Freenet つづき

ファイルの公開
  実際には, 公開するファイルは descriptive text で暗号化する
    (inserted files are encrypted by the descriptive text)
    検索した人は descriptive text を知っているので読める
    (the user requested the file can decode the file)
    ディスク領域を「貸している」だけの人は読めない
      (but the owner of the storage cannot decode it)
      否認が可能   (i.e. the owner can deny storing it)

Managing data
  ディスク領域は LRU (Least Recently Used) で管理
    (disk space is managed in the LRU fashon)
    参照されない古いファイルは捨てる (unrequested files are faded)
    ファイルは永久に保存されるわけではない
  routing table も同様 (same for the routing table)

Performance
 モデルによるシミュレーション(simulation results on a model)
---
network convergence - adaptivity of the routing
  1000 nodes
  datastore: 50 items
  routing table: 250 addresses
    initialized to two neighbors in a ring
    key: hash of its address

  insertion と request を繰り返す
    insert
      to: random node
      key: random
      hops-to-live: 20
    request
      hops-to-live: 20

  probe: hops-to-live = 500
    request pathlength
      number of hops taken before finding the data
      500 if the request failed
      これで性能評価 (evaluate the system with average pathlength)

  Figure 2
    1/4, 1/2, 3/4 点の変化
    中間値は6まで低下した (the median point falled to 6 hops)
---
scalability of a growing network
  20ノードから始めて (starting with 20 nodes,)
  ノードを追加しつつ request pathlength を測定
    (measure the path-length while adding nodes)
    node announcement の hops-to-live = 10

  Figure 3
    x軸はノード数(対数)
    1/4, 1/2, 3/4 点の変化
    pathlength は対数的に増加 (increases logarithmically)
      40000ノードのところで傾きが変化 (change the slope)
      routing table が溢れたためだろう (overflow of the routing table)

Fault tolerance
  1000 ノードから始めて (starting with 1000 nodes,)
  ノードを徐々に削除する (delete nodes gradually)

  Figure 4
    500 が失敗ということ (pathlength=500 means search failure)
    30% のノードがfailしても, パス長は20以下
    (even 30% of the nodes fail, the pathlength is below 20 hops)

```
================================================================================
"Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications"
Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan
ACM SIGCOMM '01
--------------------------------------------------------------
Introduction
  structured P2P
  key -> node の対応(mapping)をつける protocol
  DHT (Distributed Hash Table) のひとつ

  Consistent Hashing にもとづく

Hashing
  表の検索技法の一つ (table lookup method)
  検索対象(key) -(hash function)-> hash value (H)
  table[H mod tablesize] に格納(store)

  hash function は検索対象を均一に分散させる
  (hash function distributes keys uniformly)
  table[]には検索対象が均一に分布
  検索の計算量は O(1)
  (lookup is done in constant time)

「Consistent」
  nodeの増減によって, key->nodeの対応がほとんど変化しない
  (change in the number of nodes does not affect matchings already done)
  (上例の mod では, tablesize が変化するとすべて変化)
  (the above example using 'mod' operation is not consistent)
--------------------------------------------------------------
3. System Model

Load balance:
  Chord acts as a distributed hash function,
  spreading keys evenly over the nodes; this provides a degree
  of natural load balance.

Decentralization:
  Chord is fully distributed: no node is
  more important than any other. This improves robustness and
  makes Chord appropriate for loosely-organized peer-to-peer
  applications.

Scalability:
  The cost of a Chord lookup grows as the log of
  the number of nodes, so even very large systems are feasible.
  No parameter tuning is required to achieve this scaling.

Availability: arrival or leave
  Chord automatically adjusts its internal tables
  to reflect newly joined nodes as well as node failures, ensuring
  that, barring major failures in the underlying network, the
  node responsible for a key can always be found. This is true
  even if the system is in a continuous state of change.

Flexible naming:
  Chord places no constraints on the structure
  of the keys it looks up: the Chord key-space is flat. This
  gives applications a large amount of flexibility in how they
  map their own names to Chord keys.
--------------------------------------------------------------
N: number of nodes
ノードの出入りがない状態で (without node arrival nor leave)
  ノードの必要メモリ量 O(log N) (necessary memory at each node)
  key->nodeの検索時間 O(log N) (time to look up the key)
ノードの出入りの際に必要なメッセージ数 O((log N)^2)
(number of messages necessary for node arrival or leave)
--------------------------------------------------------------
```

多様なアプリケーションが Chord の上に実装可能
(various applications are possible on Chord)
  Cooperative Mirroring
  Time-Shared Storage
  Distributed Indexes
  Large-Scale Combinatorial Search
---
4. Basic Chord Protocol
---
  consistent hashing such as SHA-1 (160bit = m)
  node (IP addr) -> m-bit identifier
  key   -> m-bit identifier

  identifier circle (modulo 2^m)
  ex. m=3 なら, 0から7までの環状

  node, key とも, この circle 上にあると考える
  (both nodes and keys are located on the circle)

  key k を担当する node -- successor(k)
    k 以上の identifier をもつ最初の node (modulo 2^m)
    (the first node whose id >= k)

    時計回りで最初のノード
    (the first node after the key in clockwise)

  Figure 2 に例

  node n が join
    successor(n)が担当していた key の一部を担当する
    (n will be assigned some keys previously assigned to successor(n))

  node n が leave
    担当していた key を successor(n) に譲る
    (hand assigned keys to successor(n))

  node 当たりの key 数は均等になる
  (number of keys per node will be equal)
  Theorem 1

  故意に key が同じになるようなデータを選ぶことは
  (user might choose the key to result arbitrally hash value...)
  hash function の性質上むずかしいので安心
  (but is difficult from the property of the hash function)

4.3 Scalable Key Location

個々の node の持つ情報 (data held by each node) - Table 1
  successor   - 自分の直後の node(のidentifier と IPアドレス)
    これだけあれば, O(N)での検索が可能
    (this is sufficient for O(N) lookup)

  predecessor - 自分の直前のnode(のidentifier と IPアドレス)
    join などで使用 (後述)
    (used in node join - explained later)

  finger table - m エントリの表 (has m entries)
    m: identifier の bit数 (number of bits of hash value - ex. 160)
    1,2,4,8...と離れた key の担当 node を記録
    (record nodes which are distant by 1,2,4,8... from the node)

    finger[i] = successor(n + 2^(i-1))   (i=1...m)
      n: その node の identifier

    自分の近くは詳しいが, 遠くについてはおおまか
    (have detailed knowledge about neighbors)

  表記 (notations)

```
  finger[i].start     (n + 2^(i-1)) mod 2^m (1<=i<=m)
  finger[i].interval  [finger[i].start, finger[i+1].start)
  finger[i].node      first node >= finger[i].start
      (i.e. successor(finger[i].start))

  Figure 3
```

検索の実行 (lookup)
  key k の担当 node (successor(k)) を求めるには
  (find the node which is assigned the key k -- successor(k))
  finger table を利用して, 時計回りに目的地へ近付く
  (using finger table, approach the node clockwise)

```
// 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 - id未満の最後のノード
n.find_predecessor(id)
  n' = n;
  while (id !∈ (n',  n'.successor] )
    // !∈: 区間に含まれない (not included in the section)
    n' = n'.closest_preceding_finger(id);
  return n';

// return closest finger preceding id
n.closest_preceding_finger(id)
  for i = m downto 1
    if (finger[i].node ∈ (n, id))
      return finger[i].node;
  return n;
```

Figure 4 の疑似コード
  n.f() は遠隔手続き呼び出し
  (remote procedure call - evaluate f() on the node n)
  n.v は遠隔変数参照
  (remote variable reference)

Figure 3(a) で検索の例
  node 3 が successor(1) を知りたい
    正解は 1 (the result will be 1)
  3.find_successor(1)
    3.find_predecessor(1)
      3.closest_preceding_finger(1)
        3.finger[3].node(=0) ∈ (3, 1)
      return 0
    return 0
  0.successor is 1
  return 1

Theorem 2
検索時にコンタクトするノード数は O(log N)
  (number of nodes contacted in the lookup process is O(log N))
  目的地までの残り距離が半分に減少していくため
  (because the distance left is reduced to half in each step)

実際には (1/2)logN になる
(more closely, (1/2)logN)

4.4 Node joins

key の検索が常にできるための invariant
(invariants for successful lookup of the key)
1. 各ノードの successor が正しい
   (each node has correct successor pointer)
2. 各キーkについて, successor(k) がその担当ノードである
   (each key is assigned to the node successor(k))

まず, single join を考える (simultaneous join は Sec. 5)

初期接続ノード n' の発見は論じない
(initial node n' is given in some way)

新規ノード n が join した時の処理は三つ
(new node n joins, three tasks will be done)
1. 新規ノード n の successor, predeccessor, finger を設定
   (data in new node n is established)
2. 既存ノードの finger を新規ノード n を反映するように修正
   (existing nodes will modify their finger table)
3. 上位層にいくつかの key の担当が n になったことを通知
   (upper layer is notified the change of the assignment of keys)
   対応するコンテンツなどを新規ノードへ転送するとか
   (and might transfer corresponding contents to the new node)

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

// node n joins the network;
// n' is an arbitrary node in the network
n.join(n')
  if (n') // there is an initial node
    init_finger_table(n'); // (1)
    update_others();       // (2)
    // move keys in (predecessor; n] from successor // (3)
  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 ith finger might be n
    p = find_predecessor(n - 2^(i-1));
    p.update_finger_table(n, i);

// if s is ith 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); // repeat
```

1. ノード n' に問い合わせることで, successor, predecessor, finger を設定
   (consulting the initial node n', n sets its own data)
  finger を設定するのに, naive にやると O(m log N)
  (finger table has m entries, so filling it naively will cost O(m log N)...)
  エントリ個数 m の中に同じものがある可能性が高いので,
  (the entries are not completely filled -- may have same nodes)
  それを処理すると, O((log N)^2)

  なぜなら
  ノードの平均間隔が (2^m)/N だから, それ未満の距離には誰もいないと

(average distance of nodes is 2^m/N, there will be no nodes below the distance)
考えると, 誰かいるエントリの数は m - log(2^m/N) = logN

2. 新規ノード n を finger table[i] に含むようなノードに, 修正を依頼する
(ask nodes which will have the new node in there finger table to modify them)
その条件: 2^(i-1) 以上前に位置して, finger[i] が n の先にある
(such nodes are: preceeds n more than 2^(i-1); finger[i] > n)

ひとり修正したら, 反時計回りに順に修正を続ける(再帰)
(repeat the modification following predecessor link - by recursion)

Figure 5

3. 新たに n が担当する key に関するデータの移動
(transfer contents newly assigned to n)
直後のノード(successor(n)になったノード)からだけ
(only from successor(n))
