Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A DoS-Resilient Information System for Dynamic Data Management

by Baumgart, M. and Scheideler, C. and Schmid, S. In SPAA 2009

Mahdi Asadpour
(amahdi@student.ethz.ch)

Outline

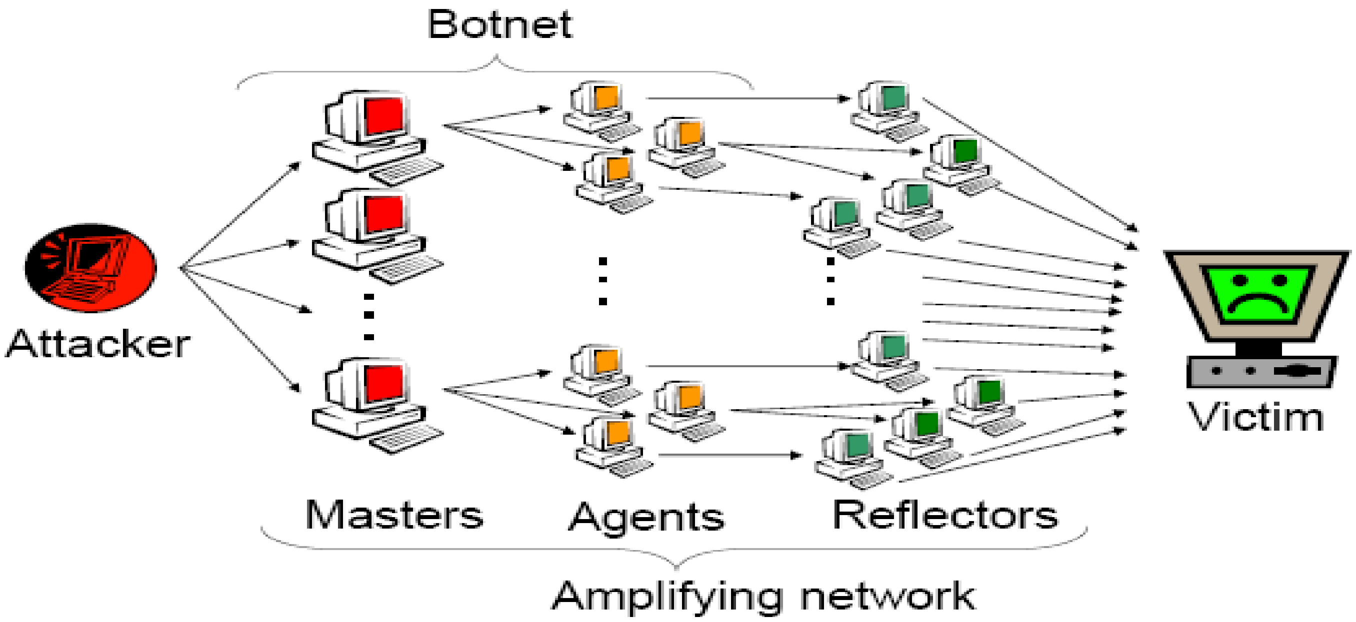
- Denial of Service Attacks
- Chameleon: System Description
- Chameleon: Operational Details
- Conclusion

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Denial of Service Attacks

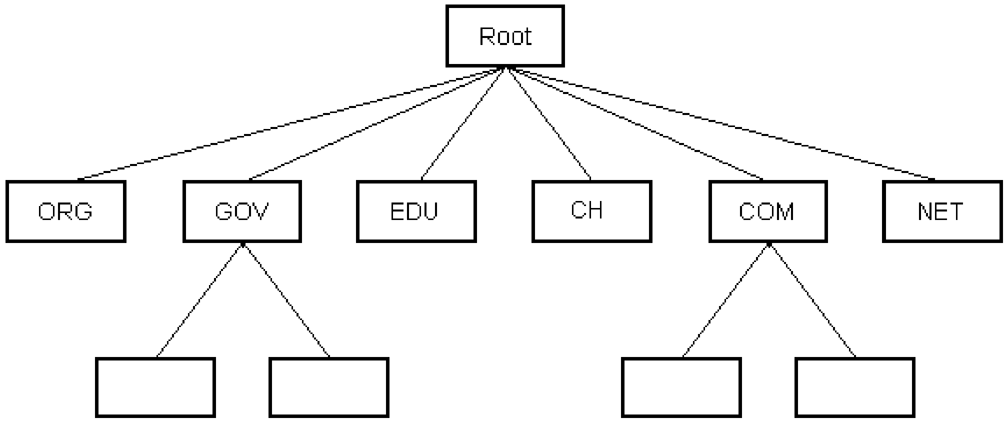
DoS attack

- (Distributed) Denial of Service (DoS) attacks are one of the biggest problems in today's open distributed systems.
- **Botnet**: A set of compromised networked computers controlled through the attacker's program (the "bot").
  - Image credit: Network Security course, Thomas Dübendorfer, ETH Zürich.

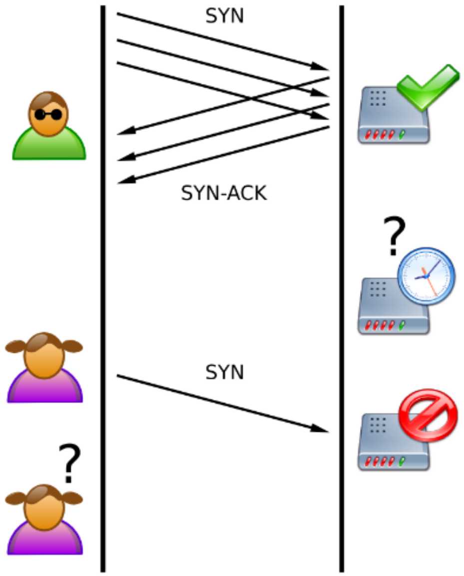


Examples

- DoS attack against the root servers of the DNS system: roots, top-level domains, ...
- TCP SYN flood attack
  - Prevention: SYN cookies
  - Image credit: http://en.wikipedia.org/wiki/SYN_flood



Root Domain Name Servers

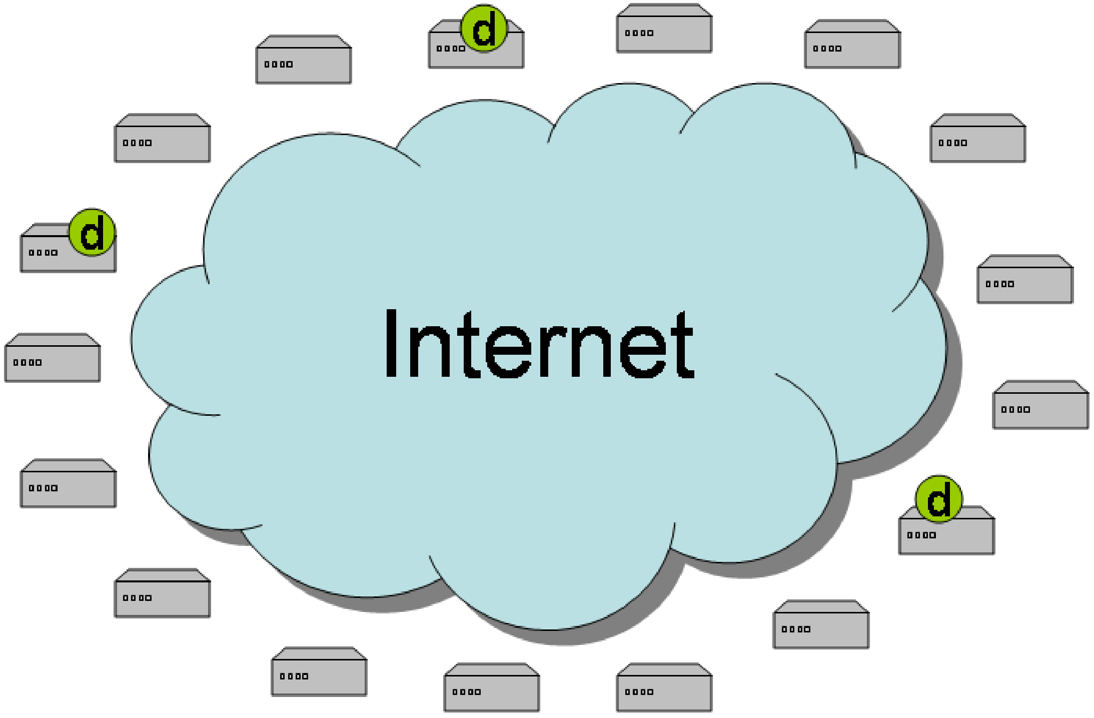


DoS prevention

- **Redundancy**: information is replicated on multiple machines.
  - Storing and maintaining multiple copies have large overhead in storage and update costs.
  - Full replication is not feasible in large information systems.
- In order to preserve **scalability**, the burden on the servers should be minimized.
  - Limited to **logarithmic** factor.
  - Challenge: how to be **robust** against DoS attacks?

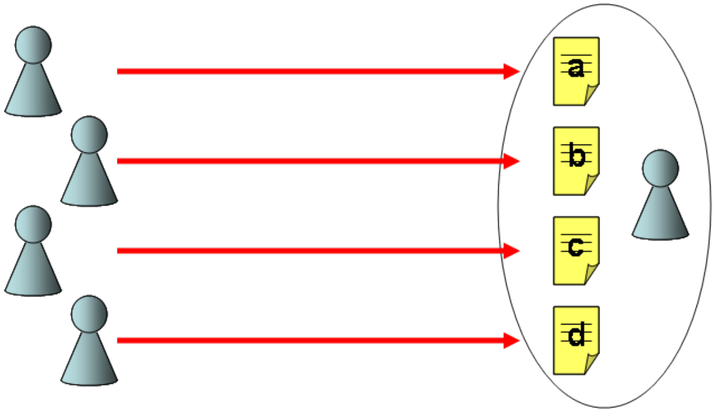
Therefore, a dilemma

- **Scalability**: minimize replication of information
- **Robustness**: maximize resources needed by attacker



Related work

- Many **scalable** information systems:
  - Chord, CAN, Pastry, Tapestry, ...
  - Not robust against flash crowds
- Caching strategies against flash crowds:
  - CoopNet, Backlash, PROOFS,…
  - Not robust against adaptive lookup attacks

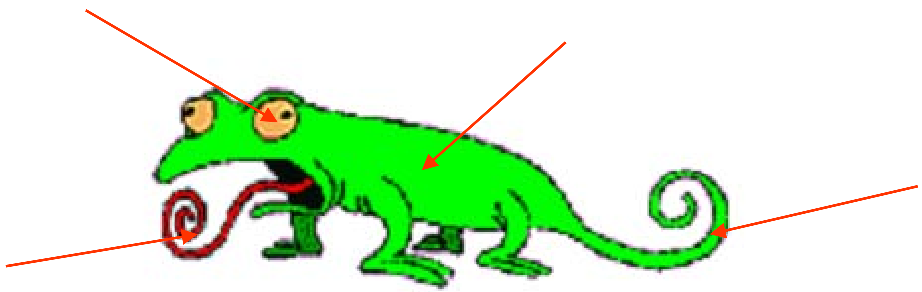


Related work, cont.

- Systems robust against DoS-attacks:
  - SOS, WebSOS, Mayday, III,…
  - Basic strategy: hiding original location of data
  - Not work against past insiders
- Awerbuch and Scheideler (DISC 07):
  - DoS-resistent information system that can only handle get requests under DoS attack

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Chameleon: System Description



Model

- Chameleon: a distributed information system, which is **provably** robust against large-scale DoS attacks.
- N fixed nodes in the system, and all are honest and reliable.
- The system supports these operations:
  - **Put(d):** inserts/updates data item d into the system
  - **Get(name):** this returns the data item d with Name(d)=name, if any.
- Assume that time proceeds in **steps** that are synchronized among the nodes.

Past insider attack

- Attacker knows everything up to some phase t0 that may not be known to the system.
  - A fired employee, for example (Image Credit: Bio Job Blog).
- Can block any ε-fraction of servers
- Can generate any set of put/get requests, one per server.

Goals

- **Scalability**: every server spends at most polylog time and work on put and get requests.
- **Robustness**: every get request to a data item inserted or updated after t0 is served correctly.
- **Correctness**: every get request to a data item is served correctly if the system is not under DoS-attack.
- The paper does not seek to prevent DoS attacks, but rather focuses on how to **maintain a good availability** and performance during the attack.

Also, distributing the load evenly among all nodes

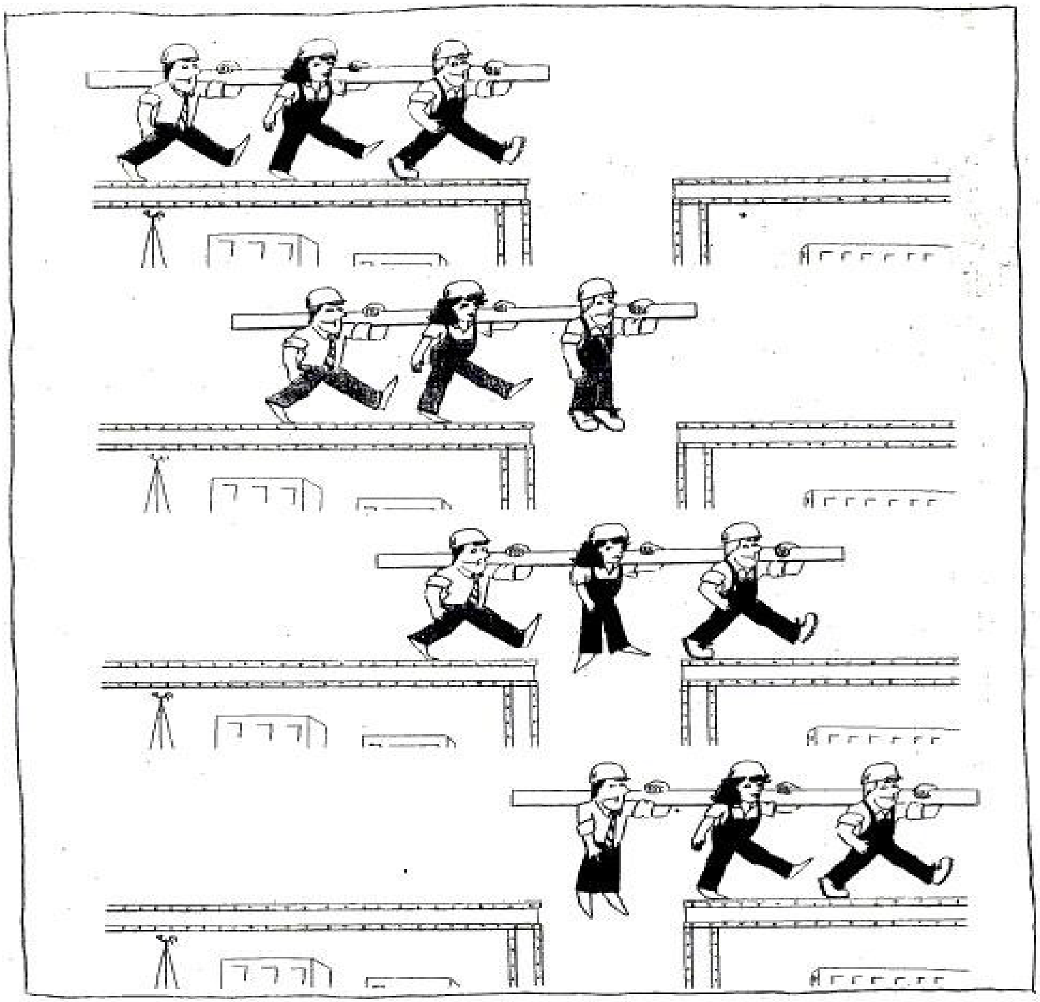
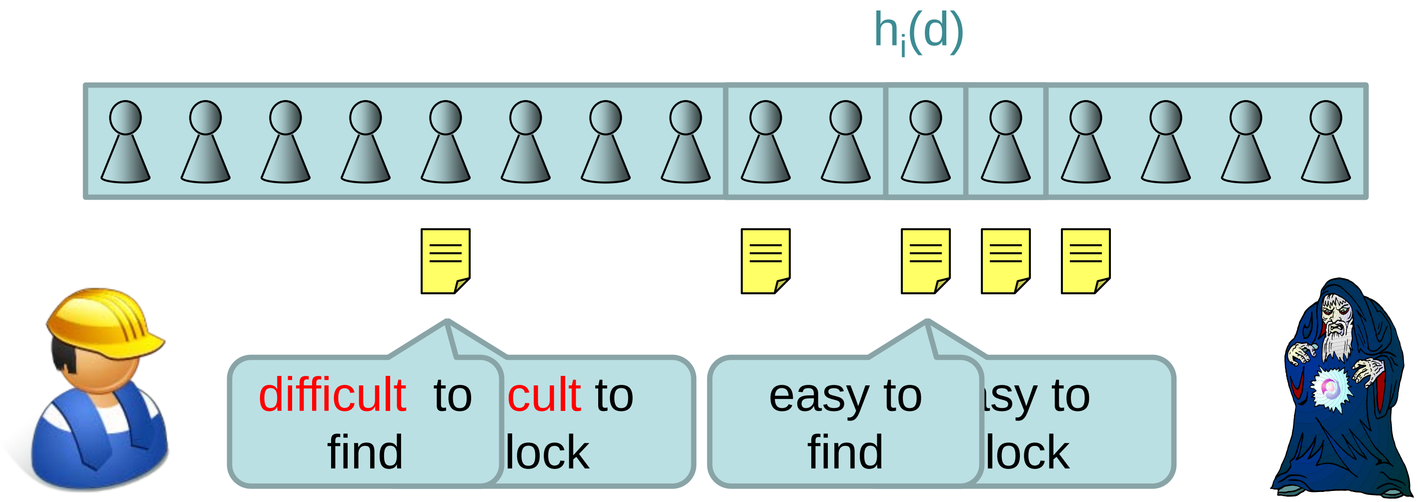


Image credit: Internet!

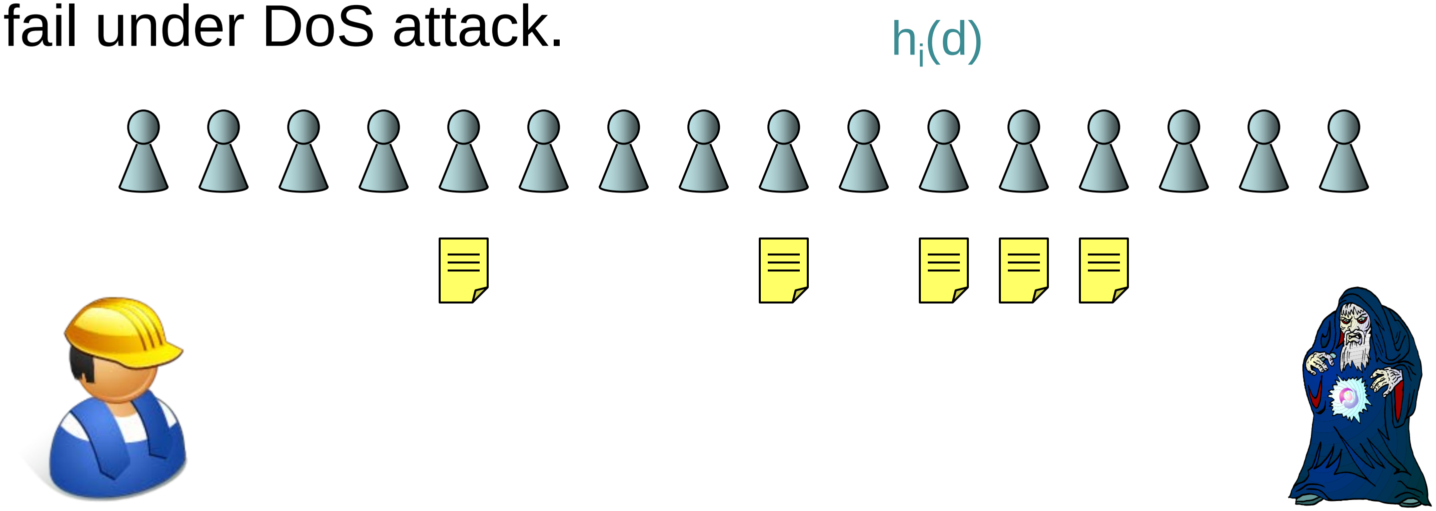
Basic strategy

- Choose suitable hash functions $h_1,..,h_c:D\rightarrow V$
  (D: name space of data, V: set of servers)
- Store copy of item d for every i and j **randomly** in a set of servers of size 2^j that contains $h_i(d)$



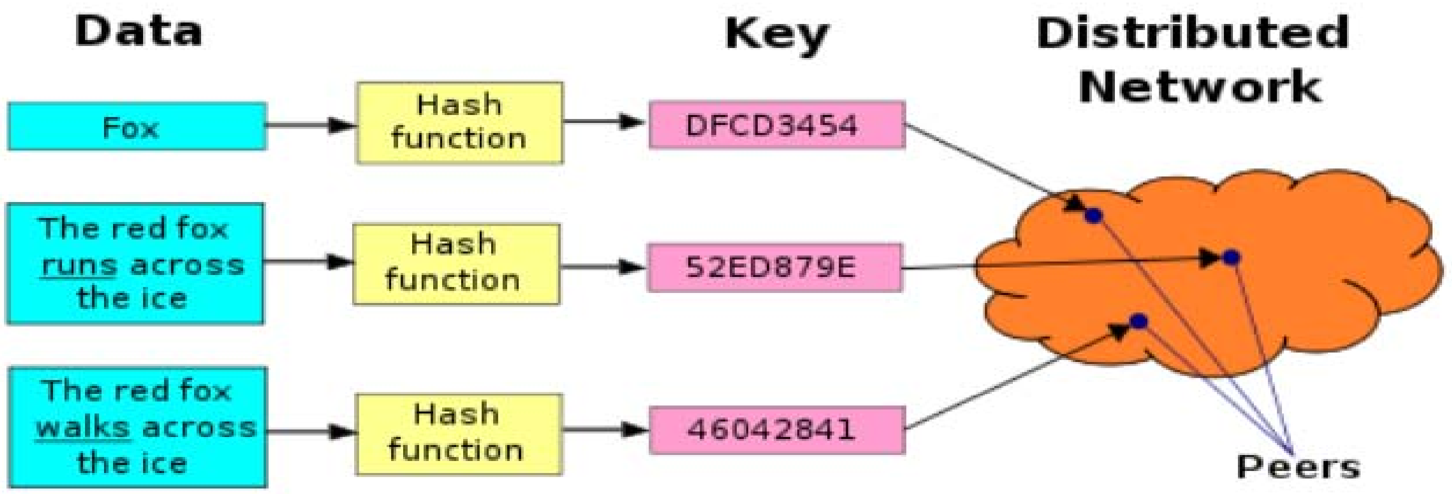
Put and Get Requests

- Most get requests can access close-by copies, only a few get requests have to find distant copies.
- Work for each server altogether just polylog(n) for any set of n get requests, one per server.
- All areas must have up-to-date copies, so put requests may fail under DoS attack.



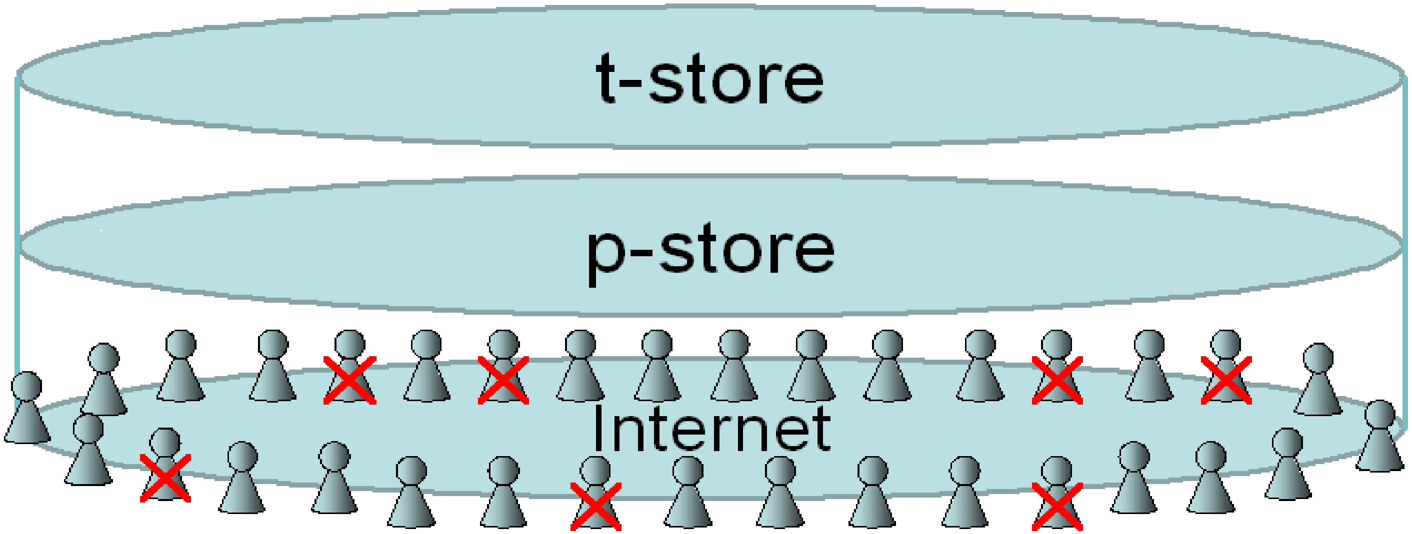
Distributed Hash Table (DHT)

- Chameleon employs the idea of DHT.
- Decentralized distributed systems that provide a lookup service of **(key, value)** pairs: any participating node can efficiently retrieve the value associated with a given key.
  - Image credit: http://en.wikipedia.org/wiki/Distributed_hash_table



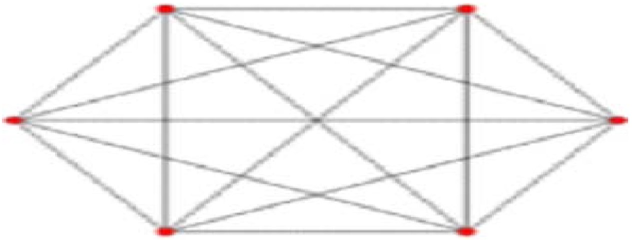
Data stores

- Data management of Chameleon relies on two stores:
  - p-store: a **static** DHT, in which the positions of the data items are fixed unless they are updated.
  - **t-store:** a classic **dynamic** DHT that constantly refreshes its topology and positions of items (**not known** to a past insider).

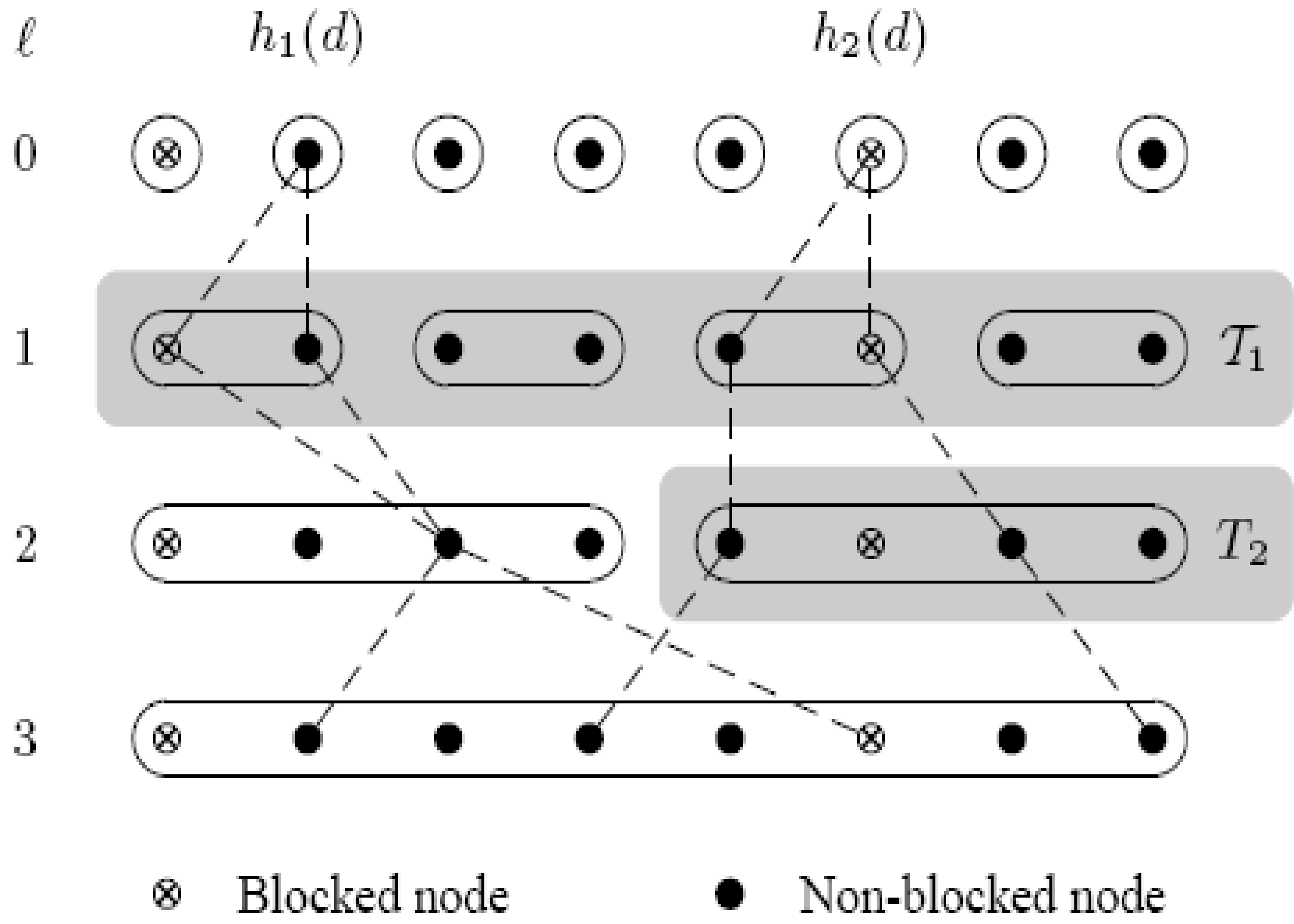


P-store

- Nodes are completely **interconnected** and mapped to **[0,1)**.
- A node i is responsible for the interval **[i/n, (i+1)/n)**. It is represented by **log n** bits, i.e. $\sum x_i/2^i$
- The data items are also mapped to **[0, 1)**, based on fixed hash functions $h_1,..,h_c : U \rightarrow [0, 1)$ (known by everybody).
- For each data item d, the lowest level i = 0 gives fixed storage locations $h_1(d), ..., h_c(d)$ for d of which O(log n) are picked at **random** to store up-to-date copies of d.
- Replicas are along **prefix paths** in the p-store.



P-store, prefix path

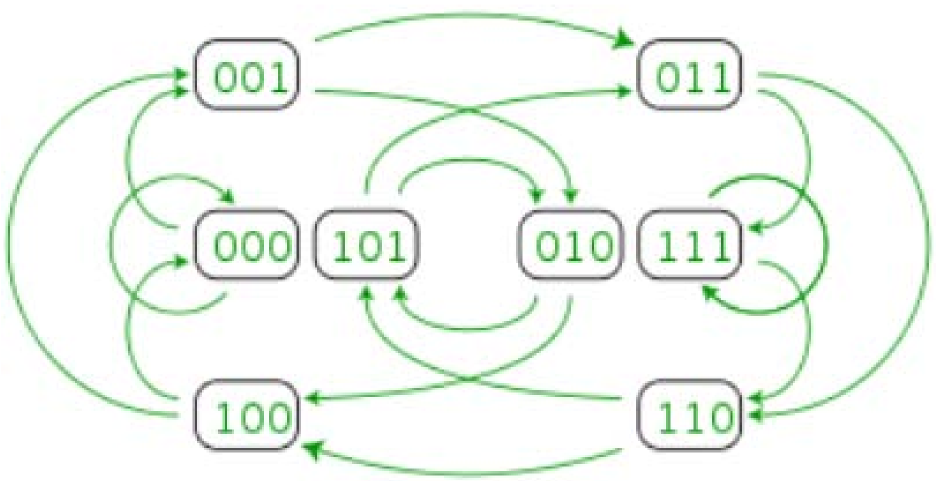


T-store

- In order to correctly store the copies of a data item d, Ω(log n) roots should be reached, which may not always be possible due to a past-insider attack. **T-store** is used to temporarily store data.
- Its topology is a de Bruijn-like network with logarithmic node degree, is constructed from scratch in every phase.
- de Bruijn graphs are useful as they have a logarithmic diameter and a high expansion.

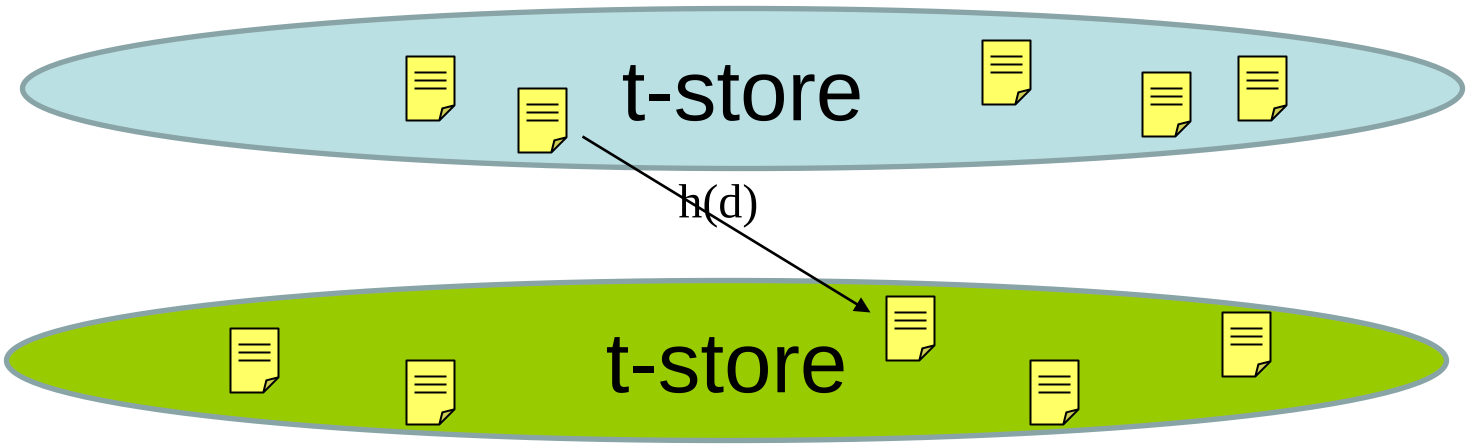
T-store, de Bruijn graph

- **[0, 1)-**space is partitioned into intervals of size ß**log n/n**.
- In every phase, every non-blocked node chooses a random position x in the interval.
- Then tries to establish connections to all other nodes that selected the positions **x, x-, x+, x/2, (x+1)/2**
- Image credit: http://en.wikipedia.org/wiki/De_Bruijn_graph



New T-store

- Once the t-store has been established, the nodes at position 0 select a **random** hash function $h : U \rightarrow [0, 1)$ (by leader election) and broadcast that to all nodes in the t-store.
  - Not known to a past insider after t0.
- h determines the locations of the data items in the new t-store.
  - **d** in the old t-store is stored in the cluster responsible for **h(d).**



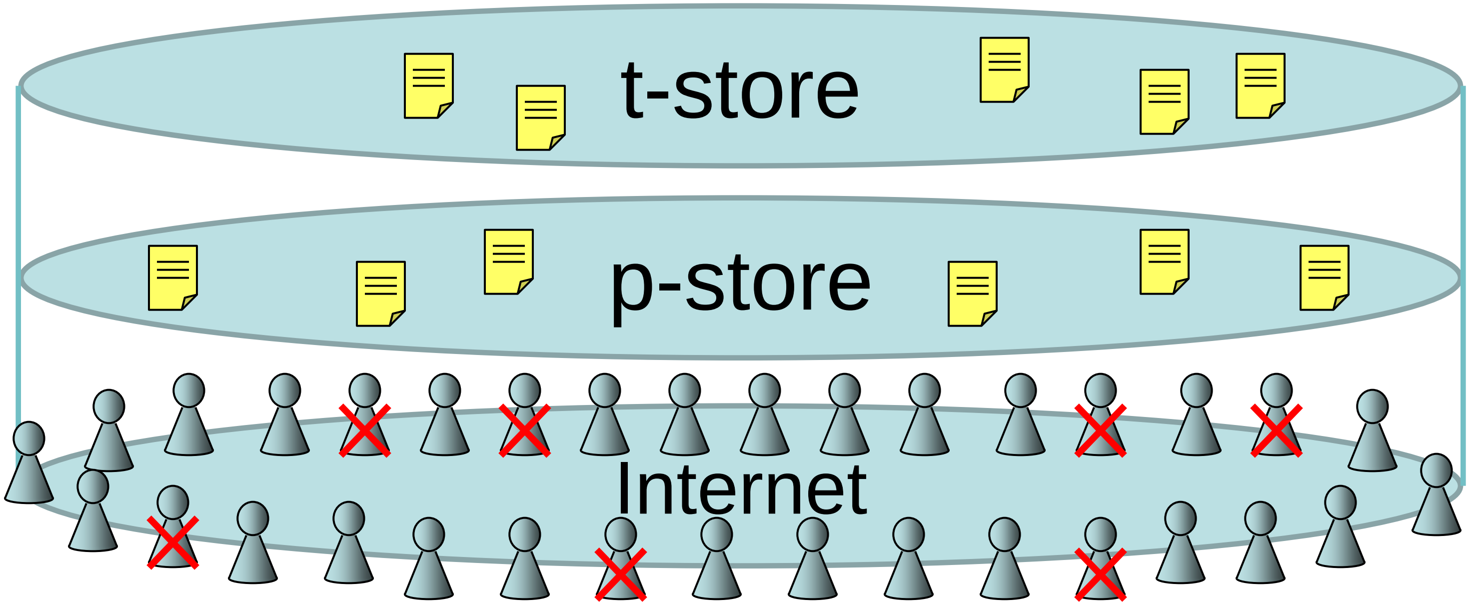
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Chameleon: Operational Details

Image credit: Internet!

Overall procedure in a phase

1. Adversary blocks servers and initiates put & get requests
2. build new t-store, transfer data from old to new t-store
3. process all put requests in t-store
4. process all get requests in t-store and p-store
5. try to transfer data items from t-store to p-store



Stages

- Stage 1: Building a New t-Store
- Stage 2: Put Requests in t-Store
- Stage 3: Processing Get Requests
- Stage 4: Transferring Items

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Stage 1: Building a New t-Store

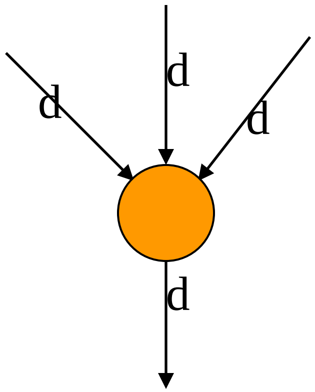
- **Join protocol:** To form a de Bruijn network.
- Every non-blocked node chooses new random location in de Bruijn network.
- Searches for neighbors in p-store using **join(x)** operation.
- Nodes in graph agree on a set of ***log n*** random hash functions $g_1, \ldots, g_{c'} : [0, 1) \rightarrow [0, 1)$ via randomized leader election.
- **Randomized leader election**: each node guesses a random bit string and the one with lowest bit string wins and proposes the hash functions, in O(log n) round/time.

Stage 1: Building a New t-Store, cont.

- **Insert protocol**: to transfer data items from the old t-store to the new t-store.
- For every cluster in the old t-store with currently non-blocked nodes, one of its nodes issues an **insert(d)** request for each of the data items d stored in it.
- Each of these requests is sent to the nodes owning $g_1(x), \ldots, g_{c'}(x)$ in the p-store, where $x = \lfloor h(d) \rfloor (\delta \log n)/n$.
- Each non-blocked node collects all data items d to point x and forwards them to those contacted it in the join protocol.
- O(n) items w.h.p.

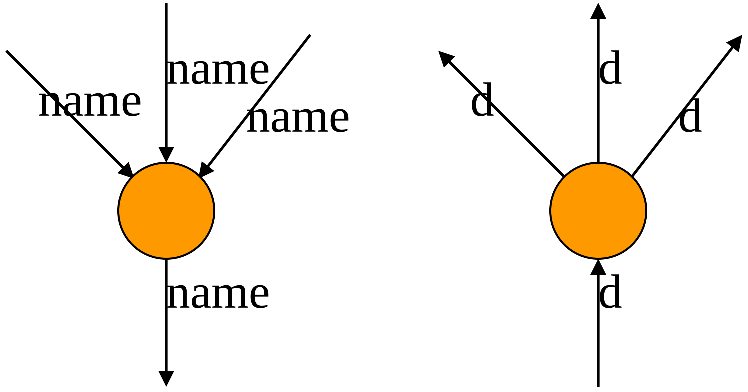
Stage 2: Put Requests in t-Store

- New put requests are served in the t-store: for a **put(d)** requests, a **t-put(d)** request is executed.
- Each t-put(d) request aims at storing d in the cluster responsible for h(d) passing.
- The t-put requests are sent to their destination clusters using **de Bruijn paths**, e.g. X → Y
  - $(x_1, \ldots, x_{\log n}) \rightarrow (y_{\log n}, x_1, \ldots, x_{\log n - 1}) \rightarrow \ldots \rightarrow (y_1, \ldots, y_{\log n})$
- **Filtering mechanism:**
  - Only one of the same t-put requests survives.
- **Routing rule**:
  - Just ρ $\log^2 n$ to pass a node
- O(logn) time, O($\log^2$ n) congestion.



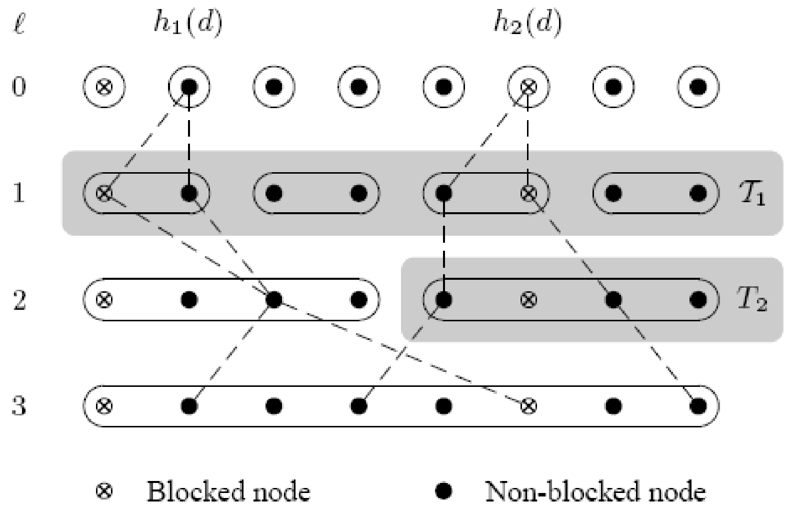
Stage 3: Processing Get Requests

- First: in the **t-store** using the **t-get** protocol
  - de Bruijn routing with combining to lookup data in t-store
  - O(log n) time and O($\log^2$ n) congestion
- Second: If cannot be served in the t-store, then store in the **p-store** using the **p-get** protocol.
  - Three stages: preprocessing, contraction and expansion
- **Filtering**: almost similar to t-put.



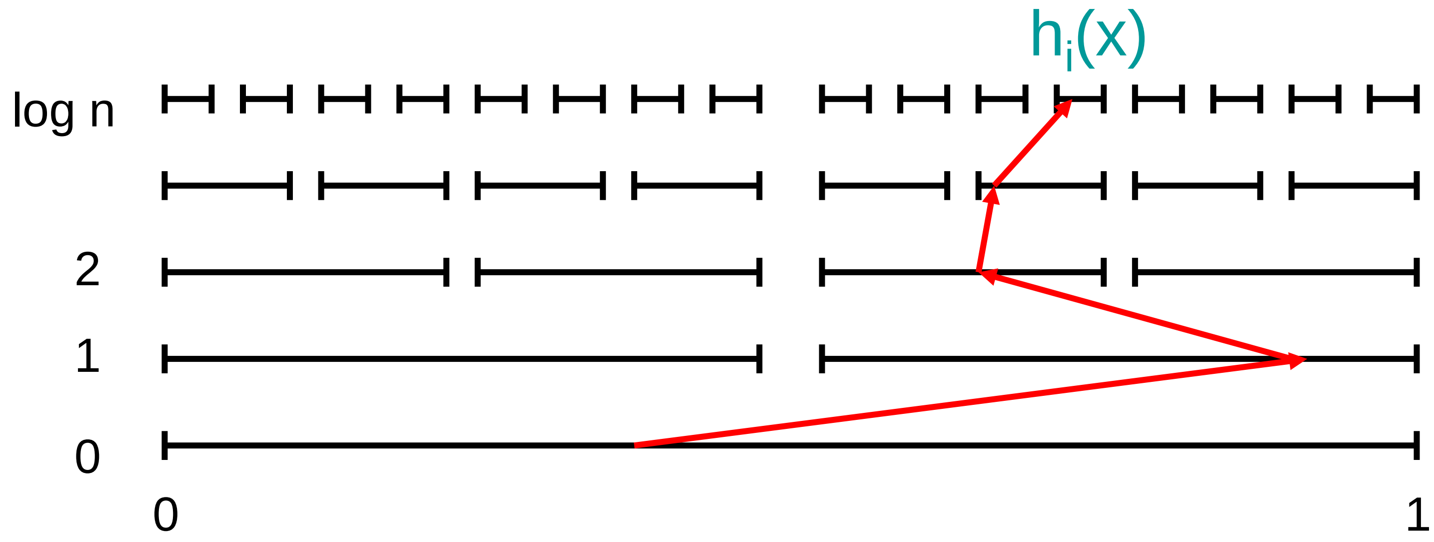
Stage 3: Processing p-Get Requests, Preprocessing

- P-get **Preprocessing**: Determines blocked areas via sampling.
  - Every non-blocked node v checks the state of **α log n** random nodes in $T_i(v)$ for every $0 \leq i \leq \log n$.
  - If >= **1/4 of the nodes** are blocked, v declares $T_i(v)$ as blocked.
- O(1) time: Since the checking can be done in parallel.
- $O(\log^2 n)$ congestion



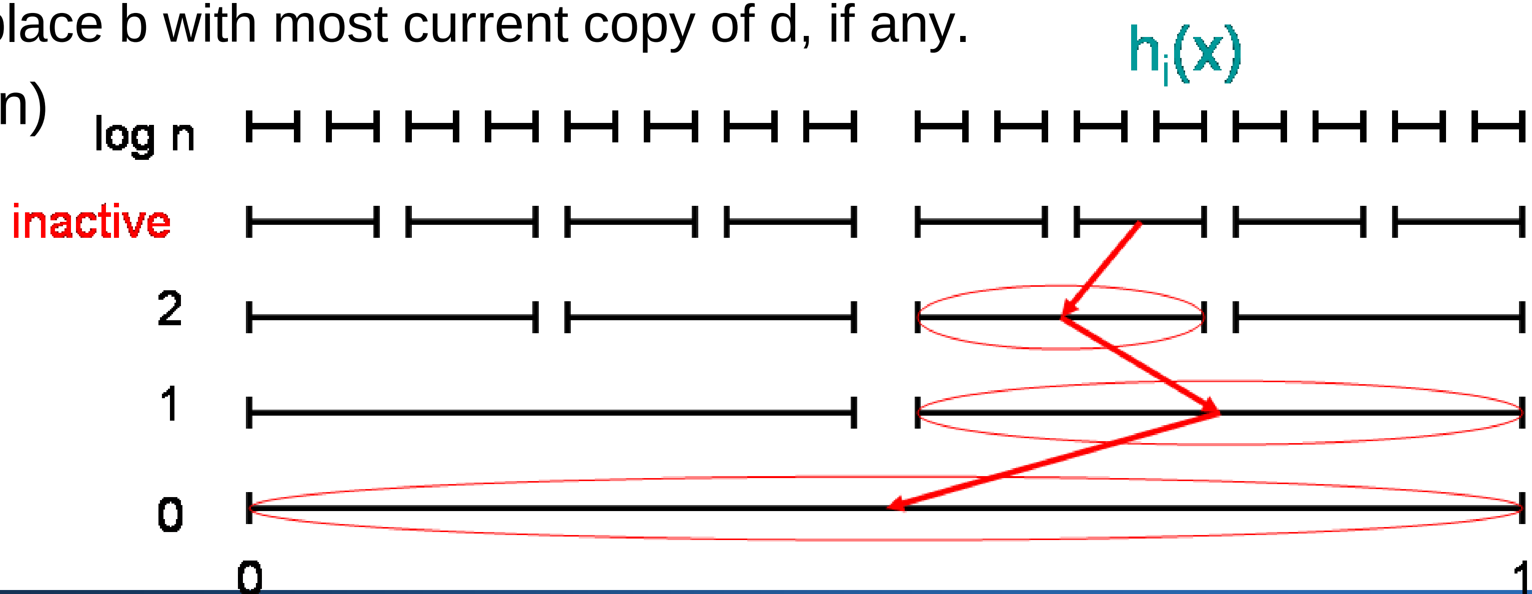
Stage 3: Processing p-Get Requests, Contraction

- Each **p-get(d)** request issued by some node v selects a random node out of all nodes and aims at reaching the node responsible for $\mathbf{h_i(d)}$, i in {1, …, c} in at most ξ **log n** hops.
- Stop: $T_l(h_i(id))$ is blocked or hops > ξ **log n** => **deactivate** i
- O(log n) time, w.h.p



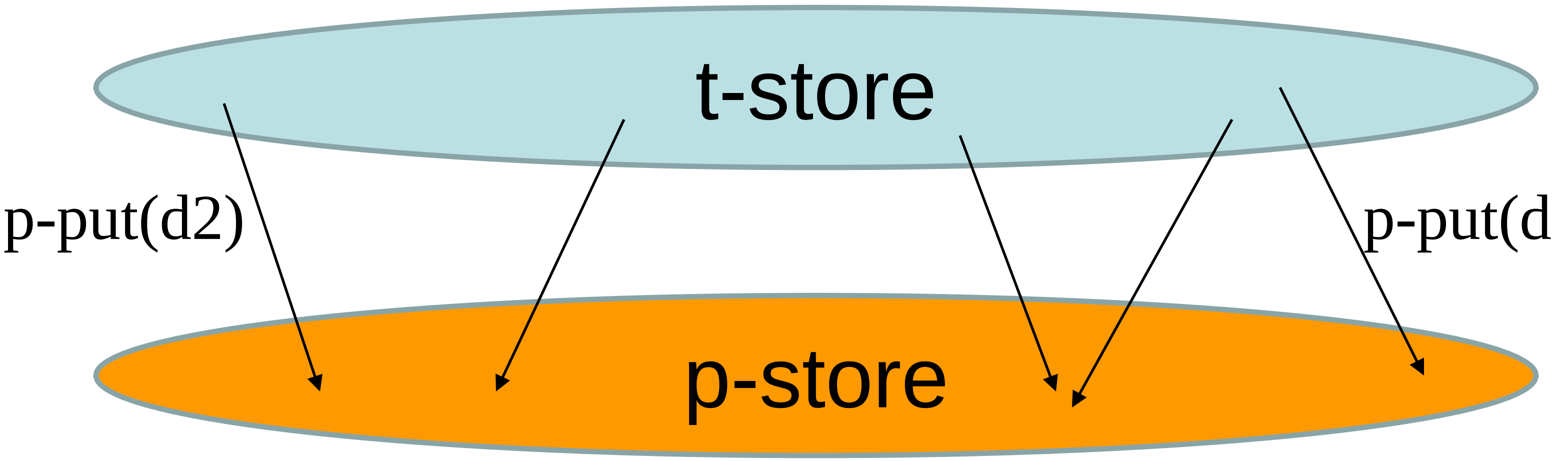
Stage 3: Processing p-Get Requests, Expansion

- Looks for copies at successively **wider** areas.
- Every not-finished **p-get(d)** request sends **(d, r, i,−)** to a non-blocked node v that was successfully contacted before.
- V maintains a copy b of d in **(d, r, i, b)** and executes O(log n) :
  - Sends **(d, r, i, b)** to a random node in the same level.
  - Replace b with most current copy of d, if any.
- $O(\log^2 n)$



Stage 4: Transferring Items

- Transfers all items stored in the **t-store** to the **p-store** using the **p-put** protocol.
- After, the corresponding data item in the t-store is removed.
- **p-put** protocol has three stages: Preprocessing, Contraction, Permanent storage

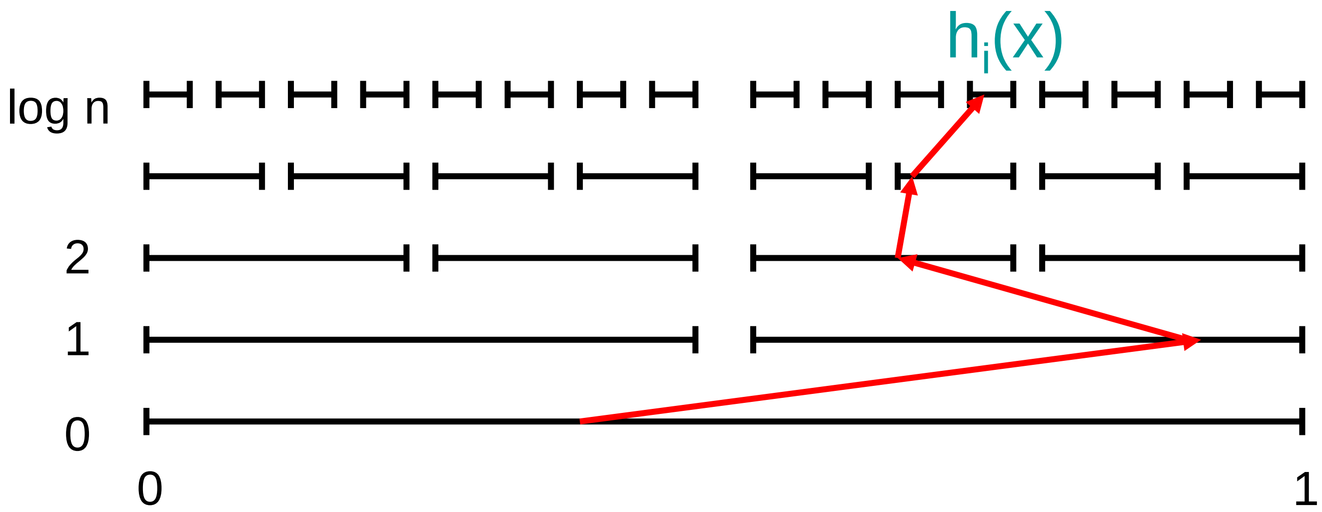


Stage 4: Transferring Items, p-Put preprocessing

- p-Put **preprocessing** is like in the **p-get** protocol
- Determines blocked areas and average load in p-store via sampling.
- O(1) time
- $O(\log^2 n)$ congestion

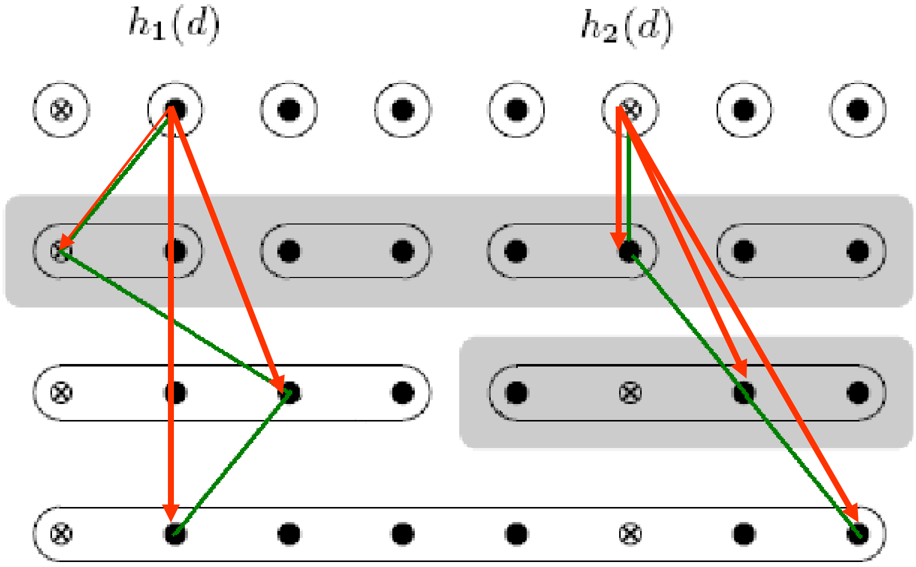
Stage 4: Transferring Items, p-put Contraction

- p-put **Contraction** is identical to the contraction stage of the **p-get** protocol.
- Tries to get to sufficiently many hash-based positions in p-store.
- O(log n) time.



Stage 4: Transferring Items, p-put Permanent storage

1. p-put **Permanent storage**: For each successful data item, store new copies and delete as many old ones as possible.
2. In the node responsible for **$h_i(d)$** (d's root node) information about the nodes storing a copy of d is stored.
3. This information is used to remove all out-of-date copies of d.



Stage 4: Transferring Items, p-put Permanent storage

4. If it is not possible (**blocking**), references to these out-of-date copies are left in the roots (be deleted later on).
5. Select a random non-blocked node in each **$T\ell(h_i(d))$** with ℓ in $\{0, . . . , \log n\}$.
6. Store an up-to-date copy of d in these nodes, and store references to these nodes in $h_i(d)$.
7. O(**log n**) time. The number of copies of d remains O(**$\log^2 n$**).

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Conclusion

Main theorem

- **Theorem:** Under any ε-bounded past-insider attack (for some constant $\varepsilon>0$), the Chameleon system can serve any set of requests (one per server) in $O(\log^2 n)$ time s.t. every get request to a data item inserted or updated after t_0 is served correctly, w.h.p.
- No degradation over time:
  - $O(\log^2 n)$ copies per data item
  - Fair distribution of data among servers

Summary

- This paper shows how to build a scalable dynamic information system that is robust against a past insider.
- Two distributed hash tables for data managements: temporary and permanent, respectively t-store and p-store.
- The authors defined many constants ξ, ß, ρ, … but did not optimize them, e.g. the replication factors.
- As also authors proposed, it would be interesting to study whether the runtime of a phase can be reduced to O(log n).
- No experimental evaluation.

References

- Some of the slides are taken from the authors, with permission.
- Main references:

1. B. Awerbuch and C. Scheideler. **A Denial-of-Service Resistant DHT**. DISC 2007.
2. B. Awerbuch and C. Scheideler. **Towards a Scalable and Robust DHT**. SPAA 2006.
3. D. Karger, et al. **Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web**. STOC 1997.

Thanks for your attention.

Any question?