Balanced-Replication Algorithms for Distribution Trees*

Edith Cohen	Haim Kaplan†
AT&T Labs–Research	School of Computer Science
Florham Park	Tel-Aviv University
NJ 07932 USA	Tel-Aviv 69978, Israel

February 12, 2004

Abstract

In many Internet applications, requests for a certain object are routed bottom-up over a tree where the root of the tree is the node containing the object. When an object becomes popular, the root node of the tree may become a hot-spot. Therefore many applications allow intermediate nodes to acquire the ability to serve the requests, for example by caching the object. We call such distinguished nodes *primed*. We propose and analyze different algorithms where nodes decide when to become primed; these algorithms balance the maximum load on a node and the number of primed nodes.

Many applications require both fully distributed decisions and smooth convergence to a stable set of primed nodes. We first present optimal algorithms which require communication across the tree. We then consider the natural previously proposed THRESHOLD algorithm, where a node becomes primed when the incoming flow of requests exceeds a threshold. We show examples where THRESHOLD exhibits undesirable behavior during convergence. Finally, we propose another fully distributed algorithm, GAP, which converges gracefully.

1 Introduction

The Internet provides a platform for decentralized applications where content or services are requested, stored, and provided by very large number of loosely coupled hosts. In these applications, there is no centralized support. Requests are forwarded from peer to peer, and each peer can become a server for each item.

Many supporting architectures for such applications are such that the search pattern for each request is "routed" that is, every peer has a list of neighbors, and for each request a peer might

*A preliminary version of this paper appeared in the Proceedings of the 10th European Symposium on Algorithms (ESA'2002) [4].

†This work was supported, in part, by the Israel Science Foundation (ISF) grant 548/00.

receive, there is a preferred neighbor that brings the request "closer" to its destination. Therefore, the forwarding pattern of each request forms a tree, with edges directed towards the root. We call this tree a *distribution tree*. The stream of requests traveling up the distribution tree is called a *flow*.

When a certain request becomes popular, the respective root of its distribution tree can become a hot-spot. In this case, hosts further down the tree can "replicate" the service – that is, acquire the ability to satisfy these requests, e.g., by "caching" data or duplicating code. The focus of this paper is proposing and evaluating algorithms for hosts to decide when to become replicas.

Before providing further details on our model, we describe some concrete applications and architectures where this setup arises. A relatively old existing application is the Domain Name Service (DNS). The DNS service already offers caching of records at intermediate nodes; in fact, caching is the default at all "intermediate" servers and DNS records are cached until they expire. A forward looking application is to place HTTP caches "near" Internet routers, augmenting these routers with the capability to intercept and forward HTTP traffic; such a service can shorten latency and network load.[1] The growing popularity of Peer-to-Peer applications (e.g., [8, 2]) sparked the development of architectures for fully distributed name-lookup, which enable clients to locate objects or services. Two such proposed architectures are Chord [18] (which is based on consistent hashing [9]) and CAN [15] ("Content Addressable Network"). With Chord and CAN, objects have keys which are hashed to points in some metric space S. Each node is responsible for some segment of the space S. Note that the metric of S has no correspondence to the underlying network distances. Each node maintains a small number of pointers $P(v)$ to nodes responsible for other regions of S. When a node receives a request for an object with key k, it forwards it to the node in $P(v)$ responsible for a region containing the point closest to the hash of k (according to the metric on S). Thus, the search paths for each object form a distribution tree, with the root of the tree being the node responsible for the region of the metric space where the hash of the object lies. Note that even though these hashing-based architectures balance the load well across different objects, replication is still necessary for avoiding hot-spots caused by highly skewed per-object demands. Thus, to alleviate hot-spots, it is proposed in [15, 18] that intermediate nodes cache items (or keys).

In all these applications, replication on intermediate nodes constitutes a tool in avoiding "hot-spots" at the roots of distribution trees. Such hot-spots are caused by flash-crowd events or objects with persistent high demand. We would like to emphasize that generally, each distribution tree corresponds to a single "item," and thus one might worry about interaction of flows from different trees. In many of these applications, however, the balance between trees is taken care of by the process selecting these trees – e.g., in CAN and CHORD, the trees are such that nodes are essentially assigned randomly to their positions in different trees. Thus, it is reasonable to consider the trees

[1] These caches can possibly be active caches refreshing stale items not necessarily upon request. See e.g. []

independently of each other and focus on a single tree, which is the problem we model and address in this paper.

We say that we *prime* a node when we give it the ability to serve requests. We refer to a set of primed nodes as an *allocation*. We look for algorithms that select an allocation. We model the problem and propose metrics for the quality of a solution. The objective is to balance two parameters: the maximum load of a node, which we would like to be as small as possible, and the total size of the allocation (number of primed nodes), which we would also like to be as small as possible. The objective of minimizing the number of primed nodes models a wide range of applications where being primed consumes resources (e.g., storage space). Other desirable properties pertain to the complexity and communication requirements of the algorithm; and the stability of the allocation. We consider two models: The first, called the *must-serve* model, requires that a primed node will serve all the flow it gets. The second, called the *serve-or-forward* model allows a node to serve some of the flow it gets and forward the rest to its parent.

In the first part of the paper we assume that there is a strict upper bound of C on the amount of flow that a primed node can serve, and on the amount of flow that a single leaf can generate. Then it is obvious that to serve a total of R flow we need to prime at least $\lceil R/C \rceil$ nodes. We prove in Section 3 that if the tree has large degrees then we may in fact have to prime $\Omega(R\Delta/C)$ nodes, where Δ is the maximum degree.

In Section 4 we consider the problem of finding the optimal allocation. Both for the *must-serve* and the *serve-or-forward* models, we present algorithms that given the tree and the flow generated at each leaf calculate a smallest allocation in time almost linear in the size of the tree. These algorithms can be implemented distributively but with some overhead of two rounds of passing messages up and down the tree.

In Section 5 we consider a simple and natural algorithm proposed with both Chord and CAN which we call the THRESHOLD algorithm. The THRESHOLD algorithm runs locally at each node and primes the node when the incoming flow is above a certain threshold. An advantage of the THRESHOLD algorithm is that it requires no communication other than the incoming flow itself. We provide bounds on the size of the allocation obtained by THRESHOLD and show that it is no more than $R\Delta/C$, the same as the worst-case lower bound.

According to the THRESHOLD algorithm, the decision of a node whether to prime itself or not depends on its incoming flow, which in turn depends on the priming decisions of its descendants. This implies that when nodes run THRESHOLD asynchronously, they may change their state from primed to unprimed multiple times before reaching a stable allocation. This dependency may result in instability. In particular when the incoming flow changes, the size of interim allocations may be much larger than the size of the final stable one. Furthermore, small changes in the incoming flow may cause large changes in the allocation. Such instability is a disadvantage for any application where there is a cost associated with a change of state from primed to unprimed or vice versa. For example to serve requests for a particular big file you have to store a copy of the file. This may be an overhead even when this storage is only temporary, in particular if the node participates in several such trees.

Finally, in Section 6, we propose a different local algorithm called GAP. Like THRESHOLD, GAP also runs independently at each node and makes a local decision. Unlike THRESHOLD, GAP makes its decision at a node v based on the total flow generated at leaf descendants of v. Notice that the value of the total flow generated below a particular node is independent of the decisions made by other nodes. Thus GAP converges immediately to an allocation which is stable. Furthermore GAP's allocation is unlikely to change significantly due to minor changes in the flow.

The algorithm GAP is randomized and as a result a node primed by GAP may end up serving more than C units of flow. We show that, with an appropriate choice of parameters, the probability that a primed node serves more than C units of flow is small. Furthermore, we show that by running GAP iteratively logarithmically many time, one can find an allocation of expected size $O(\frac{R\Delta}{C})$ in which no node serves more than C units of flow with high probability.

Related Work

There has been extensive work on caching and replicating content in order to improve response time and reduce the load on particular servers. In particular several groups have proposed and analyzed may aspects of cooperating caches [5, 7, 13, 3, 10]. This work is related to ours only in the sense that load is distributed among many caches. The main question considered in these papers is how to distribute the load among the cooperating caches assuming typically that each client can send requests to any cache. We ask a different question that is which nodes should we distinguish as caches so that there are no hot-spots on one hand, and not too many nodes become caches on the other hand.

A recent analysis of traffic of Internet content delivery systems [16] suggests that P2P traffic now accounts for the majority of HTTP bytes transferred, exceeding traffic due to WWW accesses by nearly a factor of three. P2P documents are large so P2P flows last longer, and many of them are concurrently open. All this, and the fact that a small number of extremely large objects account for a large fraction of the traffic, indicates the large possible gain of replication in a P2P system. The large file sizes also indicates that there a non-negligible overhead in priming a node, even temporarily.

The same study [16] also observed that outbound and inbound P2P traffic through a node at the border of some local network may be imbalanced. This indicates that the location of a node in the underlying network may effect the load it has in the overlay distribution trees which it belongs too. It is an interesting topic for future work to try to model this relation between distribution trees and the underlying overlay network, and its effect on the ideas that we present here.

Plaxton, Rajaraman, and Richa proposed a related scheme that also takes into account the underlying topology when overlaying the distribution network [14]. Other related work focused on balancing load through replication, but mainly addressed a different aspect – the layout of the distribution trees (placing caches as "nodes") on top of larger network (e.g. [1, 11]). A conceptually similar problem is that of placing proxy caches in an existing distribution tree as to minimize average number of hops. This problem was explored by Li et al, which provided a quadratic dynamic programming algorithm [12]. The different objective function we use versus the one of [12] is due to a basic difference between the applications they considered and ours. Placing caches in a network is slow manual operation that is better optimized via a centralized calculation. In our setting, nodes already have "cache" capabilities and the goal is to best utilize resources and to do so in a distributed manner.

2 Model

A *distribution tree* is a rooted tree with flow generated at the leaves and flowing towards the root. We suggest algorithms that would *prime* some nodes in the tree. A primed node can serve flow coming into it while a non-primed node only forwards flow to its parent. The goal is to prime as few as possible nodes such that all flow is serviced subject to constraints that we define below. As explained in the introduction, flow corresponds to requests, and *priming* corresponds to configuring the node such that it is able to service requests.

Formally, the input to our problem is the tree T and a function $f : \{v \mid v \text{ is a leaf}\} \rightarrow R_+$ mapping each leaf v to a non-negative real which describes the flow generated at v. We also assume another input parameter, denoted by C, which is a number that bounds the maximum amount of flow a node can serve. We also assume that each leaf generates at most C units of flow.

An algorithm that solves the problem outputs an *allocation*, which is a subset of the nodes that are primed. The conditions which the allocation has to satisfy vary between the following two service models. These two models differ by whether a primed node is allowed to forward flow or not.

• The *must-serve* model: In this model a primed node serves all its incoming flow. An allocation constitutes a solution in this model if when we cut the tree by removing the edge from each primed node to its parent then the flow generated at the leafs of each subtree is at most C (and therefore can be served at the root of this subtree). In other words, an allocation is a good solution if when we process nodes from the leafs up forwarding all the flow into an unprimed node to its parent then each time we reach a primed node it has at most C units of flow coming into it.

• The *serve-or-forward model*: In this model a primed node serves at most C units of the flow coming into it and forwards the rest. Here an allocation constitutes a solution if the following process ends such that the amount of flow reaching the root is at most C if it is primed and 0 if it is not primed. We process the nodes from the leaves up. When an unprimed node is processed we "forward" all its flow to the parent; when a primed node is processed we assign it to serve all the flow it receives, up to C units, and forward the rest to its parent.

In both models an allocation constitutes a solution if there is an *assignment* of all flow to primed nodes (each unit of flow is assigned to a node on its way to the root) such that the *load* (assigned flow) of each primed node is at most C and the load of other nodes is 0. The *size* of an allocation is the number of nodes it primes. An allocation is *optimal* if it is correct and of minimum size.

The must-serve model is more restrictive than the serve-or-forward model; thus must-serve allocations constitute serve-or-forward allocations. Figure 1 illustrates an example flow serviced in the two service models. The serve-or-forward model is appropriate for applications where "servicing" is a more expensive operation than "forwarding" (e.g., when servicing involves intensive computation or large file transfers); The must-serve model is appropriate for applications with low service cost such as simple database lookups. In the must-serve model the traffic traversing each edge (link) is also bounded by C, which is important in some applications.

We analyze a static setting where the flow generated by each leaf remains fixed through time.[2] We consider both centralized and distributed algorithms. For distributed algorithms we assume that each node is able to communicate with its parent and children. In the distributed settings, priming decisions of different nodes can depend on each other, and we consider the "convergence" process until the nodes decide on a stable allocation. We measure the convergence time in units of the maximum "reaction time" of a node, that is, the maximum elapsed time until it reacts to a new input.

One of our distributed algorithms, GAP, assumes flow reporting, where each child reports to its parent on the flow generated below it (regardless of whether it is served or not). This reporting process is similar to "hit count" reporting in cache hierarchies.

3 What allocation size is necessary?

Let R be the total flow generated in the tree. An obvious lower bound on the required allocation size is R/C. This bound can be further refined if we also consider the maximum number of children of a node, which we denote by Δ. The following lemma shows a better bound for the must-serve model.

Lemma 3.1 *For any C and $\Delta > 4$, there are instances such that $\Delta R/(4C)$ primed nodes are necessary in the must-serve model.*

Proof: Consider a complete Δ-ary tree of depth d where each leaf generates $2C/\Delta$ flow. Therefore the total flow is $R = 2C\Delta^{d-1}$.

Consider a 2-level subtree of this tree consisting of a node x of depth $d-1$ and its Δ children which are leaves. At least $\lceil \Delta/2 \rceil$ of the children of x must be primed, since otherwise x receives

[2] The algorithms, however, are applicable in a dynamic context where flow changes.

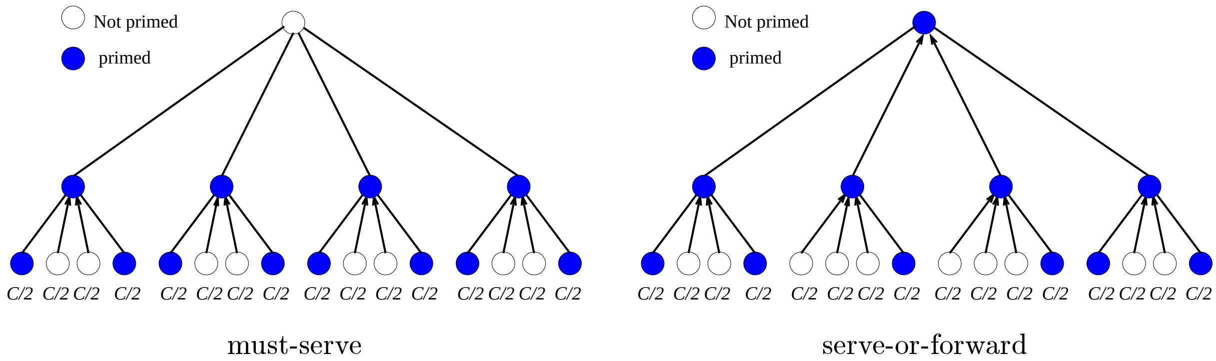

Figure 1: Example of optimal allocation for the two variants of the problem. The tree has 16 leaves each generating a flow of $C/2$, thus the total flow is 8C. Serve-or-forward primes 11 nodes and must-serve primes 12 nodes.

more than C units of flow. Furthermore, there is an optimal selection that primes exactly $\lceil \Delta/2 \rceil$ of the children of x. This is because we can always prime x instead of all but $\lceil \Delta/2 \rceil$ of its primed children without increasing the allocation size.

Let $s(x)$ be the nearest primed ancestor of x (Note that $s(x) = x$ if x is primed). Since the combined value of the flow of all the unprimed children of x strictly exceeds $C/2$ it follows that if $x \neq y$ are two nodes of the depth $d-1$ then $s(x) \neq s(y)$. (Otherwise $s(x)$ must serve more than C units of flow.) Thus, there is one primed internal node for each node x of depth $d-1$.

Since there are Δ^{d-1} nodes of depth $d-1$ we obtain that the size of an optimal solution of this instance is $\Delta^{d-1}(1 + \lceil \Delta/2 \rceil) \geq \frac{R\Delta}{4C}$. □

A similar bound for the serve-or-forward model is given by the following lemma.

Lemma 3.2 *For any C and $\Delta > 5$, there are instances such that $\Delta R/(5C)$ storage is necessary with serve-or-forward.*

Proof: Consider a complete Δ-ary tree of depth d where all leaf nodes generate $2C/\Delta$ flow and therefore the total flow is $R = 2C\Delta^{d-1}$, as in the proof of Lemma 3.1. The total number of internal nodes is $(\Delta^d - 1)/(\Delta - 1)$ and therefore the total amount of flow which these nodes can serve is $C(\Delta^d - 1)/(\Delta - 1)$. The remaining flow must be served, less efficiently, at primed leaves. The amount of flow remaining for leaves to serve is at least

$$R - C(\Delta^d - 1)/(\Delta - 1) \geq C\Delta^d \left(\frac{2}{\Delta} - \frac{1}{(\Delta - 1)} \right)$$

flow. Each leaf generates $2C/\Delta$ flow and hence this is the greatest amount of flow any leaf can serve. So we need to prime at least

$$\Delta^d \left(\frac{2}{\Delta} - \frac{1}{(\Delta - 1)} \right) \frac{\Delta}{2} \geq \frac{R\Delta}{5C}$$

leaf nodes, for $\Delta \geq 6$. $\square$

Remark: Note that the proof of Lemma 3.2 applies also to the must-serve model and gives a slightly worse constant than the proof of Lemma 3.1.

4 Optimal algorithms

In this section we present algorithms that find an optimal allocation in both the serve-or-forward and must-serve models. In Section 4.1 we present an optimal algorithm for the serve-or-forward model, and in Section 4.2 we present an optimal algorithm for the must-serve model.

4.1 Serve-or-forward

The algorithm OPT_SoF works in two phases. In the first phase it processes the nodes from the leaves upward, and makes some priming decisions. In the second phase additional nodes are primed throughout the tree. Each node primed in the first phase is assigned C units of flow that are generated below it. Nodes primed in the second phase may be assigned less than C units of flow.[3]

Our algorithm utilizes the notion of a *residual problem.* In each iteration of the first phase the algorithm primes a node, and allocates C units of flow for it to serve. Then it generates a residual instance by transferring children of the primed node v that have remaining unserved flow to v's parent. In the following we denote by $f(v)$ the flow into node v at the "current" residual problem.

Algorithm OPT_SoF for serve-or-forward:

Phase 1 While there is a node v other than the root such that $f(v) > C$ do the following. Let v be a deepest non-root node such that $f(v) > C$ and let $v_1, v_2, ...$ be the children of v ordered by increasing flow value, i.e. $f(v_1) \leq f(v_2) \leq \cdots$.[4] Let j be the smallest integer such that $\sum_{i \leq j} f(v_i) \geq C$. Prime the node v. Assign to v all the flow coming up from $v_1, \ldots, v_{j-1}$ and $C - \sum_{i \leq j-1} f(v_i)$ of the flow from leaf descendants of v_j (chosen arbitrarily). Make the children $v_{j+1}, v_{j+2}, \ldots$ of v with their respective flow values $f(v_{j+1}), f(v_{j+2}), \ldots$ children of the parent of v. Make v_j with the $f(v_j) - C + \sum_{i<j} f(v_i)$ units of flow generated in its subtree and not served by v, a child of the parent of v.[5]

Phase 2 Let v be the root and let $v_1, v_2, ...$ be the children of the root ordered by increasing flow value, i.e. $f(v_1) \leq f(v_2) \leq \cdots$. If $f(v) > C$ let j be the smallest integer such that

[3] Note that there could be several such mappings for each optimal allocation, including mappings where nodes assigned in the first phase obtain less than C units of flow. We consider particular assignments where this holds.

[4] If there is more than one node v which is deepest with $f(v) > C$ we break ties arbitrarily.

[5] In the residual problem we eliminate the flow served by v. Other than that the subtrees hanging off of the children of v, which are made children of the parent of v, remain the same.

$\sum_{i \leq j} f(v_i) \geq C$.

Prime the nodes $v_{j+1}, v_{j+2} \ldots$, and assigned each of them to serve all the flow generated in its subtree.

If $\sum_{i \leq j} f(v_i) > C$, prime also the node v_j and assign it to serve all the flow generated in its subtree.

If $f(v) > 0$ prime v and assign it to serve all remaining unserved flow in the tree.

We can implement OPT_SOF to run in $O(n \log n)$ time where n is the number of nodes in the tree. To do that we maintain the children of every node in a heap. When processing a node v we do a series of delete-mins on the heap of v until we have accumulated the lightest children whose weight is above C. We reinsert the last deleted child back into the heap with its adjusted flow value if the cumulative flow of the deleted children is greater than C. Finally we meld the heap of v with the heap of the parent of v.

A distributed implementation of OPT_SOF follows these two phases. The first "upward" phase, communication occurs from the leaves to the root with nodes primed along the way. In this phase each node sends to its parent the identity and the current flow value of each transferred child. In the second "downward" phase, the root "decides" to prime a subset of its children. Since some of these children are nodes that have been transferred up during the first phase, we then communicate these priming decisions to the nodes themselves. The amount of communication passed when children are transferred could be (in the worst case) of the order of the number of descendants of the transferring node.

We now establish the correctness and optimality of OPT_SOF.

Theorem 4.1 OPT_SOF *produce a correct allocation of minimum size.*

To establish correctness we have to verify that each primed node is assigned to serve no more than C units of flow and that all flow is served. By the definition of the algorithm it is clear that each node v which gets primed at the first phase is assigned to serve exactly C units of flow. When the first phase ends all children of the root receive less than C units of flow. Therefore having a subset of these children serve the flow they generate, and having the root serve the rest, produces a legal solution as long as the subset is large enough so that at most C units of flow are served at the root. The subset primed by the second phase clearly satisfies this requirement.

We now show that the allocation produced is of minimum size. Consider an iteration of the first phase. Let T be the network when the iteration starts, let v be the node which is primed during the iteration and let A be the flow assigned to v. Let T' be the residual problem generated from T at the end of the iteration. It is not hard to see that if T has an optimal allocation in which v is primed and serves A, then we can obtain an optimal solution to T from any optimal solution of T', by using the allocation of the optimal solution of T' together with priming v and assigning v to serve A. Thus by induction on the iterations of the first phase, it suffices to show that the choice of v and A can be extended to an optimal solution of T. As a base case of the induction we show that the solution of the residual problem obtained by the second phase is optimal. The optimality of the allocation of the second phase follows since we prime the children of the root with largest incoming flow. Therefore we prime as small subset of them as possible so that the root can serve the remaining flow.

It remains to show that T has an optimal solution in which v is primed and serves A. Consider any optimal solution O to T. We show that we can change it to another solution with the same number of primed nodes in which v serves A.

If v is not primed in O then we first change O so v is primed as follows. If v has a primed descendant then let v' be such a primed descendant where the nodes on the path from v to v' are not primed. Clearly if we prime v instead of v' and have v serve the flow previously served by v' we still have an optimal solution. In case v does not have a primed descendant, let u be the closest ancestor of v which is primed. Since $f(v) > C$ if u serves flow which is not generated below v then some ancestor of u serves flow generated below v. It follows that by interchanging flow assigned to u with flow assigned to its ancestors, we can change the flow assignment so that all the flow that u serves is generated below v. Finally since all the flow which u serves is generated below v we can prime v instead of u and have it serve the flow that u serves.

Last we have to show that we can change the allocation and the flow assignment so that v serves A. First notice that since the flow reached to each child of v is at most C we may assume that O does not contain prime descendants of v which are not immediate children of v. (We can prime a child of v instead of each such primed descendant of v without increasing the allocation.) Furthermore if v has a primed child v_1, and a child v_2 which is not primed, such that $f(v_2) > f(v_1)$, we can change O and prime v_2 instead of v_1. Ancestors of v that served the flow coming up from v_2 now assigned to serve the flow coming up from v_1 instead. Since less flow is coming up from v_1 this allocation is still legal and hence optimal. So we may assume that the primed children of v are those from which the largest amount of flow originates.

Finally, if v serves C units of flow but this flow is not identical to A we can interchange the flow assigned to v with the flow assigned to its ancestors so that v serves exactly the flow in A. If v serves less than C units of flow we can increase the flow assigned to it so that it serves exactly the flow in A and assign less flow to v's primed child that serves the smallest amount of flow among the children of v.

4.2 Must-serve

Must-serve instances have a simpler definition of the residual problem: Given an instance and a subset of primed nodes, the residual problem is obtained by eliminating all subtrees rooted at primed nodes along with the flow generated in them.

Unlike the case with Serve-or-forward instances, we do not need to consider flow assignments for obtaining the residual instance. The identity of the primed nodes suffices.

We now state the optimal must-serve algorithm, OPT_MS. The algorithm iteratively primes nodes; it considers the residual network after each priming step. Computationally, the algorithm performs one bottom-up pass, but information can propagate one level down from parents back to children during that pass.

Algorithm 1 **(**OPT_MS**)** *Let v be a deepest node with $f(v) > C$.*
Order the children of v by decreasing flow $v_1, v_2, \ldots$ with $f(v_1) \geq f(v_2) \cdots$.
Let j be the smallest integer such that $\sum_{i \leq j} f(v_i) \geq f(v) - C$.
Prime the nodes $v_1, \ldots, v_j$.
If there is no v with $f(v) > C$ and positive flow enters the root them prime the root.

The correctness and optimality analysis is similar (and somewhat simpler) than the one performed in the serve-or-forward model.

Lemma 4.1 *Algorithm* OPT_MS *is correct and finds a minimum-size must-serve allocation.*

Proof: Consider a deepest node v with $f(v) > C$. We claim that any solution with the heaviest child of v being unprimed can be transformed to a solution with the same number of primed nodes where the heaviest child of v is primed and with v serving or transferring at most the same amount of flow. The claim concludes the correctness proof since it implies that the repeated step of choosing a deepest node with $f(v) > C$, priming its heaviest child, and focusing on the residual problem, is consistent with an optimal solution.

We now prove the claim. Consider an optimal solution where the heaviest child of v is not primed. First observe that no node can receive flow value that exceeds C in a solution; the first primed ancestor that this flow arrives at will have to serve it all, which is a contradiction. Therefore, node v must have at least one primed proper descendant in its subtree. First we show that there is an optimal solution where one of the children of v is primed. Suppose no one of the children of v is primed. It follows that v must have a primed descendant u which is not a child of v. Then the child of v which is an ancestor of u can be primed instead of u. This child can serve all the flow that u was serving plus additional flow.

Suppose now that the heaviest child of v is not primed and another child u is primed. If we prime the heaviest child and unprime u then the total unserved flow reaching v can only decrease. $\square$

We next consider algorithms where nodes make local decisions on when to become primed, by simply observing incoming flow.

5 The THRESHOLD algorithm

The THRESHOLD(C, Δ) algorithm runs independently at each node. The only communication it receives is the amount of incoming flow. We also assume each node knows Δ, the indegree of its parent node (or some bound on it). THRESHOLD produces an allocation which is good for the must-serve model.

A copy of THRESHOLD running at a node v performs as follows:

Algorithm 2 (THRESHOLD(C, Δ)) *Let $f(v)$ be the flow into the node v with respect to the current allocation assuming each primed descendant of v serves all the flow it gets.*
The node v is primed if and only if

- $f(v) > C/\Delta$, *or*
- v *is the root and* $f(v) > 0$.

In our model nodes may take their decisions asynchronously. Let u be an ancestor of v such that there is no primed node which is a proper descendant of u and a proper ancestor of v in the current allocation. We assume that if node v decides to change its priming status from primed to unprimed or vice versa the flow into u changes instantaneously to reflect this change. We also assume that each node runs the algorithm above at least every t seconds. Therefore if the incoming flow into a node changes so that it crosses the threshold C/Δ, and it remains stable at its new value for at least t seconds, then the priming status of v would change.

In reality there may be a delay between the moment a node v changes its priming status until an ancestor u of v realizes that its incoming flow has changed. Furthermore, this delay may be proportional to the distance between u and v. We discuss at the end of this section how this may affect the performance of THRESHOLD.

Initially, the algorithm starts with some configuration of primed nodes (possibly no primed nodes). Since each node makes local decisions that depend on decisions made by the nodes below it, it may change its decision multiple times. We argue that eventually, a stable allocation is reached which is good for the must-serve model. In fact, THRESHOLD(C, Δ) converges to the same allocation generated by the following iterative centralized algorithm. (Assuming it starts with an empty allocation)

- Prime all nodes such that the flow generated below them exceeds C/Δ, and the flow generated below each of their children does not exceed C/Δ.
- Truncate the tree by removing these newly primed nodes along with all nodes and flow generated below them, and repeat this process.

Let N be the set containing each node v such that the total flow generated in the subtree of v exceeds C/Δ and the total flow generated below each child of v is no larger than C/Δ. The centralized algorithm above primes the nodes in N in its first iteration. We claim that THRESHOLD also primes N, and unprimes all descendants of nodes in N, after at most $2t$ seconds, starting with any allocation. Indeed, regardless of the initial allocation, descendants of nodes in N must see incoming flow of at most C/Δ. Thus, THRESHOLD cannot prime them, and if they started out being primed, then after at most t seconds THRESHOLD will unprime them. Once all descendants of nodes in N are unprimed, each node in N receives flow exceeding C/Δ so it will become primed in the following t seconds. Once the nodes in N are primed and all their descendants are unprimed, this state will prevail as long as the input flow does not change. By induction one can complete this argument to show that above the nodes of N, THRESHOLD also primes exactly the same nodes as the iterative algorithm above when applied to the truncated tree.

Observe that since each child of a node in N sends to its parent at most C/Δ flow, the total flow that each node in N receives is at most C. So when THRESHOLD primes a node in N, this node can serve all the flow generated below it and it transfers 0 flow to its parent. The following lemma follows from this equivalence between these two algorithms.

Lemma 5.1 THRESHOLD(C, Δ) *eventually reaches a correct stable allocation. The allocation produced is legal in the must-serve model and thus also applies to the serve-or-forward model. The final allocation is of size at most $R\Delta/C$ (since each primed node serves at least C/Δ flow.)*

We define the *convergence time* of THRESHOLD as the time it takes for it to reach the stable state. To bound the convergence time of THRESHOLD recall that we defined t to be the maximum time it takes for a node to respond to a change in its incoming flow, where by *responding* we mean a change of state from primed to unprime or vice versa. It is easy to prove by induction on i that after $i * t$ seconds all nodes of depth at least $d-(i-1)$ have reached their stable state. Indeed, after t second the nodes of depth d reach their final state, since their incoming flow does not depend on decisions made by other nodes and therefore cannot change. Once the nodes of depth d are stable, the incoming flow into nodes of depth $d-1$ will not change anymore. So after an additional t seconds nodes of depth $d-1$ reach their stable allocation as well. In general after $k * t$ seconds nodes of depth at least $d-(k-1)$ have reached their stable state so the flow into nodes of depth $d-k$ will not change anymore, and consequently these nodes of depth $d-k$ reach their stable state during the next t seconds.

How far can THRESHOLD be from the optimal solution? We provide examples where the worst-case ratio approaches Δ for both the serve-or-forward and must-serve models. We start with the serve-or-forward model. For an integer i, construct a binary tree of depth i; and then attach Δ children to each of the 2^i leaves of the binary tree. Now suppose that each of the $\Delta 2^i$ leaves of the modified tree generates flow of value $C/\Delta+\epsilon$ ($\epsilon < C/(\Delta 2^i)$). The THRESHOLD algorithm will prime all leaves, and result in an allocation of size $\Delta 2^i$. The algorithm OPT_SOF will prime all depth-i nodes and the root node of the tree, and thus will obtain an allocation of size 2^i+1. The ratio of the allocation size obtained by THRESHOLD to the optimal allocation of OPT_SOF approaches Δ as

i increases.

For the must-serve model consider a tree of depth one consisting of a node and Δ children. Among these Δ children $\Delta - 1$ generate $C/\Delta + \epsilon$ flow and the remaining child generates $C/\Delta - (\Delta - 1)\epsilon$ flow. The THRESHOLD algorithm primes all $\Delta - 1$ children and the root node, thus producing an allocation of size Δ. The optimal algorithm OPT_MS primes only the root. A family of examples with arbitrarily large values of total flow can be obtained by attaching Δ children generating flow as defined above to each of the leaves of a binary tree. For small enough ϵ, the THRESHOLD algorithm will prime $(\Delta - 1)2^i$ of the leaves and all 2^{i-1} nodes of depth $i - 1$. The algorithm OPT_MS will prime all depth-i nodes. The ratio between the size of the allocation of THRESHOLD and the size of the allocation of OPT_MS is $\Delta - 0.5$.

Observe that any algorithm that takes independent local decisions at each node would not be able to improve on this approximation ratio. Intuitively, since no information is available to a node on what its siblings send to its parent, the algorithm cannot allow sending more than C/Δ units of flow to the parent and at the same time guarantee that every node receives at most C units of flow.

5.1 Convergence of THRESHOLD

We learned that the worst-case final allocation obtained by THRESHOLD is within Δ of optimal. We now consider the "intermediate" allocations obtained during the convergence process.

Consider a tree that consists of a path of n nodes $v_1, \ldots, v_n$, (ordered from the bottom to the root); each path node has $\Delta - 1$ leaves attached to it. The $\Delta - 1$ leaves at each path node generate together $\epsilon C/\Delta$ flow. (To simplify the presentation we assume that $1/\epsilon$ is integral.) The total flow entering the tree is $R = n\epsilon C/\Delta$. Observe that THRESHOLD will never prime a leaf node, thus only the nodes on the path get primed.

Consider the following possible convergence scenario. (1) Initially, all nodes are not primed; thus the ith path node v_i sees $i\epsilon C/\Delta$ flow. (2) Every v_i for $i \geq 1/\epsilon$ receives more than C/Δ flow and primes itself. (3) Next, all primed nodes above $v_{1+1/\epsilon}$ see only $\epsilon C/\Delta$ flow and unprime themselves. (4) The deepest primed path node $v_{1/\epsilon}$ remains primed from now on, since all nodes below it remain unprimed. (5) The process continues on the new tree obtained after truncating the current tree at $v_{1/\epsilon}$. The new tree has the same structure with $n' = n - 1/\epsilon$ interior path nodes.

It is not hard to see that convergence takes $\epsilon n = R\Delta/C$ such "iterations." The average number of nodes that change state (from primed to unprimed or vice versa) in an iteration is $\Omega(n)$. The largest intermediate allocation occurs at the first iteration and is of size $n - 1/\epsilon = (R\Delta/C - 1)/\epsilon$. That is, there is a factor of $1/\epsilon$ between intermediate and final allocation sizes.

Interestingly, these undesirable properties are not merely an artifact of the "sharp" threshold exhibited by THRESHOLD: a similar convergence pattern with very large size of intermediate allocation and slow convergence time is possible even under a natural modification of THRESHOLD. In this modification we prime a node when the incoming flow exceeds C/Δ and unprime a node when its flow drops to L/Δ or below (for some $\epsilon C < L < C$). This variant of THRESHOLD would result in worse-quality final allocations (which can be C/L times the allocation size obtained by the original version of THRESHOLD.) The worst-case convergence patterns, however, are similar.

In our model we assumed that the flow incoming into any node v changes instantaneously to reflect any change in the allocation of the descendants of v. Clearly the scenario above where intermediate allocations are large used this property of the model. Such bad convergence patterns are less likely to happen in cases where by the time a node v figures out the value of its incoming flow, the allocation in its subtree is stable or close to stable. We can force this to happen if each node v starts out primed and unprimes itself only when the allocation in its subtree is final and the local decision of THRESHOLD at v is to be unprimed. Notice, however, that the bad convergence pattern of THRESHOLD may reoccur at any time where the input changes even if we manage to avoid it at initialization time. Furthermore, for some applications priming a large number of nodes, even temporarily may be undesirable.

Another undesirable property of THRESHOLD occurs when the flow incoming into the leaves changes and THRESHOLD adjusts its allocation to the new flow. It is not hard to see that relatively small changes in the input may trigger large changes in the allocation. In particular a node v may change its allocation even if the total flow generated in its subtree has not changed. Consider for example a node v with three children v_1, v_2, and v_3 that are leaves. Assume v_1 generates 2ϵ flow, v_2 generates $\frac{C}{\Delta} - \epsilon$ flow, and v_3 generates $\frac{C}{\Delta} + \epsilon$ flow. THRESHOLD will prime v_3 and v. Assume now that the flow changes slightly so that v_1 generates 0 flow, v_2 generates $\frac{C}{\Delta} + \epsilon$, and v_3 still generates $\frac{C}{\Delta} + \epsilon$ flow. Then, THRESHOLD primes v_2 and unprimes v, even though the total flow generated below v has not changed.

To summarize, we showed that the simple local THRESHOLD algorithm can have certain convergence and instability patterns that may be considered as serious drawbacks for some applications. In the Section 6 we propose and analyze a different local algorithm, called GAP, that overcomes some of problems of THRESHOLD.

5.2 Parent Priming

By the definition of THRESHOLD, no node can send to its parent more than C/Δ units of flow in the final allocation. This requirement is necessary to guarantee that no node serves more than C units of flow in situations where a node has no information on what its siblings send to the parent. As a result THRESHOLD may converge to an allocation of size $\frac{R\Delta}{C}$. Indeed we showed in Section 3 that for some trees and flows an allocation of size about $\frac{R\Delta}{C}$ may be necessary, but for other inputs THRESHOLD may produce an allocation that is off from the optimal by a factor of Δ. In this section we suggest a technique that improves this worst-case approximation ratio of Δ by a constant factor and may perform even better in practice.

Consider running THRESHOLD using the bound $C' = \Delta C$ rather than C. This, of course, can result in a primed node serving up to ΔC units of flow. To solve this problem, and turn the solution to a valid one, we allow a correcting *parent priming* phase, where nodes that were primed in the first phase are allowed to prime some of their children. Specifically, a node that ends up with more than C units of flow (that is, flow in $(C, \Delta C]$ primes its children, in decreasing order of flow value, until its own flow drops to C or fewer units.

It is easy to see that this additional local phase will always result in a correct allocation; since before the parent priming phase each unprimed child was delivering at most C units of flow. The next lemma upper bounds the approximation ratio of an allocation produced using the parent priming technique with respect to the optimal allocation.

Lemma 5.2 *The approximation ratio with parent priming is* $(\Delta + 2)^2/(4\Delta)$.

Proof: Every node that is primed in the first phase serves at least C (and at most ΔC) units of flow. Consider a node primed in the first phase that serves $\alpha C \geq C$ units of flow. An optimal algorithm must use at least α primed nodes to serve the flow. Our algorithm primes at most $\lceil \Delta(\alpha - 1)/\alpha \rceil$ children and thus uses at most $2 + \Delta(\alpha - 1)/\alpha$ primed nodes to serve the flow. The ratio is thus $(2 + \Delta)/\alpha - \Delta/\alpha^2$. The maximum on the interval $\alpha \in [1, \Delta]$ is obtained at $\alpha = (2\Delta)/(\Delta + 2)$ and is $(\Delta + 2)^2/(4\Delta)$. □

6 The GAP algorithm

The GAP algorithm runs locally at each node. Like THRESHOLD, GAP makes its decisions locally. However in contrast to THRESHOLD, decisions made by each node are independent of the state of other nodes. The only information GAP relies on at each node is the total amount of flow generated below each of its children (regardless of where it is served). Note that if nodes are running THRESHOLD and all start unprimed then the initial value of the incoming flow to each node is the total flow generated in its subtree. In practice, nodes can propagate this information bottom-up on the distribution tree by periodic reports from children to parents.

Let $F(v)$ be the total flow generated under a node v and let $F_s(v)$ be the flow generated under its heaviest child, that is $F_s(v) = \max\{F(u) \mid u \in \text{children}(v)\}$. If v is a leaf then we define $F_s(v) = 0$.[6]

We consider GAP in the must-serve model. GAP is a randomized algorithm which depends on two parameters. The first is C, which is also an upper bound on the maximum flow generated by a leaf. In previous sections we used C as an upper bound on the flow a primed node is allowed to serve. Here, since GAP is randomized, we will allow some nodes to serve more than C units of flow.

[6] We use here $F(v)$ to distinguish it from $f(v)$ which we used in the previous section to denote the flow entering node v.

The second parameter of GAP is a fraction $0 \leq \mu \leq 1$. This fraction controls the average number of primed nodes, which increases as μ decreases. As μ decreases the probability that a node will serve more than C units of flow also decreases.

Specifically, μ determines the expected amount of flow served by a primed node. However whatever this expectation is, the probability that we deviate from it decreases exponentially. For example, if $\mu = \frac{1}{\Delta}$ then the expected amount of flow that a node will serve is no greater than C, and the probability that a node will have to server more than $L * C$ units of flow decreases exponentially with L.

Algorithm 3 (GAP(C, μ) algorithm) *At each node v, GAP(C, μ) performs the following.*

1. *If $F(v) < \mu C$, then v is not primed.*
2. *If $F(v) \geq \mu C$ then*
   - *if $F_s(v) < \mu C$, v is primed.*
   - *otherwise, v is primed with probability* $\min\{1, (F(v) - F_s(v))/(\mu C)\}$.
3. *If v is the root, it did not get primed in 2, and it has flow entering it, then prime v.*

In our subsequent analysis, we assume that only leaf nodes have flow below μC. We do that without loss of generality since the operation of GAP on an instance is equivalent to GAP operating on the same instance when maximal subtrees rooted at a node v with $F(v) < \mu C$ are contracted to a single leaf with flow $F(v)$.

We first bound the expected size of the allocation produced by GAP.

Lemma 6.1 *The expected number of nodes primed by GAP at step 2 is at most $2R/(\mu C) - 1$, for $R \geq \mu C$. (The actual expected number of primed nodes is at most one larger because of priming the root, and it is one if $R < \mu C$)*

Proof: Let $M(x)$ be the maximum, over flow trees with flow x, of the expected number of primed nodes allocated by GAP in step 2. By the definition of GAP, $M(x)$ satisfies the following. If $x < \mu C$ then $M(x) = 0$. For $x \geq \mu C$ we have that

$$\begin{aligned} M(x) &= \max(\{\min\{1, (x - S)/(\mu C)\} + M(S) \mid S < x\} \\ &\quad \cup \{1 + M(S_1) + \cdots + M(S_i) \mid \sum_{j=1}^{i} S_j \leq x \text{ and } \forall j\ S_j \geq \mu C \text{ and } i \geq 2\}) \end{aligned} \tag{1}$$

The first set over which we take the maximum takes into account all trees where all children of the root except possibly the largest one get less than μC units of flows. In this case there cannot be primed descendants of any child of the root except the largest one. The second set takes into account trees where more than one child of the root gets at least μC units of flow. In this case by the definition of GAP we always mark the root at step 2.

Using induction on the flow value x, and relation (1), it is easy to establish that $M(x) \leq 2x/(\mu C) - 1$. $\square$

Since GAP is a randomized algorithm, there is a positive probability that nodes are assigned flow that exceeds C. The following lemma gives probabilistic bounds on the maximum flow entering a node when we apply GAP. Notice that since each leaf generates at most C units of flow, it is clear that a primed leaf never serves more than C units of flow. So the lemma only considers the maximum flow entering a prime internal node.

Lemma 6.2 *The expected amount of flow serviced by a primed internal node is at most $\Delta\mu C$. Furthermore, the probability that the serviced flow exceeds $L\Delta\mu C$ is at most $(1 - 1/e)^L$.*

Proof: To simplify the calculations, we normalize the flow to be in units of μC. In these units we first have to show that the expected amount of flow serviced by a primed node is at most Δ.

Consider an internal node v_0. Since v_0 is internal $F(v_0) \geq 1$. Consider a child v_1 of v_0. We claim that the expected value of flow that v_0 receives from v_1 is at most 1. Thus the total expected flow passed to v_0 from all its children together is at most Δ.

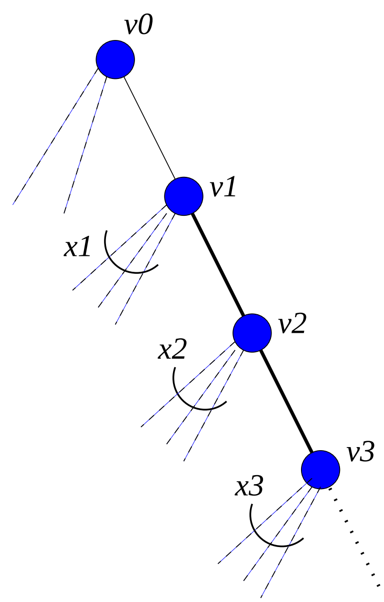

Figure 2: Illustration for the proof of Lemma 6.2. Thick lines represent edges to heaviest child.

We now prove the claim. We can assume without loss of generality that $F(v_1) > 1$ (since otherwise the claim trivially follows). We may also assume that v_1 is not a leaf, since if it was then it is primed and therefore does not contribute any flow to v_0. Let v_2 the child of v_1 with the largest amount of flow generated below it, and in general let v_{i+1} be the child of v_i with the largest amount of flow generated below it, if v_i is not a leaf. We call $v_1, v_2, \ldots, v_\ell$ the *heavy path* hanging from v_1. (see Figure 2.) Note that v_ℓ is a leaf. Let $x_i = F(v_i) - F_s(v_i) = F(v_i) - F(v_{i+1})$ $(i \geq 1)$ be the flow getting into v_i from all its children but v_{i+1}. We define a node on the *heavy path* hanging from v_1 to be *always-primed* if one of the following conditions holds.

1. $x_i \geq 1$. I.e. the flow that enters v_i from children other than v_{i+1} is at least 1.
2. $F(v_i) \geq 1$ and $i = \ell$. I.e. v_i is the leaf at the end of the path and it generates at least one unit of flow.
3. $F(v_i) \geq 1$, $i < \ell$, and $F(v_{i+1}) < 1$. Note that this case can happen only if $i = \ell - 1$ since we assume that only leaves can have $F(v) < 1$.

Note that by the definition of GAP an *always-primed* node is primed with probability one. Furthermore, either v_ℓ or $v_{\ell-1}$ must be always-primed, so in particular we have at least one always-primed node on the path. If v_1 is always-primed then the claim obviously holds, otherwise, let $k \geq 1$ be the maximum such that v_k is not always-primed. Observe that we must have that $x_i < 1$ for $1 \leq i \leq k$.

Consider a run of GAP and let $p \leq k$ be the lowest index such that v_p is primed; define $p = k+1$ if there is no primed node before v_{k+1} on the path. By definition, the maximum amount of flow that v_0 gets from v_1 is $\sum_{j \leq k+1} x_j$. To estimate the expected amount of flow that v_0 gets from v_1 we define G_i, $1 \leq i \leq k$, to be a random variable describing the flow value passed to v_{i-1} from v_i. We also denote the expectation of G_i by g_i. Since GAP leaves node v_i unmarked with probability $1 - x_i$, we have that $g_k = (1 - x_k)x_k$, and for $i < k$

$$g_i = (1 - x_i)(x_i + g_{i+1}) \quad .$$

So in particular

$$g_1 = (1 - x_1)(x_1 + (1 - x_2)(x_2 + (1 - x_3)(\cdots + (1 - x_{k-1})(x_{k-1} + (1 - x_k)x_k)\cdots)) \ .$$

We show by a backward induction on i that $g_i \leq 1$. It is easy to see that $g_k = (1 - x_k)x_k \leq 1$. Assume inductively that $g_i \leq 1$, then $g_{i-1} = (1 - x_i)(x_i + g_i) \leq (1 - x_i)(x_i + 1) \leq 1$. For $i = 1$ we obtain that $g_1 \leq 1$ and the claim follows.

We now prove the second part of the lemma. We start with the following claim.

Claim 6.1 *The probability that the value of G_i exceeds L is at most $(1 - 1/e)^L$, for all $L \geq 0$.*

We prove the claim by downward induction on i. The base of the induction is the random variable G_k. The outcome is x_k with probability $1 - x_k$ and 0 otherwise. The likelihood that the outcome exceeds $L > x_k$ is 0. The likelihood that it exceeds $L \leq x_k$ is $(1 - x_k) \leq (1 - 1/e)^{x_k} \leq (1 - 1/e)^L$. (we use the tautology $(1 - a) \leq (1 - 1/e)^a$ for $0 \leq a$.)

For the random variable G_i, and $L \leq x_i$ we have

$$P[G_i \geq L] = (1 - x_i) \leq (1 - 1/e)^{x_i} \leq (1 - 1/e)^L.$$

For $L > x_i$ we have the relation

$$P[G_i \geq L] = (1 - x_i) P[G_{i+1} \geq L - x_i] \ .$$

From the induction hypothesis we have $P[G_{i+1} \geq L - x_i] \leq (1 - 1/e)^{L - x_i}$. Substituting the above we obtain that

$$P[G_i \geq L] \leq (1 - x_i)(1 - 1/e)^{L - x_i} \leq (1 - 1/e)^L \ .$$

(The last inequality follows again from the inequality $(1 - a) \leq (1 - 1/e)^a$ for $(0 < a < 1)$.) This completes the proof of the claim.

Claim 6.1 shows that the probability that one child passes more than L units of flow to its parent is at most $(1 - 1/e)^L$. To complete the proof of the lemma we need to show that when a node has Δ children then the probability that it receives ΔL units of flow from all of them is still at most $(1 - 1/e)^L$. This intuitively should hold since priming decisions at nodes which are unrelated (neither is a descendant of another) are independent. The following claim establishes this fact and finishes the proof of the lemma.

Claim 6.2 *Let $Y_1, \ldots, Y_\Delta$ be independent random variables such that $P(Y_i \geq L) \leq (1 - 1/e)^L$ for all $L \geq 0$. Let $Z = Y_1 + Y_2 + \cdots + Y_\Delta$. Then $P(Z \geq \Delta L) \leq (1 - 1/e)^L$ for all $L \geq 0$.*

We prove Claim 6.2 using the notion of stochastic domination between random variables. We say that a random variable Z *stochastically dominates* a random variable X if $P(X \geq a) \leq P(Z \geq a)$ for every a. It is known (see e.g. [17, page 52]) that if $Z_1, \ldots, Z_k$ are independent, and $X_1, \ldots, X_k$ are independent, and Z_i stochastically dominates X_i for every $1 \leq i \leq k$, then $\sum_{i=1}^k Z_i$ stochastically dominates $\sum_{i=1}^k X_i$. For completeness we prove this here.

Since Z_k stochastically dominates X_k we obtain that

$$\begin{aligned}
P(\sum_{i=1}^{k} X_i \geq a) \ &= \\
&= \int_{x_1=0}^{\infty} \int_{x_2=0}^{\infty} \cdots \int_{x_{k-1}=0}^{\infty} P(X_1 = x_1) \cdots P(X_{k-1} = x_{k-1})) P(X_k \geq a - x_1 - \cdots x_{k-1}) \\
&\leq \int_{x_1=0}^{\infty} \int_{x_2=0}^{\infty} \cdots \int_{x_{k-1}=0}^{\infty} P(X_1 = x_1) \cdots P(X_{k-1} = x_{k-1})) P(Z_k \geq a - x_1 - \cdots x_{k-1}) \\
&= P(\sum_{i=1}^{k-1} X_i + Z_k \geq a) \ .
\end{aligned}$$

We can now apply the same argument to the variable $X_1, \ldots X_{k-1}$ and Z_k and show that

$$P(\sum_{i=1}^{k-1} X_i + Z_k \geq a) \leq P(\sum_{i=1}^{k-2} X_i + Z_{k-1} + Z_k \geq a) \ .$$

For that we expand $P(\sum_{i=1}^{k-1} X_i + Z_k \geq a)$ as a multiple integral whose variables are the values of $X_1, \ldots, X_{k-2}$, and Z_k, and use the fact that the probability of a tail of X_{k-1} is dominated by the probability of the same tail of Z_{k-1}. By repeating this argument k times we finally obtain that

$$P(\sum_{i=1}^{k} X_i \geq a) \leq P(\sum_{i=1}^{k} Z_i \geq a) \ .$$

To prove Claim 6.2, let D be an exponential random variable whose cumulative distribution function is $F(x) = P(D \leq x) = 1 - e^{-\lambda x}$, for $x \geq 0$, where $\lambda = -\ln(1 - 1/e)$. Notice that for every $L \geq 0$, $P(Y_i \geq L) \leq P(D \geq L)$ so D stochastically dominates Y_i for every $1 \leq i \leq \Delta$. (for every $a < 0$, $P(Y_i \geq a) = P(D \geq a) = 0$.) It follows that if $D_1, \ldots, D_\Delta$ are Δ independent copies of D then $\sum_{i=1}^{\Delta} D_i$ stochastically dominates $\sum_{i=1}^{\Delta} Y_i$. So

$$P(\sum_{i=1}^{\Delta} Y_i \geq \Delta L) \leq P(\sum_{i=1}^{\Delta} D_i \geq \Delta L) \ .$$

To finish the proof we observe [6] that the distribution of the sum of Δ exponential random variables with parameter λ is $Gamma(\Delta, \lambda)$ whose density is

$$g(x) = \frac{\lambda e^{(-\lambda x)} (\lambda x)^{(\Delta - 1)}}{(\Delta - 1)!}$$

Next we show that $g(a\Delta) \leq f(a)$, for every $a \geq 0$, where $f(x) = \lambda e^{-\lambda x}$ is the density function of the exponential random variable D. Then the lemma follows since $P(\sum_{i=1}^{\Delta} D_i > \Delta L) \leq P(D > L)$.

To see that $g(a\Delta) \leq f(a)$ we substitute the two densities and obtain that we have to show that

$$\frac{\lambda e^{(-\lambda a \Delta)} (\lambda a \Delta)^{(\Delta - 1)}}{(\Delta - 1)!} \leq \lambda e^{-\lambda a} \ .$$

Dividing both sides by $\lambda e^{-\lambda a} e^{(-\lambda a \Delta)}$ we obtain the equivalent inequality

$$\frac{e^{(\lambda a)} (\lambda a \Delta)^{(\Delta - 1)}}{(\Delta - 1)!} \leq e^{\lambda a \Delta} \ . \tag{2}$$

Using the Taylor expansion of the exponential function we obtain that Equation (2) is equivalent to

$$\left(1 + \lambda a + \frac{(\lambda a)^2}{2!} + \cdots\right) \frac{(\lambda a \Delta)^{(\Delta - 1)}}{(\Delta - 1)!} \leq \left(1 + \lambda a \Delta + \frac{(\lambda a \Delta)^2}{2!} + \cdots\right) \ . \tag{3}$$

Since

$$\frac{1}{k!} \leq \frac{\Delta^k}{\Delta(\Delta + 1)(\Delta + 2) \cdots (\Delta + (k - 1))}$$

we obtain that Equation (3) holds since the left hand side is term by term smaller than the tail of the series at the right hand side starting from the term whose index is $\Delta - 1$. □

The analysis of GAP implies that if we set $\mu = \frac{1}{c\Delta \log(R\Delta/C)}$, for some small constant c, then there is a constant probability that the load served by any primed node is at most C. We next show how we can use $\mu = 1/(\Delta + 1)$ together with iterative application of GAP in order to obtain better trade-off.

6.1 Applying GAP iteratively

GAP can be applied iteratively as follows. We apply the first iteration of GAP as described before. After the iteration ends if we chop off primed nodes from their parents we get a forest. We apply the next iteration only to trees of the forest where the root serves more than C units of flow. When an iteration ends we further partition the forest by chopping off primed nodes, and apply the next iteration only to trees where the root serves more than C units of flow. The algorithm terminates when all primed nodes serve less than C units of flow. We will show that for an appropriate choice of μ the algorithm indeed terminates and we will analyze what is the expected number of iterations until it stops.

We can apply this algorithm distributively by passing messages between nodes as follows. For each unprimed node v we define $F_i(v)$ to be the amount flow generated in the subtree rooted by v at iteration i. For a node v which was primed at iteration $j < i$ we define $F_i(v) = 0$. Each unprimed node v locally waits until it knows $F_i(v)$ for increasing values of i. When v knows $F_i(v)$ it runs GAP locally and decides whether it gets primed at iteration i.

Each node v can calculate $F_i(v)$ if nodes report their $F_i()$ values to their parents. If v was primed at iteration $j < i$ then no flow travels from v to its parent at iteration i. So in this case v reports to its parent that $F_i(v)$ is 0. Otherwise v waits for its children to report their $F_i()$ values. Once v gets all these values then $F_i(v)$ is $\sum F_i(u)$ where the sum is over all children u of v. Node v then reports $F_i(v)$ to its parent.

When we apply this distributed implementation then unprimed nodes continue to apply GAP in subsequent iterations although they may already have a primed ancestor which serves less than C units of flow. Therefore to make our distributed algorithm complete, each node v which is primed at iteration i, and $F_{i+1}(v) \leq C$ sends a message downwards to its descendants announcing that they may stop running GAP. A node v that receives a message that it has an ancestor that got primed at iteration i and serves no more than C does the followings.

1. Node v propagates the message further down to its descendants.

2. If v got primed at iteration j and $i < j$ then v unprimes itself.

3. Node v does not have to run GAP, or propagate $F_j(v)$ in any iteration $i < j$.

Our key lemma below shows that if we apply GAP with $\mu = \frac{1}{\Delta+1}$ then on average a constant fraction of the flow is served by primed nodes that serve less than C units of flow. This lemma implies the following theorem about the performance of iterative GAP.

Theorem 6.1 *Iterative GAP with $\mu = \frac{1}{\Delta+1}$ stops after $O(\log(R/C))$ iterations with probability at least $1 - (\frac{C}{R})^c$ where c is some constant. The expected size of the allocation that iterative GAP produces is $O(R\Delta/C)$.*

Proof: By Lemma 6.3 at each iteration of GAP, the probability that a unit flow is served by a primed node which serves less than C units of flow is at least $p = 1/e^2$. Each such unit of flow does not participate in subsequent iterations. So the probability that a unit of flow participates at iteration $i+1$ is at most $(1-p)^i$. For $i = \frac{1}{p} a \ln(R/C)$ this probability is at most $(\frac{C}{R})^a$. Let X be a random variable which equals to the expected amount of flow participating at iteration $i+1$. Multiplying our bound on the probability that a unit of flow participates at iteration $i+1$ by the total amount of flow we obtain that $E(X) = R * (\frac{C}{R})^a = C(\frac{C}{R})^{a-1}$. By Markov's inequality the probability that the amount of flow that participates at iteration $i+1$ exceeds C is bounded by

$$Prob(X \geq C) \leq \frac{E(x)}{C} = \frac{C(\frac{C}{R})^{a-1}}{C} = \left(\frac{C}{R}\right)^{a-1} .$$

Clearly if the total flow which participates at iteration $i+1$ is not larger than C then iteration $i+1$ is the last. So we obtained that for $i = \frac{1}{p} a \ln(R/C)$, iteration $i+1$ is the last, with probability at least $1 - \left(\frac{C}{R}\right)^{a-1}$. If we define $c = a - 1$ the first part of the theorem follows.

The expected number of primed nodes in each iteration is bounded by $(\Delta+1)/C$ times the expected flow value at that iteration. Since the expected flow value decreases geometrically by a factor of at least $1-p$ from iteration to the next, we obtain an allocation of expected size $O(R\Delta/C)$.
□

We now prove our key lemma saying that a unit of flow does not go through to the next iteration of GAP with some constant probability.

Lemma 6.3 *Consider a unit of flow generated at a leaf. When nodes are primed according to GAP, then with probability at least $1/e^2$ the primed node serving the flow (the deepest primed node on the path from the leaf to the root) serves a total of at most $(\Delta+1)\mu C$ units of flow.*

Proof: In the proof of Lemma 6.2 (using Claim 6.2) we proved that the probability that the Δ children of a node x transfer more than $\Delta L \mu C$ units of flow to x is at most $(1-1/e)^L$. It is easy to see that the same argument in fact shows that any set of M independent nodes (that is a set of nodes such that neither is an ancestor of the other but they are not necessarily siblings) transfer to their parents $ML\mu C$ units of flow with probability at most $(1-1/e)^L$. We shall repeatedly invoke this observation in the proof.

Consider a unit of flow generated at a leaf and its path to the root. Let v be the deepest node on the path with $F(v) \geq \mu C$. We split the rest of the proof into the following cases.

Case 1: For every child u of v, $F(u) < \mu C$.

In this case v is primed and therefore serves our flow-unit. By the observation above the probability that v receives more than $\Delta\mu C$ flow from its Δ children is at most $1 - 1/e$.

Case 2: Node v has a child u, such that $F(u) \geq \mu C$.

Let w be the heaviest child of v, and let v_0 be the closest ancestor of v which is not a heaviest child of its parent, or the root in case every ancestor of v is a heaviest child of its parent. (Note that $v_0 = v$ in case v is not the heaviest child of its parent.) Denote by $v_0, v_1, \ldots, v_i = w$ the nodes on the path from v_0 to w. Note that $v_{i-1} = v$. Let x_j, $0 \leq j < i$, be the flow generated at the children of v_j other than v_{j+1}. (See Figure 2.)

Subcase 2a: $\sum_{j=0}^{i-1} x_j \leq \mu C$.

Assume first that v_0 is not the root and let $p(v_0)$ be the parent of v_0. Since $F(v_0) \geq F(v_i) \geq \mu C$ and v_0 is not the heaviest child of $p(v_0)$ we obtain that $F(p(v_0)) - F_s(p(v_0)) \geq \mu C$ so $p(v_0)$ is primed by the definition of GAP. Therefore our unit flow is either served by $p(v_0)$ or by some v_j where $0 \leq j \leq i-1$.

If our unit flow is served by v_j where $0 \leq j \leq i-1$ then the probability that v_j serves more than $\Delta\mu C$ units of flow is at most the probability that v_i passes up more than $(\Delta - 1)\mu C$ units of flow. That happens with probability at most $(1 - 1/e)^{\Delta - 1}$ by Claim 6.1.

If our unit flow is served by $p(v_0)$ then the probability that $p(v_0)$ serves more than $(\Delta + 1)\mu C$ units of flow is at most the probability that its $\Delta - 1$ children other than v_0, and the node $v_i = w$ transfer to their parents more than $\Delta\mu C$ units of flow. By the observation above this probability is at most $1 - 1/e$.

If v_0 is the root then clearly our unit of flow is served by v_j for some $0 \leq j \leq i-1$. The probability that v_j serves more than $\Delta\mu C$ units of flow is at most the probability that v_i passes up more than $(\Delta - 1)\mu C$ units of flow. That happens with probability at most $(1 - 1/e)^{\Delta - 1}$.

Subcase 2b: $\sum_{j=0}^{i-1} x_j > \mu C$.

Let k be the maximum such that $\sum_{j=k}^{i-1} x_j > \mu C$. If v_k is the root or $x_k > \mu C$ then clearly one of $v_k, \ldots, v_{i-1}$ serves our unit flow. Otherwise since by Claim 6.1 node v_{k-1}, the parent of v_k, receives from v_k more than μC flow with probability at most $1 - 1/e$, it follows that with probability at least $1/e$ one of $v_k, \ldots, v_{i-1}$ is primed.

Let v_j be the deepest primed node among $v_k, \ldots, v_{i-1}$ given that one of these nodes is primed. The probability that v_j serves more than $(\Delta + 1)\mu C$ units of flow is at most the probability that its $\Delta - 1$ children other than v_{j+1}, and v_i transfer to their parents more than $\Delta\mu C$ units of flow. By the observation above this probability is at most $1 - 1/e$.

To summarize we obtained that with probability at least $1/e$ one of $v_k, \ldots, v_{i-1}$ is primed, and assuming that one of $v_k, \ldots, v_{i-1}$ is primed, the deepest node among them which is primed served less than $(\Delta + 1)\mu C$ units of flow with probability at least $1/e$.

$\square$

Acknowledgment

We would like to thank the referees for many valuable comments that helped us make the presentation much clearer.

References

[1] B. Awerbuch, Y. Bartal, and A. Fiat. Distributed paging for general networks. *J. Algorithms*, 28(1):67–104, 1998.

[2] Clip2.com. The gnutella protocol specification v0.4. In *http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf*, 2000.

[3] E. Cohen and H. Kaplan. Aging through cascaded caches: performance issues in the distribution of Web content. In *Proceedings of the ACM SIGCOMM Conference*. ACM, 2001.

[4] E. Cohen and H. Kaplan. Balanced-replication algorithms for distribution trees. In *Proceedings of the 10th Annual European Symposium*, volume 2461 of *Lecture Notes in Computer Science*, pages 297–309, 2002.

[5] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[6] W. Feller. *An Introduction to Probability Theory and Its Applications, volume 2.* Wiley, 1971.

[7] S. Gadde, J. Chase, and M. Rabinovich. A taste of crispy Squid. In *Proceedings of the Workshop on Internet Server Performance (WISP'98)*, 1998.

[8] Napster Inc. The Napster homepage. In *http://www.napster.com/*, 2001.

[9] D. Karger, Lehman E., T. Leighton, M. Levine, D. Lewin, and R. Panigraphy. 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*, pages 654–663. ACM, 1997.

[10] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *Proceeding of the eighth international conference on World Wide Web*, pages 1203–1213. Elsevier North-Holland, Inc., 1999.

[11] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. *J. Algorithms*, 38(1):260–302, 2001.

[12] B. Li, M. Golin, G. Italiano, X. Deng, and K. Sohraby. On the optimal placement of Web proxies in the Internet. In *Proceedings of the IEEE Infocom*, pages 1282–1290, 1999.

[13] R. Malpani, J. Lorch, and D. Berger. Making world wide web caching servers cooperate. In *Proceedings of the Fourth International World Wide Web Conference*, December 1995.

[14] C. Plaxton, R. Rajaram, and A.W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory Comput. Syst.*, 32(3):241–280, 1999.

[15] S. Ratnassamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM'01 Conference*, 2001.

[16] S. Saroiu, K. P. Gummadi, R. Dunn, S. D. Gribble, and H. M. Levy. An analysis of Internet content delivery systems. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.

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

[18] I. Stoica, R. Morris, D. Karger, and H. Frans Kaashoek, M. ad Balakrishnana. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM'01 Conference*, 2001.