A Repair Mechanism for Fault-Tolerance for Tree-Structured Peer-To-Peer Systems

Cédric Tedeschi

WG GRAAL - 24 mai 2006

Outline

Outline

Context

- Resource discovery in grid context
- New needs facing the development of grids
  - large scale
  - no central infrastructure
  - dynamic joins and leaves of nodes
- Adopt peer-to-peer technologies
  - Pure decentralized algorithms
  - Scalable algorithms to retrieve objects
  - Fault-tolerance

Context

- Resource discovery in grid context
- New needs facing the development of grids
  - large scale
  - no central infrastructure
  - dynamic joins and leaves of nodes
- Adopt peer-to-peer technologies
  - Pure decentralized algorithms
  - Scalable algorithms to retrieve objects
  - Fault-tolerance

Context

- Resource discovery in grid context
- New needs facing the development of grids
  - large scale
  - no central infrastructure
  - dynamic joins and leaves of nodes
- Adopt peer-to-peer technologies
  - Pure decentralized algorithms
  - Scalable algorithms to retrieve objects
  - Fault-tolerance

Outline

P2P technologies

- Unstructured P2P approaches
  - flooding based
  - non-exhaustive researches
- Distributed Hash Tables
  - routing based
  - exhaustive search
  - scalable :
    - logarithmic local state
    - logarithmic number of hops
  - fault-tolerance
    - periodic scanning
    - replication
  - drawbacks
    - no locality awareness
    - assumptions of homogeneity
    - only exact match queries
    - replication is costly

P2P technologies

- Unstructured P2P approaches
  - flooding based
  - non-exhaustive researches
- Distributed Hash Tables
  - routing based
  - exhaustive search
  - scalable :
    - logarithmic local state
    - logarithmic number of hops
  - fault-tolerance
    - periodic scanning
    - replication
  - drawbacks
    - no locality awareness
    - assumptions of homogeneity
    - only exact match queries
    - replication is costly

P2P technologies

- Unstructured P2P approaches
  - flooding based
  - non-exhaustive researches
- Distributed Hash Tables
  - routing based
  - exhaustive search
  - scalable :
    - logarithmic local state
    - logarithmic number of hops
  - fault-tolerance
    - periodic scanning
    - replication
  - drawbacks
    - no locality awareness
    - assumptions of homogeneity
    - only exact match queries
    - replication is costly

P2P technologies

- Unstructured P2P approaches
  - flooding based
  - non-exhaustive researches
- Distributed Hash Tables
  - routing based
  - exhaustive search
  - scalable :
    - logarithmic local state
    - logarithmic number of hops
  - fault-tolerance
    - periodic scanning
    - replication
  - drawbacks
    - no locality awareness
    - assumptions of homogeneity
    - only exact match queries
    - replication is costly

P2P technologies

- Unstructured P2P approaches
  - flooding based
  - non-exhaustive researches
- Distributed Hash Tables
  - routing based
  - exhaustive search
  - scalable :
    - logarithmic local state
    - logarithmic number of hops
  - fault-tolerance
    - periodic scanning
    - replication
  - drawbacks
    - no locality awareness
    - assumptions of homogeneity
    - only exact match queries
    - replication is costly

Trie Based Lookup (2/2)

- Range queries
  - automatic completion
  - logarithmic Latency
- Approaches
  - Skip Graphs (complexities)
  - Nodewiz (no fault-tolerance)
  - Prefix Hash Tree (static trie)
  - P-Grid (static trie)
  - locality awareness issue

Trie Based Lookup (2/2)

- Range queries
  - automatic completion
  - logarithmic Latency
- Approaches
  - Skip Graphs (complexities)
  - Nodewiz (no fault-tolerance)
  - Prefix Hash Tree (static trie)
  - P-Grid (static trie)
  - locality awareness issue

Outline

DLPT - original design

- *Distributed Lexicographic Placement Table*
- On-line building of a Greatest Common Prefix Tree
- Mapping
  - DHT-based (load balancing)
  - each physical node maintains one or more nodes of the logical GCP Tree
- Replication based fault-tolerance
- Greedy locality awareness

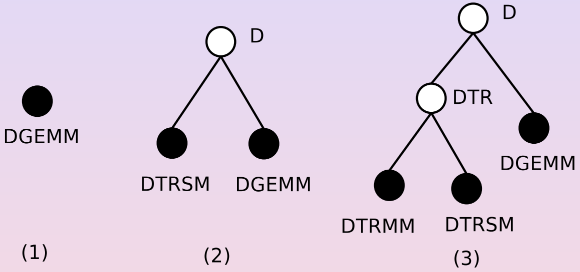
DLPT - logical structure (1/2)

- Alphabet A finite set of letters
- $\prec$ an order on A
- Word w finite set of letters of A, $w = a_1, \ldots, a_i, \ldots, a_l, l > 0$
- u, v two words, uv *concatenation* of u and v
- $|w|$ length of w
- ϵ the empty word, $|\epsilon| = 0$

DLPT - logical structure (2/2)

- $u = prefix(v)$ if $\exists w$ s.t. $v = uw$
- $GCP(w_1, w_2, \ldots, w_i, \ldots, w_n)$ is the longest prefix shared by $w_1, w_2, \ldots, w_i, \ldots, w_n$
- GCP Tree labeled rooted tree s.t.
  - The node label is a proper prefix of any label in its subtree
  - The node label is the Proper Greatest Common Prefix of all its son labels

DLPT - on-line construction



- Contact
- Routing
- Inserting

Routing

- Object o to be inserted
- L set of labels currently in the tree

$$p = max_{|m|}\{m \mid m = GCP(l, o), l \in L\}$$

$$U = \{l \in L \mid GCP(l, o) = p\}$$

- t target label of the routing

$$t = min_{|u|}\{u \in U\}$$

Inserting

- Once the target is found, four cases :
  - $t = o \rightarrow$ insert o on $node(t)$
  - $o = tu$ $(u \neq \epsilon)$
    - new node $node(o)$ son of $node(t)$
    - insert o on $node(o)$
  - $t = ou$ $(u \neq \epsilon)$
    - new node $node(o)$ father of $node(t)$
    - insert o on $node(o)$
  - Default
    - $node(t)$ and $node(o)$ siblings (no father)
    - new node $node(o)$ father of $node(t)$
    - new node $node(GCP(o,t))$ father of $node(t)$ and $node(o)$
    - insert o on $node(o)$

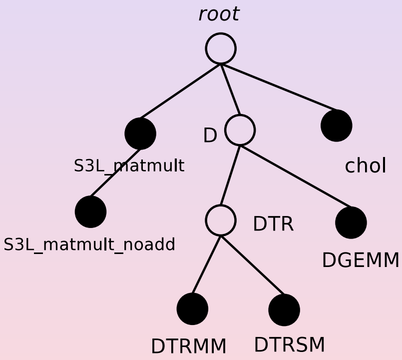
DLPT - Fault Tolerance and locality awareness (1/2)

- Static replication factor k
- Greedy locality awareness
- Periodically initiated by the root
- Replication of the root
- Election of one replica to launch the process in the subtree

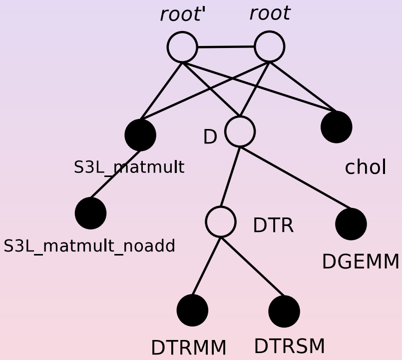
DLPT - Fault Tolerance and locality awareness (1/2)

- Static replication factor *k*
- Greedy locality awareness
- Periodically initiated by the root
- Replication of the root
- Election of one replica to launch the process in the subtree

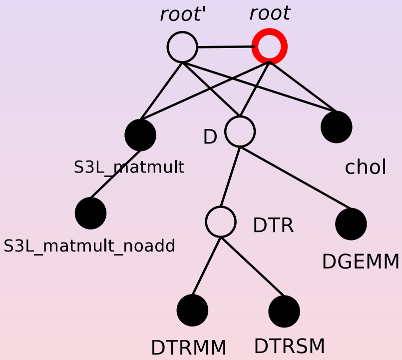
DLPT - Fault Tolerance and locality awareness (2/2)



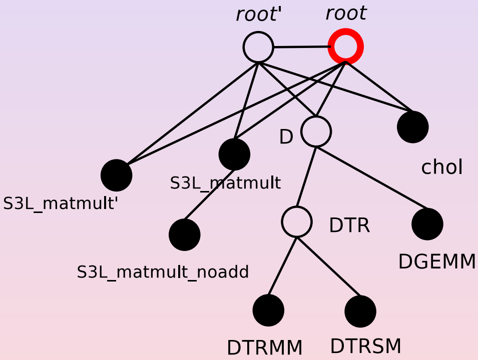
DLPT - Fault Tolerance and locality awareness (2/2)



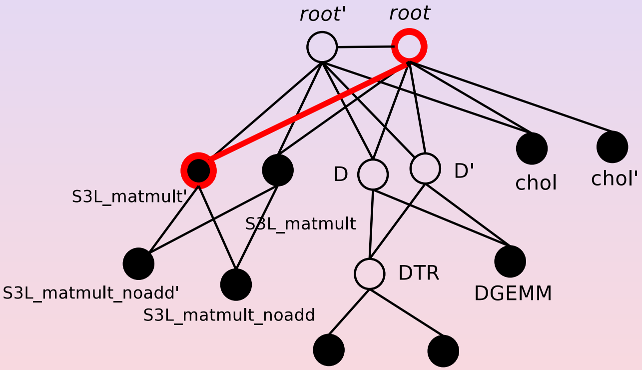
DLPT - Fault Tolerance and locality awareness (2/2)



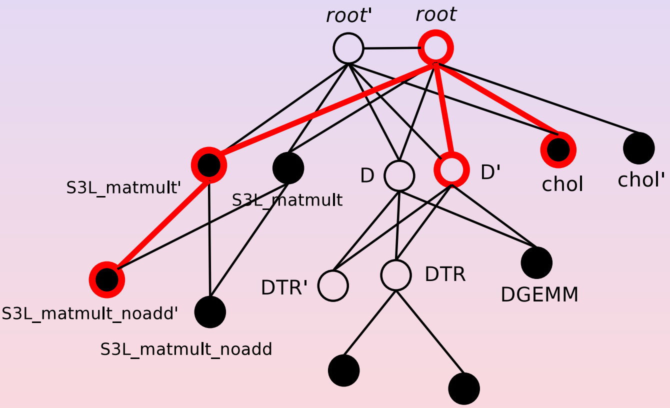
DLPT - Fault Tolerance and locality awareness (2/2)



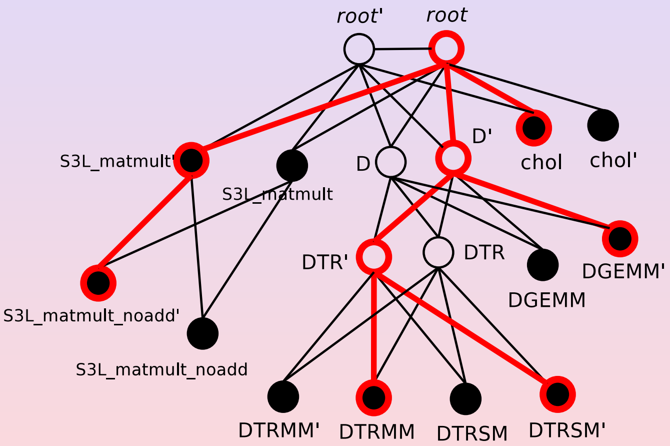
DLPT - Fault Tolerance and locality awareness (2/2)



DLPT - Fault Tolerance and locality awareness (2/2)

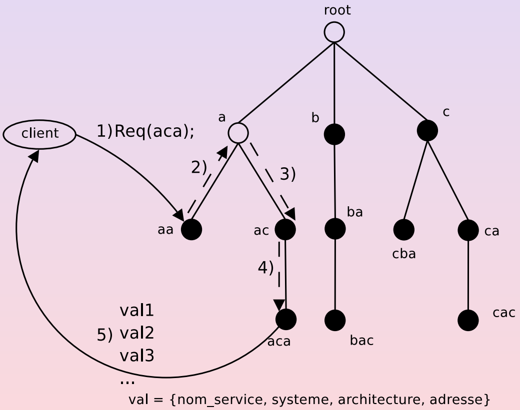


DLPT - Fault Tolerance and locality awareness (2/2)



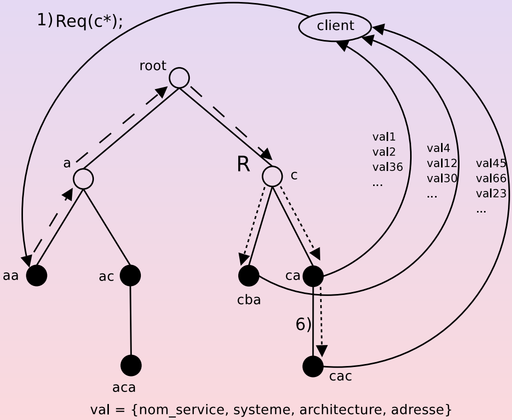
Querying

- Exact match query
- Range query (automatic completion)
- Multicriteria lookup



Querying

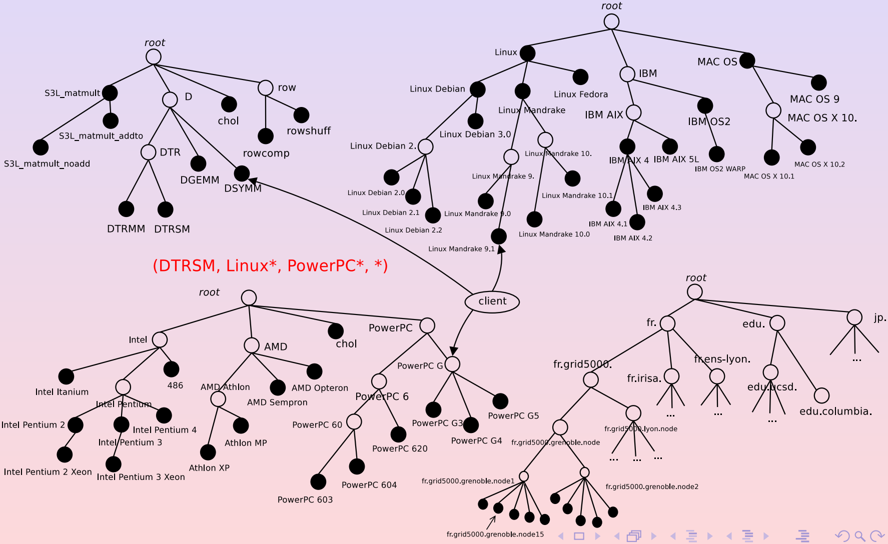
- Exact match query
- Range query (automatic completion)
- Multicriteria lookup



Querying

- Exact match query
- Range query (automatic completion)
- Multicriteria lookup

Querying


root
S3L_matmult
S3L_matmult_addto
S3L_matmult_noadd
D
DTR
DTRMM
DTRSM
DGEMM
DSYMM
chol
row
rowcomp
rowshuff
root
Linux
Linux Debian
Linux Debian 2.
Linux Debian 2.0
Linux Debian 2.1
Linux Debian 2.2
Linux Debian 3.0
Linux Mandrake
Linux Mandrake 9.
Linux Mandrake 9.0
Linux Mandrake 9.1
Linux Mandrake 10.
Linux Mandrake 10.0
Linux Mandrake 10.1
Linux Fedora
IBM
IBM AIX
IBM AIX 4
IBM AIX 4.1
IBM AIX 4.2
IBM AIX 4.3
IBM AIX 5L
IBM OS2
IBM OS2 WARP
MAC OS
MAC OS 9
MAC OS X 10.
MAC OS X 10.1
MAC OS X 10.2
(DTRSM, Linux*, PowerPC*, *)
client
root
Intel
Intel Itanium
486
Intel Pentium
Intel Pentium 2
Intel Pentium 2 Xeon
Intel Pentium 3
Intel Pentium 3 Xeon
Intel Pentium 4
AMD
AMD Athlon
Athlon MP
Athlon XP
AMD Opteron
AMD Sempron
chol
PowerPC
PowerPC G
PowerPC G3
PowerPC G4
PowerPC G5
PowerPC 6
PowerPC 60
PowerPC 603
PowerPC 604
PowerPC 620
root
fr.
fr.grid5000.
fr.grid5000.grenoble.node
fr.grid5000.grenoble.node1
fr.grid5000.grenoble.node15
fr.grid5000.grenoble.node2
fr.grid5000.lyon.node
fr.irisa.
fr.ens-lyon.
edu.
edu.ucsd.
edu.columbia.
jp.
...

Complexities

- N size of the tree
- Assumptions
  - A finite
  - T upper bound on the length of the labels
- Number of hops of routing bounded by $2T$
- Local state bounded by $|A|$
- Local decision of routing in $O(1)$
- (Multicriteria) range query, replication/locality process
  - latency bounded by T
  - linear number of messages

Replicating or repairing ?

- Replication
  - Preventing approach
  - How to tune the replication factor ?
  - Costly to maintain (resources/local state)
- Repair
  - let the tree split into a forest
  - *a posteriori* reconnection and reordering of nodes

Outline

1. Introduction
2. Related Work
3. DLPT
4. Protocol
   - Tree recovery
   - Tree reorganization
5. Conclusion

Two phases

- Tree recovery
- Tree reorganization

Local reconnection

- p detects the lost of its father
- Obtain the set of remaining physical nodes PN (DHT traversal)
- p builds the set of remaining logical nodes N
- p computes the set of nodes in its subtree T
- Choose a temporary father within $N \backslash T$
- If $N \backslash T = \emptyset$, p is the root of the tree
- Drawback : cycles may appear

Breaking cycles

- Temporary father *tf*
- *p* sends a `HELLO` message to *tf*
- On receipt, *tf* forwards the `HELLO` to its own (temporary) father
- Step by step, two possible situations
  - The *real* root is reached (sends a message `NO_CYCLE`)
  - Local *ID* is the *ID* of the initiator
    - A cycle is detected
    - The cycle is broken (leader election)

Correctness proof (1/2)

Assumption 1

If a node crashes at time t, then for every $t' > t$, no crash occurs.

Lemma 1

Under Assumption 1, the recovery protocol terminates, and when this occurs, the system contains one tree only.

Correctness proof (2/2)

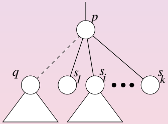
Proof

- By contradiction, assume no node sends a `NO CYCLE` message
- A `HELLO` message never reaches the *real* root
- Every `HELLO` messages traverses only cycles
- When the initiator of a `HELLO` message receives it, a cycle is broken
- Cycles must be infinitely created
- C the number of cycles, each one composed of at least two nodes
- When cycles are broken, at most C/2 leaders reconnects to another tree
- In the next phase, $C' \leq C/2$ reaching 0 (since no other crashes occurs)

Routing the false sons (1/2)

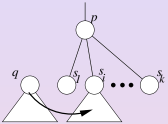
Each node p having a false son q initiates the routing of q
Two cases :

- $q = prefix(p)$, p moves q to its father
- $p = prefix(q)$, four cases.

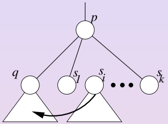


(i) $p.val = prefix(q)$ and $p.val = GCP(s_1, \ldots, s_k)$.

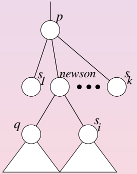
Routing of the false sons (2/2)



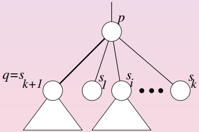
(a) There exists s_i such that $s_i.val = prefix(q.val)$.



(b) There exists s_i such that $q.val = prefix(s_i.val)$.



(c) There exists s_i such that $GCP(q.val, s_i.val) > p.val$.



(d) $p.val = prefix(q.val)$.

Merging (1/2)

- New objects can be inserted during the recovery phase
- A new subtree may have been created at the place of a false root
- Need to merge two trees
- initiated by a `MERGE` message

Tree reorganization

```
upon receipt of <MERGE,fs> from q do
        Gluing(q) ;
        Sorting of p.sons in the lexicographic order in Table t_s ;
        for i = 0 to t_s.length() do
                if t_s[i].val = t_s[i + 1].val
                then send <MERGE,t_s[i + 1]> to t_s[i] ;
                        i := i + 1 ;
                elseif t_s[i].val = prefix(t_s[i + 1].val)
                then send <MOVE,t_s[i + 1]> to t_s[i] ;
                        p.sons := p.sons \ {t_s[i + 1]} ;
                        i := i + 1 ;
                elseif p.val < GCP(t_s[i].val, t_s[i + 1].val)
                then p.sons := p.sons ∪ Newnode(GCP(t_s[i].val, t_s[i + 1].val),
                        t_s[i], t_s[i + 1]) ;
                        p.sons; = p.sons \ {t_s[i], t_s[i + 1]} ;
                        i := i + 1 ;
                endif
        done
```

Correctness proof (1/3)

Lemma 2

Under Assumption 1 and assuming that the system contains one tree only, the reorganization protocol terminates, and when this occurs, the tree is a *GCP* Tree.

Correctness proof (2/3)

Proof (1/2)

If no merging is required. Two cases

1. $p = prefix(fs)$
   - refer to previous figure
   - all cases clearly results in GCP Trees

2. $p \neq prefix(fs)$
   a. $p.father = \perp$
      - $fs = prefix(p)$ fs becomes the root node (GCP Tree)
      - fs and p becomes the two sons of the root node labeled $GCP(p, fs)$ (GCP Tree)

   b. $p.father \neq \perp$, fs is moved to $p.father$
      - eventually reach q s.t. $q = prefix(fs)$ (Case 1.)
      - eventually reach the root of the tree (Case 2.a.)

Correctness proof (3/3)

Proof (2/2)

If merging is required. Four cases

i $\exists s_i, s_j$ s.t. $s_i = prefix(s_j)$

- s_j is moved to s_i
- similar to previous Case 1. (a) and (b) on previous figure

ii $\exists s_i, s_j$ s.t. $GCP(s_i, s_j) > p$

- s_i and s_j sons of a new node $GCP(s_i, s_j)$
- similar to previous Case 1. (c) on the previous figure

iii $\exists s_i, s_j$ s.t. $s_i = s_j$

- recursive merging between s_i and s_j
- solved by induction on s_i and s_j

- $\nexists s_i, s_j$ satisfying either (i), (ii) or (iii) (GCP Tree)

From Lemmas 1 and 2 follows :

Theorem 1

Under Assumption 1, our protocol provide a *GCP* tree reconstruction after the crash of a physical node.

Outline

Conclusion

- Fault-tolerance protocol facing node crashes in a GCP Tree
  - Reconnection and reorganization of subtrees
  - Guaranty of recovering a GCP Tree after a finite time
  - Avoid/coupled with a replication strategy
- Future Work
  - Connecting replication and repair mechanisms to minimize the cost of fault-tolerance
  - Develop and validate a prototype on the Grid'5000 platform