Algoritmos e Sistemas Distribuídos

Aula teórica 9

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

Definição de eventual causal+consistency (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 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?

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

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
- Solução:
  - ordem total das operações (updates têm de ser serializados, por exemplo por uma réplica primária)
  - operações “read-only” são serializadas no passado (podem ler de qualquer réplica, mesmo que desatualizada)

Problema: escalabilidade do Dynamo

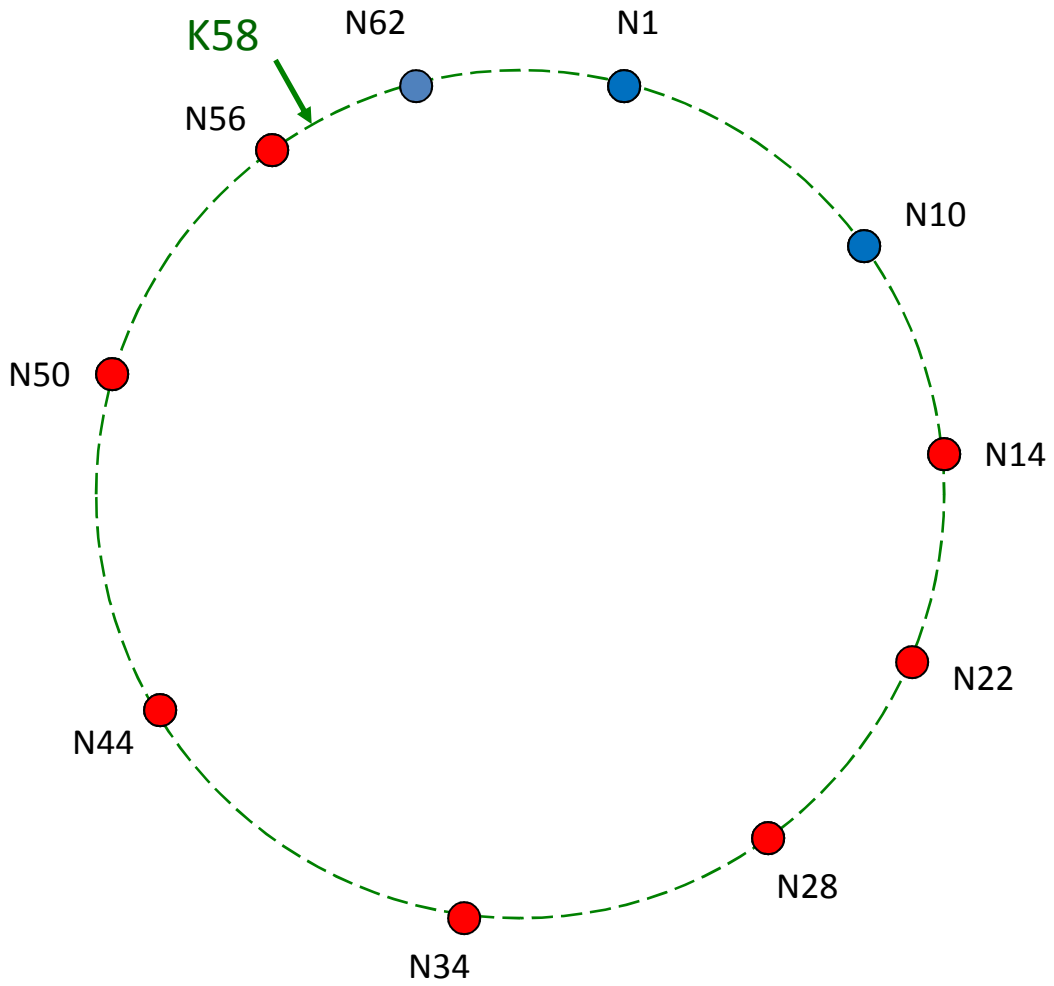
- Quantidade de dados a armazenar na DHT é muito elevada: necessário repartir por vários grupos de réplicas
- Problema: como determinar o conjunto de réplicas para cada par <key,value> ?

Consistent hashing

- Usar algoritmos aleatórios
- Réplica e chaves têm identificadores num mesmo espaço de identificadores
  - Por exemplo, ambos são números com m bits
- Identificadores de réplicas e chaves são atribuídos de forma a estarem distribuídos de forma uniforme nesse espaço
  - Não devem existir colisões entre ids
- Por exemplo, id_r = Hash(IP) / id_c = Hash(chave)

Consistent hashing

- ids estão dispostos num círculo módulo 2^m
- Chave com id k é replicada nas n réplicas que sucedem a k no círculo de ids
- Exemplo (n=3)



Vantagens

- Fácil determinar réplica
  - Basta saber o conjunto de servidores disponíveis
  - Amazon: qualquer servidor pode processar pedido de entrada/saída de outro servidor; pedido é propagado aos restantes nós
- Boa distribuição da carga pelos vários servidores (número de chaves por servidor)
- Movimento de pares <chave,valor> reduzido quando um servidor entra ou sai do sistema

Propriedades do consistent hashing

- Dado um sistema com N nós, K chaves, e supondo que não há replicação (n=1), demonstra-se que, com elevada probabilidade:

1. Cada nó é responsável por no máximo (1+ ε)K/N chaves
2. Quando um nó entra ou sai, o número de chaves que muda de responsável é O(K/N)

- O valor de ε pode ser arbitrariamente reduzido se substituirmos cada servidor por O(log N) nós virtuais com ids diferentes
- Demonstrações: ler livro de “Randomized algorithms”

Desafio

- Como correr uma DHT num ambiente mais adverso do que o centro de dados da Amazon?
- Em particular, e se for difícil de manter informação completa sobre o conjunto de servidores no sistema, e.g.:
  - o número seja muito elevado (e.g., dezenas de milhões)
  - a taxa de entrada e saída seja muito elevada (e.g., tempo de permanência na ordem das horas)
- Alguém conhece ambientes com estas características?

Sistemas peer-to-peer

- 1999: Napster, Freenet, SETI@home
- Propriedades que os definem:
- Descentralizados (ou maioritariamente descentralizados nalguns casos)
- Auto-organizáveis
- Múltiplos domínios administrativos

Objectivos / características dos sistemas p2p

- Facilidade de “deployment”
  - comparado com cliente-servidor
- Escalabilidade
  - Quanto mais participantes mais o serviço escala
- Resiliência a ataques ou falhas
  - Não há ponto central de falha
- Diversidade de recursos
  - SW, HW, geografia, jurisdição, fonte de energia, etc.

Aplicações

- Partilha e distribuição de ficheiros
- Distribuição de conteúdo / streaming
- Chamadas de audio/video
- Computações cooperativas / voluntárias

Redes sobrepostas (overlays)

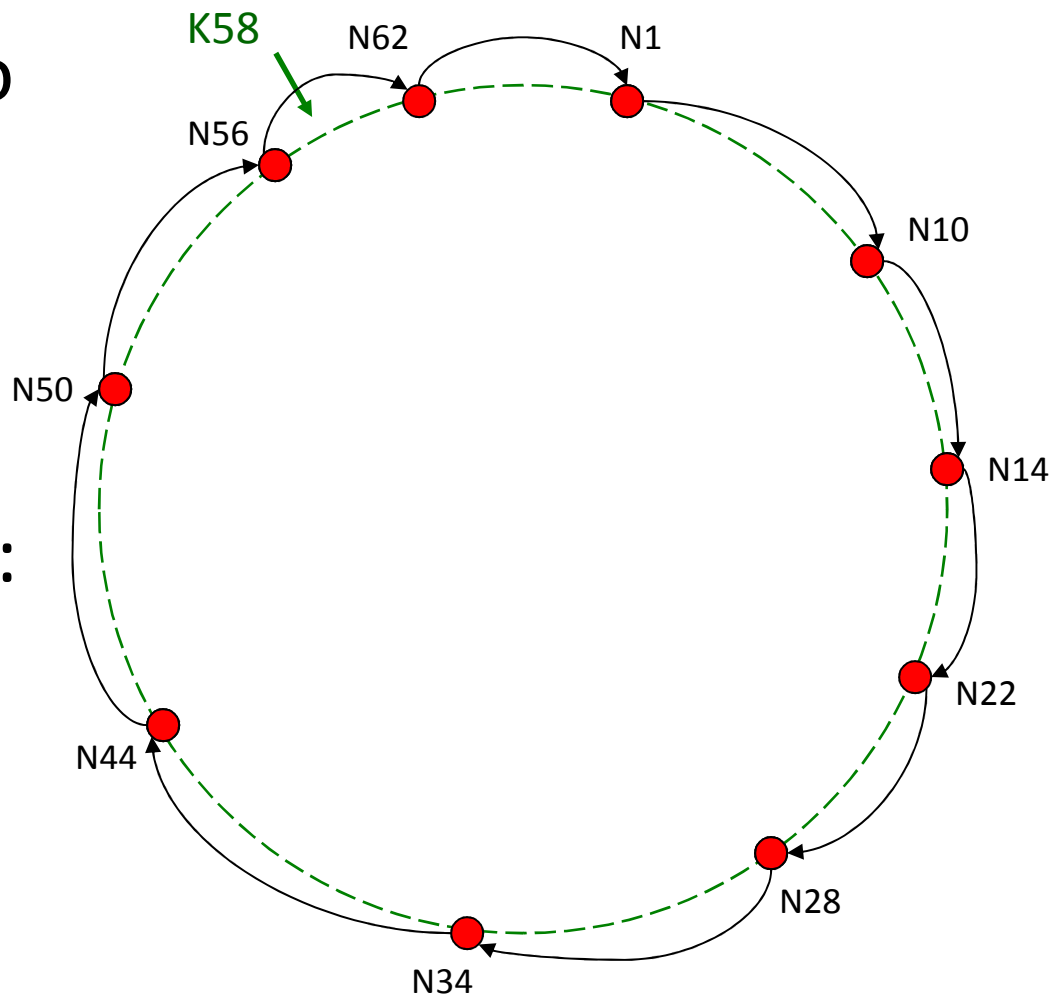
- Sistemas p2p formam uma rede ao nível aplicacional
- Rede pode ser vista como grafo G = (V,E)
  - Cada aresta em E implica que um nó (processo) conhece o endereço / identificador de outro
  - Encaminhamento pela rede G para chegar ao nó de interesse
- Dois tipos de redes sobrepostas
  - Não estruturadas – sem restrições sobre G
  - Estruturadas – estrutura específica. Exemplo: **Chord**

Chord: visão geral

- Chord usa consistent hashing
  - Chave k replicada nos N sucessores de k
- Identificadores dos nós: n=SHA-1(IP:porto)
  - 160 bits
- Primitiva central: find_successor(k)
  - Encaminha procura pelo grafo G até encontrar o sucessor da chave
  - Este vai contactar as restantes réplicas

Primeira tentativa

- Cada nó conhece o seu sucessor
- Encaminhamento percorre os sucessores até encontrar i tal que: i < k < i.successor
- Vantagens / desvantagens?



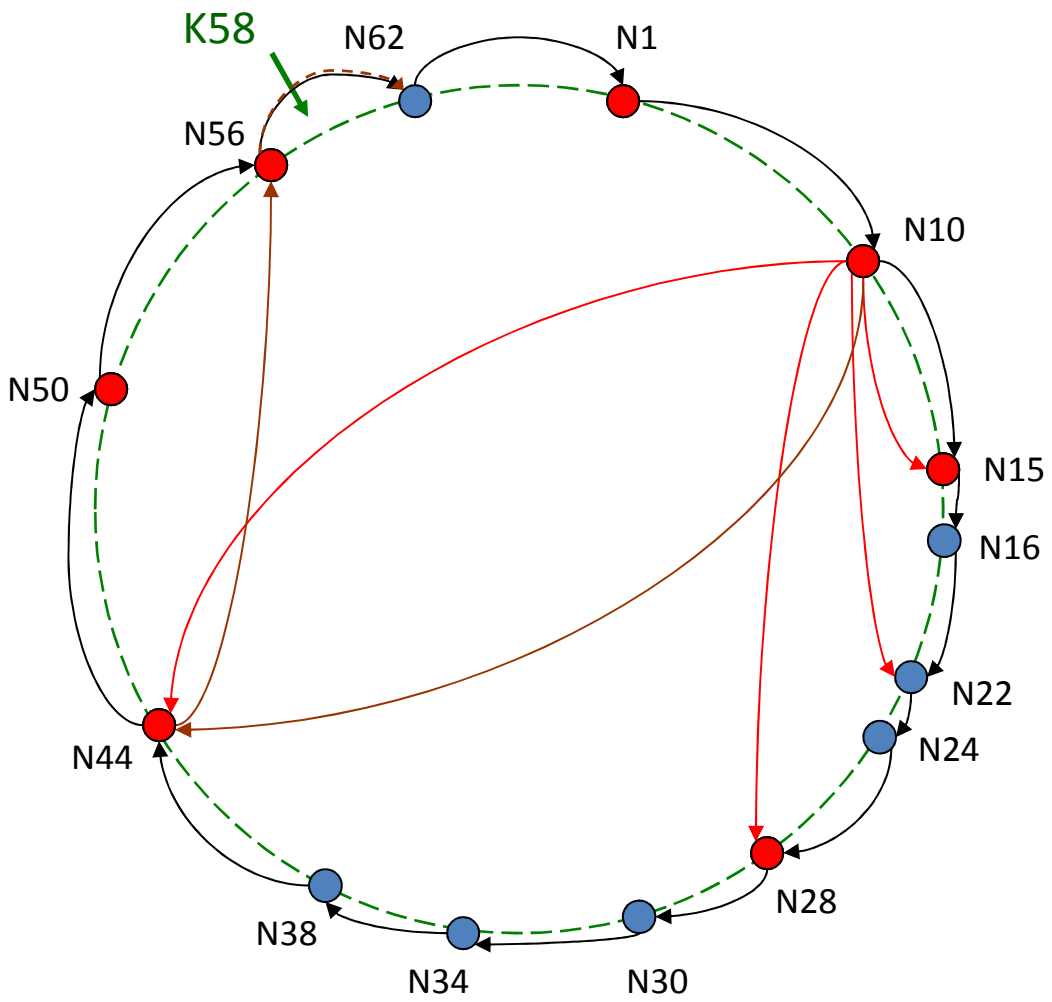
Solução: mais estado para “routing”

- Informação extra não é necessária para a correcção, apenas para performance
- Consiste numa tabela de “fingers” com m elementos (m=número de bits no id)
- Cada entrada é um nó do sistema, ou seja, o par <identificador,endereço IP>
- Permite dar “saltos” no círculo
  - Quanto mais longe, mais espaçados são os saltos consecutivos que se conseguem dar (aumenta a precisão à medida que se aproxima do destino)
- finger[i] = sucessor($n+2^{i-1}$), i=1,..,m

Exemplo

- **m = 6**
- **Qual a tabela de fingers do nó com id = 10?**

- **Como se pode encontrar o responsável pela chave 58?**



Algoritmo de “lookup”

```
// ask node n to find id's successor
n.find_successor(id) {
    n' = find_predecessor(id);
    return n'.successor;
}

// ask node n to find id's predecessor
n.find_predecessor(id) {
    n' = n;
    while (id ∉ ]n',n'.successor])
        n' = n'.closest_preceding_finger(id);
    return n';
}
```

Eficiência do lookup

- Teorema: O número de nós contactados num lookup num sistema com N nós é O(log N)
- Intuição da prova: distância é reduzida a metade em cada passo
- Estado em cada nó também é O(log N)
  - Necessário ter em conta que últimos fingers estão vazios