authenticated data structures

B. Palazzi contributed to early versions of these slides. All mistakes are mine.

Authenticated Data Structure (ADS)

- an ADS is a data structure that is “easy” to check for integrity, even for parts of it
- basics
  - collects elements
  - associates a cryptographic hash h with its content
    - h is called root hash or basis
    - value of $h \leftrightarrow$ content of the ADS
- integrity verification
  - queries: come with a *proof* that can be checked against h
  - updates: update h

typical use cases

- by using an ADS, a client can detect small tampering in large data set, efficiently

- typical applications
  - legal
    - “legal” proof of correctness or tampering of storage
    - service level agreement verification
  - backup check
  - cloud
  - cryptocurrencies

cloud storage example

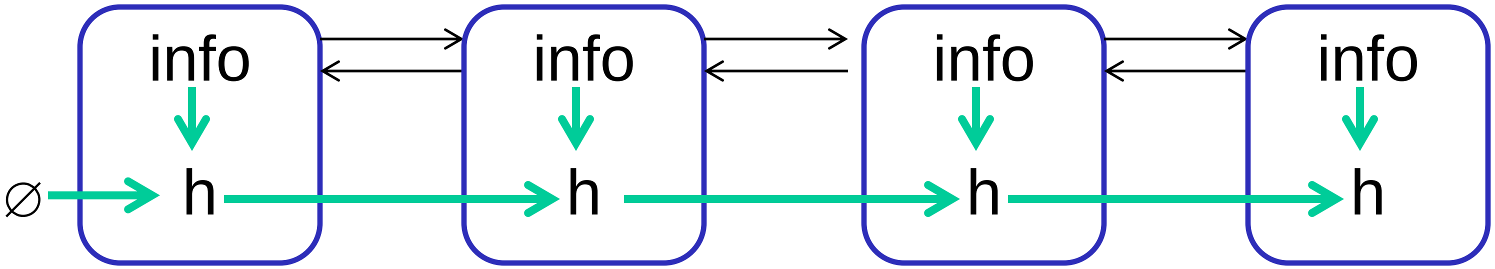
- cloud based storage
  - virtually unlimited, cheap, **untrusted**
- local storage
  - limited, expensive, **trusted**
  - e.g. IoT device, mobile, your PC
- store a large dataset on the cloud and just h locally
- equip the dataset with an ADS
  - query results, with their proof, are checked against trusted h
  - updates change remote dataset, remote ADS and local h

(some) ADS quality metrics

- as for regular data structures
  - time complexity for queries
  - time complexity for updates
  - space overhead
- plus...
  - time complexity for proof construction
  - time complexity for proof check
  - space complexity for proof

a very simple ADS: authenticated list

- a linked list plus...
- ... each element contain a field *h*
  h = hash(info | prev.h)



- each h is a crypt. hash of current info and all previous info

authenticated list: (in)efficiency

- append an element $O(1)$
- update of info of a generic element $O(n)$
  - n is the number of elements
  - this is not $O(1)$, all following hashes should be updated!
- query $O(n)$
- proof space $O(n)$, time $O(n)$
  - it is made of previous h and all subsequent info
- closely related with Bitcoin blockchain
  - where append is the most important operation

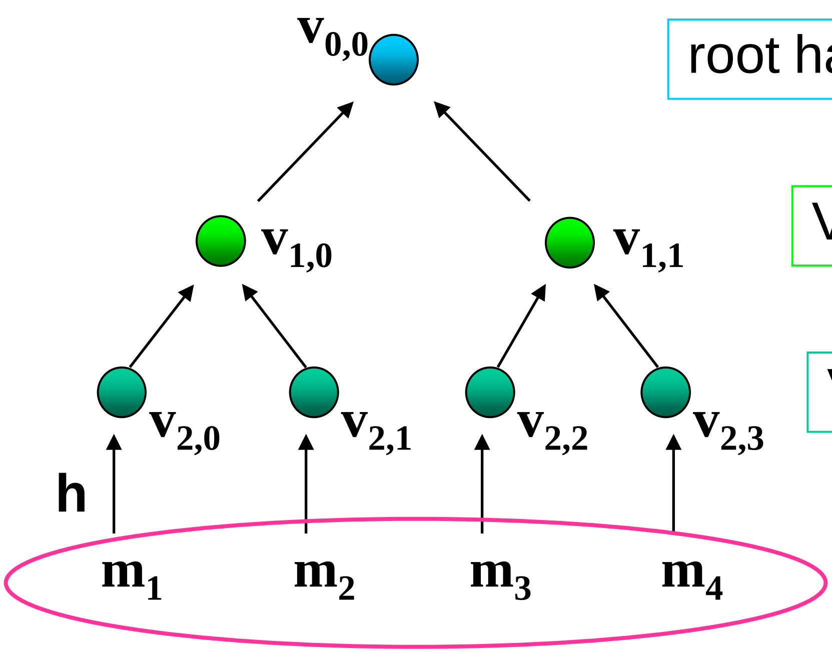
other ADSes

- Merkle Hash Tree (MHT)
  - a.k.a Merkle Tree or Hash Tree
- authenticated skip list

- static or dynamic
  - e.g. for backup check a static data structure is ok
  - MHT are mostly used in their static flavor
- deterministic or randomized
  - skip list are typically randomized

MHT: how does it work

- a (balanced binary) tree
- each node *v* contains a hash of the data associated with leaves of the subtree rooted at *v*



root hash= $V_{0,0} = h(V_{1,0} | V_{1,1})$

$V_{1,1} = h(V_{2,2} | V_{2,3})$

$V_{2,2} = h(m_3)$

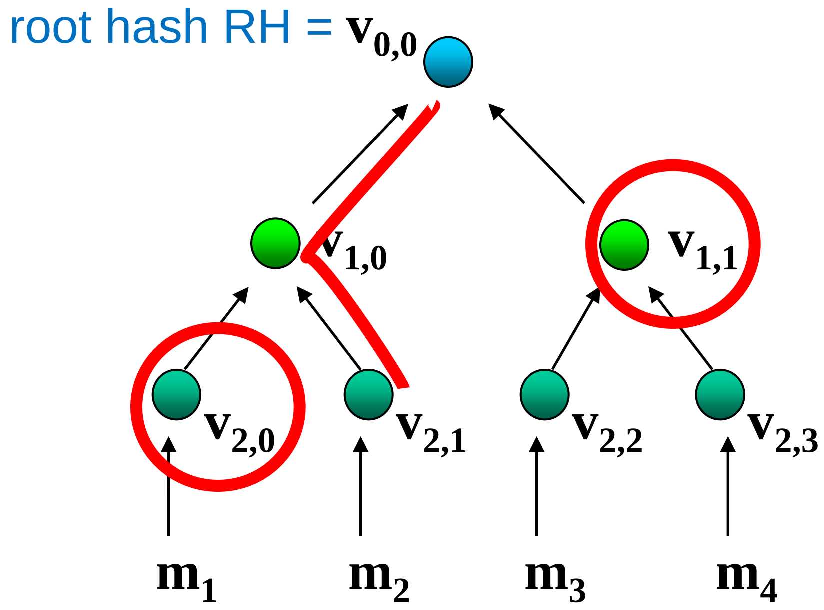
$V_{2,3} = h(m_4)$

data must be ordered

h(.) is a cryptographic hash function

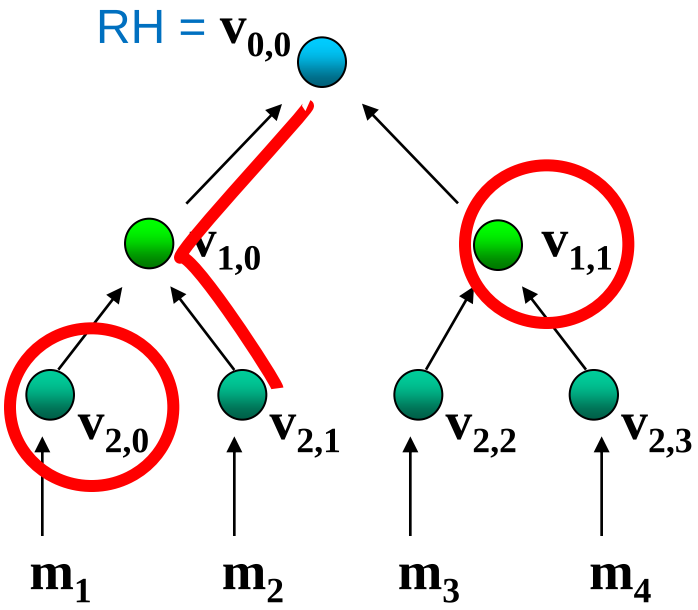
MHT: query verification

- proof for m_i:
  - consider the path p from m_i to root (excluded)
  - the proof is made of “steps”, one for each node v of p
  - each step is a pair
    - label **L**eft or **R**ight depending on how parent of v is entered
    - (hash in the) sibling of v
- example: m_2
  - p = $\mathbf{v}_{2,1}\,\mathbf{v}_{1,0}$
  - proof
    - R $\mathbf{v}_{2,0}$
    - L $\mathbf{v}_{1,1}$



MHT: query verification

- suppose that verifier has a trusted version of the root hash: tRH
- procedure for integrity check
  - from proof re-compute RH, in the example $RH = h(h(\mathbf{v}_{2,0} | h(m_2)) | \mathbf{v}_{1,1})$
  - compare RH == tRH



MHT: query verification semantic

- client is sure that the data of the reply comes from the dataset associated with the trusted version of the root hash

MHT: query verification

- correctness (no false positives)
  - client reconstructs part of the MHT
- security (no false negatives)
  - i.e., tampering of data or MHT, but same RH
  - means that attacker has found a collision for the cryptographic hash

MHT: efficiency

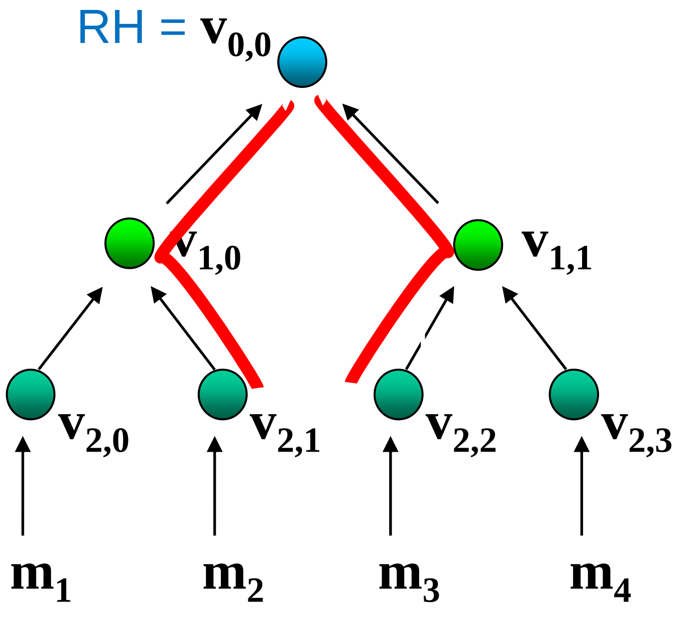
- for a balanced MHT creating and checking a proof is efficient
- length of the proof is O(log *n*)
  - *n:* size of the stored data

MHT: query verification (for empty result)

- proving absence is equivalent to proving two elements are consecutive
  - for ordered sets
- consider proofs for *m* and *m*' (*m* < *m*')
- *m* and *m'* are consecutive iff the label sequences of their proofs satisfy the following system of regular expressions
  - labels of proof of m = *x***L***z*
    labels of proof of m' = y**R***z*
    x = **R***
    y = **L***
  - for perfectly balanced trees |x|=|y|, z possibly empty

MHT: query verification (for empty result)

- check:
  - isolate common part in the two poofs (z)
  - check label sequences for the non common part of the paths (should be R*L and L*R)
- example: prove that $\mathbf{m_2}$ $\mathbf{m_3}$ are consecutive
  - common path empty
    - just the root is common
  - proof for $\mathbf{m_2}$
    RL
  - proof for $\mathbf{m_3}$
    LR



MHT: query verification (for empty result)

correctness and security derive from...

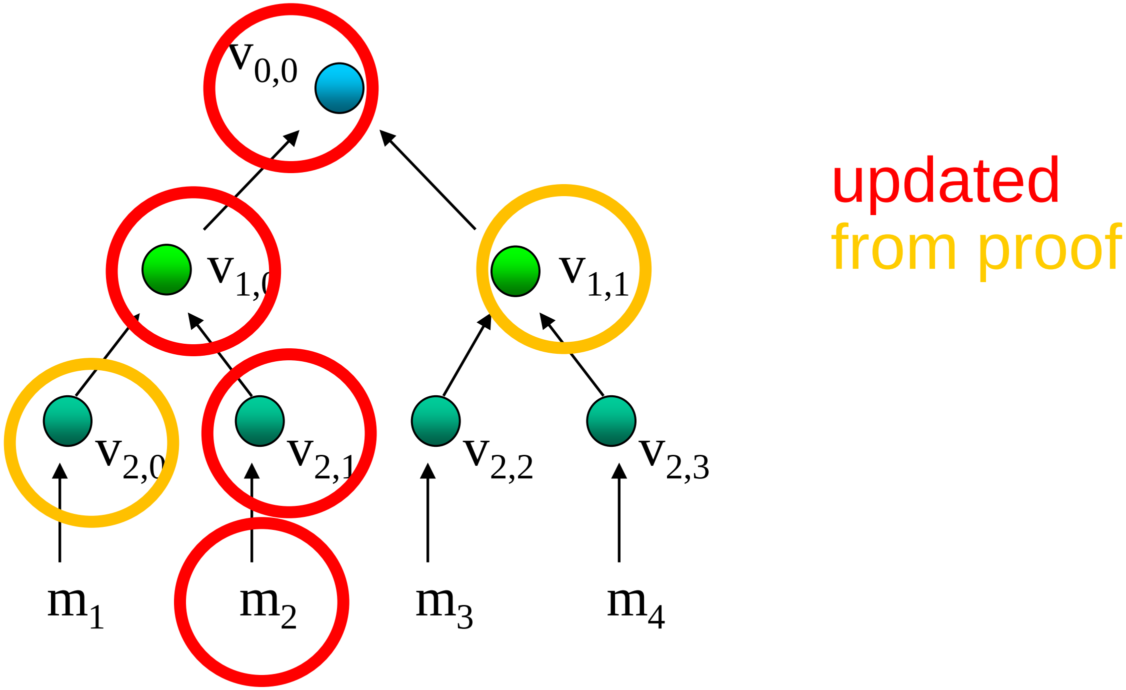
- correctness and security of proofs of m and m’
- correspondence between structure of the tree and the regular expressions

MHT: update

- we have to update *m* to a new version *m'*
  - root hash will change as well as several internal hashes
- procedure on the trusted side (e.g. client)
  - get proof *p* for *m* and check it
  - compute the new hashes of the path to the root following *p* substituting *m'* in place of *m*
  - the lastly computed hash is the new trusted root hash
- procedure on the untrusted side (e.g. server)
  - update the hashes of the path to the root substituting *m'* in place of *m*

MHT: update

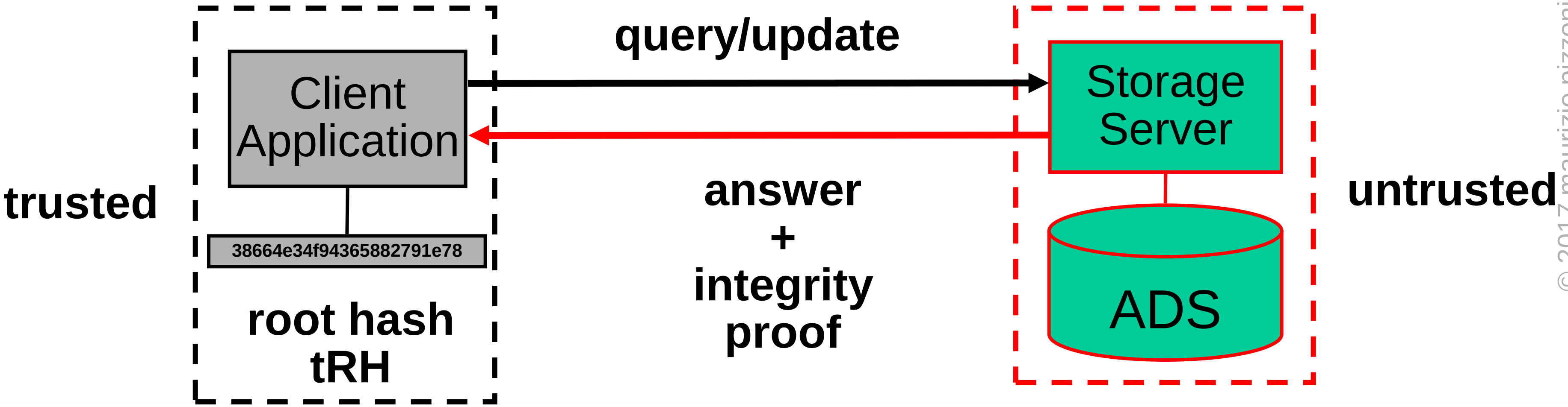
- example: update m_2 to a new version m_2'



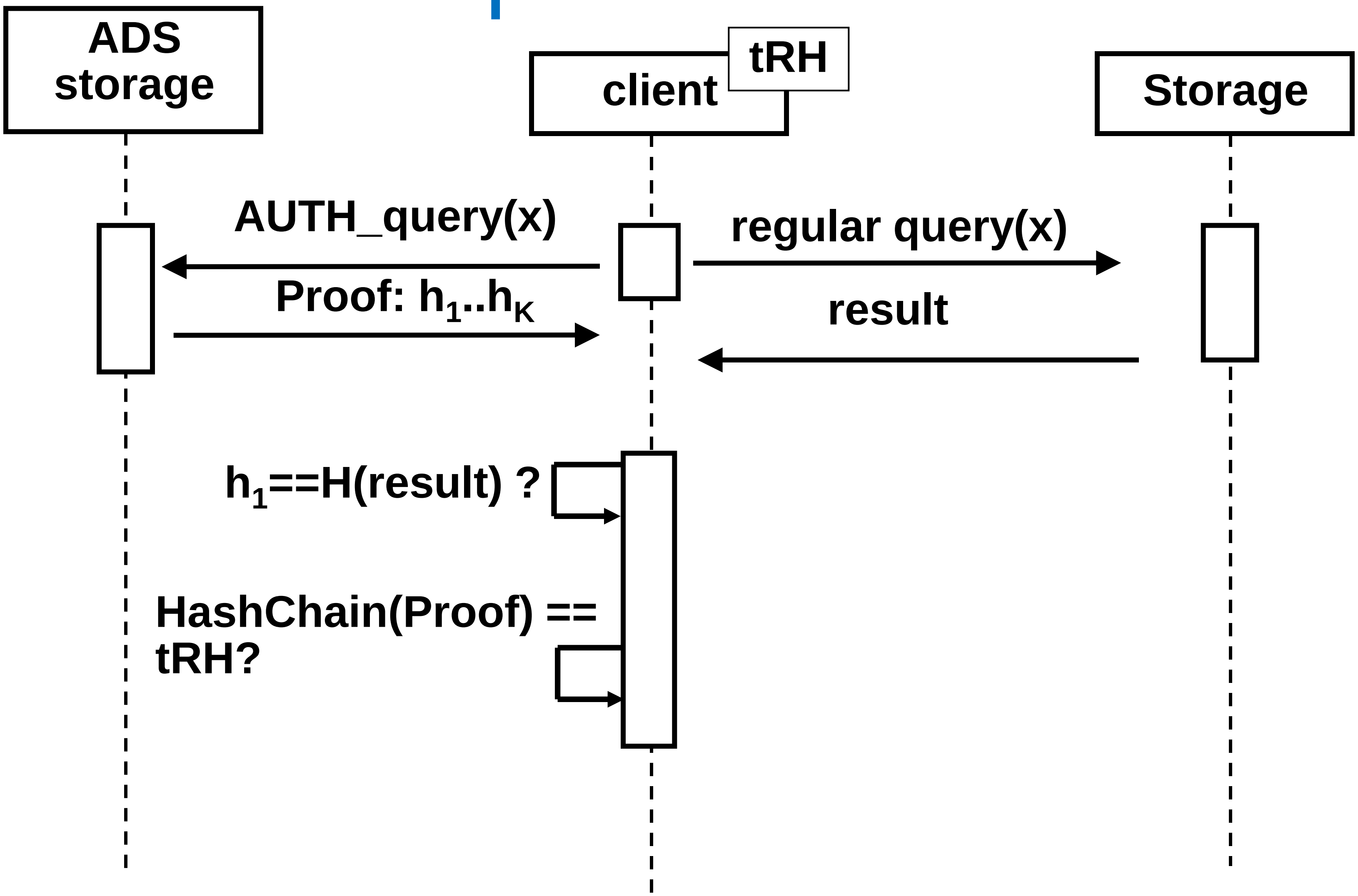
- O(log *n*) time for balanced trees

an ADS use case: check for malicious cloud server

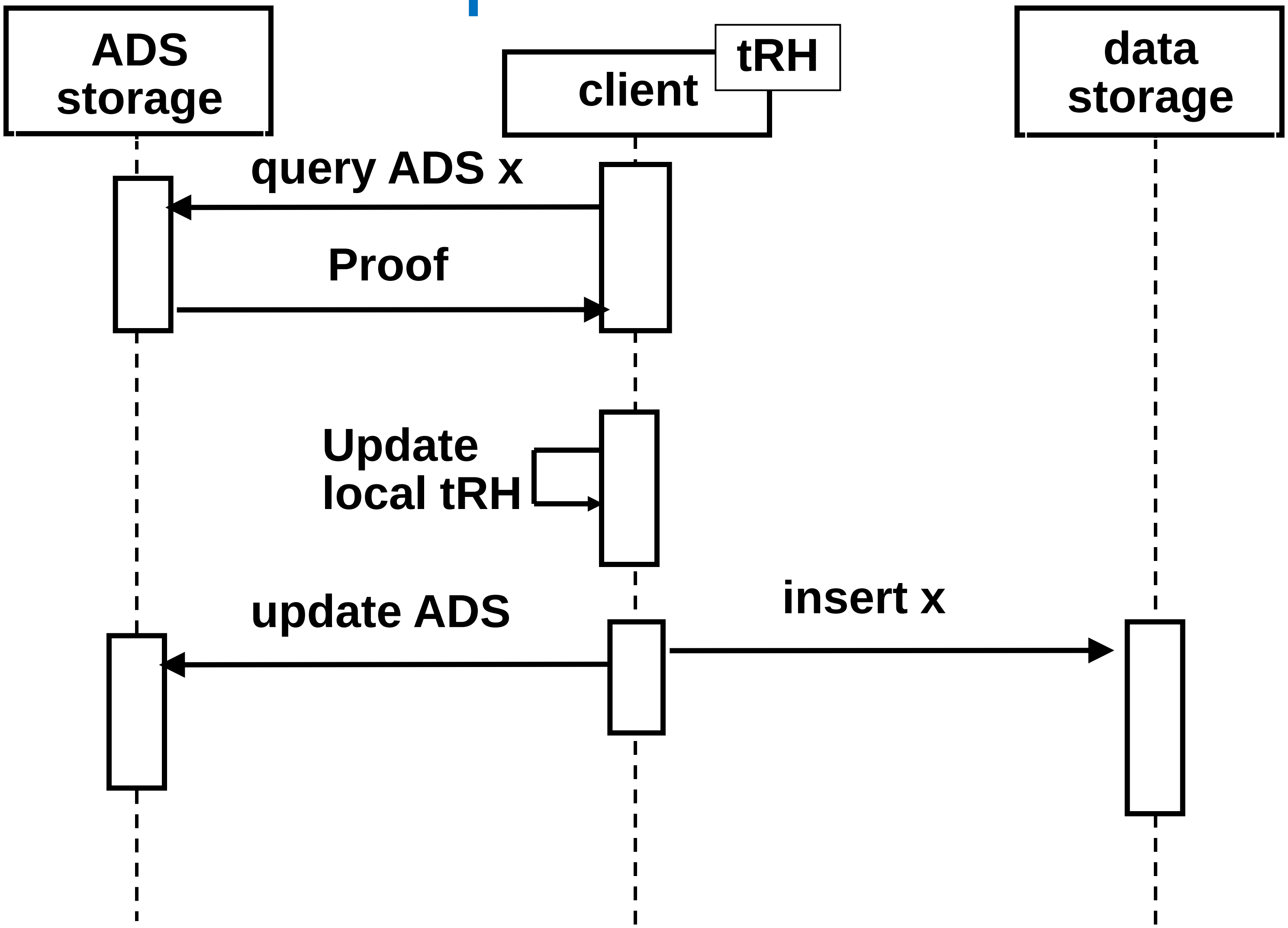
- client stores root hash locally
- ADS can be stored in cloud
- ADS can be applied to regular cloud storage
  - i.e., storage might not know about ADS
  - ADS should be properly represented in the storage



ADS authenticated query protocol



ADS authenticated update protocol



security remarks

- tampering with the ADS cannot lead to undetected data tampering
- if an ADS is lost, it could be re-created from data
- caveat: usually root hash depends not only by data but also from ADS internal structure (e.g. tree balancing)