Randomized Caching, Probabilistic Queuing, and Denial of Service Attacks

Seth Voorhies[†], Hyunyoung Lee[†], and Andreas Klappenecker[‡]

August 5, 2003

Abstract

We propose and analyze a randomized caching scheme in which an item is cached only with a certain probability. We use such caches to build a probabilistic queuing mechanism which can provide a fair share of resources to the users by discriminating them based on the frequency of requests. We demonstrate that denial of service attacks can be better sustained when this probabilistic queuing mechanism is employed.

Keywords: denial of service, randomization, caching, probabilitic queues.

1 Introduction

Denial of service attacks try to disrupt client-server computing by flooding the server with numerous nonsensical requests or messages. The aim of the attackers is to consume a large amount of the server's resources such that benign clients are effectively denied service. The communication protocols and applications of the server queue the incoming requests. The attackers explore the fact that current queuing algorithms do not discriminate well between requests from benign and malicious clients; hence, a server under heavy load will drop many benign requests. We propose a new probabilistic queuing algorithm which attempts to alleviate this problem.

A crucial part of our queuing algorithm is utilizing a randomized cache. An incoming request is inserted with a certain probability α into the cache, and elements are evicted with a least-recently-used strategy. We model the state of the cache by a Markov chain and determine its stationary distribution. We derive the probability to find a client at position m in the cache. If we assume that a malicious client sends more requests than a benign client (which is reasonable, since otherwise the malicious client will hardly cause any harm), then our results show that the malicious client is more likely to be found in the cache. If the queue of incoming requests exceeds a certain length, meaning the load is too high, then our probabilistic queuing algorithm deletes the requests from the queue, which belong to the clients in the cache. The main theme of this strategy is to identify and penalize malicious clients, thus providing better service to benign clients.

[†]Department of Computer Science, University of Denver, 2360 S. Gaylord St., Denver, CO 80208
[‡]Department of Computer Science, Texas A&M University, College Station, TX 77843-3112

This paper is organized as follows. We introduce some notations and conventions in the next section. The probabilistic queuing algorithm and the randomized caching algorithm are given in Section 3. In Section 4, we give an analysis of the randomized caching algorithm, focusing on aspects that are relevant for the performance of our queuing algorithm. We demonstrate in Section 5 that denial of service attacks can be better sustained when standard queuing algorithms are replaced by our probabilistic queuing algorithm.

2 Preliminaries

We abstract each incoming request or incoming packet by an *item*. We assume that each item m has an identifier id which can be extracted by the function $\texttt{id}(m)$. We assume that the sender of the request is uniquely identified by this identifier. We call the process that handles or serves the incoming items a *server*. We refer to a sending entity as a *client* or a *user*.

The server maintains a queue Q of incoming request items. The queue Q can contain a maximal number of items, called the *allowable* number of items. A queue Q has two operations: enqueue and dequeue. The operation Q.enqueue(m) inserts m at the tail of Q. The operation Q.dequeue(id) removes *every* item with identifier id from Q. The operation Q.dequeue( ) without an argument removes the item from the head of Q.

The server also maintains a cache T of size t. We call the first element in T the *top* element, and the last (i.e., t-th) element the *bottom* element. The cache T is associated with the two operations: insert and delete. If T does not contain the identifier id, then the operation T.insert(id) inserts id at the top of T; otherwise it moves the id to the top of T. When T contains t elements (i.e., T is full) and a new item is going to be inserted, the item at the bottom of T will be evicted by the operation T.delete( ). Each incoming item will be inserted into T with some probability α.

3 The Algorithm

The probabilistic queuing algorithm enqueues each incoming item, unless the queue gets too full. If the queue is full, then the content of the randomized cache determines which items are deleted from the queue. The pseudocode is shown in Algorithm 1.

The randomized caching algorithm inserts the id of each incoming item m at the top of the cache T with probability α. The function $\texttt{random\_put}(\alpha)$ returns true with probability α and false otherwise. The function $\texttt{id}(m)$ returns the id of the request m. We omit the straightforward implementation details of `random_put` and `id`. The pseudocode of the randomized caching algorithm is shown in Algorithm 2.

If a process sends x requests, then the probability that its id will end up in the cache at some point in time is $1 - (1 - \alpha)^x$. Thus, processes sending more requests are more likely to be *inserted* into the cache. Our basic assumption for achieving fairness is that a malicious client sends more requests than a benign client. Under this assumption, it is intuitively clear that identifiers of malicious clients are more likely to reside in the cache

than identifiers of benign clients. We will show in the next section that this is indeed the case.

Algorithm 1 Probabilistic Queuing.

ProbabilisticQueue(α, *allowable*)

/* $\alpha > 0$ denotes the probability of an incoming request being put into T */
/* *allowable* denotes the maximum possible number of pending incoming requests */
/* Data structures: a list T of maximum size t realizing an LRU cache, */
/* a queue Q for pending requests of size *allowable* ($t \ll$ *allowable*). */

Initialization:

1: *pending* := 0; T := empty list; Q := empty queue;

When a request item m **arrives:**

2: *pending* := *pending* + 1;
3: **if** (*pending* > *allowable*) **then** /* need to dequeue requests in Q */
4: **if** (T is empty) **then**
5: choose at most t differing *id*s among the most recent requests and put these *id*s in T;
6: **forall** *id* in T **do** Q.dequeue(*id*);
7: *pending* := *pending*−(the number of dequeued requests);
8: Q.enqueue(m);
9: RandomCaching(id(m),α);

Algorithm 2 Randomized Caching.

RandomCaching(*id*, α)

/* $\alpha > 0$ denotes the probability of an incoming request being inserted into the cache T */
1: **if** (random_put(α) = true) **then** /* *id* will be inserted into T */
2: **if** (*id* exists in T) **then**
3: move *id* to the top of T;
4: **else** /* *id* does not exist in T */
5: **if** (T contains t elements) **then** /* T is full */
6: evict the bottom element of T using T.delete();
7: create new element for *id* and put it at the top of T using T.insert(*id*);

4 Analysis

The list T realizes a randomized cache with least-recently-used update strategy. The purpose of this cache is to discriminate between malicious and benign clients. We assume that the clients submitting requests to the system are given by a finite set U containing n elements. We assume that the requests of the clients $u \in U$ are independent and identically distributed with probability $\Pr[u] > 0$, $\sum_{u \in U} \Pr[u] = 1$. The cache is fairly small in an actual system, typically much smaller than the length of the pending queue. Therefore, the long term behavior of this cache is of particular interest.

We assume that the cache T is of size $t \leq n$. Requests are inserted into the cache T with probability $\alpha > 0$. Since we are only interested in the long term behavior, we may assume that the cache contains t identifiers. Hence, a state of the cache can be described by a string of t letters over the alphabet U, which contains no repetitions. The set of all possible states of the cache T is denoted by S. We use a Markov chain M_α to model the behavior of the cache T. The states of M_α are given by the set S of all states of the cache. We have $\binom{n}{t}$ selections of client identifiers in the cache, and $t!$ possible orderings, which gives a total of $\binom{n}{t}t! = n!/(n-t)!$ different states of the Markov chain.

The admissible transitions of the Markov chain reflect the move-to-front rule of the cache. Several components contribute to the transition probabilities of the Markov chain M_α: the insertion probability α, and the probability $\Pr[u]$ that client u issues a request. A state $s = (u_1, \ldots, u_t)$ of the cache remains unchanged if a message is not included into the cache or if a request of u_1 is selected for inclusion into T; the transition probability $P(s,s)$ is therefore given by $P(s,s) = 1 - \alpha + \alpha \Pr[u_1]$. If a message by $u \in U$ is selected to be included into the cache T and u is not contained in T, then the Markov model makes a transition from the current state $s = (u_1, \ldots, u_{t-1}, u_t)$ to the state $s' = (u, u_1, \ldots, u_{t-1})$ with probability $P(s,s') = \alpha \Pr[u]$. If a message by client u is selected, and u is already in T, but not at the top of the cache, then the Markov model makes a transition from the current state $s = (u_1, \ldots, u_\ell, u, u_{\ell+2}, \ldots, u_t)$ to the state $s = (u, u_1, \ldots, u_\ell, u_{\ell+2}, \ldots, u_t)$ with probability $P(s,s') = \alpha \Pr[u]$.

In the Markov chain M_α there is a nonzero probability to go from one state to any other state (in t steps), hence M_α is irreducible. Since there is a nonzero probability to stay in the same state, M_α is aperiodic. It follows that there exists a limiting probability measure π on S, which satisfies

$$(\pi(s) : s \in S)P = (\pi(s) : s \in S), \tag{1}$$

where $P = (P(s,s'))_{s,s' \in S}$ is the transition matrix of the Markov chain M. No matter in which state the Markov chain is initially, the sequence of states will approach this probability distribution [11].

Theorem 1 *The stationary distribution π on the state space S of the Markov chain M_α, with insertion probability $\alpha > 0$, is given by*

$$\pi(s) = \Pr[u_1] \prod_{k=2}^{t} \frac{\Pr[u_k]}{\left(1 - \sum_{\ell=1}^{t-k+1} \Pr[u_\ell]\right)}, \tag{2}$$

where s is the state $s = (u_1, \ldots, u_t)$.

Proof. We verify by direct calculation that the probability measure π given in (2) satisfies the stationarity condition (1). According to (1) and the transition rules of the Markov

chain M_α, we find that $\pi(s) = \pi(u_1, \ldots, u_t)$ satisfies the equation

$$\pi(u_1, \ldots, u_t) = (1-\alpha)\pi(u_1, \ldots, u_t) + \alpha \Pr[u_1] \left(\sum_{u \neq u_1, \ldots, u_t} \pi(u_2, \ldots, u_t, u) + \sum_{m=1}^{t} \pi(u_2, \ldots, u_m, u_1, u_{m+1}, \ldots, u_t) \right).$$

The first term on the right hand side models the fact that the state remains unchanged if an arriving item m of client u_1 is not included in T. The last two terms model all possible states of T, which lead to s after inclusion of m. Subtracting $(1-\alpha)\pi(s)$ from both sides and dividing by α yields

$$\pi(u_1, \ldots, u_t) = \Pr[u_1] \left(\sum_{u \neq u_1, \ldots, u_t} \pi(u_2, \ldots, u_t, u) + \sum_{m=1}^{t} \pi(u_2, \ldots, u_m, u_1, u_{m+1}, \ldots, u_t) \right). \tag{3}$$

Clearly, it suffices to check that (2) satisfies (3). It will be convenient to denote by $d_i(s)$ the term

$$d_i(s) = 1 - \sum_{\ell=1}^{t-i+1} \Pr[u_\ell].$$

Substituting (2) for $\pi(u_2, \ldots, u_t, u)$ yields

$$\sum_{u \neq u_1, \ldots, u_t} \pi(u_2, \ldots, u_t, u) = \sum_{u \neq u_1, \ldots, u_t} \Pr[u] \prod_{k=2}^{t} \frac{\Pr[u_k]}{d_{k-1}(s) + \Pr[u_1]} = \sum_{u \neq u_1, \ldots, u_t} \Pr[u] \prod_{k=1}^{t-1} \frac{\Pr[u_{k+1}]}{d_k(s) + \Pr[u_1]}$$

Note that the sum $\sum_{u \neq u_1, \ldots, u_t} \Pr[u] = d_1(s)$. Since the product term on the right hand side does not depend on u, it follows that

$$\sum_{u \neq u_1, \ldots, u_t} \pi(u_2, \ldots, u_t, u) = d_1(s) \prod_{k=1}^{t-1} \frac{\Pr[u_{k+1}]}{d_k(s) + \Pr[u_1]}.$$

Similarly,

$$\sum_{m=1}^{t} \pi(u_2, \ldots, u_m, u_1, u_{m+1}, \ldots, u_t) = \sum_{m=1}^{t} \frac{\prod_{k=1}^{t} \Pr[u_k]}{\prod_{i=2}^{m} d_i(s) \prod_{i=m}^{t-1} (d_i(s) + \Pr[u_1])}$$

Substituting the last two equations into equation (3) for $\pi(s)$, we find that

$$\pi(s) = \left(\frac{\prod_{k=1}^{t} \Pr[u_k]}{\prod_{i=2}^{t} d_i(s)} \right) \left[\frac{\prod_{i=1}^{t} d_i(s)}{\prod_{k=1}^{t-1} (d_k(s) + \Pr[u_1])} + \sum_{m=1}^{t} \frac{\Pr[u_1] \prod_{i=m+1}^{t} d_i(s) \prod_{i=1}^{m-1} (d_i(s) + \Pr[u_1])}{\prod_{i=1}^{t-1} (d_i(s) + \Pr[u_1])} \right].$$

We can simplify this expression to

$$\pi(s) = \left(\frac{\prod_{k=1}^{t} \Pr[u_k]}{\prod_{i=2}^{t} d_i(s)} \right) \left[\frac{\prod_{i=1}^{t} d_i(s) + \left(\Pr[u_1] \sum_{m=1}^{t} \prod_{i=m+1}^{t} d_i(s) \prod_{i=1}^{m-1} (d_i(s) + \Pr[u_1]) \right)}{\prod_{i=1}^{t} (d_i(s) + \Pr[u_1])} \right],$$

where we have used the fact that $d_t(s) + \Pr[u_1] = 1$. It turns out that the term in brackets is equal to 1; this is a consequence of the polynomial identity

$$\prod_{i=1}^{t} x_i = \prod_{i=1}^{t} (x_i + \lambda) - \lambda \sum_{m=1}^{t} \prod_{k=m+1}^{t} (x_k + \lambda) \prod_{\ell=1}^{m-1} x_\ell.$$

It is not difficult to prove this identity. Indeed, expanding the right hand side yields

$$\begin{aligned}
(x_1 + \lambda)(x_2 + \lambda) \cdots (x_t + \lambda) & - \lambda (x_2 + \lambda)(x_3 + \lambda) \cdots (x_t + \lambda) \cdot 1 \\
& - \lambda (x_3 + \lambda)(x_4 + \lambda) \cdots (x_t + \lambda) \cdot x_1 \\
& - \lambda (x_4 + \lambda)(x_5 + \lambda) \cdots (x_t + \lambda) \cdot (x_1 x_2) \\
& \quad \vdots \qquad \vdots \qquad \vdots \\
& - \lambda (x_t + \lambda) \cdot (x_1 x_2 \cdots x_{t-2}) \\
& - \lambda \cdot (x_1 x_2 \cdots x_{t-1})
\end{aligned}$$

Combining the first two terms gives $(x_2 + \lambda) \cdots (x_t + \lambda) \cdot x_1$. Likewise, we proceed by subsequently combining the first two terms of the resulting expression. Repeating this process $t - 2$ times yields $(x_t + \lambda)(x_1 x_2 \cdots x_{t-1}) - \lambda (x_1 x_2 \cdots x_{t-1})$. Simplifying this last expression yields the desired product. ■

Suppose that $\Pr[u_1] \geq \Pr[u_2] \geq \cdots \geq \Pr[u_n]$. The preceeding theorem shows that the most likely state in the limiting probability measure is the state $(u_1, \ldots, u_t)$. Notice that all Markov chains M_α reach the same limiting probability measure π, regardless of the insertion probability α. The main difference is that the process will converge more slowly to this limiting distribution for small values of α. It is worth pointing out that it is not recommended to have the value $\alpha = 1$, since this would allow a malicious attacker to orchestrate an attack which produces a maximal amount of cost in the cache update. The randomization $0 < \alpha < 1$ makes this more difficult to achieve.

Denote by $\Pr[u_i, m]$ the probability to find the identifier of client u_i at position m in the cache in the stationary distribution. The following theorem gives a precise analytic result for this probability:

Theorem 2 *If the requests of the clients are independent and identically distributed, then the probability* $\Pr[u_i, m]$ *to find the identifier of client* u_i *at position* m *in the cache in a stationary state is given by*

$$\Pr[u_i, m] = \Pr[u_i] \sum_{z=0}^{m-1} (-1)^{m-1-z} \binom{n-1-z}{m-1-z} \sum_{\substack{|Z|=z \\ u_i \notin Z}} [1 - Q_Z]^{-1}, \quad 1 \leq m \leq t, \qquad (4)$$

for any inclusion probability $\alpha > 0$. The inner sum is taken over all subsets Z of the set of clients U, and $Q_Z = \sum_{u \in Z} \Pr[u]$.

Proof. Suppose that the cache is in a stationary state. Let the history of past requests, which have been selected for inclusion in the cache, be given by

$$(\cdots, u_{j_3}, u_{j_2}, u_{j_1}).$$

If the most recent request of client u_i is request j_{k+1}, then u_i is at position m in the cache if and only if the set $X_k = \{u_{j_k}, \ldots, u_{j_1}\}$ has cardinality $m-1$. We use this simple observation to determine the probability for client u_i to be at position m in the cache.

It follows from our assumptions that the requests u_{j_k}, which are included in the cache, are independent, and identically distributed. The request u_{j_k} occurs with probability $\Pr[u_{j_k}]$. According to our previous observation, client u_i is at position m in the cache if and only if for some $k \geq 0$, $u_{j_{k+1}} = u_i$, and the random subset X_k of the past k requests does not contain u_i, and $|X_k| = m - 1$. This allows us to express the probability $\Pr[u_i, m]$ in the form

$$\Pr[u_i, m] = \sum_{k=0}^{\infty} \Pr[u_{j_{k+1}} = u_i, u_i \notin X_k, |X_k| = m - 1],$$

because the events in the brackets are disjoint for different values of k. We can state the right hand side more explicitly in terms of subsets Y of cardinality $m-1$ of the universe U of clients:

$$\Pr[u_i, m] = \sum_{k=0}^{\infty} \Pr[u_i] \sum_{\substack{|Y|=m-1 \\ u_i \notin Y}} \Pr[X_k = Y] \quad = \Pr[u_i] \sum_{k=0}^{\infty} \sum_{\substack{|Y|=m-1 \\ u_i \notin Y}} \Pr[X_k = Y].$$

The inclusion-exclusion principle yields

$$\Pr[X_k = Y] = \sum_{Z \subseteq Y} (-1)^{|Y-Z|} \Pr[X_k \subseteq Z].$$

In other words,

$$\Pr[X_k = Y] = \sum_{Z \subseteq Y} (-1)^{|Y-Z|} \Pr[u_{j_k} \in Z, \ldots, u_{j_2} \in Z, u_{j_1} \in Z] = \sum_{Z \subseteq Y} (-1)^{|Y-Z|} Q_Z^k,$$

where $Q_Z = \sum_{u \in Z} \Pr[u]$. Combining this expression for $\Pr[X_k = Y]$ with our previous formula for $\Pr[u_i, m]$ yields, after exchanging sums,

$$\Pr[u_i, m] = \Pr[u_i] \sum_{\substack{|Y|=m-1 \\ u_i \notin Y}} \sum_{Z \subseteq Y} (-1)^{|Y-Z|} \sum_{k=0}^{\infty} Q_Z^k.$$

A straightforward reformulation of this expression gives

$$\Pr[u_i, m] = \Pr[u_i] \sum_{\substack{|Z| \le m-1 \\ u_i \notin Z}} \sum_{\substack{Z \subseteq Y \\ |Y| = m-1 \\ u_i \notin Y}} (-1)^{|Y-Z|} \left[1 - Q_Z\right]^{-1},$$

which can be simplified to

$$\begin{aligned} \Pr[u_i, m] &= \Pr[u_i] \sum_{z=0}^{m-1} \sum_{\substack{|Z|=z \\ u_i \notin Z}} (-1)^{m-1-z} \binom{n-1-z}{m-1-z} \left[1 - Q_Z\right]^{-1} \\ &= \Pr[u_i] \sum_{z=0}^{m-1} (-1)^{m-1-z} \binom{n-1-z}{m-1-z} \sum_{\substack{|Z|=z \\ u_i \notin Z}} \left[1 - Q_Z\right]^{-1}, \end{aligned}$$

which concludes the proof. ■

Equation (4) shows, for instance, that client u_i is found at the top of the cache with probability $\Pr[u_i, 1] = \Pr[u_i]$. Hence the client sending most requests has the highest chance to be at the top of the cache.

For the application to denial of service attacks, we are particularly interested in the probability that a malicious client can be found ahead of a benign client in the cache.

Theorem 3 *In the stationary distribution, the probability to find client u_i ahead of client u_j in the cache is given by*

$$\Pr[u_i \text{ is ahead of } u_j] = \frac{\Pr[u_i]}{\Pr[u_i] + \Pr[u_j]},$$

regardless of the cache inclusion probability $\alpha > 0$.

Proof. Let us assume that all clients are initially in some total order. The move-to-front rule is applied when a client is included in the cache, introducing a new order. This way, the first t clients represent the state of our randomized LRU cache, and the new order ensures that a client which is evicted from the cache will be ahead of all clients outside the cache.

The probability to find client u_i ahead of client u_j after k requests is

$$\begin{aligned} &\Pr[u_i \text{ is ahead of } u_j \text{ after } k \text{ requests}] \\ &\quad = \frac{1}{2}(1 - \alpha \Pr[u_i] - \alpha \Pr[u_j])^k + \sum_{m=1}^{k} (1 - \alpha \Pr[u_i] - \alpha \Pr[u_j])^{k-m} \alpha \Pr[u_i], \end{aligned}$$

where the first term on the right hand side describes the case that u_i was initially ahead of u_j and neither were included in the cache during these k requests; the second term represents

the case that u_i is included in the cache at time $m \geq 1$, and u_j was not included after time m. A straightforward proof by induction shows that

$$\begin{aligned}&\Pr[u_i \text{ is ahead of } u_j \text{ after } k \text{ requests}] \\ &\quad = \frac{\Pr[u_i]}{\Pr[u_i] + \Pr[u_j]} - (1 - \alpha \Pr[u_i] - \alpha \Pr[u_j])^k \frac{\Pr[u_j] - \Pr[u_i]}{2(\Pr[u_i] + \Pr[u_j])}.\end{aligned}$$

In the limit $k \to \infty$, we get

$$\Pr[u_i \text{ is ahead of } u_j] = \frac{\Pr[u_i]}{\Pr[u_i] + \Pr[u_j]}.$$

This proves the claim, since we assumed that u_i and u_j are in the cache. ■

If a malicious client u_m sends $c > 1$ times more requests than a benign client u_b, then $\Pr[u_m \text{ is ahead of } u_b] = c/(1+c)$, whereas $\Pr[u_b \text{ is ahead of } u_m] = 1/(1+c)$. This shows that malicious clients are more likely to be near the top of the cache, and benign clients are more likely to be evicted from the cache or to be outside the cache.

Remark. There exists an extensive literature on the move-to-front rule for sorting [1, 2, 5, 12, 15], which corresponds to the case $\alpha = 1$ and $n = t$. More general deterministic LRU caches, with $\alpha = 1$, have been studied in [3, 9]. Although the goal of these papers is usually to estimate the expected computational cost or the expected cache miss ratio, one can learn valuable lessons from these classical works.

5 Application: A Defense Mechanism Against Denial of Service Attacks

Our probabilistic queuing algorithm was designed to provide a server with the ability to better sustain a denial of service attack. We compare our algorithm to five standard queueing algorithms which are typically used in communication protocols. We confirm experimentally that our algorithm allows to discriminate well between malicious and benign clients, as predicted by our theoretical considerations.

Background. Denial of service attacks are among the most prolific threats to client-server computing. A typical denial of serivce attack is orchestrated as follows: A malicious attacker subverts a number of machines, known as zombies, and launches an attack on the server of the victim by sending numerous packets from the zombies. The general idea is that processing the flood of messages will consume resources of the victim, and will disrupt service to benign clients; see [6, 8, 10, 13, 14] for details and variations.

An attacker has a variety of tools available, such as Trinoo, TFN, TFN2K, Shaft, and Stacheldraht, which help to coordinate and execute a denial of service attack. Even an unsophisticated individual can launch a devastating attack with the help of these tools; see [10] and the references therein.

There are a number of policies that can be used to mitigate a denial of service attack. A consequent employment of ingress and egress packet filtering in routers can greatly reduce the number of packets with forged source IP addresses in a network [4]. Disabling unused

ports and services is an easy way to reduce the computational load of the server, particularly when under attack. However, none of the aforementioned methods works perfectly. A final defense involves the ability of a server to distinguish between legitimate traffic and illegitimate traffic, so that it can still provide service to benign users. For instance, Huang and Pullen [7] suggest a random early detection based packet filtering scheme for this purpose.

Experimental Comparison. The main strategy of a denial of service attack is to consume resources of the victim server by sending many messages. The flood of requests or messages will inevitably have the effect that queues of the server handling the incoming requests will reach their maximum allowed length, causing the server to drop requests to reduce the load. It was observed in [10] that the choice of the queuing algorithm largely influences the ability of a server to sustain a denial of service attack; this observation motivated our work.

We compared our probabilistic queuing algorithm to the following five queuing algorithms: *DropTail* which implements a FIFO queue, *Fair Queuing (FQ)* which attempts to fairly share bandwidth among all queues, *Stochastic Fair Queuing (SFQ)* which uses hash functions to map flows to a queue, *Random Early Detection (RED)* which randomly discards packets to avoid congestion, and *Class Based Queuing (CBQ)* which queues packets according to specifiable rules.

We tested the ability of servers utilizing the above queuing algorithms to sustain a denial of service attack. We used the ns-2 network simulator to generate denial of service attacks using the network topology shown in Figure 1. Our main goal in testing these different algorithms was to see which would drop the least number of packets from the benign users and the largest number of packets from the attackers, in other words, which algorithm would provide the most bandwidth to the benign users and the least bandwidth to the attackers.

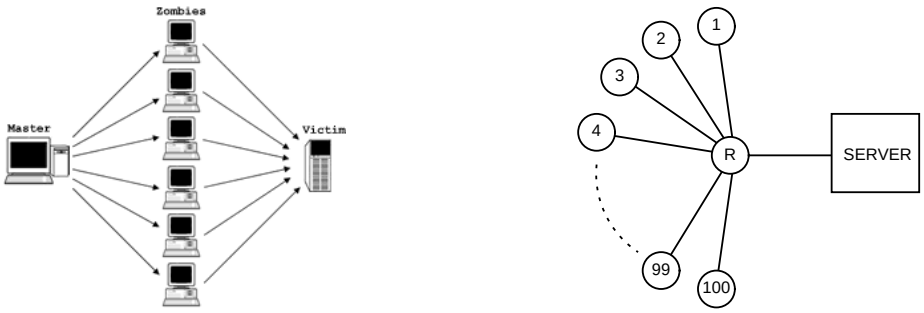

Figure 1: A typical denial of service attack scenario is shown on the left, where one attacker remotely controls numerous zombie machines. The network topology used in our simulations is shown on the right.

Each algorithm used a queue of length 50 and the simulators defaults. For FQ and CBQ we used an underlying RED queue. In our probabilistic queuing algorithm, we used in all simulations a cache of size 10. We simulated denial of service attacks using a topology with 100 clients reaching one server through one router, as shown in Figure 1. In the first simulation, the clients 1–50 are malicious (attackers), and clients 51–100 are benign. In the second simulation, the clients 1–75 are malicious, and clients 76–100 are benign.

Each simulation was run for 5 seconds during which each client in the simulation sent

UDP packets at a constant bit rate to the target web server. Every packet is of size 500 bytes. The benign clients sent packets at an interval of 0.1 seconds, while the attackers sent packets at an interval of 0.01 seconds. The target web server had a bandwidth of 2 Mbps and a delay of 5ms, while all network links had a 1 Mbps bandwidth with a 10 ms delay.

In the first simulation, with 50 malicious clients, DropTail, FQ, CBQ, RED and SFQ all provided only an average of 6% of the requested service to the benign clients, with SFQ providing no service to 13 of the 50 benign clients. Our probabilistic queuing algorithm was able to provide approximately 70% of the requested service to the benign clients, as shown in Figure 2 (a). The second simulation with 75 attacking nodes showed similar results (Figure 2 (b)).

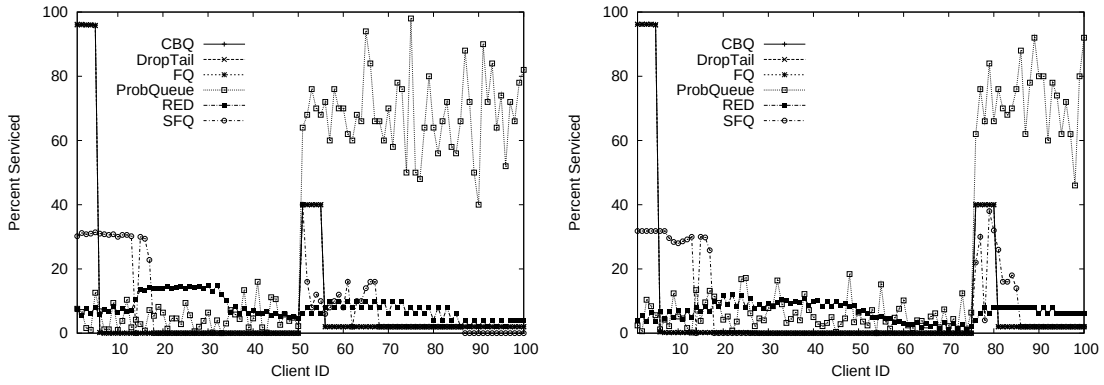

Figure 2: The graphs show for each client the percentage of packets serviced. Results for a system with 100 clients: (a) The first 50 (on the left) are malicious clients, and the last 50 (on the right) are benign clients. (b) The first 75 are malicious, and the last 25 are benign clients. Our probabilistic queuing algorithm is the only queuing algorithm providing significant service to benign clients.

It should be noted that with the probabilistic queuing algorithm, the gap between the amount of service received by the attackers and the amount of service received by the benign clients actually grew in favor of the benign clients as the number of attackers increased. However, with all of the other algorithms, the gap remained constant or increased in favor of the attackers, as the number of attackers increased.

Cache Inclusion Probability. The cache inclusion probability α of the probabilistic queuing algorithm determines the computational cost. At the same time, it affects the frequency of cache updates. For instance, a smaller value of α is computationally more efficient, but also yields less frequent cache updates so that the cache may not reflect correctly the distribution of items in the queue. We performed experiments to determine a value of α, which is reasonably small but can still guarantee a fair service to benign clients.

We ran tests based on the same two simulations as above: one with 50 malicious clients and 50 benign clients, and the other with 75 malicious and 25 benign clients. We varied the cache inclusion probability α over the range of 2%–20%. In each case, we calculated the standard deviation of the service for the benign clients. We found that a value of α around 10% provided about the same low standard deviation as in the case of $\alpha = 20\%$, whereas

smaller values of $\alpha < 10\%$ resulted in a larger standard deviation, hence a less fair service among the benign clients.

6 Conclusions

Protecting a server against denial of service attacks is a nontrivial task. We introduced a randomized caching and probabilistic queuing algorithm that can help to better sustain flooding attacks, especially when used in combination with consequent filtering mechanisms. The simplicity of the scheme allows to replace existing queuing mechanisms in servers or routers without much effort. This yields improved service without significant increase of costs.

We analyzed our randomized caching algorithm using a Markov chain model and derived a closed form for its stationary distribution. We computed, in closed form, the probability that a client resides at a certain position in the cache. Furthermore, we calculated the probability that a client is ahead of another client in the cache. Our results show that malicious clients are more likely to reside in the cache. Furthermore, we experimentally demonstrated that our algorithm is superior to other queuing algorithms in servicing requests of benign clients.

Acknowledgements. We thank Narasimha Reddy for introducing us to the concept of his partial state router [16]. The research of A.K. is supported in part by NSF grant EIA 0218582, and a Texas A&M TITF grant.

References

[1] J.R. Bitner. Heuristics that dynamically organize data structures. *SIAM J. Comput.*, 8(1):82–110, 1979.

[2] P.J. Burville and J.F.C. Kingman. On a model for storage and search. *J. Appl. Prob.*, 10(3):697–701, 1973.

[3] E.G. Coffman, Jr. and P.J. Denning. *Operating Systems Theory.* Prentice Hall, Englewood Cliffs, NJ, 1973.

[4] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing. Network Working Group, RFC 2827, 2000.

[5] W.J. Hendricks. The stationary distribution of an interesting Markov chain. *J. Appl. Prob.*, 9(1):231–233, 1972.

[6] J.D. Howard. *An analysis of security incidents on the Internet.* Phd thesis, Carnegie Mellon University, 1998.

[7] Y. Huang and J.M. Pullen. Countering denial-of-service attacks using congestion triggered packet sampling and filtering. In *Tenth Intl. Conf. Computer Communications and Networks, 2001. Proceedings.*, pages 490–494. IEEE, 2001.

[8] F. Kargl, J. Maier, and M. Weber. Protecting web servers from distributed denial of service attacks. In *Proc. 10th Intl. WWW Conference*, pages 514–524, 2001.

[9] W.F. King, III. Analysis of demand paging algorithms. In *Information Processing 71 (IFIP Congress, Ljubljana, Yugoslavia, 1971)*, pages 485–490. North-Holland, 1972.

[10] F. Lau, S.H. Rubin, M.H. Smith, and L. Trajkovic. Distributed denial of service attacks. In *Proc. 2000 IEEE Int. Conf. on Systems, Man, and Cybernetics, Nashville, TN*, volume 3, pages 2275–2280. IEEE Press, 2000.

[11] G.F. Lawler and L.N. Coyle. *Lectures on Contemporary Probability.* Student Mathematical Library, IAS/Park City Mathematical Subseries. AMS, 1999.

[12] J. McCabe. On serial files with relocatable records. *Operations Research*, 13:609–618, 1965.

[13] J. Mirkovic, J. Martin, and P. Reiher. A taxonomy of DDoS attacks and DDoS defense mechanisms. Technical report, Computer Science Department, UCLA, 2002.

[14] R.M. Needham. Denial of service: An example. *Comm. ACM*, 37(11):42–46, 1994.

[15] R. Rivest. On self-organizing sequential search heuristics. *Comm. ACM*, 19(2):63–67, 1976.

[16] Smitha and A.L.N. Reddy. LRU-RED: An active queue management scheme to contain high bandwidth flows at congested routers. In *Global Telecommunications Conference, 2001. GLOBECOM '01*, volume 4, pages 2311–2315. IEEE, 2001.