Chord つづき

5. Concurrent Operations and Failures

5.1 Stabilization

前節の join 処理は, 影響を受けるノード全てをその場で修正
(join processing explained above will do all the task at once)
  ノード数増大
  同時に join
では現実的ではない
(it is not practical for large number of nodes and simultaneous joins)

correctness と performance の分離 (separation)

correctness
  successor が正しく update されていること
  そのための処理をstabilization と呼ぶ

performance
  finger table の update
  successor を利用して更新
  (update finger table using successor link)

join による検索への影響 - 三つのケース
(effect of joins on lookup - three cases)
- finger table が影響されない - 影響なし
 (finger table is not affected - no effect)
- finger table の更新がまだ - 検索が遅くなるだけ
  (finger table is not updated yet - only need extra step for lookup)
  predecessor が(反時計回りに)行き過ぎる
  (find_predecessor answers the predecessor of correct one)
  最後の段階で successor をとったところで, まだ先だったらもう一段
  (one more successor step is sufficient)
- successor も未更新 - 検索失敗
  (successor link is not updated yet - lookup might fail)
  上位層で retry とか (retry from the upper layer)

```
======
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);
======
```

新規ノード n は, join(n') を呼ぶ
  (new node calls join(n'); n' is the initial node)
  自分(n)の successor を設定するだけ
  (only set the successor of n)

全ノードは stabilize() を定期的に呼び出す

(all nodes call stabilize() periodically)
  successor, predecessor の更新 (update)

新規ノード加入による不整合は一時的なもの
(inconsistency caused by node join is not permanent)
Theorem 4, 5

例) 新規ノード参加の後の双方向リンクの確立
(before) n1 - - - n2

(join n) n1 - n - n2
  n.successor = n2

(n.stabilize())
  n2.notify(n)
    (before: n2.predecessos = n1)
    n2.predecessor = n

(n1.stabilize())
  (before: n1.successor = n2)
  n1.successor = n
  n.notify(n1)
    (before: n.precesessor = nil)
    n.predecessor = n2

例) 二つのノードが successorを s と思い込む
(if two nodes mistakenly point the same node s as their successor,)
stabilize の中で, どちらも s に notify
(those two nodes both notify s in their stabilize procedure)
s は近い方を自分の predecessor に採用
(s will adopt nearer node as its predecessor)
遠い方は次回の stabilize で, 自分のsuccessorを更新
(the mistaken node will adjust its successor at the next invocation
 of stabilize)

finger table の調整 (adjustment)
加入ノードがあっても, 検索に大きな影響はなし
(node join has no effect for lookup except ..)
目標の直前に新規ノードが加入した場合だけ
(when the new node joins just before the target node)
  finger で到達したノードから, successor リンクを
  たどればよい
  (it costs only one more successor step)

Theorem 6
N 個までの新規ノードの加入は検索時間に影響せず
(node joins up to N don’t affect the lookup time)
  O(log N)

5.2 Failures and Replication

ノードの故障 (node failure)
  successorリンクの維持が大切
  (maintain the successor links are important)

各ノードは successor-list を保持
(each node keeps successor-list)
  r個の近い successors
  successor が fail したら, そのリストから
  順次探して successor を再設定
  (if the successor fails, find next one from the list)

finger table からアクセスしたら故障していた場合
(if the finger table points a node which was failed)
  time out
  それより近い finger table entry のノードへ転送
  (use other node listed in the previous finger table entry)

Theorem 7, 8
長さ O(log N) の successor-list を保持していれば,
(by keeping O(log N) successor list,)
リストの全ノードが確率1/2で故障しても検索可能
(if all node fail with probability 0.5, lookup is still possible)
時間は O(log N)

説明) r 個のsuccessor-listが全滅する確率は
(all of r nodes in the successor-list will fail simultaneously with
 probability:)
(1/2)^(r) = (1/2)^(log(N)) = 1/N
ほとんどの場合, 全滅しない.
(this is very rare for large N)

successor-list を使って, データのバックアップも可能
(successor-list is also usable for data backup)
(上位層で replica を配布しておく)
---
6. Simulation and Experimental Results

6.1 Protocol Simulator
検索の方法 (lookup scheme)
  iterative - 問い合わせたノードが全通信を行なう
        (the query node sends all messages)
  recursive - 中間ノードが次のノードに問い合わせ
        (intermediate nodes relay queries to the next node)
ここでは iterative を実装
        (used iterative scheme for the simulation)

6.2 Load Balance
キーがノードに平均的に分散するか
(key distributes evenly in nodes?)
ノード (the number of nodes) 10^4
キー (the number of keys) 10^5 から 10^6 まで
20回ずつ実験 (20 simulations)
Fig 8(a) 割り当てキー数 (1%, 平均, 99%)

分布 (キー 5 * 10^5) - Fig 8(b)
  平均は 50 のはず (average will be 50)
  最高は 457 (平均の 9.1倍) (maximum 457)

ノードが等間隔で配置されているわけではないため
(because nodes are not located evenly on the circle)

解決策 virtual node
ひとつのノードに, log N 個の virtual node を担当させる
(one node runs log N virtual nodes)
  キー分担はより均一になる
  (the key distribution will be more even)
総ノード数 (the number of virtual nodes) は N log N
検索時間 は不変 (lookup time is unchanged)
O(log(N log N)) = O(logN + log(logN)) = O(logN)

その検証 (verification by simulation)
virtual node r個 / 実ノード(10^4個)
キー (10^6個) は virtual node に割り当て
Fig 9
r = 1, 2, 5, 10, 20 で, キー個数/実ノード (# of keys/real node)
(1%, 平均, 99%)
分散が減少している (variance is reduced)

virtual node の問題点 (problems on virtual nodes)
実ノードが管理する table が r倍に
(table size increases r times)
  実害はなさそう (not serious)

6.3 Path Length
検索でのホップ数 (# of hops in lookup)

ノード (# of nodes) 2^k
キー (# of keys) 100 * 2^k
k = 3 ... 14

Fig 10(a)
パス長 (path length) (1%, 平均, 99%)
対数で増加 (increase logarithmically)
さらに, 定数は 1/2 (constant factor is 0.5)
理由
  identifier space での距離を二進数で表現
  (by representing distance as a binary number in identifier space,)
  finger table は, ビットが「1」のところで使う
  (finger table is used if the corresponding bit is '1')
  したがって, 平均的には 1/2 の確率になる
  (its probability is 0.5)

6.4 Simultaneous Node Failures
複数の故障が同時に発生した時の検索の成功率
ノード (# of nodes) 10^4
キー (# of keys) 10^6
全体の p の割合のノードを故障させる
(p of all the node will fail simultaneously)
stabilize するまで待つ
(wait for the completion of stabilization)
キーの検索数を測定
(examine the success rate of lookup)

Fig 11
故障率 p と 検索失敗率 (failure rate)
ほぼ一致している (almost same)
  故障ノードが持っていたキーだけが検索不能になった
  (only keys held by the failed node is inaccessible)
故障の影響を受けないといえる
(other keys are not affected by failures)

仮にリングが二つに分裂とかしてしまうと,
(if the ring splits to two independent rings,)
もっと高い失敗率になるはず
(the failure rate would be high)
robust だったと
(such splitting was not occured i.e. robust)

6.5 Lookups During Stabilization
stabilize 完了前の検索の失敗 - 二つの可能性 (two cases for failure)
  - キーを担当するノードの故障
        (failure of the node which hold the key)
  - finger table, predecessor の不正 (inconsistency)

目標キーの現在のsuccessorに到達したら成功
(the success is to get to the node which is current successor of the key)
失敗の際に retry はしない
(no retry in case of failure)

「参加離脱の頻度」と, 「stabilize の起動間隔」の比が重要
(the ratio of the rate of join/leave and the interval of stabilization
 is important)

検索は 1回/1秒 の ポアソン分布 (1 lookup/sec, Poisson distribution)
参加離脱は 平均R/secのポアソン分布
  (R join,leave/sec, Poisson)
stabilization は 30秒に1回 (1 stabilization/30 sec, random)
  Fig 7 のコードと違い, finger table の全要素を更新
  (update all finger table entries -- different from the code in Fig 7)
初期ノード数は 500 (initial # of nodes is 500)

Fig 12
x軸 - failure rate (R)
0.01 (1 failure / 3 stabilize)

0.1 (3 failure / 1 stabilize)

平均パス長(average path length)は 1/2 log(500) ~ 5
そこに故障したノード(k個とする)が含まれる確率 - 5k/500
(the path contains k failed nodes with the probability 5k/500
右端では 3 failure -> 検索失敗 3% のはず
(failure rate should be 3%)
実際は6%強
(but actually the rate is 6%)
  一度の stabilize では完全には正しくならないからだろう
  (one stabilization is not enough to make consistent)

6.6 Experimental Results
Internet で実験した
  RON test-bed 内の 10サイト
    California, Colorado, Massachusetts, New York,
    North Carolina, Pennsylvania
ノード数を増やすために, 同一サイトで複数の実行

Fig 13
ノード数と検索時間の関係 (5%,median,95%)
(# of nodes, lookup time)
180ノードの場合
  検索に4往復 + 1往復 (4 roundtrips for lookup, 1 for data)
  round trip time = 60ms
  60ms * 5 = 300ms
  実際の中間値は 285ms (measured median value)

"A Scalable Content-Addressable Network"
Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker
ACM SIGCOMM '01
---
1 Introduction
hash table は key を value にマップ(map)する仕組み
  software system の core building block
分散環境でも hash は有用であるはず
(hash is useful in distributed environment)

Content-Addressable Network(CAN)
  distributed Internet-scale hash table の総称
  (現在では DHT(distributed hash table)と呼ぶ)

P2P file sharing の展開
  Napster - file の locating は集中方式 (central server)
  Gnutella - 分散しているが(distributed), flooding
    not scalable
    may fail to find

CAN は file sharing 以外にも用途 (other usage than file sharing)
  large scale storage management system
  wide-area name resolution system
    DNS との対比 (comparison with Domain Name Service)
      名前の構造(naming scheme)に制約
        (hierarchically structured)

CAN の基本動作
  insertion, lookup, deletion of (key, value) pairs

各ノードはハッシュテーブルの一部(zoneと呼ぶ)を担当
(each node takes charge of some potion of the hash table -- zone)
各ノードは, 少数の隣接する zone に関する情報を持つだけ
(each node only holds information of adjacent zones)

distributed
scalable
fault-tolerant

2. Design
in its most basic form

a virtual d-dimensional Cartesian coordinate system on a d-torus
d次元直交座標系 (トーラス)

全空間はすべてのノードに(長方形に)分割されている
(all the space is divided into rectangular nodes)
Fig 1の例

情報の格納 (Ki, Vi) key-value pair
Ki -(uniform hash function)->a point P
その情報は P を含む領域を持つノードが担当する
(the node in charge of the point P will hold Ki)

Ki に関する検索 (lookup)
Ki -(uniform hash function)->a point P
P を自ノードが含んでいなければ, 隣接ノード経由でルーティング
(if the node does not contain P, route the query
 via the adjacent node)

各ノードは隣接ノードの領域情報とIPアドレスを保持
(each node holds the zone information and IP address
 of adjacent nodes)

2.1 Routing in a CAN
自ノードから P への直線に沿ってルーティング
(route the query along the line from the node to the point P)

d-次元(dimension)のとき
平均検索パス長は (d/4)(n^(1/d))
  ノード数 (# of nodes) n
  各軸方向のノード数 N = n^(1/d)
  (# of nodes along a dimension)
  各軸方向の平均距離 N/4 (トーラスなので, 0..N/2)
  (average distance along a dimension -- because it is torus)
  軸が d 方向
  (average distance in general)
  d * N/4 = (d/4) N = (d/4)(n^(1/d))
各ノードの保持する情報
(# of information held by a node is the number of adjacent nodes)
  隣接ノード数に等しい d*2

目的ノードへのパスは複数存在
(many paths for the target node)
  fail したら次の候補を試せる
  (may try other paths in case of failure)

目的方向への隣接ノードをすべて失った場合
(if all paths for the target fail,)
  しかも repair (2.3で後述) が未完了 (and repair is not complete (explained later))
  "expanding ring search"
    stateless, controlled flooding
    over the unicast CAN overlay mesh
  自分より目的地に近いノードを探す
  (find a node which is nearer to the target than the node)

2.2 CAN construction

新規ノード参加時の座標空間の分割
(divide the zone for new nodes joined)
既存ノードのどれかが自分の割り当てゾーンを半分に分割
(an existing node will give half of its zone)

1. 新規ノードは既存ノードを一つ探す
        (find an initial node)
2. ゾーンを分割してもらうノードを CAN routing で探す
        (find a node to get zone via CAN routing from the initial node)

3. 分割したゾーンの周辺ノードに通知
(inform adjacent nodes of the division of the zone)

Bootstrap
初期接続の問題 (initial connection)
boot strap nodes を DNS で発見する方法
ノードの(一部の)リストを保持 (keep nodes in a list)
その中からランダムに紹介 (select a node from the list)

Finding a Zone
新規ノードは座標空間にランダムに一点 P を選ぶ
(the new node select a point P randomly)
P に対して JOIN リクエストを送出 (bootstrap node 経由)
(send JOIN request to P)
CAN routing を使って P を保持するノードに到着
(the request will arrive at a target node via CAN routing)

P を保持するノードは自分のゾーンを二分割
(the target node divide its zone in half)
二分割する軸の順序を定めておく
(the order of the dimension to divide should be defined)
ノード離脱時の併合
(merging zone at node departure)
分割したゾーンに含まれる(key,value)ペアも渡す
(data contained in the divided zone will be handed)

Joining the Routing
新規ノードは隣接ノードのIPアドレスを得る
(the new node gets IP addresses of adjacent nodes)
分割してくれたノードから
(from the node which handed the zone)
分割してくれたノードも隣接リストを更新
(the parent node update the list also)
隣接ノードに知らせる
(inform adjacent nodes)

ノードは即時の更新メッセージの他に
定期的に自分の領域を隣接ノードに送信
(nodes send the zone information to adjacent nodes periodically)

Fig. 2, 3
ノード 7 が参加したときの変化
(the change after the joining of node 7)

新規ノード参加時の変化は非常に局所的
(the change after node join is very local)
O(d)
総ノード数には無関係
(it is independent of the number of total nodes)

2.3 Node departure, recovery and CAN maintenance

ノードの離脱 (node departure)
それまで担当していたゾーン(とデータ)をだれかに渡す必要
(the leaving node should hand its zone to some node)
通常は隣接ノードのどれかに自分から渡す
(usually to one of adjacent nodes)
ゾーンの併合が可能なら, そのノードに渡す
(if the zone can be merged with some node, hand it to the node)
そうでなければ, 一番狭い隣接ノードへ
(otherwise, to the node with smalllest zone)

故障の場合(ノード, ネットワーク) (failure of nodes, networks)
"immediate takeover algorithm"

隣接ノードのいずれかがとりあげる
ただし, (key,value)ペアは一時的に喪失
(data will be inavailable temporally)

データの本来の持ち主が refresh するまで
(until the original owner of the data will refresh)

故障の検出は定期的な更新メッセージを利用
(detect failures by periodic update messages)

detect failure -> takeover timer 起動
初期値は自分のゾーンの大きさに比例
(initial value of the timer is proportional to the size of its zone)

時間が来たら TAKEOVER メッセージを送信
(when the timer expires, send TAKEOVER message..)
故障ノードの全隣接ノードへ
(to all the adjacent nodes of the failed node)
? どうやって知るのか 自分が隣接してないノードを ?

TAKEOVER を受信したら (on receiving TAKEOVER message,)
ゾーンが自分より小さければ承認
(if the zone is smaller than its own zone, approve the takeover)
自分の方が小さければ, 自分も TAKEOVER 送信
(if its zone is smaller, the receiving node also send TAKEOVER message)

これで, 最小のノードが引き受けることになる
(eventually, the node with smallest zone will takeover
the zone of failed node)

同時に故障が起きた場合 (simultaneous failure)
隣接ノードで到達可能が半分以下になると
inconsistent になる場合あり ??
expanding ring search を行なって
故障ノードのむこう側のノードを得る

ゾーンの細分化 (fragmentation) の防止
background zone-reassingment algorithm (App. A)