Lesson n.11
Data Structures for P2P Systems: Bloom Filters, Merkle Trees

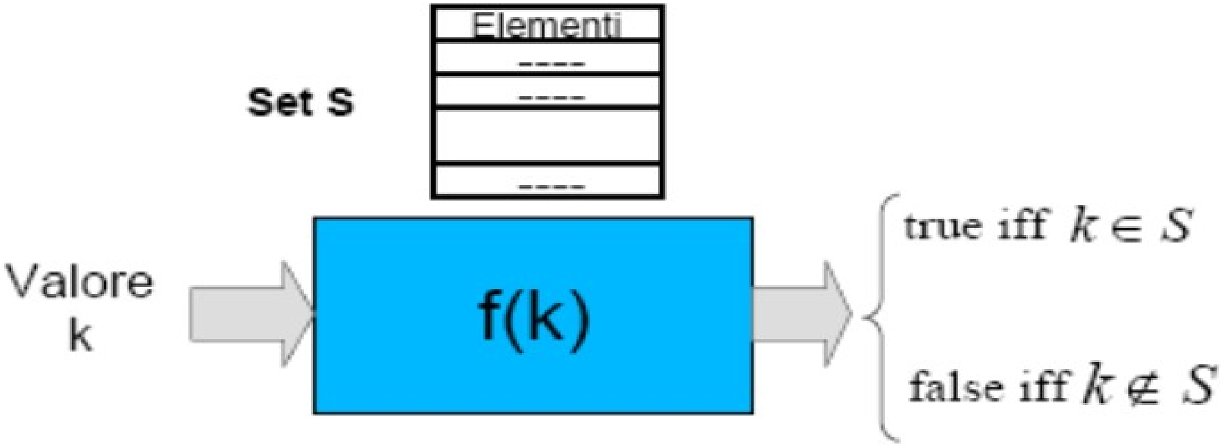
Didactic Material
Tutorial on Moodle

Laura Ricci

15/11/2013

SET MEMBERSHIP PROBLEM

Let us consider the set $S=\{s_1,s_2,\ldots,s_n\}$ of n elements chosen from a very large universe U. Define a 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

CHAIN HASHING

- different hashing techniques optimizing space or time
  - balls e bins model: throw m balls in n buckets, each shot is independent from the other ones
- chain hashing:
  - a structure with m positions (bins), an hash function to insert the n elements of the set S (balls) in one position.
  - random hash function; the location of each element is chosen uniformly at random
  - non approximate look-up
  - all the elements of the same bins are chained in a list
  - look-up time is proportional to the number of elements for each bin
    - problem: compact bin representation (a lot of bins are empty)
    - trade off: number of bins/look up times

FINGERPRINT HASHING

- Goal: space optimization
- Exploit a hash function to compute the fingerprint of each element
  - a 32-bits fingerprint for a 64 bits element
  - look-up: search in the fingerprint lists
  - approximate set membership problem
    - false positives the same fingerprint for different elements
    - the probability of false positives depends on the number of bis used for the fingerprint
    - requires at least n bits, to guarantee a low false positive probability

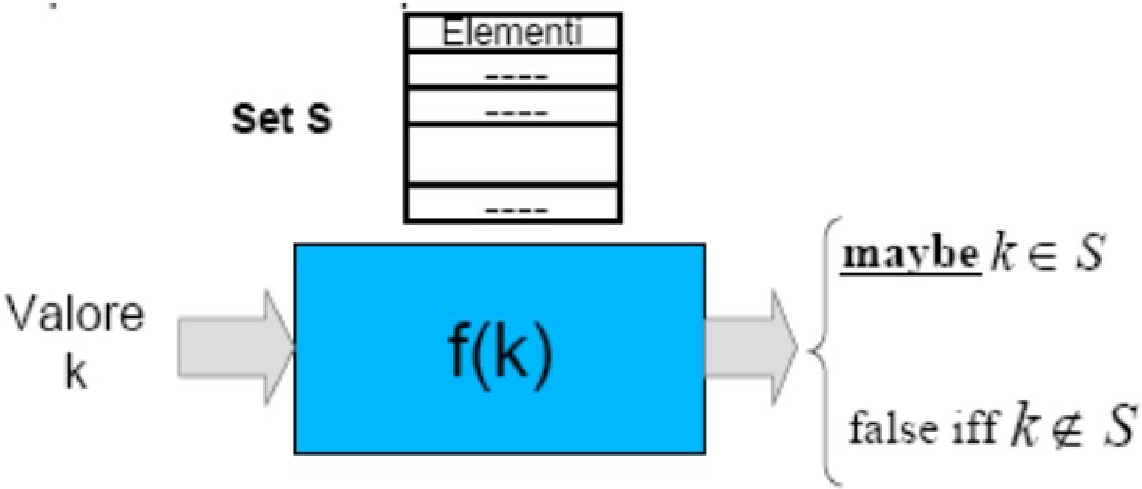
APPROXIMATE SET MEMBERSHIP PROBLEM

- S may be
  - a set of keywords describing the files shared by a peer, selected form the Universe of all the keywords (Gnutella 0.6)
  - The set of pieces of file owned by a peer (BitTorrent)
  - The set of the 'crackable password' with the goal of showing to the user which passwords have to be avoided

- 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
    - to reduce the space required to represent the elements, the results may be approximated
    - possibility of returning false positives

APPROXIMATE SET MEMBERSHIP PROBLEM

An approximate solution to the set membership problem:



Trade off between:

- Space required
- Probability of false positives

BLOOM FILTERS: COSTRUZIONE

m bits (initially set to 0)

k hash functions

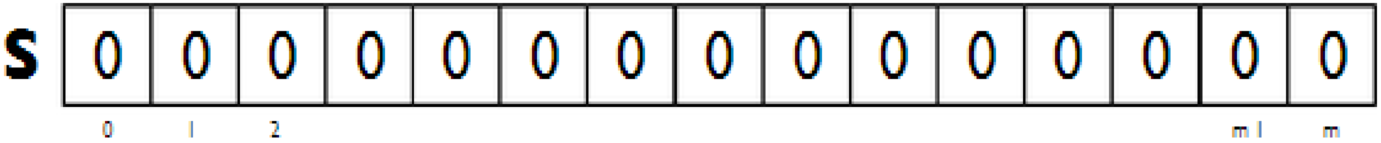
S	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	1	2											m 1	m

BLOOM FILTERS: COSTRUZIONE

m bits (initially set to 0)

k hash functions

Add



S	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	1	2											m-1	m

BLOOM FILTERS: COSTRUZIONE

m bits (initially set to 0)

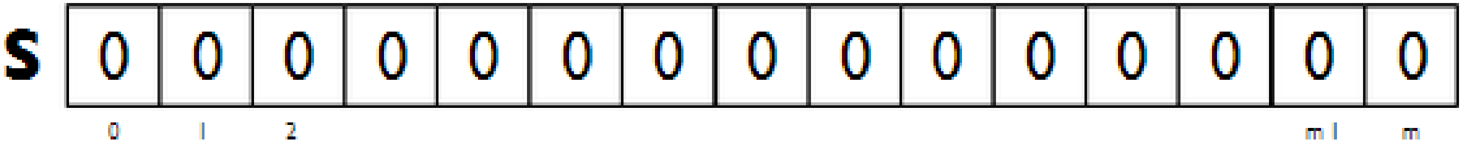
k hash functions

x

if f(x) = A,

set S[A] = 1

Add



S 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

BLOOM FILTERS: COSTRUZIONE

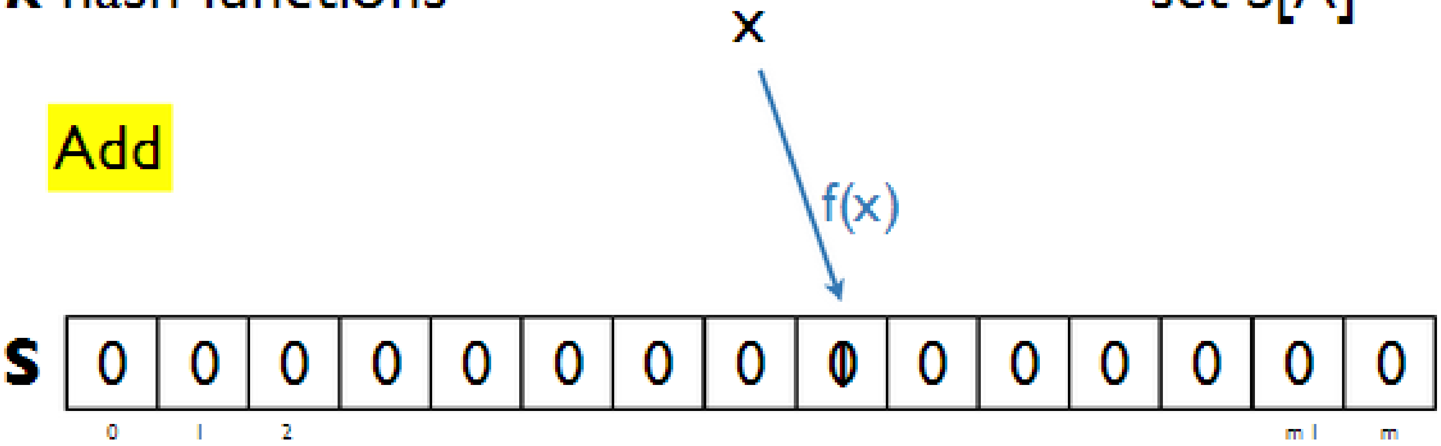
m bits (initially set to 0)

k hash functions

if f(x) = A,

set S[A] = 1

Add



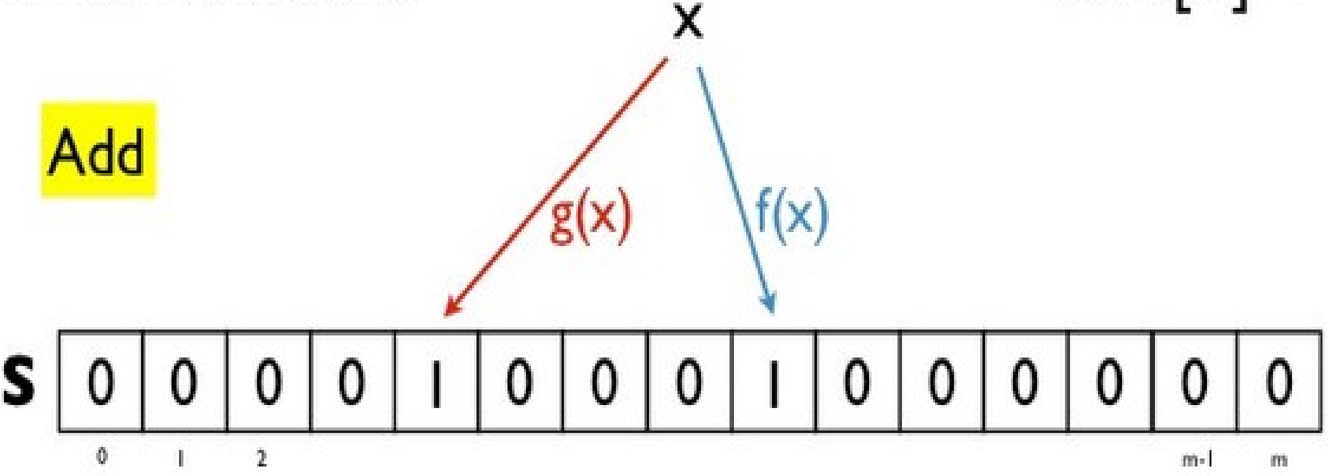
BLOOM FILTERS: COSTRUZIONE

m bits (initially set to 0)

k hash functions

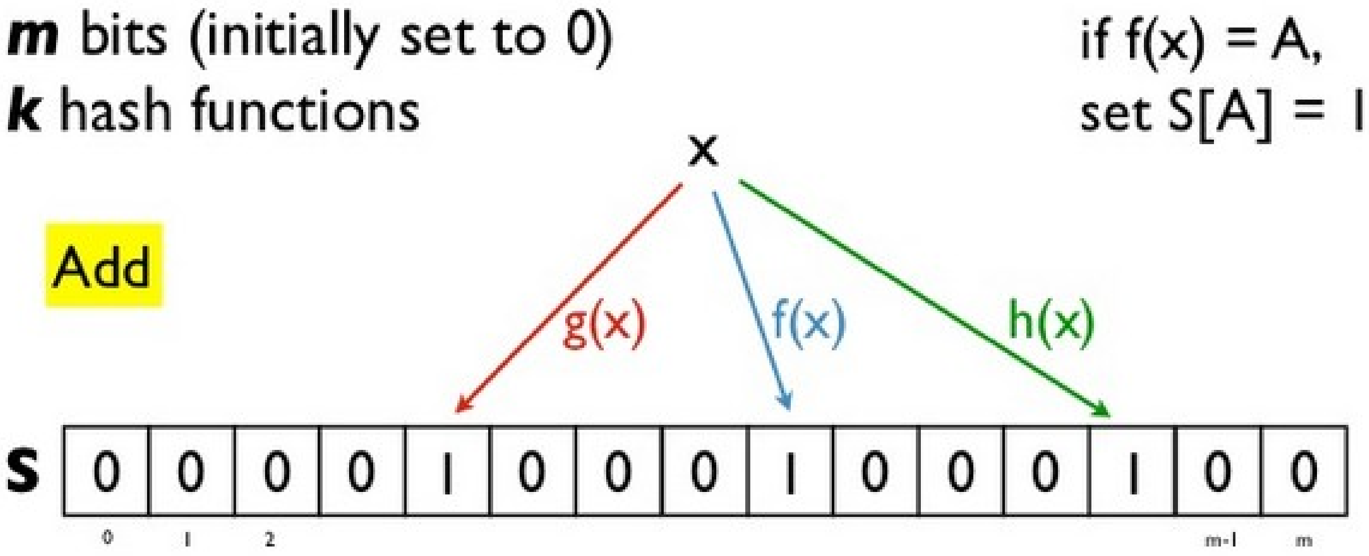
if f(x) = A,

set S[A] = 1



S	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
	0	1	2											m-1	m

BLOOM FILTERS: COSTRUZIONE



m bits (initially set to 0)
k hash functions

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

Add

S	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0
	0	1	2											m-1	m

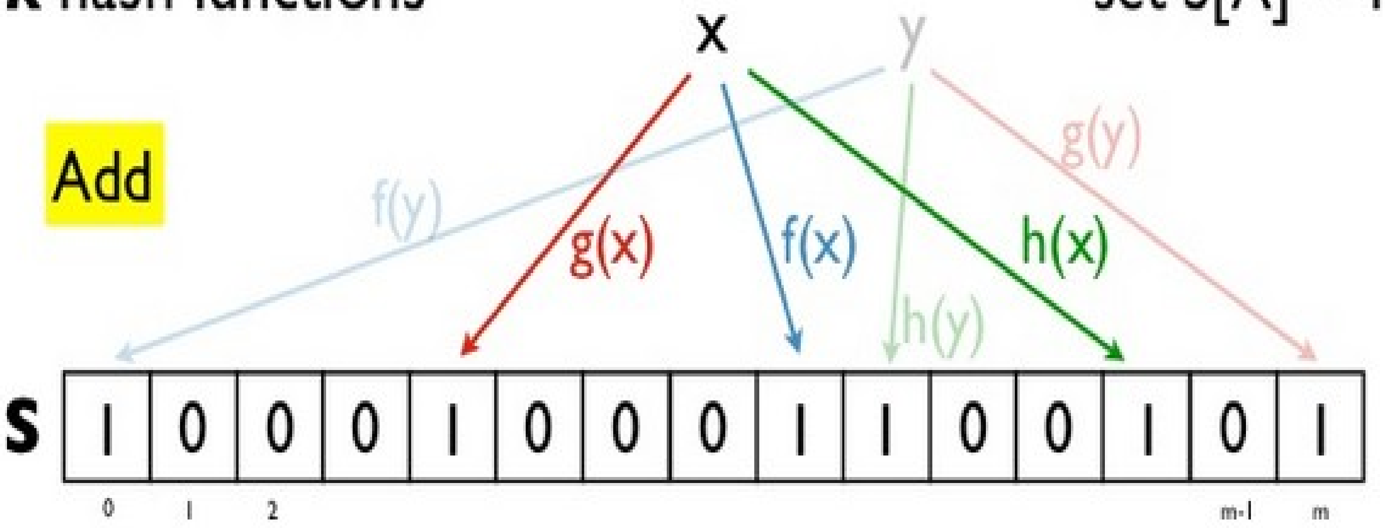
BLOOM FILTERS: COSTRUZIONE

m bits (initially set to 0)

k hash functions

if f(x) = A,

set S[A] = 1



1	0	0	0	1	0	0	0	1	1	0	0	1	0	1
0	1	2											m-1	m

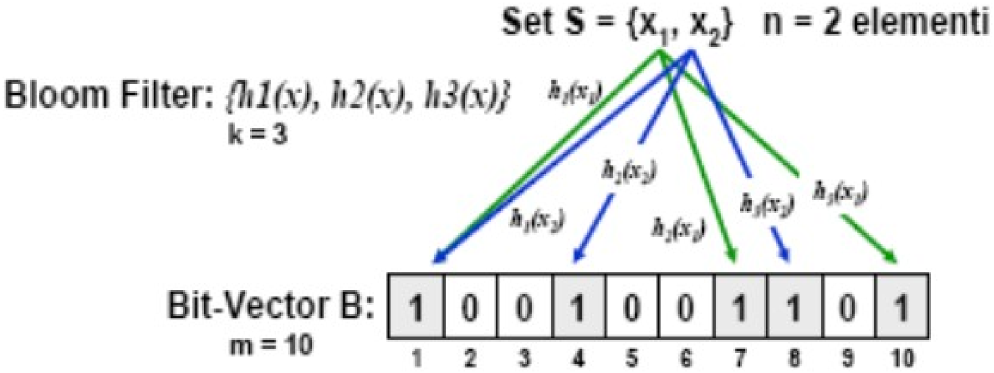
BLOOM FILTERS: COSTRUCTION

Given

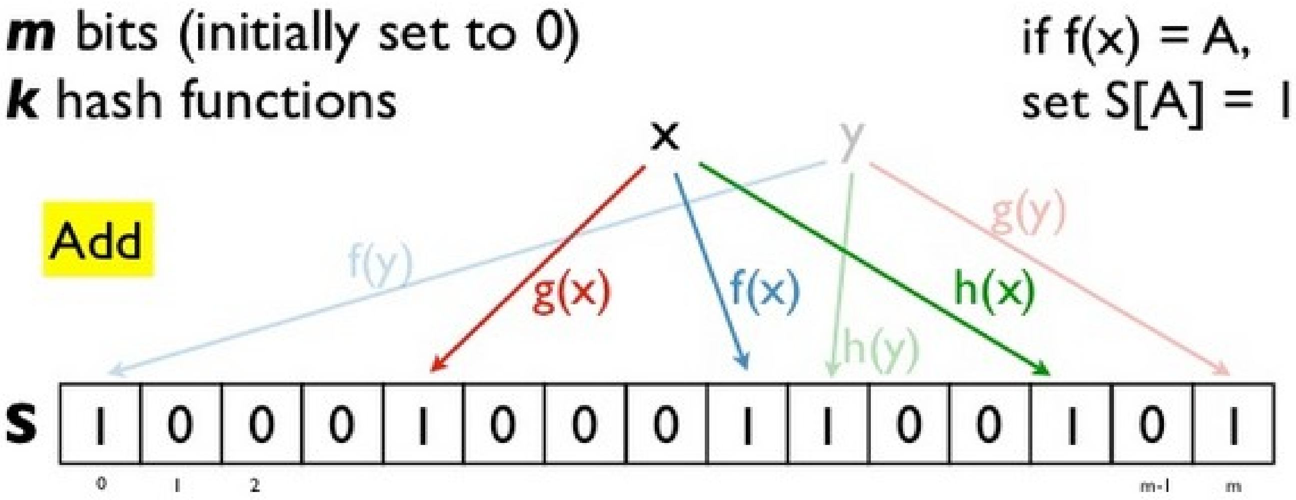
- a set $S=\{s_1,s_2,\ldots,s_n\}$ of n elements
- a vector B of m (n<<m) bits, $b_i \in \{0,1\}$
- k hash independent functions $h_1, \ldots, h_k$, for each h_i: $S \subseteq U \rightarrow [1..m]$, which return a value uniformly distributed in the range [1..m].

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



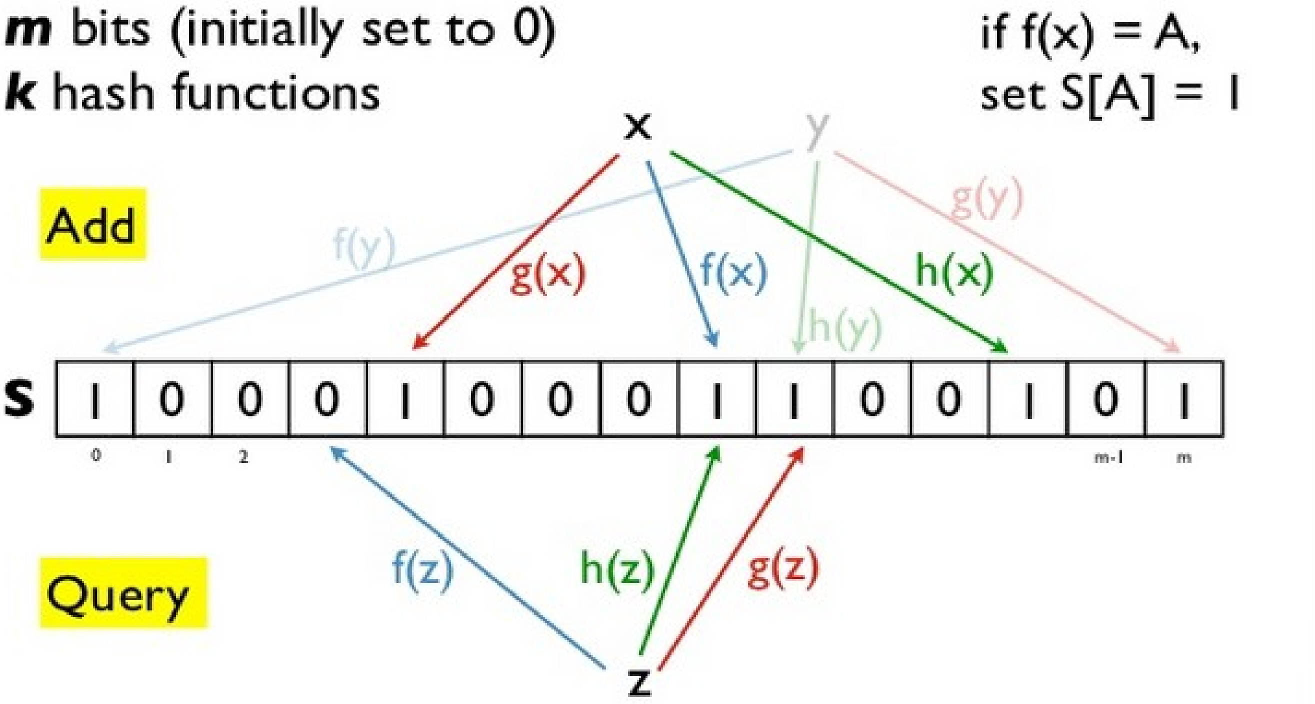
m bits (initially set to 0)
k hash functions

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

Add

Query

BLOOM FILTERS: LOOK UP

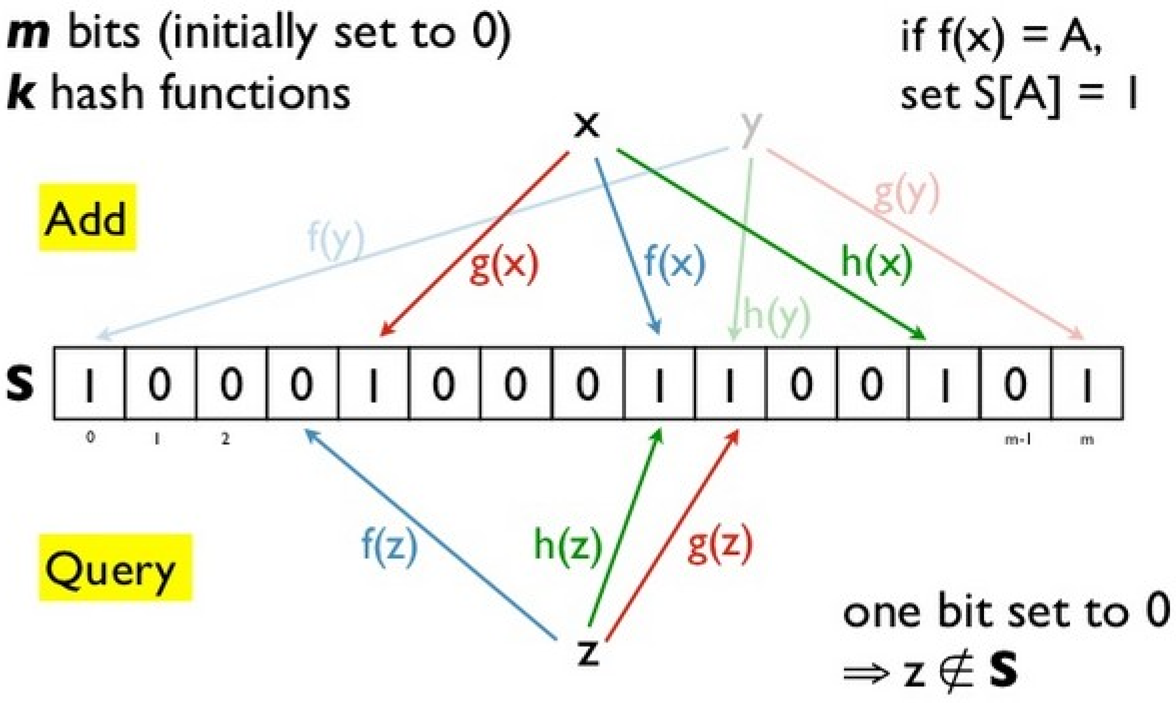


m bits (initially set to 0)
k hash functions

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

S	1	0	0	0	1	0	0	0	1	1	0	0	1	0	1
	0	1	2											m-1	m

BLOOM FILTERS: LOOK UP



m bits (initially set to 0)
k hash functions

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

one bit set to 0
⇒ z ∉ **S**

BLOOM FILTERS: LOOK UP

To verify if y belongs to the set S mapped on the Bloom Filter, apply the k hash functions to y

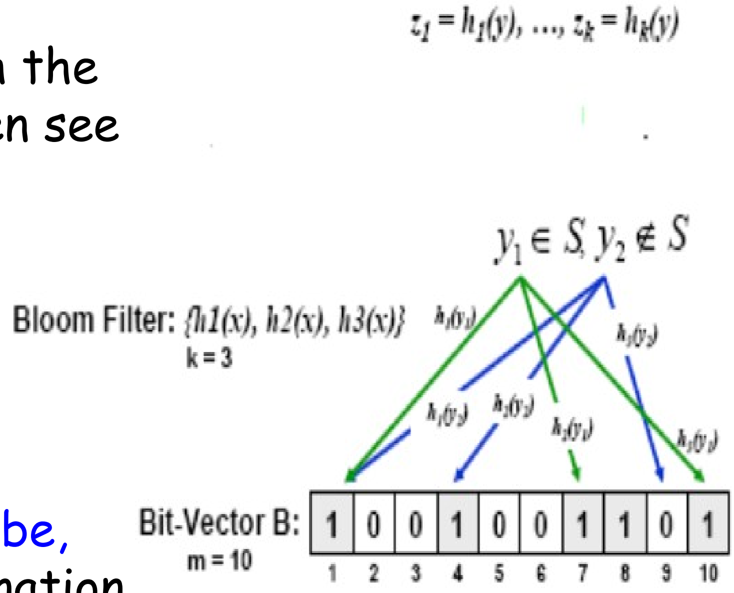
- $y \in S$ if $B[h_i(y)]=1$, $\forall$ $i=1,..k$
- if at least a bit = 0, the element does not belong to the set.

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.



$z_1 = h_1(y), \ldots, z_k = h_k(y)$

1	2	3	4	5	6	7	8	9	10
1	0	0	1	0	0	1	1	0	1

BLOOM FILTERS: TRADE OFF

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 hi(x) = MD5(x + i) or MD5(x || i) would work

- basic assumption: hash functions random and independent
    - balls e 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 elements are mapped to the vector, a specific bit of the filter has still value 0

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

The approximation is derived from the definition of e

$$\lim_{x\to\infty}\left(1 - \frac{1}{x}\right)^{-x} = \mathrm{e}$$

PROBABILITY OF FALSE POSITIVES

- A fraction of $e^{-kn/m}$. bits are 0, after its construction.

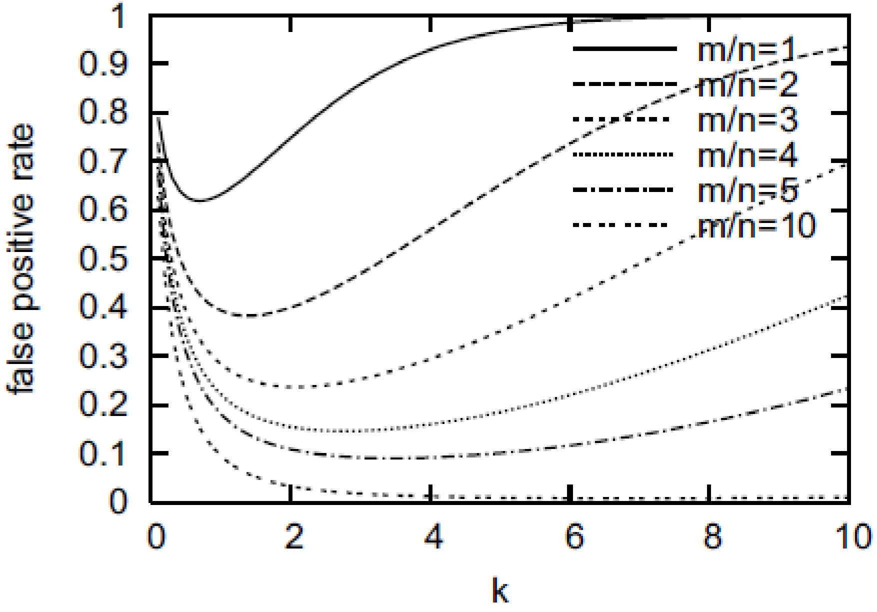
- Consider an element not belonging to the set: apply the k functions, a false positive is obtained if, all hash functions, return a value = 1

- 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 number of elements to be checked and the precision of the method. Hence the probability of false positive should decrease...

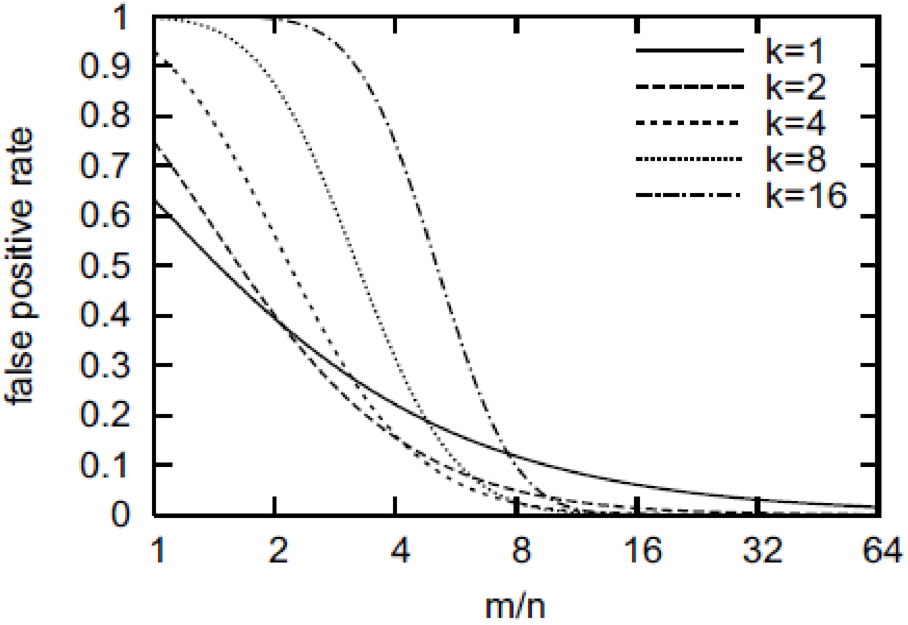
PROBABILITY OF FALSE POSITIVES

- lFix the ratio m/n, the probability of false negatives first decreases, then increases, when considering increasing values of k
  - es: m/n=2, a few bits for each element, “too much hash functions” cannot be exploited because they fill the filter of 1.
  - es: m/n=10, a larger number of hash functions decreases the probability of false positives

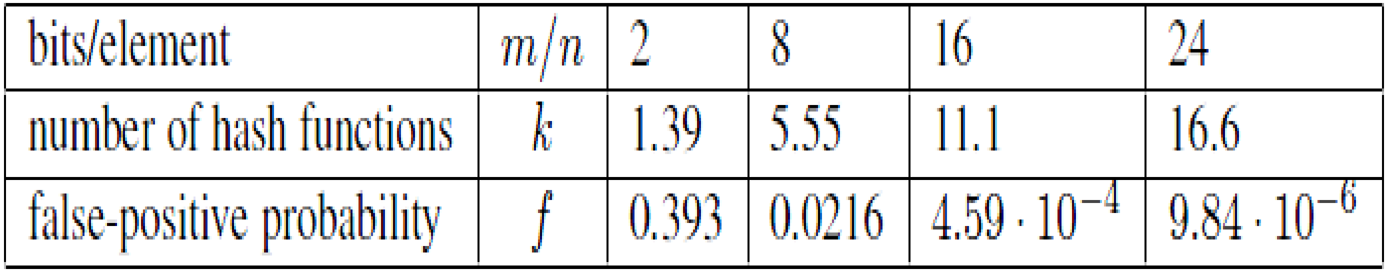


PROBABILITY OF FALSE POSITIVES

- 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 probablity is higher for large values of k



PROBABILITY OF FALSE POSITIVES



bits/element	m/n	2	8	16	24
number of hash functions	k	1.39	5.55	11.1	16.6
false-positive probability	f	0.393	0.0216	$4.59 \cdot 10^{-4}$	$9.84 \cdot 10^{-6}$

A Bloom filter becomes effective when m= c × 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

PROBABILITY OF FALSE POSITIVES

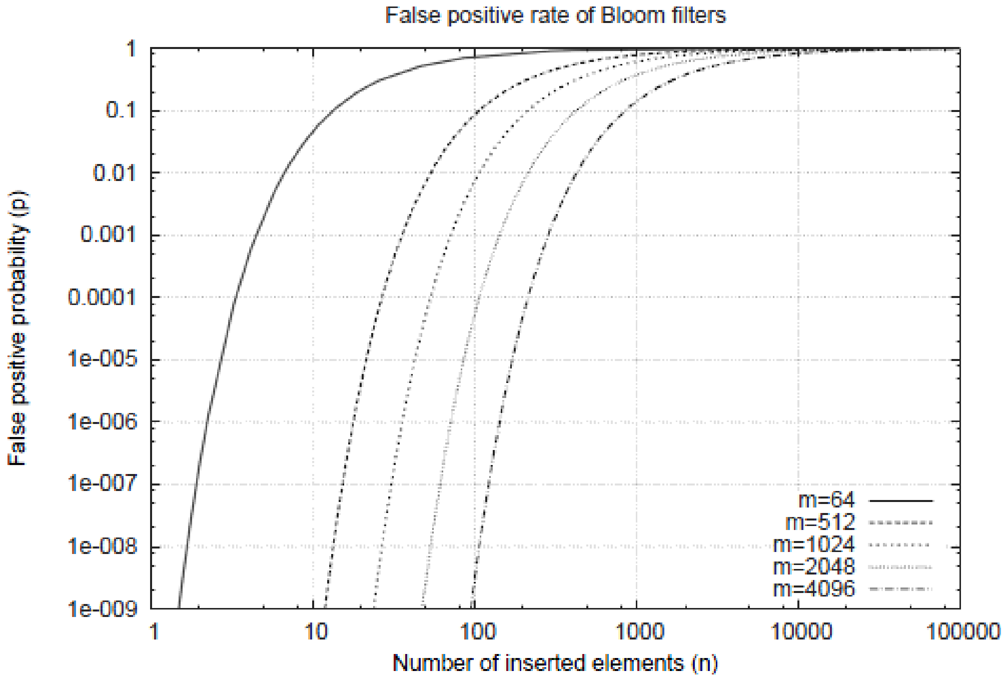
- trade-off between space/number of hash functions/probability of false positives
- if n and m are fixed (fix the number of bits for each element)
  - determine which is the value of k minimizing the probability of false positives
  - compute the derivative of the function of the previous slide so obtaining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

$$k = \frac{m}{n} \ln 2,$$

To this value, corresponds a value of the probability equal $0.62^{m/n}$

PROBABILITY OF FALSE POSITIVES

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



PROBABILITY OF FALSE POSITIVES

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.

BLOOM FILTERS: OPERATIONS

- Union - Given two Bloom Filters, B1 e B2 representing, respectively, the set S1 and S2 through the same number of bits and the same number of hash functions, the Bloom filter representing S1 ∪ S2 is obtained by computing the bitwise OR bit of B1 and B2

- 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: OPERAZIONS

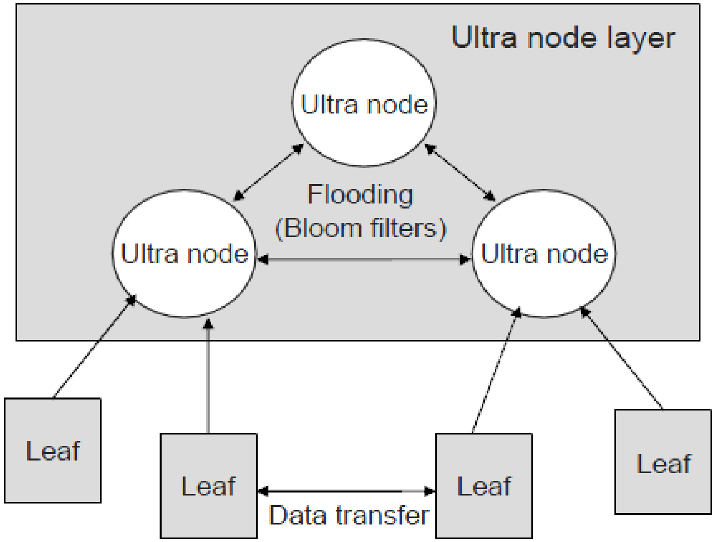
- Intersection: Given two Bloom Filters B1 and B2 representing respectively, the sets S1 and S2 through the same number of bits and the same number of hash functions.
  - Bloom Filter obtained by computing the bitwise and of B1 and B2 approximates S1 ∩ S2
  - as a matter of fact, if a bit is set to 1 in both Bloom filters, this may happen because:
    - this bit corresponds to an element ∈ S1 ∩ S2, therefore it is set to 1 in both filters: in this case no approximation
    - this bit corresponds to an element ∈ S1 - (S1 ∩ S2) and to an element ∈ S2 - (S1 ∩ S2) hence it does not correspond to any element in the intersection

BLOOM FILTER: APPLICATIONS

- Approximate Set Re conciliation (BitTorrent)
- Bloom Filter Based Routing, implemented in Gnutella 0.6
- Several applications in the P2P area....

Bloom Filter routing

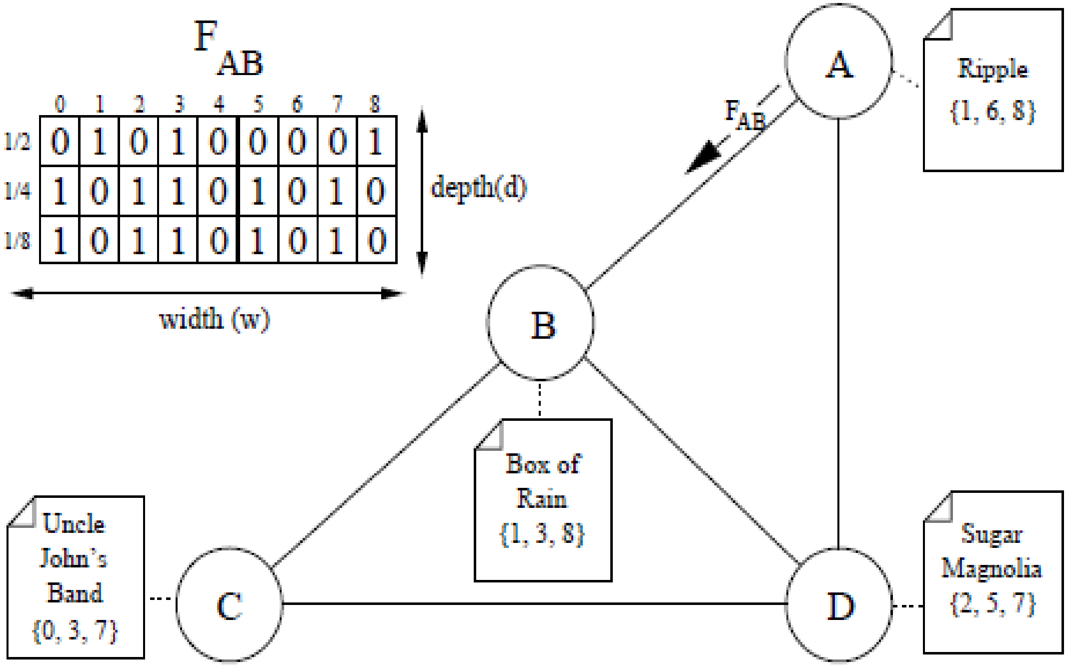
in Gnutella



BLOOM FILTER BASED ROUTING

- a naive implementation in Gnutella 0.6
- a more refined implementation: Attenuated Bloom Filters
- each peer:
  - has a vector of bloom filters for each connection C in the P2P overlay
  - the first filter in the vector summarizes the resources published by the one-hop neighbour connected through C
  - the i-esimo filter is obtained by merging the filters of all the neighbours which can be reached by i hops through the connection C.
  - Bloom Filters guide the routing
  - attenuated: exponentially decreasing waeights are paired with the different filtering levels

ATTENUATED BLOOM FILTERS



F_{AB}

	0	1	2	3	4	5	6	7	8
1/2	0	1	0	1	0	0	0	0	1
1/4	1	0	1	1	0	1	0	1	0
1/8	1	0	1	1	0	1	0	1	0

ATTENUATED BLOOM FILTERS

- Query evaluation with respect to an Attenuated Bloom Filter
  - An example: in the attenuated Bloom Filter F_{AB} the value associated to the query „Uncle Jhon's Band" is 1 /4 +1/8 = 3/8 and this gives the probability that the query is solved by exploiting the connection with node B in 2 or 3 steps

- Query Routing Protocol:
  - depth first search guided by the evaluations returned by the attenuated Bloom Filters

- Update Protocol:
  - flooding of the modified filter with TTL=d
  - to reduce the number of message
    - store the filters of the neighbours and sends only the modified values

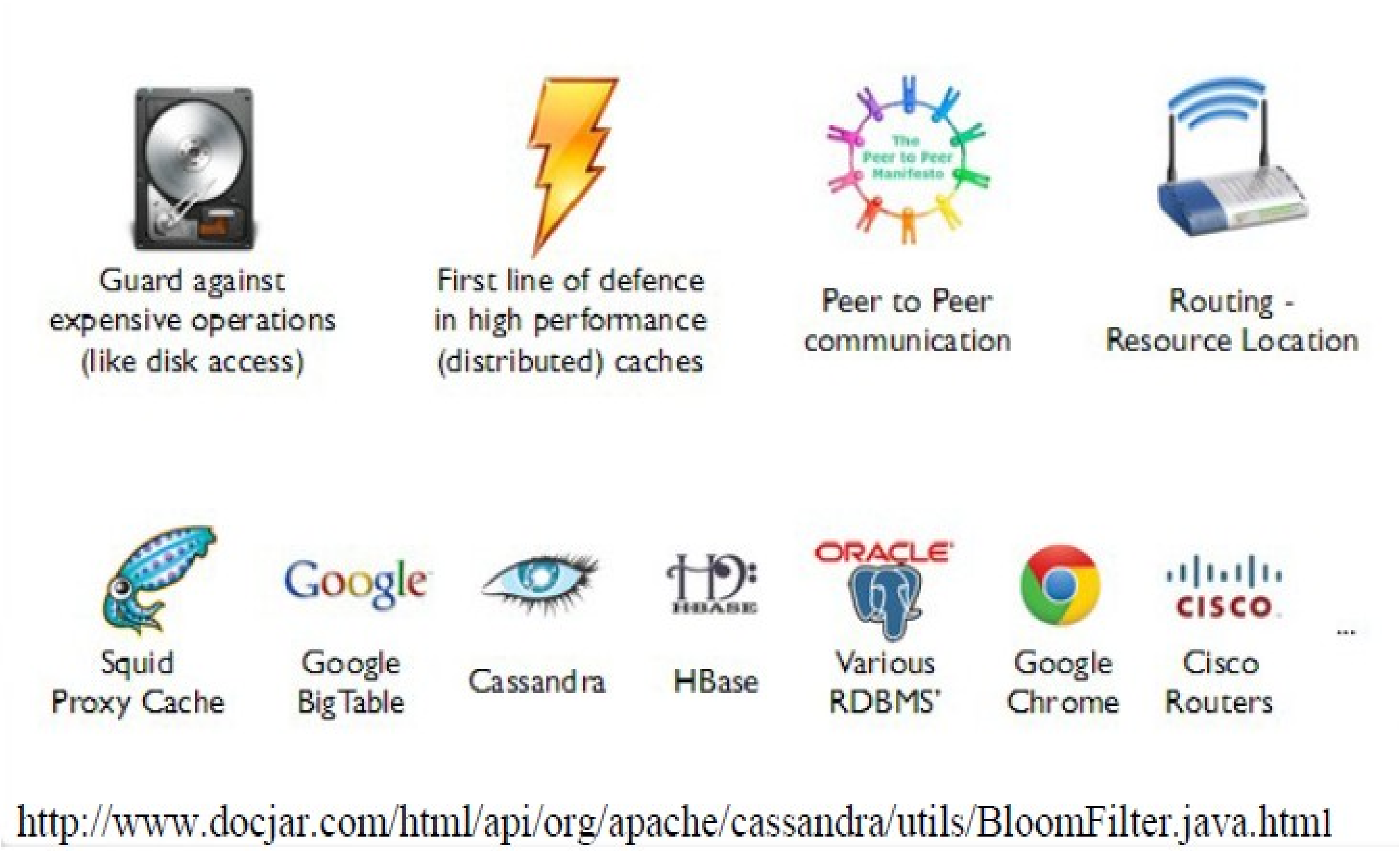
BLOOM FILTERS APPLICATIONS

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

...

http://www.docjar.com/html/api/org/apache/cassandra/utils/BloomFilter.java.html

PROBLEM CONTEXT

- let us first consider the problem context which brings to the definition of the Merkle Trees
- many fragments of a single object are distributed across many nodes of a distributed system
  - a file is divided into chunks and distributed to the peers of a P2P network (eMule, Bittorrent)
- there is a trusted 3rd party
  - must guarantee the integrity of the data
  - may be a trusted server which does not belongs to the P2P network

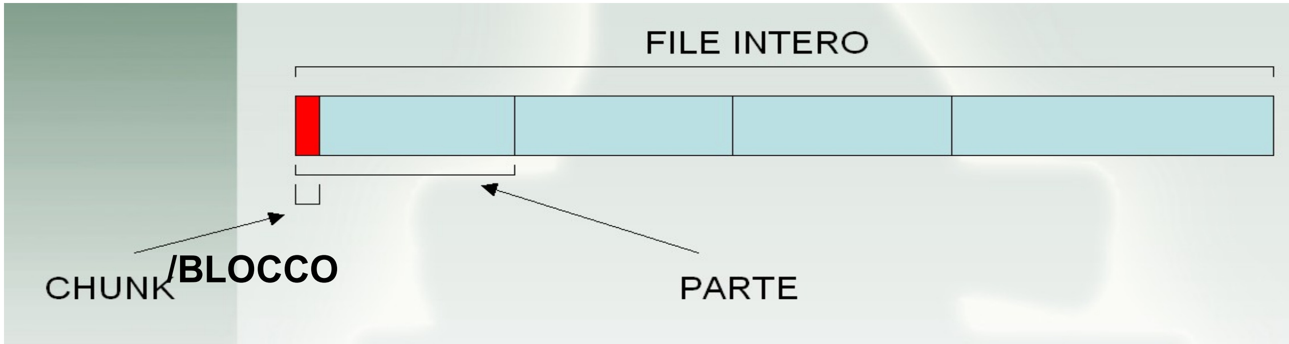
SOLUTION 1: SINGLE HASH

- Hash the object
  - the 3rd party stores the hash
- pros:
  - simple
  - small amount of state stored per object
- cons:
  - must retrieve all fragments before checking
  - every write does work proportional to the size of the

SOLUTION 2: HASH EACH FRAGMENT

- Hash each fragment
- the 3rd party stores the hashes
- pros:
  - fine-grained data integrity
  - each write does work proportional to the size of the fragment
- cons:
  - large amount of state per object, i.e. doesn't scale
  - checking entire object requires many hash function checks

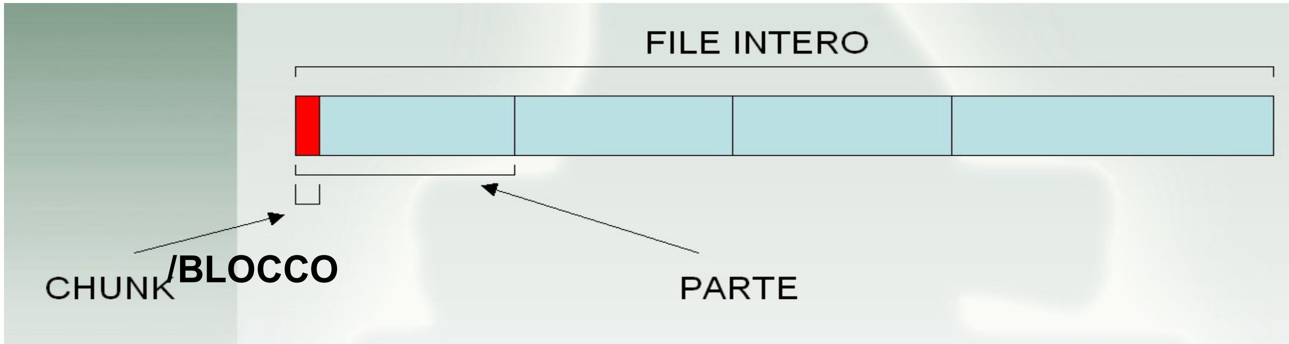
EMULE: INTELLIGENT CORRUPTION HANDLING



- Each file is decomposed into a set of parts and each part is fragmented
  - 9.28 MB part
  - each part is divided into chunks(blocks) of 180K bytes (53 blocks for each part)
- compute a MD4 hash for each part
- HashSet: includes the hash of each part of the file
- Link e2k with HashSet

ed2k://|file|<nome file>|<dimensione file>|<file hash>|p=<hash set>|

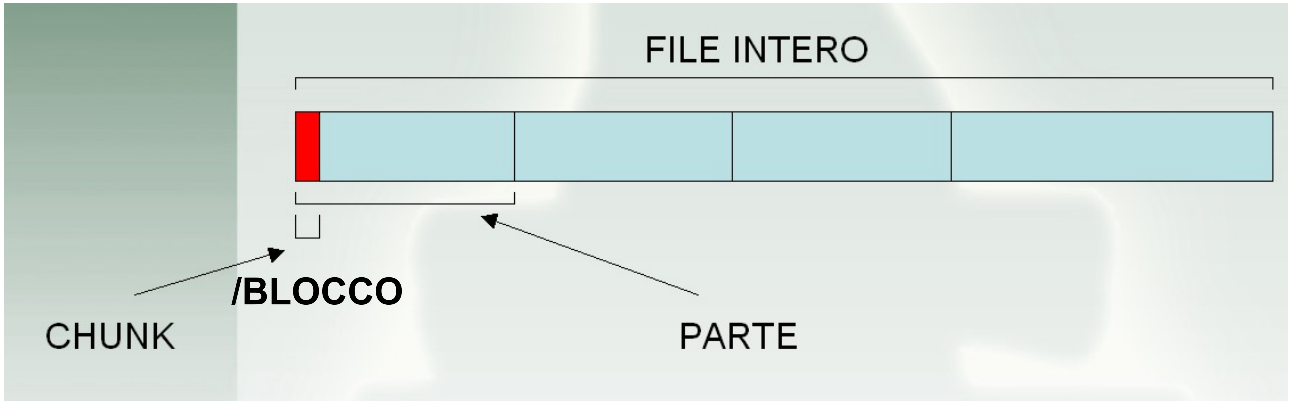
EMULE: INTELLIGENT CORRUPTION HANDLING



- Each file is decomposed into a set of parts and each part is fragmented
  - 9.28 MB part
  - each part is divided into chunks(blocks) of 180K bytes (53 blocks for each part)
- compute a MD4 hash for each part
- HashSet: includes the hash of each part of the file
- Link e2k with HashSet

  ed2k://|file|<nome file>|<dimensione file>|<file hash>|p=<hash set>|

EMULE: INTELLIGENT CORRUPTION HANDLING



An example of e2k link for a file composed by a single part

- ed2k://|file|ubuntu-5.10-install-i386.iso|lungh.file|901E6AA2A6ACD.........

An example of e2k link for a file composto composed by a set of parts

- ed2k://|file|ubuntu-5.10-install-i386.iso|lungh.file|901E6AA2A6ACD......|

  p=264E6F6B.....:17B9A4D1DCE0E4C.....|/

includes the hash of the entire file and of each part of the file

INTELLIGENT CORRUPTION HANDLING

- Basic idea of ICH: repeat the download of each chunk of the part, until the part is repaired

- The client download the hash set of the file before any part

- each time a client download a part of the file, it computes the hash H of that part and compares H with the corresponding value in the hash set
  - if the values are equal, the part is not corrupted and it is put in the shared folder
  - If the values are not equal, the is repeated, chunk by chunk
  - For each chunk/block downloaded, the hash of the entire part is computed. If the part is "repaired", the download of further chunks is avoided

- with this procedure the 50% of the chunk download are avoided on the average

SOLUTION 3: MERKLE TREE

- Hash trees or Merkle trees
  - a data structure summarizing information about a big quantity of data with the goal to check the content

- introduced by Ralph Merkle in 1979

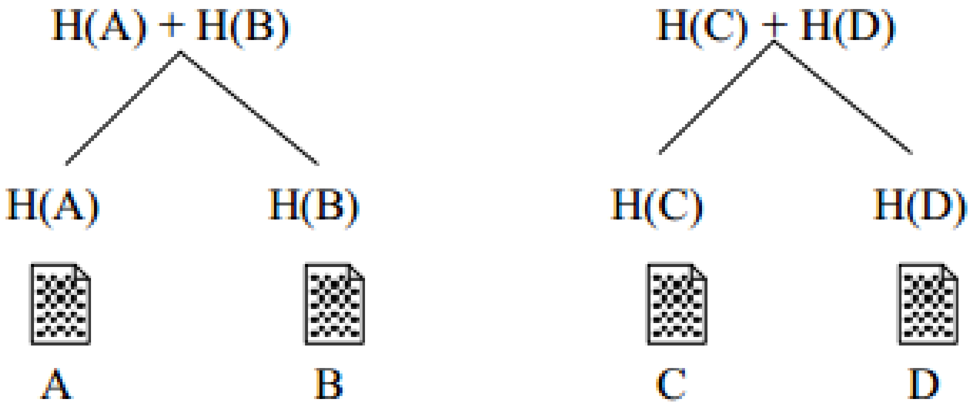
- characteristics
  - simple
  - efficient
  - versatile

- a complete binary tree built starting form a initial set of symbole
  - exploits a hash function H (SHA1, MD5)
  - leaves: H applied to the initial symbols
  - internal nodes: H applied to the sons of a node

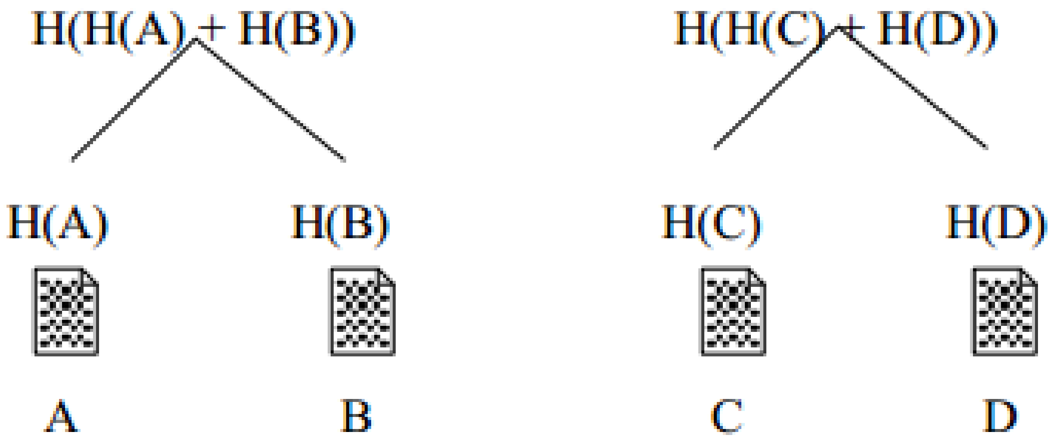
SOLUTION 3: MERKLE TREE

H(A)
H(B)
H(C)
H(D)
A
B
C
D

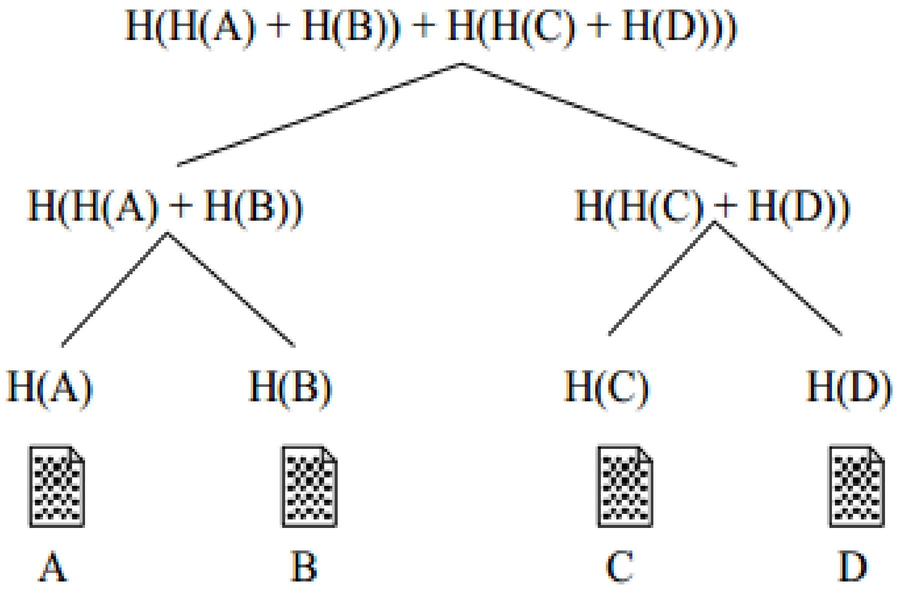
SOLUTION 3: MERKLE TREE



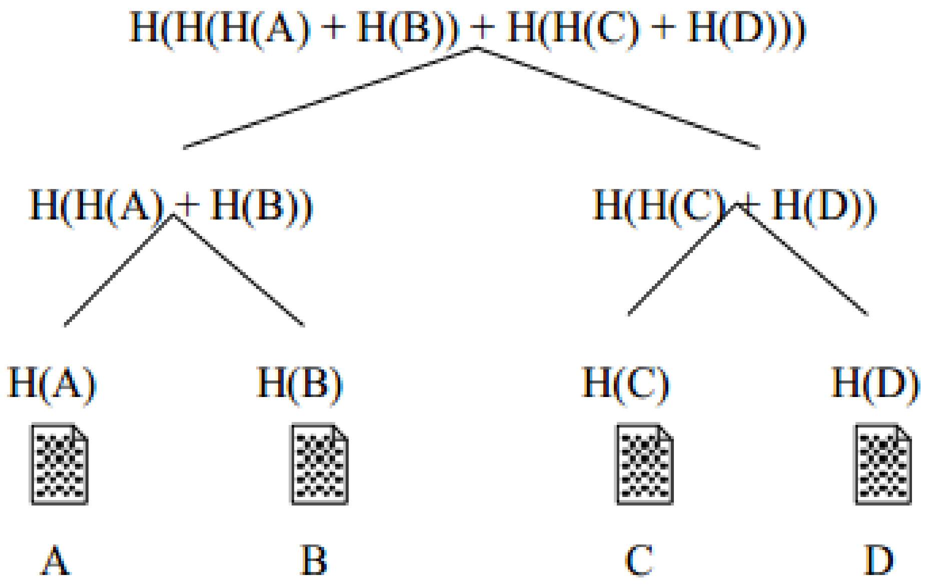
SOLUTION 3: MERKLE TREE



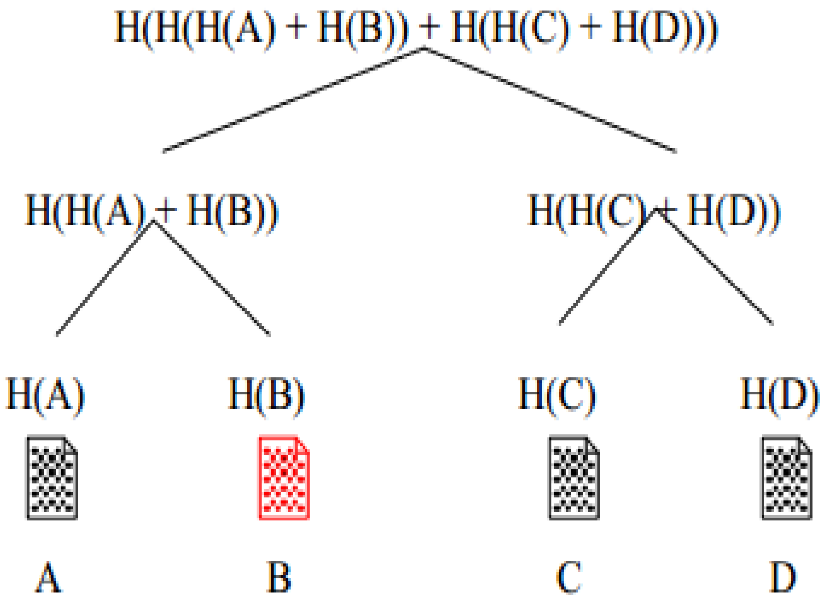
SOLUTION 3: MERKLE TREE



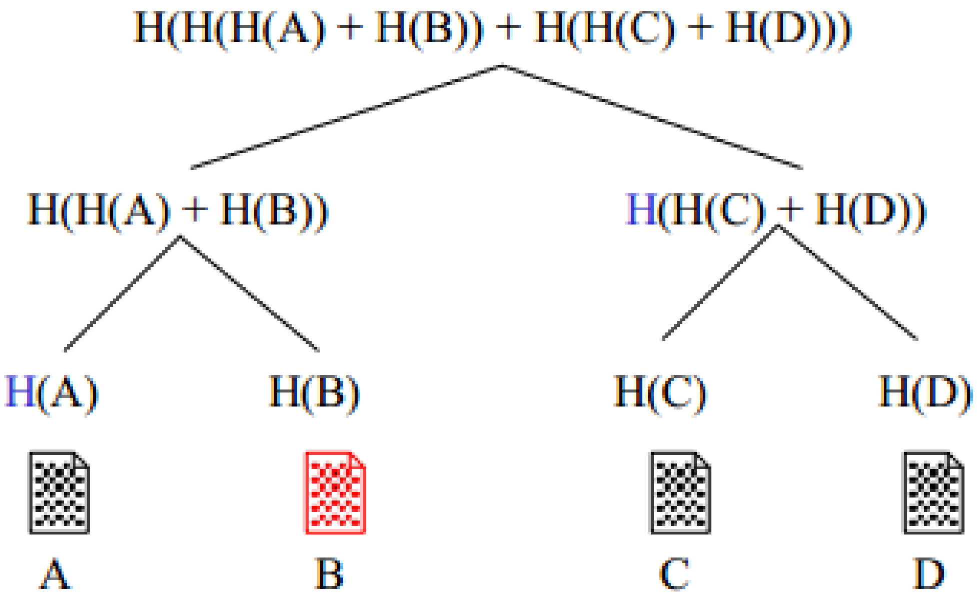
SOLUTION 3: MERKLE TREE



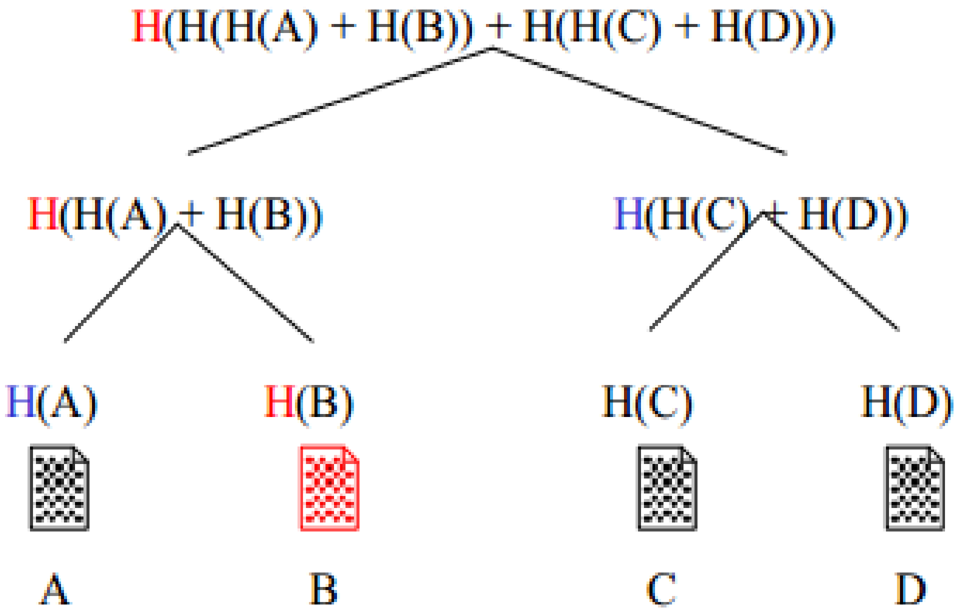
SOLUTION 3: MERKLE TREE



SOLUTION 3: MERKLE TREE



SOLUTION 3: MERKLE TREE



SOLUTION 3: MERKLE TREE

- Build Merkle tree
- 3rd party 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 TREE DEFINITION

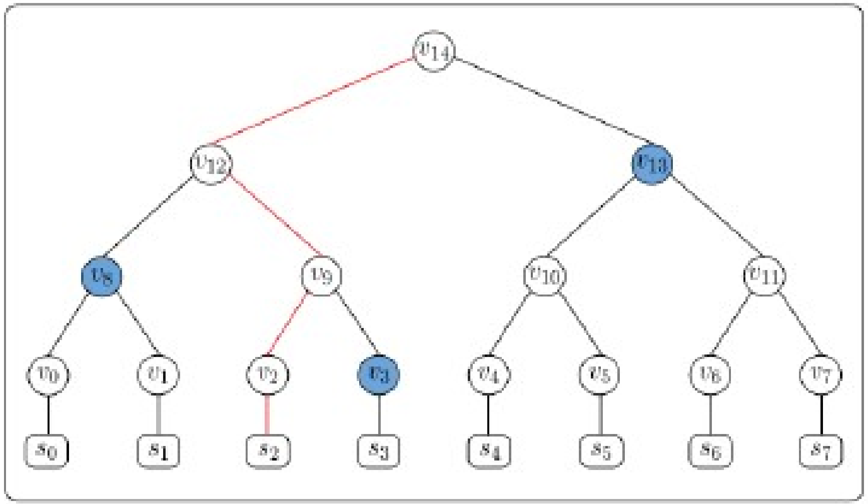
- Let us consider an initial set of symbols S, S={s_1, ..., s_n}, n=2^h, h tree height

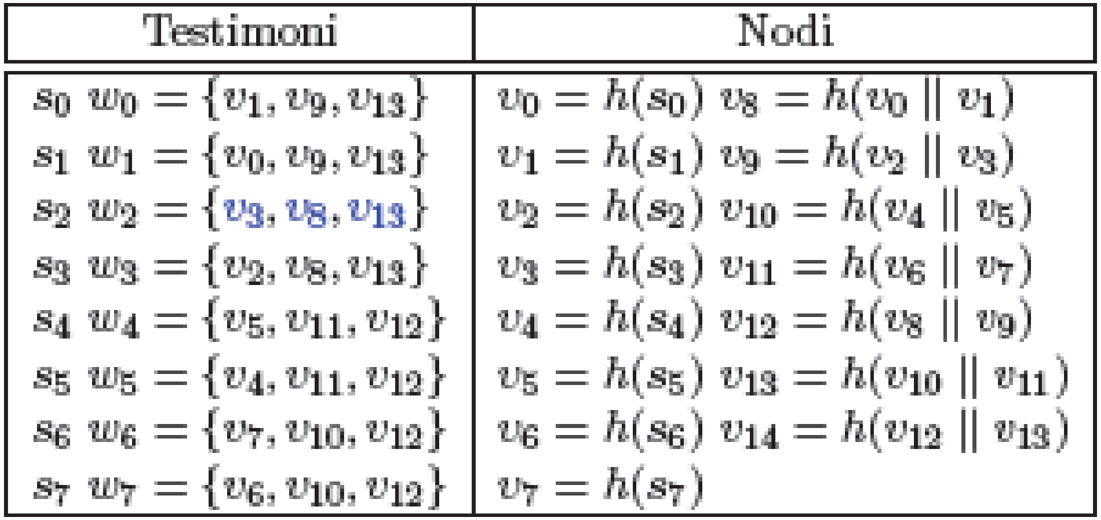
- Merkle Tree MTP Merkle Tree Procedure= <CMT,DMT>

- 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 = 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 = siblings of the nodes on the path from the leaf s_i to the root

MERKLE TREE DEFINITION



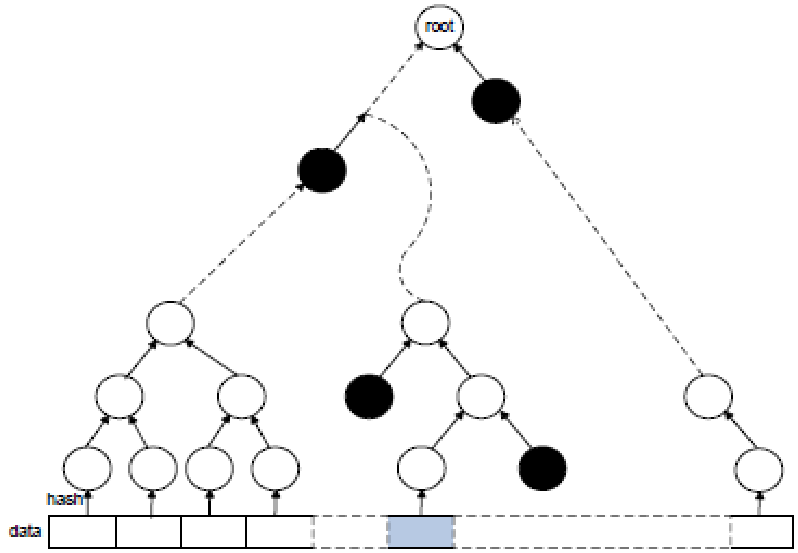


Testimoni	Nodi
s_0 $w_0 = \{v_1, v_9, v_{13}\}$	$v_0 = h(s_0)$ $v_8 = h(v_0 \parallel v_1)$
s_1 $w_1 = \{v_0, v_9, v_{13}\}$	$v_1 = h(s_1)$ $v_9 = h(v_2 \parallel v_3)$
s_2 $w_2 = \{v_3, v_8, v_{13}\}$	$v_2 = h(s_2)$ $v_{10} = h(v_4 \parallel v_5)$
s_3 $w_3 = \{v_2, v_8, v_{13}\}$	$v_3 = h(s_3)$ $v_{11} = h(v_6 \parallel v_7)$
s_4 $w_4 = \{v_5, v_{11}, v_{12}\}$	$v_4 = h(s_4)$ $v_{12} = h(v_8 \parallel v_9)$
s_5 $w_5 = \{v_4, v_{11}, v_{12}\}$	$v_5 = h(s_5)$ $v_{13} = h(v_{10} \parallel v_{11})$
s_6 $w_6 = \{v_7, v_{10}, v_{12}\}$	$v_6 = h(s_6)$ $v_{14} = h(v_{12} \parallel v_{13})$
s_7 $w_7 = \{v_6, v_{10}, v_{12}\}$	$v_7 = h(s_7)$

MERKLE TREE

- DMT(s_i,w_i,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



To verify the authenticity of a block, compute the labels of the internal nodes, through the hash function. If the computed value for the root of the Merkle tree is the same of that found on a trusted server, the block is correct

If someone would corrupt that block, it would be very computationally expensive to define an authentication path returning the correct hash starting from the corrupted value of the block.

AICH:ADVANCED INTELLIGENT CORRUPTION HANDLING

- the standard ICH is pretty effective although it has its limitations as only the whole 9.28 MB can be verified and no finer blocks

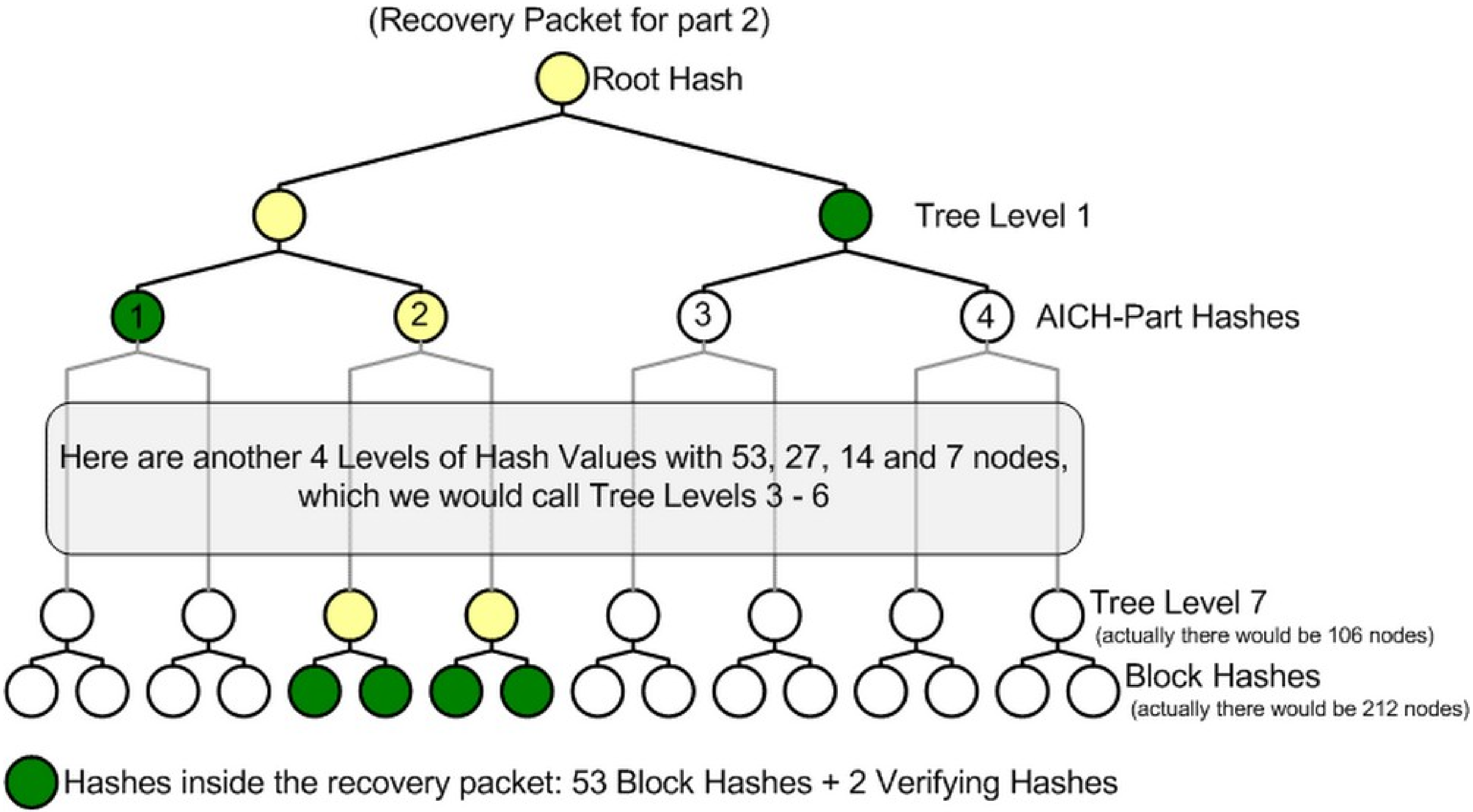
- AICH creates much finer hashes and cares about their integrity

- recall: 9.28 MB parts in a file. Each part is divided into 180 KB blocks, resulting in 53 blocks for each part

- AICH computes a hash value for each block (block hash) using the SHA1 hash algorithm
  - the size of this hash set may be huge: 24.000 hash for a file of 4 GB

- The hash set of a file: set of hashes of all the blocks of the file
  - starting from the block hashes, the hash set is organized as a Merkle tree computed by eMule for each shared file
  - stored in /config/known_64.met

APPLICATION OF MERKLE TREE: EMULE


(Recovery Packet for part 2)
Root Hash
Tree Level 1
1
2
3
4
AICH-Part Hashes
Here are another 4 Levels of Hash Values with 53, 27, 14 and 7 nodes, which we would call Tree Levels 3 - 6
Tree Level 7
(actually there would be 106 nodes)
Block Hashes
(actually there would be 212 nodes)
Hashes inside the recovery packet: 53 Block Hashes + 2 Verifying Hashes

AICH:ADVANCED INTELLIGENT CORRUPTION HANDLING

- when a peer detects a corrupted part, it looks for a "recovery packet"
  - it includes the 53 block-hash of the corrupted part
  - a “Merkle witness” or check-hash for that packet to verify its correctness
  - the number of witness is such that $2^x \geq$ [n° parts] , where x is the number of the witness (check hash).

- when the peer detecting the corrupted part receives the check-hash
  - checks the correspondence between the trusted root hash and the root computed through the check hashes
  - if the two root hashes are equal, it compares the 53 hashes of the corrupted part with those received in the packet. When a correspondence is detected, the block is maintained, otherwise the block is corrupted and it is downloaded again

- for instance, if a single byte is corrupted, eMule maintains all the blocks, with the exception of the single block including the corrupted byte

AICH:ADVANCED INTELLIGENT CORRUPTION HANDLING

- The Root Hash must be trusted

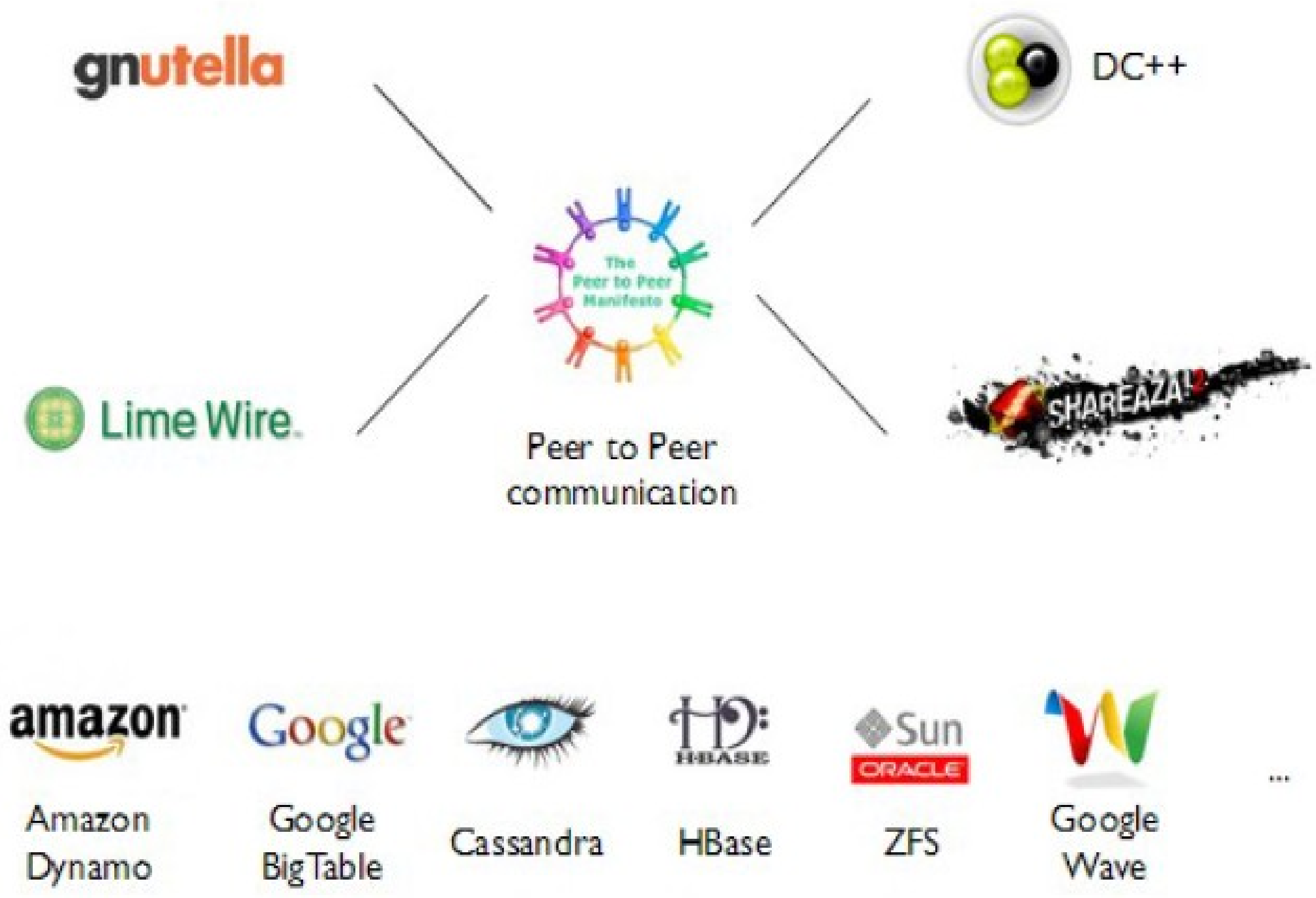
- If the root hash is included in a link e2k and the server publishing that link is considered trusted, the Root hash is considered trusted as well

- Otherwise, the link may be returned from the peers which are fonts of the file

- In this case the root hash is considered valid if at has received the root hash from at least 10 different fonts
  - voting algorithm

- If the AICH system cannot be exploited, because a root hash has not been detected, eMule switches to the ICH to recover the corrupted part

MERKLE TREES: APPLICATION


gnutella
DC++
The Peer to Peer Manifesto
Lime Wire
SHAREAZA
Peer to Peer communication
amazon
Amazon Dynamo
Google
Google BigTable
Cassandra
HBASE
HBase
Sun
ORACLE
ZFS
Google Wave
...