A DoS-Resilient Information System for Dynamic Data Management

Matthias Baumgart, Christian Scheideler, Stefan Schmid

TUM-I0906
März 09

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

Matthias Baumgart, Christian Scheideler, Stefan Schmid

Institut für Informatik, Technische Universität München,
Boltzmannstraße 3, D-85748 Garching, Germany
{baumgart,scheideler,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 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 replication policy requiring a polylogarithmic overhead only. All requests in Chameleon can be processed in polylogarithmic time and work at every node.

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 [WBKS05, WVB$^+$06]. Examples of such attacks include downloading large files [Rat05], causing computationally expensive operations [KKJB05], or just overloading servers with junk. Some popular information services like Google and Akamai are under constant DoS attacks, and the Domain Name System has been hit several times by major DoS attacks during the last years [Law07].

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 be 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

[*]Research supported by the DFG project "Algorithms for Self-Stabilizing Overlay Networks".

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 [AS05], 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, 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, 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 the time proceeds in time steps that are synchronized among the nodes. Note however that using local synchronizers, our algorithms also work in asynchronous settings. All we need is a bounded transmission time between two 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 acknowledgement 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 or 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.

1.2 Related Work

Due to their importance, DoS attacks are a well-studied problem (e.g., [DMDR05, MR04] 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 [Maz08], blacklisting particular IP addresses, using TCP SYN cookies [Ber08], pushback [IB02], etc., problematic [WBKS05]. This observation has led some researchers to propose means how legitimate clients can "speak up" and thus be identified [WBKS05, WVB+06], 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., [BKR+04, DR01, HJS+03, RFH+01, SML+02]). In particular, we follow a *consistent hashing* approach [KLL+97] in order to store the data and employ the *continuous-discrete techniques* presented in [NW03] for communication between the servers.

DoS-resistant systems based on DHTs have already been studied in [KMW01, KMR02, MSC+03]. For instance, the Secure Overlay Services approach [KMR02] uses *proxies* on Chord to defend against flooding DoS attacks. A Chord overlay is also used by the Internet Indirection Infrastructure *i3* [SAZ+02] to achieve resilience to DoS attacks. Other DoS limiting architectures have been proposed in [OMRR06, YWA05]. 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 [PS02], Backslash [SMB02] or PROOFS [SRS02], and there is also theoretical work [NW03]. 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 [MV84] allow one to overcome these limitations [AS06], 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 [AS05]. The authors consider the same past-insider DoS attack as we do in this paper, but the strategies there can only handle get requests, which limits their approach to archival and information retrieval systems like Google or Akamai. Instead, our system 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 [AS05] which consists of a complex mix of topology and data management techniques as well as proper routing strategies, as can be seen from the quite 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. In particular, this paper shows 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.

2 The Chameleon System

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. 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 DHTs.

While the t-store is a dynamic DHT that constantly refreshes its topology as well as the positions of its data items, the p-store is a static DHT, in which the positions of the data items are fixed unless they are updated. The t-store can afford to replace all of its data items in each phase as it only holds $O(n)$ many, while the p-store may hold an arbitrary number of data items. 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 [AS05], 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. 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 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*. Let $pre(x, y)$ denote 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 $i > 0$, $y_i \in T_\ell(x)$, 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 $h_1(d), \ldots, h_c(d)$ (where the randomness may have some bias due to blocked and congested nodes). 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 an past-insider attack. This is why we also need a t-store. More details about how to select prefix paths for the copies will be given when we explain the put strategy for 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., [Lei92]). 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 $\{0, \delta \log n/n, 2\delta \log n/n, 3\delta \log n/n, ...\}$. 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, then the supernodes form 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 issues the following five requests: $join(x)$, $join(x_-)$, $join(x_+)$, $join(\lfloor x/2 \rfloor_{\delta \log n/n})$ and $join(\lfloor (1+x)/2 \rfloor_{\delta \log n/n})$. With the $join(x)$ operation, for any position x, a node tries to find all other nodes that are executing $join(x)$ for the same x. The $join(x)$ operation is executed in four substages that are synchronized among the nodes.

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 $\alpha_1 \geq 3$. If more than half of the nodes in $T_i(v)$ is 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. Afterwards, each non-blocked node v chooses a set U_v of $\alpha_2 \log n$ random nodes in V for some fixed and sufficiently large constant $\alpha_2 \geq 3$. 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). This graph can then 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). 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.

Contraction Stage.

Initially, all join requests are active. Each $join(x)$ request issued by some node v selects a random node $v_0^{(i)} \in T_{\log n}(g_i(x))$ (i.e., out of all nodes in the system) for all $i \in \{1, ..., c'\}$ and aims at reaching the node responsible for $g_i(x)$ within at most $\beta \log n$ hops, for some fixed and sufficiently large constant $\beta \geq 6$. Let the nodes that are visited in these hops be called $v_1^{(i)}, v_2^{(i)}, \ldots$ For hop t, v checks if $v_{t-1}^{(i)}$ is blocked or not. If $v_{t-1}^{(i)}$ is blocked and $v_{t-1}^{(i)}$ was sampled out of $T_j(g_i(x))$, then $v_t^{(i)}$ is chosen at random out of $T_j(g_i(x))$, otherwise $v_i^{(i)}$ is chosen at random out of $T_{j-1}(g_i(x))$. If the level $j = 0$ is reached, or a node $v_t^{(i)}$ is reached that declares $T_j(g_i(x))$ as blocked, or $t = \beta \log n$, then v stops going forward for index i and deactivates index i at level j. At the end of the contraction stage, node v declares $join(x)$ 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).

The contraction stage obviously needs at most $O(\log n)$ time. The following lemma also states a logarithmic congestion bound, 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 2.1. *The preprocessing and contraction stages require at most $O(\log n)$ time, and each node is involved in at most $O(\log n)$ many message transmissions per time step, w.h.p.*

Proof. Since the time bound is obvious, it remains to prove the congestion bound. We just focus here on the congestion of the contraction stage. Recall that each node is the origin of 5 join requests that are based on some point x chosen independently at random out of $[0, 1)$. Consider some fixed node v and a $join(x)$ request that is currently at level j for some fixed j (that may or may not depend on other requests). Given that x is chosen at random, the probability that $join(x)$ probes node v is equal to

$$\Pr[v \in T_j(x)] \cdot \Pr[v \text{ chosen} \mid v \in T_j(x)] = (2^j/n) \cdot 1/2^j = 1/n$$

as both probabilities are independent of the probabilities for the requests issued by other nodes, and we have to sum up the congestion over $\log n + 1$ different levels, the Chernoff bounds imply that the congestion at any time is $O(\log n)$ w.h.p. □

Moreover, we can show the following two lemmas, which will help us in the analysis of the next stage.

Lemma 2.2. *For every $join(x)$ request belonging to level ℓ and every active index $i \in \{1, ..., c'\}$ in that level, $T_{\ell'}(g_i(x))$ contains at least $2^{\ell'}/3$ non-blocked nodes for every $\ell' \geq \ell$ w.h.p.*

Proof. Consider any $join(x)$ request belonging to some level ℓ and let $i \in \{1, ..., c'\}$ be any active index in that level. Suppose that there is a set $T_{\ell'}(g_i(x))$ for some $\ell' \geq \ell$ that contains less than $2^{\ell'}/3$ non-blocked nodes. In the preprocessing stage, each node $v \in T_{\ell'}(g_i(x))$ samples $\alpha_1 \log n$ nodes out of $T_{\ell'}(g_i(x))$. Each sample has a probability of more than $2/3$ to be a blocked node. Hence, the Chernoff bounds imply that at least half of the samples will be blocked nodes, w.h.p., so v will declare $T_{\ell'}(g_i(x))$ as being blocked. Hence, during the contraction stage index i must have been deactivated when passing $T_{\ell'}(g_i(x))$, which contradicts our assumption that i is still active at level ℓ. □

Lemma 2.3. *If $\epsilon < 1/72$, then for every $\ell \in \{0, \ldots, \log n\}$ there are at most $6\epsilon n/2^\ell$ points whose join requests belong to level ℓ w.h.p.*

Proof. We start with some notation. Let P be the set of all possible points and $\mathcal{G}$ be the collection of hash functions $g_1, \ldots, g_{c'}$. We know that $|P| \leq n$. Given a set S of points and a $k \in \mathbb{N}$, we call $F \subseteq S \times \{1, \ldots, c'\}$ a *k-bundle* of S if every $x \in S$ has exactly k many tuples (d, i) in F. In other words, a k-bundle guarantees that each point is represented with k different indices. Given $g_1, \ldots, g_{c'}$ and a level ℓ, let $\Gamma_{F,\ell}(S)$ be the union of the sets involved in these indices from T_ℓ, i.e., $\Gamma_{F,\ell}(S) = \bigcup_{(d,i)\in F} T_\ell(g_i(x))$. Given a $0 < \sigma < 1$, we call $\mathcal{H}$ a *(k, σ)-expander* if for any $\ell \leq \log n$, any $S \subseteq P$ 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 [AS05], the following claim can be shown.

Claim 2.4. *If the hash functions $g_1, \ldots, g_{c'}$ are chosen uniformly and independently at random, it holds that $\mathcal{G}$ is a $(c'/3, \sigma)$-expander w.h.p., for any $c' \geq 6 \log n$ and $0 < \sigma \leq 1/24$.*

Let D_ℓ be the set of points x with join requests that become inactive at level ℓ due to too many inactive indices. For any ℓ and any $T \subseteq \mathcal{T}_\ell$, we call T *blocked* if the attacker blocks more than a third of its nodes with its DoS attack. Consider any point x. We call x *blocked* at level ℓ if at least $c'/3$ of its c' sets $T_\ell(g_i(x))$ are blocked, and we call it *weakly blocked* in level ℓ if there are blocked sets $T_{\ell_1}(g_{i_1}(x)), T_{\ell_2}(g_{i_2}(x)), \ldots, T_{\ell_k}(g_{i_k}(x))$ with $\ell_1, \ldots, \ell_k \geq \ell$ and $k = c'/3$ and $i_1, \ldots, i_k$ being pairwise different. Let WB_ℓ denote the set of weakly blocked data items at level ℓ. We start with the following claim.

Claim 2.5. *Whenever a $join(x)$ request deactivates some index i in level $\ell \geq 1$, then $T_\ell(g_i(x))$ is blocked, w.h.p.*

Proof. Consider any fixed $x \in [0,1)$ and $i \in \{1,\ldots,c'\}$. First, suppose that i is deactivated at some level j because some node $v_t^{(i)}$ is visited in that level that declares $T_j(x)$ as being blocked. In this case, more than half of the $\alpha_1 \log n$ random nodes sampled by $v_t^{(i)}$ in $T_j(x)$ must have been blocked in the preprocessing stage, where γ is a (sufficiently large) constant. If, however, $T_j(x)$ contains at most $2^j/3$ many blocked nodes, then the probability for each sampling to hit a blocked node is at most 1/3, so the expected number of blocked nodes noticed is at most $\alpha_1 \log n/3$. Since the samples are made independently at random, it follows from the Chernoff bounds that the probability that more than $\alpha_1 \log n/2$ blocked nodes are sampled is polynomially small in n (where the exponent depends on α_1). Hence, if some node $v_t^{(i)}$ is visited in that level that declares $T_j(x)$ as being blocked, then $T_j(x)$ must contain at least $2^j/3$ blocked nodes w.h.p.

It remains to prove the lemma for the case that i is deactivated because $t = \beta \log n$, the end of the contraction stage has been reached. Lemma 2.2 implies that if there is a level ℓ with at least $(2/3)2^\ell$ blocked nodes in $T_\ell(x)$, index i cannot pass it as it will be deactivated there w.h.p. Suppose that there is no level ℓ so that $T_\ell(x)$ contains at least $(2/3)2^\ell$ blocked nodes. Then the sequence of nodes $v_0^{(i)}, v_1^{(i)}, v_2^{(i)}, \ldots$ will need at most 3 probes on expectation to lower the level by 1. This can be modeled as a sequence of binary random variables $X_0, X_1, X_2, \ldots$ with X_i being 1 if and only if the level is lowered by 1. Since $\Pr[X_i = 1] \geq 1/3$ independently of the other random variables, the Chernoff bounds (for positively correlated random variables) can be used to prove that it takes at most $\beta \log n$ many hops for a sufficiently large constant β until level 0 is reached w.h.p. (given that no non-blocked node is reached that declares its level as being blocked, which is covered by the case at the beginning of the proof). In fact, if the constant β in the $\beta \log n$ bound for the hops is at least 6, then the sequence will also end at level 0 in the contraction stage w.h.p.

Combining all cases, the claim follows. □

Suppose that a $join(x)$ request becomes inactive at level ℓ due to at least $c'/3$ deactivated indices. Then it follows from Claim 2.5 that x is weakly blocked, w.h.p. For weakly blocked points, the following claim holds.

Claim 2.6. *If s blocked nodes can cause a set of b weakly blocked points at level ℓ, then s blocked nodes can also cause a set of b blocked points at level ℓ.*

Proof. Consider point x to be weakly blocked, and let $T_{\ell_1}(g_{i_1}(x)), T_{\ell_2}(g_{i_2}(x)), \ldots, T_{\ell_k}(g_{i_k}(x))$ be the sets witnessing that with $k = c/3$. Any route through a set $T_{\ell'}(g_{i'}(x))$ with $\ell' > \ell$ contains exactly $2^{\ell'-\ell}$ sets $T \in \mathcal{T}_\ell$, and each of these sets T has a size of $|T_{\ell'}(g_{i'}(x))|/2^{\ell'-\ell}$. Thus, when distributing the nodes causing $T_{\ell'}(g_{i'}(x))$ to be blocked evenly among all $T \in \mathcal{T}_\ell$ in $T_{\ell'}(g_{i'}(x))$. We can turn any set of b weakly blocked points into blocked points at level ℓ. □

If the adversary can block at most ϵn nodes, then 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 points at level ℓ. Then there is a $c'/3$-bundle F for S. According to Claim 2.4, it holds that $|\Gamma_{F,\ell}(S)| \geq 2^\ell |S|$ if $|S| \leq \sigma n/2^\ell$. Since the largest possible size of $\Gamma_{F,\ell}(S)$ is $3\epsilon n$, it follows that $|S| \leq 3\epsilon n/2^\ell$, which is less than $\sigma n/2^\ell$ (so that Claim 2.4 implies an upper bound on $|S|$) if $3\epsilon < 1/24$, or $\epsilon < 1/72$. Hence, if the adversary can block at most ϵn nodes, then it can cause at most $3\epsilon n/2^\ell$ blocked points x in level ℓ. According to Claim 2.6, this implies that $|WB_\ell| \leq 6\epsilon n/2^\ell$. Since Claim 2.5 implies that $|D_\ell| \leq |WB_\ell|$, w.h.p., the lemma follows. □

Interestingly, the lemma even holds if the adversary knows $g_1, \ldots, g_{c'}$.

Expansion Stage.

The expansion stage starts with $\gamma_1 \log n + 1$ dissemination rounds numbered from 0 to $\gamma_1 \log n$, where $\gamma_1 \geq 9$ is a fixed and sufficiently large constant. In round 0, every $join(x)$ request from some node v that belongs to level ℓ sends a message (v, x, ℓ') to $\gamma_2 \log n$ random nodes in $T_{\ell'}(g_i(x))$ for every index i that was still active at level ℓ', where $\ell' \geq \ell$ is the smallest value so that $|T_{\ell'}(g_i(x))| \geq \gamma_2 \log n$ and $\gamma_2 \geq 96$ is a fixed and sufficiently large constant. In this and the other rounds, all messages that have been sent to some node w are recorded by w, and multiple messages of the same form are merged into one. In each round $r \geq 1$, every node w sends every message (v, x, ℓ) recorded by it with $\gamma_1 \ell \geq r$ to a random node in $T_\ell(w)$. If there is a level ℓ for which w receives more than $\gamma_3 c \log n$ many messages for some fixed and sufficiently large constant $\gamma_3 \geq 6\delta$ or a message of the form (ℓ, ∞), then w deletes all of them and replaces them by (ℓ, ∞), which means that there are too many messages for level ℓ in the set $T_\ell(w)$. Let us call a set $T \in \mathcal{T}_\ell$ *non-congested* if there are at most $\gamma_3 c \log n$ different messages (v, x, ℓ) sent to T in round 0 (which implies that a message (ℓ, ∞) will not be created in it). We can show the following result.

Lemma 2.7. *For any non-congested set $T \in \mathcal{T}_\ell$ and any message (v, x, ℓ) sent to it, at least $1/3$ of its non-blocked nodes store (v, x, ℓ) at the end of the dissemination rounds, w.h.p.*

Proof. First of all, it follows from Lemma 2.2 that for any node v with $join(x)$ request that belongs to level ℓ it holds for every active index i that $T_\ell(h_i(x))$ contains at least $2^\ell/3$ non-blocked nodes w.h.p. Consider some fixed $T = T_\ell(h_i(x))$ with such a property. Since v sends out messages of the form (v, x, ℓ) to $\gamma_2 \log n$ random nodes in T, on expectation, at least $(\gamma_2/3) \log n$ non-blocked nodes will be informed, and also at least $(\gamma_2/6) \log n$ w.h.p. if γ_2 is sufficiently large (which follows from the Chernoff bounds). Hence, the message (v, x, ℓ) will not get lost initially.

Suppose that T is non-congested, i.e., it receives at most $\gamma_3 \log n$ different messages in round 0. In this case, a node in it will never create the message (ℓ, ∞) which would delete other messages of the form (v, x, ℓ) in T. So we can focus on the spreading of a message (v, x, ℓ) in T. Suppose that k non-blocked nodes in T are currently informed about (v, x, ℓ), where $k \leq m/3$ and $m \geq 2^\ell/3$ is the number of non-blocked nodes in T. Then the probability that an uninformed node $w \in T$ will be informed in the next round is equal to $k/2^\ell$. Hence, the expected number of uninformed non-blocked nodes that will be informed is

$$(m - k)k/2^\ell \geq [(2/3)m/2^\ell]k \geq (2/9)k$$

and at least $k/9$ w.h.p. (due to the Chernoff bounds) if $\gamma_2 \geq 96$ is a sufficiently large constant. Standard calculations yield that $t \geq 9\ell$ rounds are sufficient until $(1 + 1/9)^t \geq m/3$, so at least a third of the non-blocked nodes will know (v, x, ℓ) at the end of the dissemination rounds w.h.p. □

At the end of the expansion stage, every node w sends each node v that sent a message (v, x, ℓ) to it in round 0 a message containing all nodes u with entries (u, x, ℓ') in w for any ℓ'. Finally, every node v with a $join(x)$ request will tell all nodes u reported to it in this way that it has sent a $join(x)$ request as well.

As v has sent out (v, x, ℓ) to $\gamma_2 \log n$ many nodes in round 0 for each active index i, and for any such index, a third of the nodes in $T_\ell(g_i(x))$ is non-blocked w.h.p. (Lemma 2.2), we can show the following lemma.

Lemma 2.8. *For any two non-congested sets $T \in \mathcal{T}_\ell$ and $T' \in \mathcal{T}_{\ell'}$ with $T \subseteq T'$ it holds for any two messages (v, x, ℓ) and (v', x, ℓ') with $x \in T$ that v will be notified about v' in the second last communication round w.h.p.*

Proof. Consider any two non-congested sets $T \in \mathcal{T}_\ell$ and $T' \in \mathcal{T}_{\ell'}$ with $T \subseteq T'$ and any two messages (v, x, ℓ) and (v', x, ℓ') with $x \in T$. We know from Lemma 2.7 that at least $1/3$ of the non-blocked nodes in T' will know (v', x, ℓ') at the end of the broadcasting rounds w.h.p., which implies together with Lemma 2.2 that at least $1/9$ of the nodes in T' will know (v', x, ℓ') at the end of the broadcasting rounds w.h.p. The probability that a specific non-blocked node in T belongs to these nodes is at least $1/9$. On the other hand, we know from the proof of Lemma 2.7 that after round 0 at least $(\gamma_2/6)\log n$ non-blocked nodes in T will know (v, x, ℓ) w.h.p. The probability that none of them knows (v', x, ℓ') is at most $(1-1/9)^{(\gamma_2/6)\log n}$ which is polynomially small in n for $\gamma_2 \geq 96$. Hence, v will learn about v' in the second last communication round w.h.p., which finishes the proof. $\square$

Also, the following lemma holds, which is based on Lemma 2.3.

Lemma 2.9. *If the current phase is beyond t_0, then all sets $T \in \mathcal{T}_\ell$ used in the expansion stage are non-congested w.h.p.*

Proof. Consider some fixed level ℓ. We define a set $T \in \mathcal{T}_\ell$ to be *non-blocked* if for every $\ell' \geq \ell$, at most $1/3$ of the nodes in the $T' \in \mathcal{T}_{\ell'}$ with $T \subseteq T'$ are blocked. The following claim holds. Its proof follows from the insights of Claim 2.6.

Claim 2.10. *Given that the adversary can block at most ϵn nodes, there are at least $3\epsilon n/2^\ell$ non-blocked sets $T \in \mathcal{T}_\ell$.*

From Lemma 2.3 we know that there are at most $6\epsilon n/2^\ell$ points x whose join requests belong to level ℓ. Let P be the set of these points. Let the indices of the corresponding join requests be partitioned in any way into active and inactive indices so that at most $c'/3$ indices of any join request are declared inactive. Since the hash functions $g_1, \ldots, g_{c'}$ are chosen uniformly and independently at random, it follows that the active indices distribute among a group of sets $T \in \mathcal{T}_\ell$ that includes all non-blocked sets in $\mathcal{T}_\ell$, according to Claim 2.5. Since there are at least $3\epsilon n/2^\ell$ non-blocked sets in $\mathcal{T}_\ell$, and each of them would be successfully passed w.h.p., it follows that each of them has a probability of at most $2^\ell/(3\epsilon n)$ of being selected by an active index. The other sets in $T \in \mathcal{T}_\ell$ have a lower probability as it is not guaranteed any more that an active index would pass T w.h.p. Hence, the expected congestion due to active indices in any $T \in \mathcal{T}_\ell$ is at most

$$(6\epsilon n/2^\ell) \cdot (2\delta \log n) \cdot c \cdot 2^\ell/(3\epsilon n) = 4\delta c \log n$$

where the first term is the number of points, the second the maximum number of join requests per point, the third the maximum number of active indices and the last our probability bound. Furthermore, since the probability distribution over the sets in $\mathcal{T}_\ell$ applies independently for each index, the Chernoff bounds imply that the congestion in any $T \in \mathcal{T}_\ell$ is at most $6\delta c \log n$ w.h.p. Hence, if $\gamma_3 \geq 6\delta$, then the lemma follows. $\square$

We need the fact that the hash functions $g_1, \ldots, g_{c'}$ are chosen at random and that they are not known to the adversary. Let us now recall what we know so far. Let v be a node with a $join(x)$ request belonging to the lowest level among all other nodes v' with join requests to x. We know that any request belonging to level ℓ has at least $2c'/3$ active indices in ℓ. Hence, v and v' share a common active index i, so Lemma 2.8 implies that v will learn about v' in the second last communication round w.h.p. Thus, after the last communication round, every node v with $join(x)$ knows every other node v' with $join(x)$, which implies the following lemma.

Lemma 2.11. *At the end of the expansion stage, every node v with a $join(x)$ request knows all other nodes with a $join(x)$ request, w.h.p.*

Obviously, the runtime of the expansion stage is $O(\log n)$, and given that every node will send out at most $O(c \log n)$ messages to random nodes for each level ℓ, every node will receive at most $O(c \log n)$ messages for each level ℓ w.h.p., which yields the following result.

Lemma 2.12. *The expansion stage requires at most $O(\log n)$ time, and each node is involved in at most $O(\log^2 n)$ many message transmissions per time step, w.h.p.*

Construction Stage.

Finally, the network of the t-store is built from the member information the nodes obtained from their join requests. Since the nodes already have all the connectivity information they need for that, this does not involve any communication.

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 $\rho_1 \geq 2\delta$. If that cap is exceeded, data is deleted, with a priority on the older data, until the cap is reached.

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 2.13. *If the past phase was beyond t_0, then any adversarial attack within our model 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. $\square$

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.) calls $insert(d)$ for each of the data items d stored in it. The insert requests are sent along 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. As (1) the hash function for the new t-store is chosen at random, (2) there is at most one $insert(d)$ request for each data item d, and (3) each node is the starting point of at most $O(\log n)$ many data items (w.h.p.), it follows from standard Chernoff bounds that the congestion caused by the routing problem is $O(\log^2 n)$ in each cluster w.h.p. Hence, all requests reaching a cluster in a time unit can be passed on in the next time unit, which implies the following result.

Lemma 2.14. *All insert requests can be served by the t-store in at most $O(\log n)$ communication rounds, w.h.p. Moreover, 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: Processing all put requests in the t-store

Once the new t-store has been built, the new put requests are served in the t-store, with at most one put request per node (to enforce our model). For each of these $put(d)$ requests, we execute a t-$put(d)$ request that is routed along the same path as described above for the insert requests. However, the critical issue remains that there might be many t-put requests for the same name. To solve the problem, we use a simple 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.

In order to bound the congestion for this routing problem with combining, it suffices to determine the number of distinct data items d whose t-$put(d)$ requests pass through the same cluster. This can easily be shown to be $O(\log^2 n)$ w.h.p. using standard Chernoff bounds, as long as the adversary does not know h (i.e., the current phase is beyond t_0). To prevent too much congestion in case the adversary knows h, the following simple rule suffices:

t-Store Routing Rule: If more than $\rho_2 \log^2 n$ many t-store messages pass a node at any time, for some fixed and sufficiently large constant $\rho_2 \geq 2\delta$, then any set of messages is deleted to get their number down to $\rho_2 \log^2 n$.

Since de Bruijn routing is used, each cluster receives messages from only two other clusters, which implies together with the congestion bound for the distinct data items that each cluster sends and receives at most $O(\log^2 n)$ messages within any time step. This implies the following lemma.

Lemma 2.15. *If at most one t-put request is issued per node, all t-put requests can be served in at most $O(\log n)$ communication rounds, w.h.p. Moreover, only one update for each name is successfully stored and 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 2.14 and 2.15, 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 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 all 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 insert and 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.

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 2.15.

Lemma 2.16. *Given that we are beyond time t_0 and every non-blocked node issues at most one t-get request, every cluster has to serve at most $O(\log n)$ t-get requests and all t-get requests can be served in at most $O(\log n)$ time, 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 w.h.p. (see Lemma 2.16). 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.

The p-get protocol is similar to the lookup protocol in [AS05], with two differences. (1) In Chameleon, a get request will proceed to the next level in the contraction stage only if at least $5c/6$ of its indices are still active (in [AS05] the limit is $3c/4$ indices) and (2) in our system, we do not have to deal with multiple p-get requests to the same name. Point (1) (as well as the fact that a node may initiate a constant number of p-get requests and not just one) can be handled with a slight adaptation of the analysis in [AS05] and point (2) just simplifies the situation studied [AS05].

Lemma 2.17. *Given that we are beyond time t_0 and every non-blocked node issues at most a constant number of p-get requests, all p-get requests are served correctly in at most $O(\log^2 n)$ communication rounds, w.h.p.*

Note that the p-get protocol is the only protocol whose runtime exceeds $O(\log n)$, otherwise a phase would just need $O(\log n)$ time. However, a runtime of $O(\log^2 n)$ seems only necessary if the system is under adversarial attack. It is easy to modify the lookup protocol in [AS05] to obtain a p-get protocol so that as long as there is no attack, its runtime is $O(\log n)$ w.h.p. (see also [AS06]).

2.6 Stage 4: Transferring the data items from the t-store to the p-store

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 2.15 it follows that if every cluster evenly distributes the p-put requests among its nodes, then each node only has to issue a constant number of p-put requests.

The p-Put Protocol

The p-put protocol consists of three substages: a preprocessing stage, a contraction stage and a storage stage. Recall the preprocessing stage of the join protocol in which every non-blocked node v checks the state of $\Theta(\log n)$ random nodes in $T_i(v)$ for every $0 \leq i \leq \log n$. If more than half of the nodes in $T_i(v)$ is blocked, v declares $T_i(v)$ as blocked and otherwise unblocked. We will use that information in the contraction stage as well.

Preprocessing Stage.

Every non-blocked node v picks $\alpha_4 \log n$ random nodes from the entire node set for a fixed and sufficiently large constant α_4. 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. The following lemma can be shown for this.

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 $\alpha_4 \geq 24\lambda$ 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, $\bar{L}_a$, 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 $\delta \geq 1$,

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

Thus, $L \leq 2\bar{L}$ w.h.p. if the constant α 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, $\bar{L}_a$, 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, $L \geq \bar{L}/2$ w.h.p. if the constant α 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 $\lambda \geq 4$ (or more than half of the sampled nodes are blocked), 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. *If $\epsilon \leq 1/(8\lambda)$, 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 for the further analysis.

Contraction Stage.

Each p-$put(d)$ 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 $\beta \log n$ hops, for some fixed and sufficiently large constant $\beta \geq 6$. This is done in the same way as in the join protocol. If the level $j = 0$ is reached, or a node $v_t^{(i)}$ is reached that declares $T_j(h_i(x))$ as blocked, or a node $v_t^{(i)}$ is reached that received more than $\beta' c$ p-put requests with the same index i during the current time step, or $t = \beta \log n$, then v stops going forward and deactivates index i at level j, where β' is a sufficiently large constant. At the end of the contraction stage, node v declares $join(x)$ to belong to level ℓ, where ℓ is the smallest level that contains at least $2c/3$ active indices.

The contraction stage obviously needs at most $O(\log n)$ time. Moreover, we can show the following crucial result.

Lemma 2.20. *If $\epsilon < 1/144$ and $\beta' \geq 1/\epsilon$, then at most $8\epsilon n$ of the data in the t-store does not belong to level 0 w.h.p.*

Proof. We will apply concepts similar to the proof of Lemma 2.3. 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 T_ℓ, 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 [AS05], the following claim can be shown.

Claim 2.21. *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 $0 < \sigma \leq 1/36$.*

Our goal is to upper bound the number of p-put requests which do not reach level 0. This analysis follows the same lines as analyzing the lookup protocol in [AS05].

Given a set $T \in \mathcal{T}_\ell$ for some level ℓ, we call T *blocked* if the adversary blocks more than a fourth of the nodes during the DoS attack. Analogously, T is called *congested* if more than a fourth of the nodes in T have a congestion of at least $\beta' c$.

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 round r 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.

Recall Claim 2.6. Furthermore, we make the following observation, whose proof follows along the same lines as the proof of Claim 2.5.

Claim 2.22. *Whenever a request for some data item d deactivates some index i at level ℓ, the set $T_\ell(h_i(d))$ is either blocked or congested, w.h.p.*

We assume that a request for some data item d becomes inactive at level ℓ due to at least $c/3$ deactivated indices. By the pigeon hole principle, there are at least $c/6$ indices for which the first condition in Lemma 2.22 is true, or there are at least $c/6$ indices for which the second condition is true. Together with Lemma 2.22 this implies that d is either weakly blocked or weakly congested.

If the adversary can block up to ϵn nodes, at most $4\epsilon n/2^\ell$ of the $n/2^\ell$ sets in $\mathcal{T}_\ell$ can be blocked, which covers at most $4\epsilon n$ 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 or congested sets. Due to Claim 2.21, 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 $4\epsilon n$, we have that $|S| \leq 4\epsilon n/2^r$, which is less than $\sigma n/2^\ell$ (so that Claim 2.21 implies an upper bound on $|S|$) if $4\epsilon < 1/36$, or $\epsilon < 1/144$. Hence, if the adversary can block up to ϵn nodes, this entails at most $4\epsilon n/2^\ell$ blocked data items at level ℓ. Together with Lemma 2.6 this implies that if the adversary can block at most ϵn nodes, then there are at most $4\epsilon n/2^\ell$ weakly blocked data items at level ℓ.

Now, we observe that a weakly blocked data item at level ℓ is also weakly blocked in each level ℓ' with $\ell' < \ell$. The same holds for weakly congested data items. Hence, we have to determine the number of weakly blocked data items in round 0 and the number of weakly congested data items in round 0, respectively. As seen above, the number of weakly blocked and the number of weakly congested data items cannot be larger than $4\epsilon n$, that is, we have at most $8\epsilon n$ weakly blocked or congested data items in total. From this observation, we directly obtain the lemma. $\square$

Since all p-put requests that made it to level 0 will be served in the p-store, the t-store is left with at most n data items after the phase and at most $2n$ data items during the next phase, w.h.p., which makes sure that none of them is removed in the next phase (if the constants ρ_1 and ρ_2 in our t-store rules are sufficiently large).

Permanent Storage Stage.

Each data item d whose p-put request does not manage to reach level 0 will remain in the t-store. Otherwise, select $\gamma_1 \log n$ random indices among the active indices of d and deactivate 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 2.23. *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 for Chernoff bounds of negatively correlated random variables). $\square$

When combining all of our results, we obtain Theorem 1.1.

3 Conclusion

This paper has shown for the first time how to build an information system that is robust to an important class of DoS attacks where a past-insider adversary can bring down a constant fraction of the servers. Our solution is efficient in the sense that information is only replicated by a polylogarithmic factor, and put and get requests require polylogarithmic work and time. We believe that these properties renders our solution interesting both from a theoretical and a practical point of view.

However, several important questions remain. In future research, we want to investigate whether the runtime of a phase can be reduced to a logarithmic bound (only the p-get protocol prevents that) and whether our algorithms can be simplified to something more compact and easier to implement. Also we would like to explore whether our concepts 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.

References

[AS05] Baruch Awerbuch and Christian Scheideler. A Denial-of-Service Resistant DHT. In *Proc. 16th International Symposium on Algorithms and Computation (ISAAC)*, 2005.

[AS06] Baruch Awerbuch and Christian Scheideler. Towards a Scalable and Robust DHT. In *Proc. 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 318–327, 2006.

[Ber08] D.J. Bernstein. SYN Cookies. In *http://cr.yp.to/syncookies.html*, 2008.

[BKR$^+$04] Ankur Bhargava, Kishore Kothapalli, Chris Riley, Christian Scheideler, and Mark 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.

[DMDR05] D. Dittrich, J. Mirkovic, S. Dietrich, and P. Reiher. *Internet Denial of Service: Attack and Defense Mechanisms*. Prentice Hall PTR, 2005.

[DR01] 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`.

[HJS$^+$03] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proc. 4th Conference on USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[IB02] John Ioannidis and Steven M. Bellovin. Implementing Pushback: Router-Based Defense Against DDoS Attacks. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2002.

[KKJB05] Srikanth Kandula, Dina Katabi, Matthias Jacob, and Arthur 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.

[KLL$^+$97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel 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.

[KMR02] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *Proc. ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 61–72, 2002.

[KMW01] F. Kargl, J. Maier, and M. Weber. Protecting Web Servers from Distributed Denial of Service Attacks. In *Proc. World Wide Web (WWW)*, 2001.

[Law07] G. Lawton. Stronger Domain Name System Thwarts Root-Server Attacks. In *IEEE Computer*, pages 14–17, May 2007.

[Lei92] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes*. Morgan Kaufmann Publishers (San Mateo, CA), 1992.

[Maz08] Mazu Networks Inc. http://mazunetworks.com. 2008.

[MR04] J. Mirkovic and P. Reiher. A Taxonomy of DDoS Attacks and Defense Mechanisms. *ACM SIGCOMM Computer Communications Review*, 34(2), 2004.

[MSC$^+$03] 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.

[MV84] Kurt Mehlhorn and Uzi Vishkin. Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories. *Acta Inf.*, 21(4):339–374, 1984.

[NW03] Moni Naor and Udi Wieder. Novel Architectures for P2P Applications: the Continuous-Discrete Approach. In *Proc. 15th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 50–59, 2003.

[OMRR06] G. Oikonomou, J. Mirkovic, P. Reiher, and M. Robinson. A Framework for Collaborative DDoS Defense. In *Proc. Annual Computer Security Applications Conference (ACSAC)*, 2006.

[PS02] Venkata N. Padmanabhan and Kunwadee Sripanidkulchai. The Case for Cooperative Networking. In *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 178–190, 2002.

[Rat05] E. Ratliff. The Zombie Hunters. In *The New Yorker*, 2005.

[RFH$^+$01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of the ACM SIGCOMM*, 2001.

[SAZ$^+$02] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet Indirection Infrastructure. In *Proc. ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2002.

[SMB02] Tyron Stading, Petros Maniatis, and Mary 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.

[SML$^+$02] 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.

[SRS02] Angelos Stavrou, Dan Rubenstein, and Sambit Sahu. A Lightweight, Robust P2P System to Handle Flash Crowds. In *Proc. 10th IEEE International Conference on Network Protocols (ICNP)*, pages 226–235, 2002.

[WBKS05] Michael Walfish, Hari Balakrishnan, David Karger, and Scott Shenker. DoS: Fighting Fire with Fire. In *Proc. Workshop on Hot Topics in Networks (HotNets)*, 2005.

[WVB$^+$06] Michael Walfish, Mythili Vutukuru, Hari Balakrishnan, David Karger, and Scott Shenker. DDoS Defense By Offense. *SIGCOMM Comput. Commun. Rev.*, 36(4):303–314, 2006.

[YWA05] X. Yang, D. Wetherall, and T. Anderson. A DoS-Limiting Network Architecture. In *Proc. ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2005.