BROWN UNIVERSITY

Implementation of Methods for Distributed Data Authentication

---

Project Report

Student: Qiao Xie **Email:** qiao.xie.55@gmail.com

Advisor: Nikos Triandopoulos **Email:** nikos@cs.brown.edu

May 15 2010

Abstract

We build this project based on the paper “Efficient Content Authentication in Peer-to-peer Networks” by Roberto Tamassia and Nikos Triandopoulos[1]. The project is an implementation of the authentication model presented in the paper. The main component is an efficient construction of a distributed Merkle tree (DMT). It is built on top of an open source distributed hash table (DHT) “OpenChord” [2][3] using only the basic APIs – get(key) , put(key,data) and remove(key). The DMT provides secure version of the above APIs to enable data verification. It also has the properties of a BB(alpha) tree, where rotations often happen in the lower level of the tree. We improve the performance by caching key-node mappings, storing redundant information and using concurrent retrieval.

Introduction

Distributed P2P storage networks are becoming the most popular data storage solution. Traditional authentication models need to evolve to adapt the changes. This project implements the first distributed version of Merkle tree, which is described in the research paper mentioned in “Abstract” section.

The authentication model consists of:

1. A trusted data source which maintains a data set in the DHT (put and remove data items).
2. An untrusted distributed P2P network which provides the put, get and remove functionalities.
3. Users who issues queries and retrieve data items in the data set.

The DMT provides the secure version of basic APIs: auth_put, auth_get and auth_remove. The data source uses auth_put/auth_remove to store/remove data in the DHT. Users use auth_get to retrieve data from DHT.

We implement two versions of DMT: the ordered and the unordered

In the ordered version, all the data items are sorted by a search key. This feature is necessary for providing negative proofs, which is the proof of the non-existence of a data item. However, to maintain this order, rotations are needed in the auth_put function. The unordered version cannot provide negative proofs but has better performance because no rotation is needed when putting data into the DHT.

Project Overview

This research project can be divided into the following milestones:

1. Study different open source Distributed Hash Table systems and choose one as the authentication model's P2P network

2. Implement the basic BB(alpha) tree to verify insert and delete algorithms

3. Implement the DMT using APIs provided by Openchord

4. Implement the authentication model on top of DMT

5. Implement the caching functionality

6. Implement the ordered DMT to provide negative proofs

7. Project demo on local machine and department network

1. Distributed Hash Table (DHT)

The authentication model in the paper can be implement on any DHT that provides put(key,data), get(key) and locate(key) APIs. In our implementation, we select OpenChord (http://open-chord.sourceforge.net/), which is an open source implementation of Chord, as the underlying storage system.

Openchord can create a local virtual DHT network as well as a real DHT network using the socket protocol. For a virtual DHT, each node is a separate thread in memory. We can create as many nodes as we want if there is enough memory. For a real DHT, each node is identified by an IP address and a port. It takes up physical network resources and listens on the ports.

1.1 The main APIs of OpenChord

public final Set<Serializable> retrieve(Key key)

Parameters:
key: the key of key-value pair, associate with a specific data set

Description:
This is similar as the get(key) API in a standard DHT.
OpenChord is different from standard DHT in that it associates a key with a set of data items.
But in our implementation, we do not need this extension, therefore we always store one data item in the set.

public final Set<Serializable> retrieve_with_cache(Key key)

Parameters:
key: the key of key-value pair, associate with a specific data set

Description:
This is the get(key) API with the caching feature.
Caching is implemented by remembering key<-->node_identifier mappings and query it before performing the findSuccessor() method.

public final Node findSuccessor(ID key)

Parameters:
key: The ID of a key is determined by inputting the key into a hash function.
In our implementation, i.e.
ID id = this.hashFunction.getHashKey(key);

Returns:

Returns the node_identifier (socket or local virtual node identifier) that stores data items associate with a particular key.

Description:
Similar as the locate() function in standard DHT.
Used by retrieve_with_cache() to determine the responsible node when a cache miss happens.

public final void remove(Key key, Serializable s)

Parameters:
key, s: key-value pair for deletion

Description:
Because in OpenChord, one key can associate with multiple data items (although we always use one in our implementation),
we need to explicitly specify the data item to remove.

1.2 Set up a DHT network using OpenChord

Deploying and using OpenChord is quite easy. It consists of creating a bootstrap node and join other peer nodes to it.

Three steps to set up a DHT network:

1. Choose the communication protocol

Two protocols to choose from:

a. Local protocol to create virtual nodes on one local machine:
String protocol = URL.KNOWN_PROTOCOLS.get(URL.LOCAL_PROTOCOL);

b. Socket protocol to create physically distributed nodes over the network:
String protocol = URL.KNOWN_PROTOCOLS.get(URL.SOCKET_PROTOCOL);

2. Create the bootstrap node

The bootstrap node is the first chord node in a DHT network.
It provides initial configuration information to newly joining nodes so that they may successfully join the DHT.
All other nodes that want to join the DHT need to connect to this node.
In order to improve fault tolerant, some DHT maintain multiple bootstrap nodes via DNS service.
However, in OpenChord, there is only one bootstrap node whose address is fixed.

Therefore if the bootstrap node crashes, no new node can join the DHT until it comes up again.

Example of creating a bootstrap node:

```
bootstrapURL = new URL(protocol + "://192.168.100.3:10000/");   //specify the node address (IP + port number)
Chord chord0 = new de.uniba.wiai.lspi.chord.service.impl.ChordImpl(); //create a Chord instance
chord0.create(bootstrapURL); //create the bootstrap node
```

3. Join other nodes to the DHT

Once the bootstrap node is on, other nodes can connect to it and join the DHT.

Example of joining a bootstrap node (on another physical machine)

```
URL localURL = new URL(protocol + "://192.168.100.4:10001/"); //specify the local node address
bootstrapURL = new URL(protocol + "://192.168.100.3:10000/");   //specify the bootstrap node address
Chord chord1 = new de.uniba.wiai.lspi.chord.service.impl.ChordImpl(); //create a Chord instance
chord1.join(localURL, bootstrapURL); //join the DHT network
```

See details in the OpenChord manual and the test cases in Tests package.

2. Distributed Merkle Tree (DMT)

The DMT is the major component of this project. It is a combination of BB(alpha) tree and merkle tree.

There are two versions: unordered version and ordered version.
The basic version is simpler and faster (has less rotations comparing to ordered version).
The ordered version can provide negative proof (proof of the nonexistence of a data item) and search ability based on search key.
The biggest difference between these two version lies in the insert() function.
In the basic version, when inserting a data item, we always follow the branch that has a smaller weight. Therefore no rotation is needed during insertion.
In contrast, for the ordered version, we need to follow the search keys and insert the data item to the correct sequential position. Therefore rotations may happen during insertion.

The tree nodes are distributively stored over the DHT network. Each node is identified by a key.
In terms of tree traversal, the DMT is less efficient than local merkle tree and sensitive to network latency since each tree node may reside on a different physical node.
In our implementation, we use redundant storage and caching to reduce network communications.

For a inner node, we store the keys of itself, its right child, left child and parent. Keys are like pointers in a non-distributed tree.
Given a key, to retrieve the associated data, a locate(key) operation and a retrieve(key) operation are needed.
The locate(key) function is called by retrieve(key) function to determine which node has the data. With caching, the locate(key) operation can be skipped.
Caching uses a HashMap data structure in memory to remember key<-->Node mappings to avoid communication with network nodes.

We store three hash values: its hash value, left child's hash value and right child's hash value. This redundancy reduce the number of nodes we need to contact when retrieving hash values along the proof path.

2.1 Important Classes of DMT

DMT.inner_data
This is the data structure for inner nodes of the DMT. It stores hash values and structural information.Each inner node associates with a key, it also stores the keys of its left child, right child and parent.

DMT.leaf_data
This is the data structure for leaf nodes of the DMT. It stores data items and proof path information.Each leaf node associates with a key, it also stores the keys of its parent and all inner nodes in its proof path.

The build_proof_path(Chord node_to_connect) function of leaf_data is used to create/update proof path information stored in a list at the leaf.It goes up from leaf to root and add all keys to the list one by one.

DMT.Tree / DMT.Tree_Ordered (ordered version)
The main difference between ordered version and unordered version lies in insertion.
The Tree class is used to create/delete inner/leaf nodes in the DHT. The instances themselves do not store node data.
Tree nodes are distributively stored in the DHT and from any node we can follow the keys and traverse the whole tree (keys serve as pointers).

Important Data Structures:

private StringKey node_id
Key of the tree node, a unique identifier.

private static Integer inner_node_id
Used by the Tree class constructor to generate unique tree node keys, initially set to 0.

private static double alpha
This is the alpha factor for our BB(alpha) tree. It is used by checkRot() to test whether a rotation is needed.
ratio = left_weight/total_weight
If ratio is less than alpha or greater than (1-alpha), rotation is needed to restore balance.
The valid range of alpha is: 0<alpha<0.5, bigger alpha will result in more rotations during insertion and deletion.

public static int rotation_count

This global variable keeps track of the number of rotations performed during insertion and deletion. It is used during debug and testing.

private static KeyPair keyPair

This is the public-private key pair used to sign and verify root hash.
The key pair is set by calling setKeyPair(). The key pair must be set before performing any insertion or deletion.
The private/public key is retrieved by getPrivate()/getPublic()

public static StringKey root_key

This global variable keeps track of the key of tree root. It is set by setRootKey().
Initially, it is the first inner node created by the data source. The tree root may change during rotations.

Important Methods:

public boolean insert(Chord chord_node, StringKey tree_node, StringKey leaf_key, leaf_data leaf_data)

Parameters:
chord_node: The chord node to connect (can be any active node in the DHT) for inserting data
tree_node: The tree node to perform insert
leaf_key, leaf_data: key-value pair for insertion

Returns:
Returns whether the insertion success or not.
If data is tampered or network is unreachable, it will return false or throw exception.

Description:
The major component of authPut(). The data source uses authPut() to insert key-data pairs into the DHT.
Verification is performed before insertion. This function is used by the unordered DMT.
Through recursive calling, the insertion always happens in the less heavier subtree.

public boolean insert_rot(Chord chord_node, StringKey tree_node, StringKey leaf_key, leaf_data leaf_data)

Parameters:
Same as insert()

Returns:
Same as insert()

Description:
This is the ordered version of insert(). The tree nodes are ordered by "search_key" in the tree node data structure.
This procedure can be roughly divided into four steps:
  1. Travel down the tree following search_keys to locate the leaf node for inserting new data.
2. Retrieve the proof of that leaf node and verify it.
3. Insert the new data and update the hash values.
4. Call checkRot() to check every node from leaf to root and perform rotation when necessary.

public boolean delete_rot(Chord chord_node, StringKey delete_key)

Parameters:
chord_node: The chord node to connect
leaf_key: The key associate with the data to delete

Description:
The major component of authRemove(). The data source uses authRemove() to remove key-value pairs
Verification is performed before the actual delete operation. If the data to delete is tampered, it will return false.

public boolean delete(Chord chord_node, StringKey delete_key)

Description:
This delete() do not perform any rotation and leave the DMT in a unbalanced status. It can be used in tests as a comparison to delete_rot().

public boolean auth_get(StringKey leaf_key, Chord chord_node)

Parameters:
chord_node: The chord node to connect
leaf_key: The key of data to retrieve

Returns:
Returns whether the retrieval success or not.
If data is tampered or network is unreachable, it will return false or throw exception.

Description:
The data source and client uses this function to retrieve data items from DHT.
Verification is performed before returning the data. f the data retrieved is tampered, it will return false.

public StringKey checkRot(Chord chord_node, StringKey node_key)

Parameters:
chord_node: The chord node to connect
node_key: The tree node (must be an inner node) to check balance.

Description:
The function is called by insert_rot() and delete_rot() to check whether rotations are needed from bottom up.
Verification will be performed before the actual delete operation. If the data to delete is tampered, it will return false or throw exception.
It computes the ratio and compare it with the alpha variable (a global variable set in advance) to determine whether a rotation is needed and performs the rotation.

public void sign_and_store_rootHash(Chord chord_node, String root_hash)

Parameters:
chord_node: The chord node to connect
root_hash: The root hash (computed by data source) to be stored in the DHT

Description:
Sign and store the root hash in the DHT.
This function is called by insert(), insert_rot() and delete_rot() to update the root hash in the DHT.

public String retrieve_rootHash(Chord chord_node)

Parameters:
chord_node: The chord node to connect

Returns:
Returns the root hash stored in the DHT

Description:
The signed root hash is stored in the DHT under the key "rootHash". This function retrieves and verifies it using the locally stored public key.
It is called by insert(), insert_rot(), delete_rot() and auth_get() to retrieve the root hash from DHT and compare it with the locally computed root hash.

public String compute_current_rootHash(LinkedList<HashStruct> proof_path)

Parameters:
proof_path: Contains a list of hash values and structural information needed to compute the root hash

Returns:
Returns the computed root hash

Description:
This function cooperates with hashValues_retrieve() to retrieve hash values and structural information needed from DHT and compute the root hash.

public LinkedList<HashStruct> hashValues_retrieve(StringKey leaf_key, Chord chord_node)

Parameters:
chord_node: The chord node to connect
leaf_key: The key corresponding to the data to verify

Returns:
Returns a LinkedList data structure which contains all information needed to compute root hash

Description:
This function contacts all the tree nodes from leaf to root (the proof path) and retrieve necessary information for computing root hash.
The keys of the tree nodes in the proof path is stored in the leaf data structure along with the actual data.
This allows the function to retrieve information concurrently from the DHT.

We do not store the actual information in leaf nodes because the update cost will be very high during insertions and deletions.

public void proof_path_rebuild(Chord chord_node, inner_data innerNode)

Parameters:
chord_node: The chord node to connect
innerNode: The root of the affected subtree, all its descendant leaves will rebuild their proof path

Description:
The "LinkedList<StringKey> proof_nodes" data structure in the leaf stores a set of keys correspond to the tree nodes in the proof path.
When the structure of the tree changes, all leaves in the affected subtree needs to be updated for the new proof path.

public void printTree(Chord node_to_connect, StringKey root)

Parameters:
chord_node: The chord node to connect
root: The key of the root node

Description:
This function traverse the tree and prints out useful information (hash values, structural information, etc) for debugging and testing.

public void data_tamper(StringKey tamper_key, Chord chord_node)

Parameters:
chord_node: The chord node to connect
tamper_key: The key of the leaf data to tamper

Description:
This function simulates the action of a malicious DHT node tampering the data items store on it.
It connects to a leaf node, retrieves the corresponding data and sets its value to a fraud one.

public void weight_tamper(StringKey tamper_key, int weight, Chord chord_node)

Parameters:
chord_node: The chord node to connect
weight: The weight will be set to this value
tamper_key: The key of the inner data to tamper

Description:
This function has similar behavior as data_tamper().
It connects to a inner node, retrieves the corresponding data and sets its weight to a fraud one (specified by the weight parameter).

Future Work

Future work includes the following directions:

1. Deploy the project on a large scale distributed P2P network.

The current experiment environment is 1~5 linux machines in the CS department.

The main purpose of current tests is to verify the correctness of the authentication model's functionalities. The next step would be deploy the project on a geographically distributed network and collect running results from more than 100 machines.

2. Extend OpenChord to make the data stored in node persistent or choose another DHT that has this feature.

OpenChord is easy to use and deploy. However, the data items are stored in memory and will disappear when a node go down. We plan to extend OpenChord's API to support data items in files on disk. Also it will be more interesting to tamper the data item directly on disk files than to simulate it using the API provided by OpenChord.

References

[1] Tamassia, R. and Triandopoulos, N. 2007. Efficient Content Authentication in Peer-to-Peer Networks. In Proceedings *of the 5th international Conference on Applied Cryptography and Network Security,* 354-372.

[2] http://open-chord.sourceforge.net/

[3] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications* (San Diego, California, United States). SIGCOMM '01. ACM, New York, NY, 149-160.