Algoritmos e Sistemas Distribuídos

Aula teórica 6

Ano lectivo 2014/2015

Mestrado Integrado em Engenharia Informática

Primeiro teste

- 20 de Outubro, 18:00
- Matéria para o teste: Todas as aulas antes do teste (incluindo a de hoje) excepto Paxos
  - Paxos virá para o segundo teste
- Com consulta → apenas são proibídos os dispositivos com funções de comunicação
  - Portáteis, tablets, smartphones, pombo-correio, etc.
- Aula prática / resolução de exercícios-tipo: hoje
- (Terceiro teste prático fica para depois)

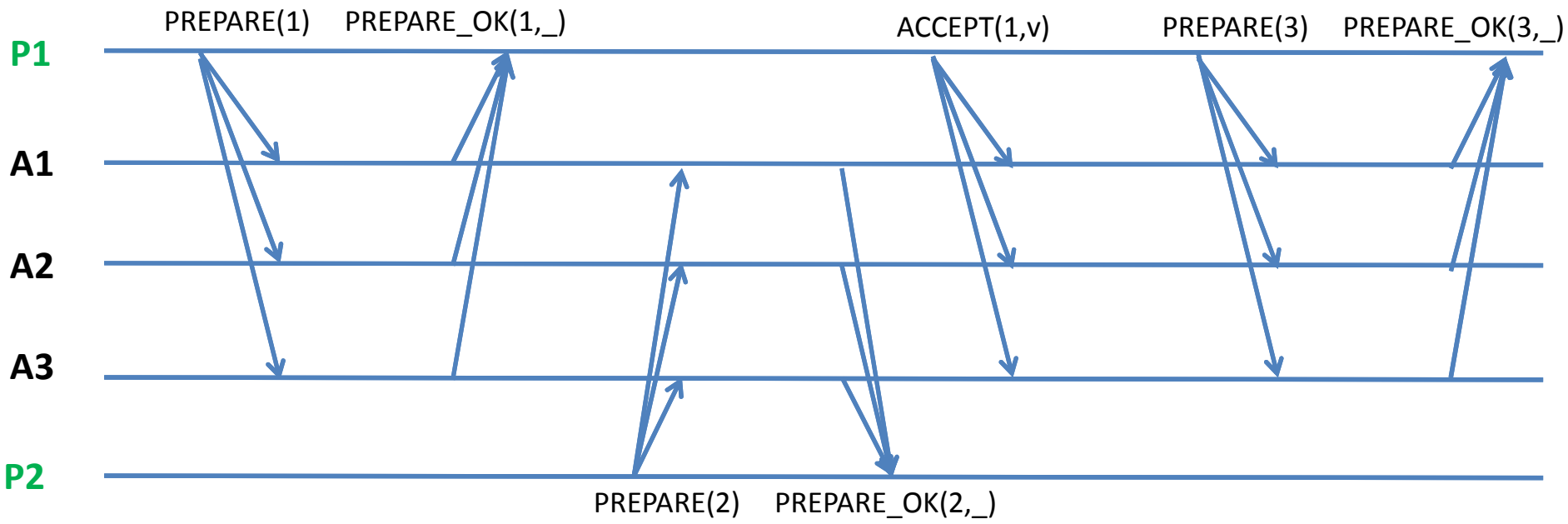
Última aula: Contornar o FLP

- Enfraquecer a propriedade de terminação
  - Apenas garantir terminação em períodos de sincronia: se a rede entregou as mensagens atempadamente durante um certo intervalo
- Algoritmo: Paxos
  - Dois processos correctos não podem decidir valores diferentes
  - Se um processo correcto decide v, então v era a proposta de algum processo

Separação de papéis

- 3 tipos de processos:
  - Proponentes (proposers)
  - Aceitantes (acceptors)
  - “Aprendedores” (learners)
- Na prática podem correr todos na mesma máquina → um processo com as três funções

Paxos: Exemplo

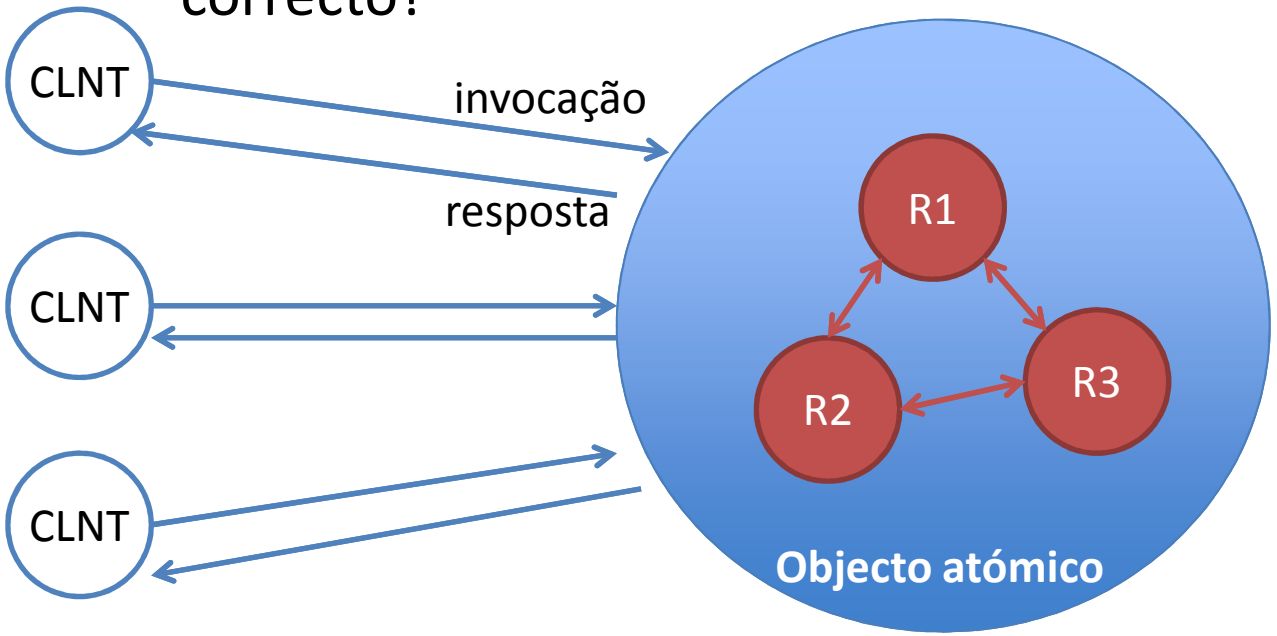


Aula de hoje

- Atomicidade / linearizibilidade
  - Especificação para um sistema replicado
- Replicação de variáveis de leitura/escrita
  - Interface mais simples do que replicação de máquinas de estados
- Modelo de falhas Bizantinas
  - Falhas arbitrárias

Atomicidade

- Contexto: Sistema replicado com interface genérica
  - Ou seja, replicação de máquinas de estados
  - Clientes invocam pedidos, e obtêm respostas
  - Do ponto do vista dos clientes, sistema replicado é uma "caixa negra". Como definir o seu comportamento correcto?



Atomicidade

- Condição de correcção para replicação de máquinas de estados, ou para outros sistemas replicados, que define:
- Comportamento do sistema replicado é igual ao de uma invocação instantânea num objecto não replicado
- Também conhecido como linearizabilidade
- Como definir de forma mais precisa?

Definições

- Variável replicada: (V, v0, invs, resps, f)
  - V: Conj. de valores
  - v0: Valor inicial
  - invs: Conjunto de possíveis pedidos (invocações de operações)
  - resps: Conjunto de possíveis respostas (resultados das operações)
  - f: invs × V → resps × V
- Execução da especificação sequencial: v0, a1, b1, v1, a2, b2, v2, ... (a_i = invocação, b_i = resposta)
  - Se finita, tem de terminar com um valor
  - $(b_i, v_i) = f(a_i, v_{i-1})$, $i > 0$.
- Trace da especificação sequencial: a1, b1, a2, b2, a3, b3, . . . (i.e., só a interface)

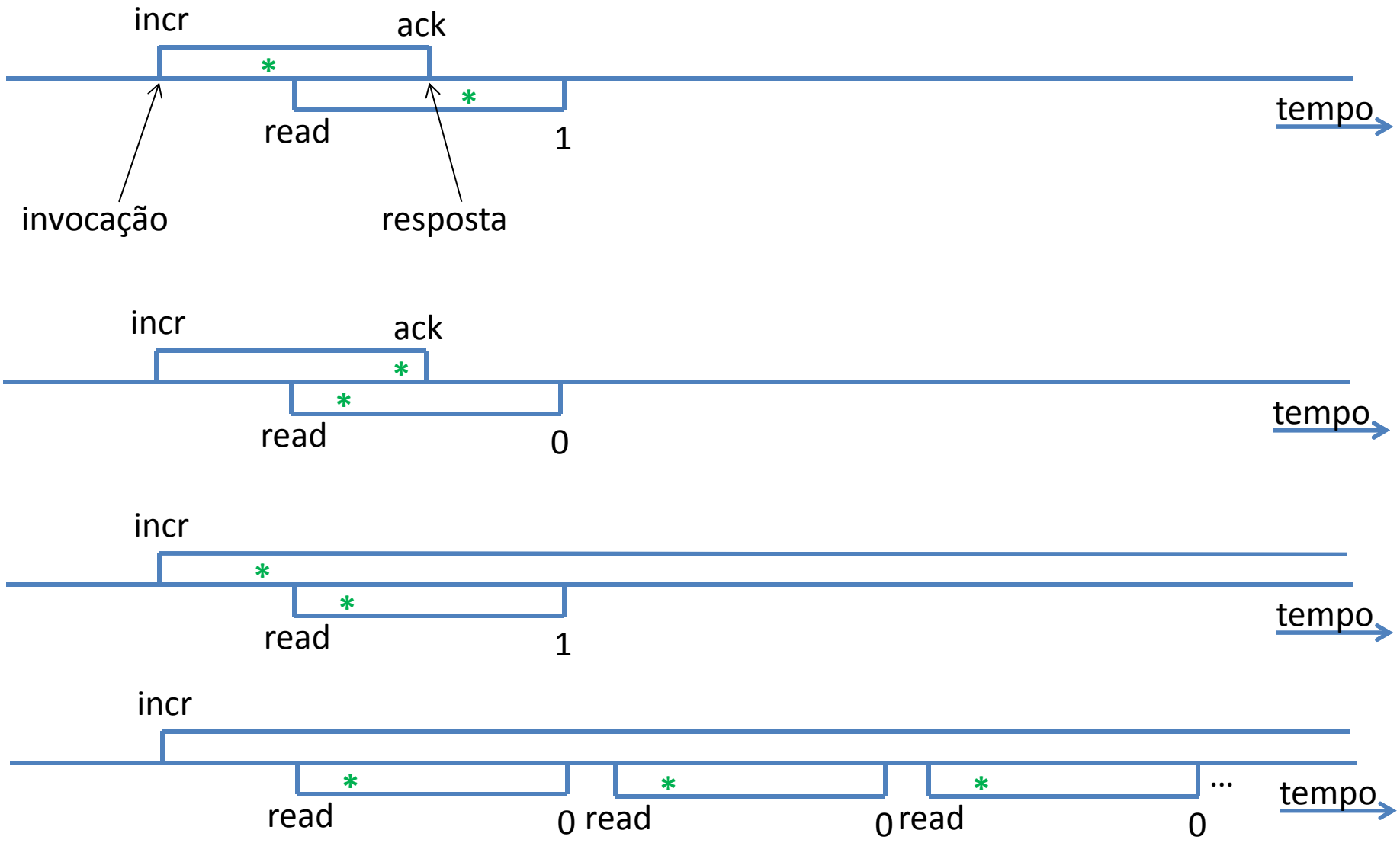
Especificação: Atomicidade / Linearizibilidade

- Um objecto A é atómico se obedecer às seguintes propriedades:
  - Well-formedness - cada cliente alterna invocações com respostas, começando por uma invocação
  - Atomicidade
  - Progresso - pedidos dos clientes são respondidos a certo ponto, “eventually”
- Como especificar a propriedade da atomicidade?

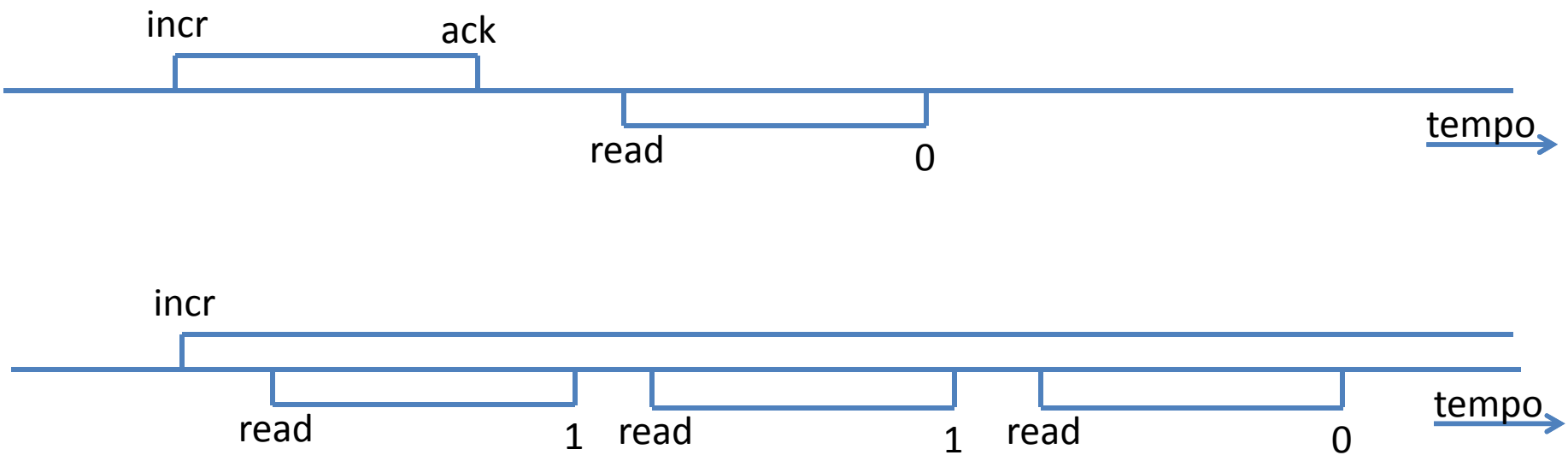
Atomicidade

- Dado um trace, para cada operação existe um ponto de serialização, entre a invocação e a resposta, tal que se movermos a invocação e a resposta para esse ponto, o trace resultante é igual ao trace da variável replicada (isto é, as operações retornam o mesmo valor)
- Se a última operação de um cliente não tem retorno, ponto de serialização pode ou não ser incluído
- Exemplo: variável replicada == contador com operações de leitura e incremento, valor inicial 0

Exemplos de execuções atómicas



Execuções não atómicas



Replicação de variável de leitura/escrita

- Caso particular de replicação de máquinas de estados
- Máquina de estados a replicar suporta apenas duas operações: read, write(v)
  - Write retorna “ack” quando terminar
  - Read retorna um valor anteriormente escrito (ou v0 – valor inicial)
- Permite uma implementação mais simples do que o Paxos

Algoritmo ABD [Attiya, Bar-Noy, Dolev]

- Pressupõe sistema assíncrono, canais fiáveis
- Usa 2f+1 réplicas para tolerar f falhas por crash
  - Ou seja, a propriedade de liveness só é garantida em execuções com menos de f falhas
- Vamos descrever a implementação de uma única variável replicada
  - Pode ser generalizado para uma memória com endereços/identificadores, através de várias instâncias do algoritmo a correr em paralelo

ABD: Estado e algoritmo de escrita

- Estado
  - val_i → valor da variável, inicialmente v0
  - tag_i → par <número de sequência,id> inicialmente <0,0>
- Algoritmo para o cliente c1 efectuar escrita: WRITE(v)
  - Fase 1: Enviar mensagem “read-tag” a todos os processos
  - Esperar por um quórum Q (e.g., maioria) de respostas
  - Seja seqmax = max{sn:<sn,id>∈ Q}
  - Fase 2: Enviar mensagem “write(<seqmax+1,c1>,v)”
  - Esperar por um quórum de acks
  - Retornar “ack”

ABD: Algoritmo da réplica i

- Ao receber read-tag:
  - Retornar tag_i
- Ao receber write(new-tag,new-val):
  - Se new-tag > tag_i então
    - tag_i = new tag
    - val = new-val
  - Retornar ack

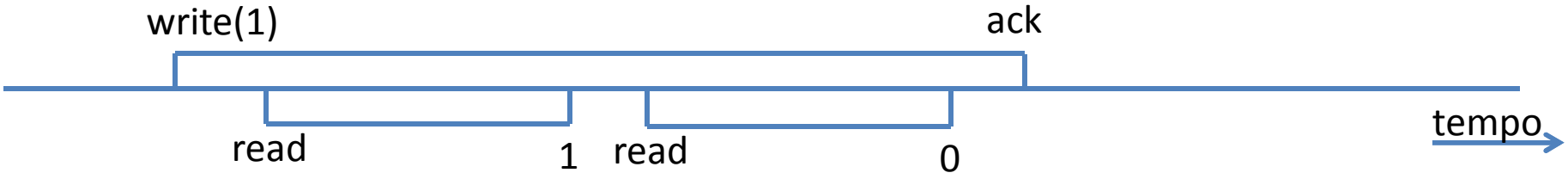
- Ao receber read:
  - Retornar <tag_i,val_i>

ABD: Algoritmo de leitura (tentativa 1)

- Algoritmo para o cliente c1 efectuar leitura: READ()
  - Enviar mensagem “read” a todos os processos
  - Esperar por um quórum Q (e.g., maioria) de respostas
  - Seja <tagmax,valmax>$\in$ Q com a tag mais elevada
  - Retornar valmax
- Será que garante atomicidade?

Tentativa 1 não garante atomicidade

- É possível construir uma execução com o seguinte trace (valor inicial 0):



- Exercício: diagrama temporal desta execução
- Solução: adicionar uma fase (write-back) à leitura

ABD: Algoritmo de leitura

- Algoritmo para o cliente c1 efectuar leitura: READ()
  - Fase 1: Enviar mensagem “read” a todos os processos
  - Esperar por um quórum Q (e.g., maioria) de respostas
  - Seja <tagmax,valmax>$\in$ Q com a tag mais elevada
  - Fase 2: Enviar mensagem “write(tagmax,valmax)”
  - Esperar por um quórum de acks
  - Retornar valmax

ABD: Correcção

- Terminação (se não houverem mais de f falhas) e well-formedness são fáceis de ver
- Atomicidade (ideia geral)
  - Possível encontrar pontos de serialização ordenados pela "tag" associada à segunda fase da leitura e da escrita, com escritas antes das leituras no caso de terem a mesma tag
  - Pela construção do algoritmo, esta serialização comporta-se como uma variável de leitura/escrita (leitura retorna a última escrita, pois esta é a escrita com a mesma tag)
  - Resta demonstrar que este ponto de serialização ocorre entre o início e o fim da operação (ideia geral: demonstrar que pode ser depois do início e pode também ser antes do fim)

Modelo de falhas Bizantino

- Processos que falham podem-se comportar de forma arbitrária
- Modelo engloba causas como bugs de software, corrupção de memória ou de dados em disco
- Também é uma ferramenta de segurança: máquina sob controlo de atacante tem uma falha Bizantina

All bets are off?

- Restrições ao comportamento de um conjunto *limitado* de processos Bizantinos
- Não podem quebrar primitivas criptográficas:
  - Não podem forjar autenticadores (usados para estabelecer canais de comunicação seguros)
  - Não podem provocar colisões de hashes
  - Se assinaturas digitais forem usadas, não podem ser forjadas
- Não podem alterar directamente o estado dos outros processos
- Mas podem fazer “replay” de mensagens antigas autenticadas

Reformulação do consenso

- Condições de correcção:
- C1 [Validade] Se todos os processos **correctos** tiverem $v_i = v$, então v é o único valor do output permitido
- C2 [Acordo] Dois processos **correctos** não podem decidir valores diferentes
- C3 [Terminação] Todos os processos **correctos** têm de decidir a certo ponto (“eventually”)

Resultado fundamental: número mínimo de réplicas para o consenso

- É impossível resolver o problema do consenso com n processos e f falhas Bizantinas, se $n \leq 3f$

- Não vamos demonstrar, por curiosidade devem ver: [N. Lynch] 6.4

Adaptar o ABD para falhas Bizantinas

- Alteração 0: Usar canais de comunicação autenticados
  - Garantir que mensagens provêm do processo certo
- Alteração 1: Mudar as propriedade de intersecção dos quóruns
- Propriedade anterior: quaisquer dois quóruns intersectam-se numa réplica
  - Garante que a leitura “vê” o último valor que foi escrito
- Qual o problema se usarmos a mesma definição?
- A réplica na intersecção pode ser Bizantina

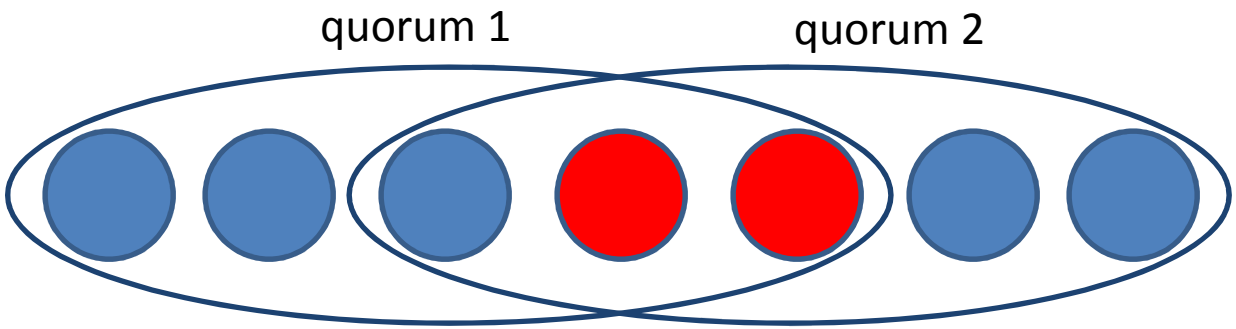
Quóruns Bizantinos

- Quaisquer dois quóruns intersectam-se numa réplica correcta
- Qual a dimensão necessária para os quórums ou o número de réplicas?

Quóruns Bizantinos

- (i) Quóruns não podem ter mais do que n-f réplicas. Porquê?
- Caso contrário pode ser impossível recolher um quórum: réplicas Bizantinas podem nunca responder
- (ii) Quaisquer dois quóruns intersectam-se em pelo menos uma réplica correcta
- (i) Q <= N-f
- (ii) N – (N-Q) – (N-Q) >= f+1

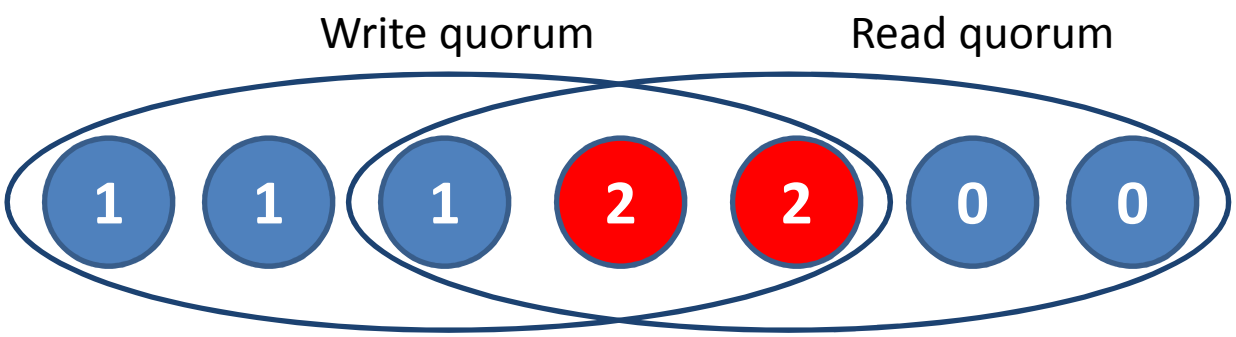
Solução óptima: N = 3f+1, Q=2f+1



Será suficiente?

- Supor que não há escritas a decorrer
- Que categorias de valores são retornados num quórum de leitura?
- Resultados correctos e actuais (quantos no mínimo?)
- Resultados correctos e desactualizados
- Resultados retornados por réplicas Bizantinas (incorrectos)

Exemplo



Solução 1: aumentar Q

- Garantir que os valores correctos e actuais têm mais votos do que os incorrectos
- Para tal, os quóruns têm de se intersectar em 2f+1 réplicas
  - Intersecção contem, no pior caso, f+1 réplicas correctas e f Bizantinas
- Requer n=4f+1, Q=3f+1
- Resultado da leitura é o maior valor retornado por >= f+1 réplicas
- Problema: pode ser impossível encontrar f+1 valores coincidentes. Em que situação?
- Torna-se necessário enfraquecer a especificação

Solução 2: clientes assinam as escritas

- No write, o cliente envia assinatura do par <tag,valor>
- Réplicas armazenam e retornam esta assinatura
- Na leitura, o cliente descarta respostas com assinatura inválida
- Necessário enviar nonce com cada pedido/resposta para evitar “replay attacks”

Estado e algoritmo de escrita

- Estado
  - val_i → valor da variável, inicialmente v0
  - tag_i → par <número de sequência,id> inicialmente <0,0>
  - sig_i → assinatura de <val_i, tag_i >
- Algoritmo para o cliente c1 efectuar escrita: WRITE(new-val)
  - Gerar nonce aleatório
  - Fase 1: Enviar mensagem “read(nonce)” a todos os processos
  - Esperar por um quórum Q de respostas válidas: correctamente autenticadas e com o nonce enviado; descartar respostas inválidas
  - Seja tmax = max{tag:read-reply(<tag,id,sig>)$\in$ Q ^ válida(sig,<tag,id>)}
  - Assinar <<tmax+1,c1>,v> com chave privada de c1, obtem-se a assinatura s
  - Fase 2: Enviar mensagem “write(<tmax+1,c1>,new-val,s,nonce)”
  - Esperar por um quórum de acks válidos (com o nonce enviado)
  - Retornar “ack”

Algoritmo de leitura

- Algoritmo para o cliente c1 efectuar leitura: READ()
  - Gerar nonce aleatório
  - Fase 1: Enviar mensagem “read(nonce)” a todos os processos
  - Esperar por um quórum Q de respostas válidas: correctamente autenticadas e com o nonce enviado; descartar respostas inválidas
  - Seja <tagmax,valmax,sigmax>$\in$ Q com a tag mais elevada
  - Fase 2: Enviar mensagem “write(tagmax,valmax,sigmax,nonce)”
  - Esperar por um quórum de acks válidos; descartar respostas inválidas
  - Retornar valmax

Algoritmo da réplica i

- Ao receber read(nonce):
  - Retornar read-reply(<tag_i,val_i,sig_i,nonce>)
- Ao receber write(new-tag,new-val,sig,nonce):
  - Se válida(sig,<new-val,new-tag>) e new-tag > tag_i então tag_i = new-tag; val_i = new-val; sig_i = sig
  - Retornar ack(nonce)

Final da matéria para o primeiro teste

- Relembrar que Paxos fica para o segundo teste
- Tudo o resto até esta aula (inclusivé) sai no teste
- Próxima aula: revisão da matéria