronização de Relógios e Precisão

- Time-of-Day utilizam NTP para sincronizar. Neste processo podem ocorrer vários erros como:
  - Em máquinas virtuais com CPU compartilhada, cada máquina virtual é pausada enquanto outra máquinas utiliza a CPU.
  - Não confiar em relógios que rodam em dispositivos que não há controle (mobile). Modificação com dados incorretos.

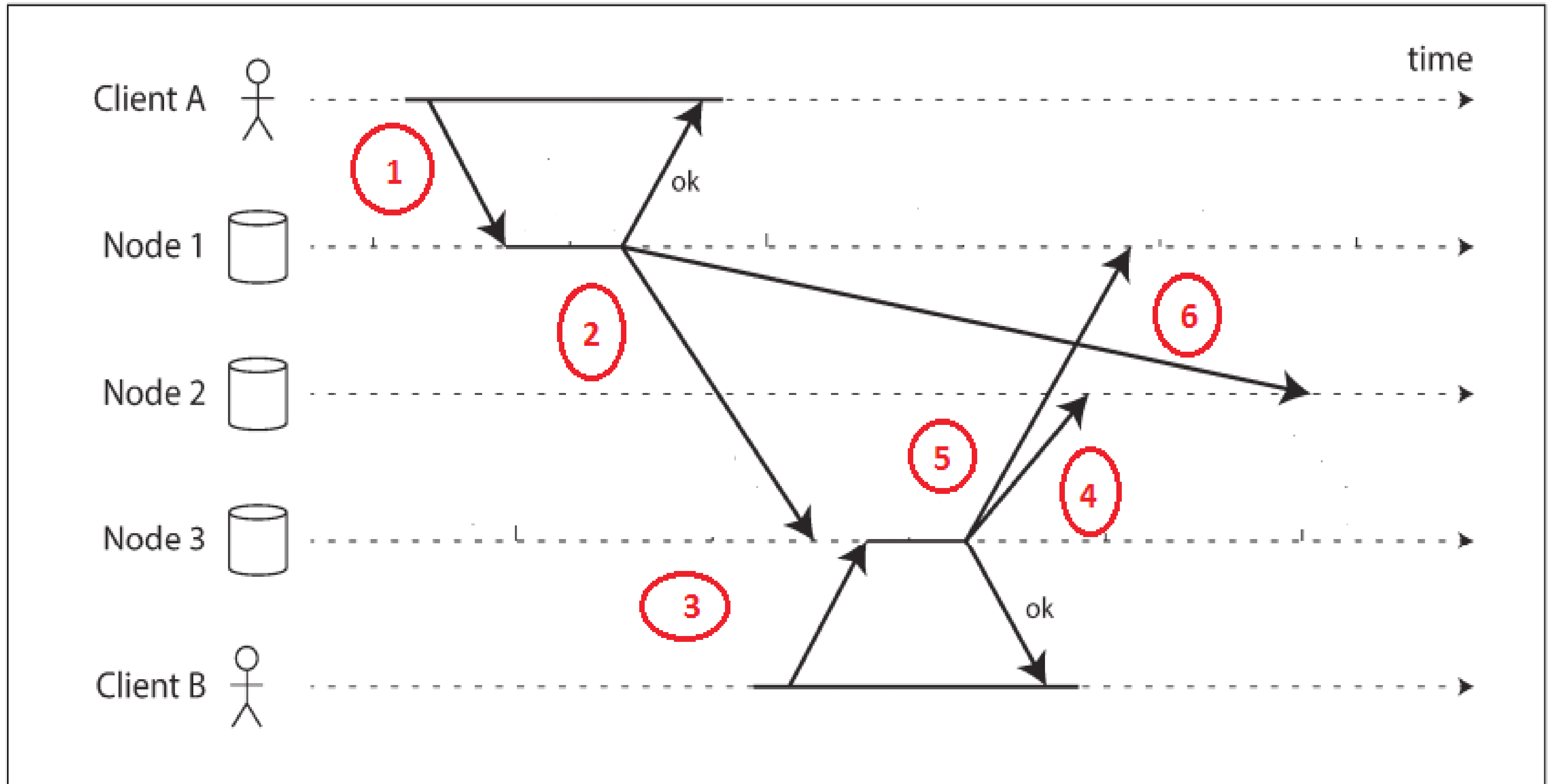
Confiando em Relógios Sincronizados

- Relógios incorretos passam despercebidos.
- Mesmo com defeito no relógio local ou servidor desconfigurado tudo parece funcionar por fora.
- Por dentro relógio poder estar com drift aumentando gradativamente.
- Relógio muito distantes do restante das máquinas deve ser removida do cluster.

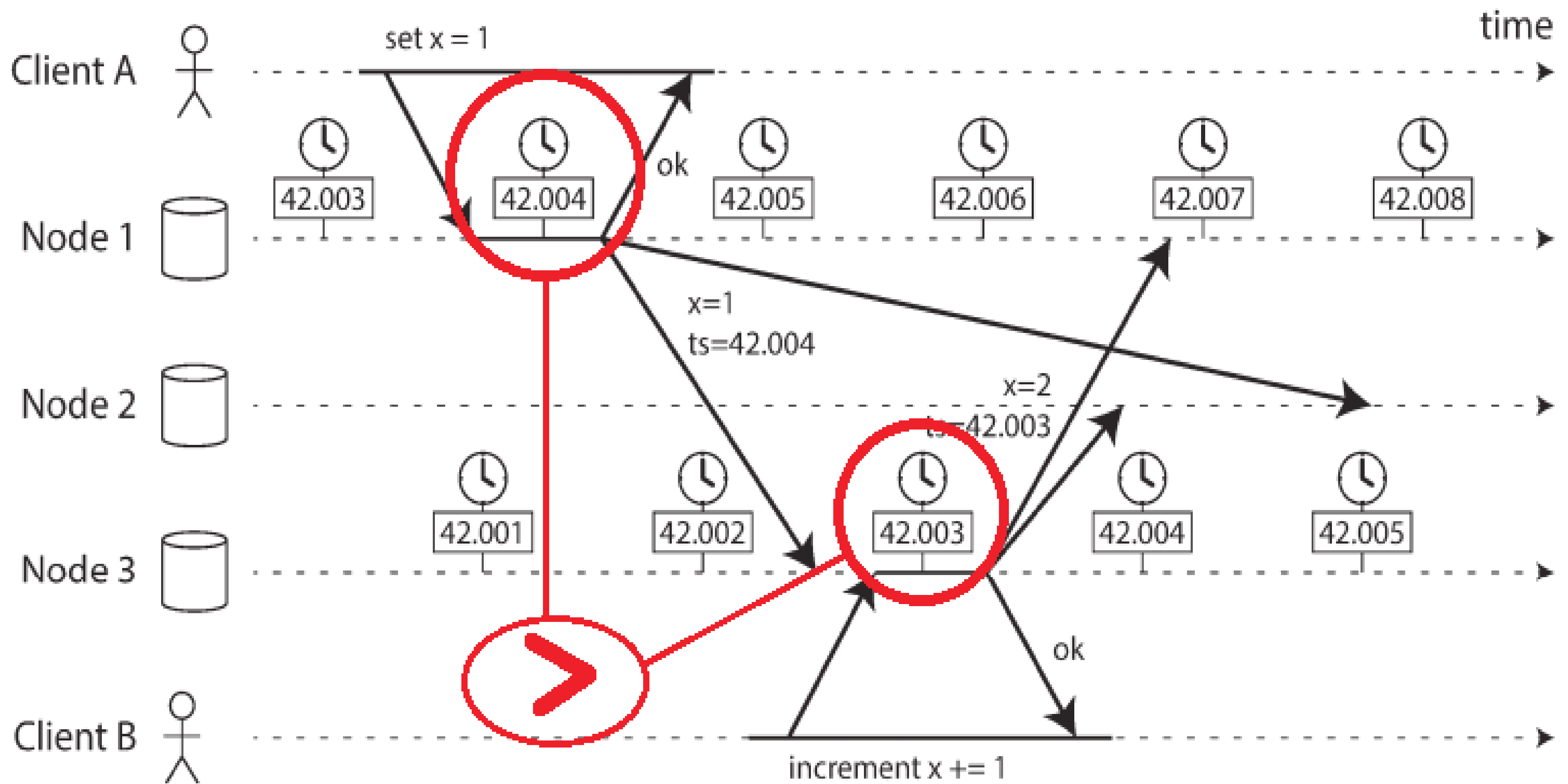
Timestamp para ordenar Eventos

- Problema: ordenar eventos em múltiplos nodos (processos). Se dois clientes quiserem escrever em um mesmo banco de dados, qual deles chegou primeiro? Qual escrita é a mais recente?
- Ordenação por timestamp.
- Se timestamp de A for maior que B, considerando relógios sincronizados, A ocorreu depois de B.

Timestamp para ordenar Eventos



Timestamp para ordenar Eventos



Timestamp para ordenar Eventos

- Quando uma escrita é executada é enviada uma mensagem e transmite junto o timestamp em que a ação foi executada de acordo com o relógio físico Time-of-Day.
- A diferença entre os relógios 1 e 3 é menor que 3ms.
- Usa LWW (*Last Write Wins*).
- No exemplo, os relógios locais causam uma falha de sincronização na escrita.

Timestamp para ordenar Eventos

- Problemas da estratégia LWW:
  - ➔ Escritas nos bancos de dados misteriosamente desaparecem: Um nodo lento pode sobrescreve informações de um nodo mais rápido enquanto o relógio atrasado não sincronizar.
    - ➢ Grande quantidade de informações excluídas sem que nenhum erro seja encaminhado a aplicação.
  - ➔ LWW não consegue distinguir rápidas sequências de escritas de duas escritas realmente concorrentes.

Timestamp para ordenar Eventos

- Problemas da estratégia LWW:
  - E possível que dois nodos gerem escritas com o mesmo timestamp. Para resolver é utilizado um número randômico na ordenação.
- Time-of-Day e Monotonic são *relógios físicos*. Uma alternativa são os *relógios lógicos* que utilizam a ordem relativa dos eventos para sincronizar (qual evento *acontece antes de*).

Relógios e Intervalo de Confiança

- O drift em um relógio local pode ser na faixa dos milisegundos.
- Uma opção melhor é utilizar um intervalo de confiança, que é uma faixa de valores (tempos) possíveis para a leitura de relógio.
- O limite de incerteza (limite inferior e superior de erro) é calculado baseado no recurso utilizado para sincronizar.

Relógios Sincronizados para Snapshot

- *Snapshot Isolation* é útil para banco de dados que suportam pequenas e rápidas transações read-write e para longas transações read-only.
- Em um sistema distribuído é necessário coordenação.
- Uma solução é usar timestamp de relógios Time-of-Day sincronizados como ID de transações.
- Apresenta uma boa sincronização e novas transações (que acontecem depois) apresentam ID maiores.
- Problema continua sendo a precisão do relógio.

Pausas de Processos

- Em um banco de dados com único líder por partição, como saber se o líder foi declarado morto pelos demais?
- Utiliza o *lease (lock com timeout).*
- O nodo que estiver com o *lease* é o líder durante uma certo período.
- Ao fim do período é necessário renovar o *lease* para continuar líder.
- Caso o nodo falhe outro obterá o *lease* e se tornará o líder.

Pausas de Processos

- Algoritmo *lease*:

```
while (true) {
    request = getIncomingRequest();

    //Caso o tempo restante para requisição do lease for menor que 10 seg. Requista.
    if (lease.expiryTimeMillis - System.currentTimeMillis() < 10000) {
        lease = lease.renew();
    }

    if (lease.isValid()) { //Se ainda é o líder
        process(request);
    }
}
```

Pausas de Processos

- 2 problemas com o algoritmo:
  - *Primeiro:* O tempo de expiração do *lease* é calculado por outra máquina e comparado com relógio local (problema de sincronia).
  - *Segundo:* Apenas usando relógios Monotonic. Durante a função em que o relógio local é medido e o processamento da requisição pode ocorrer uma pausa.
  - Se a pausa for grande pode, outro nodo tomará o lease.

Pausas de Processos

- 2 problemas com o algoritmo:
  - O nodo retorna achando que tem o lease e continua a execução de suas tarefas.
  - Pode acontecer de corromper/sobrescrever dados.

Pausas de Processos

- Razões de pausas de processos longas:
  - *Garbage collector* (GC): ocasionalmente para os processos para liberar espaço na memória.
  - Máquinas virtuais podem *suspender* (pausar a execução e salvar o estado) e *retornar* (restaura o conteúdo e continua a execução).
  - Laptops também *suspendem* as ações.
  - Troca de contexto ou troca par outra máquina virtual.

Pausas de Processos

- Razões de pausas de processos longas:
  - Acesso síncrono a disco, a thread tem que ser pausada para as ações lentas de I/O no disco.
  - Na paginação pode ocorrer um page fault, resultando em uma busca de página em disco para a memória.
  - Comandos de pausa no terminal Unix (Ctrl+Z).

Garantia de tempo de resposta

- Alguns sistemas precisam de garantia de tempo.
- Sistemas que dependem vidas (life-critical) como controle de aeronaves e foguetes.
- O sistema tem que responder em um certo limite, caso contrário pode causar *total failure*.

Knowledge, Truth, and Lies

- Um nodo na rede pode apenas obter informações nas mensagens recebidas ou não de outros nodos.
- Podemos prever o comportamento (*modelo de sistema*) e desenvolver o sistema para alcança-los.

A verdade é definida pela Maioria

- Primeira situação: Um nodo recebe, processa e envia uma requisição porem nenhum nodo recebe.
- Segunda situação: O nodo percebe que as mensagens enviadas não estão sendo recebidas e deduz que a rede está falha.
- Terceira situação: O nodo é declarado morto porém estava apenas pausado e volta a funcionar.
- Não pode confiar em apenas um nodo pois este pode falhar e para o sistema.

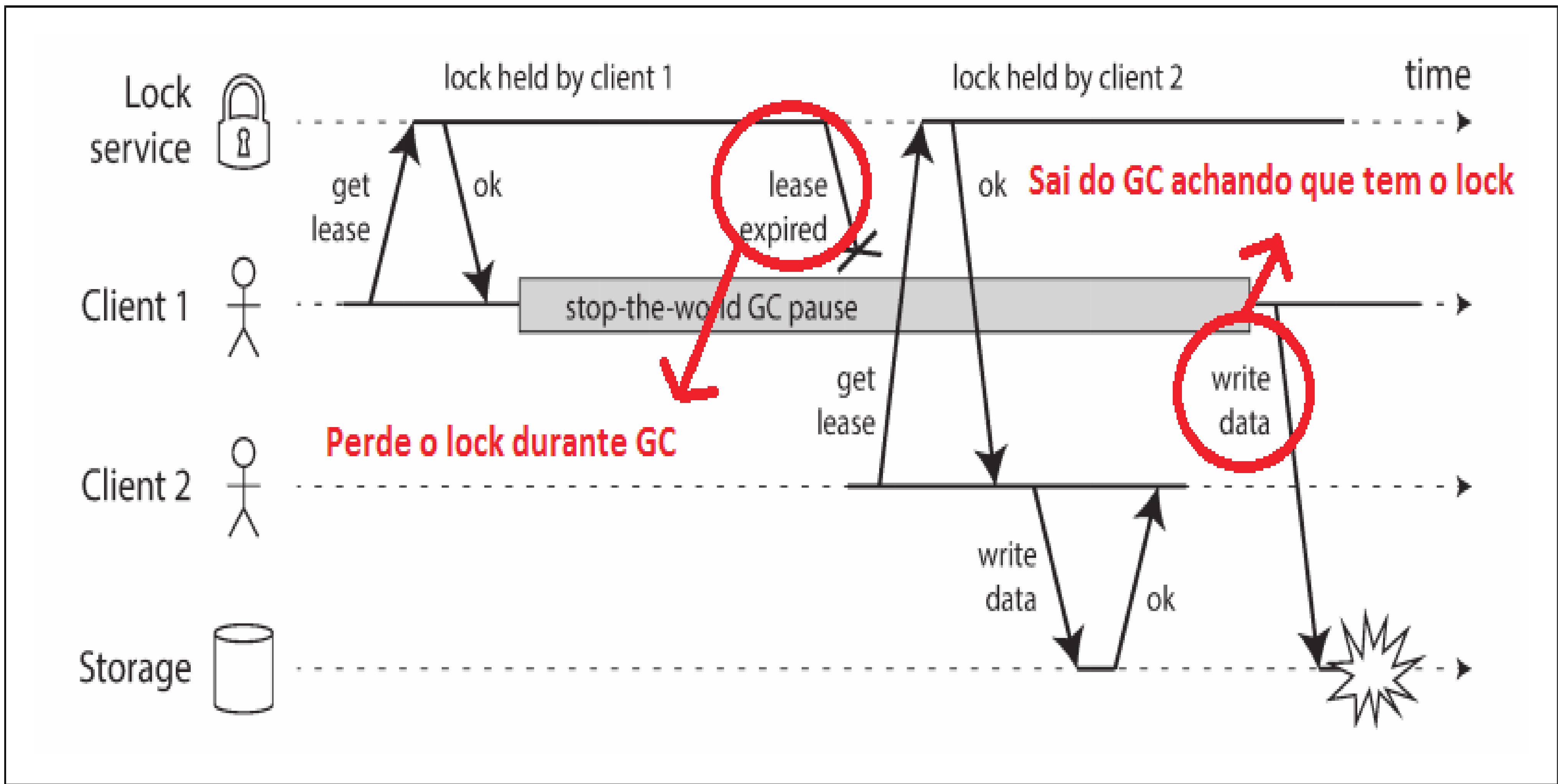
A verdade é definida pela Maioria

- Sistemas distribuídos utilizam um *quorum*.
- Decisões são tomadas a partir de um conjunto de nodos, diminuindo a dependência de um nodo em particular.
- Quorum decide qual nodo está morto ou não.
- Quorum geralmente composto da maioria dos nodos (mais que a metade).

O líder e o Lock

- O sistema requer:
  - Apenas um é o líder da partição de banco de dados.
  - Apenas uma transação ou cliente pode segura o lock (evita escrita concorrente).
  - Um nome (identificador).
- Caso um nodo acredita estar com o lock, este é declarado morto pelo quorum para manter a integridade do sistema.

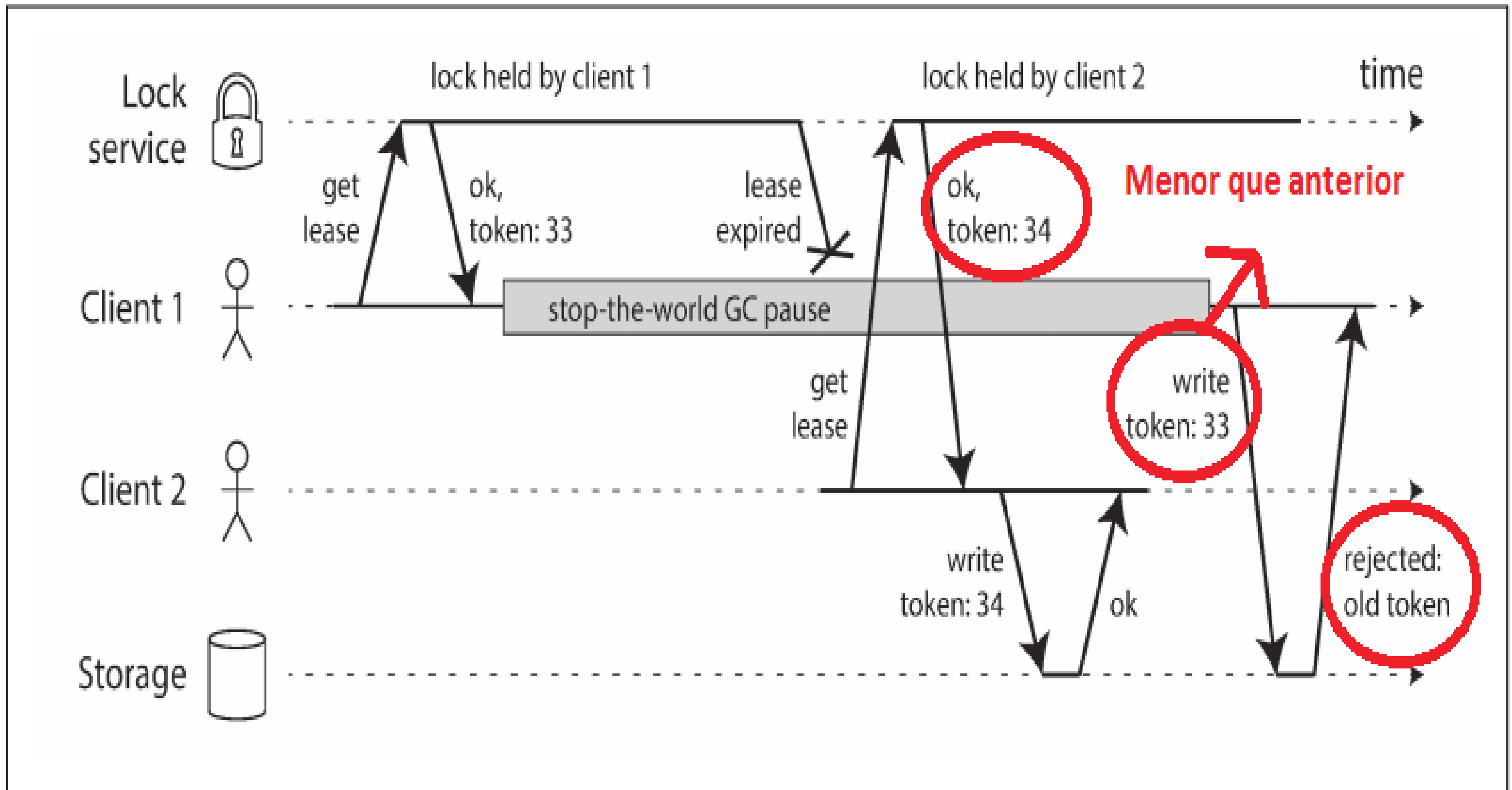
O líder e o Lock



O líder e o Lock

- Resolve usando *Fencing Tokens*:
  - Cada vez que o lock é usando o *fencing token* é incrementado.
  - Em cada escrita verifica o *token*, caso uma escrita com contador maior já foi processado o sistema rejeita a nova escrita.

O líder e o Lock



Falhas Bizantinas

- Falha Bizantina: Componente do sistema se comporta de maneira arbritária (intencional ou não).
- Podem "mentir": enviar falhas e respostas corrompidas.
- *Sistemas tolerantes a falhas bizantinas:* continuam operando mesmo se algum nodo está com defeito ou desobedecendo o protocolo.

Falhas Bizantinas

- São relevantes em:
  - No espaço, onde os dados se corrompem devida a radiação produzindo um comportamento arbitrário (não intencional).
  - Sistema com muitos participantes, onde cada um pode tentar enganar o outro (intencional).
    - Redes peer-to-peer.

Em ambiente WEB é esperado comportamento malicioso pelos clientes

Problema dos Generais Bizantinos

- N generais precisam entrar em um acordo sobre o plano de carro.
- Estão acampados em locais distantes comunicando através de mensageiros
- Porém a traidor(es) entre os generais.
- Os traidores passam mensagens confiáveis para se disfarçar

Modelo de sistemas e Realidade

- Modelo relacionado ao tempo:
  - Síncrono: todos os limites são conhecidos (atrasos, pausas e erros de relógio).
    - Não é um modelo realístico.
  - Parcialmente Síncrono: Se comporta como um modelo síncrono na maioria do tempo. Porém excede os limites algumas vezes.
  - Assíncrono: Nenhum limite é conhecido.
    - Não tem um relógio.

Modelo de sistemas e Realidade

- Modelo relacionados a falhas:
  - Crash-Stop: O componente para de funcionar completamente a partir de um instante. Não voltando mais.
    - Perde o estado interno completamente.
  - Crash-Recovery: O componente para de funcionar completamente porém pode começar a responder depois de algum tempo
    - Mantém o estado interno (armazenamento não volátil).

Modelo de sistemas e Realidade

- Modelo relacionados a falhas:
  - *Bizantina*: O componente se comporta de maneira *arbitrária.* Podendo produzir qualquer resultado (intencional ou não).
    - Pode “mentir”.

Corretude do algoritmo

- Para dizer que um algoritmo é correto podemos definir suas propriedades:
- Para o algoritmo de Fencing Token:
  - Uniqueness: Não é permitido 2 requisições do token.
  - Monotonic Sequence: X aconteceu antes de Y, então token de X é menor que de Y.
  - Availability: Um nodo que pede o token e não crash eventualmente recebe uma resposta

Safety and Liveness

- *Safety*: Nada de ruim acontece.
- Safety (definição precisa): Mesmo que tudo de errado a propriedade é garantida
  - No exemplo, Uniqueness o ruim é ter requisições para o token. A propriedade garante que não há 2 requisições.
- *Liveness*: Algo de bom acontece:
- Liveness (definição precisa): *Eventualmente*, em um ponto futuro, há esperaça de ocorrer algo bom no sistema.
  - No exemplo, Availability é bom a resposta. A propriedade diz que eventualmente chega a resposta.

Perguntas

1) Qual a diferença entre relógios Monotonic e Time-of-Day? Diga dois problemas em utilizar o Time-of-Day na sincronização? Por que as escritas no banco de dados representadas na figura não estão sincronizadas corretamente?

2)Quanto aos modelos de sistemas. Descreva quais são os modelos de falha? Por que podemos dizer que o conjunto de falhas Crash é um subconjuntos das falhas Bizantinas?

Perguntas