Chord

Tecnologias de Middleware

2006/2007

Fernando Martins - fmp.martins@gmail.com

Tópicos

- Objectivo
- Motivação
- Peer-To-Peer
- Chord – Descrição Geral
- Características Distintivas
- Comparação DNS
- Modelo do Sistema
- Aplicações e Biblioteca Chord
- Protocolo Chord
- Hash Consistente
- Localização Escalável de Chaves
- Gestão de Nós
- Simulações e Resultados
- Aplicações
- Trabalho Futuro
- Conclusão

Objectivo

- Um serviço de pesquisa (lookup) Peer-2-Peer (P2P) escalável para aplicações na Internet.

Motivação

- Localização eficiente de um nó que possui determinada informação em aplicações P2P.

Peer-To-Peer

- **Peer-To-Peer:** sistema distribuído sem qualquer controlo centralizado ou organização hierárquica onde o software presente em cada nó é equivalente em funcionalidade.
- Operação central na generalidade dos sistemas P2P é a localização eficiente da informação.
- **Exemplos:** Napster, Gnutella.

Chord – Descrição Geral

- Protocolo com operação única: dada uma chave, efectua o mapeamento da chave num nó.
- Usa uma variante de *hash consistente*, que efectua o balanceamento da carga no sistema.
- Envolve poucos movimentos de chaves aquando da entrada ou saída de nós no sistema.
- Cada nó requer informação de rota, apenas sobre alguns nós, com actualização periódica
- Tabela de rotas é distribuída, pelo que um nó resolve a função de *hash* comunicando com outros nós.

Características Distintivas

- *Chord* distingue-se da generalidade dos protocolos de pesquisa P2P em três pontos:
  1. **Simplicidade:** efectua a rota de uma chave através de outros nós.
  2. **Correctude:** apenas necessita de uma parte correcta de informação por nó de forma a garantir a rota correcta das pesquisas.
  3. **Performance:** a degradação é “simpática” quando os nós estão desactualizados.
- *Chord* distingue-se da generalidade dos serviços de pesquisa pela flexibilidade de mapeamento: mapeia chaves em nós e não directamente chaves em valores.

Comparação DNS

- Domínio de Aplicação:
  - DNS é especializado na tarefa de pesquisa de *host names*.
  - *Chord* é genérico, podendo ainda ser usado para encontrar informação que não se encontra ligada a nenhuma máquina em particular.

- Arquitectura:
  - DNS assenta em servidores especiais.
  - *Chord* não requer qualquer tipo de servidor especial.

Comparação DNS

- Mapeamento:
  - DNS disponibiliza *host name* a partir do mapeamento de um endereço IP.
  - *Chord* pode disponibilizar o mesmo serviço usando o nome como chave e o IP associado como o valor.
- Estrutura:
  - Os nomes DNS possuem uma estrutura para reflectir fronteiras administrativas.
  - *Chord* não impõe qualquer estrutura de nomes.

Modelo do Sistema

- **Carregamento Balanceado:** *Chord* actua como uma função de *hash consistente* espalhando as chaves de forma equilibrada pelos nós.
- **Descentralização:** *Chord* é totalmente distribuído. Nenhum nó é mais importante que outro.
- **Escalabilidade:** o custo de uma pesquisa do *Chord* cresce com o logaritmo do número de nós. Não requer qualquer parametrização.

Modelo do Sistema

- **Disponibilidade:** *Chord* ajusta automaticamente as tabelas internas para reflectir a entrada e saída de nós. Assegura que um nó responsável por uma chave pode ser sempre encontrado.

- **Flexibilidade:** *Chord* não coloca qualquer restrição na estrutura das chaves.

Aplicações e Biblioteca Chord

- *Chord* toma a forma de uma biblioteca ligada às aplicações cliente e servidor.
- As aplicações interagem com o *Chord* de duas formas:
  - `lookup(key)`, algoritmo que retorna o IP do nó responsável pela chave `key`.
  - *Software Chord* em cada nó notifica a aplicação das alterações que ocorram no conjunto de chaves das quais o nó é responsável.
- A aplicação que usa o *Chord* é responsável por gerir autenticação, *cache*, replicação, nomes de chaves compreensíveis para o utilizador, etc..

Protocolo Chord

- De forma simples, o *Chord* disponibiliza uma computação distribuída rápida de uma função de *hash* que mapeia chaves em nós responsáveis pelas chaves.

- Usa *hash consistente*:
  - balanceamento;
  - baixo impacto nas entradas e saídas de nós;
  - escalável;

Protocolo Chord

- Protocolo *Chord* especifica como:
  - encontrar a localização das chaves;
  - os nós entram e saem do sistema;
  - recuperar de falhas nos nós existentes.

- Nó *Chord*:
  - necessita apenas de uma pequena quantidade de informação de rota sobre os outros nós;
  - efectua a resolução de uma função *hash* através da comunicação com outros, poucos, nós.

- *Chord* tem de actualizar a informação de rota sempre que um nó entra ou sai da rede.

Hash Consistente

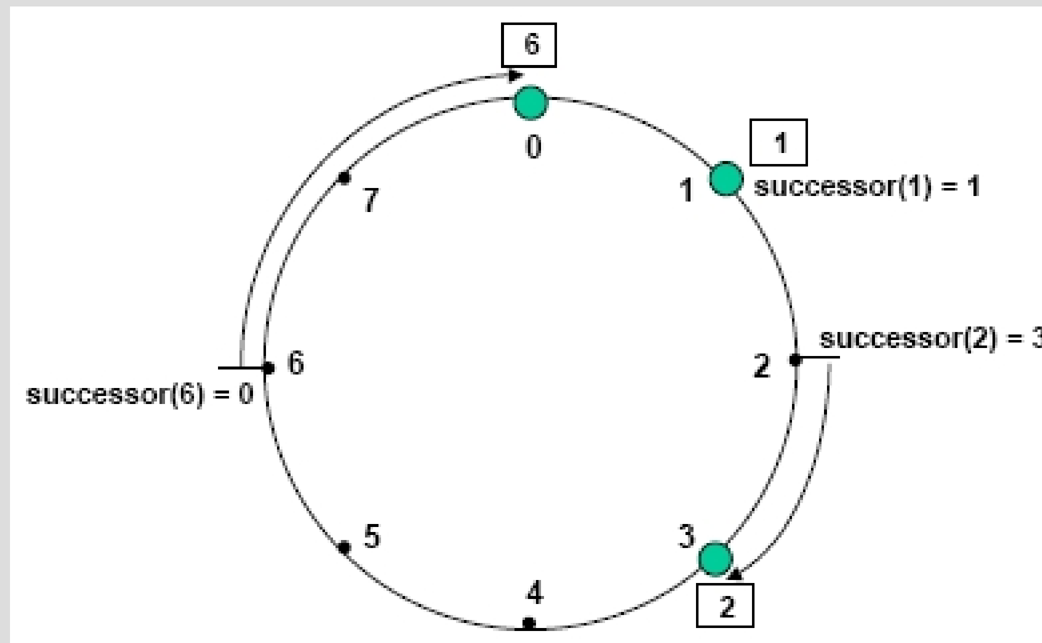
- Função *hash* atribui a cada nó e chave um identificador *m*-bit, usando como base o SHA-1.
- O identificador do nó é escolhido através do *hash* do IP do nó.
- O identificador da chave é escolhido através do *hash* da chave.

Hash Consistente

- Atribuição de chaves a nós:
  - Identificadores estão ordenados num *circulo identificador* módulo 2^m.
  - Chave *k* é atribuída ao primeiro nó cujo identificador é igual, ou seguinte ao identificador, *k*.
  - O nó seguinte é conhecido como o nó sucessor da chave *k*, denominado por `successor(k)`.

Hash Consistente

- Circulo identificador com *m*=3.
- Possui 3 nós: 0, 1 e 3.
- Sucessor do identificador 1 é o nó 1, logo a chave 1 encontra-se no nó 1.



Localização Escalável de Chaves

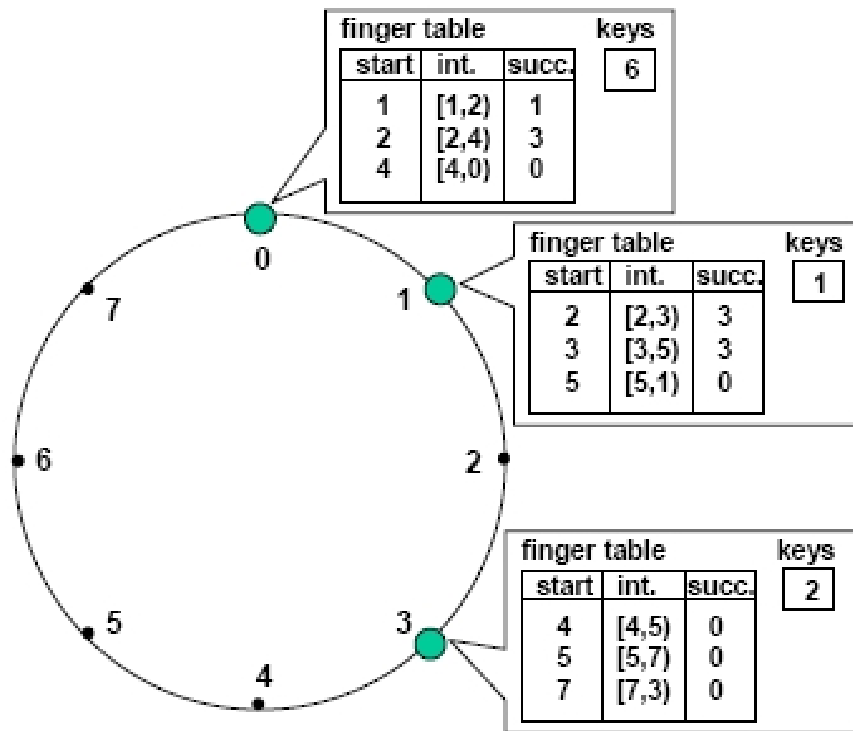
- Cada nó tem apenas de ter conhecimento sobre o seu sucessor no círculo.
- Pesquisas sobre um identificador podem ser passadas à volta do círculo através dos sucessores até encontrar um nó que sucede o identificador: o nó que a pesquisa mapeia.
- Este esquema de resolução garante a correcta resolução das pesquisas mas é insuficiente em termos de performance.

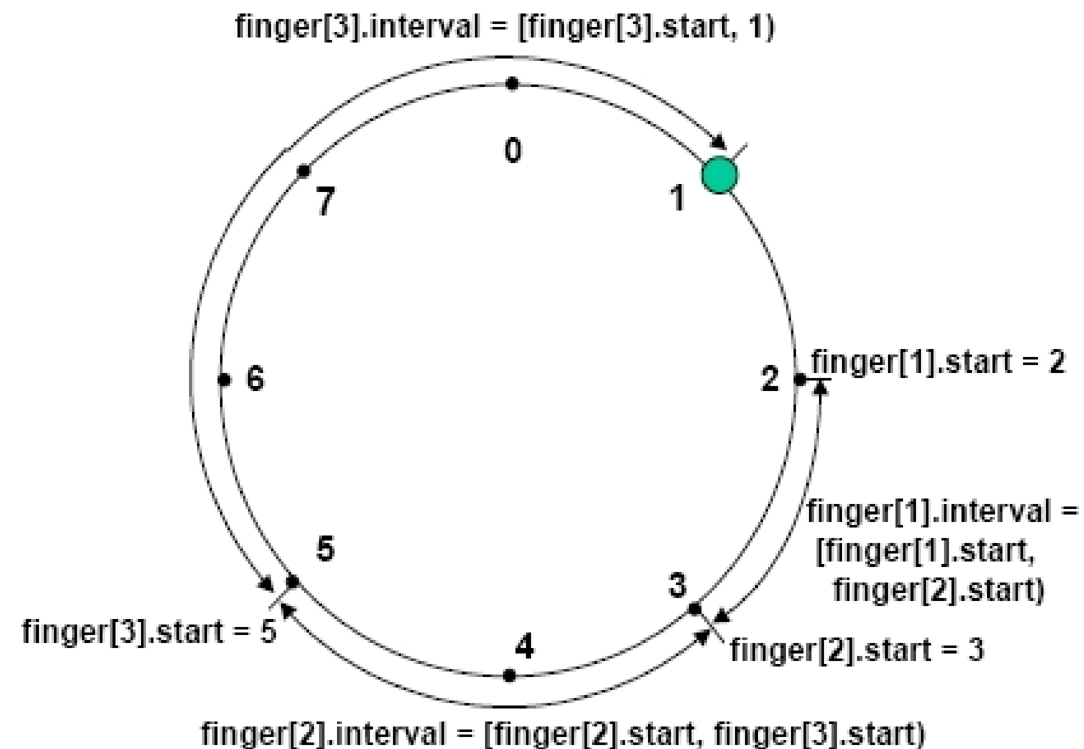
Localização Escalável de Chaves

- Tabela de rotas, ou tabela apontadora (finger):
  - presente em cada nó;
  - possui informação sobre o identificador e o IP do nó relevante;
  - contém, pelo menos, *m* entradas mas possui informação apenas sobre alguns nós;
  - primeira entrada refere sempre o nó seguinte;
  - a *i-ésima* entrada na tabela no nó *n* contém a identidade do primeiro nó, *s*, que sucede *n* por, pelo menos, 2^{i-1} no círculo identificador;
  - nó *s* é o *i-ésimo finger* do nó *n* e denomina-se por `n.finger[i].node`.

Localização Escalável de Chaves

- Tabelas de rotas
- *Finger*





Notation	Definition
finger$[k]$*.start*	$(n + 2^{k-1}) \bmod 2^m$, $1 \leq k \leq m$
.interval	$[$*finger*$[k]$*.start*, *finger*$[k+1]$*.start*$)$
.node	first node $\geq$ *n.finger*$[k]$*.start*
successor	the next node on the identifier circle; *finger*$[1]$*.node*
predecessor	the previous node on the identifier circle

Gestão de Nós

- *Chord* consegue encontrar qualquer nó na rede mesmo com nós constantemente a entrar e sair, para tal necessita de preservar dois invariantes:
  - o sucessor de cada nó é correctamente mantido;
  - para cada chave *k* o nó `successor(k)` é responsável por *k*.
- Por razões de performance, é desejável que as tabelas de rota se mantenham correctas.
- Para simplificar a gestão de entradas e saídas de nós, cada nó possui um apontador para o seu antecessor.

Gestão de Nós

- Para preservar os invariantes, o *Chord* tem de efectuar três tarefas quando um nó entra:
  1. inicializar o antecessor e a tabela de rota do nó *n*;
  2. actualizar as tabelas de rota e os antecessores dos nós existentes de forma a reflectir a adição do nó *n* na rede;
  3. notificar a camada superior de software para que possa transferir o estado, e.g. valores, associados com as chaves das quais o nó *n* é agora responsável.
- Assume-se que o novo nó conhece um nó *n'* existente através de um mecanismo externo

Gestão de Nós

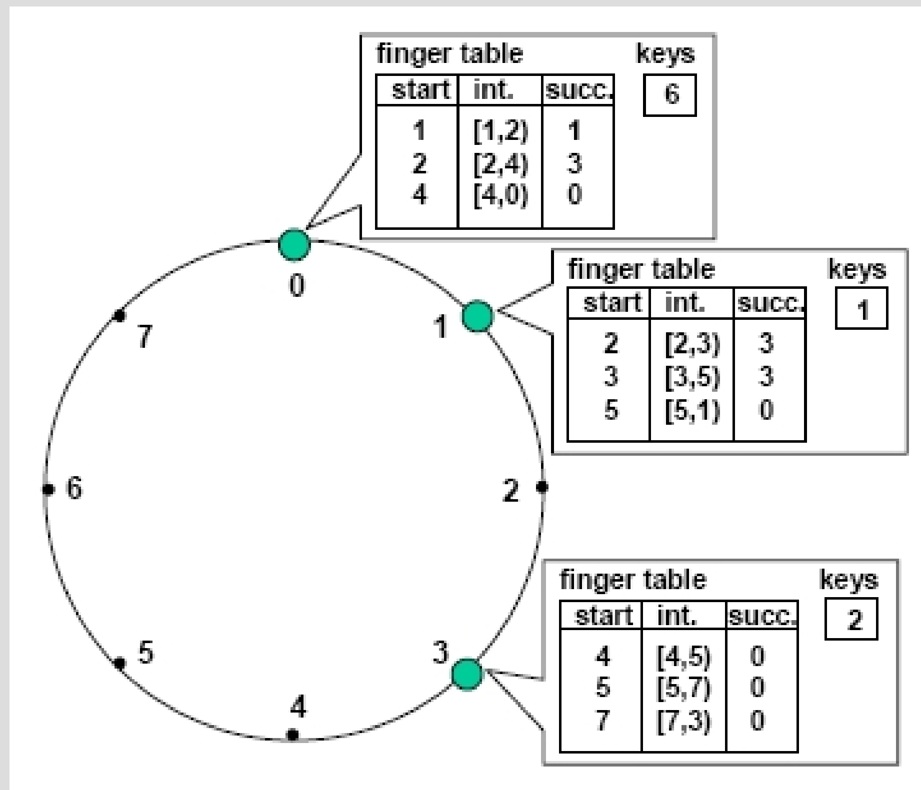
- Pseudocódigo:

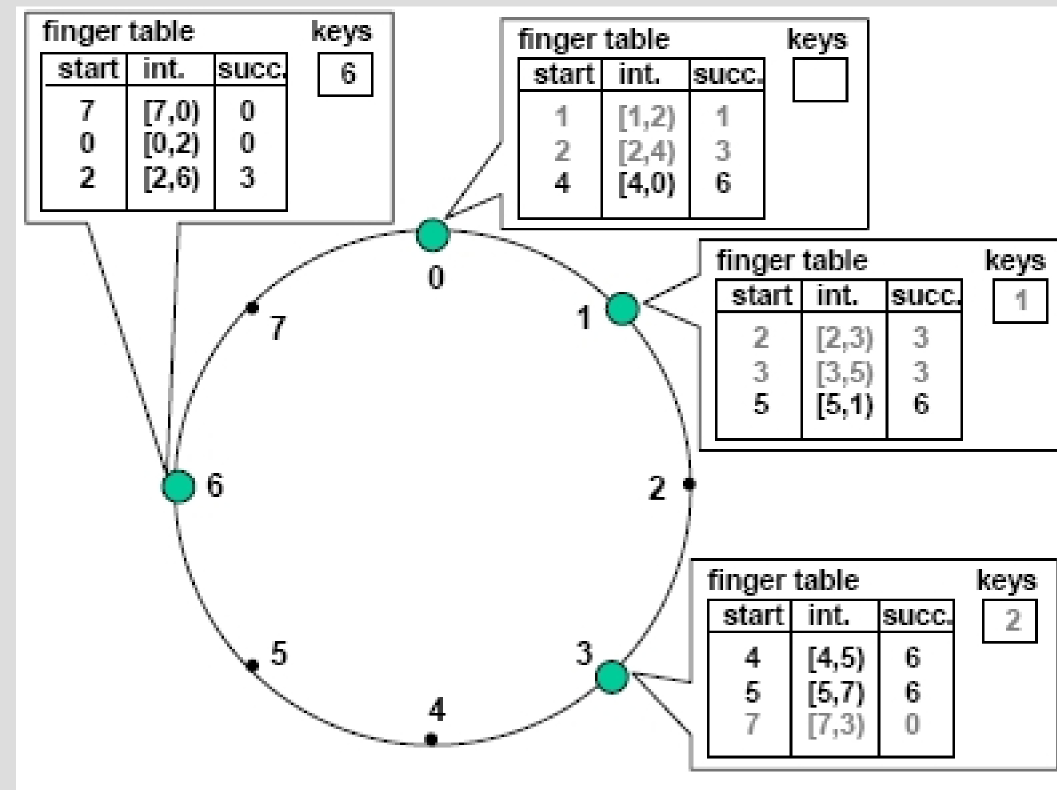
```
#define successor finger[1].node

// node n joins the network;
// n' is an arbitrary node in the network
n.join(n')
  if (n')
    init_finger_table(n');
    update_others();
    // move keys in (predecessor, n] from successor
  else // n is the only node in the network
    for i = 1 to m
      finger[i].node = n;
    predecessor = n;

// initialize finger table of local node;
// n' is an arbitrary node already in the network
n.init_finger_table(n')
  finger[1].node = n'.find_successor(finger[1].start);
  predecessor = successor.predecessor;
  successor.predecessor = n;
  for i = 1 to m - 1
    if (finger[i + 1].start ∈ [n, finger[i].node))
      finger[i + 1].node = finger[i].node;
    else
      finger[i + 1].node =
          n'.find_successor(finger[i + 1].start);
```

```
// update all nodes whose finger
// tables should refer to n
n.update_others()
  for i = 1 to m
    // find last node p whose i^th finger might be n
    p = find_predecessor(n - 2^(i-1));
    p.update_finger_table(n, i);

// if s is i^th finger of n, update n's finger table with s
n.update_finger_table(s, i)
  if (s ∈ [n, finger[i].node))
    finger[i].node = s;
    p = predecessor; // get first node preceding n
    p.update_finger_table(s, i);
```

Gestão de Nós

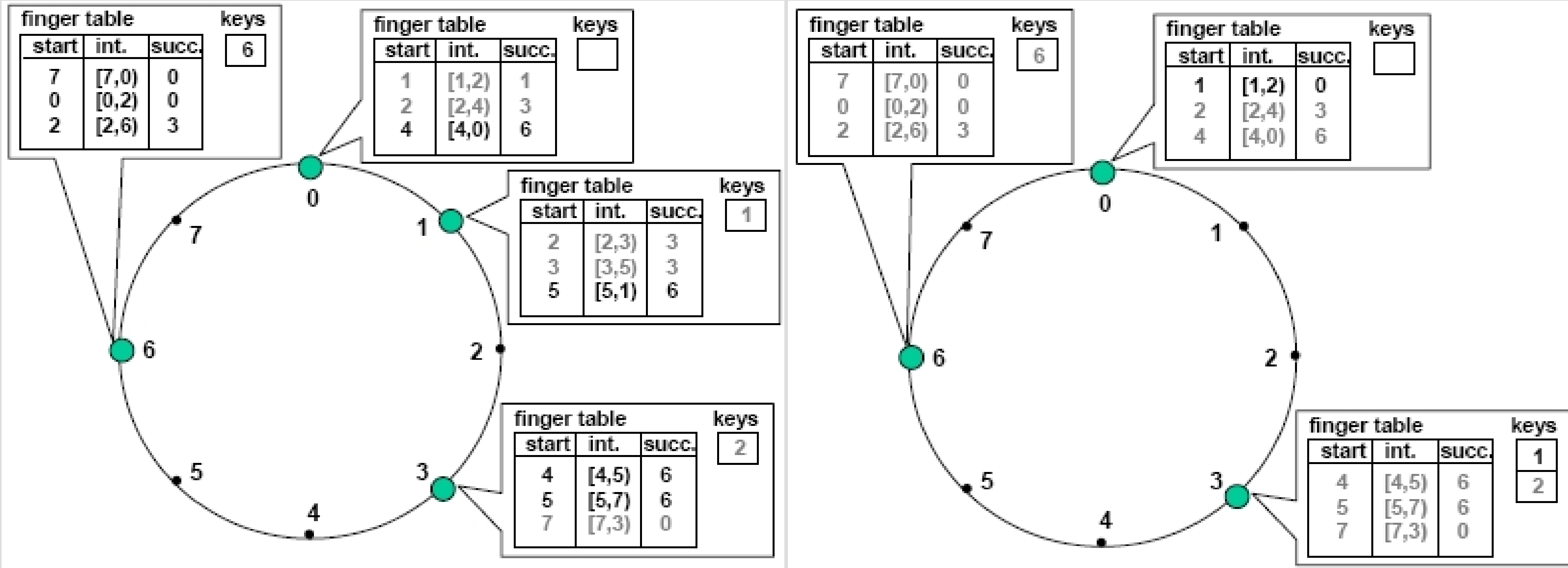
- Entrada de um novo nó 6:





Gestão de Nós

- Saída do nó 1:



Gestão de Nós

- Estabilização:
  - função `stabilize` de um nó *n* questiona o sucessor de *n* pelo antecessor do sucessor *p* e decide se *p* deve passar a ser o sucessor de *n*;
  - cada nó executa `stabilize` de forma periódica;

```
n.join(n')
    predecessor = nil;
    successor = n'.find_successor(n);

// periodically verify n's immediate successor,
// and tell the successor about n.
n.stabilize()
    x = successor.predecessor;
    if (x ∈ (n, successor))
        successor = x;
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
    if (predecessor is nil or n' ∈ (predecessor, n))
        predecessor = n';

// periodically refresh finger table entries.
n.fix_fingers()
    i = random index > 1 into finger[];
    finger[i].node = find_successor(finger[i].start);
```

Gestão de Nós

- Falhas:
  - quando um nó *n* falha, todos os nós cujas tabelas de rota incluem *n* necessitam de encontrar o sucessor de *n*;
  - cada nó mantém uma lista de sucessores dos seus *r* sucessores mais próximos.

```
n.join(n')
   predecessor = nil;
   successor = n'.find_successor(n);

// periodically verify n's immediate successor,
// and tell the successor about n.
n.stabilize()
   x = successor.predecessor;
   if (x ∈ (n, successor))
      successor = x;
   successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
   if (predecessor is nil or n' ∈ (predecessor, n))
      predecessor = n';

// periodically refresh finger table entries.
n.fix_fingers()
   i = random index > 1 into finger[];
   finger[i].node = find_successor(finger[i].start);
```

Gestão de Nós

- Replicação:
  - a lista de sucessores funciona como um mecanismo que permite que a camada mais alta de software replique dados;
  - sempre que um nó entra ou sai, o software sabe quando deve ocorrer a propagação de réplicas.

Simulações e Resultados

- Testes efectuados com uma versão menos evoluída do que a aqui apresentada.
- **Load Balance:** *hash consistente* efectua uma distribuição uniforme das chaves pelos nós.
- **Comprimento do Caminho:** cresce de forma logarítmica com o número de nós.
- **Falhas Simultâneas de Nós:** não existem falhas significativas na pesquisa de nós.

Simulações e Resultados

- **Pesquisas Durante a Estabilização:** taxa de falhas na ordem dos 3%.
- **Escalabilidade:** a latência nas pesquisas cresce lentamente com o número total de nós, resultando numa boa escalabilidade.

Aplicações

- **Réplicas Cooperativas:** réplicas de servidores de ficheiros para *download*, permitindo inclusive o balanceamento da carga entre os servidores.
- **Índices Distribuídos:** para suportar pesquisas por palavras chave como Napster ou Gnutella.
- **Pesquisa Combinatória em Larga Escala:** tal como quebra de códigos criptográficos.

Trabalho Futuro

- **Recuperação de Círculos Partidos:** *Chord* não possui mecanismos específicos para recuperar de círculos partidos.
- **Segurança:** *Chord* está sujeito a ataques uma vez que é possível a introdução de um nó no círculo com informação errada.
- **Latência de Pesquisa:** melhorar o tempo de latência nas pesquisa.

Conclusão

- Protocolo *Chord* resolve o problema da procura do nó que contém a informação usando uma abordagem descentralizada.

- Possui uma primitiva poderosa, dado uma chave, determina, de forma eficiente, o nó responsável por guardar o valor da chave.

- Garante o correcto funcionamento do sistema, embora de forma degradada, quando a informação de um nó está parcialmente correcta.

Conclusão

- Carregamento balanceado.
- Boa escalabilidade.
- Recupera de uma grande quantidade de falhas simultâneas de nós.
- Sistema genérico com várias aplicações possíveis.

Bibliografia

- Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishna. “Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications”. SIGCOMM, ACM, 2001.