Partial Lookup Services (Extended Version)

Qixiang Sun and Hector Garcia-Molina
Computer Science Department
Stanford University, Stanford, CA 94305, USA
qsun@cs.stanford.edu and hector@cs.stanford.edu

February 25, 2002

Abstract

Lookup services are used in many Internet applications to translate a key (e.g., a file name) into an associated set of entries (e.g., the location of file copies). The key lookups can often be satisfied by returning just a few entries instead of the entire set. However, current implementations of lookup services do not take advantage of this usage pattern. In this paper, we formalize the notion of a *partial lookup service* that explicitly supports returning a subset of the entries per lookup. We present four schemes for building a partial lookup service, and propose various metrics for evaluating the schemes. We show that a partial lookup service may have significant advantages over conventional ones in terms of space usage, fairness, fault tolerance, and other factors.

1 Introduction

A *lookup service* translates a *key*, e.g., the name of a file, into an associated set of *entries*, e.g., the locations for the file. Lookup services are used by many Internet applications. For example, in a music sharing application, names of songs are translated into the IDs of computers storing the song, while in a yellow pages application, a category such as "news" may yield the URLs of news web sites. In the examples we have given, the entries point to the desired resources, but in other cases, the entries themselves may be the desired resources. In the latter case, the entries could be large.

In many cases, users do not need *all* the entries associated with a key, but only "a few." For example, a user looking for a popular song may only need two or three sites to contact for the song, not the hundreds of sites that may have a copy. In this paper we study how to exploit this observation, in order to yield lookup services that are much more "efficient" than traditional services where all entries are returned.

To illustrate, consider Figure 1, where two servers, S_1 and S_2 implement a lookup service. Let us focus on a particular key k that maps to two entries $\{v_1, v_2\}$. With a traditional full-replication approach (left,

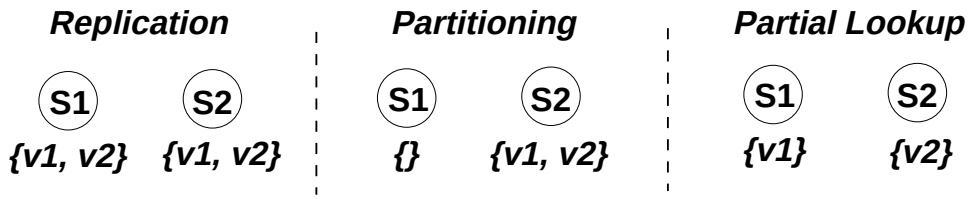

Figure 1: Three ways of managing a key with two entries v_1 and v_2.

Figure 1), the mapping of k to $\{v_1, v_2\}$ is stored on both servers. With a traditional hashing approach (center), key k is hashed to identify one server, say S_2. Thus, only S_2 stores the mapping. With a partial lookup approach (right), servers need not keep the full mapping. In our example, S_1 stores the k to $\{v_1\}$ mapping, while S_2 stores the k to $\{v_2\}$ mapping.

As we will study in this paper, the partial lookup strategy has some important advantages over the traditional approaches. For example, storage needs (and update costs) are reduced as compared to full-replication. Partial lookups can provide better load balancing than traditional hashing. For instance, in Figure 1 (center), if k is very popular, S_2 can be overloaded. Furthermore, even if S_2 is down, partial lookups can continue.

The basic idea of allowing a server to track fewer entries is rather simple. Yet, surprisingly, there are quite a few ways to implement the idea, and there are several important tradeoffs to consider. In this paper we will study and carefully evaluate various partial lookup schemes, but as a preview let us illustrate two of the options and the issues that arise. Let us return to our two server example, and consider the two schemes illustrated in Figure 2, for mapping a key k to entries $\{v_1, v_2, v_3\}$. With option (a), we place the same entry v_1 at both servers. In option (b), we use a round-robin scheme to assign individual entries to each server. Both options will return at least one entry to the user on a lookup. However, option (a) always gives out v_1 while option (b) can give out different answers depending on which server is contacted. Thus, option (b) provides more varied results, and is "fairer" because the resources represented by v_1, v_2 and v_3 are more evenly accessed. On the other hand, option (a) does not have any update cost if v_2 is deleted, while option (b) has to shuffle entries around to maintain the round-robin assignment. Option (a) also uses less storage space than (b). Other tradeoff include how resilient the service is to server failures, and the cost of processing each user lookup request.

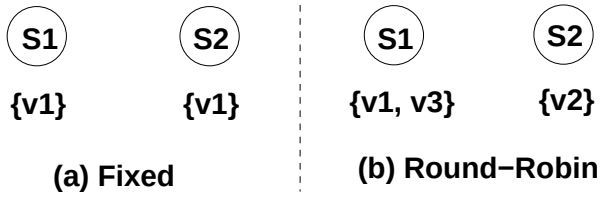

Figure 2: Two strategies of placing 3 entries on 2 servers, exploiting partial lookup.

In summary, in this paper we make the following contributions:

- We formally define the partial lookup problem (Section 2).
- We present four fundamental schemes for partial lookups, first in a static scenario with no updates (Section 3) and then in a dynamic scenario (Section 5).
- We define metrics for evaluating partial lookup schemes (Section 4).
- We evaluate our four partial lookup schemes, in some cases via simulations, in both the static and dynamic scenarios (Sections 4, 6).
- We provide guidelines or "rules of thumb" for selecting schemes.

2 Service Definition and Assumptions

A lookup service manages keys and their associated entries and allows lookup operations. In a traditional lookup service, each lookup returns all entries for a given key. Formally, we define a traditional lookup service as a service that maintains the set S = $\{(k_i, V_i)\}_i$, where k_i denotes a key and V_i denotes the corresponding set of entries for key k_i, and supports the following interface and semantics:

- *place*$(k, \{v_1, v_2, \ldots, v_h\})$: if $k = k_i$ for some $(k_i, V_i) \in S$, then $V_i \leftarrow \{v_1, v_2, \ldots, v_h\}$. Else, $S \leftarrow S \cup \{(k, \{v_1, v_2, \ldots, v_h\})\}$.
- *lookup*(k): if $k = k_i$ for some $(k_i, V_i) \in S$, then return V_i. Else, return $\emptyset$.
- *add*(k, v): if $k = k_i$ for some $(k_i, V_i) \in S$, then $V_i \leftarrow V_i \cup \{v\}$. Else, $S \leftarrow S \cup \{(k, \{v\})\}$.
- *delete*(k, v): if $k = k_i$ for some $(k_i, V_i) \in S$, then $V_i \leftarrow V_i - \{v\}$.

The *place* interface specifies a set of entries for a key in batch whereas *add* and *delete* provide incremental updates. And *lookup* returns the current set of entries for a given key. The goal is to implement this service on n servers to support higher reliability and higher user access rates. For this paper, we will not consider adding and removing servers as part of the lookup service interface.

As noted earlier, a user lookup on key k_i does not require returning the entire set V_i. Hence we define a *partial lookup service* as a traditional lookup service that supports *partial_lookup*(k, t) instead of *lookup*(k) where

- *partial_lookup*(k, t): if $k = k_i$ for some $(k_i, V_i) \in S$, then return $V \subset V_i$ such that $|V| \geq t$. Else return $\emptyset$.

As mentioned earlier, in many cases clients only need a small fraction of the entries for a key k, i.e., they only need one or a small number of copies of the resource represented by k. In such common cases, they can use *partial_lookup*(k, t) to retrieve at least t entries. The parameter t is the *target answer size* of the lookup.

The above definitions involve multiple keys. However, each key can be managed separately, e.g., replicate a single-key strategy to manage more than one key at a time. In fact, different strategies can be used to manage different types of keys. For instance, frequently updated keys require strategies with small update costs, while static keys want low lookup costs and fairness in which entries are returned. Since it is easy to generalize, we focus here on strategies that manage only one key. Hence we omit the key parameter in the interface and describe each strategy in terms of *place*$(\{v_1, v_2, \ldots, v_h\})$, *add*$(v)$, *delete*$(v)$, and *partial_lookup*$(t)$.

Focusing on the basic tradeoff decisions, we make the following two assumptions about the clients of the service. When a client C performs a *partial_lookup*(t),

- C does not care which t entries are returned in the response.
- C can access all n servers directly.

Relaxing the above two constraints yileds additional optimization problems that are discussed in Section 7

3 Strategies for Static Placement

Before considering dynamic *add* and *delete* operations, we first introduce the strategies in the static placement setting where entries are placed on the servers once via *place*$(v_1, \ldots, v_h)$, and followed by *partial_lookup* operations only.

3.1 Full Replication Strategy

The naive strategy is to store all h entries for a given key on all n servers. This traditional full replication strategy does not take advantage of the partial lookup property at all. Since every server has the same entries, a client can contact any server during a lookup operation, which spreads out the workload among the servers. Formally,

- *place*$(\{v_1, v_2, \ldots, v_h\})$: a client selects a server S at random and sends the *place*$(\{v_1, v_2, \ldots, v_h\})$ request. After server S receives the request, S broadcasts a *store*$\{v_1, v_2, \ldots, v_h\}$ message to all servers. Upon receiving the *store* message, each server makes a local copy of $\{v_1, \ldots, v_h\}$.
- *partial_lookup*(t): a client selects a random server to do the lookup. If the server has failed, keep on selecting another random server until an operational server is found. Each contacted server, returns t random entries from the h stored entries.

3.2 Fixed-x Strategy

An obvious "improvement" is to only store a *fixed* subset of the h entries at all the servers, e.g., the first x of the h entries. Thus one implementation of Fixed-x is to broadcast a client's $place(v_1, \ldots, v_h)$ request to all servers, and let each server keep the entries v_1 through v_x. Similar to full replication, a client can contact any server to do a lookup. Note that x must be sufficiently large so that no client will ever want more than x entries for a lookup, i.e., the target answer size t of a lookup is less than the parameter x for all lookups. Formally,

- $place(\{v_1, v_2, \ldots, v_h\})$: a client selects a server S at random and sends the $place(\{v_1, v_2, \ldots, v_h\})$ request. After server S receives the request, S broadcasts a $store\{v_1, v_2, \ldots, v_x\}$ message to all servers for a given parameter x. Upon receiving the *store* message, each server makes a local copy of $\{v_1, \ldots, v_x\}$.
- $partial\_lookup(t)$: a client selects a random server to do the lookup. If the server has failed, keep on selecting another random server until an operational server is found. Each contacted server returns t random entries from the x stored entries if $x \geq t$.

Fixed-x, exploiting the partial lookup property, uses less total storage space than full replication while preserving the balanced workload among the servers. Another distinction is that Fixed-x has a constant storage requirement while full replication require more space as more entries are added.

3.3 RandomServer-x Strategy

In Fixed-x, each server has the same subset of x entries. Thus another "improvement" is to place different subsets of x entries at the servers, i.e., let each server choose a random subset of x entries instead of the subset v_1 through v_x after receiving the $place(v_1, \ldots, v_h)$ broadcast. For lookups, a client can contact any of the servers. Formally,

- $place(\{v_1, v_2, \ldots, v_h\})$: a client selects a server S at random and sends the $place(\{v_1, v_2, \ldots, v_h\})$ request. After server S receives the request, S broadcasts a $store\{v_1, v_2, \ldots, v_h\}$ message to all servers. Upon receiving the *store* message, each server independently selects a random subset $V \subset \{v_1, v_2, \ldots, v_h\}$, such that $|V| = x$, and makes a local copy of V.
- $partial\_lookup(t)$: a client selects a random server to do the lookup. If the contacted server cannot return enough entries, the client continues to contact other servers in random order until the total number of distinct entries returned is more than t. Each contacted server returns t randomly selected entries stored on the server or all the entries if the total is less than t.

RandomServer-x has one major advantage over the Fixed-x strategy — x does not have be larger than any conceivable target answer size t. If a client wants more than x entries during a lookup, it can contact

multiple servers and merge the answers. Even though it is possible that all the servers chose the same subset of x entries, thus failing to service requests of more than x entries, the probability of such an event is extremely small. We discuss this issue as the *coverage* of the strategy in Section 4.3. A secondary advantage is that clients get more variety in which entries are returned on each lookup since servers store different sets of x entries. We quantify this notion of variety as *fairness* in Section 4.5

3.4 Round-Robin-y Strategy

Fixed-x and RandomServer-x share a common feature that each server independently decides which entries to store after the client's *place* request is broadcast. This approach implies that some entries may not be stored at any servers. A different approach is to ensure every entry is stored on some server. One implementation is a round-robin strategy: each entry is assigned to a server in a round-robin fashion. Specifically, when a client does a $place(v_1, \ldots, v_h)$, it sends the request to one of the servers S at random. Server S then hands out v_1 to servers 1 through y, v_2 to servers 2 through $y+1$, and so on. This placement guarantees all the servers have approximately the same number of entries (differ by at most y entries).

Because of the placement, on a lookup, a client can quickly retrieve t entries by contacting a server at random initially and then other servers in a deterministic sequence. For example, say a client contacts server 2 initially. If server 2 does not have enough entries, the client contacts server $2+y$, which has the fewest number of entries in common with server 2. If there is still insufficient number of entries, the client contacts servers $2+2y, 2+3y$, and so on. Formally,

- $place(\{v_1, v_2, \ldots, v_h\})$: a client selects a server S at random and sends the $place(\{v_1, v_2, \ldots, v_h\})$ request. After server S receives the request, for each v_i, S sends a message $store(v_i)$ to servers (i mod n), ($i+1$ mod n), ..., and ($i+y-1$ mod n) for a given parameter y. Each server, upon receiving $store(v)$, makes a local copy of v.
- $partial\_lookup(t)$: a client selects a random server s to do the lookup. If server s cannot return enough entries, the client continues to contact servers ($s+y$ mod n), ($s+2y$ mod n), and so on until the total number of distinct entries returned is more than t. If there are any server failures, choose random servers instead of using the above predetermined sequence of servers. Each contacted server returns t randomly selected entries stored on the server or all the entries if the total is less than t.

3.5 Hash-y Strategy

Instead of allocating entries to servers deterministically like Round-y, Hash-y uses y hash functions $f_1, f_2, \ldots, f_y$ to assign entries to servers, i.e. server S, after receiving a $place(v_1, \ldots, v_h)$ request, assigns each entry v_i to servers $f_1(v_i), f_2(v_i), \ldots, f_y(v_i)$. If two hash functions assign an entry to the same server, the entry is stored only once. On a lookup, a client contacts servers in a random order until retrieving enough entries. Formally,

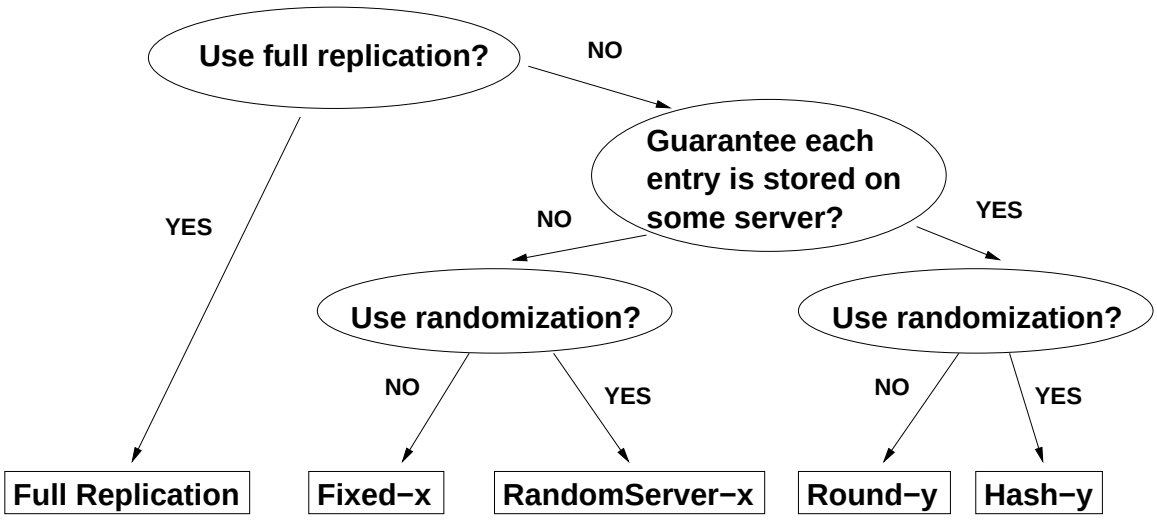

Figure 3: Classification of the five simple strategies

- *place*($\{v_1, v_2, \ldots, v_h\}$): a client selects a server S at random and sends the *place*($\{v_1, v_2, \ldots, v_h\}$) request. After server S receives the request, for each v_i, S sends a message *store*(v_i) to servers $f_1(v_i), f_2(v_i), \ldots$, and $f_y(v_i)$ for a given parameter y. Each server, upon receiving *store*(v), makes a local copy of v.

- *partial_lookup*(t): a client selects a random server to do the lookup. If the contacted server cannot return enough entries, the client continues to contact other servers in random order until the total number of distinct entries returned by the contacted servers is more than t. Each contacted server returns t randomly selected entries stored on the server or all the entries if the total is less than t.

This pseudo-random assignment of entries does not guarantee a fixed number of entries at each server. It is possible that the hash functions assign most of the entries to one server and virtually no entries to the others. This lack of guarantee implies that while a Round-y client can tell, in advance, how many servers it needs to contact for a lookup, a Hash-y client cannot. However, as we will see in Section 5, dynamic *add* and *delete* operations in Hash-y are much easier to implement than Round-y.

3.6 Summary

As a summary, we classify the strategies based on their characteristics as shown in Figure 3. Starting with the full replication decision at the root, we further divide the strategies into two categories: guarantees each entry is stored on some server or does not. Within each category, we have a deterministic version and a randomized version.

Strategy	Storage Cost
Full Replication	$h \cdot n$
Fixed-x, RandomServer-x	$x \cdot n$
Round-y	$h \cdot y$
Hash-y	$h \cdot n \cdot (1 - (1 - \frac{1}{n})^y)$

Table 1: Storage cost for managing h entries on n servers.

4 Evaluate Static Placement

In this section, we suggest five metrics for evaluating the strategies described in Section 3. These metrics only focus on the placement and the lookup aspects. We evaluate the dynamic aspects in Section 6. The first two metrics capture the operating overhead of the strategies. The last three metrics evaluate the quality of the lookup answers. In particular, we compare strategies that incur the similar overhead.

4.1 Storage Cost

The first overhead cost is the total storage required across all servers. We assume all the entries have the same size and compute the *storage cost* by counting the combined number of entries stored on all servers. Low storage cost is important if we want our entries (like IP addresses) to fit entirely in physical memory for fast access or if our entries (like music files) are so large that we might not have enough disk space.

Table 1 summarizes the costs. In particular, Hash-y's cost is less than $h \cdot y$ because collisions between multiple hash functions may result in fewer than y copies for some entries. Since each server has a probability $1 - (1 - \frac{1}{n})^y$ of storing a specific entry, the expected storage cost for Hash-y is $h \cdot n \cdot (1 - (1 - \frac{1}{n})^y)$.

Note that storage cost for Fixed-x and RandomServer-x grows as a function of the number of servers, while the cost for full replication, Round-y and Hash-y grows as a function of the number of entries for a key. Therefore, if storage capacity is predetermined and cannot be adjusted dynamically (such as the amount of physical memory), then Fixed-x and RandomServer-x is ideal, especially if there are dynamic updates.

4.2 Client Lookup Cost

The second overhead cost is the client lookup cost. We define the *lookup cost* as the expected number of servers a client will contact during a lookup. We assume there are no server failures for the purpose of computing the lookup cost. Ideally, we want the client lookup cost to be 1, e.g., a client contacts exactly one server to perform a lookup. However, if a server has less than t entries where t is the target answer size of a client, then the lookup cost will be more than 1 as the client contacts additional servers.

The ideal lookup cost of 1 is achieved by full replication since every entry is stored on each server. For Fixed-x, the lookup cost is either 1 or undefined when the target answer size t is bigger than x. Thus, with a sufficiently large x, the lookup cost is also 1. For Round-y, each server stores $\frac{y \cdot h}{n}$ entries. To satisfy a target answer size t, we need to contact $\lceil \frac{t \cdot n}{y \cdot h} \rceil$ number of servers.

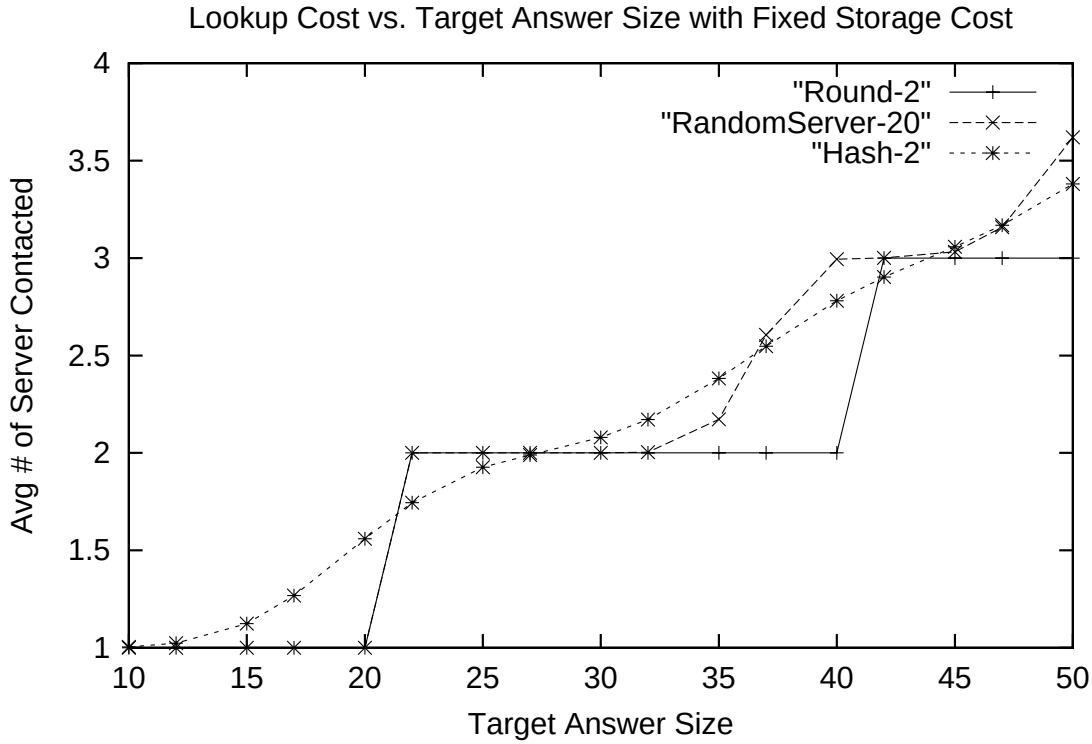

Figure 4: Lookup cost for managing 100 entries on 10 servers with a fixed storage cost of 200.

For the other two strategies, RandomServer-x, and Hash-y, finding a simple formula for the expected lookup cost is difficult because of common entries between servers and randomization. Instead, to illustrate the trade-off we performed a simulation of managing 100 entries using 10 servers while allowing each strategy to use up to 200 entries of storage space among the 10 servers. From the limit of 200 entries, we compute parameters x and y using the storage cost formula in Table 1. In this case, we get parameter x is 20 and parameter y is 2. Hence, we are comparing RandomServer-20 and Hash-2 against Round-2 and Fixed-20. The choice of 200 entries is simply to illustrate. Other limits exhibit similar behavior.

Figure 4 shows the result of the simulation with 5000 runs for each data point, where a run consists of 5000 random lookups on a particular placement of entries among the servers. We did not include Fixed-20 in the figure because it cannot answer lookups with target answer size bigger than 20. For Round-2, each server stores 20 entries in a deterministic fashion, thus a *partial_lookup* can choose servers that share no common entries. Consequently, the lookup cost increases by 1 when the target answer size t increases by 20. This creates the step curve in the figure for Round-2.

Similar to Round-2, RandomServer-20 also places 20 entries at each server. However, contacting additional servers during a lookup does not always yield 20 new entries. Therefore RandomServer-20 has higher lookup cost than Round-2, especially when the target answer size is a multiple of 20. The curve for RandomServer-20 in Figure 4 verifies the observation.

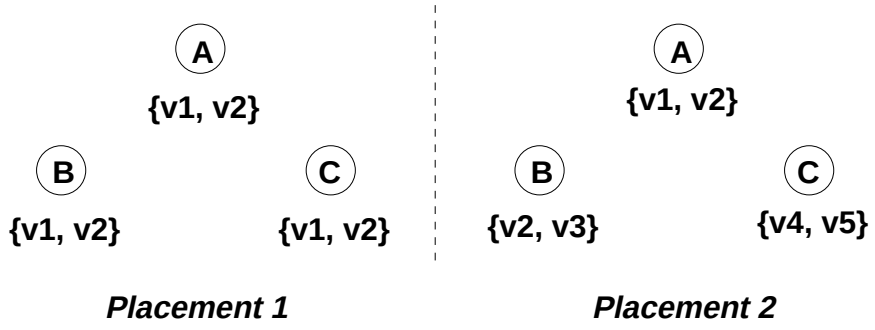

Figure 5: Two placements with different maximum coverages

Unlike Round-2 and RandomServer-20, Hash-2 does not guarantee a minimum number of entries per server. As a result, even for a small target answer size like 15, the lookup cost is 1.124 because some servers may have less than 15 entries. On the upside, for a target answer size of 25, Hash-2 may succeed in contacting only one server while all the other strategies need at least two servers as shown in the figure.

The data in Figure 4 suggests a couple of empirical rules of thumb. One, if the *partial_lookup* target answer sizes are smaller than the number of entries at each server, then avoid using Hash-y. Two, if the target answer sizes are *not* slightly more than the number of entries at each server, Round-y will give the lowest lookup cost.

4.3 Maximum Coverage

Besides the overhead, we also look at the quality of each strategy's placement. The first quality metric is the maximum number of distinct entries retrievable by a client when it contacts *all* the servers. We call this metric the *maximum coverage*. Figure 5 shows two different placements of five entries onto three servers with different maximum coverages. Both placements can satisfy a target answer size 2. However, placement 1 has a coverage of two while placement 2 has a coverage of five. The coverage establishes an upper bound on the largest target answer size t supported by a strategy. For instance, placement 1 in Figure 5 can never support a target answer size of more than 2.

There are two reasons for why the maximum coverage is interesting. First, a larger coverage implies a strategy can support a more diverse group of clients with different target answer size requirements. Second, if entries associated with a key can be deleted, then the coverage also reflects how resilient a strategy is in continuing to support a specific target answer size. Suppose we delete the entry $v2$ in placement 1 of Figure 5. The result is that all three servers only have $v1$, thus can no longer support target answer size 2. In contrast, placement 2 is less affected by such a deletion, though some lookups may require contacting up to two servers instead of just one server.

For strategies described in Section 3, full replication always has a complete coverage that includes every entry. Round-y and Hash-y also have complete coverage if the combined storage of all servers is enough to

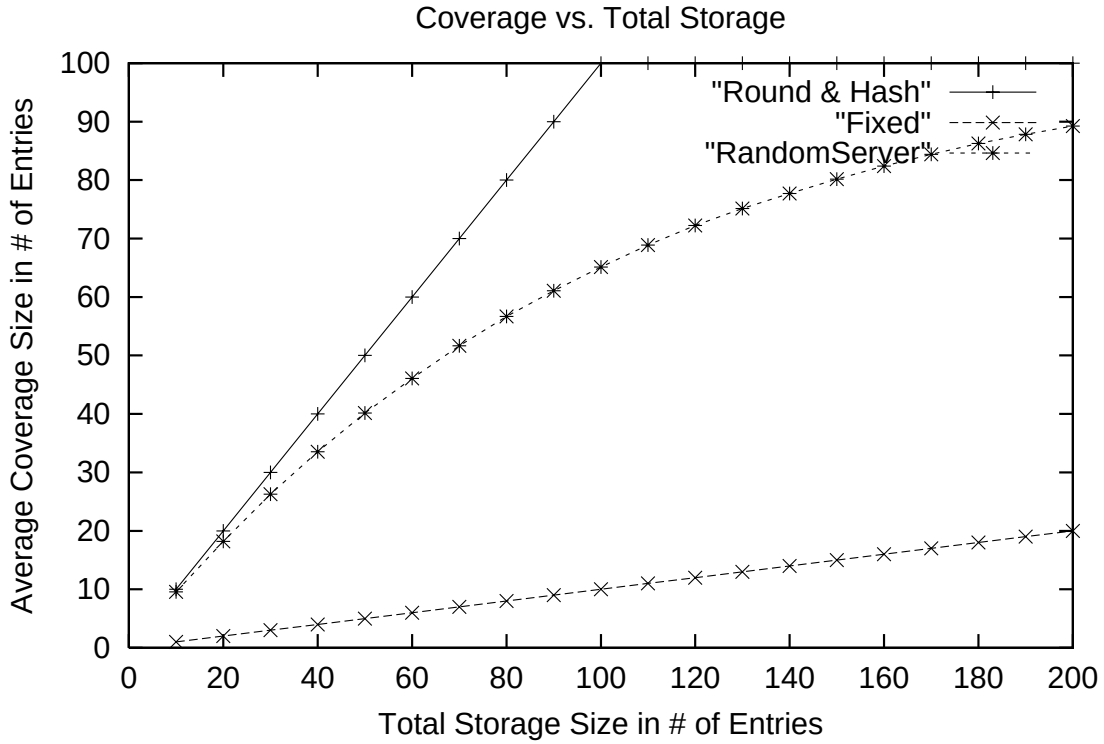

Figure 6: Coverage of strategies for managing 100 entries on 10 servers.

store each entry (v_1 through v_h) at least once. We assume when there is inadequate storage space, Round-y and Hash-y keep a subset of $(v_1, v_2, \ldots, v_h)$. Consequently, the coverage is proportional to the storage limit until every entry is stored on some server. Fixed-x, as its name implies, has a coverage of x entries. When the storage limit increases, x increases proportionally.

The coverage of RandomServer-x depends on how many entries are not stored at *any* servers. The probability that no server keeps a specific entry is $(1 - \frac{x}{h})^n$ where h is the total number of entries and n is the number of servers. Thus, the expected coverage is $h(1 - (1 - \frac{x}{h})^n)$.

In Figure 6, we shows graphically the above analysis of the coverage by varying the total storage limit. Specifically, we have 100 entries and 10 servers, and vary the total storage limit from 10 entries to 200 entries. In short, Round-y and Hash-y are ideal if clients require a large coverage, e.g., if some clients want everything.

4.4 Fault Tolerance

Another important quality metric is how many server failures can a strategy tolerate in the worst case before some client lookups fail. We consider a client lookup failed if it retrieves less than t entries specified in *partial_lookup*(t). Note that we focus on the "worst case" scenario instead of random failures. Therefore, our *fault tolerance* metric reflects the minimum number of server failures from all possible failure patterns.

In essence, we take an adversarial view in which an all-knowing adversary controls the failure pattern. This view allows us to evaluate the robustness of a strategy's placement of entries. However, computationally, the problem of finding the minimum number of server failures is equivalent to the *SET-COVER* problem, which is NP-Complete and has no constant factor approximation algorithm. Thus for the simulation data we show later, we compute the fault tolerance using a greedy heuristic that makes the server with most "endangered" entries fail first. The heuristic is described in Appendex A.

For strategies in Section 3, full replication and Fixed-x only require one operational server to service all client requests because all servers are identical. Hence, these two strategies can tolerate up to $n-1$ failures. For Round-y, the round-robin scheme assigns different entries to different servers. If a server goes down, we may lose some entries permanently. For example, consider Round-1 where each entry is assigned to exactly one server, and each server stores $\frac{h}{n}$ distinct entries. To support a target answer size t, we need at least $\lceil \frac{t \cdot n}{h} \rceil$ operational servers, i.e., we can tolerate $n - \lceil \frac{t \cdot n}{h} \rceil$ failures. In the general case of Round-y with y different round-robin iterations, the first operational server provides $y\frac{h}{n}$ entries while each additional operational server adds $\frac{h}{n}$ more distinct entries. Therefore, we can tolerate $n - \lceil \frac{t \cdot n}{h} \rceil + y - 1$ failures.

For RandomServer-x and Hash-y, computing the worse case is difficult analytically due to the randomization. We again resort to simulation. In these simulations, we first apply the strategies to place entries onto the servers. We then run our greedy heuristic to find the minimum number of tolerable server fails. We take the average of 5000 different runs for each data point. Figure 7 shows the result where we manage 100 entries on 10 servers with a storage limitation of 200 entries. As computed above for the round-robin strategy, the figure shows that increasing the target answer size by 10 reduces the fault tolerance of the strategy by 1. For RandomServer-x, we observe that it has a higher fault tolerance than round-robin because of the potential common entries between multiple servers due to the random choices. However, the same reason contributed to the higher lookup cost of RandomServer-x than Round-y, as noted earlier.

For Hash-y, the curve exhibits an S-type shape as the overall fault tolerance declines. This shape is caused by two factors. The first factor is servers in Hash-y have different number of entries. If a server with more entries than the average fails, we lose more entries. This factor implies that the curve should drop like an exponential decay. However, the second factor of having multiple copies of each entry on a random set of servers makes it more difficult to render an entry irretrievable because the locations of the entries and their duplicates are not highly correlated as in the round-robin strategy. Therefore the failure of one server is unlikely to eliminate a large group of entries from the servers. As a summary, when coverage is not important, use Fixed-x for best fault tolerance; otherwise, use RandomServer-x and Round-y for large coverage and complete coverage respectively. Hash-y should be avoided unless the target answer size is very large (e.g., more than half of the total number of entries).

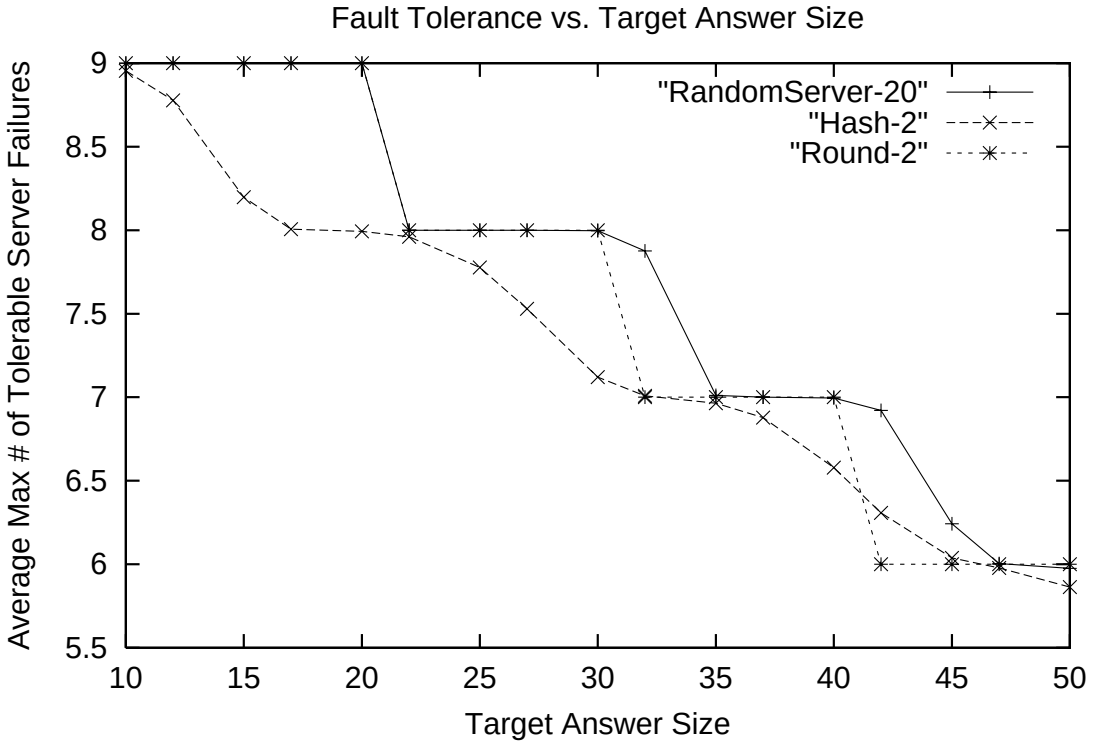

Figure 7: Fault tolerance of strategies for managing 100 entries on 10 servers using 200 entries of storage space.

4.5 Unfairness in Lookup Answers

The maximum coverage and fault tolerance are concerned with how many entries can be returned for each lookup. They do not examine whether some entries are returned more frequently than others. We capture this biasness in which entries are returned more frequently as the *unfairness* metric. The unfairness provides insights to the effects of a partial lookup service. For example, if each entry is an IP address of a service provider (e.g., a Napster client providing songs), then a biasness can overload a particular service provider.

Ideally, a "fair" strategy should return each entry with equal likelihood during any individual lookups, i.e., when a key has h entries and a user wants t entries on a lookup, a "fair" strategy has a probability $\frac{t}{h}$ of returning each entry. For example, suppose there are two entries for a key with a target answer size 1. Then a fair strategy should return either entry with a probability $\frac{1}{2}$ on a lookup. The full replication strategy satisfies the "fair" criteria because when a client asks for t entries, the server picks t entries randomly from all h entries. Clearly, not all strategies are "fair." The Fixed-x strategy only returns the first x of h entries. So a client requesting t entries has probability $\frac{t}{x}$ of getting each of the first x entries and probability 0 of getting the remaining.

To effectively distinguish unfair strategies, we first define the *unfairness of an instance*, where an instance is a specific placement of entries onto the servers as a result of executing a strategy. Let $p_I(j)$ be the

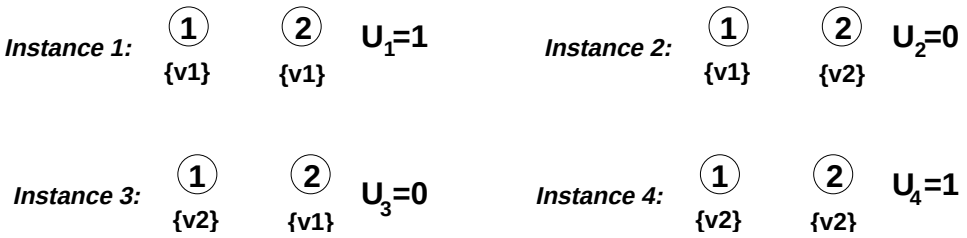

Figure 8: The four possible instances for RandomServer-1 managing 2 entries on 2 servers

probability of returning the j-th entry on a lookup in instance I. Then the unfairness U_I for this instance is

$$U_I = \frac{h}{t}\sqrt{\frac{\sum_j \left(p_I(j) - \frac{t}{h}\right)^2}{h}} \tag{1}$$

The definition above is commonly known as the coefficient of variation. The square root portion computes the standard deviation of individual entry's retrieval probability from the ideal value $\frac{t}{h}$. The $\frac{h}{t}$ factor normalizes the standard deviation with respect to the ideal value $\frac{t}{h}$. A large coefficient (e.g., on the order of 0.1 to 1) indicates some entries are returned much more frequently than others. A small coefficient implies that all the entries have about a $\frac{t}{h}$ probability of being returned on a random client lookup. For example, if we manage 2 entries on 2 servers using Fixed-1 strategy, we retrieve the first entry with probability 1 and the second entry with probability 0. This instance has an unfairness $2 \cdot \sqrt{\frac{(1-\frac{1}{2})^2+(0-\frac{1}{2})^2}{2}} = 1$. Since the coefficient is 1, the strategy is very unfair. In contrast, a round-robin strategy would have zero unfairness.

Not all strategies have only one instance like Fixed-x. For example, random choices in RandomServer-x and Hash-y create different placements of entries onto servers and result in different instances. Each of these instances can have a different unfairness. Consider RandomServer-x with 2 servers, 2 entries, and the parameter x is 1. Then depending on which entry each server decided to store, we could end up in one of the four instances shown in Figure 8. For target answer size 1, instances 1 and 4 have an unfairness 1 while instances 2 and 3 are completely fair. To accommodate the different instances, we simply take the average of the unfairnesses over all instances. Formally, let S be the set of all instances possible under a strategy. Let I be an instance in S. Let U_I denote the unfairness of instance I computed by (1). Then the *unfairness of a strategy* $\bar{U} = \frac{1}{|S|}\sum_{I \in S} U_I$. For the example in Figure 8, the unfairness of RandomServer-1 is $\frac{1}{2}$.

With random choices, computing the unfairness analytically is difficult. Instead, we use simulation that generates 10000 random lookups for each instance and computes the unfairness based on these lookups. We show the result in Figure 9. In this simulation, we manage 100 entries on 10 servers with a client target answer size 35. (Other values show similar behavior to what we illustrate here.) We vary the total storage limit from 100 entries to 1000 entries and see how unfairness changes as each server stores more and more entries. The Fixed-x, Round-y, and full replication strategies are not included in this figure. Full replication and Round-y always have 0 unfairness. Fixed-x exhibits similar behavior to RandomServer-x except an

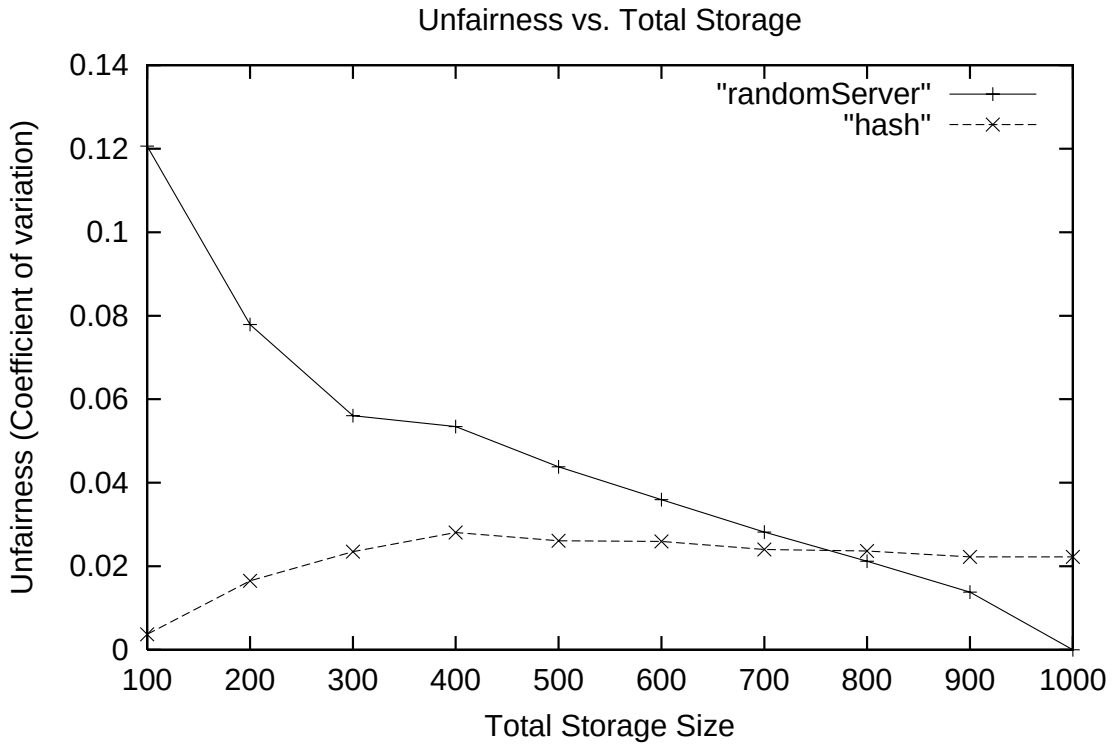

Figure 9: Unfairness of strategies when managing 100 keys on 10 servers with a target answer size 35.

order of magnitude worse.

From the figure, we see the unfairness of RandomServer-x improves in two different phases as we increase the storage. In the first phase, the unfairness decreases rapidly like an exponential decay and corresponds to low storage limit where we need to contact multiple servers for each lookup. In the second phase, the unfairness decreases linearly and corresponds to high storage limit where one server is sufficient to answer the lookup.

The exponential decay in the first phase is strongly correlated to the maximum coverage. For instance, using 200 storage space in RandomServer-x has a coverage of about 89 entries. This coverage means 11 of the 100 entries have retrieval probability 0 and imposes a lower bound on unfairness. Since the maximum coverage for RandomServer-x grows like an inverted exponential, the unfairness mirrors the effect in the first phase. The second phase has a linear characteristic because a lookup contacts exactly one server and depends heavily on which entries that server has. As we increase the storage, each server keeps more and more entries until eventually it has all the entries.

Interestingly, for Hash-y, the opposite trend is true. In the first phase, the unfairness increases rather than decreases. In the second phase, the unfairness decreases only slightly. The intuition behind the increase is that the hash functions creates an inherent biasness in which entries are stored at each server. This biasness is masked in the first phase by contacting multiple servers during a lookup. When we increased the storage,

a lookup contacts fewer and fewer servers, and the unfairness reaches the inherent biasness of the initial placement. When we increase the storage further to use more hash functions in the second phase, the number of entries stored at each server does not increase very much because multiple hash functions could map the same entry to the same server. Hence we do not get as much benefit as RandomServer-x.

In short, if we want no unfairness in the strategy, then we are forced to use either full replication or round-robin. If we can relax the unfairness constraint, then Fixed-x, RandomServer-x, and Hash-y offers several alternatives with different degrees of unfairness. One would choose the alternatives based on other metrics such as client lookup cost or dynamic update costs.

5 Dynamic Updates for the Strategies

Sections 3 and 4 have focused on the static aspects of the strategies. This section describes how each strategy handles dynamic updates.

5.1 Full Replication

For each update request, either *add* or *delete*, all servers need to be informed, thus requiring broadcasts. In this paper we assume that a client selects a server S at random, and sends S its *add* or *delete* request. Server S then broadcasts to all servers instructions to actually perform the operation. We assume that requests go to an initial server S for all strategies. Formally,

- $add(v)$: a client selects a server S at random and sends S an *add* request. Server S then broadcasts a $store(v)$ message to all servers. Each server make a local copy of v upon receiving the *store* message.
- $delete(v)$: a client selects a server S at random and sends S an *delete* request. Server S then broadcasts a $remove(v)$ message to all servers. Each server deletes the local copy of v upon receiving the *remove* message.

5.2 Fixed-x Strategy

Fixed-x uses the same approach as full replication for updates. However, since Fixed-x only keeps track of the first x entries and each server has the same entries, server S, after receiving a client $add(v)$ request, can ignore v if it already has x entries. Thus server S only does a broadcast when it has fewer than x entries. Similarly on $delete(v)$, a broadcast is needed only if server S currently stores v locally. These semantics allow Fixed-x to selectively broadcast and generate less update traffic than full replication. Formally,

- $add(v)$: a client chooses a server S at random and sends the *add* request. Server S checks whether it currently has less than x entries. If so, server S broadcasts a $store(v)$ message to all servers. Otherwise,

server S ignores the *add* request. Upon receiving a $store(v)$ message, each server stores the entry v locally.

- $delete(v)$: a client chooses a server S at random and sends the *delete* request. Server S checks whether v is currently stored locally. If so, server S broadcasts a $remove(v)$ message to all servers. Otherwise, server S ignores the *delete* request. Servers delete v locally after receiving $remove(v)$.

There are two caveats in this scheme. First, there is no concurrency control. If two *add* requests arrive at two different servers simultaneously when there are $x-1$ entries per server, both requests will be processed. Second, servers may have fewer than x entries after *deletes* because it is impossible to find a replacement for the deleted entries. Hence to support a client target answer size t, pick parameter x as $t+b$ where b is a cushion for having *deletes* without new *adds*. The cushion b does *not* guarantee the service will always return at least t entries per lookup; it merely reduces the probability of getting fewer than t entries. Note that the cushion size b could be much smaller than the number of consecutive *deletes* since not all of them will be one of the x stored entries. Section 6.2 looks at what are reasonable cushion size b.

5.3 RandomServer-x Strategy

Unlike the selective broadcast in Fixed-x, an update in RandomServer-x could affect any subset of the servers. Therefore after receiving an update request at server S, S must broadcast the request to all servers and let each server perform its randomized decisions as follows. On $add(v)$, if a server has fewer than x entries, v is stored locally on the server automatically. If a server has x entries, then it decides whether to keep its current subset of x entries or replace one of the old x entries with the new entry v. It has been shown in [8] that to maintain a uniformly random subset of x entries, a server should, with probability $\frac{x}{h+1}$, keep v and remove one of the x entries at random, where h is the current number of entries in the system. However, this incremental scheme only guarantees a uniformly random subset of x entries if there are no *deletes*.

On $delete(v)$, we use the same cushion scheme as Fixed-x. An alternative is to actively find a replacement for a deleted entries by contacting other servers, since two servers are not likely to have the same entries. This replacement alternative uses less storage because we do not need to keep cushion entries. However, neither option preserves RandomServer-x's advantage of much lower unfairness (Section 4.5) than Fixed-x. In fact, the replacement alternative results in higher unfairness than the cushion scheme when there are *deletes*. Because finding a replacement is a costly operation, we chose the simpler cushion scheme. Section 6.3 looks at how quickly the fairness deteriorate when there are *deletes*. Formally,

- $add(v)$: a client picks a server S at random and sends the *add* request. Server S broadcasts a $store(v)$ message to all servers. Upon receiving $store(v)$, each server T first increment its local counter h. T then performs a coin flip. With probability $\frac{x}{h}$, T stores v locally and deletes one of the existing x entries at random. Otherwise, T does nothing.

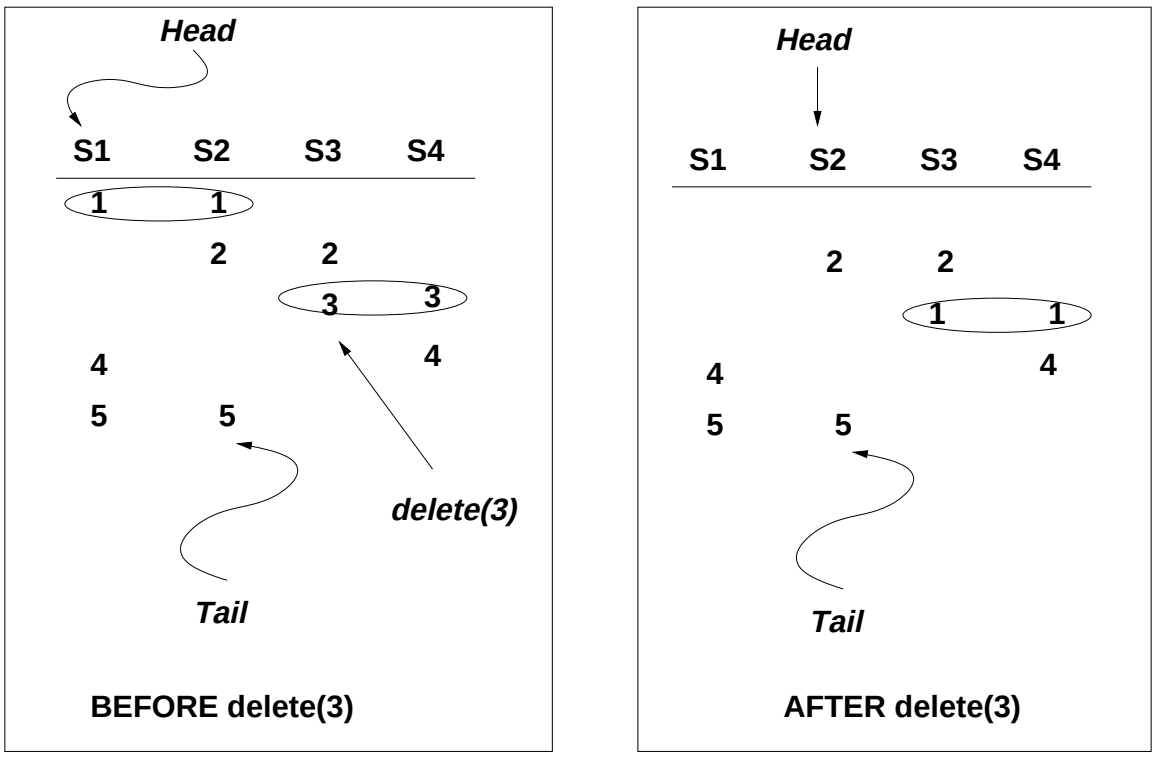

Figure 10: Using the *head* counter to plug holes left by the delete operations in the middle of the round-robin sequence.

- *delete*(v): a client picks a server S at random and sends the *delete* request. Server S broadcasts a *remove*(v) to all servers. Upon receiving the *remove* message, each server decrements its local counter h and deletes its local copy of v if one is present.

5.4 Round-Robin-y Strategy

Ensuring round-robin placements of entries onto servers with updates is problematic. For example, a series of *deletes* that affect only one server create holes in the round-robin sequence. If these holes are not fixed, we lose the benefits of lowest lookup cost, unfairness, and fault tolerance. Also, *adds* require knowledge of the current round-robin sequence.

To simplify the implementation, we maintain two dedicated counters[1] *head* and *tail* at server 1. The *head* counter indicate where the round-robin started. The *tail* counter points to the next server in the round-robin sequence. During an *add*(v), a client sends the request to server 1. Server 1 then stores y copies of v at servers *tail* through $tail + y - 1$. During *delete*(v), server 1 uses an entry currently stored at servers *head* through $head + y - 1$ to plug the hole left in the round-robin sequence by removing v. Figure 10 shows an example of this "plugging the hole." When entry 3 is removed, entry 1, pointed to by *head*, replaces

[1] The centralized *head* and *tail* scheme can be generalized to one where several servers store copies to improve reliability.

entry 3 in the middle of the round-robin sequence. Since there are y (2 in this case) different copies of entry 1, all y copies are migrated. Afterwards, *head* is incremented to reflect this migration. The drawback of this scheme is the performance bottleneck for maintaining *head* and *tail* on server 1. Also, broadcast is still necessary during $delete(v)$ to locate v. Formally,

- $add(v)$: a client sends the *add* request to server 1. Server 1 forwards the $store(v)$ message to servers $(tail \bmod n), (tail+1 \bmod n), \ldots, (tail+y-1 \bmod n)$ to store v. The counter *tail* is then incremented by 1. Each server receiving the *store* message stores v locally.

- $delete(v)$: The pseudo code for handling deletion and migration is shown in Figure 11. A client initiates the operation by sending $delete(v)$ to server 1.

5.5 Hash-y Strategy

Since Hash-y uses the hash functions to determine which servers are affected by the update request, it avoids dealing with dedicated counters or broadcasts. After receiving an $add(v)$ or $delete(v)$ request at server S, S informs the affected servers $f_1(v), \ldots, f_y(v)$ directly to add or remove the entry. Formally,

- $add(v)$: a client chooses a server S at random and sends the *add* request. Server S then sends the message $store(v)$ to servers $f_1(v), f_2(v), \ldots, f_y(v)$. After receiving the store message, a server stores v locally.

- $delete(v)$: a client chooses a server S at random and sends the *delete* request. Server S then sends the message $remove(v)$ to servers $f_1(v), f_2(v), \ldots, f_y(v)$. After receiving the remove message, a server deletes v locally.

Note Hash-y is essentially the dual of Fixed-x. Fixed-x does a lot of work (the broadcasts) for a small number of updates while Hash-y does little work on each update. We compare of the two strategies under dynamic updates in Section 6.4.

6 Evaluate Dynamic Updates

We first describe how our simulation of dynamic updates are generated. We then focus on individual strategies.

6.1 Generating Synthetic Updates

We use a discrete-time event-driven simulation to study the dynamic behavior of the strategies. We create update events with timestamps in advance and replay these events in the simulation. In order to focus on the

```
delete(v) @ server 1:
   broadcast ‘‘remove(v, head)’’
   head = (head + 1) mod n

remove(v, head) @ server X:
   if X == head then // is server X responsible for finding a replacement for v
                  // If yes,
      M[v] = 0   // initialize counter for tracking how many migration requests so far
      R[v] = u   // and use u as the replacement for v
   end

   if v is stored at this server then
       delete local copy of v
       u = migrate_[head](v) // ask server ‘‘head’’ for a replacement for v
       store u locally
   else
       ignore message
   end

migrate(v) @ server X:
   M[v] = M[v] + 1  // one more server has asked for replacement

   if M[v] == y then // serviced all migration request yet?
       // If so, remove the replacment entry R[v]
       for i=0 to y-1 do
            remove_[X+i mod n](R[v]) // ask server ‘‘X+i mod n’’ to remove R[v]
       end
   end

   return R[v] // return the replacement entry

remove(u) @ server X:
   delete u locally
```

Figure 11: Pseudo code for handling a deletion in the middle of the round-robin sequence. Each procedure describes what to do when receiving the corresponding message. The notation “proc_[X](y)” means sending a “proc(y)” message to server X and wait for the reply.

steady-state behavior, we generate the *add* events separately from the *delete* events such that the expected number of entries maintained by the servers is constant over time.

Specifically, we generate the *add* events using the Poisson arrival model with an expectation $\lambda = 10$, i.e., one *add* event per 10 time units. For each $add(v)$ event, we then determine the lifetime of the entry v using a second distribution and record the corresponding $delete(v)$ event at the end of its lifetime. For this study, we experiment with both an exponential distribution and a Zipf-like distribution for an entry's lifetime. We chose these two distributions because one is tail-heavy while the other is not.

To get the steady-state of h entries in the system when the arrival rate is λ, the distributions of an entry's lifetime are scaled so that their expectation is $\lambda \cdot h$. Thus for the exponential distribution, we get $P(t) = \frac{1}{\lambda \cdot h} e^{\frac{1}{\lambda \cdot h} t}$ for $t \geq 0$. For the Zipf-like distribution, we use $P(t) = \frac{1}{t \log(C)}$ for $t \in [1, C]$ where $\frac{C}{\log(C)} = \lambda \cdot h$. With all these distributions, we take the average of 5000 runs for each data point used in this section. If not mentioned specifically, we use a sequence of 10000 updates per run as default. The large number of runs per data point results in small confidence interval. For the 95% confidence level, the intervals is always smaller than 0.1% of the sampled mean. Therefore, we do not show the confidence interval in our graphs.

6.2 Cushion Factor for the Fixed-x Strategy

The update behavior of the Fixed-x strategy, described in Section 5.2, often results in fewer than x entries on each server when *delete* occurs. Consequently, even if we know, a priori, that no clients will request more than t entries per lookup, we cannot simply run Fixed-x with $x = t$. Instead, we need to set $x = t + b$ where b is the cushion factor. One natural question then is how big is this cushion factor b.

In Figure 12, we show the result of a simulation where our steady-state is 100 entries in the system and a client only wants 15 entries per lookup. We vary the cushion factor b from 0 to 7 entries and plot the percentage of time in which the client failed to retrieve 15 entries. Each data point in this figure is an average of 5000 runs where a run is 20000 updates. The percentage is plotted in log scale. For 0 cushion, we get over 10 percent failures. As we increase the cushion, the failure time drops exponentially. Note that the heavy-tail Zipf-like distribution tapers off at the end.

The trend in Figure 12 is similar for different target answer sizes. But the exact cushion size needed to achieve a certain failure rate depends on two factors: the target answer size in $partial\_lookup(t)$ and the expected length of an entry's life time in the system. These two factors are opposite of each other. As a client wants more entries per lookup, the cushion size grows proportionally. But as the expected life time of an entry increases, the cushion size decreases proportionally. For example in Figure 12, a cushion size 3 yields a failure rate 0.1% when the target answer size is 15 and the average life time is 1000. If the average life time doubles to 2000 time units, a cushion size 2 is sufficient for the same target answer size 15. This behavior is because longer entry life time implies the probability of multiple *deletes* occurring between two

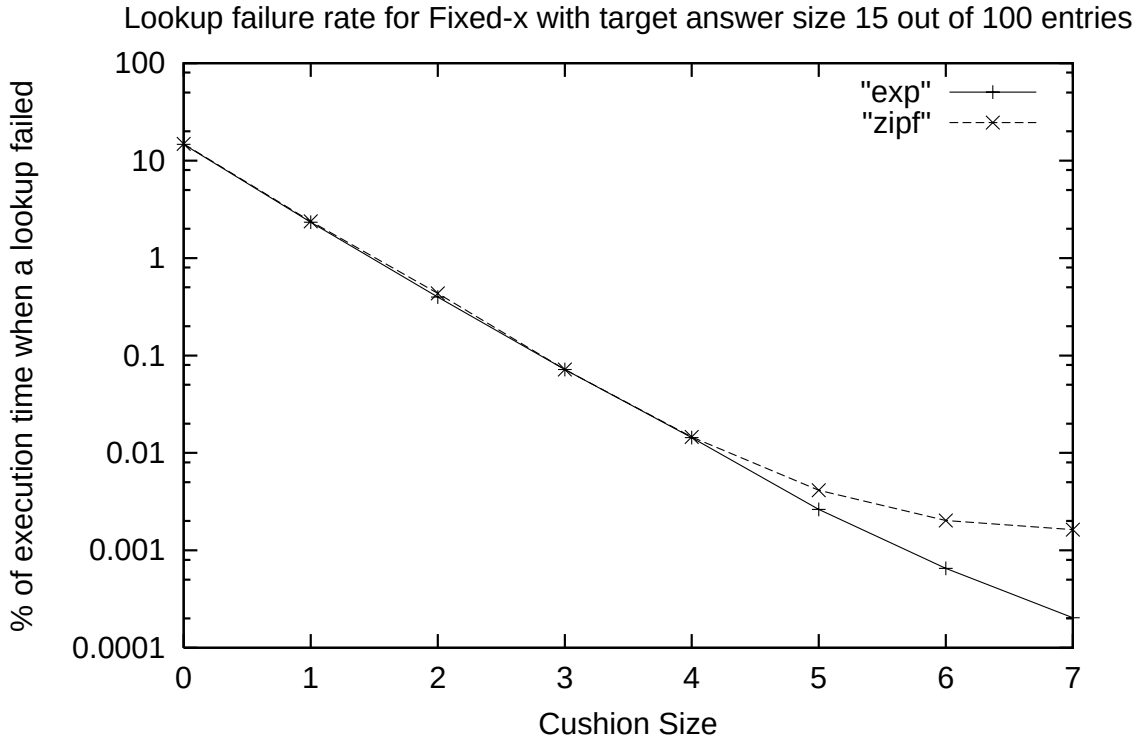

Figure 12: Percentage of time when a client failed to retrieve 15 of the 100 entries in the system using Fixed-x with different cushion factors and different distributions for an entry's lifetime.

adds is smaller.

In practice, a non-zero failure chance is not detrimental to most applications. Since failures are quickly repaired as new *add* events arrive, we can mask the failures by asking the client to retry after a time.

6.3 RandomServer-x and Round-y

In this section, we argue, qualitatively, that RandomServer-x and Round-y are not appropriate when the update rate is high. Consider RandomServer-x versus the simpler Fixed-x. The advantage of RandomServer-x is lower unfairness regarding which entries are returned to clients during *partial_lookups*. However when there are many *delete* operations, the unfairness for RandomServer-x approaches the level of Fixed-x.

Figure 13 shows the effect on the unfairness as more and more *deletes* occur. In this experiment, there are 10 servers and each server holds at most 20 entries out of the expected 100 entries in the system. From the graph, we clearly see that the unfairness deteriorates rapidly and then stabilizes as the number of *deletes* increase. This rise in the unfairness is because deleted entries are generally replaced by newer insertions, thus creating a bias towards the newer entries. Even if we try to actively find replacement immediately after a deletion as mentioned in Section 5.3, the unfairness will not improve because we are just shifting the biasness from the newer entries to the older entries. In fact, finding replacements immediately is worse in

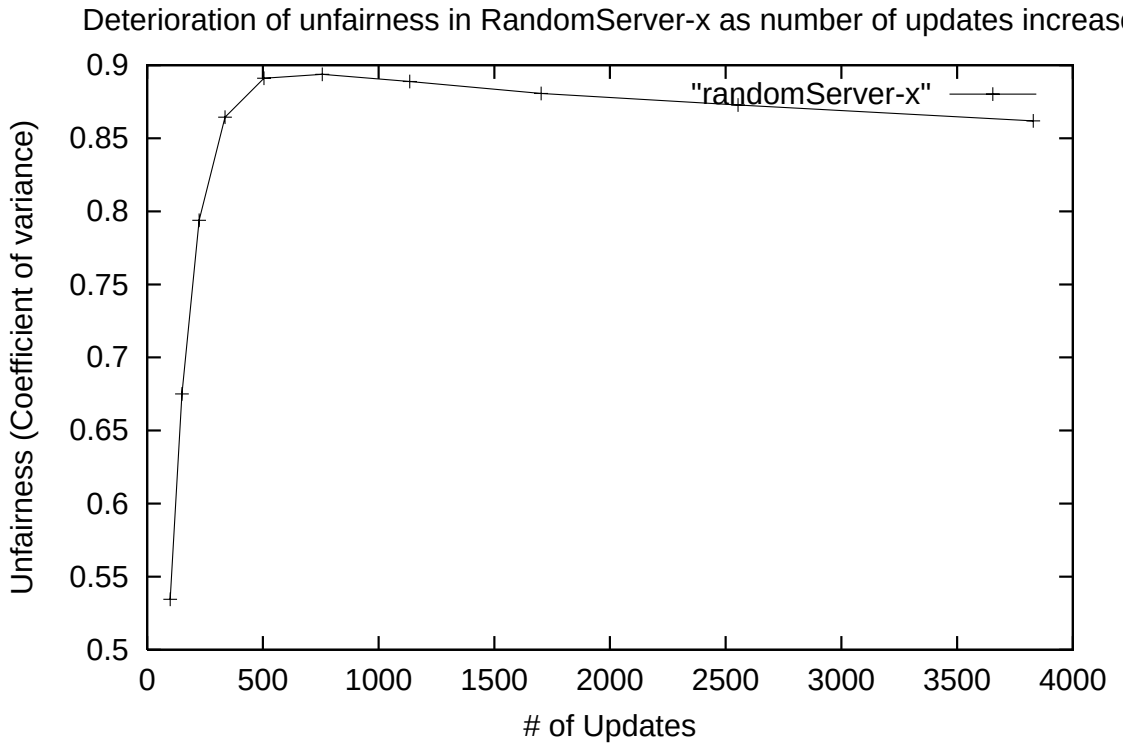

Figure 13: As the number of updates increase, the unfairness of the RandomServer-x increases rapidly

unfairness.

In comparison, Fixed-x, for this same experiment as in Figure 13, has an unfairness of 2. Therefore RandomServer-x is only a factor of 2 better than Fixed-x in unfairness as opposed to an order of magnitude better in the static case. While not getting much better unfairness than Fixed-x, RandomServer-x is also incurring five times more broadcasts than Fixed-x because RandomServer-x does a broadcast on each update compared to one-fifth of the updates for Fixed-x (keeping 20 entries out of 100). Hence Fixed-x is much more attractive under high update rates. The trade-off for using Fixed-x is the smaller coverage. For the same experiment, Fixed-x usually has half as many distinct entries as RandomServer-x.

Similar to RandomServer-x, high update rates cause problems for Round-y. Recall from Section 5.4, Round-y uses server 1 to keep track of two dedicated counters for maintaining the round-robin sequences. Consequently, all updates have to go through server 1 and create a bottleneck effect. If we replicate the dedicated counters to more than one server to reduce the bottleneck, we incur extra overhead in making sure the values for the counters are consistent. In addition to the bottleneck issue, Round-y has to migrate entries when deletions occur. In contrast, Hash-y has no such problem since the hash functions pinpoints where an entry should be stored at or deleted from. There is no migration necessary, and there is no bottleneck. The trade-off in using Hash-y over Round-y is that some client lookups may contact one extra server because some servers may have fewer entries than others.

In short, RandomServer-x and Round-y are much better in a static environment than Fixed-x and Hash-y. If a smaller coverage or a higher client lookup cost can be tolerated, Fixed-x and Hash-y are more efficient because of lower overhead in processing each update.

6.4 Comparing Fixed-x and Hash-y

The previous section has argued for choosing Fixed-x over RandomServer-x and Hash-y over Round-y. This section focuses on quantitatively comparing Fixed-x and Hash-y in terms of the overhead cost for handling updates. For the overhead cost, we count the total number of messages received and processed by all the servers in the system during simulation. Since we are counting processed messages, a broadcast has overhead cost n where n is the number of servers. A point-to-point message has cost 1.

Under this simple cost model of counting processed messages, we study how the strategies behave when clients want a small fraction of all the entries per lookup and when clients want a large fraction. This notion of small or large fraction is captured in the ratio between the client target answer size t and the number of entries in the system h. A small $\frac{t}{h}$ ratio implies clients want a small fraction while a large ratio means a large fraction. Therefore in the experiment, we will vary this $\frac{t}{h}$ ratio and simulate the overhead cost for different ratios.

For the simulation, we fix the target answer size at 40 and vary the number of entries in the system from 100 to 400, thus giving us a range of ratios between 0.1 and 0.4. We also use 10 servers for this experiment. With this setup, we choose $x = 50$ for the Fixed-x strategy. This choice gives a cushion of size 10, ensuring that requests for 40 entries can be satisfied with very high probability. For Hash-y, we cannot just fix the value of y because if we choose a large y such that the lookup cost is close to 1 when the ratio is 0.4, the strategy is unnecessarily penalized when the ratio is 0.1 where a smaller y is sufficient for the task. Similarly, if we choose a small y for the ratio 0.1, the lookup cost is much bigger than 1 (the lookup cost for Fixed-50) when the ratio is 0.4. To resolve this complication, we decided to choose an optimal y for each ratio we study, where the optimal is when the expected number of entries per server is at least the target answer size 40. This choice implies that the lookup cost is always close to 1. In the context of this experiment, we use $y = 1$ when the total number of entries h is 400, $y = 2$ when h is between 200 and 400, $y = 3$ when h is between 133 and 200, and $y = 4$ when h is between 100 and 133.

Figure 14 shows the result of the simulation as we vary the total number of entries in the system from 100 to 400. The x-axis shows the total number of entries, from which we can get the corresponding ratio. The y-axis shows the total number of messages processed by the servers. The curve for Fixed-x is inversely proportional to the total number of entries in the system because the fractions of *add* and *delete* that affect the chosen x entries is decreasing at the same rate. For Hash-y, the overhead cost is purely determined by the choice of y as described in the previous paragraph, hence follows a step curve shape with break points at 133, 200, and 400.

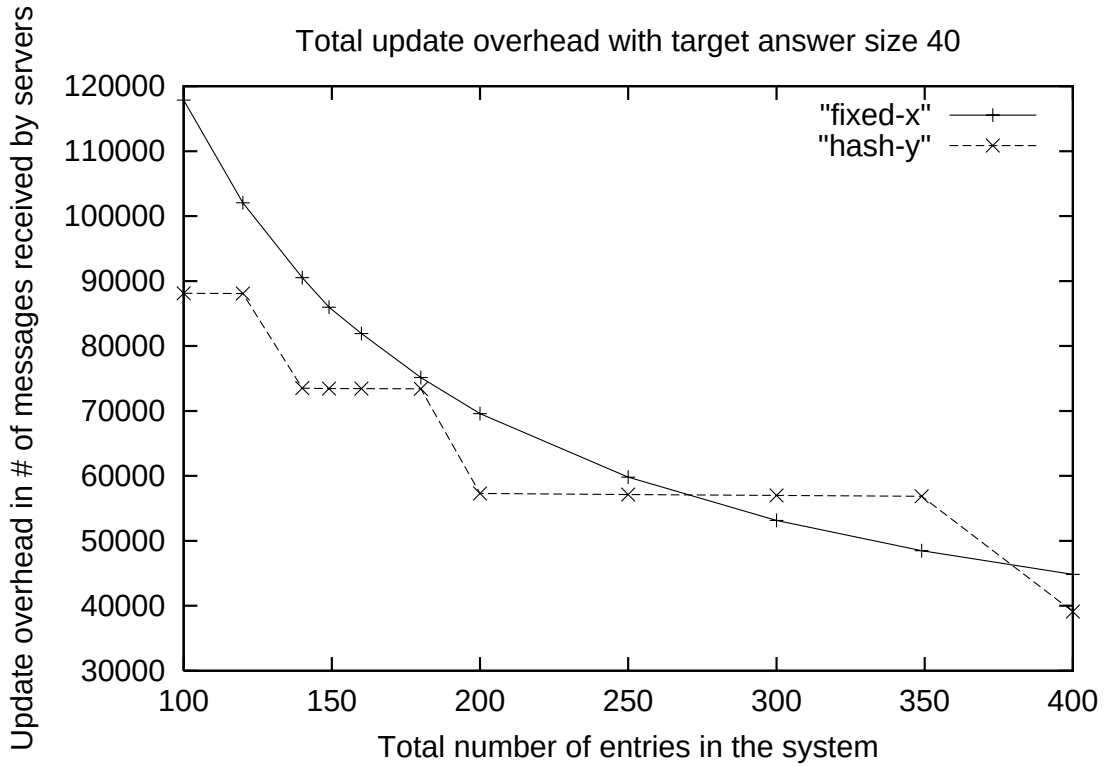

Figure 14: Total update overhead for Fixed-x and Hash-y for different number of total entries in the steady-state.

Notice that the two curves cross each other at a couple points. The crossover points can be captured analytically by computing when the two strategies have the same overhead cost. For Hash-y, we perform approximately $1+y$ operations for each update: one to process the client request and y for storing or deleting y copies in the system (barring collisions among the y hash functions). Therefore, the total cost for Hash-y is $(1+y)U$ where U is the number of updates. For Fixed-x, each update incurs cost 1 to check whether the broadcast is necessary plus the cost n if there is a broadcast, which occurs with probability $\frac{x}{h}$. Hence the total cost is $(1+\frac{x}{h}n)U$. Comparing the two costs yield the equality $\frac{x}{h}n = y$. When $\frac{x}{h}n$ is smaller than y, Fixed-x has lower overhead than Hash-y, and vice versa. This equality test only gives us one crossover point. However recall that we are using different values for y as the number of entries increase. Specifically, $y = \lceil \frac{t \cdot n}{h} \rceil$ where t is the target answer size. Therefore, the equality test is really $\frac{x}{h}n = \lceil \frac{t \cdot n}{h} \rceil$. The ceiling function creates discontinuity in the overhead cost, thus creates multiple crossover points. Note that if we continue to increase the number of entries in the system to beyond 500, we get a third crossover point where Fixed-50 has lower overhead than Hash-1. Beyond the 500 point, Hash-1 always has higher overhead because it has to store each entry at least once, thus cannot take advantage of the small ratio $\frac{t}{h}$ where $\frac{t}{h}$ is less than $\frac{1}{n}$.

As a rule of thumb, if the client target answer size is a small fraction of the total number of entries (typically less than $\frac{1}{n}$), then Fixed-x will have less update overhead. Another consideration in deciding

between Fixed-x and Hash-y is the lookup cost described in Section 4.2. Since Hash-y has higher lookup cost, the ratio between lookups and updates will also be a factor in choosing Fixed-x or Hash-y.

7 Other Variations

In this paper, we studied different strategies for partial lookups under two major assumptions: a client is satisfied with any t entries when performing *partial_lookup*(t) and all servers are accessible to each client. These two assumptions are very restrictive. Most practical uses of a partial lookup service will want to relax one or both assumptions. In this section, we discuss which variation arises when relaxing the assumptions.

7.1 Clients with Preferences

The first variation is to attach a client preference to which entries should be returned during a lookup instead of assuming any entry is okay. For example, when looking up a list of file sharing servers on the network, clients prefer those servers that has high bandwidth and low latency. We define the variation formally as for each client i, there exists a cost function C_i defined over all the entries $\{v_1, v_2, \ldots, v_k\}$. We want a partial lookup service such that *partial_lookup*(t) returns $R \subset \{v_1, v_2, \ldots, v_k\}$ to client i where $|R| = t$ and $C_i(u) \geq C_j(w)$ for all $u \in R$ and $w \in \{v_1, v_2, \ldots, v_k\} - R$. In other words, we return the t best possible answer to the client. This variation is fairly easy if the cost function C_i is known for each client i and in advance. But in reality, the cost function C_i is unknown and changes with time.

7.2 Servers with Limited Reachability

A second type of variation assumes not all servers are reachable to a client. For instance, lookups in a peer-to-peer based network, such as Gnutella, could be adhoc and rely on mechanisms like flooding. In those application-level overlays, a client can only reach nodes within a few hops of itself. Therefore the problem becomes making sure the data is placed on a set of servers such that for each client i there exists a server s where the distance between i and s is bounded by a hop count d, i.e. client i can reach server s. A more sophisticated study can measure the overhead tradeoff in deciding the best hop count limit d. This tradeoff exists because small d reduces lookup costs while increases update costs at the servers.

8 Related Work

The most well known lookup service is the Domain Name Service[2]. More recently, peer-to-peer (P2P) systems [4, 3, 7, 6] have dominated the discussion of lookup services in locating shared files. Some P2P systems like Naspter[4] and its variants are built on top of a centralized lookup service to enable file sharing between clients. These systems exhibit the *partial lookup* characteristic in that when clients search for a

Strategy	Storage		Coverage	Fault Tolerance	Fairness		Lookup Cost	Update Overhead	
	Few entries	Many entries			Few updates	Many updates		Small target size	Large target size
Fixed-x	⋆⋆⋆⋆	⋆⋆⋆⋆	⋆	⋆⋆⋆⋆	⋆	⋆	⋆⋆⋆⋆	⋆⋆⋆⋆	⋆⋆
RandomServer-x	⋆⋆⋆⋆	⋆⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆	⋆⋆⋆	⋆⋆	⋆⋆
Round-y	⋆⋆⋆⋆	⋆⋆	⋆⋆⋆⋆	⋆⋆⋆	⋆⋆⋆⋆	⋆⋆⋆⋆	⋆⋆⋆⋆	⋆	⋆
Hash-y	⋆⋆⋆⋆	⋆⋆	⋆⋆⋆⋆	⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆⋆	⋆⋆⋆	⋆⋆⋆⋆

Table 2: Informal summary of the strategies with respect to our proposed metrics. (More stars is better.)

file, only the first 50 or 100 hits are returned. However, the servers usually keep track of a large amount of information rather than discarding extra information.

Other P2P systems like Chord[7], CAN[6], Gnutella[1], and Morpheus[3] provide a lookup service by building an application-level overlay network and route lookup requests to the appropriate destinations. In Chord and CAN, a key and its associated entries are stored on one server specified by the hash value of the key, thus their approach is based on partitioning the key space rather than partitioning the entries of a key as we suggest for partial lookups. In Gnutella and Morpheus, the entries for a key are randomly scattered across the entire overlay network rather than on a group of known servers, hence a client is forced to flood the network to perform a lookup.

Aside from the P2P area, the partial lookup idea of storing just enough entries of a key at each server also resembles vertical partitioning of relations in distributed databases[5]. In vertical partitioning, a relation table is broken up by attributes and only a subset of the attributes of the table are stored at individual servers. However, the main focuses of various partitioning schemes are completeness and reconstruction, which are not concerns for partial lookup strategies.

9 Conclusion

In this paper, we demonstrated a *partial lookup service* that maintains fewer entries on a server provides significant performance improvements over the traditional full replication or hashing-based lookup services. Specifically, storage cost for partial lookup services are lower, which in turn leads to lower overhead in processing updates. Furthermore, partial lookup services are insensitive to the popular key or hot-spot problems which plague traditional hashing-based lookup services. These two advantages make partial lookup services very attractive to applications with high workload. Our paper also showed that keeping fewer entries at servers does not adversely affect most metrics that are important to users such as client lookup cost, coverage, and fairness. Table 2 informally summarizes the tradeoffs between different strategies, with respect to our proposed metrics. In the table, four stars imply the strategy is very suitable, while one star implies the strategy performs poorly. Note that no strategy is the best in all situations.

Acknowledgments

We thank Rajeev Motwani for various suggestions.

References

[1] Gnutella. Website http://gnutella.wego.com.

[2] P. Mockapetris and K. J. Dunlap. Development of the domain name system. In *Procoeedings of ACM SIGCOMM*, pages 123–133, Stanford, CA, 1988.

[3] Morpheus. Website http://www.musiccity.com.

[4] Napster. Website http://www.napster.com.

[5] M. T. Oszu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, pages 149–160, San Diego, August 2001.

[7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, pages 160–177, San Diego, August 2001.

[8] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. on Math. Software*, 11(1):37–57, March 1985.

A Heuristic for Computing Fault Tolerance

In Section 4.4, we defined the fault tolerance of a strategy as the minimum number of server failures before a client partial lookup of size t failed. For discussing how to compute the fault tolerance, we use the following notations. Let e be an entry. Let V_S be the set of entries stored on server S. Let f_e be the number of servers having entry e. The heuristic is then

1. For each operational server, compute $X_S = \sum_{e \in V_S} \frac{1}{f_e}$.
2. Find the highest X_S. Make server S faulty.
3. If the remaining servers still has a coverage greater than t, update f_e to reflect the failure of server S and goto step 1.

Basically, this heuristic computes the importance of each server (the X_S score in step 1) and greedily removes the server that is the most important. The importance of a server is determined by which entries it has. For instance, a server has high importance if it stores an entry that no other servers have. On the other hand, if a server has an entry that are stored at many other servers, then it is not very important. The commonness of an entry is captured in the $\frac{1}{f_e}$ factor for computing X_S.