Advanced Distributed Algorithms and Data Structures

Chapter 11: Applications

Christian Scheideler

Institut für Informatik

Universität Paderborn

Overview

- Monitoring
- Information system
- Publish-subscribe system
- Crypto currencies

Monitoring

Problem variants:

- determine whether there is a time point where a system predicate is true
- report all time intervals (within a given time frame) at which a system predicate is true
- evaluate a function over the system state for a given time point

Standard setting: a dedicated process called monitor is collecting information from all processes in the system

Useful:

- Logical clocks: use hybrid clocks combined with physical clock synchronization
- Convergecast: compute intersection of local intervals at which predicates are true along convergecast tree. Note: number of intervals does not increase when computing intersections. Intervals can be sent in order and merged on their way to the monitor.

Monitoring

Examples:

- Determine the average degree of the processes or the total number of connections in the network at a given time point
- Determine the number of ongoing search requests in the system at a given time point
- Determine how many processes have joined or left the network within a given time interval

Overview

- Monitoring
- Information system
- Publish-subscribe system
- Crypto currencies

Information System

Standard dictionary:

- Every element e is identified by a unique key key(e).
- insert(e): stores e under key(e) in the dictionary (resp. updates the element previously stored under key(e) to e)
- delete(x): removes element e from the dictionary with key(e)=x
- lookup(x): returns the element e with key(e)=x
  (if such an element exists, and otherwise $\perp$)

Challenge: store information among the processes so that following properties hold:

- Availability: every request can be served in finite time, even under an adversarial attack
- Integrity: the information returned from a lookup request is correct

Information System

Challenge: store information among the processes so that following properties hold:

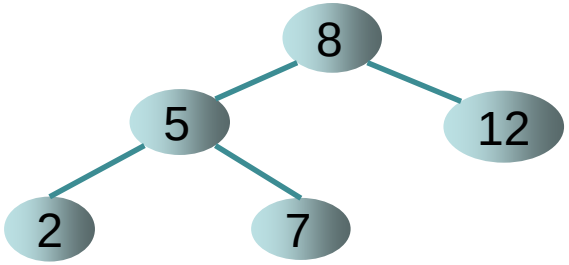
- Availability: every request can be served in finite time, even under an adversarial attack
- Integrity: the information returned from a lookup request is correct

Useful:

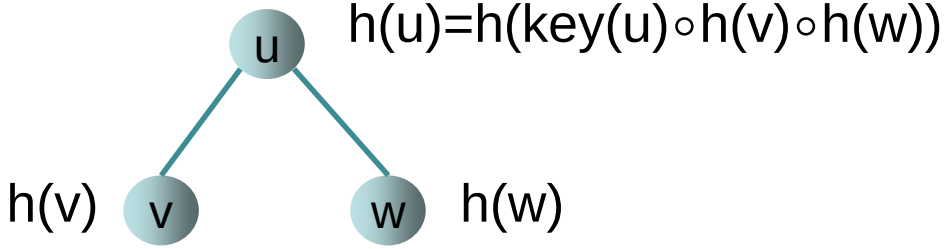
- Consistent hashing: even distribution of data among processes
- Merkle hash tree: needed for integrity

Merkle Hash Tree

- Binary search tree T

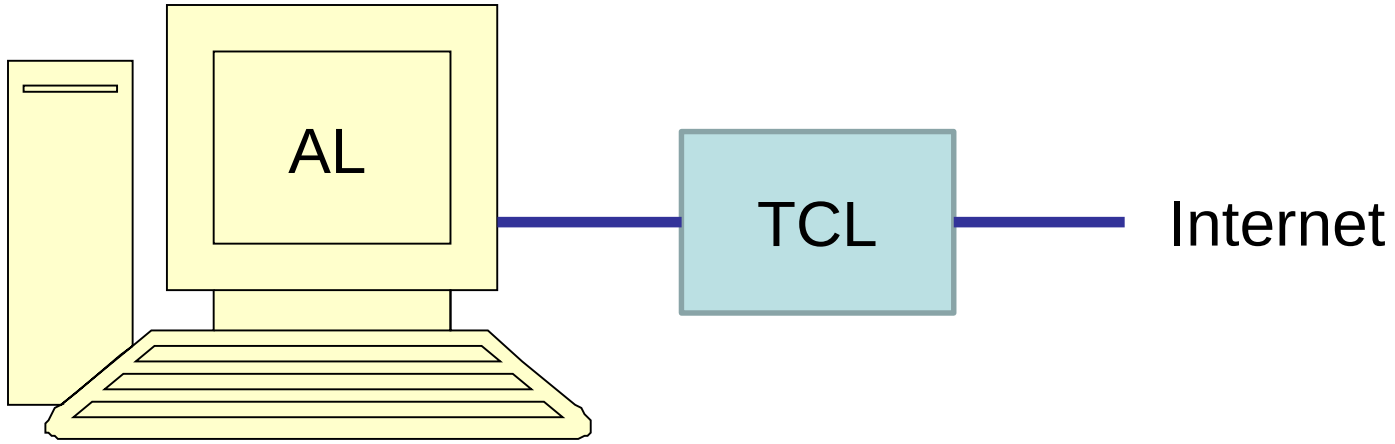


- A Merkle hash tree is a binary search tree in which hash values are computed in a bottom-up manner using a one-way hash function h



Merkle Hash Tree

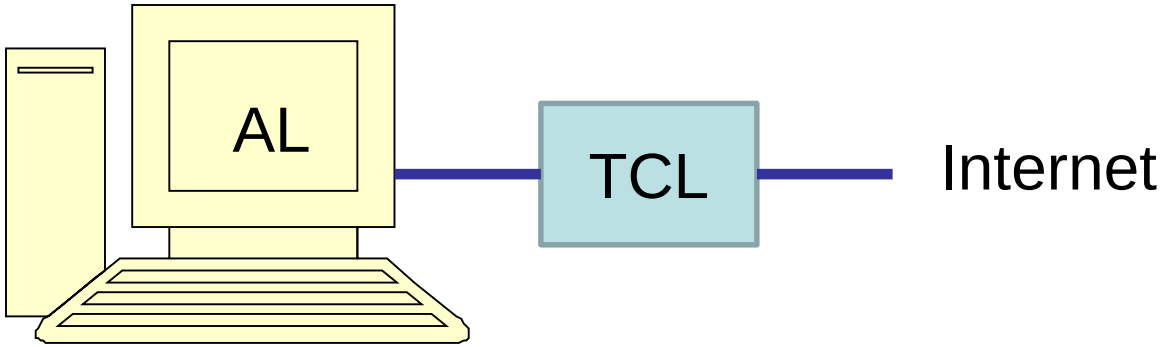
Storage of Merkle hash tree:



- AL: stores Merkle hash tree (and h)
- TCL: stores h and a digest d:=h(root) (i.e., the root hash of the Merkle hash tree)

Merkle Hash Tree

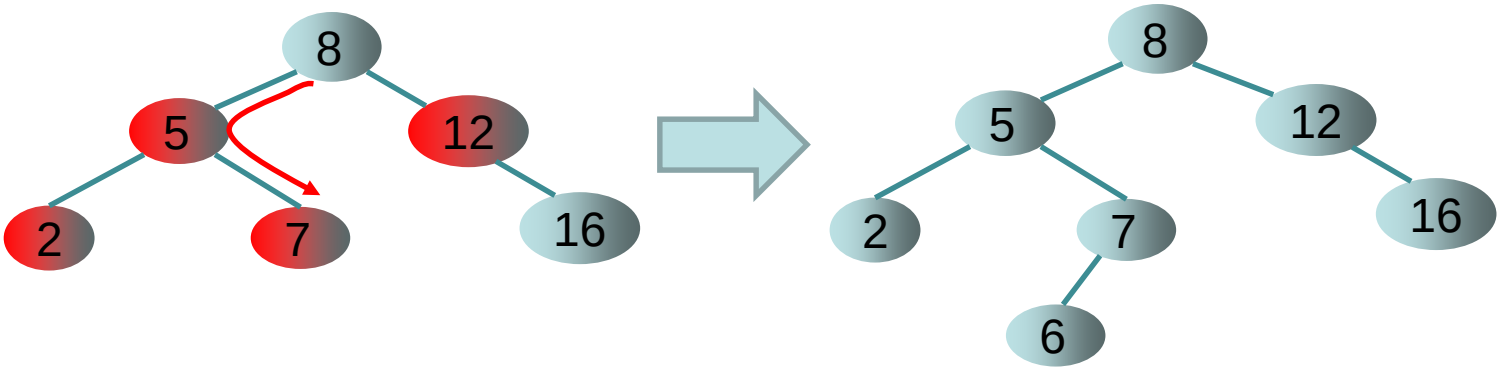
Processing of insert(e) request:



- AL: sends hash values of search path to the TCL, computes update of tree,
- TCL: checks the reported hash values and updates its digest to the new h(root)

Merkle Hash Tree

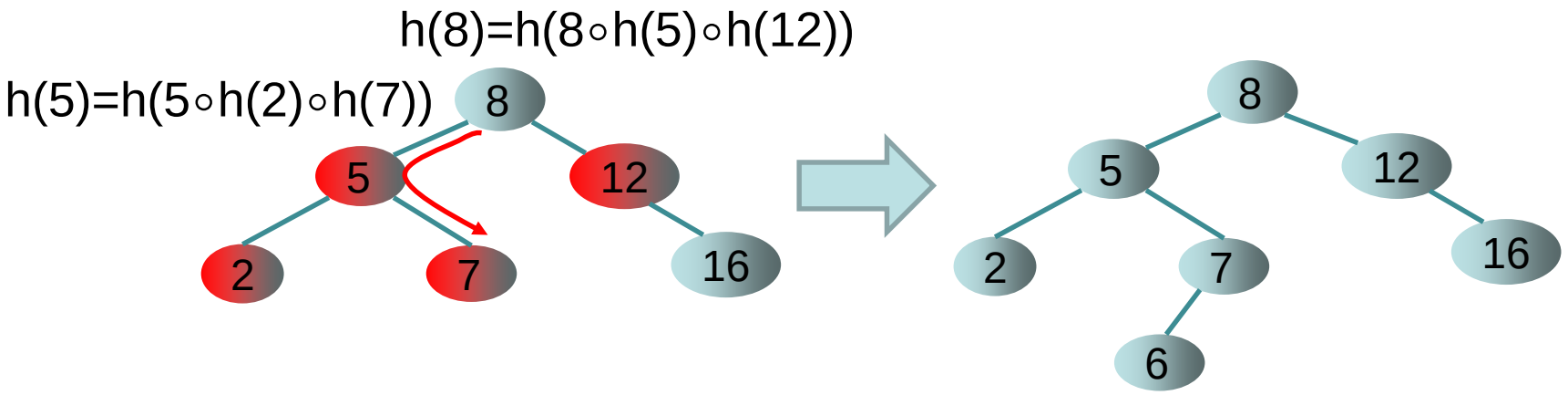
Example: insert(6)



- AL sends hashes of all children of the nodes along the unique search path in the old tree from the root to the node below which the new element is inserted to the TCL (see nodes marked in red)

Merkle Hash Tree

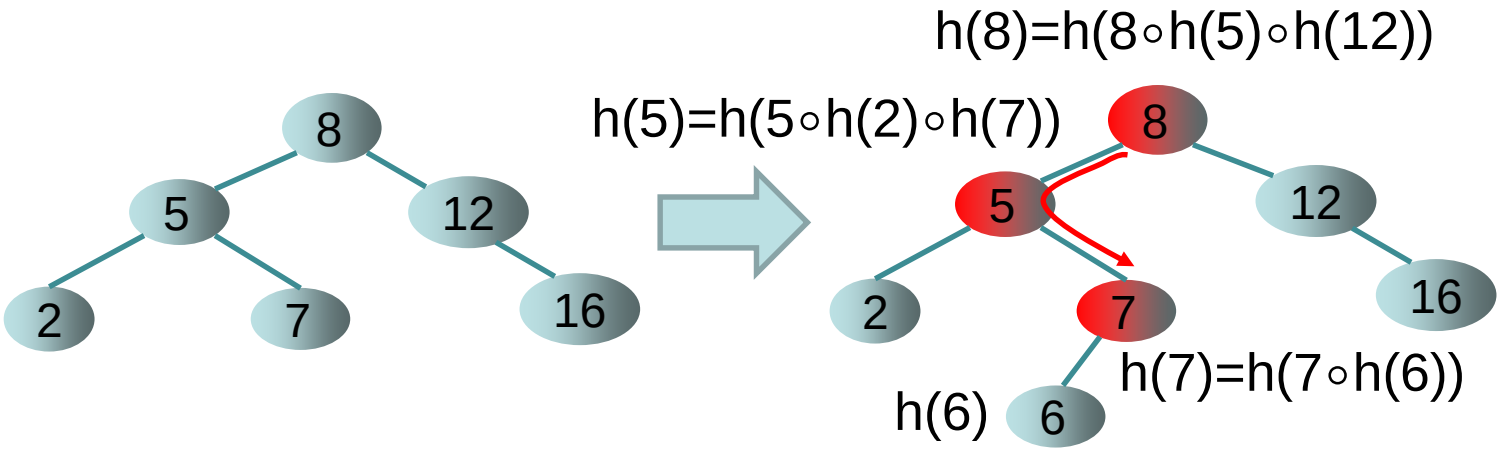
Example: insert(6)



- The TCL checks bottom-up for every node v along the search path in the old tree whether $h(v)=h(key(v) \circ h(w_1) \circ h(w_2))$.
- With the digest d the TCL can also check whether d=h(root).
- If all equalities hold, the TCL accepts the hashes, otherwise it rejects them

Merkle Hash Tree

Example: insert(6)

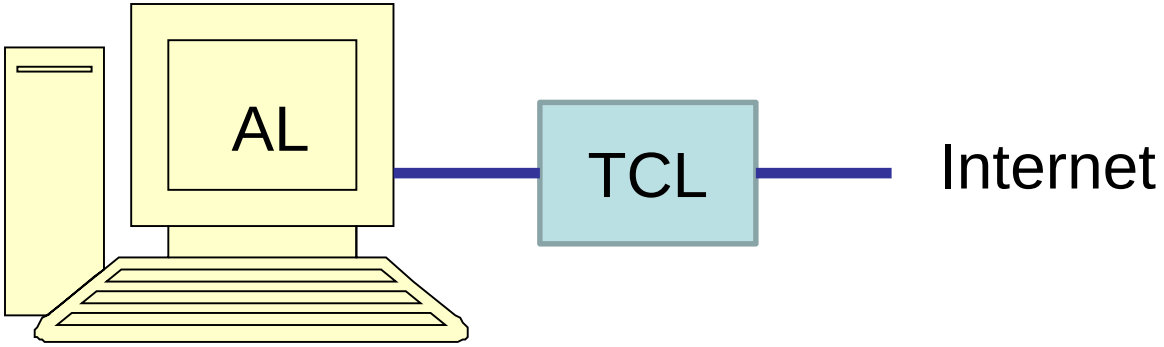


- If the hashes are correct, the TCL recomputes the hashes of the nodes along the search path in the new tree and updates its digest correspondingly
  in our example: it sets d to the new h(8)
- Also the AL updates its hashes in the same way

Merkle Hash Tree

Processing of delete(k) request:

Assumption: k exists in the tree.

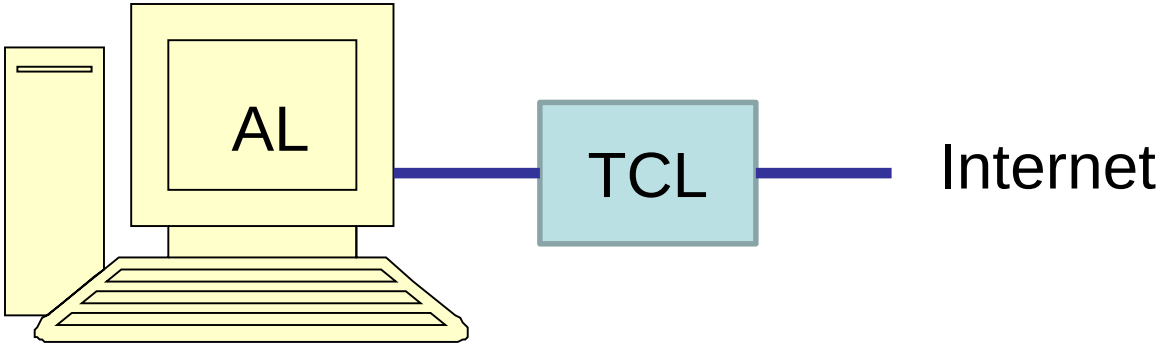


- AL: sends hash values of search path to the TCL (like in insert), computes update of tree
- TCL: checks the reported hash values and updates its digest d to the tree without k

Merkle Hash Tree

Processing of lookup(k) request:

Assumption: k exists in the tree.



- AL: sends together with the element e with key(e)=k the hash values of the search path to the TCL (like in insert)
- TCL: checks the reported hash values and accepts e if they result in its digest

Merkle Hash Tree

Basic conditions:

- Completeness: For any query q, the AL can generate an answer-proof pair (a,p) that is accepted by the TCL.
- Soundness: If, given a query q, an answer-proof pair (a,p) is accepted by the TCL, then a is the correct answer to q.

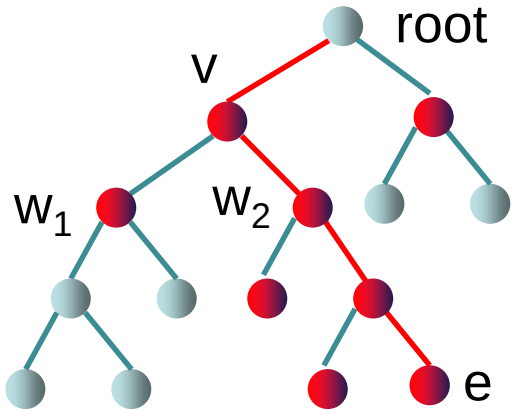
Merkle Hash Tree

Completeness:

- For any x with an element e with key(e)=x, the AL can certainly generate an answer-proof pair accepted by TCL.

Soundness:

- The AL cannot fool the TCL into accepting some element e´ that is not part of T because then there must exist a node v on the path with $h(v)=h(h(w_1)\circ h(w_2))$ where h(v) is the correct value of v but $h(w_1)$ or $h(w_2)$ is a value not occurring in the tree. Since h is a one-way hash function, the AL cannot do that within a reasonable time bound.



Problem: what if x is not in T?

Merkle Hash Tree

Problem: what if x is not in T?

Solution:

- Include dummy elements with keys $-\infty$ and ∞ in T.
- For every element e in T (except ∞), store (x,succ(x)) together with e in its node in T, where key(e)=x and succ(x) is the closest successor of x, and construct the Merkle hash tree based on these pairs.

Proof for some x not in T:

AL delivers (y,succ(y)) which contains x.

Merkle Hash Tree

Question: Why not just digitally sign every element in T and publish the public key so that the signature can be checked by any user?

Answer: Then it would be hard to revoke an element, i.e., it would open up the possibility of the AL for replay attacks.

Alternatives to Merkle hash trees:

- RSA accumulators
- Bilinear accumulators
- Multilinear accumulators

These are, in theory, better but expensive in practice!

Merkle Hash Trees

References:

- Michael T. Goodrich, Roberto Tamassia, Jasminka Hasic. An Efficient Dynamic and Distributed Cryptographic Accumulator. In ISC 2002, pp. 372-388, 2002.
- Michael T. Goodrich, Roberto Tamassia, Nikos Triandopoulos: Super-Efficient Verification of Dynamic Outsourced Databases. In CT-RSA 2008, pp. 407-424, 2008.
- M. T. Goodrich, D. Nguyen, O. Ohrimenko, C. Papamanthou, R. Tamassia, N. Triandopoulos, C. Videira Lopes. Efficient Verification of Web-Content Searching Through Authenticated Web Crawlers. In VLDB 2012.
- Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated Hash Tables. In CCS 2008.
- Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal Authenticated Data Structures with Multilinear Forms. In Pairing 2010, pp. 246-264, 2010.
- Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal Verification of Operations on Dynamic Sets. In CRYPTO 2011, pp. 91-110, 2011.
- M. Yiu, Y. Lin, K. Mouratidis. Efficient Verification of Shortest Path Search via Authenticated Hints. Proceedings of the IEEE International Conference on Data Engineering (ICDE), pp. 237-248, 2010.

Information System

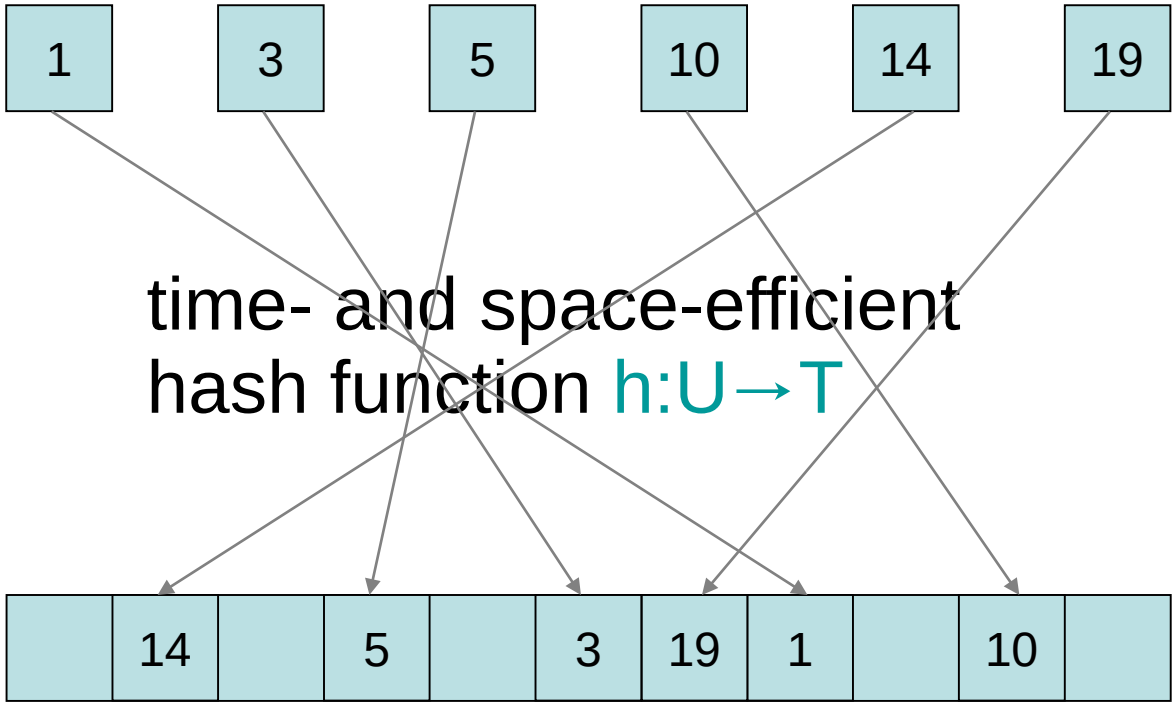
Challenge: store information among the processes so that following properties hold:

- Availability: every request can be served in finite time, even under an adversarial attack
- Integrity: the information returned from a lookup request is correct

Useful:

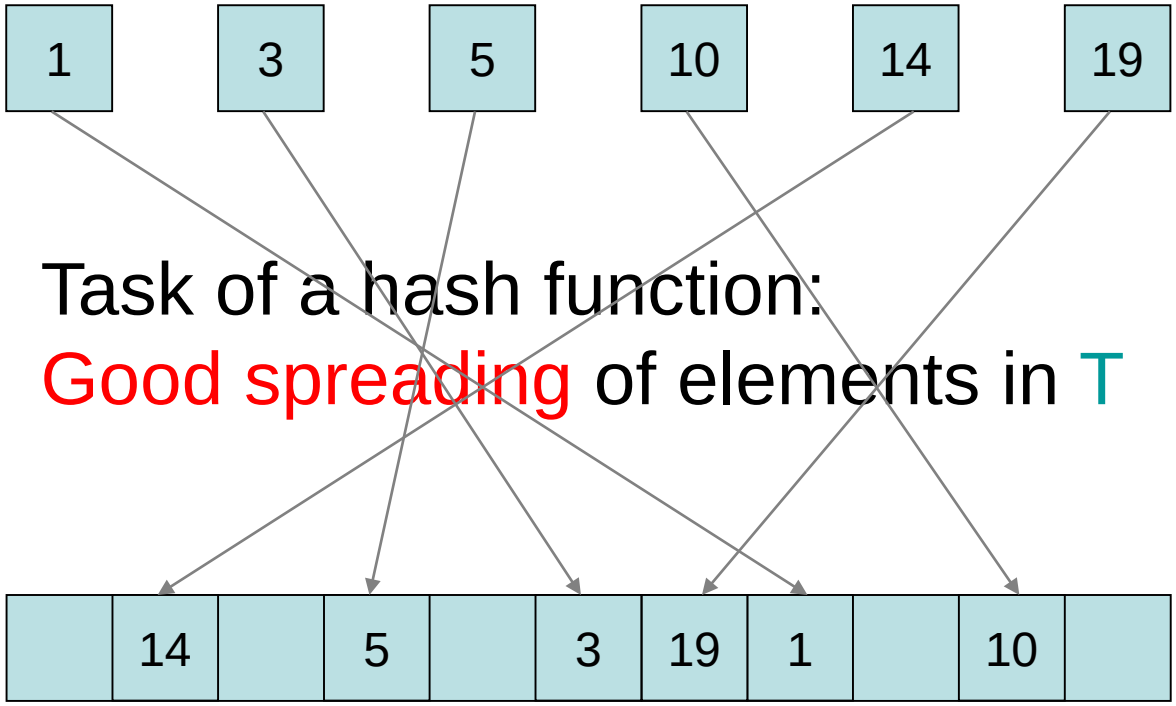
- Consistent hashing: even distribution of data among processes
- Merkle hash tree: needed for integrity

Classical Hashing



hash table T

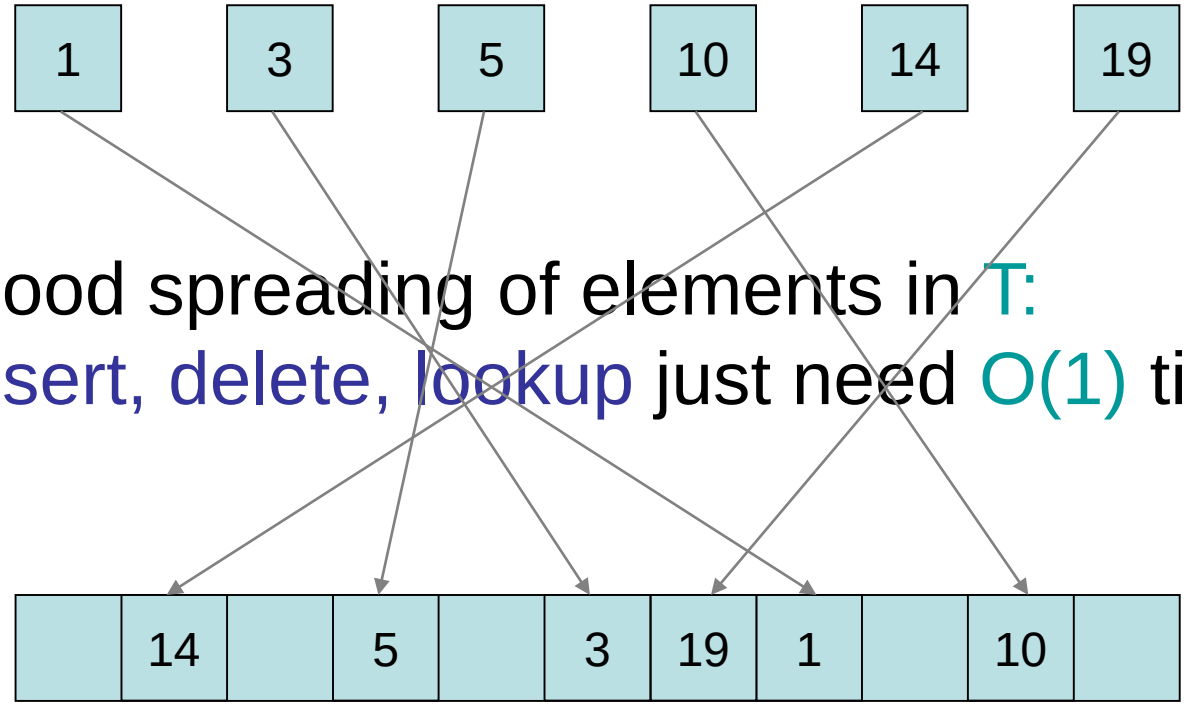
Classical Hashing



Task of a hash function:
Good spreading of elements in T

hash table T

Classical Hashing

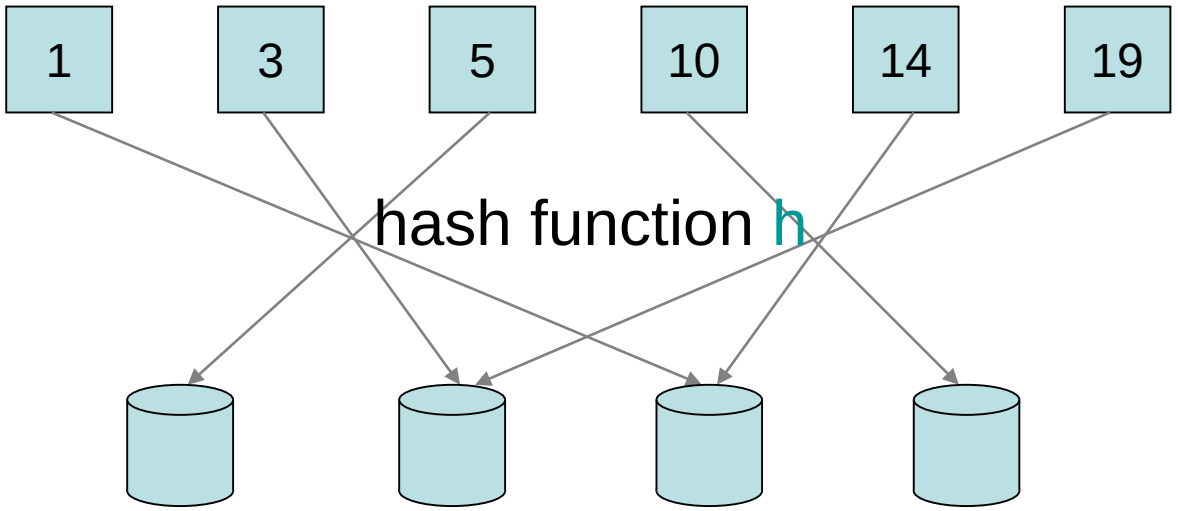


Good spreading of elements in T:
insert, delete, lookup just need O(1) time

hash table T

Distributed Hashing

Hashing also useful for distributed systems:



Problem: set of storage devices as well as their capacities can change over time

Distributed Hashing

Basic Operations:

- insert(e): inserts element e with key key(e) (which will possibly overwrite previously stored information under key(e))
- delete(k): deletes element e with key(e)=k
- lookup(k): outputs element e with key(e)=k
- join(v): device v joins the system
- leave(v): device v leaves the system

Distributed Hashing

Requirements:

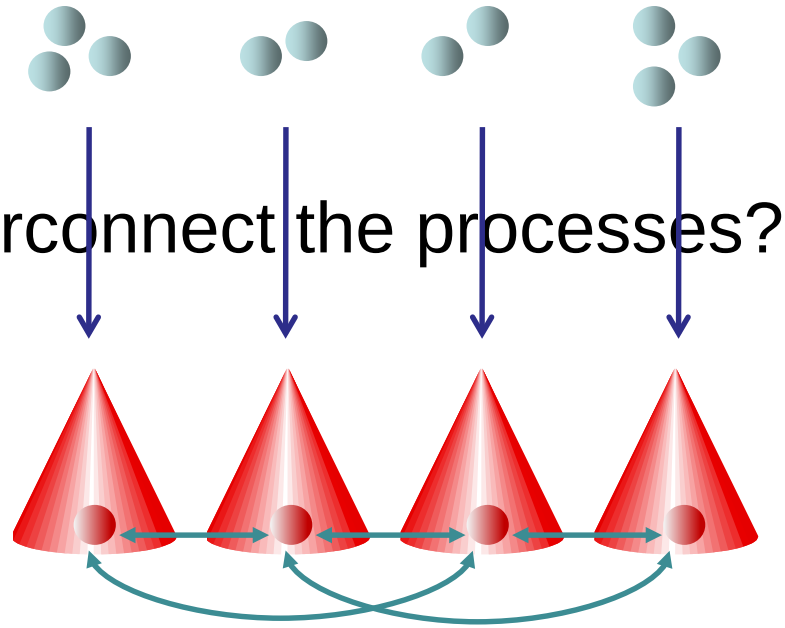
1. Fairness: Every storage device with c% of the overall capacity should get c% of the elements on expectation.
2. Efficiency: The computation of the storage location should be time- and space-efficient.
3. Redundancy: The copies of an element should be stored in different storage devices.
4. Adaptivity: For every change in capacity in the system by c%, only O(c%) of the elements should be replaced to preserve properties 1-3.

Uniform storage systems: consistent hashing

Consistent Hashing

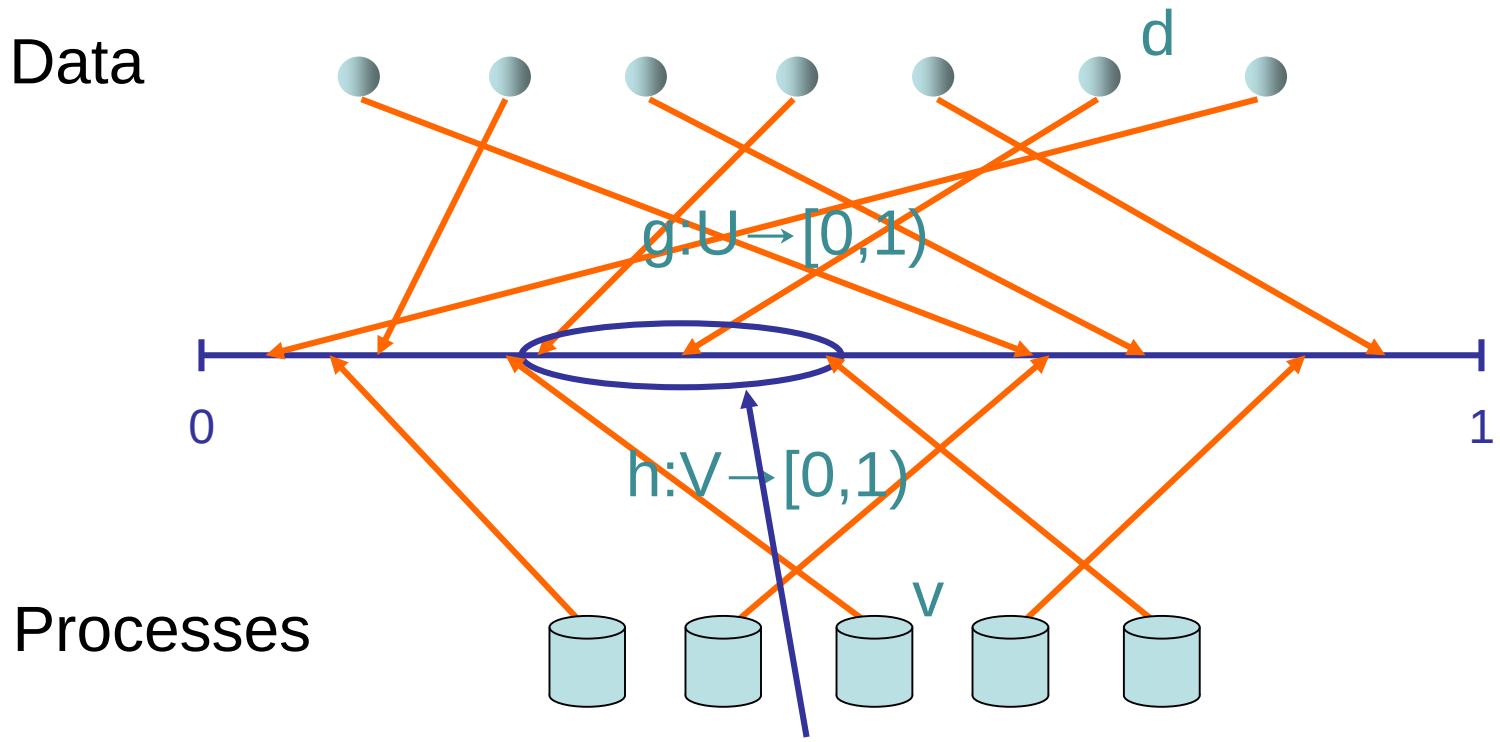
Two problems:

1. How to map the elements to the processes?
2. How to interconnect the processes?



Consistent Hashing

Choose two random hash functions h, g



Region that process v is responsible for

Consistent Hashing

- V: current set of processes
- $succ(v)$: closest successor of v in V w.r.t. hash function h (where $[0,1)$ is viewed as a cycle)
- $pred(v)$: closest predecessor of v in V w.r.t. h

Assignment rules:

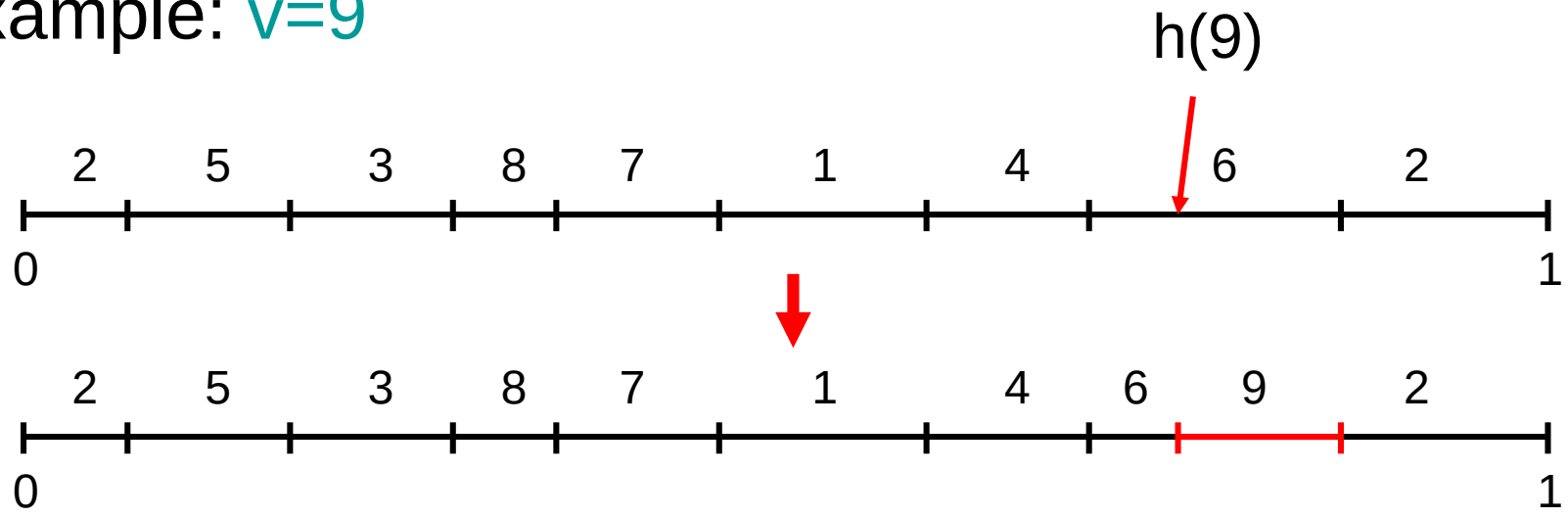
- One copy per element: process v stores all elements e with $g(key(e)) \in I(v)$, where $I(v)=[h(v), h(succ(v)))$.
- $k>1$ copies per element: e is stored in the above node v and its $k-1$ closest successors w.r.t. h

Consistent Hashing

Operations:

- join(v): determine interval I(v) of v and inform the predecessor of v to transfer those elements to v that now belong to v

Example: v=9

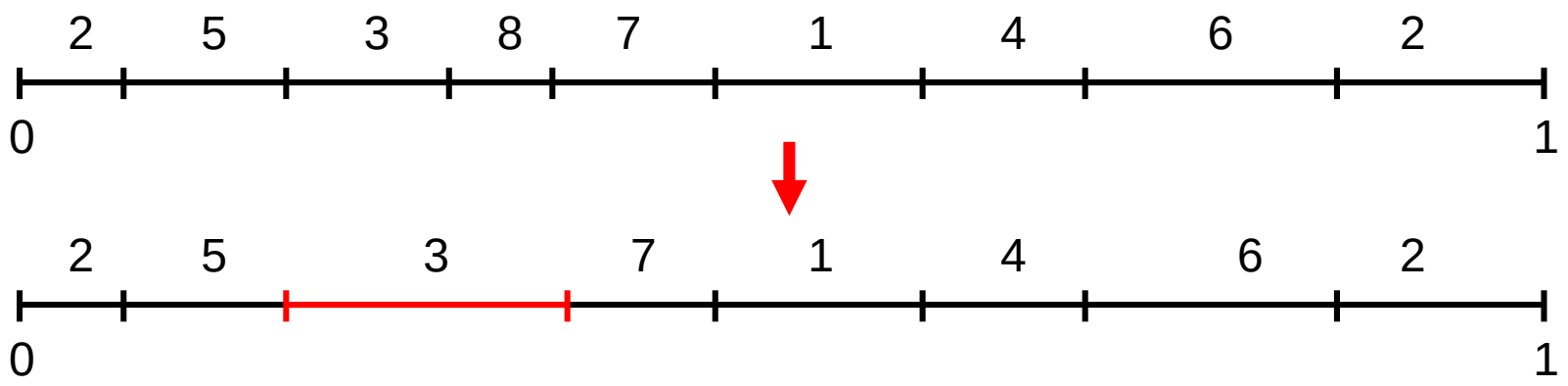


Consistent Hashing

Operations:

- leave(v): move all elements in v to its closest predecessor w

Example: v=8



Consistent Hashing

Consistent hashing works well for processes of uniform capacities.

Theorem 11.1:

- Consistent hashing is efficient and redundant.
- Every process stores on expectation $1/n$ of the elements, i.e., consistent hashing is fair for uniform capacities.
- Whenever a process joins or leaves, on expectation just a $1/n$-fraction of the elements gets replaced

Proof:

Efficiency and redundancy: see the protocol

Fairness:

- For any choice of h it holds that $\sum_{v \in V} |I(v)| = 1$ and therefore $\sum_{v \in V} E[|I(v)|] = 1$ ($E[]$: expectation).
- Given that h is drawn uniformly at random from the set H of all possible hash functions, then for every pair $v,w \in V$ there is a bijection on H, $f:H \to H$, so that for all $h \in H$, $|I(v)|$ w.r.t. $h = |I(w)|$ w.r.t. $f(h)$. Hence, $E[|I(v)|] = E[|I(w)|]$.
- Combining the two equations results in $E[|I(v)|] = 1/n$ for all $v \in V$.

Consistent Hashing

Theorem 11.1:

- Consistent hashing is efficient and redundant.
- Every process stores on expectation $1/n$ of the elements, i.e., consistent hashing is fair.
- Whenever a node joins or leaves, on expectation just a $1/n$-fraction of the elements gets replaced

Problem: deviation from 1/n too high!

Possible solutions:

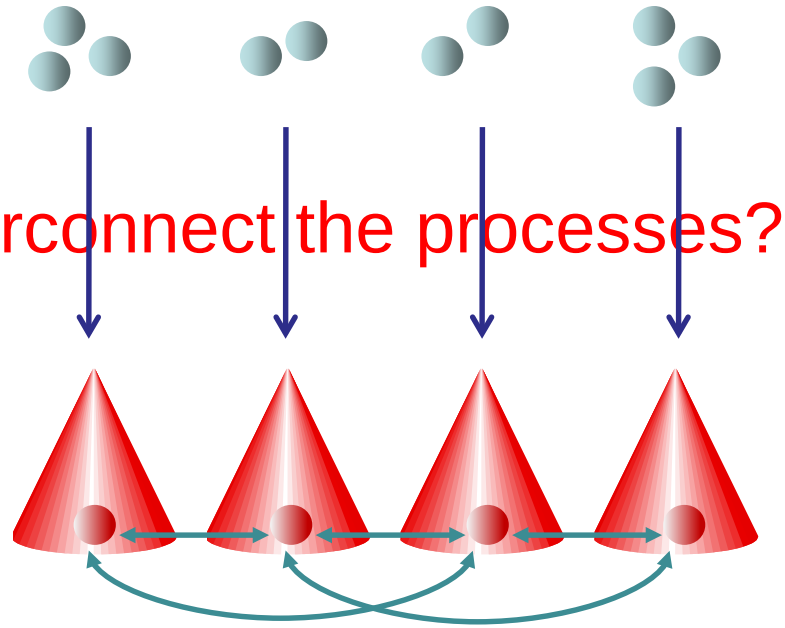
- use $\Theta(\log n)$ hash functions for the processes (i.e., every process has $\Theta(\log n)$ points in $[0,1)$).
- combine consistent hashing with linear probing, i.e., an element is moved forward along the successors until a process with a load of less than $c \cdot m/n$ for some constant $c>1$ is found, where m is the current number of elements

But make sure that the redundancy requirement is still preserved!

Consistent Hashing

Two problems:

1. How to map the data to the processes?
2. How to interconnect the processes?



Here: clique

Overview

- Monitoring
- Information system
- Publish-subscribe system
- Crypto currencies

Publish-Subscribe System

A publish-subscribe system (short, P/S system) has to provide the following basic operations:

- subscribe(v,x): process v joins subscriber group x
- unsubscribe(v,x): process v leaves subscriber group x
- publish(m,x): send message m to all members of subscriber group x

Standard assumption: process v can only publish something in group x if it belongs to group x.

Publish-Subscribe System

Strategy A: for each group x that a process v has subscribed to, v maintains the set of all processes that currently belong to group x (i.e., each subscriber group forms a clique).

Realization of operations:

- subscribe(v,x): Process v contacts a member of group x to join it. For the integration, the Build-Clique protocol is used.
- unsubscribe(v,x): Process v leaves the clique representing group x.
- publish(m,x): The processes use one of the proposed broadcast protocols to spread message m to all members of subscriber group x.

Problem: high work overhead for subscribe and unsubscribe.

Publish-Subscribe System

Strategy B: every process v only remembers the group IDs that it has subscribed to, but nothing beyond that.

Realization of operations:

- subscribe(v,x): Process v adds x to its list of subscriber groups.
- unsubscribe(v,x): Process v removes x from its list of subscriber groups.
- publish(m,x): Send message m to all members of subscriber group x.

Problem: A publish request has to be broadcast to all processes in the system to make sure that it reaches all processes of its group.

Publish-Subscribe System

Strategy C: For each group x that a process v has subscribed to, v only maintains a neighborhood of fixed size d of processes that currently belong to group x. This neighborhood is continuously refreshed using the random-neighbor introduction strategy in the Build-D2G protocol. This should make sure that a subscriber group forms a random graph of degree d.

Realization of operations:

- subscribe(v,x): Process v contacts a member of group x to join it, and this member introduces v to its neighbors.
- unsubscribe(v,x): Process v leaves the graph representing group x (using a strategy similar to the clique).
- publish(m,x): The processes use one of the proposed broadcast protocols to spread message m to all members of subscriber group x.

Publish-Subscribe System

Problem: how to ensure that all publish requests for group x have been received a group member v?

Solution: use extension of median rule.

We assume:

- Each publish request p is identified by a unique key $key(p)$.
- The publish requests that have already been received by process v form the sequence $(p_1,p_2,\ldots,p_k)$. Interpret this as a number $p(v) = 0.key(p_1) \circ key(p_2) \circ \ldots \circ key(p_k) \in [0,1]$. If clear from the context, $p(v)$ may also represent the publication sequence implied by it.

Extension of median rule:

- In each round, each process u contacts two other processes, v and w, uniformly at random and sets $p'(u):=median(p(u),p(v),p(w))$.
- Afterwards, u appends all publications in $\{p(u),p(v),p(w)\}$ that are not part of $p'(u)$ in any order to the end of $p'(u)$ and stores the result in $p(u)$.

Publish-Subscribe System

Extension of median rule:

- In each round, each process u contacts two other processes, v and w, uniformly at random and sets $p´(u):=median(p(u),p(v),p(w))$.
- Afterwards, u appends all publications in $\{p(u),p(v),p(w)\}$ that are not part of $p´(u)$ in any order to the end of $p´(u)$ and stores the result in $p(u)$.

To clarify:

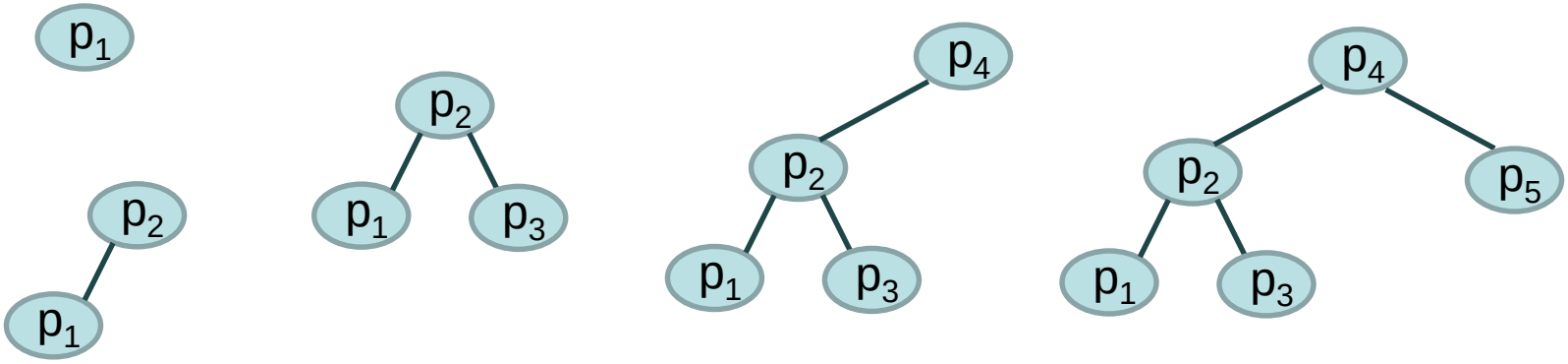
- How to efficiently compute $p´(u)$.
- How to efficiently find all publications not in $p´(u)$.

Publish-Subscribe System

Efficient computation of p´(u):

- Store $p(v)=(p_1,p_2,\ldots,p_k)$ in a balanced binary search tree in which $(p_1,p_2,\ldots,p_k)$ is the result of an in-order traversal.

Examples for sequences of length 1 to 5:

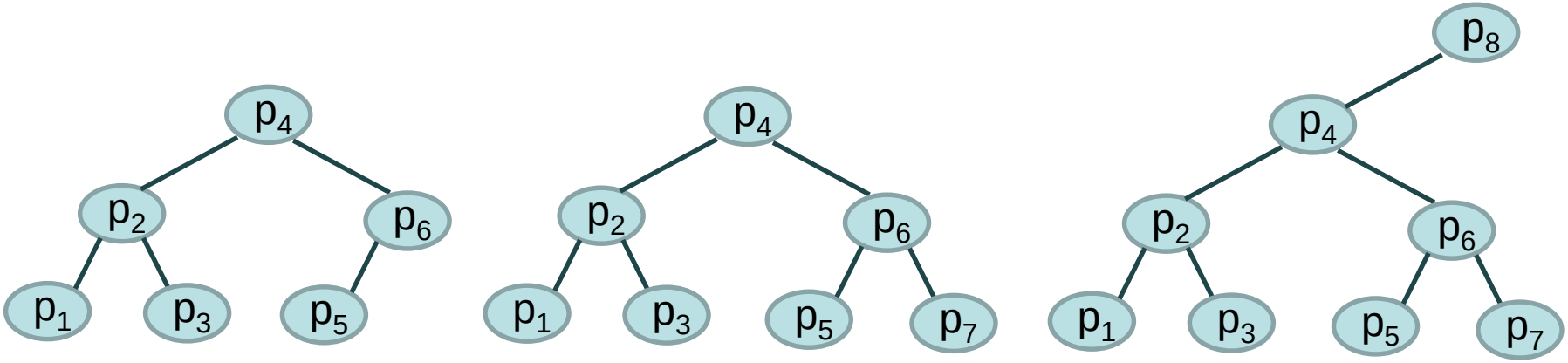


Publish-Subscribe System

Efficient computation of p´(u):

- Store $p(v)=(p_1,p_2,\ldots,p_k)$ in a balanced binary search tree in which $(p_1,p_2,\ldots,p_k)$ is the result of an in-order traversal.

Examples for sequences of length 6 to 8:



Publish-Subscribe System

Efficient computation of p´(u):

- Compute node hashes like in a Merkle hash tree and use the AL↔TCL approach to check hashes.
- Use the node hashes to determine the length of the largest common prefix lcp_1 of p(u) and p(v) as well as lcp_2 of p(u) and p(w). (The one-way hash function guarantees that if the node hashes are identical, then also the publications in the corresponding subtrees are identical and have the same order.)
- Let $x_1(u)$ be the digit at place lcp_1+1 in p(u), x(v) be the digit at place lcp_1+1 in p(v), $x_2(u)$ be the digit at place lcp_2+1 in p(u), and x(w) be the digit at place lcp_2+1 in p(w).
- With lcp_1, lcp_2, $x_1(u)$, $x_2(u)$, x(v), and x(w) we can determine the prefix of p´(u) till lcp_1 resp. lcp_2.

Publish-Subscribe Systems

Efficiently finding publications not in p´(u):

- Transfer all publications p in p(v) beyond the common prefix of p(v) and p(u) and all publications p in p(w) beyond the common prefix of p(u) and p(w) to u. Determine which are not contained in p´(u) and append them to p´(u) if so.

Usually, the missing postfixes should be short (just O(log n)) if at most one publish request is initiated for a subscriber group in each round since, due to the median rule, it takes at most O(log n) rounds to spread it to all other processes and at most O(log n) rounds for it to find its fixed place in the order of publish requests.

Overview

- Monitoring
- Information system
- Publish-subscribe system
- Crypto currencies

Crypto Currency

Problem: maintain a record of all transactions so that the balance of every user can be uniquely determined

Solution:

1. Every transaction must be authorized by the source of the transaction.
2. The authorized transactions are ordered using the extension of the median rule (based on the Merkle hash tree).
3. Transaction committed: its order has remained unchanged for O(log n) many rounds (see the conjecture on the robustness of the median rule if output is based on log of length O(log n) in Chapter 7).
4. Store committed transactions in a distributed fashion so that no process has to store all transactions.

Crypto Currency

1. Autorization of transaction:

The source and sink sign it with their private keys.

TCM model: the private key is stored in the TCL. For the signature, the TCL should offer an interface to the user that is not going through the AL to avoid fake transactions.

Crypto Currency

4. Distributed storage of committed transactions:

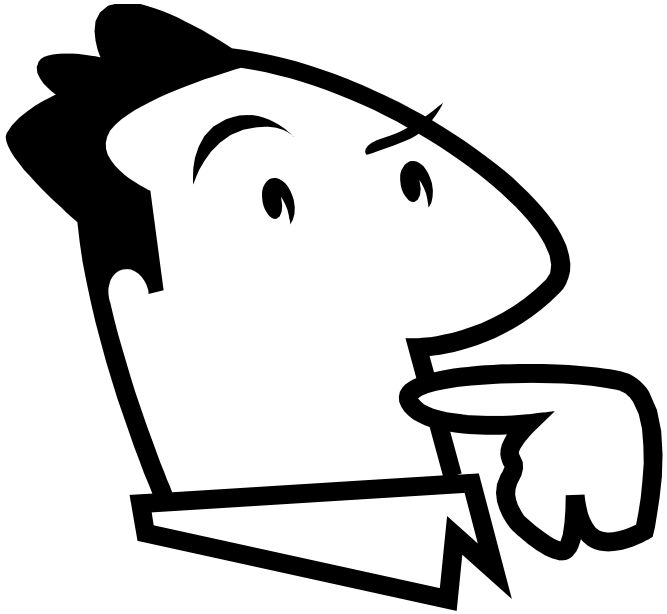
- Use the trusted information system described previously (redundant storage with consistent hashing combined with Merkle hash trees for local checking), where the key of a transaction is its order in the transaction sequence.
- Also store the hashes of the Merkle hash tree of the transactions so that the validity of a single transaction can be checked efficiently.
- Validating the balance of a single user: store the number of the previous transaction and the new balance with each transaction.

Extensions

Transactions in distributed computing:

Use a similar approach as in crypto currencies:

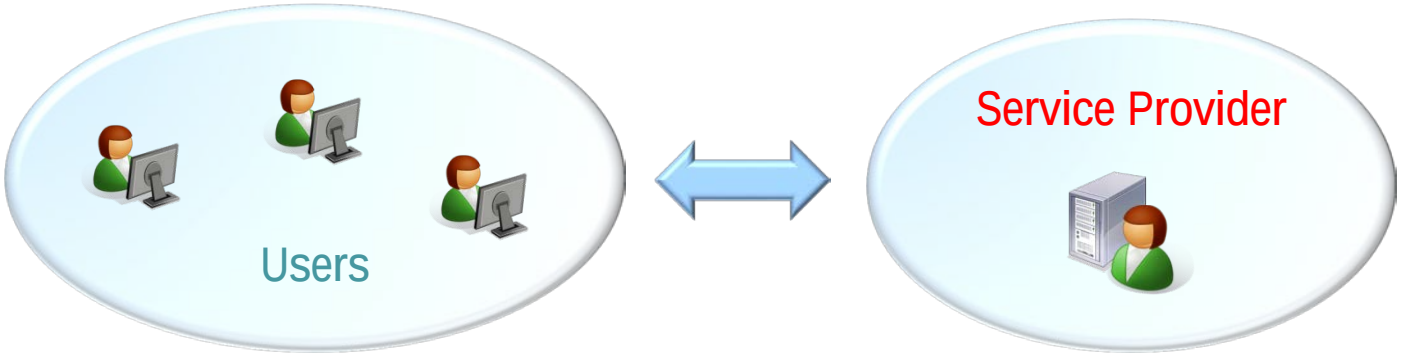
- First, wait until the order of transaction T has been committed (via extended median).
- Then do calculations based on that order at the corresponding processes and report back to the initiator.
- If the initiator correctly receives all reports in a timely manner and also all transactions before T have been committed (or have been undone) in a timely manner, it issues a commit(T). Otherwise, it issues an undo(T). It then has to start another transaction attempt.
- Once this commit has been committed (via extended median), the processes apply the results of their calculations to their state.
- With this approach, no state rollback is needed, but it may happen that processes do calculations in vein.



Questions?

Information System

Outsourcing of information and computation to untrusted third parties is becoming more and more common (→ Cloud Computing)

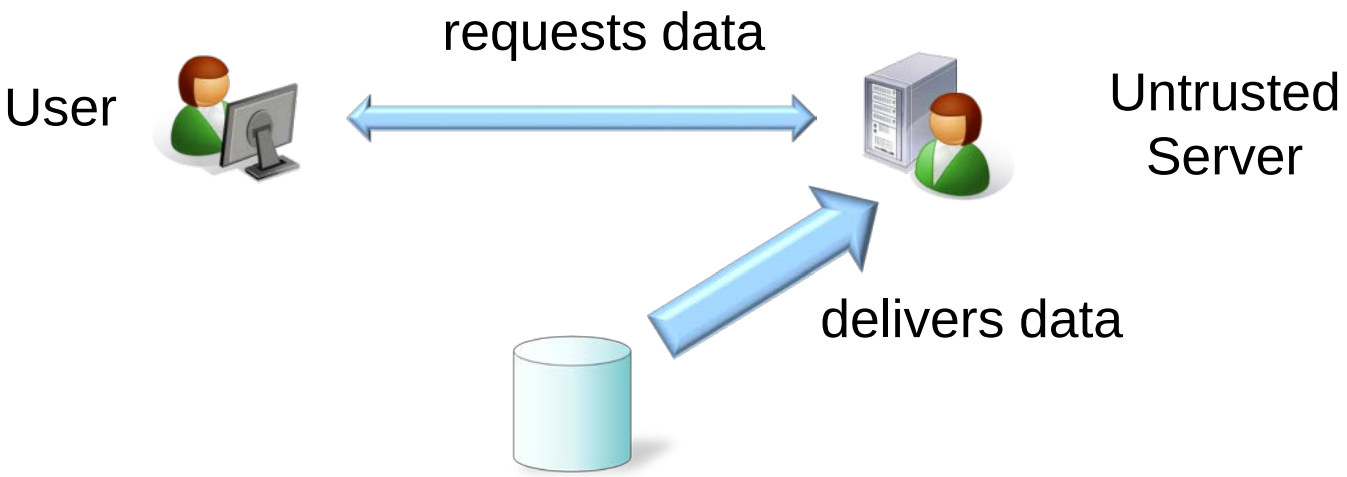


Benefit: much more effective use of resources

Problem: output verification

Information System

Scenario I:



Information System

Scenario II:

General approach:

- Data is authenticated by source (scenario I) or user (scenario II) via a so-called digest (compressed info about data) but managed by the untrusted server
- User requests proof that answer from server is correct

Information System

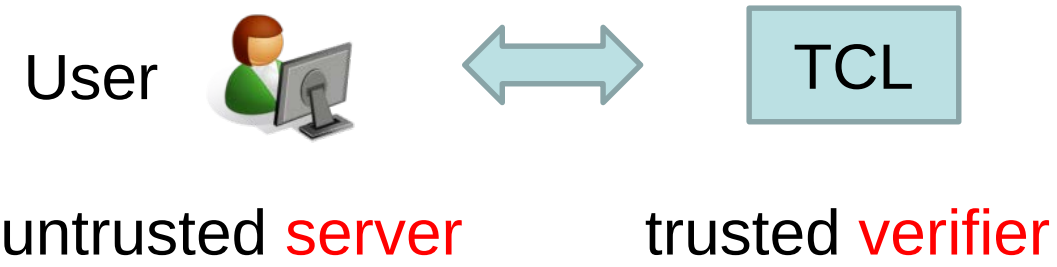
Scenario II:

Basic difference between scenario I and II:

- Scenario I: digest to check integrity of data is made publicly available by source
- Scenario II: digest to check integrity of data is kept private (at user)

Information System

Our scenario (TCM model):



- TCL: stores digest, verifies integrity of data
- User: stores data and provides a proof for any request that answer is correct