COL106 : 2021-22 (Semester I)
Project: Module 3

Chirag Bansal Venkata Koppula Akshay Mattoo

September 11, 2021

Notations

For two strings a, b, we denote $a.\texttt{concat}(b)$ by $a + b$ (that is, if $a =$ "Hello", and $b =$ "World", then $a + b =$ "HelloWorld"). For any natural number n, $[n]$ denotes the set $\{1, 2, \ldots, n\}$.

Important: There is one compulsory exercises (Exercise 2.1), which has two sub-problems that are marked (♠) as lab submission problems. The lab-submission problem is to be submitted via Moodle by 11:59PM on the day that you have your lab.

1 Introduction

In the last module, we saw authenticated lists. In this module, we will introduce another important object for building a cryptocurrency — *Merkle Trees*. First, let us consider a toy scenario to understand what are Merkle trees, and why they are needed.

1.1 Toy Scenario

Suppose you are a user (*client*) who has a large number of documents say m_1, m_2, ... m_n (assume each document can be represented as a string, and n is a power of 2). You want to safely store all these documents on a server such that later, you can query the server to send you back m_i for any $i \in [1, n]$. **Additionally, you want to ensure that the document returned by the server has not been tampered with, and is the same one you had uploaded earlier**.

The solution to this seems straightforward - we could use an Authenticated List from the last assignment, and use this to store the data on the server. Before storing the n documents to the server, the client will compute all the n digests (as discussed in Module 2), and store the last digest $\texttt{dgst}_n$ as a `summary` of the entire list (after the list is uploaded on the server, the client will delete/forget the n documents, and keep only the `summary`). Each of the n nodes on the server's list would contain a document m_i (let us call it `data`) along with a short digest $\texttt{dgst}_i$.

Pause here and think - when the client queries for the i^{th} item, the server can send m_i. How can it convince the client that this is indeed the i^{th} document that the client uploaded?
Hint: The client stores `summary`*, which is equal to* $\texttt{dgst}_n$.

Hopefully you figured out the procedure for ensuring that m_i is indeed the i^{th} document. However, **this procedure will require time which is linear in the length of the entire list - this could be huge if we have a large number of documents!**

A Better Solution: To address the above problem, we use a special structure called *Merkle Tree*.[1] This is a binary tree where each leaf node stores a document and each non-leaf node stores a `dgst`. Since we have assumed n to be a power of 2, this will be a complete binary tree of depth $d = \log(n)$.

Let us denote each node by $N_{i,j}$ where $i \in [0, d]$ is the level of the node (the root is at level 0) and $j \in [1, 2^i]$ is the index of the node at level i. So, the root is designated as $N_{0,1}$ and the leaf nodes would be $N_{d,1}$, $N_{d,2}$ ..., $N_{d,n}$. At node $N_{i,j}$, we store a string $\texttt{val}_{i,j}$, which can be computed recursively as shown below -

$$\texttt{val}_{i,j} = \begin{cases} m_j & \text{if } i = d \\ \texttt{CRF.Fn}(\texttt{val}_{i+1,2j-1} + \text{“\#”} + \texttt{val}_{i+1,2j})) & \text{otherwise} \end{cases}$$

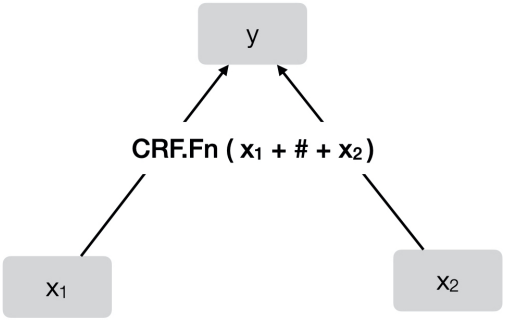

Figure 1: The value at the parent node is computed using `CRF.Fn` on the values of the children nodes. Note that the ordering of left child and right child is important here.

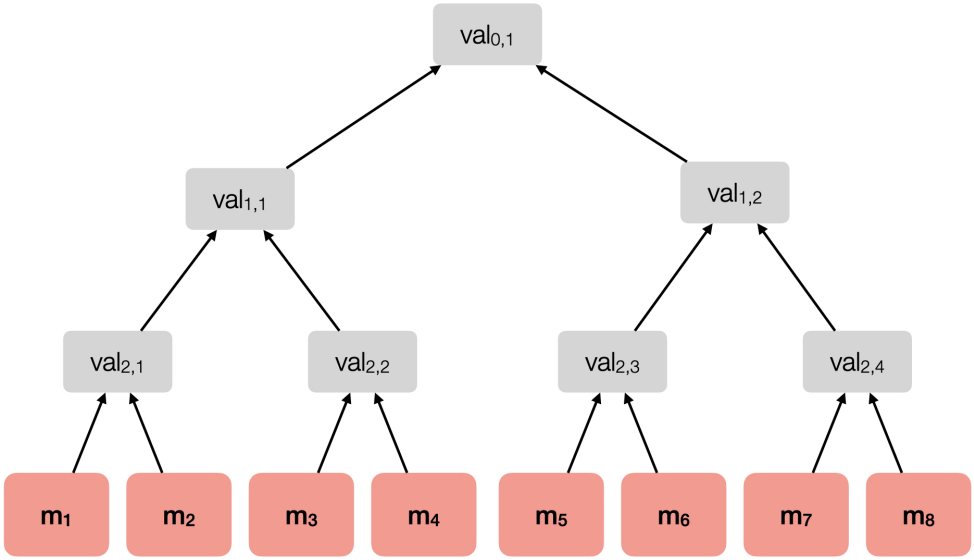

Figure 2: Bottom-up calculation

The client stores the final $\texttt{val}_{0,1}$ (that is, the string at the root node) as its `summary`. Next, we will discuss how this tree-structure helps the server prove that m_j is indeed the j^{th} document, and how the client can use `summary` to ensure that the server is not cheating.

[1] Read more about a Merkle Tree here

Proof of membership : Recall, the server wishes to prove that m_j is the element at j^{th} position, and the client will check this proof using the `summary` that it maintains. To define this proof of membership protocol, we need the the following notions:

- *Sibling of a node*: Given any node $N_{i,j}$ at level $i > 0$, the sibling of $N_{i,j}$ is the unique node N (not equal to $N_{i,j}$) at level i such that N and $N_{i,j}$ have the same parent node. If j is even, then the sibling of $N_{i,j}$ is $N_{i,j-1}$, else it is $N_{i,j+1}$. For any $i \in [d]$ and $k \leq 2^{i-1}$, we refer to the nodes $(N_{i,2k-1}, N_{i,2k})$ as a *sibling-pair*. In a complete binary tree, every node other than the root has a sibling. We define the sibling of the root to be `null`.

  Example: In Figure 2, the sibling of the m_5 node is the m_6 node. Similarly, the sibling of the $\texttt{val}_{2,4}$ node is the $\texttt{val}_{2,3}$ node. The nodes corresponding to (m_3, m_4) form a sibling-pair.

- *Path to Root*: For any leaf node $N_{d,j}$, the path from $N_{d,j}$ to the root is the (unique) sequence of its ancestor nodes in order, starting from $N_{d,j}$ all the way till the root node $N_{0,1}$.

  Example: In Figure 2, the path from m_4 to the root is the sequence of nodes corresponding to the values $(m_4, \texttt{val}_{2,2}, \texttt{val}_{1,1}, \texttt{val}_{0,1})$.

- *Sibling-Coupled Path to Root*: Given any leaf node $N_{d,j}$, we define a sibling-coupled path to root as follows: it is a sequence of $(d+1)$ sibling-pairs. The first sibling-pair in this sequence consists of sibling-nodes $(N_{d,2k_1-1}, N_{d,2k_1})$ such that $j = 2k_1 - 1$ or $j = 2k_1$ (recall, the leaf node is $N_{d,j}$, therefore j must be either $2k_1 - 1$ or $2k_1$). The i^{th} pair in this sequence is a sibling-pair $(N_{d-i+1,2k_i-1}, N_{d-i+1,2k_i})$ such that either $N_{d-i+1,2k_i-1}$ or $N_{d-i+1,2k_i}$ is the unique ancestor of $N_{d,j}$ at level $d-i+1$. Finally, the last element in this sequence is the pair $(N_{0,1}, \texttt{null})$.

  Example: In Figure 2, the sibling-coupled path to root corresponding to m_4 is the following sequence of pairs: $((m_3, m_4), (\texttt{val}_{2,1}, \texttt{val}_{2,2}), (\texttt{val}_{1,1}, \texttt{val}_{1,2}), (\texttt{val}_{0,1}, \texttt{null}))$.

Using the above notions, we can now present the *proof of membership* protocol. Suppose the server wants to prove that the string m_j is present at the j^{th} index. The server sends m_j, together with the sibling-coupled path to the root (note that the client has `summary`, which should be the root node value $\texttt{val}_{0,1}$). Let us denote the sequence of pair nodes (in the sibling-coupled path) by $(N_{d,2k_1-1}, N_{d,2k_1})$, $(N_{d-1,2k_2-1}, N_{d-1,2k_2})$, ..., $(N_{0,1}, \texttt{null})$, and $\texttt{val}_{i,j}$ is the value associated with node $N_{i,j}$. The client proceeds to authenticate by iterating over this sequence as follows:

- If $j = 2k_1 - 1$, then check that $\texttt{val}_{d,2k_1-1} = m_j$. Else, if $j = 2k_1$, then check that $\texttt{val}_{d,2k_1} = m_j$.
- Check that $\texttt{summary} = \texttt{val}_{0,1}$.
- For $i = d$ to 1, do the following:
  1. Check if $\texttt{CRF.Fn}(\texttt{val}_{i,2k_{d-i+1}-1} + \text{“\#”} + \texttt{val}_{i,2k_{d-i+1}})$ is either equal to $\texttt{val}_{i-1,2k_{d-i+2}-1}$ or $\texttt{val}_{i-1,2k_{d-i+2}}$.[2]
- If all the comparisons match, conclude that there has been no tampering, otherwise conclude that a document has been modified.

Example: In Figure 2, suppose the server wants to prove that m_5 is the 5^{th} document. It sends $((m_5, m_6), (\texttt{val}_{2,3}, \texttt{val}_{2,4}), (\texttt{val}_{1,1}, \texttt{val}_{1,2}), (\texttt{val}_{0,1}, \texttt{null}))$ to the client, which stores `summary`. The client first checks that $\texttt{summary} = \texttt{val}_{0,1}$. Next, it checks the following:

- $\texttt{CRF.Fn}(m_5 + \text{“\#”} + m_6) = \texttt{val}_{2,3}$
- $\texttt{CRF.Fn}(\texttt{val}_{2,3} + \text{“\#”} + \texttt{val}_{2,4}) = \texttt{val}_{1,2}$
- $\texttt{CRF.Fn}(\texttt{val}_{1,1} + \text{“\#”} + \texttt{val}_{1,2}) = \texttt{val}_{0,1}$

If all the above checks pass, the client is convinced that m_5 sent by the server is indeed the 5^{th} document in the list.

[2]Note that we can have a stricter check here, where we compute whether $2k_{d-i+2} - 1$ or $2k_{d-i+2}$ is the parent of $2k_{d-i+1}$, and only compare the CRF output with that node's value. For simplicity, we compare it against both the sibling values.

1.1.1 Updating a document

The Merkle-tree based approach can also be used to update the documents. Suppose the client wants to replace the old document stored at $N_{d,\texttt{doc\_idx}}$ with `new_document`. Simply setting $\texttt{val}_{d,\texttt{doc\_idx}}$ to `new_document` is a bad idea because $\texttt{val}_{i,j}$ for all ancestor nodes of $N_{d,\texttt{doc\_idx}}$ have already been computed using the old document.

Whenever the client sends a document to be updated, the server updates the document, as well as a few digests, and sends back a new `summary`. This works as follows: after replacing $\texttt{val}_{d,\texttt{doc\_idx}}$ with `new_document`, we need to recompute and update $\texttt{val}_{i,j}$ for all ancestor nodes of $N_{d,\texttt{doc\_idx}}$ as we had done while building the Merkle Tree. The `summary` should also be updated to hold the updated value of $\texttt{val}_{0,1}$.

2 Assignment Questions

You must solve all the problems below, preferably before your lab session next week. The lab evaluation questions will build upon these assignment questions, and therefore it is strongly recommended that you try these questions beforehand.

2.1 Merkle Trees

You are given the following classes:

- `public class Pair<A,B>` : Objects of this class would be used to store each element in the sequence *Sibling-Coupled Path to Root*. Check out the methods `QueryDocument` and `Authenticate` in class `MerkleTree` to see how these objects would be used. The attributes of the class are:
  - `public` A `First`: the first element stored in the object.
  - `public` B `Second`: the second element stored in the object.
- `public class TreeNode`: This class has the following attributes:
  - `public TreeNode parent`: pointer to the parent node
  - `public TreeNode left`: pointer to the left child node
  - `public TreeNode right`: pointer to the right child node
  - `public String val`: the value contained in this node
- `public class MerkleTree`: This class has the following attributes:
  - `public TreeNode rootnode`: pointer to the root node of the Merkle Tree
  - `public int numdocs`: number of documents (n) stored in this Merkle tree. **You can assume that this is a power of 2.**

  This class has the following methods:

  To be implemented:
  - `public String Build (String[] documents)`: Accepts an array of n documents and builds a Merkle Tree using it. It returns $\texttt{val}_{0,1}$ as the `summary`.

    The method also sets $\texttt{numdocs} = n$. You can assume n is a power of 2.
  - `public List<Pair<String,String>> QueryDocument (int doc_idx)`: Accepts the index of a document and returns the *Sibling-Coupled Path to Root*.

– `public static boolean Authenticate(List<Pair<String,String>> path , String summary)`: Accepts the *Sibling-Coupled Path to Root* and returns True if no tampering is detected and False otherwise.

– `public String UpdateDocument(int doc_idx, String new_document)`: Updates $\texttt{val}_{\log(n),\texttt{doc\_idx}}$ with `new_document` in the Merkle Tree and returns $\texttt{val}_{0,1}$ as the new `summary`.

Exercise 2.1. ***Merkle Tree - Proof of membership and updates***

Implementing methods of class `MerkleTree`

(a) ♠ `Build`*: This method accepts an array of n documents and build a Merkle Tree using it.*

- *Each of these documents would be stored at the leaf nodes (i.e.* $\texttt{val}_{\log(n),j} = \texttt{documents}[j-1]$ $\forall\ j \in [n]$*).*[3]
- *Each of the non leaf nodes would store the CRF output after concatenating the strings stored of the two child nodes (i.e.* $\texttt{val}_{i,j} = \texttt{CRF.Fn}(\texttt{val}_{i+1,2j-1} + \text{“\#”} + \texttt{val}_{i+1,2j}))$*).*
- *At the end, the root of the tree should be stored at* `rootnode` *and n (length of* `documents`*) should be stored at* `numdocs`*.*
- *Hint: While building the Merkle Tree you would need the first two nodes to create the value of the parent node. What type of data structure supports this operation?*

(b) QueryDocument: Accepts the index of a document and returns the Sibling-Coupled Path to Root

- *The first element of this list should be the pair of* $\texttt{val}_{\log(n),\texttt{doc\_idx}}$ *and its sibling (in correct order).*
- *Each element thereafter should be* $(\texttt{val}_{i,2k-1}, \texttt{val}_{i,2k})$ *where these are the values stored at* $N_{i,2k-1}$ *and* $N_{i,2k}$*, and one of* $N_{i,2k-1}$ *or* $N_{i,2k}$ *is an ancestor node of* $N_{\log(n),\texttt{doc\_idx}}$*.*
- *In this way, the last pair will store* $\texttt{val}_{0,1}$ *and* `null`*.*

(c) Authenticate: Accepts the path from a leaf till the `rootnode` *(Sibling-Coupled Path to Root) and checks whether the path remains untampered.*

- *Start iterating from the first element up to the last element. At each index i, confirm whether the CRF output (upon concatenating the values of sibling nodes) matches with the* `val` *of the parent node (stored at next index).*
- *If any check fails, it means the document has been modified - so return* False
- *Else if all checks pass and* $\texttt{val}_{0,1}$ *matches with* `summary`*, return* True

(d) ♠ *UpdateDocument: Accepts a* `doc_idx` $\in [1, n]$ *and replaces* $\texttt{val}_{\log(n),\texttt{doc\_idx}}$ *with* `new_document` *in the Merkle Tree*

- *Start from* $N_{0,1}$ *and traverse down the Merkle Tree until you reach* $N_{\log(n),\texttt{doc\_idx}}$
- *Replace* $\texttt{val}_{\log(n),\texttt{doc\_idx}}$ *with* `new_document`
- *Traverse back up from* $N_{\log(n),\texttt{doc\_idx}}$ *to the root* $N_{0,1}$ *while updating* $\texttt{val}_{i,j}$ *for each ancestor node* $N_{i,j}$ *encountered in the path.*
- *Return* $\texttt{val}_{0,1}$ *as the updated* `summary`*.*

Useful predefined functions:

- `Fn`*: Compressing function defined in the* `CRF` *class in the “Includes” directory.*

[3] Note the array indices of the `documents` array are from 0 to $n-1$; this is slightly inconsistent with the description in Section 1.1, where the array is indexed from 1 to n.

Exercise 2.2. ***Optional Question: Building CRFs from Bounded-Domain CRFs***

We have provided an implementation of a CRF that maps unbounded length strings to `outputsize` *character hexadecimal strings. How are these functions implemented? In this optional exercise, we will study two different approaches for building such a CRF.*

Both these constructions use a 'core CRF' that maps $2 * \texttt{outputsize}$ *characters to* `outputsize` *character hexadecimal strings. Let us refer to this 'core CRF' as* `BoundedMsgFn`*. We will assume that we are given such a function.*[4] *Now, we would like to implement a CRF that maps arbitrary length strings to* `outputsize` *character hexadecimal strings.*

The first approach is a sequential approach, along the lines of our authenticated lists construction. You can read about it here. The second approach is using Merkle trees. Implement both these approaches.

- *Which of these approaches is 'multithreading-friendly'? That is, suppose you use the same* `BoundedMsgFn` *for both these implementations. Suppose you want to compute the output on an input of size* $n \cdot$ `outputsize`*. The first method will require* $\approx n$ *sequential applications of* `BoundedMsgFn`*. The Merkle tree based approach also requires* $\approx n$ *applications of* `BoundedMsgFn`*, but some of the computations can be done in parallel. Can this be exploited using multithreading?*
- *Which of these constructions are easier to attack? Suppose you wish to find two strings, of length* $n \cdot$ `outputsize` *each, such that their corresponding CRF outputs are the same. Can you do something better than the* birthday bound *based randomized algorithm?* [5]

Exercise 2.3. ***Optional : Proof of non-membership*** *The setup is similar to the toy scenario we discussed above. The client has n documents, and assume these documents are in sorted order. The client computes a* `summary` *of these n documents, and then stores them on the server. Later, the client asks the server if document m is present on the server. If the document is not present, the server must send a 'proof' to the client, convincing it that m is not in the set of documents. How can this be achieved using the Merkle-tree based approach?*

3 Instructions

- Do not change the accessibility, names or signatures of the attributes and methods in the driver code. You may add your own attributes and methods to any of the above classes as and when required.
- The default constructor is used to instantiate objects of all the above classes. It is your responsibility to ensure appropriate initialization of the attributes of a newly created object.
- You are allowed to use other data structures from the package java.util.
- **Submission instructions for lab-submission problem:** You must create a directory whose name is your entry number, followed by "Module3" (for example, if your entry number is "xyz120100", then the folder name should be "xyz120100Module3"). The directory must contain `MerkleTree.java`. Finally, compress this directory, and upload it on Moodle. The file name should be `entrynumModule3.zip` (that is, `xyz120100Module3.zip` in the above example).

Acknowledgements

This is Version 1 of the document. If you find any errors, please send an email to `kvenkata@iitd.ac.in` (and participate in the 'error-finding competition'). Many thanks to Rishabh Dhiman, Vansh Kacchwal, Tanish Gupta, Riya Sawhney, Soumil Aggarwal, Adarsh Roy, Revanth Vasireddi, Adit Malhotra, Avani Jain for finding typos in Version 0 of the lab-sheet, and a bug in the sample test cases (Version 0).

[4] Implementing such a function is not too difficult, but will require a couple of crypto lectures, which is beyond the scope of our discussion.

[5] If your algorithm works better than the 'birthday bound', email your solution, together with the time complexity, to `kvenkata@iitd.ac.in`.