Algoritmos e Sistemas Distribuídos

Aula teórica 8

Ano lectivo 2014/2015

Mestrado Integrado em Engenharia Informática

Última aula: 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

Problema: reconciliar divergência

- Como lidar com escritas que se executaram de forma concorrente?
- 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

Dynamo: interface

- 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
- Também chamada uma DHT (distributed hash table)

Dynamo: implementação

- Quóruns de leitura e escrita: sub-conjuntos de R e W réplicas, com R+W<N (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) as escritas concorrentes
- Problema: como detectar que escritas são concorrentes?

Como detectar concorrência?

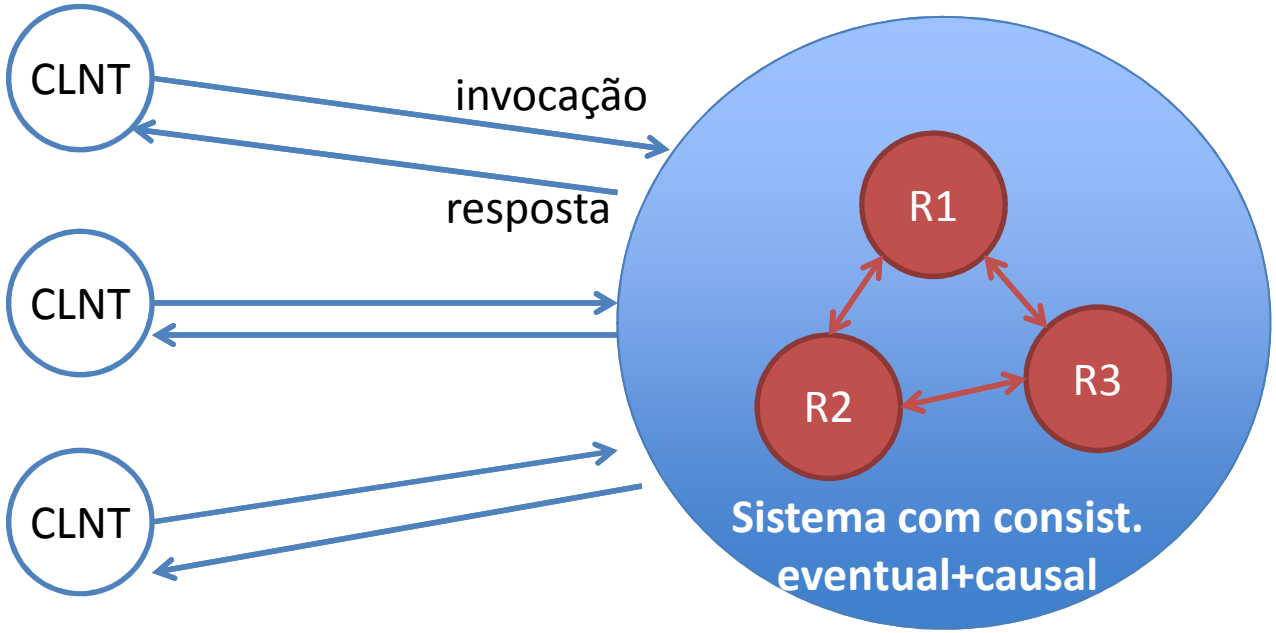
- Relógio vectorial: uma entrada por réplica
  - i-ésima entrada é um contador das escritas aceites pela réplica i
- Se duas réplicas avançarem o relógio vectorial de forma independente então as escritas foram concorrentes
- Comparação de relógios vectoriais R1 e R2:
  - R1>=R2 se e só se R1[i]>=R2[i] 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

Noção de causalidade

- Num sistema com consistência eventual, é importante garantir a causalidade, de forma a 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
  - No caso do mesmo cliente, causalidade deve também incluir a monotonicidade (também chamadas "**garantias de sessão**") das operações desse cliente: se o mesmo cliente emite B depois de A, então B vê os efeitos de A

Generalizar e formalizar a consistência eventual+causal

- Modelo mais genérico:
  - considerar operações genéricas (tal como no state machine replication)
  - ao contrário do Dynamo, não associamos a escrita à leitura anterior (cada operação é autónoma)



Definição de consistência eventual+causal (safety apenas)

- Dada uma execução de um conjunto de operações O por vários clientes do sistema, a execução obedece à consistência eventual+causal se existir uma ordem parcial < sobre O tal que o resultado visto durante a execução da operação x é igual ao resultado de "executar" as operações que precedem x de acordo com a ordem parcial <
  - O que significa "executar" uma ordem parcial?
  - Podemos considerar uma extensão linear da ordem parcial, mas como lidar com operações concorrentes?

Definição de consistência eventual+causal

- A noção de “executar” uma ordem parcial depende da política de resolução de divergência:
  - Last writer wins executa operações concorrentes pela ordem pré-definida
  - Se a divergência for exposta às aplicações é feito um “fork” da execução e os dois estado finais dos dois lados do fork são expostos à operação que faz a junção
  - Se existir uma merge procedure ela é executada, passando como parâmetros: ou os dois estados finais do “fork” antes da junção, ou as operações que foram executadas concorrentemente
  - Se todas as operações forem comutativas então podem-se executar em qualquer extensão linear da ordem parcial

Garantia de causalidade

- Definição anterior garante que se operação B viu os efeitos de operação A, então qualquer operação que veja B também vê A (uma vez que < é uma ordem parcial, e tendo em conta que A<B, logo se B<C então A<C)
- Definição anterior não garante a monotonicidade (garantias de sessão). Estas podem ser adicionadas com a seguinte restrição:
  - Se um cliente emite D antes de E, então a ordem parcial terá que incluir: D<E

Exercício: parte 1

- Considerar uma máquina de estados que implementa um conjunto com uma operação que adiciona valor ao conjunto e retorna o conjunto de valores actual
  - add(int val) → returns set of vals
  - Operações comutativas (merge executa em qualquer extensão linear da ordem parcial)
- Considerar o seguinte trace:

c1: invoca add(5), c1: retorna {5},

c2: invoca add(8), c2: retorna {5,8},

c3: invoca add(7), c3 retorna {7,8}

- Este trace obedece à consistência eventual+causal? Porquê?

Exercício: parte 2

- Considerar o seguinte trace:

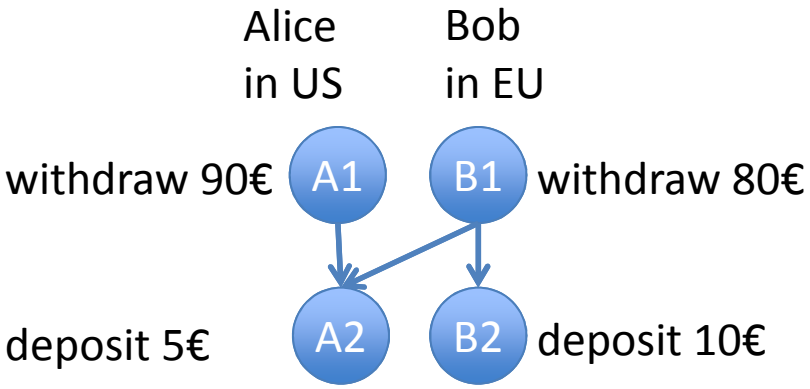
c1: invoca add(5), c1: retorna {5},

c2: invoca add(8), c2: retorna {8},

c3: invoca add(7), c3 retorna {5,7,8}

- Este trace obedece à consistência eventual+causal? Porquê?

Exemplo



- Possíveis ordens de execução
  - Réplica nos US:

    A1, B1, A2, B2
  - Réplica na EU:

    B1, B2, A1, A2

Considerando que há duas réplicas,
uma na EU e outra nos US,
quais as ordens de execução em cada
réplica que levariam a esta ordem parcial?

Supondo um saldo inicial de 100€
Qual o problema com a eventual+causal consistency que este exemplo mostra?

Resultado fundamental

- Teorema CAP: é impossível num sistema replicado ter simultaneamente as seguinte garantias:
  - **C**onsistência forte – nomedamente atomicidade
  - Disponibilidade (**A**vailability) – as operações têm que completar “eventually”*
  - Tolerância a **P**artições – situações em que a rede separa permanentemente o conjunto de réplicas em dois componentes (apenas links dentro de cada componente são fiáveis)

  * Supondo que os links entre os processos são fiáveis

Demonstração do CAP

- Por contradição, existe algoritmo que obedece a C,A,P
- Supor variável de leitura/escrita dois processos p1,p2 em lados distintos da partição (pela propriedade P, todas as mensagens entre p1 e p2 são perdidas)
- Execução e1:
  - Processo p1 executa write(v), pela propriedade A tem de concluir a certo ponto → considerar instante em que conclui, t1
  - Processo p2 inicia read() em t2=t1+k. Pela propriedade C e A tem de concluir e retornar v.
- Execução e2:
  - Processo p1 não escreve
  - Processo p2 inicia read() em t2=t1+k. Pela propriedade C e A tem de concluir e retornar v0.
- Execuções e1 e e2 são indistinguiveis para p2, deveria retornar o mesmo valor no read

Exercício

- Esboce uma implementação que obedeça a:
  - A+P
  - A+C
  - C+P

Timeline consistency (sistema PNUTS da Yahoo!)

- Tenta um meio termo entre atomicidade e eventual consistency
- Por um lado quer tornar algumas operações mais rápidas / maior disponibilidade
- Por outro lado quer evitar a anomalia de levantar demasiado dinheiro da conta

Especificação de timeline consistency

- Separa operações entre leituras e actualizações (updates)
  - Leituras não alteram o estado da réplica
- Actualizações são feitas na mesma ordem em todas as réplicas
  - Existe umaordem total entre as operações
- Leituras retornam valor possivelmente antigo, mas vêem um estado correspondente a um prefixo das actualizações
  - Podem ser serializadas numa posição anterior à atual nessa ordem total

Implementação

- Actualizações são serializadas por uma réplica denominada “master” → atribui número de sequência e propaga updates às restantes
- Várias variantes de leitura:
  - Read-any: lê qualquer versão da réplica mais próxima
  - Read-critical(v): lê um número de sequência superior a v (contacta várias réplicas até encontrar tal versão, se necessário); permite implementar causalidade
  - Read-latest: lê da réplica master