A Rank-Based Skip Lists in Dynamic Provable Data Possession

Juexin Wang
Brown University
wjx@cs.brown.edu

Abstract

This system is designed for data integrity proving at untrusted servers. In cloud storage, the client may not fully trust the server who stores the data, therefore users would like to check if their data has been tampered with. In provable data possession (PDP) model, the client processes the data to get a small metadata before outsourcing it to the storage server. After sending the whole data out, only the metadata is kept by the client, the client will ask the server to prove that the stored data has not been tampered with without downloading the actual data. However, the original PDP scheme applies only to static (or append-only) files.

In the paper [1], Chris et al. have provided a dynamic provable data possession (DPDP) mechanism, which extends the PDP model to support updates to the stored data. Here, we provide an implementation of the DPDP, use a new version of authenticated dictionaries based on rank information. With this rank information together with the digest on the retrieval path, our approach can support insertions, deletions and modifications on the outsourced data while maintain the provability of the data integrity. The evaluation shows that the performance of this implementation follows the theoretical results, the price of updating time and the space is O(log n).

Introduction and Previous Works

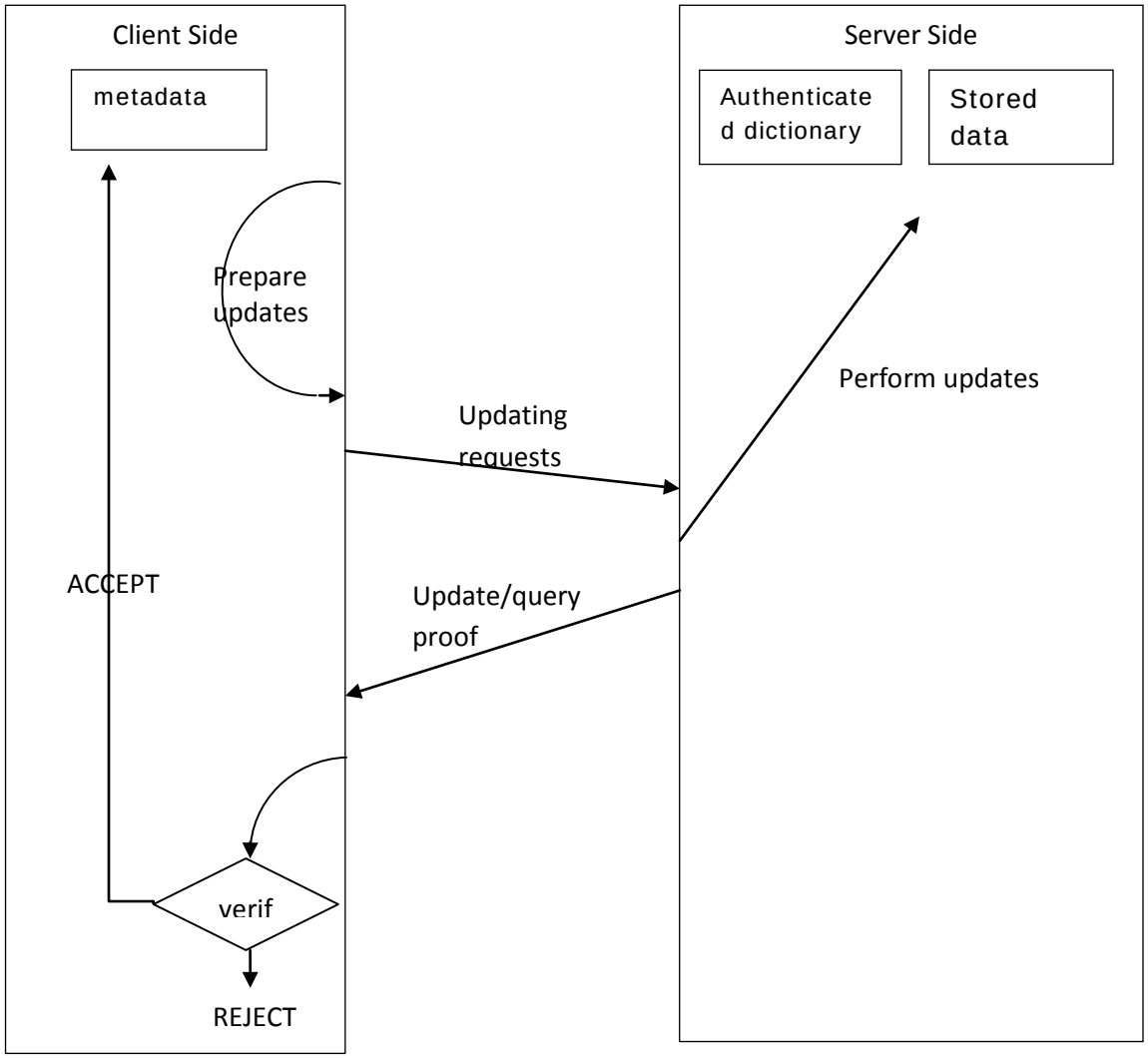
In cloud storage systems, the server that stores the client's data is not necessarily trusted. Therefore, users need a scheme to check if their data is still accurate. However, it's inefficient to download all stored data in order to validate its integrity. Ateniese et al. have formalized a model called *provable data possession* (PDP), it hires cryptographic algorithms, provides probabilistic guarantees of possession, where the client checks a random subset of stored blocks with each challenge. But the PDP and related schemes, apply only the case of static archival storage, at most support the appending at the end of current, not for updating any data block at any position of the data set.

In the paper [1], former members in our group have already generated the idea of a framework of DPDP, which extends from the PDP model to support provable updates on the stored data. Given a file F consisting of n blocks, it defined an update as either insertion of a new block (anywhere in the file, not

only append), or modification of an existing block, or deletion of any block. This solution is based on a new variant of authenticated dictionaries, where uses *rank* information to organize dictionary entries. Thus we are able to support efficient authenticated operations on files at the block level, such as authenticated insert and delete. It theoretically should bring a low price in performance changes while maintaining the same (or better, respectively) probability of misbehavior detection.

Model

Our system consists of two players, the client and storage server. The most unique feature of this system is that it supports the updating on stored data, that is, after clients uploading the original files to the server while keeping the metadata local, the client can send update requests and verify if the server perform this update correctly. The process of these behaviors can be described as follow:



Implementation Details

Rank-based skip list

In order to implement our first DPDP construction, we develop a modification of the authenticated skip list data structure, which we call a rank-based authenticated skip list. We recall that in a skip list, each node v stores two pointers, denoted rgt(v) and dwn(v), that are used for searching. In an authenticated skip list, a node v also stores a label f(v) computed by applying a collision-resistant hash function to f(rgt(v)) and f(dwn(v)). We can use an authenticated skip list to check the integrity of the file blocks. [1]

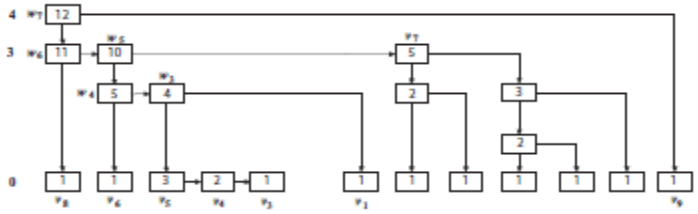


Figure 1: Example of rank-based skip list.

In Figure 1, we show an example of rank-based skip list, each node has a integer value as its rank. This integer value states the number of nodes at the bottom level that can be reached from it. The bottom level nodes represented the data blocks one by one. We call this value the rank of node v and denote it with r(v). An insertion, deletion, or modification of a file block affects only the nodes of the skip list along a search path. We can recompute bottom-up the ranks of the affected nodes in constant time per node.

In implementation, for a n-blocks data set we build n columns of nodes, in each column, some nodes within it is have the effect in the skip list and others not, those affective nodes are logically exist, they are actually counted in the retrieval path, as drawn in Figure 1. Since the top leftmost node of a skip list will be referred to as the start node that can reach to every bottom blocks, we create two special columns which do not refer to any actual data blocks. These two columns are head and tail, with the highest level. The Figure 2 is the same example as Figure 1 but in implementation view. This will help understand the structure.

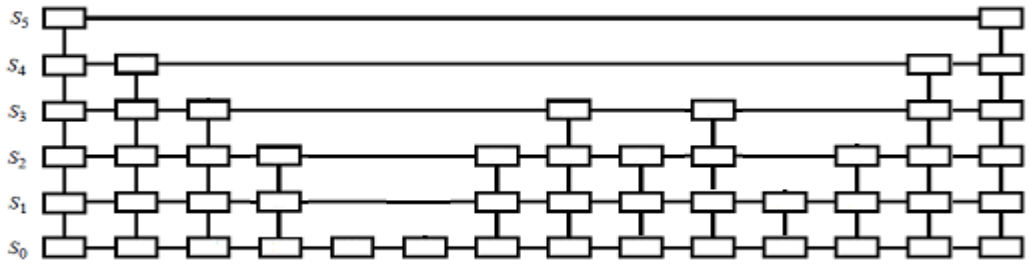


Figure 2: Example of a skip list.

Commutative Hash

Given a collision resistant hash function h, the label f(v) of a node v of a rank-based authenticated skip list is defined as follows. [1]

Case 0: v = null

f(v) = 0 ;

Case 1: l(v) > 0

f(v) = h(l(v), r(v), f(dwn(v)), f(rgt(v))) ;

Case 2: l(v) = 0

f(v) = h(l(v), r(v), x(v), f(rgt(v))) .

Comparing the implementation view and the logic view, we can classify the nodes into plateaus and towers. An element that exists in S_{i-1} but not in Si is said to be a plateau element of S_{i-1}. An element that is in both S_{i-1} and Si is said to be a tower element in S_{i-1}.

In figure 2 we can easily see, if one node's right pointing node is plateau, his attributes(except level) will be same as all nodes who are under it in same column until reaching one whose right pointing node is another plateau. To save space and computation time, we can only record the nodes who have a plateau right neighbor as effect nodes. Then we come to Figure 1.

Retrieval Path

Using the ranks stored at the nodes, we can reach the i-th node of the bottom level by traversing a path that begins at the start node, as follows. For the current node v, assume we know low(v) and high(v). Let w = rgt(v) and z = dwn(v). We set

high(w) = high(v) ,

low(w) = high(v) − r(w) + 1 ,

high(z) = low(v) + r(z) − 1 ,

low(z) = low(v) .

If i [low(w), high(w)], we follow the right pointer and set v = w, else we follow the down pointer and set v = z. We continue until we reach the i-th bottom node. Let $v_k, \ldots, v_1$ be the path from the start node, v_k, to the node associated with block i, v_1. The reverse path $v_1, \ldots, v_k$ is called the retrieval path of block i. Only the effective nodes (described in the last part) should be counted in the path. Figure 3 shows two paths of retrieving two blocks separately.

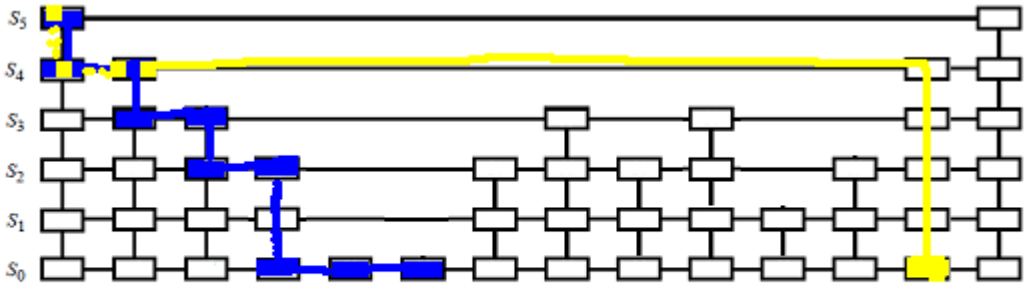


Figure 2: Example of a skip list.

Predict the updated metadata

The algorithms of querying the i'th block and verify the proof of a response are already introduced in [1], this implementation mostly followed these ideas. One more important thing is the client to prepare the update.

In preparing a update, a client should generate a atRank(i-th) query, then use the response together with the current metadata to predict a new metadata. After this preparation, the client then can ask the server to perform the update. At last, the client generate another query and check the metadata by computing from the response to see if the newly computed metadata matches the predicted one. The algorithm 1 and 2 described the prediction of an insertion update and removing update separately.

Algorithm 1: predict the metadata for insert(i-th, level, newItem, metadata).

```
/* parameters :
        int i-th, i.e., the index of the block which the new item will be added after.
        int level, i.e., the level that the new item would be.
        Object newItem, i.e., the item which is being added.
        metadata, i.e., the current metadata.
   Return:
        void
*/

1:      response_i  = atRank(i-th)
2:      if response_i is ACCETPABLE then
3:              proof_i = response_i.getProof
4:              // find the node on the path that need to be modified:  add_point
5:              for all proof nodes pn from size()-1 to 0
6:                      if pn.level >= level && ( pn.next.level < level || pn is tail) then
7:                              add_point = pn
8:              construct a new proof: proof_new, used to proof the new item
9:              for all nodes lower than the add_point
10:                     if its direction is DOWN then
11:                             move this node from proof_i to proof_new's beginning
12:             create the last-bottom node newItemNode  for newItem,  i.e., node referring to the data block.
13:             Add newItemNode to the proof_new's beginning.
```

```
14:                 proof_i[0].neighborRank = (level > 0) ? 0 : 1;
15:                 proof_i[0].neighborLabel = (level > 0) ?
16:                         hash(0, 0):
17:                         hash(0, newItemNode.neighborRank+1, item,  newItemNode.neighborLabel)
18:                 sigmaRank = 0, label = proof_i.blockHash
19:                 for all nodes vi between add-point and proof[0] in proof_i
20:                         sigmaRank += vi.neighborRank
21:                         if its direction is DOWN then
22:                                 label = h(level, sigmaRank, label, vi.neighborLabel);
23:                         else
24:                                 label = h(level, sigmaRank, vi.neighborLabel, label);
25:                 // decide if need to create a new node before add-point in proof_new
26:                 if  (level == add_point.level &&
27:                         ( add-point. direction == DOWN || add_point is head of proof_i) ) then
28:                         change add-point's direction to RIGHT  // must be DOWN before
29:                         add-point.neighborRank = sigmaRank
30:                          add-point.neighborLabel = label
31:                 else
32:                         new_node = node(level, RIGHT, sigmaRank, label)
33:                         add new_node to proof_new's tail
34:                 append nodes in proof_i of index [add-point, proof_i.size()) to proof_new
35:
36:                 sigmaRank = 0, newMetadata = proof_new.blockHash
37:                 for all nodes vi in proof_new
38:                         sigmaRank += vi.neighborRank
39:                         if its direction is DOWN then
40:                                 newMetadata = h(level, sigmaRank, newMetadata, vi.neighborLabel);
41:                         else
42:                                 newMetadata = h(level, sigmaRank, vi.neighborLabel, newMetadata);
```

Algorithm 2: predict the metadata for remove(i-th, metadata).

```
/* parameters :

        int i, i.e., the index of the block which the new item will be added after.
        int level, i.e., the level that the new item would be.
        Object newItem, i.e., the item which is being added.
        metadata, i.e., the current metadata.
    Return:
        int , level of the removed item
*/

1:      response_i  = atRank(i), response_pre = atRank(i – 1);
2:      if response_i  and response_pre are both ACCETPABLE then
3:              proof_i = response_i.proof, proof_pre = response_pre.proof,
4:              // find the lowest node that two path shared:  divide_point
5:              int divide_pre = proof_pre.size -1, divide_i = proof_i.size-1;
6:              for(; index_pre >= 0 && index_i >= 0 ; index_pre--, index_i--)
7:                      if proof_pre[index_pre].direction != proof_i[index_i].direction then
8:                              removing level is current level;
9:                              break;
10:                     if index_pre == 0 then
```

```
11:                                 level = 0;
12:                                 break;
13:
14:             node_i = proof_i[0], node_pre = proof_pre[0]
15:             for all nodes from index 0 to next-to-divide_point in proof_i
16:                     if node_pre goes further than divide_point then
17:                             break;
18:                     else if node_i.level == node_pre.level then
19:                             if node_i is proof_i[0] ||  node_pre.direction == DWN then
20:                                     replace node_pre's fetures by node_i's features
21:                             else if node_pre.direction == RIGHT && node_pre.level < node_i.level then
22:                                     insert node_i into proof_pre at current index of proof_pre
23:                                     divide_pre++
24:                             node_i  = next, node_pre = next
25:                     else if node_i.level < node_pre.level then
26:                             insert node_i into proof_pre at current index of proof_pre
27:                             divide_pre++
28:                             node_i  = next
29:                     else
30:                             node_pre = next

31:             if proof_pre[ divide_pre ] . level == proof_i[ divide_i ] . level then
32:                     replace features' values of proof_pre[ divide_pre ] by the values of proof_i[ divide_i ]'s
33:             else remove node of index divide_pre from proof_pre
34:
35:             // Now use the predecessor path proof_pre to predict the supposed new metadata
36:             sigmaRank = 0, newMetadata = proof_pre blockHash
37:             for all nodes vi in proof_pre
38:                     sigmaRank += vi.neighborRank
39:                     if its direction is DOWN then
40:                             newMetadata = h(level, sigmaRank, newMetadata, vi.neighborLabel);
41:                     else
42:                             newMetadata = h(level, sigmaRank, vi.neighborLabel, newMetadata);
43:             return level
44:     return -1;
```

Guide to run the program

- The authenticated dictionary of class RkASLAuthenticatedDictionary is supposed to be run on the server side, and the implementation of the interface RkClient should be on the client side. All clients who want to work with a ranked based skip list authenticated dictionary should implement the RkClient interface.

- In a run with the basic implementation of RkClient, class RkBasicClient, a instance of the client should first construct a new dictionary on the server, or reference a already-existed one to this client.

- After assigned a RkASLAuthenticatedDictionary to the client, we can call these dictionary's public methods within the client to get some information about the dictionary:

- Int _dict.size(): return the current dictionary's size. It can tell how many data blocks are in the dictionary.
- Boolean _dict.isEmpty(): return true if the dictionary currently has no item.
- Int _dict.level(): return the current level of the skip list. This equals to the highest node in the list.
- Int _dict.rootHash(): return the root hash of the skip list. It creates a copy of the root hash to prevent tampering with the original value.
- Void draw(): This will display the dictionary's contents, visualize the structure, including rank, key and/or hash. level from bottom to top is 0 to maxLevel.

- The client can also call the RkASLAuthenticatedDictionary 's public method atRank() to perform a query for a data block at specific index. The atRank() will return a RkASLAuthenticResponse, then the client should call its method verifyProof() to verify this response to see if it's could be ACCEPT.
- From the client side, users should use insertAndVerify( ) method to insert a data into the Authenticated Dictionary. If it returns a ACCEPT, the data is successfully inserted into the skip list and a new metadata(root Hash) is already passed to the client. This method will automate verify the proof of the update. If provided by an invalid index, the insertion will insert the new item into beginning, if the index is less than 0, or into the end, if the index is larger than the dictionary's size.
- Similarly, users should call the client's removeAndVerify(), modifyAndVerify() methods to perform other kinds of updates to the Authenticated Dictionary, only need to care about the Boolean return.
- If provided by an invalid index, the above 2 methods will directly return false without doing anything towards the dictionary.
- If the proof of an update is not ACCEPT, there are two possible steps that this may occur. The first one is the proof of the first atRank() query is fail, the other one is the updated metadata mismatch the predicted one. In the first scenario, the dictionary (skip list) is secure but in the second scenario the dictionary (skip list) must be already modified.
- If only test the RkASLAuthenticatedDictionary, as the main() function in it shows, after constructing a dictionary we can keep calling the insert() and remove() functions to insert or remove a data block into/from the dictionary to see if there is any failure. Call the draw() when you want to visualizedly check the dictionary.

Evaluations

We evaluated our rank-based skip list DPDP implementation on local environment to test the algorithm and implementation's efficiency regards the networks' interfere. Our test cases are run on PC with 4 core 3.2GHz CPU (AMD Phenom X4 955) and 3.2G memory.

Test case 1:

Do 100 times insertions, removes and lookups separately. Run on a data set of size from 50 to 200000, record the operation time. The inserting position, inserting level are randomly picked up. Also the deletion index, lookup index are also randomly picked up. The result is shown in Figure 3.

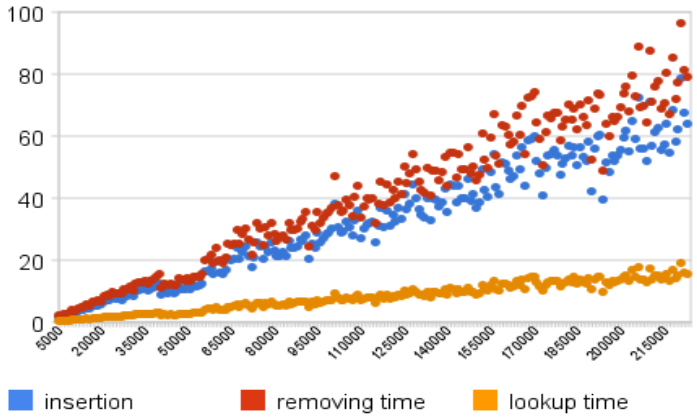


Figure 3: update and look up time

Test case 2:

Do 100 times lookups on a data set of size from 5000 to 200000, record the proof length. The lookup index is randomly picked up. The result is shown in Figure 4.

Theoretically, the update time and the proof size is O(log n), where n is block amount of dictionaries. This implementation

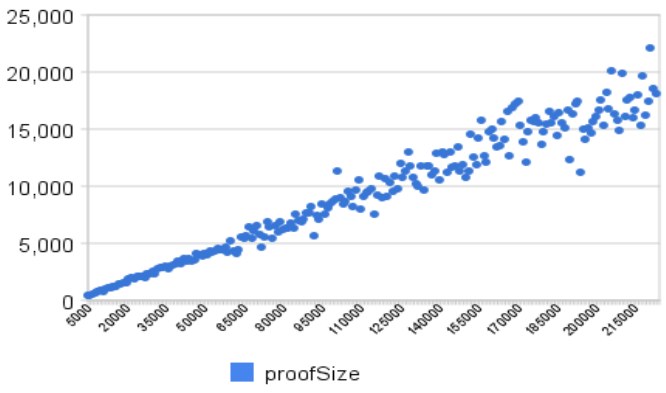


Figure 4: Proof size of each lookup

Classes/Interfaces Details

Package:

stms.authdict.rkasl

Interface/Class:

RkClient

RkBasicClient

RkASLAuthenticatedDictionary

RkASLAuthenticResponse

RkASLAuthResponseEntry

RkASLBasis

1. Class stms.authdict.rkasl.RkASLAuthenticatedDictionary

- Authenticated Dictionary using ranked based skip list.
- The key and the element are the same thing.
- The key in the bottom level are not sorted, their position(index) is specified by inserting parameters
- There are sentinel values used for the head and tail nodes.
- The head node is the starting point for all traversals.
- Contained in both source and mirror dictionaries.
- Based on paper: "Efficient Authenticated Dictionaries with Skip Lists and Commutative Hashing" and "Dynamic Provable Data Possession"

Version:

$Id: RkASLAuthenticatedDictionary.java,v 1.5 2010-05-01

Author:

Juexin Wang
stms@cs.brown.edu

Important Public Methods:

int RkASLAuthenticatedDictionary.size()	Runs in O(1) time. This is due to the fact that we cache the value of the size of the container instead of traversing the list each time there is a query for the size.	
int insert(int pos, Object key, int babyLevel) throws InvalidKeyIndexException, IncompatibleDataException	Runs in expected O(log n) time, where n is the number of elements in the skip list. Creates and inserts a locator with the given height and containing the given object	**Parameters:** pos: position(index) to insert at key: key to insert babyLevel: height of locator to create **Returns:** level of node if successful -1 otherwise. **Throws:** InvalidKeyIndexException -

		IncompatibleDataException - if the object is not of the correct type.
int insert(int pos, Object key) **throws** **IncompatibleDataException,** **InvalidKeyIndexException**	Runs in expected O(log n) time, where n is the number of elements in the skip list. Creates and inserts a locator with a random height containing the given object.	**Parameters:** pos: position to insert at key: key to insert. **Throws:** authdict.api.IncompatibleDataException - if the object is not of the correct type or if we tried to insert an existing key. InvalidKeyIndexException -
Boolean remove(int ith) **throws** **InvalidKeyIndexException**	Runs in expected O(log n), where n is the number of elements. Remove the ith item in the bottom level of the skip list	**Parameters** ith index of the item to remove. **Returns:** true -if successful. False -otherwise. **Throws:** InvalidKeyIndexException
AuthenticResponse **atRank(int ith)**	Runs in expected O(log n) time, where n is the number of elements. Answer the query atRank() described in the paper, return a Response with the item in Byte and its proof	**Parameters:** ith: index of bottom level to retrieval. **Returns:** the authenticated response **Throws:** InvalidKeyIndexException - if the index is out of the bound: (1,size()). ## index starts from 1 authdict.api.NotYetInitializedException - if the object is not of the correct type.
byte[] simpleHash**(**Object x**)**	Hash function for one element. Since we already use the commutative hash, cm, we set: h(x) = cm(x, x)	**Parameters:** **x** the object to be hashed **Returns:** the hash of x
void draw()	Display the dictionary, visualize the structure, including rank, hash, key level from bottom to top is 0 to maxLevel	

2. Class stms.authdict.rkasl.RkASLAuthenticResponse

- Authenticated response. Consist of:
- sub_: the subject
- blockhash: the hash of the item;
- proof: the proof this response, it's a list of RkASLAuthResponseEntry

Version:

$Id: RkASLAuthenticResponse.java,v 1.5 2010-05-01 11:55:15 $

Author:

Juexin Wang
stms@cs.brown.edu

Important Public Methods:

void addProofEntry (RkASLAuthResponseEntry pe)	Add one proof entry into the proof list The order of the entrys in the proof should to carefully match the path	**Parameters:** pe: the proof entry to be added in
ArrayList<RkASLAuthResponseEntry> **getProof()**	return the proof	
boolean validatesAgainst**(**Basis **b)** **throws** NotYetInitializedException	Checks to see that the response is authentic. First, checks to see that the authentication data contains the subject (if subjectContained() == true), or two adjacent elements, one larger than the subject and one smaller (if subjectContained() == false). Then verifies that the sequence of authentication data is correct by using it to recompute the basis. This recomputation involves hashing the elements of the sequence using the commutative hashing function specified by the basis.	**Parameters:** **b** The basis against which to check this response. **Returns:** true iff the response is a valid conclusion from the given basis. Must return false if `isValidatable()` returns false. **Throws:** NotYetInitializedException

3. Interface stms.authdict.rkasl.RkClient

The interface of the client side program. This interface content the methods that a client can call to perform requests to the dictionary that on a server This interface doesn't content the network communication part. So a real "client" class should implement this interface together with others that needed to have full functionility The RkBasicClient.java provide a basic implementation of this interface for testing

Version:
$Id: RkClient.java,v 1.5 2010-05-01 11:55:15 $

Author:
Juexin Wang
stms@cs.brown.edu

Important Contents:

public static boolean ACCEPT = true; public static boolean REJECT = false;		
boolean RkClient.verifyProof(int i, byte[] metadata, AuthenticResponse resp)	verify the proof which contained by a AuthenticResponse. The algorithm is described in the DPDP paper. The client need to store the metadata somewhere	**Parameters:** **metadata** the supposed rootHash. will compute a rootHash from the proof, and compare with it. **i** the index of the data block that challenged/queried **resp** the response come from a request, should contain the T and proof **Returns:** ACCEPT/REJECT
boolean	the entrance of the insertion operation. Prepare the insertion,	**Parameters:** **i** the index the new item

RkClient.insertAndVerify(int i, byte[] metadate, Object newItem, int level)	ask the dictionary on the server to perform this insertion, and finally verify the inserting proof	should be added at; **metadata** the current rootHash, used to predict a new rootHash; **newItem** **level** **resp** the response get from atRank() to verify the accuracy. **metadate** **Returns:** ACCEPT/REJECT
boolean RkClient.removeAndVerify(int i, byte[] metadate)	the entrance of the removing operation. Prepare the deletion, ask the dictionary on the server to perform this deletion, and finally verify the deletion proof	**Parameters:** **i** the index the new item should be deleted; **metadata** the current rootHash, used to predict a new rootHash; **resp** the response get from atRank() to verify the accuracy. **metadate** **Returns:** ACCEPT/REJECT
boolean RkClient.modifyAndVerify(int i, byte[] metadata, Object newItem, AuthenticResponse resp)	the entrance of the modify a block. the modify should remove the old block, and insert a new one at same position with same level.	**Parameters:** **i** the index the new item should be added at; **metadata** the current rootHash, used to predict a new rootHash; **newItem** **resp** the response get from atRank() to verify the accuracy. **Returns:** ACCEPT/REJECT

4. Class stms.authdict.rkasl.RkASLAuthResponseEntry

The proof of a data block is a list of objects of this Entry. Each this entry consists of the informations of one node which on the retrieval path: [level, direction, neighbor_rank, neighbor_label] , Together with:

- _exist: if this node is still exist in the updated path
- _atHead: if this node is on the Head AuthenticatedDictionary.ASLLocator

@ Date: 2010-04-15
Author: wjx@cs

5. Class stms.authdict.rkasl.RkBasicClient

A basic implementation of the interface RkClient.

Achieved the methods of insert, remove, modify the blocks in a dictionary and verufy the response's proof.

Version:
$Id: RkClient.java,v 1.5 2010-05-01 11:55:15 $

Author:
Juexin Wang
stms@cs.brown.edu

Important Contents:

byte[] RkBasicClient._metadata	The metadata (root hash) of a dictionary which is associated with this client	
byte[] RkBasicClient._tempMetadata	The predicted metadata (root hash) of a dictionary which is being updated. If finally the proof is accept after this modification, the old metadata will be replaced by this tempMetadata	
boolean RkBasicClient.insertAndVerify(int i, byte[] metadata, Object newItem, int level) Specified by: insertAndVerify(...) in RkClient	Implementation of the abstract method in the interface RkClient.java. In this implementation, it will call the insert() function to perform the action, and call verify() to verify the proof.	**Parameters:** **i** the index the new item should be added at; **metadata** **newItem** **level** **Returns:** ACCEPT/REJECT
boolean RkBasicClient.insert(int i, Object newItem, byte[] metadata, int level)	Private inner method that ask the dictionary to perform the insertion. Before actually insert, this function will predict a new metadata. After insertion, the actual metadata should match with this(check in insertAndVerify() function)	**Parameters:** **i** the index the new item should be inserted at **newItem** the key to be inserted **metadata** the current rootHash, used to predict a new rootHash **level** client decide to insert the new item into which level
boolean RkBasicClient.removeAndVerify(int i, byte[] metadata) Specified by: removeAndVerify(...) in RkClient	Implementation of the abstract method in the interface RkClient.java. In this implementation, it will call remove() function to perform the	**Parameters:** **i** the index the new item should be deleted; **metadata** the current rootHash, used to predict a new rootHash;

	action, and call verify() function to verify the proof.	**resp** the response get from atRank() to verify the accuracy. **Returns:** ACCEPT/REJECT
boolean RkBasicClient.remove(int i, byte[] metadata)	Private inner method to ask the dictionary to perform the deletion. Before actually insert, this function will predict a new metadata. After deletion, the actual metadata should match with this(check in removeAndVerify() function)	**Parameters:** **i** the index the new item should be deleted; metadata: the current rootHash, used to predict a new rootHash; **metadata**
boolean RkBasicClient.modifyAndVerify(int i, byte[] metadate, Object newItem, AuthenticResponse resp)	Implementation of the abstract method in the interface RkClient.java.	
boolean RkBasicClient.verifyProof(int i, byte[] metadata, AuthenticResponse resp) Specified by: verifyProof(...) in RkClient	Implementation of the abstract method **verifyProof()** in the interface RkClient.java. The algorithm is described in the DPDP paper.	**Parameters:** **i** **metadata** **resp** **Returns:** ACCEPT/REJECT
RkBasicClient.RkBasicClient()	Constructor Use the DEFAULT_MIN, DEFAULT_MAX, DEFAULT_MAX_LEVEL, CommutativeHash ch, Comparator comparator to construct a new Dictionary for this client while construct the client itself	

References

[1]. C. Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, Roberto Tamassia. **Dynamic Provable Data Possession.** *CCS'09,* November 9–13, 2009, Chicago, Illinois, USA.

[2]. Michaelt. Goodrich, Roberto Tamassia. **Efficient Authenticated Dictionaries with Skip Lists and Commutative Hashing.** January 13, 2001.

[3]. Charalampos Papamanthou, Roberto Tamassia. **Time and Space Efficient Algorithms for Two-Party Authenticated Data Structures.**