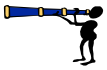
FP-LBS: A Flexible Privacy Enhanced Location Based Group Services System Framework and Practice

Y. Sun, T. F. La Porta

PENNSTATE 1855

Location Privacy vs. Service Flexibility

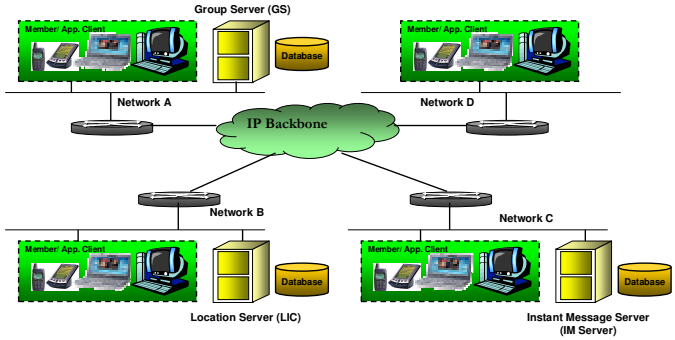


Location Privacy is critical to provide LBS

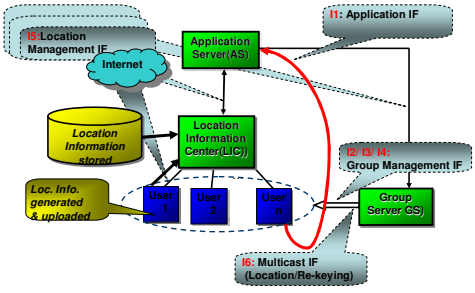
- user controlled location information access
- a group of entities defined by user to access its location information

- Flexible access to location information is important to support desirable service model and easy deployment
  - hierarchical coding of location information with different granularity
  - providing keys to the group members to decrypt location information

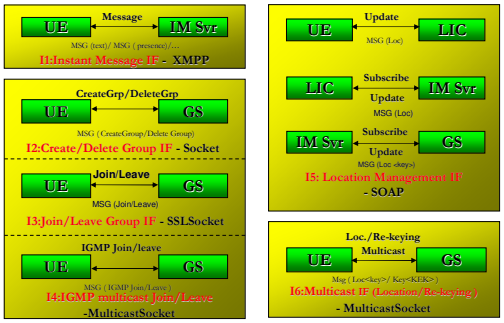
LBGS Network Diagram



LBGS System Interfaces



LBGS Interface Protocols



LBGS Implementation

Implementation platform: Java 2 Platform, J2SE 1.5.0

Group Management Practice:

- *KeyTree* package: LKH extension and 4 key tree schemes support
- *GServer* package: group management server implementation
- *GClient* package: group management client implementation

Application Integration:

- *IMServerExt* package: Wildfire IM Server development package based IM Server extension to support location sharing
- *plugin.locationgroups* package: Spark IM Client development package based client plugin to support location extension

Location Management:

- *LICServer* package: JSOAP develop. package based Location Server
- *Location* package: GPS location tracking / other location tracking

MIKEY and LKH

MIKEY as the basis of keying protocol

Two limitations of MIKEY:

- no hierarchy of group key servers support
- no re-keying support

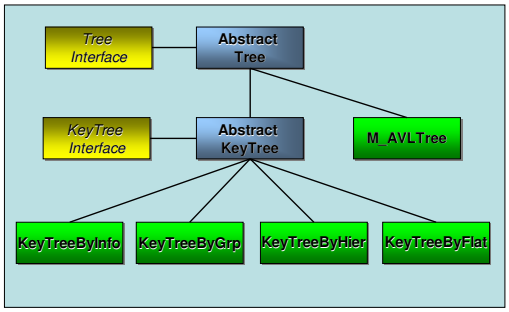
Basic Solution:

- apply LKH to MIKEY to improve scalability with re-keying

Further Extension – 4 schemes:

- Coding by Information Class
- Coding by Group/ Hierarchy Coding / Flat coding

KeyTree Class Hierarchy

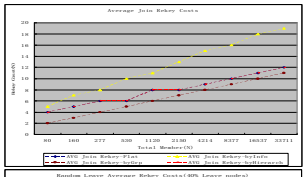


M_AVL Key Tree Algorithm

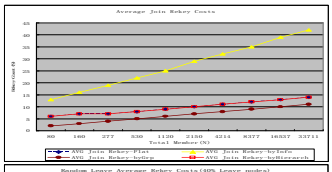
```
/*c is a new member's data */
Algorithm Insert( NodeData c){
1.  if ( tree is empty) {root = createTreeNode(c); return;}
2.  if (tree is full) {
3.      TreeNode e = createTreeNode(c);
4.      TreeNode itnode = createInternalNode();
5.      Current root becomes the left child of itnode;
6.      e becomes the right child of itnode;
7.      root = itnode;
8.  } else {/* if tree is not full*/
9.      TreeNode base = findInsertBase( );
10.     TreeNode e = createTreeNode(c);
11.     TreeNode itnode = createInternalNode();
12.     base becomes the left child of itnode;
13.     e becomes the right child of itnode;
14.     itnode becomes the child of base's parent ;
15. }
16. Update node information along the insertion path;
}

Algorithm findInsertBase(){
1.      TreeNode base = root;
2.      while (base is not leaf node){
3.        if ( (left or both subtrees of base have shallowest node)
4.          { base = base.left; }
5.        else { base = base.right;}
6.      }
7.      return base;
}
```

```
/*n is a member node */

Algorithm Remove( TreeNode n){
1.      if ( n == root) {root = null; return;}
2.      TreeNode base = n.parent;
3.      TreeNode base_gn = base.parent;
4.      TreeNode sn = n.getSibling();
5.      if (gn == root){/*remove a child of the root*/
6.        sn.parent = null;
7.        remove current root and n;
8.        root = sn;
9.      } else {
10.       sn becomes a child of base_gn;
11.       remove base and n;
12.     }
13.     Update node information along the remove path;
14.     rebalance();
}

Algorithm rebalance(){
1.      while ( root is not balance){
2.        /*Find the highest deepest balance subtree*/
3.        TreeNode bn = findHighestDeepestBalanceSubtree();
4.        /*Find the shallowest most unbalance subtree above bn*/
5.        TreeNode nbn = findShallowestUnbalanceSubtree();
6.        swap(bn, nbn);
7.        Update node information along the two paths;
8.      }
}
```

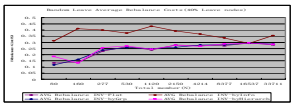
Performance Evaluation



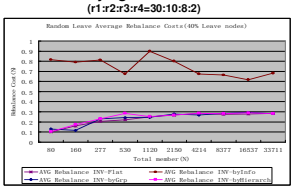
Hierarchical LKH re-keying overhead
(r1:r2:r3:r4=30:10:8:2)



Hierarchical LKH re-keying overhead
(r1:r2:r3:r4 = 2:8:10:30)



Leave re-keying rebalance overhead
(r1:r2:r3:r4=30:10:8:2)



Leave re-keying rebalance overhead
(r1:r2:r3:r4 = 2:8:10:30)

Realistic Application Practice and Analysis

Flexible Extension for Different Application:

Small Group Size(10's)

Sample application - *Instant Message Service*

- Small number of user update/Small number of user subscription
- Small number of re-keying: multicast or /uni-cast

Big Group Size(1000's)

Sample application - *Campus/Disaster/Emergency Service*

- Small number of location update/ large number of user subscription and message multicasts to large audience
- Large number of re-keying: multicast

Medium Group Size(100's)

Sample application – *IM/Game/Company Group Service*

- Medium number of location update/ Medium number of user subscription
- Medium number of re-keying: multicast

Sponsored by IBM