Algoritmos e Sistemas Distribuídos

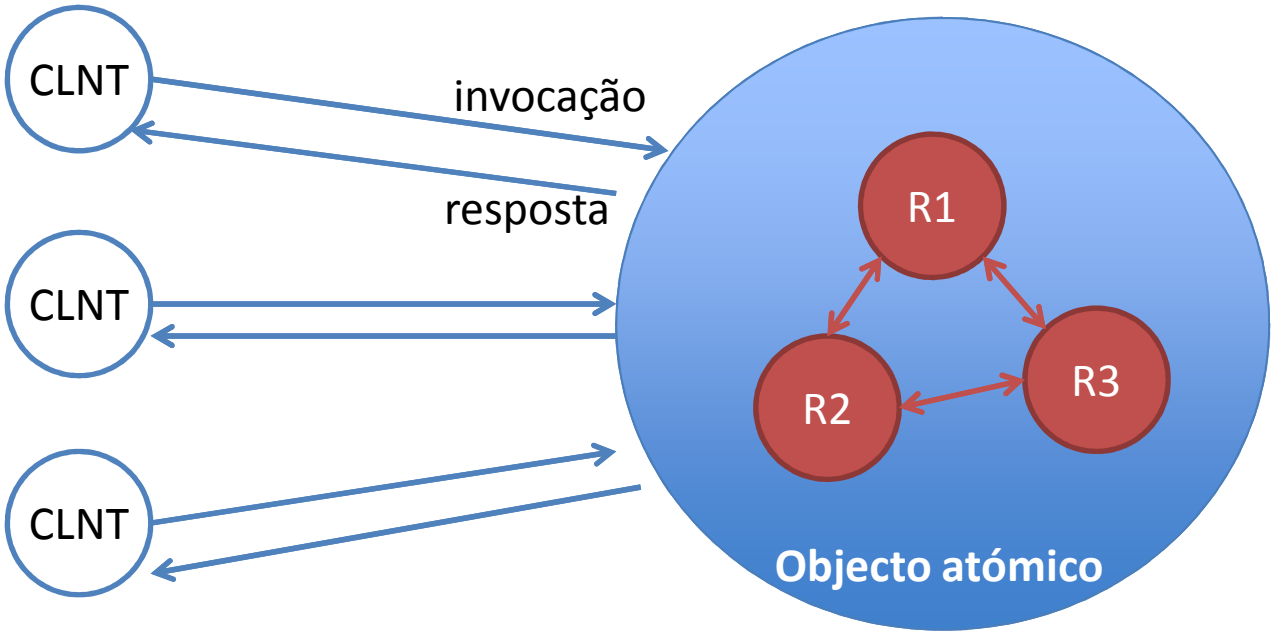
Aula teórica 7

Ano lectivo 2014/2015

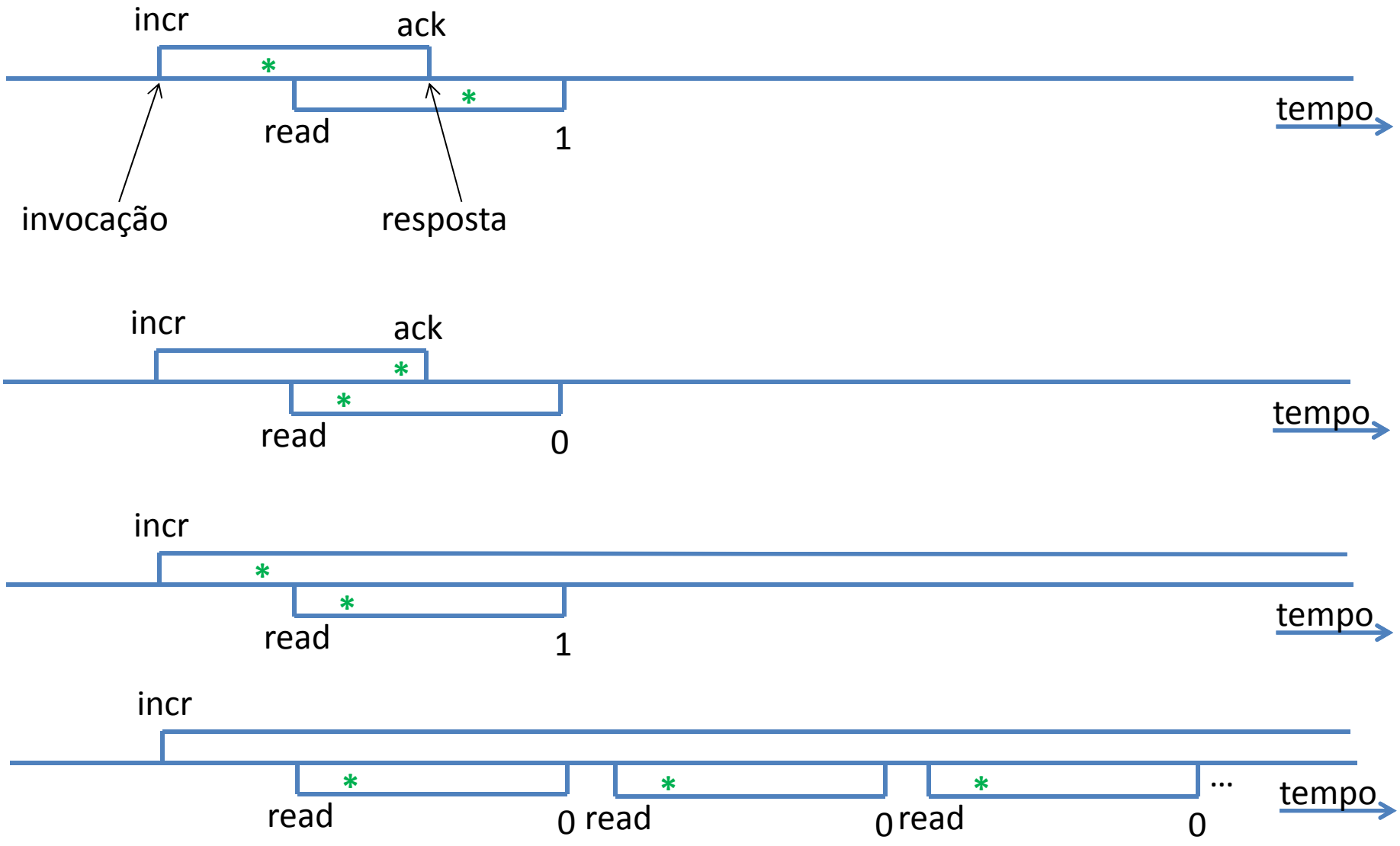
Mestrado Integrado em Engenharia Informática

Última aula: atomicidade

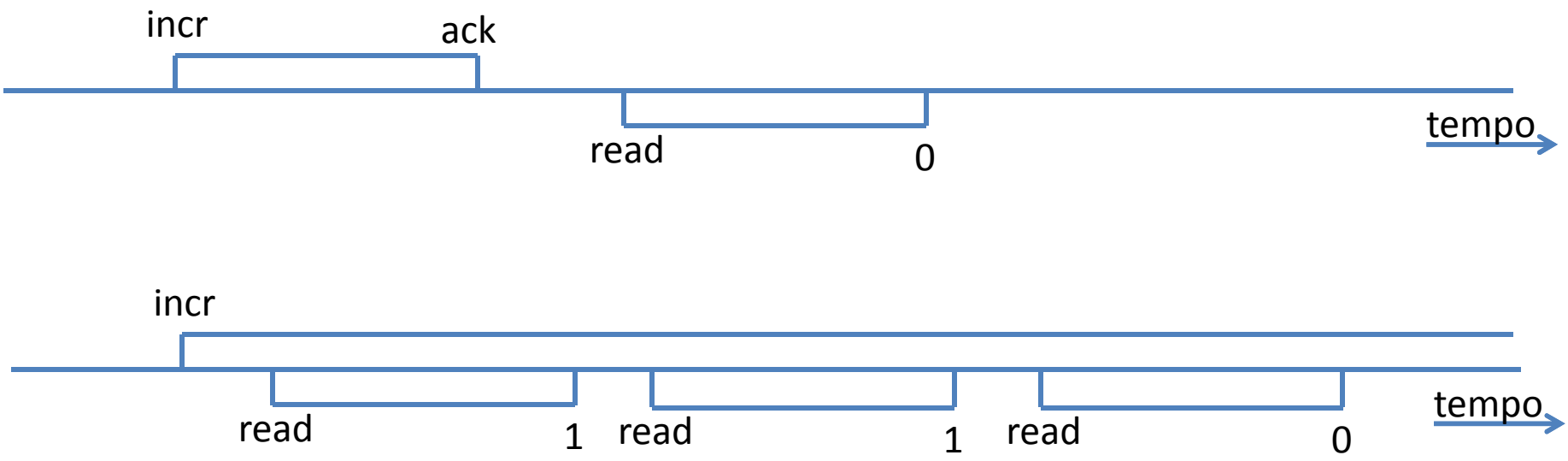
- Comportamento do sistema replicado é igual ao de uma invocação instantânea num objecto não replicado



Exemplos de execuções atómicas



Execuções não atómicas



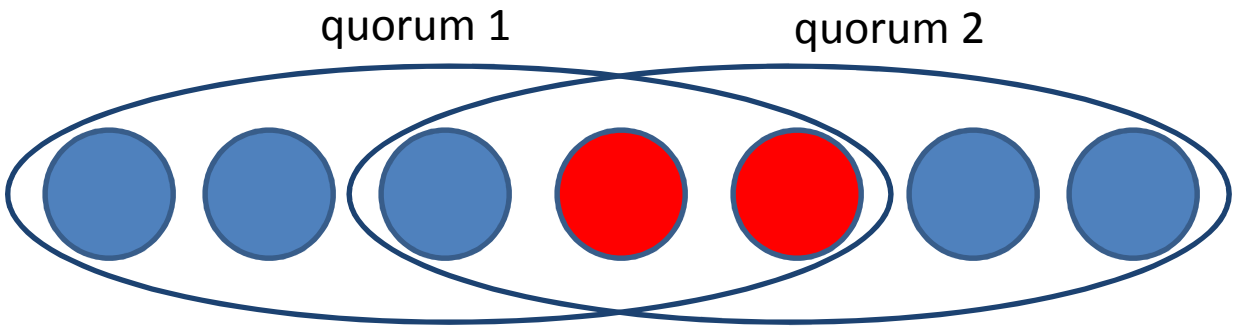
Replicação de variável de leitura/escrita

- 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

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

Quóruns Bizantinos: N = 3f+1, Q=2f+1



Hoje: Consistência fraca

- Consistência forte (nomeadamente atómica) requer contactar um quórum de réplicas para executar operações
  - De forma a garantir que a leitura retorna a última escrita concluída
- Em vários cenários contactar todas as réplicas de um quórum levanta problemas, e.g.:
  - Sistemas geo-replicados (latência inter-continental)
  - Sistemas com clientes/replicas com conectividade intermitente (e.g., smartphones, portáteis)
  - Sistemas com exigências de performance (e.g., SLAs)

Exemplo: Eventual consistency

- Ideia: operações são executadas em apenas uma (ou poucas) réplicas
  - Propagadas às restantes em background
  - Pode ser necessário reconciliar operações que executaram sem estarem cientes uma da outra
- Usado pela Amazon no sistema Dynamo
  - suporte a vários serviços, e.g., carrinho de compras
  - várias réplicas, em centros de dados diferentes, basta contactar um conjunto pequeno para retornar resultado
  - clientes são “proxies” da Amazon que depois fazem rendering
- Usado também em vários sistemas móveis
  - clientes executam operações baseados em cópias locais dos dados
  - em background, sincronizam-se com um servidor central

Qual das eventual consistencies?

- Existem várias definições e implementações
- Vamo-nos focar no sistema Dynamo (Amazon)
- Artigo não define de forma precisa as garantias, por isso o nosso plano vai ser:
  - Definir intuitivamente os requisitos
  - Explicar a implementação
  - Definir mais precisamente consistência eventual

Requisitos (intuitivos) da EC no Dynamo

- Replicação geográfica com elevada disponibilidade e desempenho
- Operações que actualizam o estado podem-se executar sem ver os efeitos uma da outra
  - Por exemplo, dois utilizadores podem comprar o último bilhete para um dado vôo
- "Eventually", todas as operações são propagadas a todas as réplicas
  - A partir desse ponto, ao listar os passageiros fica visível que houve "overbooking"

Requisitos (intuitivos) da EC no Dynamo

- Ponto de partida: interface de leitura/escrita
  - "Key-value store", duas operações:
    - get(key) → returns "value"
    - put(key,value) → returns "ack"
  - Equivale a várias instâncias de variáveis de leitura/escrita
  - Igual à interface do "extent server"
- Dynamo extende esta interface, devido à possibilidade de escritas concorrentes
- Associa put ao get anterior pelo mesmo cliente
  - Da mesma forma que o lock server permite fazer um get e um put de forma atómica
  - Permite determinar que uma operação de put reflecte um conjunto de puts que a antecedem (os puts que foram "vistos" pelo get)

Problema: reconciliar divergência

- Como lidar com escritas que se executaram de forma concorrente?
  - Por exemplo, dois puts que se basearam no mesmo valor do get
  - Notar que pode ser devido a verdadeira concorrência entre as operações (como no yfs) ou devido ao atraso de propagação dos puts (devido ao uso de eventual consistency).
- Várias possibilidades:
  - Política “last writer wins”: arbitra-se uma ordem para os updates concorrentes, perdem-se todos excepto o mais recente
  - Réplicas conhecem uma “merge procedure” que sabe como reconciliar updates concorrentes
  - Expor a divergência aos clientes, aplicação cliente reconcilia e escreve novo valor

Nova interface para a key/value store

- get(key) → returns <list of values,context>
  - Retorna lista de escritas mais recentes que não viram os efeitos uma da outra (de forma a nenhuma se perder)
  - Context descreve o conjunto de escritas que são reflectidas na lista de valores retornada
- put(key,value,context) → returns “ack”
  - Escreve o novo valor “value”
  - Indica o contexto associado ao get mais recente no qual a escrita se baseia

Dynamo: implementação

- Para simplificar vamos supor uma única “key”, e um conjunto fixo de N réplicas
  - Paper inclui detalhes sobre particionamento de chaves e tolerência a falhas
- Quóruns de leitura e escrita: sub-conjuntos de R e W réplicas, com R+W<Q (podem não se intersectar)
- Operações executam-se numa única fase:
  - Cliente contacta uma das réplicas escolhida aleatoriamente
  - Réplica propaga pedido de get/put para N réplicas
  - Espera resposta de R/W réplicas antes de retornar ao cliente
  - No caso do get, retorna o(s) valor(es) mais recente(s), mais do que um no caso de escritas concorrentes
- Problema: como detectar que escritas são concorrentes?

Detecção de concorrência

- Hipótese 1: retornar no contexto a lista de “puts” que constituem toda a história de modificação da “key”
  - Este contexto é enviado aquando do put e armazenado com os dados
- Exemplo: réplicas A,B,C; clientes c1,c2; W=1,R=2
- Cliente c1 faz get da versão inicial em A,B e efectua escrita D1 em A (envia contexto vazio)
- Cliente c2 faz get da versão inicial em B,C e efectua escrita D2 em B (envia contexto vazio)
- Cliente c3 faz get em A,B → detecta que A e B não viram os efeitos um do outro, logo são concorrentes
  - Get deve retornar a lista de valores {D1,D2}, contexto {D1,D2}
- Cliente c3 faz put de D3, envia contexto {D1,D2} (reconciliou)

Problema: escalabilidade

- Esquema anterior funciona mas overheads são muito elevados, como comprimir esta informação?
- Intuição: basta numerar as escritas que cada réplica processa e lembrar-se da mais recente
  - Implica que haja uma única réplica que “processa” a escrita e depois a propaga às restantes réplicas
- Ideia subjacente aos relógios vectoriais
- Cada escrita tem associada um valor de relógio, que consiste num vector com uma entrada por cada réplica
  - Este valor corresponde a um número de sequência da escrita mais recente que cada réplica processou

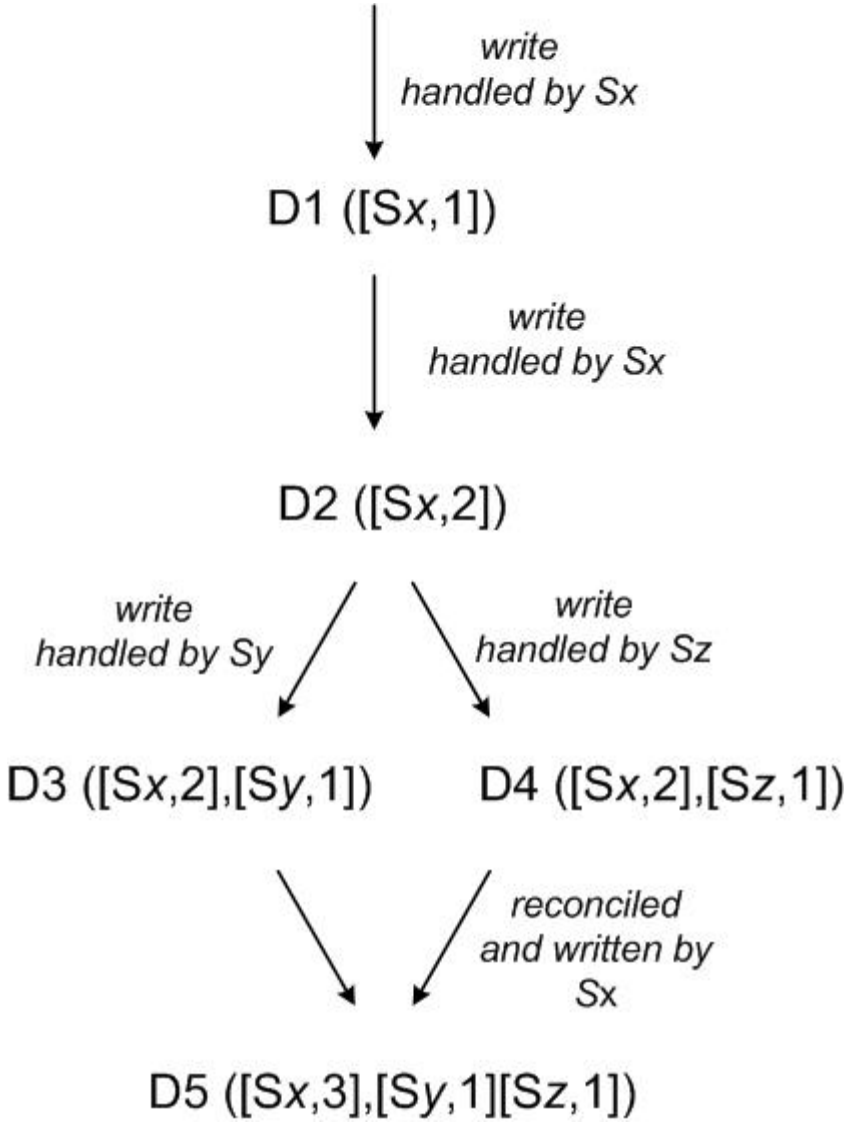
Como detectar concorrência?

- Se duas réplicas avançarem o relógio vectorial de forma independente então as escritas foram concorrentes
- Comparação de relógios vectoriais R1 e R2:
  - R1>=R2 se e só se R1[i]>=R2[i] para todas as entradas do vector
- Se não for o caso que R1>=R2 nem R2>=R1 então R1 e R2 são incomparáveis
  - Indica que réplicas processaram escritas concorrentes

Exemplo: relógios vectoriais

- Cliente c1 faz get da versão inicial [0,0,0] em A,B e efectua escrita D1 em A (envia contexto [0,0,0])
  - Réplica A associa [1,0,0] a D1
- Cliente c1 faz get da versão [1,0,0] em A e [0,0,0] em B e efectua escrita D2 em A (envia contexto [1,0,0])
  - Réplica A associa [2,0,0] a D2
- Cliente c1 faz get da versão [2,0,0] em A e [0,0,0] em B e efectua escrita D3 em B (envia contexto [2,0,0])
  - Réplica B associa [2,1,0] a D3
- Cliente c2 faz get da versão [2,0,0] em A e [0,0,0] em C e efectua escrita D4 em C (envia contexto [2,0,0])
  - Réplica C associa [2,0,1] a D4
- Cliente c3 faz get da versão [2,1,0] em B e [2,0,1] em C → incomparáveis, logo concorrentes
  - Get retorna ambos {D3,D4} valores e contexto [2,1,1]
  - Aplicação do lado do cliente faz o merge, e faz um put de D5 com o contexto [2,1,1], escrita em A
  - Réplica A associa [3,1,1] a D5

Exemplo: relógios vectoriais



Fonte: G. DeCandia et al., "Dynamo: Amazon's Highly Available Key-Value Store", in the Proceedings of the 21st ACM Symposium on Operating Systems Principles, Stevenson, WA, October 2007.

Noção de causalidade

- A implementação do Dynamo garante também a causalidade, que é importante para tornar a experiência do utilizador mais intuitiva. Em particular:
  - Se escrita B viu os efeitos de escrita A, então qualquer operação que veja B também vê A
  - B vê os efeitos de A porque escrita foi precedida de um get que leu A (ou uma versão que sucedeu a A)
  - (No caso do mesmo cliente, causalidade deve também garantir a **monotonicidade** das operações desse cliente: se cliente c emite B depois de A, então B vê os efeitos de A)
- Para distinguir de definições sem garantia de causalidade, vamos usar o termo "consistência eventual+causal"