A DoS-Resilient Information System for Dynamic Data Management[*]

Matthias Baumgart
Dept. of Computer Science
Technische Universität München
D-85748 Garching bei München
Germany
baumgart@in.tum.de

Christian Scheideler
Dept. of Computer Science
University of Paderborn
D-33102 Paderborn
Germany
scheideler@upb.de

Stefan Schmid
Dept. of Computer Science
Technische Universität München
D-85748 Garching bei München
Germany
schmiste@in.tum.de

ABSTRACT

Denial of service (DoS) attacks are arguably one of the most cumbersome problems in the Internet. This paper presents a distributed information system (over a set of completely connected servers) called *Chameleon* which is robust to DoS attacks on the nodes as well as the operations of the system. In particular, it allows nodes to efficiently look up and insert data items at any time, despite a powerful "past-insider adversary" which has complete knowledge of the system up to some time point t_0 and can use that knowledge in order to block a constant fraction of the nodes and inject lookup and insert requests to selected data. This is achieved with a smart randomized replication policy requiring a polylogarithmic overhead only and the interplay of a permanent and a temporary distributed hash table. All requests in Chameleon can be processed in polylogarithmic time and work at every node.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Algorithms, Theory

1. INTRODUCTION

It is widely believed that distributed denial of service (DoS) attacks are one of the biggest problems in today's open distributed systems, such as the Internet. Attackers use the fact that Internet servers are typically accessible to anyone in order to overload them with bogus requests from so-called *bot nets*, which are large groups of machines that are under their control [31, 32]. Examples of such attacks include downloading large files [24], causing computationally expensive operations [9], or just overloading servers with junk. Some popular information services like Google and Akamai are under constant DoS attacks, and also the Domain Name System has been hit several times by major DoS attacks during the last years [13].

[*]Research partly supported by the DFG-Project SCHE 1592/1-1.

The predominant approach to deal with the threat of DoS-attacks is the introduction of *redundancy*. Information which is replicated on multiple machines is more likely to remain accessible during a DoS attack. However, storing and maintaining multiple copies of each data item can entail a large overhead in storage and update costs. In order to preserve scalability, it is therefore vital that the burden on the servers is minimized.

This paper presents a distributed information system called *Chameleon* which is provably robust against large-scale DoS attacks. This is even true if the attacker is a past insider with full knowledge of the system's internals up to a certain time point t_0 (which may be unknown to the system). As has been pointed out in [2], robustness to such attacks is a crucial feature, as many security breaches in corporate systems are caused by human error and negligence (which may temporarily expose the system to the outside world) as well as past insiders (such as temporary or fired employees). The Chameleon system can process put and get requests efficiently at any time despite a massive ongoing DoS attack and even though the put and get requests were selected by the adversary. The trick of our system is that it employs a smart replication strategy whose appearance cannot be predicted by the attacker (hence the system's name) though the data can still be efficiently located. In fact, in Chameleon, it is sufficient to employ a logarithmic redundancy, even if we allow the adversary to block a constant fraction of all servers.

1.1 Model

In a distributed information system (e.g., a collaborative system consisting of many servers located at different company sites), data is distributed among multiple servers, simply called *nodes* in the following. We assume that we are given a name space U and each data item d is identified by its name in that space. All data items are of unit size (e.g., we are dealing with a block-level storage system). To provide a basic lookup service, the following operations have to be implemented:

- Put(d): this inserts data item d into the system (if nothing has been stored under its name before) or updates it (if its name has already been used).
- Get(*name*): this returns the data item d with Name(d)=*name*, or $\perp$ if no such data item exists.

We assume that the set of nodes in the system is fixed and that all nodes are honest and reliable (since we are dealing with a server-based system). However, there is an adversary that has the power to shut down (or block) up to ϵn nodes at any time (by a DoS attack), for some constant $\epsilon > 0$ that we would like to be as large as possible without harming the functionality of the system. In order to keep the description of our problem at a reasonable level, we assume that time proceeds in *steps* that are synchronized among the nodes.

Note however that using local synchronizers [23], our algorithms also work in asynchronous settings. All we need is a bounded transmission time between two non-attacked nodes. In each time step, every node is able to send and receive a polylogarithmic amount of information and as long as this bound is satisfied, any message sent out by some node v to some node w will arrive at w within the next time step (or be dropped if w is blocked). In this way, a node can easily determine whether another node is blocked by not receiving an acknowledgment of its message within two time steps.

We allow the adversary to block any set of nodes and issue any put and get requests, but the rate at which it can do this is limited. For simplicity, we will assume a batch-like mode in which the time is partitioned into so-called *phases* (that should be as short as possible; in our case $O(\log^2 n)$ time steps suffice). At the beginning of each phase, the adversary selects an arbitrary, fixed set of ϵn nodes that will be blocked throughout that phase. It also selects an arbitrary set of put and get requests (including multiple requests to the same data item or get requests to non-existing data items) with at most one request per non-blocked node. The goal of the system is to serve all of these requests within the given phase without overloading any node with data over time. A get request for some data item d is served *correctly* if a *most recent* version of d is returned and this most recent version is *unique*. That is, a version of d is delivered that belongs to a put request in a most recent phase (including the current phase), and between two phases with updates of d, all get requests for d return the same version of d. This implies that if multiple put requests are issued for the same name in the same phase, then only one of them will win, i.e., will determine the unique version that will be stored under that name.

A data item d is said to have a *redundancy* of r if r times more storage (including any control storage for d) is used for d than is needed when just storing the plain item. A node is called *overloaded* if its storage load is by more than a constant factor larger than the average load in the system.

Of course, if the adversary knows everything about the system and the data items have a small redundancy, then it is impossible for the system to serve all requests in a correct way. Hence, we assume that the adversary is a *past insider*, i.e., it only knows everything about the system up to some phase t_0 that may not be known to the system. After t_0, the adversary cannot inspect nodes or communication between the nodes anymore—it can only block nodes and issue requests. The goal of the system will be to ensure the following properties in any phase (before and after t_0, *without* knowing t_0):

1) *Scalability*: Every node spends at most polylogarithmic time (number of communication rounds) and work (number of messages) in order to serve all requests in a phase and no node will get overloaded over time.

2) *Robustness*: All get requests for data that was inserted or last updated *after* t_0 are served correctly under any adversarial attack within our model.

Achieving these conditions is not an easy task as the system cannot afford to continuously replace all the data in it (recall that the system does *not* know t_0 and we have no bound on the number of data items in the system). Also, no long-term information hiding techniques can be used (as the adversary has *full* knowledge of the system up to phase t_0). Yet, there is a solution. The Chameleon system we propose in this paper is the first system that can achieve all of these goals. In fact, it just needs a logarithmic redundancy (when using Reed-Solomon coding, for example) and phases of polylogarithmic length.

For simplicity, we will assume that the total number of nodes, n, is a power of two, and that the nodes are numbered from 0 to $n-1$. The size of the name universe U is defined as m, where m is polynomially bounded in n.

1.2 Related Work

Due to their importance, DoS attacks are a well-studied problem (e.g., [5, 18] for an overview). Unfortunately, it is often difficult to distinguish DoS traffic from legitimate traffic, which renders many network-layer and transport-layer DoS prevention tools such as installing a box to filter out anomalies [15], blacklisting particular IP addresses, using TCP SYN cookies [3], pushback [8], etc., problematic [31]. This observation has led some researchers to propose means how legitimate clients can "speak up" and thus be identified [31, 32], for example.

In this paper, we do not seek to prevent DoS attacks, but rather focus on how to maintain a good availability and performance during the attack. Our system is based on the distributed hash table (DHT) paradigm (e.g., [4, 6, 7, 25, 30]). In particular, we follow a *consistent hashing* approach [10] in order to store the data and employ the *continuous-discrete techniques* presented in [20] for communication between the servers.

DoS-resistant systems based on DHTs have already been studied in [11, 12, 19]. For instance, the Secure Overlay Services approach [12] uses *proxies* on Chord to defend against flooding DoS attacks. A Chord overlay is also used by the Internet Indirection Infrastructure *i3* [29] to achieve resilience to DoS attacks. Other DoS limiting architectures have been proposed in [21, 33]. Many of these systems are based on traffic analysis or some indirection approach.

Replication strategies have already been investigated in the context of *flash crowd* problems in DHTs. Important literature in the systems community includes CoopNet [22], Backslash [27] or PROOFS [28], and there is also theoretical work [20]. However, these works only consider scenarios where many requests are targeted to the same data item, but not to many different items *at the same location*. Techniques originally proposed for CRCW PRAMs [17] allow one to overcome these limitations [1], although only for application layer attacks (i.e., the adversary selects the put and get requests but does not block nodes) and not DoS attacks.

This paper builds upon the archival system by Awerbuch and Scheideler [2]. The authors consider the same past-insider DoS attack as we do in this paper. They observe that the basic approaches for distributed systems fail in this context: *Explicit data structures* (e.g., skip graphs) do not achieve correctness and robustness under DoS-attacks; *distributed hash tables* are not robust since, in our model, the adversary knows exactly where data will be replicated; and *random placement strategies* imply costly lookup operations that do not scale. Hence, a *hybrid version* of a hash table and random placement is proposed: A data item is replicated randomly in increasingly large vicinities of the location given by a (public) hash function on the data item; the larger the vicinity, the higher the lookup cost, but the smaller the probability that the adversary blocks the replica. Unfortunately, however, the random placement strategies in [2] can only handle get requests, which limits the approach to archival and information retrieval systems like Google or Akamai. Instead, our system described here can also handle *put requests* while an attack is going on. Being able to handle arbitrary combinations of put and get requests requires a significant extension of [2] which consists of a complex mix of topology and data management techniques as well as proper routing strategies, as can be seen from the lengthy description of our system in the rest of this paper.

1.3 Our Contributions

To the best of our knowledge, this is the first work to present a distributed information system that can process any set of put and get requests in a correct and scalable manner even when the system is under a past-insider attack. This is achieved with a novel put algorithm and the interplay of two distributed hash tables, a temporary and a permanent one. We show the following result:

THEOREM 1.1. *Chameleon requires only a logarithmic redundancy so that any set of put and get requests with at most one per non-blocked node can be processed in a scalable and robust manner, w.h.p., for any past-insider adversary within our model.*

Throughout the paper, *with high probability*, or *w.h.p.*, means with probability at least $1 - 1/n^c$ for a constant c that can be made arbitrarily large. A logarithmic redundancy requires Reed-Solomon codes. If coding strategies are not allowed, the redundancy of our system is $O(\log^2 n)$. The runtime needed to process all put and get requests in a phase is $O(\log^2 n)$.

Notice that we are *not* proposing a peer-to-peer system for robust storage management as n is fixed and the servers are assumed to be honest and reliable. Thus, we can afford to assume in Chameleon that all the servers know each other as these days even laptops can easily store millions of IP addresses in their main memory. Our main concern is to store the data items in a scalable way. Designing scalable and dynamic topologies of potentially untrusted sites that can withstand massive DoS attacks appears to be very challenging (if not impossible) and is left for future research.

1.4 Tools

In our paper, we will frequently make use of the well-known Chernoff bounds [16]:

THEOREM 1.2 (CHERNOFF BOUND). *Let $X_1, \ldots, X_n$ be independent binary random variables. Consider $X = \sum_{i=1}^{n} X_i$ and let $\mu = \mathrm{E}[X]$. Then it holds for all $\delta \geq 0$ that*

$$\begin{aligned}\Pr[X \geq (1+\delta)\mu] &\leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} \qquad (1)\\ &\leq e^{-\frac{\delta^2 \mu}{2(1+\delta/3)}}\end{aligned}$$

Furthermore, it holds for all $0 \leq \delta \leq 1$ that

$$\begin{aligned}\Pr[X \leq (1-\delta)\mu] &\leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu} \qquad (2)\\ &\leq e^{-\delta^2 \mu/2} .\end{aligned}$$

Inequality (1) also holds for any $\mu \geq \mathrm{E}[X]$, and Inequality (2) also holds for any $\mu \leq \mathrm{E}[X]$.

1.5 Paper Organization

The remainder of this paper is organized as follows. Our information system is described in Section 2. After giving an overview on the overall architecture and the protocols, we describe the permanent and temporary DHTs in Sections 2.1 and 2.2. In the subsequent four subsections, the four stages of our protocol are described and analyzed in detail. These stages are used to build the temporary DHT and process the requests; they are executed in every phase. The paper is concluded in Section 3.

2. THE CHAMELEON SYSTEM

The data management of the Chameleon system relies on two *stores*: the permanent *p-store*, and the temporary *t-store*. The two stores can be regarded as extensions of distributed hash tables (DHTs). The *p-store* is a static DHT (holding an arbitrary number of data items), in which the positions of the data items are fixed unless they are updated. The *p-store* is similar to the hybrid structure introduced in [2], where data is replicated at increasingly random positions in the vicinity (w.r.t. the identifier space) of the locations given by the (public) hash functions. The lookup is essentially done in increasingly large vicinities by *flooding*: this costs more in larger vicinities, but at the same time, it becomes increasingly harder for the adversary to block sufficiently many positions to block nodes with up-to-date information on the requested data item with sufficient probability.

The *t-store* is a classic dynamic DHT that constantly refreshes its topology as well as the positions of its data items in an efficient manner. The random structure of the *t-store* is hence not known to a past insider. The *t-store* is redundant in the sense that small, completely interconnected clusters of nodes are responsible for each identifier. It serves as a buffer before the data items are successfully transferred to and replicated randomly in the *p-store*. The *t-store* can afford to replace all of its data items in each phase as it provably only contains $O(n)$ many w.h.p.

On a high level, a phase of the Chameleon system proceeds as follows:

1. Build a new *t-store* from scratch and transfer all data from the old *t-store* to the new *t-store* (if possible). As we will see, the *t-store* is based on a logarithmic-degree network and there will never be too much data in the *t-store*, w.h.p., so that this step is not too expensive.
2. Process all put requests in the *t-store*.
3. Process all get requests in the *t-store* and if a get request cannot be served there (because no information is available for the given name), process it in the *p-store*.
4. Try to transfer all data items in the *t-store* to the *p-store*. Any data item that cannot be stored in the *p-store* (due to blocked, congested or overloaded nodes) is left in the *t-store*.

In the following, we start with a description of the *p-store* and the *t-store*, which is followed by a detailed description of each of the stages above. Whenever we say "for a fixed and sufficiently large constant $x \geq y$", we mean a constant x that can be any number at least y, and the larger the constant, the better is the exponent γ in our high probability bounds of the form $1 - 1/n^{\gamma}$. Sometimes, y may be large because we did not try to optimize constants. In our analysis, we will assume that our hash functions are like truly random functions, but $O(\log n)$-universal hash functions suffice for our temporary hash functions so that they can be efficiently disseminated.

2.1 The p-Store

The *p-store* is similar to the archival system by Awerbuch and Scheideler [2], with some extensions to be able to handle put requests. In the *p-store* the nodes are completely interconnected. Like in consistent hashing, nodes and data items are mapped to points in the $[0, 1)$-interval. For each $i \in \{0, \ldots, n-1\}$, node i is associated with the point i/n and *responsible* for the interval $[i/n, (i+1)/n)$, i.e., it stores all data items that are mapped to a point in its interval. Since n is a power of two, for any point $x \in [0, 1)$ with binary representation $x = \sum_{i \geq 1} x_i/2^i$, we only need the first $\log n$ bits $x_1, \ldots, x_{\log n}$ in order to determine the responsible node. Hence, w.l.o.g., we assume that all points x considered below only use $\log n$ bits.

The mapping of the data items to $[0, 1)$ is based on $c = \Theta(\log m)$ hash functions $h_1, ..., h_c : U \to [0, 1)$. This set of hash functions is fixed and hence also known by the past insider. To be useful for our system, the hash functions have to fulfill certain expansion properties to be explained later (cf also [2]).

In order to select suitable points for the data items, the *p-store* organizes the nodes into levels i that are consecutively numbered from 0 to $\log n$. 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. These locations are called the *roots* of d. For larger levels, the same number of copies is stored, but an increasing randomness is introduced in the storage locations. Thus, for larger levels, searching becomes more expensive as the entropy of the location increases. However, the probability that the adversary manages to block all copies of a data item in some level declines.

Concretely, we seek to store replicas along so-called *prefix paths* in the *p-store* (cf Figure 1). Let $pre(x, y)$ denote the length of the *longest* common prefix of x and y, that is, $pre(x, y) = i$ if and only if $x_1 = y_1, x_2 = y_2, \ldots, x_i = y_i$ and $x_{i+1} \neq y_{i+1}$. We define $T_\ell(x) = \{z \in \{0,1\}^{\log n} \mid pre(x, z) \geq \log n - \ell\}$ to be the set of all points $z \in [0, 1)$ (using the encoding $z = \sum_{i \geq 1} z_i/2^i$) such that at most ℓ of the least significant bits of x and z are different. A sequence $R = (y_\ell, y_{\ell-1}, \ldots, y_0)$ of points such that $y_0 = x$ and for each $i > 0$, $y_i \in T_i(x)$, is called a *prefix path* to x of length ℓ. The set of all possible prefix paths to x of length ℓ is denoted by $\mathcal{R}_\ell(x)$. A *random prefix path* to x is a path R that is chosen uniformly and independently at random from $\mathcal{R}_\ell(x)$. Given an $\ell \in \mathbb{N}$, let $\mathcal{T}_\ell = \{T_\ell(x) \mid x \in [0, 1)\}$. Certainly, $|\mathcal{T}_\ell| = n/2^\ell$ and each member of $\mathcal{T}_\ell$ contains 2^ℓ points.

Our goal will be to store up-to-date copies of each data item d along $\Theta(\log n)$ randomly chosen prefix paths of length $\log n$ to points in $h_1(d), \ldots, h_c(d)$. More precisely, we will try to enforce the following rule.

***p-store* Storage Rule**: For any data item d in the *p-store*, a set I_d of $\gamma_1 \log n$ indices is used to store d, where I_d has been picked uniformly at random out of at least $2c/3$ indices in $\{1, \ldots, c\}$. All roots of these indices store (the current value of) d, and for every level $\ell \geq 1$ and index $i \in I_d$ there is a node picked uniformly at random out of at least $2/3$ of the nodes in $T_\ell(h_i(d))$ that stores d.

The details on how to enforce that rule will be given when we explain the put strategy for the *p-store*. In addition to this, we will also make sure that at most $O(\log n)$ outdated copies of d are still around in each level. If this is true then the redundancy of our storage strategy is limited to $O(\log^2 n)$, and if we employ Reed-Solomon coding in each level, the redundancy can be reduced to $O(\log n)$. Each root $h_i(d)$ keeps track of the positions of all the (current and outdated) copies of d stored along prefix paths to $h_i(d)$. Thus, in order to correctly store the copies of a data item d, we have to have access to $\Omega(\log n)$ roots, which may not always be possible due to a past-insider attack. This is why we also need a *t-store*.

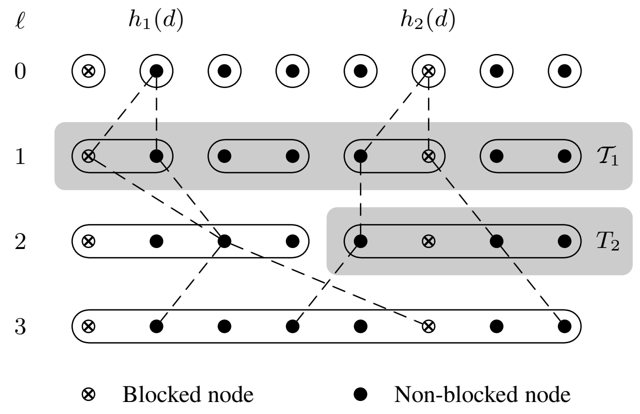

Figure 1: Replication in the p-Store.

2.2 The t-Store

In order to temporarily store data that cannot be stored in the *p-store* due to a DoS attack, we use the *t-store*. The topology of the *t-store* is a de Bruijn-like network with logarithmic node degree that is constructed from scratch in every phase. De Bruijn graphs are useful here as they have a logarithmic diameter and a high expansion (e.g., [14]). In order to form this network, we partition the $[0, 1)$-space into intervals of size $\delta \log n/n$ for some fixed and sufficiently large constant $\delta \geq 2$. For any $i \geq 0$, position $i \cdot \delta \log n/n$ is responsible for the interval $[i \cdot \delta \log n/n, (i + 1) \cdot \delta \log n/n)$. At the beginning of the current phase, each non-blocked node v in the system chooses uniformly at random one position x from the set $\{0, \delta \log n/n, 2\delta \log n/n, 3\delta \log n/n, \ldots\}$. Thus, $\delta \log n$ many nodes will share the same position on expectation and $\Theta(\delta \log n)$ many w.h.p.

Each node that selected position x tries to establish connections to all other nodes that selected the positions x (the *cluster connections*), $x_- := x - \delta \log n/n$ and $x_+ := x + \delta \log n/n$ (the *cycle connections*), and $\lfloor x/2 \rfloor_{\delta \log n/n}$ and $\lfloor (1 + x)/2 \rfloor_{\delta \log n/n}$ (the *de Bruijn connections*), where $\lfloor a \rfloor_b$ means rounding a to the closest integer multiple of b from below. This results in the union of a redundant cycle with a redundant form of the de Bruijn graph. In fact, when viewing the cluster of nodes assigned to the same position x as a single supernode and the edges between adjacent clusters as an edge between their corresponding supernodes, then the supernode graph forms the union of a cycle and a de Bruijn graph. Once the *t-store* has been established, the nodes at position 0 select a random hash function $h : U \to [0, 1)$ (by leader election) and broadcast that to all nodes in the *t-store*. The hash function determines the locations of the data items in the new *t-store*. More precisely, for any data item d in the old *t-store*, we now want to store d in the cluster responsible for $h(d)$ (i.e., whose interval contains $h(d)$) in the new *t-store*. In order to do this, each cluster of nodes from the old *t-store* will initiate appropriate insert requests for its old data items. The details are explained in the upcoming Section 2.3.

2.3 Stage 1: Building a New t-Store

We first describe how the nodes can find the nodes they are supposed to connect to in the new *t-store*. This is done with the so-called *join protocol*. Afterwards, we show how to transfer the data in the old *t-store* to the new *t-store*, which is done with the *insert protocol*.

The Join Protocol

In order to learn about its neighbors and build all necessary links between the nodes, a node v that selected position x_v issues the following five requests: $join(x_v)$, $join((x_v)_-)$, $join((x_v)_+)$, $join(\lfloor x_v/2 \rfloor_{\delta \log n/n})$ and $join(\lfloor (1 + x_v)/2 \rfloor_{\delta \log n/n})$. With the $join(x)$ operation a node tries to find all other nodes that are executing $join(x)$ for the same x. The $join(x)$ operation is executed in two substages that are synchronized among the nodes.

Preprocessing Stage.

Each non-blocked node v chooses a set U_v of $\alpha \log n$ random nodes in V for some fixed and sufficiently large constant $\alpha \geq 3$. The edge set $E = \{\{v, w\} \mid v \in V \ \wedge \ w \in U_v\}$ can be shown to form an expander graph of logarithmic degree among the non-blocked nodes w.h.p. (given that the adversary can only block a small constant fraction of the nodes). Now, this graph can be used to agree on a set of $c' = \Theta(\log n)$ random hash functions $g_1, \ldots, g_{c'} : [0, 1) \to [0, 1)$ via randomized leader election (each node guesses a random bit string and the one with lowest bit string wins and proposes the hash functions). The process is folklore and can be easily shown to require just $O(\log n)$ communication rounds w.h.p. until all non-blocked nodes are informed. Thus, we do not go into details here.

Construction Stage.

Next, each non-blocked node v sends, for each of its $join(x)$ requests, a message to the nodes owning $g_1(x), \ldots, g_{c'}(x)$ in the *p-*

store. As shown in the next lemma, this only creates a logarithmic congestion.

LEMMA 2.1. *The first round of the construction stage causes a congestion of at most $O(\log n)$ at any node w.h.p.*

PROOF. Consider some fixed node w. For any node v let the binary random variable X_v be 1 if and only if v sends a message to w. Since v chooses its point x_v uniformly at random and generates five join requests based on x_v, $\Pr[X_v = 1] \leq 5/n$ for any given hash function g_i. When summing up over all i and nodes v, this results in an expected number of $c' \cdot n \cdot 5/n = O(\log n)$ nodes. Since the X_v's are independent, it follows from the Chernoff bounds that the number of nodes contacting w is also $O(\log n)$ w.h.p. As each node generates only 5 messages, this bound also holds for the messages. □

Thus, every non-blocked node can receive all messages sent to it in that step. For a node w let M_x be the set of nodes that sent a message to it for point x. If w sends M_x to all nodes $v \in M_x$, then every node v with a $join(x)$ request will learn about all other nodes in the system with such a request, as shown in the next lemma.

LEMMA 2.2. *If the adversary can block at most $n/2$ nodes and the current phase is beyond t_0, then for every pair of nodes u and v that send out a join request for the same x there is at least one node w that receives both of their messages, w.h.p.*

PROOF. Focus on any fixed point x. Since the hash functions are chosen independently and uniformly at random, it holds for any fixed i that the probability that the owner of $g_i(x)$ is blocked is at most $1/2$. Thus, the probability that all owners of $g_1(x), \ldots, g_{c'}(x)$ are blocked is at most $(1/2)^{c'}$, which is polynomially small in n if $c' = \Theta(\log n)$ is sufficiently large. Hence, summing up over all possible points x still gives a polynomially small probability, which results in the lemma. □

Thus, the correct topology of the *t-store* can be built from the information obtained by the nodes. Note that Lemma 2.1 also holds under a current insider attack whereas Lemma 2.2 only works for a past insider. Altogether, we obtain the following result.

LEMMA 2.3. *The join protocol needs $O(\log n)$ communication rounds with congestion $O(\log n)$ w.h.p. Moreover, if the current phase is beyond t_0, then the nodes form the correct* t-store *topology at the end, w.h.p.*

The Insert Protocol

Subsequently, the data items which have been stored in the old *t-store* are transferred to the new *t-store*. In order to make sure that this does not cause too much work, we will enforce the following rule:

***t-store* Load Rule:** At any time, every cluster stores at most $\rho_1 \log n$ data items that belong to the *t-store*, for some fixed and sufficiently large constant $\rho_1 > \delta$. If that cap is exceeded, data is deleted, with a priority on the older data, until the cap is reached.

This rule is needed to ensure robustness against storage attacks also before t_0. Besides this rule, we need the following lemma, which uses the fact that the clusters are formed by random node sets that are not known by the adversary if it was already a past insider at that point.

LEMMA 2.4. *If the past phase was beyond t_0, then any adversarial attack on at most $n/3$ nodes will only block a constant fraction of the nodes in each cluster of the old* t-store, *w.h.p.*

PROOF. The lemma directly follows from the fact that the adversary does not know the membership of the clusters in the old *t-store*, and since each cluster consists of a random subset of the nodes of (sufficiently large) size $\Theta(\log n)$, the Chernoff bounds imply that the adversary will only manage to block at most half of the nodes in each cluster with a DoS attack on at most $n/3$ nodes, w.h.p. □

With the help of this lemma, we can use the following strategy: For every cluster in the old *t-store* with currently non-blocked nodes, one of its nodes (which may be determined by some randomized local leader election that can be implemented with runtime $O(\log n)$ w.h.p.) 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 w collects, for any x, all data items d to point x in D_x and forwards D_x to all nodes in M_x that contacted it in the join protocol.

From the load rule it follows similar to Lemma 2.1 that the congestion caused by the messages is $O(\log n)$ at any node w.h.p. Also, Lemma 2.2 implies that at the end all nodes in a cluster that is supposed to store d will know about d (given that the current phase is beyond t_0). Hence, we get the following result.

LEMMA 2.5. *The insert protocol needs just $O(1)$ communication rounds. Moreover, if the past phase was beyond t_0, then all data items in the old* t-store *are successfully transferred to the new* t-store *and every node (as well as cluster) in the new* t-store *has to store at most $O(\log n)$ data items, w.h.p.*

2.4 Stage 2: Put Requests in t-Store

Once the new *t-store* has been built, the new put requests are served in the *t-store*. These *put* requests cannot be served in the same way as the insert requests below since there can be multiple *put* requests to the same data item, so we need a different, more careful approach.

For each of the *put*(d) requests, we execute a *t-put*(d) request. Each *t-put*(d) request aims at storing d in the cluster responsible for $h(d)$. The *t-put* requests are sent to their destination clusters using the generic de Bruijn paths. More precisely, a request starting at point $x = (x_1, \ldots, x_{\log n})$ and ending at point $y = (y_1, \ldots, y_{\log n})$ is sent along the cluster nodes responsible for the points $x, (y_{\log n}, x_1, \ldots, x_{\log n-1}), (y_{\log n-1}, y_{\log n}, x_1, \ldots, x_{\log n-2}), \ldots, (y_2, \ldots, y_{\log n}, x_1), y$. These cluster nodes are indeed connected due to the de Bruijn rule of selecting edges. In order to handle many *t-put* requests for the same name, we use a simple, well-known filtering mechanism during the routing: Whenever two or more *t-put* requests for the same name meet in a node, then only one of them survives and the others are deleted. If a *t-put*(d) request arrives at its destination cluster and this cluster already stores an old data item d' with $name(d') = name(d)$, then d' is replaced by d.

If too many *t-put*(d) requests for *different* d have to be forwarded by a cluster (which does not happen w.h.p. unless we are in a phase before t_0), we use the following filtering mechanism to keep the congestion low.

***t-store* Routing Rule**: If *t-put* requests for more than $\rho_2 \log^2 n$ many data items pass a node at any time, for some fixed and sufficiently large constant ρ_2, then only requests for the first $\rho_2 \log^2 n$ data items are handled and the rest is deleted.

Fortunately, this rule only has to be used if we are in a phase before or at t_0, as shown in the following lemma, so for each data item with at least one *t-put*(d) request, at least one *t-put*(d) request will reach its destination cluster.

LEMMA 2.6. *If the current phase is beyond t_0 and $\rho_2 \geq 4\delta$ is a sufficiently large constant, then the number of different data items with requests leaving a cluster in a step is at most $\rho_2 \log n$ w.h.p.*

PROOF. Since the hash function h is chosen at random and each cluster contains at most $2\delta \log n$ nodes w.h.p., the de Bruijn routing strategy ensures that for any distribution of the *put*(d) requests among the nodes, the expected number data items with *t-put* requests passing a node is at most $2\delta \log n$ after i hops of the de Bruijn routing. Also, since the data items have destination clusters that are independent of each other, the Chernoff bounds imply that there are at most $4\delta \log n$ such data items w.h.p. □

Since the combining makes sure that every node will forward at most $O(\log n)$ *t-put* requests for each data item (namely, to the nodes of the next cluster in the de Bruijn path), the following result holds.

LEMMA 2.7. *Given that every non-blocked node issues at most one* t-put *request, all* t-put *requests can be served in at most $O(\log n)$ communication rounds and with congestion at most $O(\log^2 n)$ in each step, w.h.p. Moreover, at most one request is served for each name with a* t-put *request and also at least one if the current phase is beyond t_0. Also, every cluster in the new* t-store *has to store at most $O(\log n)$ data items for these requests, w.h.p.*

When combining Lemmas 2.5 and 2.7, it follows that every cluster in the new *t-store* has to store at most $O(\log n)$ data items, w.h.p., which sums up to a total of $O(n)$ data items in the new *t-store*. However, since the O-notation ignores constants, we also need to show that there is an absolute bound of $\phi \cdot n$ for some constant ϕ that is not violated over time after time point t_0. We will address this in Stage 4.

2.5 Stage 3: Processing Get Requests

The processing of the get requests proceeds in two further stages. First, the get requests are processed in the *t-store* using the *t-get* protocol (with at most one t-get request per node), and all get requests that cannot be served in the *t-store* are processed in the *p-store* using the *p-get* protocol.

The t-Get Protocol

For each *get*$(name)$ request, a *t-get*$(name)$ request is executed in the *t-store*. These requests are sent along the same routes as the *t-put* requests above. Like in the *t-put* protocol, we have to deal with the problem that multiple *t-get* requests exist for the same name. This can be handled by using combining and splitting. More precisely, whenever two or more *t-get* requests meet at some node during the routing, then only one of them is forwarded and the others are left in that node. Once the *t-get* requests have reached their destinations, they look up the requested data item, if it exists in the *t-store* and send it back to their sources along the same paths they came from. Whenever a returning *t-get* request hits a node that stores *t-get* requests to the same name (which were left behind in the forward phase), the answer of that request is stored in the other requests and all of them are sent backwards to their destinations. We use the *t-store* Routing Rule above in order to filter out *t-get* requests if requests for too many different data items pass a node.

As the forward phase of the *t-get* protocol is equivalent to the *t-put* protocol and the backward phase is just the reverse of the forward phase, the following lemma follows from Lemma 2.7.

LEMMA 2.8. *Given that every non-blocked node issues at most one* t-get *request, all* t-get *requests can be processed in at most $O(\log n)$ communication rounds and with congestion at most $O(\log^2 n)$ in each step, w.h.p. Moreover, all requests are served correctly if the current phase is beyond t_0 and every clusters serves at most $O(\log n)$* t-get *requests, w.h.p.*

The p-Get Protocol

For each destination cluster of a *t-get* request that cannot serve that *t-get* request, a *p-get* request is issued for that name in the *p-store*. Thus, we have at most one *p-get* request for each name. Distributing these *p-get* requests evenly among the nodes of each cluster results in a constant number of *p-get* requests per node w.h.p. (see Lemma 2.8). Once they have all been served, the destinations of the corresponding *t-get* requests will receive the answers which are then delivered back to the sources of the *t-get* requests in the same way as in the *t-get* protocol. Hence, it remains to describe how to execute the *p-get* protocol in the *p-store*.

For the *p-get* protocol to work, we assume that the *p-store* Storage Rule in Section 2.1 is satisfied for all requested data items (which turns out to be true for each data item that was last inserted or updated after t_0). The *p-get* protocol consists of three stages: a preprocessing stage, a contraction stage and an expansion stage. The basic approach in these stages goes back to the lookup protocol in [2] though we did some significant modifications to make the contraction stage more lightweight.

Preprocessing stage

Every non-blocked node v checks the state of $\alpha_1 \log n$ random nodes in $T_i(v)$ for every $0 \leq i \leq \log n$, for some fixed and sufficiently large constant $\alpha_1 \geq 8$. If more than $1/4$ of the (sampled) nodes in $T_i(v)$ are blocked, v declares $T_i(v)$ as *blocked* and otherwise *unblocked*. Since the checking can be done in parallel in our model, this only needs two communication rounds.

Contraction stage

Each *p-get*(d) request issued by some node v selects a random node $v_0^{(i)} \in T_{\log n}(h_i(d))$ (i.e., out of all nodes in the system) for all $i \in \{1, ..., c\}$ and aims at reaching the node responsible for $h_i(d)$ within at most $H = \beta_1 \log n$ hops, for some fixed and sufficiently large constant $\beta_1 \geq 6$. Initially, every index i is active. Let the nodes that are visited in these hops be called $v_1^{(i)}, v_2^{(i)}, \ldots$. We call a node $v_t^{(i)}$ *congested at level j* if $v_t^{(i)}$ receives more than $\beta_2 cH$ different *p-get* requests for level j at time t. For hop t, v checks if $v_{t-1}^{(i)}$ is blocked or congested. If so and $v_{t-1}^{(i)}$ was sampled out of $T_j(h_i(x))$, then $v_t^{(i)}$ is chosen at random out of $T_j(h_i(x))$, otherwise $v_i^{(i)}$ is chosen at random out of $T_{j-1}(h_i(x))$. If level $j = 0$ is reached, or a node $v_t^{(i)}$ is reached that declares $T_j(h_i(x))$ as blocked, or $t = \beta \log n$, then v stops going forward for index i. In the latter two cases, it deactivates index i at level j (i.e., i is considered to be inactive for level j) and otherwise calls i *successful*. At the end of the contraction stage, node v declares *p-get*(d) to *belong to level ℓ* where ℓ is the smallest level that contains at least $2c/3$ active indices (i.e., indices that were not deactivated at ℓ or earlier). If $\ell = 0$, then *p-get*(d) is called *successful*.

Each successful *p-get*(d) request can be immediately served by contacting all nodes in level 0 with a successful i, as implied by the *p-store* Storage Rule. The others have to continue on to the expansion stage.

The contraction stage obviously needs at most $O(\log n)$ time. Let a set $T \in \mathcal{T}_\ell$ be called *blocked* if at least $1/3$ of its nodes are blocked. Also, let T be called *congested at time t* if at least $2\beta_2 cH \cdot 2^j$ messages are sent to it in some time step t. The following lemmas hold.

LEMMA 2.9. *For any set $T \in \mathcal{T}_\ell$ that is blocked it holds that for all* p-get(d) *requests and indices i with $T = T_\ell(h_i(d))$, index i will be deactivated at a level $j \geq \ell$.*

PROOF. Suppose that set $T \in \mathcal{T}_\ell$ is blocked, and let v be any non-blocked node in T. Then, on expectation, at least $1/3$ of the

$\alpha_1 \log n$ nodes sampled by v out of T are blocked, and since these nodes are sampled independently at random, the Chernoff bounds imply that at least $1/4$ of the sampled nodes are blocked w.h.p. (given that $\alpha_1 \geq 8$ is sufficiently large). Hence, every node in T will consider T to be blocked in the preprocessing stage w.h.p. So according to the *p-get* protocol, any *p-get*(d) request will deactivate any index i with $T = T_\ell(h_i(d))$ at latest at level ℓ. □

LEMMA 2.10. *Consider any p-get(d) request and index i with $T = T_\ell(h_i(d))$ that is at level ℓ at a time when T is congested. Then index i will be deactivated at level ℓ.*

PROOF. If T is congested, then the expected congestion of every node in T is at least $2\beta_2 cH$ and therefore also more than $\beta_2 cH$ w.h.p. (if the constant β_2 is sufficiently large). Hence, according to the *p-get* protocol, all requests probing nodes in T at level ℓ for some index i will not advance to a lower level w.h.p. So the congestion will be at least as large in the remaining time steps, which implies the lemma. □

Hence, the congestion at any node cannot exceed $O(c \log^2 n)$ throughout the contraction stage, w.h.p., implying that the contraction stage is correctly executed (i.e., all requests sent to non-blocked nodes can be handled within two communication rounds so that blocked nodes are correctly identified).

LEMMA 2.11. *The preprocessing and contraction stages require at most $O(\log n)$ time and each node is involved in at most $O(c \log^2 n)$ many message transmissions per time step, w.h.p.*

It remains to prove the following crucial result, which bounds the number of *p-get* requests that belong to a certain level.

LEMMA 2.12. *If $\epsilon < 1/108$ and $\beta_2 \geq 2/\epsilon$ and at most $2n$ p-get requests have to be served, then at most $6\epsilon n/2^\ell$ of the p-get requests belong to level ℓ w.h.p.*

PROOF. Recall that U is the name universe and $m = |U|$. Let $\mathcal{H}$ be the collection of hash functions $h_1, \ldots, h_c$. Given a set $S \subset U$ of data names and a $k \in \mathbb{N}$, we call $F \subseteq S \times \{1, \ldots, c\}$ a *k-bundle* of S if every $d \in S$ has exactly k many tuples (d, i) in F. In other words, a k-bundle guarantees that each data item is represented with k different indices. Given $h_1, \ldots, h_c$ and a distance ℓ, let $\Gamma_{F,\ell}(S)$ be the union of the sets involved in these indices from $\mathcal{T}_\ell$, i.e., $\Gamma_{F,\ell}(S) = \bigcup_{(d,i)\in F} T_\ell(h_i(d))$. Given a $0 < \sigma < 1$, we call $\mathcal{H}$ a *(k, σ)-expander* if for any $\ell \leq \log n$, any $S \subseteq U$ with $|S| \leq \sigma n/2^\ell$, and any k-bundle F of S, it holds that $|\Gamma_{F,\ell}(s)| \geq 2^\ell |S|$. Similar to Lemma 1 in [2], the following claim can be shown.

CLAIM 2.13. *If the hash functions $h_1, \ldots, h_c$ are chosen uniformly and independently at random, it holds that $\mathcal{H}$ is a $(c/6, \sigma)$-expander w.h.p., for any $c \geq 12 \log m$ and $0 < \sigma \leq 1/36$.*

PROOF. We can adapt the proof of [2] for our setting. Suppose that, for randomly chosen functions $h_1, \ldots, h_c$, $\mathcal{H}$ is not a $(c/6, \sigma)$-expander. Then there exists an $i \leq \log n$ and a set $S \subset U$ with $|S| \leq \sigma n/2^i$, and a $c/6$-bundle F of S with $|\Gamma_{F,i}(S)| < 2^i|S|$. We claim that the probability $p_{s,i}$ that such a set S of size s exists is at most

$$\binom{m}{s}\binom{cs}{cs/6}\binom{n/2^i}{s} \cdot \left(\frac{s}{n/2^i}\right)^{cs/6} .$$

This holds because there are $\binom{m}{s}$ ways of choosing a subset $S \subset U$. Furthermore, there are $\binom{cs}{cs/6}$ ways of choosing $cs/6$ pairs (d, j) for F and at most $\binom{n/2^i}{s}$ ways of choosing a set W of s sets in $\mathcal{T}_i$ witnessing a bad expansion of the pairs in F. The fraction of collections $\mathcal{H}$ for which the selected pairs (d, j) indeed have the property that $T_i(h_j(d)) \subseteq W$ is equal to $(s/n/2^i)^{cs/6}$ because the hash functions $h_1, \ldots, h_c$ are chosen independently and uniformly at random.

Next we simplify $p_{s,i}$. Using the conditions on c and σ in the lemma it holds that

$$\begin{aligned}
&\binom{m}{s}\binom{cs}{cs/6}\binom{n/2^i}{s} \cdot \left(\frac{s}{n/2^i}\right)^{cs/6} \\
&\leq \left(\frac{em}{s}\right)^s (6e)^{cs/6} \left(\frac{en}{s2^i}\right)^s \left(\frac{s2^i}{n}\right)^{cs/6} \\
&= \left[\frac{em}{s} \cdot \left(6e^{1+6/c} \cdot \left(\frac{s2^i}{n}\right)^{1-6/c}\right)^{c/6}\right]^s \\
&\leq \left[m \cdot \left(6e^{1+6/c} \cdot \sigma^{1-6/c}\right)^{c/6}\right]^s = \left[m \cdot \left(\frac{1}{2}\right)^{c/6}\right]^s \leq \frac{1}{m^s}
\end{aligned}$$

if $c \geq 12 \log m$ and m is sufficiently large. Hence, summing up over all possible values of s and i, we obtain a probability of having a bad $c/6$-bundle of at most $(2 \log n)/m$, which proves the lemma. □

Given a set $T \in \mathcal{T}_\ell$ for some level ℓ, we call T *blocked* if the adversary blocks more than a third of its nodes during the DoS attack. T is called *congested* if there is a time point in which more than a third of its nodes are congested, i.e., received more than $\beta_2 cH$ different *p-put* requests for level ℓ. (Note that this definition is different from the prior definition of a congested set T.)

Let d be a data item. We call d *blocked* at level ℓ if at least $c/6$ of its c sets $T_\ell(h_i(d))$ are blocked, and we call d *weakly blocked* at level ℓ if there are blocked sets $T_{\ell_1}(h_{i_1}(d)), T_{\ell_2}(h_{i_2}(d)), \ldots, T_{\ell_k}(h_{i_k}(d))$ with $\ell_1, \ldots, \ell_k \geq \ell$, $k = c/6$, and $i_1, \ldots, i_k$ being pairwise different. Similarly, we call d *congested* at level ℓ if at least $c/6$ of its c sets $T_\ell(h_i(d))$ are congested, and we call it *weakly congested* at level ℓ if there are congested sets $T_{\ell_1}(h_{i_1}(d)), T_{\ell_2}(h_{i_2}(d)), \ldots, T_{\ell_k}(h_{i_k}(d))$ with $\ell_1, \ldots, \ell_k \geq \ell$, $k = c/6$, and $i_1, \ldots, i_k$ being pairwise different. We start with the following claim.

CLAIM 2.14. *Whenever a p-get request deactivates some index i in level $\ell \geq 0$ then there must be a level $j \geq \ell$ so that $T_j(h_i(d))$ is blocked or congested, w.h.p.*

PROOF. Suppose that none of the sets $T_\ell(h_i(d))$ is blocked or congested. Then they would also be non-congested when ignoring the request for d. In this case, the probability that index i of the *p-get* request hits a node in some level ℓ that is neither blocked nor congested is at least $1/3$ at any time. Thus, when defining the binary random variable X_t as being 1 if and only if the *p-get*(d) request hits a node that is neither blocked nor congested at time t (or level 0 has already been reached successfully), then $\Pr[X_t = 1] \geq 1/3$. Consider $X = \sum_{t=1}^{\beta_1 \log n} X_t$. Since the probability bound above for X_t holds independently of the previous $X_{t'}$'s, we can use the Chernoff bounds to prove that $X > \log n$ w.h.p., that is, index i successfully reaches level 0 w.h.p.

Thus, if a *p-get* request deactivates some index i in level $\ell \geq 0$ then there must be a level $j \geq \ell$ so that $T_j(h_i(d))$ is blocked or congested, w.h.p. □

Hence, if a *p-get*(d) request is not successful, then there are at least $c/6$ indices for which the first condition in Claim 2.14 is true, or there are at least $c/6$ indices for which the second condition is true. Together with Claim 2.14 this implies that d is either weakly blocked or weakly congested. Hence, by bounding the number of weakly blocked and congested data items we obtain an upper bound on the number of unsuccessful requests. For weakly blocked data items, the following claim holds.

CLAIM 2.15. *If s blocked nodes are sufficient for b weakly blocked data items at level ℓ then s blocked nodes are sufficient for b blocked data items at level ℓ.*

PROOF. Consider item d to be weakly blocked and let $T_{\ell_1}(h_{i_1}(d)), T_{\ell_2}(h_{i_2}(d)), \ldots, T_{\ell_k}(h_{i_k}(d))$ be the sets witnessing that with $k = c/6$. Any route through a set $T_{\ell'}(h_{i'}(d))$ with $\ell' > \ell$ contains exactly $2^{\ell'-\ell}$ sets $T \in \mathcal{T}_\ell$, and each of these sets T has a size of $|T_{\ell'}(h_{i'}(d))|/2^{\ell'-\ell}$. Thus, when distributing the nodes causing $T_{\ell'}(h_{i'}(d))$ to be blocked evenly among all $T \in \mathcal{T}_\ell$ in $T_{\ell'}(h_{i'}(d))$, we can turn any set of b weakly blocked data items into blocked data items at level ℓ. □

A similar claim also holds for weakly congested data items:

CLAIM 2.16. *If s congested nodes are sufficient for b weakly congested data items at level ℓ then s congested nodes are sufficient for b congested data items at level ℓ.*

Now we are ready to bound the number of weakly blocked and weakly congested data items. First, consider the weakly blocked data items. If the adversary can block up to ϵn nodes (where ϵ combines here the blocked and overloaded nodes), at most $3\epsilon n/2^\ell$ of the $n/2^\ell$ sets in $\mathcal{T}_\ell$ can be blocked, which covers at most $3\epsilon n$ nodes. Suppose the attacker can block a set S of data items at level ℓ. Then there is a $c/6$-bundle F for S, i.e., we can identify $c/6$ indices to blocked sets. Due to Claim 2.13, if $|S| \leq \sigma n/2^\ell$ then $|\Gamma_{F,\ell}(S)| \geq 2^\ell |S|$. As $\Gamma_{F,\ell}(S)$ is of size at most $3\epsilon n$, we have that $|S| \leq 3\epsilon n/2^r$, which is less than $\sigma n/2^\ell$ (so that Claim 2.13 implies an upper bound on $|S|$) if $3\epsilon < 1/36$, or $\epsilon < 1/108$. Hence, if the adversary can block up to ϵn nodes, this entails at most $3\epsilon n/2^\ell$ blocked data items at level ℓ. Together with Claim 2.15 this implies that if the adversary can block at most ϵn nodes, then there are at most $3\epsilon n/2^\ell$ weakly blocked data items at level ℓ.

It remains to bound the number of weakly congested data items. Recall that the contraction stage uses a congestion bound of $\beta_2 c H$. Thus, when we have a total of at most $2n$ *p-get* requests, at most $6n/(\beta_2 H 2^\ell)$ of the $n/2^\ell$ sets in $\mathcal{T}_\ell$ can be congested at a single time point and therefore at most $6n/(\beta_2 2^\ell)$ of the sets in $\mathcal{T}_\ell$ can be congested over all time points. When setting $\beta_2 = 2/\epsilon$, this means at most $3\epsilon n/2^\ell$ congested sets in $\mathcal{T}_\ell$. Using the same arguments as for the blocked sets together with Claim 2.16 implies that there can be at most $3\epsilon n/2^\ell$ weakly congested data items at level ℓ.

Thus, altogether at most $6\epsilon n/2^\ell$ data items are weakly blocked or congested at level ℓ, which implies that at most $6\epsilon n/2^\ell$ *p-get* requests belong to level ℓ. □

Expansion stage

The expansion stage works in the same way as for the lookup protocol in [2]. For completeness, we present it here again.

The expansion stage proceeds in rounds numbered from 1 to $\log n$. In round r, every *p-get* request for some data item d that belongs to round $r' \leq r$ and is not finished yet sends a message of the form $(d, r, i, -)$ (where "$-$" is an empty placeholder for a most current copy of d) for each index i that was still active in the level the request belongs to. This message is sent to the non-blocked node v in $T = T_r(h_i(d))$ that was successfully contacted at level r in the contraction stage. Each such node v remembers the nodes that sent messages to it in the set S_v and stores the messages it received from them into its active pool of messages A_v, one copy for each $(d, r, i, -)$. If $|A_v| > 3c/\sigma$ (for a constant σ satisfying Claim 2.13), then any set of messages is discarded from A_v to get down to $|A_v| = 3c/\sigma$. For any remaining $(d, r, i, -)$ in A_v for which v stores a copy b of d (due to the data storage strategy defined above), $(d, r, i, -)$ is replaced by (d, r, i, b). Afterwards, every node v in the system executes the following push strategy $O(\log n)$ many times:

- v sends every message (d, r, i, b) in A_v to a random node in $T_r(h_i(d))$.
- For each message (d, r, i, b) received by a non-blocked node v, v first checks whether A_v already contains some message (d, r, i, b'). If so, the message with the most current copy of d is kept and the other is deleted. Otherwise, v checks if $|A_v| = 3c/\sigma$. If so, v discards the message.

If after these steps $|A_v| = 3c/\sigma$, then v sends for each node $w \in S_v$ with original message $(d, r, i, -)$ the message $(d, r, i, *)$ back to w, where the "$*$" indicates that v was too congested. Otherwise, v sends (d, r, i, b) in A_v back to w.

Each *p-get* request that receives at most $c/6$ many replies of the form $(d, r, i, *)$ (among the at least $2c/3$ replies) returns the message (d, r, i, b) with the most up-to-date b (which may also be "$-$" if no copy was found) to whoever generated the request and is finished. Otherwise, it continues to participate in round $r + 1$.

The runtime of the expansion stage is $O(\log^2 n)$. Using the bound on the number of requests belonging to level ℓ in Lemma 2.12, one can show inductively with the same arguments as in Lemma 2.12 that the number of requests belonging to a level $\ell' < \ell$ that are not finished in level ℓ is at most $6\epsilon n/2^\ell$. Hence, together with those requests that belong to level ℓ, there are at most $12\epsilon n/2^\ell$ requests that the expansion stage has to take care of at level ℓ. Finally, at level $\log n$, all remaining requests can be finished w.h.p. (cf [2]).

Whenever a *p-get* request finishes, it will receive the most up-to-date copy of a data item for at least $c/2$ indices, w.h.p., and since none of the sets $T_\ell(h_i(d))$ explored for that were blocked w.h.p. (see Lemma 2.9), it follows from the *p-store* Storage Rule at least least $c/6$ sets $T_\ell(h_i(d))$ were successfully explored that contain an up-to-date copy of d (in a potentially blocked node) and at most a third of their nodes are blocked, so w.h.p. there is at least one set $T_\ell(h_i(d))$ in which the up-to-date copy of d is in a non-blocked node (if d has already been inserted). This implies the following result.

LEMMA 2.17. *Given that the current phase is beyond t_0 and there are at most n p-get requests with at most a constant number per node, all p-get requests are served correctly in at most $O(\log^2 n)$ communication rounds, w.h.p.*

Note that the expansion stage is the only part in a phase whose runtime exceeds $O(\log n)$. Otherwise, a phase would just need $O(\log n)$ time. A runtime of $O(\log^2 n)$ is only necessary if the adversary can adaptively choose the *p-get* requests in order to create a high congestion in some parts of the system. If the names for the *p-get* requests are selected independently of the hash functions $h_1, \ldots, h_c$, then each *p-get* request only has to do broadcasts in the level it belongs to in the expansion stage, which can be done in $O(\log n)$ steps w.h.p.

2.6 Stage 4: Transferring Items

Finally, we try to transfer all items stored in the *t-store* (i.e., the old and new ones) to the *p-store* using the *p-put* protocol; if the transfer of a certain data item d is successful, that is, if sufficiently many replicas of d can be stored correctly in the *p-store*, the corresponding data item in the *t-store* is removed. Otherwise, the item is left in the *t-store*. From the *t-store* Load Rule and Lemma 2.7 it follows that if every cluster evenly distributes the *p-put* requests among its nodes, then each node only has to issue at most a constant number of *p-put* requests.

The p-Put Protocol

The *p-put* protocol consists of three stages: a preprocessing stage, a contraction stage and a permanent storage stage.

Preprocessing Stage.

Like in the *p-get* protocol, every non-blocked node v checks the state of $\alpha_1 \log n$ random nodes in $T_i(v)$ for every $0 \leq i \leq \log n$, for some fixed and sufficiently large constant $\alpha_1 \geq 8$. If more than $1/4$ of the (sampled) nodes in $T_i(v)$ are blocked, v declares $T_i(v)$ as *blocked* and otherwise *unblocked*. Also, each non-blocked node v picks $\alpha_2 \log n$ random nodes from the entire node set for a fixed and sufficiently large constant α_2. If at most half of them are blocked (which is the case w.h.p. when $\epsilon < 1/3$) then v computes the average data load $\bar{L}_v$ of the non-blocked nodes in the *p-store*. Since the checking can be done in parallel in our model, this only needs two communication rounds. The following lemma can be shown for $\bar{L}_v$.

LEMMA 2.18. *Let $\bar{L}$ be the average load in the system and $L_{\max}$ be the maximum load at a node. If $L_{\max} \leq 2\lambda\bar{L}$, $\epsilon \leq 1/(8\lambda)$, and $\alpha \geq 36\lambda$ is sufficiently large then for every node v, $\bar{L}_v \in [\bar{L}/2, 2\bar{L}]$ w.h.p.*

PROOF. Let $\bar{L}$ and $L_{\max}$ be defined as in the lemma. First, we prove an upper bound on $\bar{L}_v$. If $\epsilon \leq 1/3$ then no matter which ϵ-fraction of the nodes is shut down by the adversary, the average load of the non-blocked nodes, $\bar{L}_{nb}$, is at most

$$(n \cdot \bar{L})/(1-\epsilon)n \leq (3/2)\bar{L}.$$

Consider any node v and let $L_1, \ldots, L_k$ be random variables denoting the loads of the $k = \alpha \log n$ random nodes picked by v. Given that previously $L_{\max} \leq 2\lambda\bar{L}$, $L_i \leq 2\lambda\bar{L}$ for every i, and $\mathrm{E}[L_i] \leq (3/2)\bar{L}$. Hence, for $L = \sum_{i=1}^{k} L_i$ it holds that $\mathrm{E}[L] \leq (3k/2)\bar{L}$. Furthermore, the Chernoff-Hoeffding bounds imply that, for any $0 < \delta \leq 1$,

$$\Pr[L \geq (1+\delta)\mathrm{E}[L]] \leq e^{-\delta \mathrm{E}[L]/(3L_{\max})} .$$

Thus, we have $L \leq 2k\bar{L}$ w.h.p. if the constant $\alpha \geq 36\lambda$ is sufficiently large.

Next, we prove a lower bound on $\bar{L}_v$. If $L_{\max} \leq 2\lambda\bar{L}$ and $\epsilon \leq 1/(8\lambda)$, then no matter which ϵ-fraction of the nodes is shut down by the adversary, the average load of the non-blocked nodes, $\bar{L}_{nb}$, is at least

$$(n \cdot \bar{L} - \epsilon n \cdot 2\lambda\bar{L})/(1-\epsilon)n \geq (3/4)\bar{L}.$$

Hence, $\mathrm{E}[L_i] \geq (3/4)\bar{L}$ for every i, which implies that $\mathrm{E}[L] \geq (3k/4)\bar{L}$. Furthermore, the Chernoff-Hoeffding bounds imply that, for any $0 < \delta < 1$,

$$\Pr[L \leq (1-\delta)\mathrm{E}[L]] \leq e^{-\delta^2 \mathrm{E}[L]/(2L_{\max})}.$$

Thus, $L \geq k\bar{L}/2$ w.h.p. if the constant $\alpha \geq 36\lambda$ is sufficiently large. □

If v's own data load L_v satisfies $L_v > \lambda \cdot \bar{L}$ for some fixed and sufficiently large constant $\lambda \geq 4$, then it considers itself to be overloaded and will behave in the rest of the *p-put* protocol as if it is blocked when contacted by other requests. As v will not get any new data in this case, Lemma 2.18 guarantees that there will never be a node (w.h.p.) whose load exceeds $2\lambda\bar{L}$, which satisfies our scalability requirement in Section 1.1. Also, the number of overloaded nodes is not too high as stated by the following lemma.

LEMMA 2.19. *The number of nodes that consider themselves to be overloaded is at most $2n/\lambda$, w.h.p.*

It immediately follows from Lemma 2.18 and the fact that there can be at most $2n/\lambda$ nodes with a load of more than $(\lambda/2)\bar{L}$. Thus, if λ is sufficiently large, we can just treat all of them as being blocked, which is done so for the rest of the description and analysis of the *p-put* protocol.

Contraction Stage.

The contraction stage of the *p-put* protocol is identical to the contraction stage of the *p-get* protocol. Lemma 2.12 immediately implies the following result.

LEMMA 2.20. *If $\epsilon < 1/108$ and $\beta_2 \geq 2/\epsilon$ and at most $2n$ p-put requests have to be served, then at most $12\epsilon n$ of the p-put requests are unsuccessful w.h.p.*

Those *p-put* requests that successfully made it to level 0 will be served in the *p-store* as described below. The other at most $12\epsilon n$ requests will remain in the *t-store*. Given that the *t-store* had at most $2n$ data items initially, it has to serve at most $(1+12\epsilon)n \leq 2n$ *p-put* requests in the next round, so the number of data items remaining in the *t-store* is stable w.h.p.

Permanent Storage Stage.

Each node whose *p-put*(d) request was successful selects $\gamma_1 \log n$ random indices among the active indices of d and deactivates all others for some fixed and sufficiently large constant γ_1. Let i be an index that remains active.

We want to prevent the accumulation of obsolete data items in our system. In order to achieve this, we maintain in the node responsible for $h_i(d)$ — d's root node — information about the nodes storing a copy of d w.r.t. index i. In order to support *updates* of a data item d in our system, we use this information to remove all out-of-date copies of d w.r.t. i. Clearly, since some nodes may be blocked, this may not always be possible. If it is not possible, references to these out-of-date copies are left in the roots so that they may be deleted at some later *p-put* request. If more than $\gamma_2 \log n$ out-of-date copies remain for some fixed and sufficiently large constant γ_2 (which would only happen w.h.p. if the system is under an insider attack, as we will see) then d is only updated in the root $h_i(d)$. Otherwise, we select a random non-blocked node in each $T_\ell(h_i(d))$ with $\ell \in \{0, \ldots, \log n\}$ (which requires at most $O(\log n)$ attempts w.h.p.), store an up-to-date copy of d in these nodes, and store references to these nodes in $h_i(d)$.

LEMMA 2.21. *Given that at t_0 the total number of (obsolete and up-to-date) copies of data item d in the p-store is $O(\log^2 n)$ (which is enforced by the permanent storage stage), the number of copies of d remains $O(\log^2 n)$ w.h.p. at any time after t_0.*

PROOF. Consider some fixed data item d, index i and level ℓ. Certainly, every node $v \in T_\ell(h_i(d))$ will only store one copy of d at a time because whenever it receives a newer copy, the older one will be deleted. Let the random variable X_t be one if and only if v stores a copy of d for index i and level ℓ at the beginning of phase t, and let $p_t = \Pr[X_t = 1]$. Suppose that v is blocked at some phase t in which d is updated. Then $p_{t+1} = p_t$ as nothing changes for v. Otherwise, suppose that v is non-blocked. If i is not active for the *p-put*(d) request, then $p_{t+1} = p_t$ as well. Otherwise, $p_{t+1} \leq 3/2^\ell$ as $T_\ell(h_i(d))$ contains at least $2^\ell/3$ non-blocked nodes w.h.p. and a random set of $\gamma \log n$ of these nodes is picked for the up-to-date copies of d. Hence, given that the number of obsolete copies of d was $O(\log^2 n)$ at time point t_0, the expected number of obsolete copies of d remains at $O(\log^2 n)$. This also holds w.h.p. as the probabilities are negatively correlated (see, e.g., [26] for Chernoff bounds of negatively correlated random variables). □

3. CONCLUSION

This paper has shown for the first time how to build a scalable dynamic information system that is robust against a past insider. Several important questions remain open. First of all, note that we did not try to optimize constants; from a practical perspective, it is crucial to get them to smaller (resp. larger) values. It would also be interesting to study whether the runtime of a phase can be reduced to $O(\log n)$—only the *p-get* protocol prevents that—and whether our algorithms can be simplified. An important challenge on our research agenda is to explore whether our concepts can be adapted to bounded-degree peer-to-peer systems with potentially unreliable peers. Finally, although we believe that our replication factors are optimal, we still do not have a lower bound.

4. REFERENCES

[1] B. Awerbuch and C. Scheideler. Towards a Scalable and Robust DHT. In *Proc. 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 318–327, 2006.

[2] B. Awerbuch and C. Scheideler. A Denial-of-Service Resistant DHT. In *Proc. 21st International Symposium on Distributed Computing (DISC)*, 2007.

[3] D. Bernstein. SYN Cookies. In *http://cr.yp.to/syncookies.html*, 2008.

[4] A. Bhargava, K. Kothapalli, C. Riley, C. Scheideler, and M. Thober. Pagoda: A Dynamic Overlay Network for Routing, Data Management, and Multicasting. In *Proc. 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 170–179, 2004.

[5] D. Dittrich, J. Mirkovic, S. Dietrich, and P. Reiher. *Internet Denial of Service: Attack and Defense Mechanisms*. Prentice Hall PTR, 2005.

[6] P. Druschel and A. Rowstron. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001. See also `http://research.microsoft.com/∼antr/Pastry`.

[7] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proc. 4th Conference on USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[8] J. Ioannidis and S. M. Bellovin. Implementing Pushback: Router-Based Defense Against DDoS Attacks. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2002.

[9] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-Sale: Surviving Organized DDoS Attacks that Mimic Flash Crowds. In *Proc. 2nd Conference on Symposium on Networked Systems Design & Implementation (NSDI)*, pages 287–300, 2005.

[10] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 654–663, 1997.

[11] F. Kargl, J. Maier, and M. Weber. Protecting Web Servers from Distributed Denial of Service Attacks. In *Proc. World Wide Web (WWW)*, 2001.

[12] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *Proc. ACM SIGCOMM*, pages 61–72, 2002.

[13] G. Lawton. Stronger Domain Name System Thwarts Root-Server Attacks. In *IEEE Computer*, pages 14–17, May 2007.

[14] T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes*. Morgan Kaufmann Publishers (San Mateo, CA), 1992.

[15] Mazu Networks Inc. http://mazunetworks.com. 2008.

[16] McDiarmid. Concentration. In M. Habib, C. McDiarmid, J. Ramirez-Alfonsin, and B. Reed, editors, *Probabilistic Methods for Algorithmic Discrete Mathematics*, pages 195–247. Springer Verlag, Berlin, 1998.

[17] K. Mehlhorn and U. Vishkin. Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories. *Acta Inf.*, 21(4):339–374, 1984.

[18] J. Mirkovic and P. Reiher. A Taxonomy of DDoS Attacks and Defense Mechanisms. *ACM SIGCOMM Computer Communications Review*, 34(2), 2004.

[19] W. G. Morein, A. Stavrou, D. L. Cook, A. D. Keromytis, V. Misra, and D. Rubenstein. Using Graphic Turing Tests to Counter Automated DDoS Attacks Against Web Servers. In *Proc. 10th ACM Int. Conference on Computer and Communication Security (CCS)*, pages 8–19, 2003.

[20] M. Naor and U. Wieder. Novel Architectures for P2P Applications: The Continuous-Discrete Approach. *ACM Trans. Algorithms*, 3(3), 2007.

[21] G. Oikonomou, J. Mirkovic, P. Reiher, and M. Robinson. A Framework for Collaborative DDoS Defense. In *Proc. Annual Computer Security Applications Conference (ACSAC)*, 2006.

[22] V. N. Padmanabhan and K. Sripanidkulchai. The Case for Cooperative Networking. In *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 178–190, 2002.

[23] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[24] E. Ratliff. The Zombie Hunters. In *The New Yorker*, 2005.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. ACM SIGCOMM*, 2001.

[26] C. Scheideler. *Probabilistic Methods for Coordination Problems*. HNI-Verlagsschriftenreihe 78, University of Paderborn, 2000.

[27] T. Stading, P. Maniatis, and M. Baker. Peer-to-Peer Caching Schemes to Address Flash Crowds. In *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 203–213, 2002.

[28] A. Stavrou, D. Rubenstein, and S. Sahu. A Lightweight, Robust P2P System to Handle Flash Crowds. In *Proc. 10th IEEE International Conference on Network Protocols (ICNP)*, pages 226–235, 2002.

[29] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *Proc. ACM SIGCOMM*, 2002.

[30] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Kalakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Technical Report MIT*, 2002.

[31] M. Walfish, H. Balakrishnan, D. Karger, and S. Shenker. DoS: Fighting Fire with Fire. In *Proc. Workshop on Hot Topics in Networks (HotNets)*, 2005.

[32] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS Defense By Offense. *SIGCOMM Comput. Commun. Rev.*, 36(4):303–314, 2006.

[33] X. Yang, D. Wetherall, and T. Anderson. A DoS-Limiting Network Architecture. In *Proc. ACM SIGCOMM*, 2005.