Balanced-Replication Algorithms for Distribution Trees*

Edith Cohen
AT&T Labs–Research
Florham Park
NJ 07932 USA

Haim Kaplan
School of Computer Science
Tel-Aviv University
Tel-Aviv 69978, Israel

August 1, 2002

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 analyse 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 receive, there is a preferred neighbor that brings the request "closer" to its destination. Therefore,

*A preliminary version of this paper will appear in the Proceedings of the 10th European Symposium on Algorithms (ESA'2002).

the forwarding pattern of each request forms a tree, with edges directed towards the root. We refer to such a structure as a *distribution tree.*

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 (possibly active caches) "near" Internet routers and augment them with the capability to intercept and forward HTTP traffic; such a service can shorten latency and network load. The growing popularity of Peer-to-Peer applications (e.g., [3, 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 [9] (which is based on consistent hashing [4]) and CAN [8] ("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 [8, 9] 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 can be caused by flash-crowd events or objects with persistent high demand. We would like to iterate that generally, each distribution tree corresponds to a single "item," and thus one might worry about interaction of flows from different trees and that "priming" actions performed at different trees will not balance out across nodes). 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 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 also would like to be as small as possible. Other desirable properties pertain to the complexity and communication requirements of the algorithm; and the stability of the allocation while the algorithm converges.

We first relate the worst-case allocation size needed to satisfy a flow of certain size. We provide optimal algorithms which minimize the size of the allocation, given a bound on maximum load a node can process. The algorithms are distributed and requires linear number of messages (two rounds of passing messages up and down the tree).

We then consider a simple natural algorithm proposed with both Chord and CAN: The THRESHOLD algorithm runs locally at each node and primes the node when the incoming flow of requests is above a certain threshold. The THRESHOLD algorithm has obvious advantages as it requires no additional communication as decisions are made only according to the amount of incoming flow. We provide bounds on the size of the allocation obtained and show that it is close to the worst-case upper bound. We show, however, that since priming decisions depend on decisions made by other nodes, THRESHOLD can suffer from very spiky convergence where interim allocation size can be considerably higher than final size.

Finally, we propose a different local algorithm GAP. Like THRESHOLD, GAP also makes its decision based on the amount of incoming flow and also meets the same upper bound. In contrast to THRESHOLD, it considers only the total flow generated below the deciding node and thus priming decisions are independent of decision made at other nodes. It follows that GAP obtains right away a stable allocation.

Related Work

Plaxton, Rajaraman, and Richa proposed a related scheme that also takes into account the underlying topology when overlaying the distribution network [7]. 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, 5]). 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 Golin et al, which provided a quadratic dynamic programming algorithm [6]. The different objectives functions we use versus the one of [6] 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 leafs and flowing towards the root. We suggest algorithms that would *prime* some nodes in the tree. A prime node can serve flow coming into it while a nonprime 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 $f : v \to R_+$ 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 can serve 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 leafs 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 objective of minimizing the number of primed nodes models a wide range of applications where being primed consumes resources (e.g., storage space).

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. The must-serve model also bounds by the traffic traversing each edge (link), which is important in some applications.

We analyse a static setting where the flow generated by each leaf remains fixed through time.[1] 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?

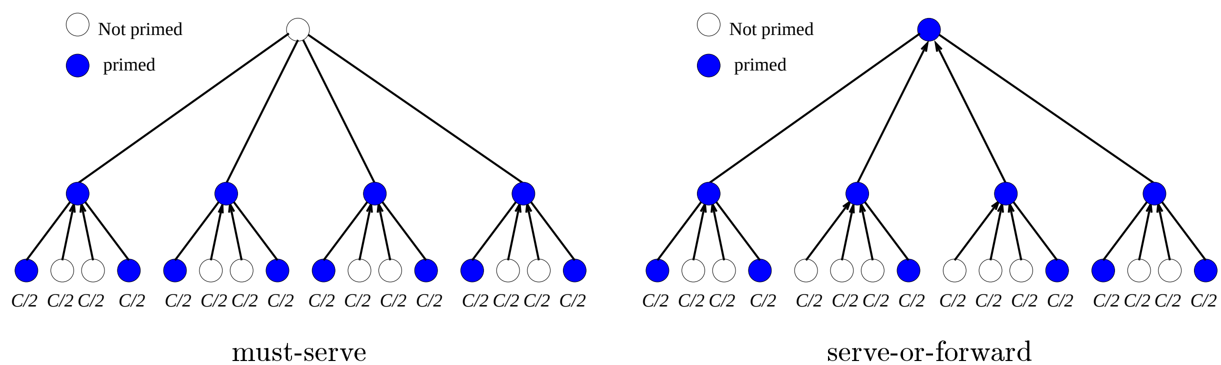

Figure 1: Example of optimal allocation for the two variants of the problem. The tree has 16 leaves each generating a flow of $C/2$, thus the total flow is 8C. Serve-or-forward primes 11 nodes and must-serve primes 12 nodes.

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 Δ. The following lemma show a better bound for the must-serve model.

Lemma 3.1 *For any C and $\Delta > 4$, there are instances such that $\Delta R/(4C)$ storage is necessary with must-serve.*

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

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

Consider a 2-level subtree of this tree consisting of a node and Δ leaves at the bottom of the tree. At least $\lceil \Delta/2 \rceil$ of its children must be primed, since otherwise the parent receives more than C units of flow. Furthermore, there is an optimal selection that uses exactly $\lceil \Delta/2 \rceil$ children. Otherwise, since the parent node can serve all remaining children, it can replace all "extra" primed children, serving at least the same flow with at most the same coverage.

The number of the remaining unprimed children is $\lfloor \Delta/2 \rfloor$. The combined value of the flow of these unprimed children strictly exceeds $C/2$, therefore at or above the parent level these flows are forwarded or served as complete units, (each primed ancestor node can serve the flow of exactly one of these remaining-children-flows). Thus, there is one primed internal node for each of the Δ^{d-1} sets of remaining children.

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 $\Delta > 8$, there are instances such that $\Delta R/(8C)$ 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. Thus, since each leaf generates $2C/\Delta$ flow we need to prime at least

$$\Delta^d \left(\frac{2}{\Delta} - \frac{1}{(\Delta - 1)} \right) \frac{\Delta}{2} \geq R\Delta/(8C) \ .$$

leaf nodes. □

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.

The algorithm makes implicit assignment of flow to primed nodes. 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.[2]

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 T' 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 1 (OPT_SOF for serve-or-forward)
Iterate *until all nodes are processed:*
Let v be a lowest node such that $f(v) > C$.
Order the children of v by increasing flow: $v_1, v_2, ...$ with $f(v_1) \leq f(v_2) \leq \cdots$.
Let j be the smallest integer such that $\sum_{i \leq j} f(v_i) \geq C$.
prime the node v.

- *if v is not the root (generate the residual problem):*
  transfer the children $v_{j+1}, v_{j+2}, \ldots$ with their respective flow values $f(v_{j+1}), \ldots$ and the node v_j with flow value $f(v_j) - C + \sum_{i<j} f(v_i)$ to the parent of v.

- *if v is the root:*
  prime the children nodes $v_{j+1}, \ldots$. If $\sum_{i \leq j} f(v_i) > C$, prime the node v_j.

If the root node has $0 < f(r) \leq C$ and it has not been primed, then prime the root.

The implicit flow assignment made to the node v selected in each iteration of the first phase is all the flow from the children $v_1, \ldots, v_{j-1}$ and any $C - \sum_{i<j} f(v_i)$ units of flow from v_j. It is easy to see that there is one-to-one correspondence between solutions of the residual instance T' and solutions of the original instance T that prime v and allocate to it these C units of flow from $v_1, \ldots, v_j$.

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 may need to reinsert the last deleted child, and finally we meld the heap of v with the heap of the parent of v.

[2]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.

A distributed implementation of OPT_SoF follows the two phases. The first "upward" phase, communication occurs from the leaves to the root with nodes primed along the way. The information of transferred children is transferred up the tree. In the second "downward" phase, the root node "decided" which descendants need to be primed, and this information is passed down the tree to these nodes. 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. It is not hard to see that the communication can be reduced considerably if an aggregation is performed of all transferred children of similar sizes. In this case, logarithmic size messages yield an approximate solution.

We now establish the correctness and optimality of OPT_SoF.

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

We first establish correctness. A residual problem is generated after each iteration of the first phase. Once a node is primed in the first phase it gets exactly C units of flow in the residual problem and therefore no more flow can be assigned to it later on. All unserved flow that was generated below it, is transferred to its parent. The transferring is ok, since any ancestor is able to serve a unit of flow. When the first phase is completed all the children of the root in the residual network generate less than C flow. Therefore having a subset of these children serve the flow they generate, and having the root serve the rest, produces a legal solution.

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 A be the primed node v along with the flow assigned to it, and let T' be the residual problem of T with respect to A. It is not hard to see that if T has an optimal solution that extends A, then any combination of A with an optimal solution of T' yields an optimal solution of T. Thus by induction on the iterations of the first phase, it suffices to show that the choice of A can be extended to an optimal solution of T. As a base case of the induction we also have to show that the solution of the residual problem obtained by the second phase is optimal. The optimality of the priming of the second phase is straightforward: Since all the children of the root receive less than C units of flow it is easy to see that there is an optimal solution that primes only the root and a subset of its children. The following claim shows that a choice A of OPT_SoF can be extended to an optimal solution.

Claim 4.1 *Consider an instance T and let v be a lowest node with flow exceeding C. There exist an optimal solution with v being primed and fully utilized.*

Proof: Consider an optimal solution S where v is not fully utilized. We construct another solution S' of the same size in which v is primed and fully utilized. Consider the following three cases.

- Node v is primed in S but not fully utilized. We re-assign flow generated at descendants of v to be served at v. Since there are at least C units of flow generated below v, v is fully utilized in S' after this reassignment.

- Node v is not primed in S, but at least one descendant u of v is primed in S. We prime v instead of u, and have it serve the flow that u serves. As before, if v is not fully utilized (because u was not fully utilized) then we reassign flow generate at descendants of v to be served at v. This must make v fully utilized in S'.
- Node v and neither of its descendants is primed in S. Consider the path from v to r. All flow generated below v is served by primed nodes along this path. Let u be the primed node closest to v along this path (u must exist since the flow below v must be served.). We show that we can prime v instead of u. If u only serves flow generated at v or below then we are done. Otherwise, since the flow out of v is at least C, there is a primed node w above u that serves flow generated below v. We can reassign the flow to the primed nodes such that some flow from v assigned to w is assigned to u, and the same amount of flow assigned to u which does not come from v is assigned to w. We repeat this process of reassigning flow until all the flow served by u comes from v and then prime v instead of u. We then make v fully utilized as in the cases above.

□

Claim 4.1 shows that the first node v primed by OPT_SOF is also primed and fully utilized in some optimal solution. It remains to show that the flow assigned to v by OPT_SOF is also consistent with some optimal solution. This is established in the following claim and concludes the optimality proof.

Claim 4.2 *Let v be the first node primed by* OPT_SOF. *Consider a optimal solution S where v is primed and fully utilized. We can restructure S such that the flow allocated to v is the same as the flow allocated to v by* OPT_SOF.

Proof: Since the flow reached to each child of v is at most C we may assume that S does not contain prime descendants of v which are not immediate children of v. If S does not contain primed children of v at all, then units of flow generated below v are interchangeable in terms of where they can be served. Thus, any assignment of flow to nodes can be changed such that the flow assigned to v is as that assigned by OPT_SOF.

So assume S does contain i primed children of v. Let F_c be the total flow generated below v which is served by children of v and let F' be the flow generated below v which is served by v or one of its ancestors. (since v is fully utilized at S, $F' \geq C$.) Let $v_1, \ldots, v_k$ be the children of v ordered by increasing incoming flow and let j be the smallest integer such that $\sum_{\ell > k-j} f(v_\ell) \geq F_c$. Clearly $j \leq i$. We replace the i original primed nodes with $v_k, v_{k-1}, \ldots, v_{k-j+1}$. Nodes $v_k, \ldots, v_{k-j+2}$ serve all flow generated underneath them and v_{k-j+1} serves $F_c - \sum_{\ell > k-j+1} F(v_\ell)$ of the flow generated in its subtree. With this assignment, F_c of the flow is served below v, using $j \leq i$ primed nodes. The remaining F' flow generated underneath v is interchangeable with the original F' flow generated underneath v and served by v and its ancestors. Thus we may assume that the C units of flow underneath the "lightest" (in terms of generated flow) children of v is assigned to v. $\square$

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 2 (OPT_MS) *Let v be a lowest 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: First observe that no node can pass 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, any node v that has flow which exceeds C must have at least one primed node in its subtree. Consider a lowest 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 at most the same amount of flow.

The claim concludes the correctness proof since it implies that the repeated step of choosing a lowest 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. Recall that v must have at least one other primed descendant. 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 and we assume each node knows Δ, the indegree of its parent node (or some bound on it).

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

Algorithm 3 (THRESHOLD(C, Δ)**)** *Let $f(v)$ be the flow into the node v.*
The node v is primed if and only if

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

Initially, the algorithm starts with some configuration of primed nodes (possibly no primed nodes), and each node runs a copy of THRESHOLD. 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.

In fact, THRESHOLD(C, Δ) converges to the same allocation generated by the following iterative algorithm.

- 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.

To see that THRESHOLD indeed generates the same solution as the algorithm above consider one iteration, and denote by N the respective set of "lowest" nodes with flow exceeding C/Δ. Descendants of nodes in N must see incoming flow of at most C/Δ. Thus, THRESHOLD cannot make them primed, and if they started out being primed THRESHOLD will eventually unprime them. Once all descendants of nodes in N become unprimed, each node in N receives flow exceeding C/Δ and gets primed. 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. 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. 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.

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 we let t be the 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 prime to unprime or vice versa and/or updating the outgoing flow. It is easy to prove by induction that if d is the depth of the network T then the convergence time is at most $d * t$.

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 at 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 size of "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 $\epsilon C/\Delta$ flow in total. (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 gets primed. (3) Next, all primed nodes above $v_{1+1/\epsilon}$ see only $\epsilon C/\Delta$ flow and unprime themselves. (4) The lowest 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.

To summarize, we showed that the simple local THRESHOLD algorithm converges to an allocation of size $O(\Delta R/C)$. It can have, however, undesirable convergence patterns. In the next section we propose and analyse a different local algorithm, GAP.

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). In practice, this can be accomplished 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 $F_s(v) = 0$.

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 flow. This will happen however, with exponentially small probability. 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 this fraction decreases. As μ decrease the probability that a node will serve more than C units of flow also decrease.

Algorithm 4 (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 node 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 average 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 average number of primed nodes allocated by GAP in step 2. By the definition of GAP, $M(x)$ satisfies the followings. If $x < \mu C$ then $M(x) = 0$. For $x \geq \mu C$ we have that

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

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.

By induction it is easy to see that $M(x) \leq 2x/(\mu C) - 1$.

□

Since GAP is a randomized algorithm, there is 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 an internal prime 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.

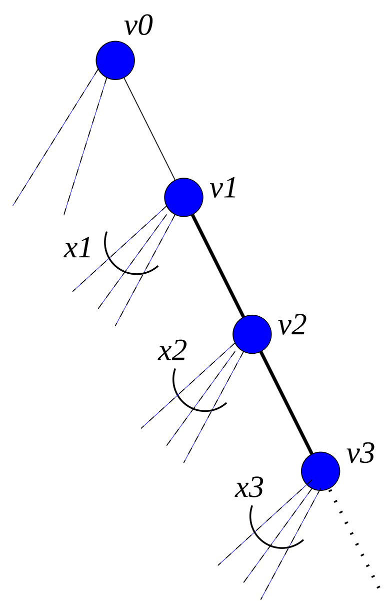

Figure 2: Illustration for the proof of Lemma 6.2. Thick lines represent edges to heaviest child.

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 to be *always-primed* if $x_i \geq 1$, or $F(v_i) \geq 1$ and either v_i is the last node on the path or $F(v_{i+1}) < 1$. (Note that the latter can happen only if v_i is the last or next to last node on the path since we assume that $F(v) < 1$ only if v is a leaf.) It is easy to see that v_ℓ or $v_{\ell-1}$ must be always-primed. Therefore there is an always-primed node on the heavy 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. It is easy to see that

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

It also follows from our definitions that $g_i = (1 - x_i)(x_i + (1 - x_{i+1})(x_{i+1} + \ldots))$ and $g_{i-1} = (1 - x_i)(x_i + g_i)$. 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 $(1 - 1/e)^L$.*

We prove the claim by downward induction on i. The base of the induction is the r.v. 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 r.v. G_i we have the relation

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

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

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

(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 $P(Y_i \geq L) \leq (1-1/e)^L$. Let $Z = Y_1 + Y_2 + \cdots + Y_\Delta$. Then $P(Z \geq \Delta L) \leq (1-1/e)^L$.*

To prove the claim, 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 $\lambda = \ln(1-1/e)$. We first show that

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

where ΔD is the sum of Δ independent copies of the random variable D. From our bound on the tail of the distribution of Y_Δ follows that

$$\begin{aligned}
P(\sum_{i=1}^{\Delta} Y_i \geq \Delta L) \ &= \\
&= \int_{x_1=0}^{\infty} \int_{x_2=0}^{\infty} \cdots \int_{x_{\Delta-1}=0}^{\infty} P(Y_1 = x_1) \cdots P(Y_{\Delta-1} = x_{\Delta-1}))P(Y_\Delta \geq \Delta L - x_1 - \cdots x_{\Delta-1}) \\
&\leq \int_{x_1=0}^{\infty} \int_{x_2=0}^{\infty} \cdots \int_{x_{\Delta-1}=0}^{\infty} P(Y_1 = x_1) \cdots P(Y_{\Delta-1} = x_{\Delta-1}))P(D \geq \Delta L - x_1 - \cdots x_{\Delta-1}) \\
&= P(\sum_{i=1}^{\Delta-1} Y_i + D \geq \Delta L) \ .
\end{aligned}$$

We can now apply the same argument to the variable $Y_1, \ldots Y_{\Delta-1}$ and D and show that

$$P(\sum_{i=1}^{\Delta-1} Y_i + D \geq \Delta L) \leq P(\sum_{i=1}^{\Delta-2} Y_i + 2D \geq \Delta L) \ .$$

For that we expand $P(\sum_{i=1}^{\Delta-1} Y_i + D \geq \Delta L)$ as a multiple integral whose variables are the values of $Y_1, \ldots, Y_{\Delta-2}$, and D, and use the fact that the probability of a tail of $Y_{\Delta-1}$ is dominated by the probability of the same tail of D. By repeating this argument Δ times we finally obtain that

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

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

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

One can easily check that $g(a\Delta) \leq f(a)$, for every $a \geq 0$, where $f(x)$ is the density function of the exponential random variable D. Therefore $P(n\Delta > \Delta L) \leq P(D > L)$ and the lemma follows. $\square$

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 $\mu = 1/c\Delta$ together with iterative application of GAP in order to obtain better tradeoff.

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. We say that a node x is *in iteration* i when x knows whether it is primed or not at stage i. All nodes are initially at iteration zero and unprimed. A node x in iteration i acts as follows.

1. If x is primed then it reports that it transfers zero flow to its parent in iteration i.

2. Otherwise (x is not primed) x waits for each of its children to report how much flow it transfers to x in iteration i. When x knows these values for all its children then it sums up these value and reports this sum as the flow value that it transfers to its parent in iteration i.

3. Last, if x is not primed then it runs GAP using the flow values of iteration i that it received from each of its children, to calculate whether it gets primed in iteration $i + 1$. Node x then is in iteration $i + 1$.

When we apply this distributed algorithm 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 which is primed in iteration i that receives less than C units of flow from all its children combined, sends a message downwards to its descendants announcing that they may stop running GAP. A node that got primed in iteration j that receives a message saying that its has a primed ancestor that serves less than C units of flow in iteration $i < j$ unprimes itself.

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 means that in each iteration, the expected total flow injected into "active" subtrees decreases by a constant factor. [3] It follows that on average, after $O(\log(R/C))$ iterations the total flow in active trees is at most C and the algorithm halts.

The expected number of primed nodes in each iteration is bounded by $(\Delta + 1)/C$ times the average flow value. Since the average flow value decreases geometrically, at the end we obtain an allocation of expected size $O(R\Delta/C)$. Recall that with this version of GAP no primed nodes serves more than C units of flow. We now prove our key lemma.

Lemma 6.3 *Consider a unit of flow generated at a leaf. When nodes are primed according to* GAP, *there is a constant probability that the primed node serving the flow (the "lowest" 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_1 be the lowest node on the path with $F(v_1) \geq \mu C$. We split the rest of the proof into the following cases.

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

In this case v_1 is primed and therefore serves our flow-unit. By the observation above the probability that v_1 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 $F(u) > \mu C$.

Let v_0 be the heaviest child of v_1, and let $i \geq 1$ be the minimum such that the node at distance i from v_0 towards the root of the tree is not a heaviest child of its parent. We denote by v_j the ancestor of v_0 at distance j from v_0 for $1 \leq j \leq i$. Let x_j, $1 \leq j \leq i$, be the flow generated at the children of v_i other than v_{i-1}.

[3] In other words we have that in each iteration with constant probability the flow into active subtrees decreases by a constant fraction.

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

Assume first that v_i is not the root and let v_{i+1} be the parent of v_i. By the definition of GAP node v_{i+1} is primed. Therefore our unit flow is served by v_j for some $1 \leq j \leq i+1$. If $j < i+1$ then the probability that v_j serves more than $\Delta\mu C$ units of flow is at most the probability that v_0 passes up more than $(\Delta-1)\mu C$ units of flow. That happens with probability at most $(1-1/e)^{\Delta-1}$. If $j = i+1$ then the probability that v_{i+1} serves more than $(\Delta+1)\mu C$ units of flow is at most the probability that its $\Delta - 1$ children other than v_i and v_0 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_i is the root then clearly our unit of flow is served by v_j for some $1 \leq j \leq i$. The probability that v_j serves more than $\Delta\mu C$ units of flow is at most the probability that v_0 passes up more than $(\Delta - 1)\mu C$ units of flow. That happens with probability at most $(1 - 1/e)^{\Delta-1}$.

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

Let k be the minimum such that $\sum_{i=1}^{k} x_j > \mu C$. If v_k is the root or $x_k > \mu C$ then clearly one of $v_1, \ldots, v_k$ serves our unit flow. Otherwise since 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 $v_1, \ldots, v_k$ is primed. Let v_j be the lowest primed node among $v_1, \ldots, v_k$. 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_0 transfer to their parents more than $\Delta\mu C$ units of flow. By the observation above this probability is at most $1 - 1/e$.

$\square$

7 Parent Priming

The THRESHOLD algorithm and iterative GAP are such that in the final allocation, no node can send to its parent more than C/Δ units of flow. This requirement was 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. The resulting allocations were of size at most $\frac{R\Delta}{C}$ in case of THRESHOLD, and $O(\frac{R\Delta}{C})$ in case of iterative GAP. This is larger by a factor of at most Δ in case of THRESHOLD, and at most $O(\Delta)$ in case of iterative GAP, from size of the optimal allocation. The technique we present in this section improves this approximation ratio by a constant factor and may be useful in practice.

Consider now variants of THRESHOLD and iterative GAP where each unprimed node in the final allocation is allowed to send up to C units of flow to its parent. (For THRESHOLD a node primes itself if and only if it gets more than C units of flow. For iterative GAP we apply it with $C' = \Delta C$.) This, of course, can result in a primed node obtaining 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 each unprimed child sends 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 7.1 *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 $(\Delta + 2)^2/(4\Delta)$. □

References

[1] B. Awerbuch, Y. Bartal, and A. Fiat. Distributed paging for general networks. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 574–583. ACM-SIAM, 1996.

[2] Open Source Community. Gnutella. In *http://gnutella.wego.com/*, 2001.

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

[4] 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.

[5] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms*. ACM, 1999.

[6] 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 '99 Conference*, 1999.

[7] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. 9th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320. ACM, 1997.

[8] 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.

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