P2P Systems and Blockchains

Spring 2019,
instructor: Laura Ricci
laura.ricci@unipi.it

Lesson 12:
DATA STRUCTURE TOOLBOX FOR BLOCKCHAINS
12/4/2019
HASH POINTERS

- an hash pointer is:
  - a pointer to where some info is stored
  - a cryptographic hash of the info
- if we have a hash pointer, we can
  - ask to get the info back
  - verify that it hasn’t changed

Tamper-evident data pointer = Hash Pointer
HASH DATA STRUCTURES

• key idea
  • build data structures with hash pointers

• block chain:
  • linked list with hash pointers
    • important: hash the entire block, also its hash pointer!
  • each block tells us where the value of the previous block is located and a digest to verify that the value has not been changed

• use case: tamper-evident log, a basic data structure in Bitcoin
Tamper-evident linked list: blockchain
Tamper-evident linked list: blockchain
HASH DATA STRUCTURES

- if someone tampers the k-th block of the chain, the hash of block k+1 is not going to match up

- this is because the hash is collision resistant: an adversary cannot tamper the data so that its hash is the same of the data before the tampering

use case: tamper-evident log, a basic data structure in Bitcoin
HASH DATA STRUCTURES

- The adversary can continue to change the next block as well, reaching the head of the list.

- The structure is tamper free if:
  - The hash pointer at the head of the list is stored in a place that the adversary cannot change, (a trusted location),
  - In this way the adversary is unable to change any block without being detected.

- The modification of all the subsequent blocks is computationally unfeasible.
TAMPER EVIDENT BINARY TREES: MERKLE TREES

- binary tree with hash pointers
- remember just the root of the tree in a secure place
- if an adversary tampers some blocks at the bottom of the tree, the hash pointers one level up do not match, and the so on...

![Diagram of TAMPER EVIDENT BINARY TREES: MERKLE TREES](image)
TAMPER EVIDENT BINARY TREES: MERKLE TREES

- binary tree with hash pointers
- remember just the root of the tree in a secure place
- if an adversary tampers some blocks at the bottom of the tree, the hash pointers one level up do not match, and the so on...
MORE GENERALLY...

- can use hash pointers in any pointer-based data structure that has no cycles, for instance, in a DAG.

- several applications:
  - **Advanced Intelligent Corruption Handling**, introduced in eMule from version 0.44:
    - used to check that the block of a file which has been downloaded from the network has not been tampered
    - exploits Merkle trees
      - binary trees with hash pointers
  - **Bitcoin**: uses SHA-256 and RIPEMD-160 (double) cryptographic hash functions
    - the block chain of Bitcoin is a list of blocks of transactions chained through hash pointers.
  - many other applications....
BLOOM FILTERS: SET MEMBERSHIP PROBLEM

- consider the set $S = \{s_1, s_2, ..., s_n\}$ of $n$ elements chosen from a very large universe $U$.
- define an efficient data structure supporting queries like “$k$ is an element of $S$?”

- the function $f$ returns value true or false according to the presence of $k$ in the given set.
APPROXIMATE SET MEMBERSHIP PROBLEM

- S may be
  - a set of keywords describing the files shared by a peer, selected from the universe of all the keywords (Gnutella 0.6)
  - the set of pieces of file owned by a peer (BitTorrent)
  - a set of bitcoin addresses:
    - lightweight (mobile) mobile nodes build Bloom filters with the interesting addresses
    - send the bloom filter to the full nodes: bandwidth saving

- Problem: choose a representation of the elements in S such that:
  - the result of the query is computed efficiently
  - the space for the representation of the elements is reduced
    - the results may be approximated to save space
    - possibility of returning false positives
An approximate solution to the set membership problem:

\[ f(k) \]

\[ \begin{cases} \text{maybe } k \in S \\ \text{false iff } k \notin S \end{cases} \]

- Trade off between:
  - space required
  - probability of false positives
$m$ bits (initially set to 0)
$k$ hash functions
$m$ bits (initially set to 0)

$k$ hash functions

Add

- \begin{array}{cccccccccccccccccc}
S & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & m & m
\end{array}
**Building Bloom Filters**

- \( m \) bits (initially set to 0)
- \( k \) hash functions

If \( f(x) = A \), set \( S[A] = 1 \)

Add

```
\[ S = \begin{array}{ccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]
```
BUILDING BLOOM FILTERS

- $m$ bits (initially set to 0)
- $k$ hash functions

If $f(x) = A$, set $S[A] = 1$

Diagram:

```
  Add

S   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 1 2 m 1 m
```

Laura Ricci
Data structure toolbox for blockchains
BUILDING BLOOM FILTERS

- \( m \) bits (initially set to 0)
- \( k \) hash functions

Add:

\[
\begin{align*}
S &= 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
0 & \quad 1 \quad 2 \quad m-1 \quad m
\end{align*}
\]

if \( f(x) = A \), set \( S[A] = 1 \)
BUILDING BLOOM FILTERS

$m$ bits (initially set to 0)
$k$ hash functions

if $f(x) = A$, set $S[A] = 1$

Add

$S = 000001000100001000$
Building Bloom Filters

$m$ bits (initially set to 0)
$k$ hash functions

if $f(x) = A$, set $S[A] = 1$
Given

- a set $S=\{s_1,s_2,\ldots,s_n\}$ of $n$ elements
- a vector $B$ of $m$ ($n<<m$, generally $nk<m$) bits, $b_i \in \{0,1\}$
- $k$ hash independent functions $h_1, \ldots, h_k$, for each $h_i: S \subseteq \mathbb{U} \to [1..m]$, which return a value uniformly distributed in the range $[1..m]$.
- construction procedure for a Bloom Filter $B[1..m]$:
  for each $x \in S$, $B[h_j(x)]=1$, $\forall$ $j = 1,2,\ldots,k$
- a bit in $B$ may be target for more than $1$ element
BLOOM FILTERS: LOOK UP

- \( m \) bits (initially set to 0)
- \( k \) hash functions

If \( f(x) = A \), set \( S[A] = 1 \)

**Add**

**Query**
BLOOM FILTERS: LOOK UP

$m$ bits (initially set to 0)
$k$ hash functions

if $f(x) = A$, set $S[A] = 1$

Add

Query
BLOOM FILTERS: LOOK UP

\( m \) bits (initially set to 0)
\( k \) hash functions

if \( f(x) = A \),
set \( S[A] = 1 \)

Add

Query

one bit set to 0

\[ \Rightarrow z \notin S \]
To verify if $y$ belongs to the set $S$ mapped on the Bloom Filter, apply the $k$ hash functions to $y$

- $y \in S$ if $B[h_i(y)] = 1$, $\forall \ i = 1,..k$
- if at least a bit = 0, the element does not belong to the set.

**False positives:**

To test for membership, you simply hash the string with the same hash functions, then see if those values are set in the bit vector.

If they aren't, you know that the element isn't in the set.

If they are, you only know that it might be, because another element or some combination of other elements could have set the same bits.
The price paid for this efficiency is that a Bloom filter is a *probabilistic data structure*: it tells us that the element either definitely is not in the set or may be in the set.
PROBABILITY OF FALSE POSITIVES

- Let us consider a set of \( n \) elements mapped on a vector of \( m \) bits through \( k \) hash functions.

- The hash functions used in a Bloom filter should be independent and uniformly distributed and as fast as possible.
  - For instance, \( h_i(x) = \text{MD5}(x + i) \) or \( \text{MD5}(x || i) \) would work.

- Basic assumption: hash functions random and independent.
  - Balls and bins paradigm: like throwing \( k \times n \) balls in \( m \) buckets.

- Goal: evaluate the probability of false positives.
PROBABILITY OF FALSE POSITIVES

First step: compute the probability that, after all the n elements are mapped to the vector, a specific bit of the filter (of size m) has still value 0?

\[ p' = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}. \]

The approximation is derived from the definition of e

\[ \lim_{x \to \infty} \left(1 - \frac{1}{x}\right)^{-x} = e \]
PROBABILITY OF FALSE POSITIVES

- a percentage of \( e^{-kn/m} \) bits are 0, after its construction.

- consider an element not belonging to the set: apply the \( k \) functions, a false positive is obtained if all hash functions return a value = 1

- probability of false positives = \( (1 - e^{-kn/m})^k \)

  depends on
  
  - \( m/n \): number of bits exploited for each element of the set
  - \( k \): number of hash functions

- if \( m/n \) is fixed, it seems two conflicting factors for defining \( k \) do exist....
  
  - decreasing \( k \) increases the number of 0 and hence the probability to have a false positive should decrease, but....
  
  - increasing \( k \) increases the precision of the method. Hence the probability of false positive should decrease...
Fixed the ratio \( m/n \), the probability of false negatives first decreases, then increases, when considering increasing values of \( k \):

- \( m/n=2 \), a few bits for each element, “too much hash functions” cannot be exploited because they fill the filter of 1.
- \( m/n=10 \), a larger number of hash functions decreases the probability of false positives.
let us now suppose that $k$ is fixed, the probability of false positives exponentially decreases when $m$ increases ($m$ number of bits in the filter).

for low values of $m/n$ (a few bits for each element), the probability is higher for large values of $k$
PROBABILITY OF FALSE POSITIVES

A Bloom filter becomes effective when $m = c \times n$, with $c$ constant value (low value), for instance $c = 8$

In this case with 5-6 hash functions the probability of false positives is low

Good performances with a limited number of bits

<table>
<thead>
<tr>
<th>bits/element</th>
<th>$m/n$</th>
<th>2</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of hash functions</td>
<td>$k$</td>
<td>1.39</td>
<td>5.55</td>
<td>11.1</td>
<td>16.6</td>
</tr>
<tr>
<td>false-positive probability</td>
<td>$f$</td>
<td>0.393</td>
<td>0.0216</td>
<td>$4.59 \cdot 10^{-4}$</td>
<td>$9.84 \cdot 10^{-6}$</td>
</tr>
</tbody>
</table>
PROBABILITY OF FALSE POSITIVES

- trade-off between space/number of hash functions/probability of false positives
- if n and m are fixed (fix the number of bits for each element)
  - determine which is the value of k minimizing the probability of false positives
  - compute the derivative of the function of the previous slide so obtaining the minimum \((\ln 2 \approx 0.7)\)

\[
k = \frac{m}{n} \ln 2,
\]

To this value, corresponds a value of the probability equal \(0.62^{m/n}\).
with the values computed in the previous slides, the probability that one bit is still equal to 0 after the application of the $k$ functions is

$$p = e^{-kn/m} = \frac{1}{2}$$

the optimal values are obtained when the probability that a bit is equal to 0 after the application of the $k$ functions to the $n$ elements is equal to $\frac{1}{2}$

an “optimal” Bloom Filter is a “random bitstring” where half of the bits chosen uniformly at random, is 0.
PROBABILITY OF FALSE POSITIVES

- probability as a function of the number of elements of the set
  - \( k \) optimum, given \( n \) and \( m \)
- logarithmic scale
- if the number of bits for each element is not sufficient the probability exponentially grows

![Graph showing the false positive rate of Bloom filters](image)
**BLOOM FILTERS: OPERATIONS**

- **Union** - Given two Bloom Filters, $B_1$ and $B_2$ representing, respectively, the set $S_1$ and $S_2$ through the same number of bits and the same number of hash functions, the Bloom filter representing $S_1 \cup S_2$ is obtained by computing the bitwise OR bit of $B_1$ and $B_2$.

- **Delete**: note that it is not possible to set to 0 all the elements indexed by the output of the hash functions, because of the conflicts.
  - **Counting Bloom Filters**: each entry of the Bloom Filter is a counter, instead of a single bit.
    - exploited to implement the removal of elements from the Bloom filter.
    - at insertion time, increment the counter.
    - at deletion time, decrement the counter.
BLOOM FILTERS: OPERATIONS

- intersection: given two Bloom Filters B1 and B2 representing respectively, the sets S1 and S2 through the same number of bits and the same number of hash functions.

- the intersection of the Bloom filters is obtained by computing the bitwise and of B1 and B2 and approximates $S_1 \cap S_2$

- as a matter of fact, if a bit is set to 1 in both Bloom filters, this may happen because:
  - this bit corresponds to an element $\in S_1 \cap S_2$, therefore it is set to 1 in both filters: in this case no approximation
  - this bit corresponds to an element $\in S_1 - (S_1 \cap S_2)$ and to an element $\in S_2 - (S_1 \cap S_2)$ hence it does not correspond to any element in the intersection
Some applications....

- Google BigTable and Apache Cassandra use Bloom filters to avoid costly disk lookups considerably and to increases the performance of a database query.

- The Google Chrome web browser uses a Bloom filter to identify malicious URLs. Any URL is first checked against a local Bloom filter and only upon a hit a full check of the URL is performed.

- Bitcoin uses Bloom filters to verify payments without running a full network node.

- Gnutella 0.6 exploits a simplified version of Bloom Filters

but many other ones currently exist....
BLOOM FILTERS: APPLICATIONS

Guard against expensive operations (like disk access)
First line of defence in high performance (distributed) caches
Peer to Peer communication
Routing - Resource Location

Squid Proxy Cache
Google BigTable
Cassandra
HBase
Various RDBMS'
Google Chrome
Cisco Routers

MERKLE TREE

• Hash trees or Merkle trees
  • a data structure summarizing information about a big quantity of data with the goal to verifying the correctness of the content

• introduced by Ralph Merkle in 1979

• characteristics
  • simple
  • efficient
  • versatile

• a complete binary tree of hashes built starting form a initial set of blocks
  • exploits a hash function $H$
  • leaves: $H$ is applied to the initial blocks
  • internal nodes: $H$ is applied to the concatenation of the hashes of the sons of a node
• to simplify the presentation, internal nodes are simply shown as concatenation of the hashes of the sons
• actually, hash is recursively applied to the concatenation
Suppose that we want to check that block 2 is not tampered
• a Merkle proof consists of
  • a chunk: 2
  • the root hash of the tree: the red one
  • the "branch" consisting of all of the hashes going up along the path from the chunk to the root.
• this is sufficient to verify that 2 is really in the tree and in that position
MERKLE TREE FORMAL DEFINITION

- Let us consider an initial set of symbols $S$, $S=\{s_1, \ldots, s_n\}$, $n=2^h$, $h$ tree height

- Merkle Tree Procedure $\text{MTP} = <\text{CMT}, \text{DMT}>$

- $\text{CMT}$ (Coding Merkle Tree): starting from the initial set $S$ of symbols, build the complete binary tree of height $h$
  - Leaves $= H(s_i)$
  - Internal Nodes $= \text{hash of the concatenation of the hash values of the sons} \ H(L||R)$

- Output
  - the root of the binary tree
  - a “witness” $w_i$ for each symbol $s_i$
    - $w_i = \text{siblings of the nodes on the path from the leaf } s_i \text{ to the root}$
• DMT\((s_i, w_i, \text{root})\): Decoding Merkle Tree

• The soundness of \(s_i\) is deduced from the comparison between the root generated during the decoding phase and the root generated in the coding phase

• Each symbol \(s_i\) is authenticated by considering the witness \(w_i\) and the root
MERKLE TREE

- Build Merkle tree
- 3rd party trusted stores root
- $\log(n)$ hashes are sufficient for checking each fragment
- “Data integrity over untrusted storage with small communication cost”
- Pros:
  - scales logarithmically in the number of objects
  - fine-grained data integrity
  - each write does work proportional to the size of the fragment
- Cons:
  - static: smallest segment size = smallest unit of verification
MERKLE TREES: BITTORRENT

- a trusted site (for instance that indexing the .torrent) stores:
  - the root hash
  - the total size of the file
  - the piece size

- a peer:
  - receives a piece from another peer together with the hashes of the piece's sibling and of its uncles, that is the sibling $Y$ of its parent $X$, and the uncle of that $Y$ until the root is reached.
  - calculates the hash of that piece.
  - request the root hash from a trusted site and
  - using this information the client recalculates the root hash of the tree, and compares it to the root hash it received from the trusted source.

- subject to possible pre-image attacks.
MERKLE TREES: APPLICATIONS

- Merkle trees and peer to peer applications:
  - can help ensuring in a P2P network that data blocks received from other peers are received undamaged and unaltered
  - to check that the other peers do not lie and send fake blocks.

- BitTorrent: uses Merkle trees to ensure that the files you download from peers haven’t been tampered.

- Similar for eMule

- Distributed Version Control (Git/Mercurial)

- Copy-On-Write Filesystems (btrfs/ZFS)

- Distributed NoSQL Databases (Cassandra, Riak, Dynamo)

- The distributed web! (IPFS)

- Cryptos (Bitcoin, Ethereum)
MERKLE TREES: APPLICATION

Peer to Peer communication

Amazon Dynamo
Google BigTable
Cassandra
HBase
ZFS
Google Wave

Laura Ricci
Data structure toolbox for blockchains
stores a set of text strings and defines an efficient implementation of the following operations: given a string $x$:

- `search(x)`
- `insert(x)`
- `delete(x)`

why do we care about it?

- Ethereum stores the state of the execution of the smart contract on the blockchain
- the state is a set of key value pairs and
- Ethereum needs an efficient way to retrieve a value given the string of the key
TRIE DATA STRUCTURE

- trie: a data structure for storing a set of strings
- name is from the word “retrieval”
- $\Sigma$ is the alphabet of the strings
- assume that all strings end with “$” (not in $\Sigma$)

Set of strings: \{bear, bid, bulk, bull, sun, sunday\}
PROPERTIES OF A TRIE

- a multi-way tree.
- each node has from 1 to d children (d size of the alphabet)
- each edge of the tree is labeled with a character.
- each leaf node corresponds to the stored string, which is a concatenation of characters on a path from the root to this node

assumptions:
- \( n \) : strings to store
- \( N \) : characters in total
- \( m \) : length of a string \( x \)
- size of the alphabet \( d = |\Sigma| \)
Trie-Search(t, P[k..m])
01 if t is leaf then return true
02 else if t.child(P[k])=nil then return false
03 else return Trie-Search(t.child(P[k]), P[k+1..m])

- The search algorithm just follows the path down the tree (starting with Trie-Search(root, P[0..m]))

Trie-Insert(t, P[k..m])
01 if t is not leaf then //otherwise P is already present
02 if t.child(P[k])=nil then
03 Create a new child of t and a "branch" starting with that child and storing P[k..m]
04 else Trie-Insert(t.child(P[k]), P[k+1..m])

- How would the delete work?
**COMPLEXITY ANALYSIS**

- `t.child(c)`
  - operation to see if the child of node `t` corresponding to the character `t` exists.

- what is the node structure? What is the complexity of the `t.child(c)` operation?
  - an array of child pointers of size `d`: waist of space, but `child(c)` is \( O(1) \)
  - a hash table of child pointers: less waist of space, `child(c)` is expected \( O(1) \)
  - a list of child pointers: compact, but `child(c)` is \( O(d) \) in the worst-case
  - a binary search tree of child pointers: compact and `child(c)` is \( O(lg\ d) \) in the worst-case

- size:
  - \( O(N) \) in the worst-case

- search, insertion, and deletion (string of length `m`):
  - depending on the node structure: \( O(dm) \), \( O(m\ lg\ d) \), \( O(m) \)
COMPACT TRIES

- observation:
  - having chains of one-child nodes is wasteful
  - replace a chain of one-child nodes with an edge labeled with a string of more than one character
  - each non-leaf node (except root) has at least two children
COMPACT TRIE: IMPLEMENTATION

- strings are stored external to the trie, in one array
- edges are labeled with indices in the array \((\text{from}, \text{to})\) pointing to one of the occurrences of the string
- can be used to do **word matching**: find where the given word appears in the text.
  - use the compact trie to “store” all the words in the text
  - each child in the compact trie has a list of indices in the text where the corresponding word appears.
EXPLOITING COMPACT TRIES

to find a word $P$:

- at each node, follow the edge $(i,j)$, such that $P[i..j] = T[i..j]$
- if there is no such edge, there is no $P$ in $T$, otherwise, find all starting indices of $P$ when a leaf is reached
Patricia: Practical Algorithm To Retrieve Information Coded In Alphanumeric

- a compact trie where each edge’s label (from, to) is replaced by (T[from], to – from + 1)
MERKLE PATRICIA TRIES IN ETHEREUM

- in the Ethereum Blockchain, the requirement is to store the state of the contracts on the blockchain
- the state is combination of key value pair.
  - for instance: the address of an account is the key and account balance is the value that need to be stored
- Ethereum uses Merkle Patricia tries
  - a combination of Merkle tree and Patricia trie
  - the idea
    - Merkle tree maintains data integrity
    - Patricia tree enable faster search of data so in Ethereum both these trees are combined together.