Balanced-Replication Algorithms for Distribution Trees

Edith Cohen[1] and Haim Kaplan[2]

[1] AT&T Labs–Research,
Florham Park, NJ 07932 USA. edith@research.att.com
[2] School of Computer Science, Faculty of exact sciences, Tel-Aviv University,
Tel Aviv 69978, Israel. haimk@post.tau.ac.il

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 hotspot. 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, 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 hotspot. 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 [10] (which is based on consistent hashing [4]) and CAN [9] ("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 a 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 hotspots caused by highly skewed per-object demands. Thus, to alleviate hotspots, it is proposed in [9, 10] that intermediate nodes cache items (or keys).

In all these applications, replication on intermediate nodes constitutes a tool in avoiding "hotspots" at the roots of distribution trees. Such hotspots 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 [8]. 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].

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.

We analyse a static setting where the flow generated by each leaf remains fixed through time.[3] 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

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

it is served or not). This reporting process is similar to “hit count” reporting in cache hierarchies.

3 What allocation size is necessary?

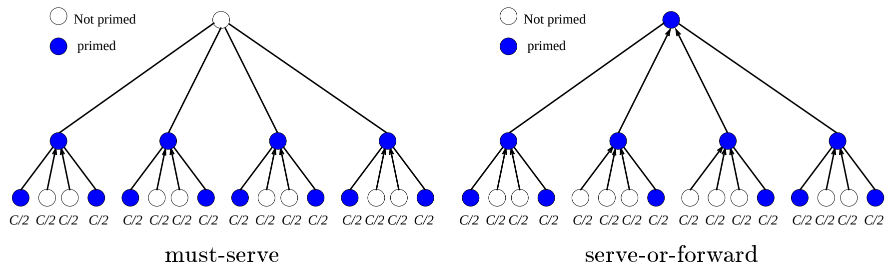

Fig. 1. Example of optimal allocation for the two variants of the problem. The tree has 16 leaves each generating a flow of $C/2$, thus the total flow is 8C. Serve-or-forward primes 11 nodes and must-serve primes 12 nodes.

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. A similar bound also exists for the server-and-forward model.

Lemma 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 $R = 2C\Delta^{d-1}$.

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
$\Delta^{d-1}(1 + \lceil \Delta/2 \rceil) \geq \frac{R\Delta}{4C}$. □

4 Optimal algorithms

In this extended abstract we present only the optimal algorithm for the serve-or-forward model (OPT_SoF). The algorithm for the must-serve model is somewhat simpler and will appear in the full version of the paper.

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.[4]

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

[4] 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.

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

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 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, which is straightforward. The following claim shows that a choice A of OPT_SOF can be extended to an optimal solution. We omit the proof from this extended abstract.

Claim. 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.

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. We omit the proof from this extended abstract.

Claim. 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.

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 2 (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.

It is not hard to prove, by induction on the iterations of the procedure above, that THRESHOLD indeed generates the same solution as the algorithm above.

The following lemma follows from this equivalence between these two algorithms.

Lemma 2. 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 describe an example where the worst-case ratio approaches Δ for the serve-or-forward model. There exists a similar example for the must-serve model that we will describe in the full version of the paper. 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.

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. [5]

[5] 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. Similarly to THRESHOLD, decisions are made locally, but unlike THRESHOLD, decisions made at 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 $F_s(v) = \max\{F(u) | u \in \text{children}(v)\}$. If v is a leaf then $F_s(v) = 0$.

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 node with flow $F(v)$.

We first bound the expected size of the allocation produced by GAP. (The proof is omitted.)

Lemma 3. *The expected number of nodes primed by* GAP *at step 2 is at most $2R/(\mu * C) - 1$, for $R \geq \mu C$.*

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 4. *In the must-serve model, 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 $i * \Delta * \mu * C$ is at most $(1 - 1/e)^i$.*

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 children combined is at most Δ.

We now prove the claim. We can assume without loss of generality that $F(v_1) > 1$ (since otherwise the claim trivially follows). 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. We call $v_1, v_2, \ldots, v_\ell$ the *heavy path* hanging from v_1. Note that v_ℓ is a leaf. Let $x_i = F(v_i) - F_s(v_i) = F(v_i) - F(v_{i+1})$ $(i \geq 0)$ be the difference in flow associated with the edge (v_{i-1}, v_i) of the heavy path. 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. Let k be the maximum such that v_k is not "always primed". Observe that we must have that $x_i < 1$ for $1 \leq i \leq k$. (That is v_{k+1} is the first always primed node on the path.)

Consider a run of dice 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 amount of flow v_0 gets from v_1 is at most $\sum_{j \leq p} x_p$. The probability that v_j $(1 \leq j \leq k)$ is the lowest-indexed primed node on the path is $x_j \Pi_{b=1}^{j-1}(1 - x_b)$. It follows from the above that the expected amount of flow passed to v_0 from v_1 is at most

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

We show that $g_1 \leq 1$. Define $g_i = (1-x_i)(x_i + (1-x_{i+1})(x_{i+1} + \ldots))$. ($g_i$ is the expected flow value passed to v_{i-1} from v_i). By definition we have $g_{i-1} = (1-x_i)(x_i + g_i)$. 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 $g_{i-1} = (1-x_i)(x_i + g_i) \leq (1-x_i)(x_i + 1) \leq 1$.

We now prove the second part of the lemma. We show that the probability, over runs of GAP, that g_i exceeds L is at most $(1-1/e)^L$. Let G_i be the random variable that corresponds to the value of g_i in different runs of GAP. We show 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 $(0 < a < 1)$.)

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 tautology $(1 - a) \leq (1 - 1/e)^a$ for $(0 < a < 1)$. $\square$

The analysis of GAP implies that if we set $\mu = \Theta(1/(\Delta \log(R\Delta/C)))$, then there is a constant probability that the maximum load served by a primed node is at most C. In the full version of this paper we show how we can use $\mu = \Theta(1/\Delta)$ together with iterative application of GAP in order to obtain better tradeoff.

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. Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the ACM SIGMETRICS'02 Conference*, 2002.
8. 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.
9. 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.
10. 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.