A Decentralized, Adaptive Replica Location Mechanism

Matei Ripeanu [1]
[1] Computer Science Department,
The University of Chicago
matei@cs.uchicago.edu

Ian Foster [1,2]
[2] Mathematics and Computer Science Division
Argonne National Laboratory
foster@cs.uchicago.edu

Abstract

We describe a decentralized, adaptive mechanism for replica location in wide-area distributed systems. Unlike traditional, hierarchical (e.g, DNS) and more recent (e.g., CAN, Chord, Gnutella) distributed search and indexing schemes, nodes in our location mechanism do not route queries, instead, they organize into an overlay network and distribute location information. We contend that this approach works well in environments where replica location queries are prevalent but the dynamic component of the system (e.g., node and network failures, replica add/delete operations) cannot be neglected.

We argue that a replica location mechanism that combines probabilistic representations of replica location information with soft-state protocols and a flat overlay network of nodes brings important benefits: genuine decentralization, low query latency, and flexibility to introduce adaptive communication schedules.

We support these claims in two ways. First, we provide a rough resource consumption evaluation: we show that, for environments similar to those encountered in large scientific data analysis projects, generated network traffic is limited and, more importantly, is comparable to the traffic generated by a request routing scheme. Second, we provide encouraging performance data from a prototype implementation.

1. Introduction

Wide-area distributed systems often replicate entities in order to improve reliability, access latency, or availability. As a result, these systems need mechanisms for locating replicas, i.e., mechanisms for mapping a *replica identifier* to the one or more *replica locations*. A number of distributed systems require such location mechanisms. For example, nodes participating in a cooperative Internet proxy cache [1-3] must locate a cached Web page given its URL, while nodes in a distributed object system need to find an instantiation of an object given an object handle [4]. These systems need to scale to millions of replicated entities and hundreds of sites at least, all while operating in dynamic environments where network outages or site failures are to be expected.

A recent paper [5] introduces the replica location problem in a Data Grid [6, 7]: given a unique *logical* identifier for desired data content, we need a mechanism to determine the *physical* locations of one or more copies of this content. A slightly different semantic makes this problem different from cooperative Internet caching: a replica location mechanism for Data Grids might have to serve requests for *many* or *all* replicas corresponding to a given logical identifier. Requests for *many* (or N) replicas might be generated by a brokering service searching for a replica with suitable performance characteristics. Requests for *all* replicas might be generated, for example, by a service implementing file system functionalities. However, in a distributed, asynchronous environment, where nodes leave the system without warning, it is impossible to provide a completely consistent system view [8, 9] - and thus impossible to serve "all replicas" requests reliably in a decentralized manner.

Giggle [5] contends that the performance of the overall system benefits from relaxed consistency semantics at lower system levels, and that stronger guarantees can be added within the limited set of high-level components requiring them. (For example, a versioning mechanism can be used to handle file updates.) Therefore, a replica location service designed in this context can adopt inconsistency as a 'modus-operandi' and make tradeoffs between inconsistency levels and operational costs.

In this article, we present a probabilistic approach to the replica location problem and show that relaxed consistency constraints allow for a decentralized, low-latency, low-overhead solution. In contrast to traditional hierarchical, (e.g., DNS) and recent distributed search and indexing schemes (e.g., CAN, Chord, Gnutella), nodes in our location mechanism do not route queries but organize into an overlay network and distribute location information. Each node that participates in the distribution network builds, in time, a view of the whole system and can answer queries locally without forwarding requests. This straightforward design brings benefits (e.g., reduced query latency, load sharing,

robustness, etc.) in environments such as GriPhyn's [10] large-scale data analysis projects. These environments are characterized by high query rates and significant, albeit lower, rates of replica creation and deletion, and node and network failures. However, as an environment becomes more dynamic and replica create/delete operations start to prevail over queries, query-routing schemes might sa better tradeoff.

Our replica location system design integrates three techniques: a flat overlay network of nodes (to obtain genuine decentralization and resilience when facing network and node failures), probabilistic representations of replica location information (to achieve important space and bandwidth reductions), and soft-state protocols (to decouple node state and achieve robustness). These are well-known techniques; the merit of this paper is to put them together in a flexible design and investigate emerging synergies.

The rest of this paper is organized as follows. The next section details the replica location problem requirements and the terminology we use. Section 3 briefly presents related work while Section 4 introduces the three techniques we use to build the location mechanism. Section 5 presents the replica location service as a whole and Section 6 documents our experience to date in building this service. We conclude in Section 7.

2. Replica location problem

In this section we briefly introduce the terminology (mainly adopted from [5]) used throughout this paper, as well as functional and performance requirements for a replica location service.

2.1. Terminology

A *logical file name (LFN)* is a unique logical identifier for desired data content. The location service must identify one or more physical copies (replicas) of the logical file. Each physical copy is identified by a *physical file name (PFN),* which specifies its location on a storage site.

A number of *storage sites (SS)* collaborate to share their storage capabilities to all users. A *replica location node (RLN)* aggregates LFN to PFN mappings from one or more SSs and collaborates with other RLNs to build a distributed catalog of LFN mappings.

RLNs offer both a query interface to clients and a registration interface that SSs can use enlist PFN to LFN mappings for files stored locally. RNLs also organize into a search network to allow remote searches. Nodes in this network distribute compressed information on the set of LFN mappings stored locally in the form of *node digests*.

2.2. Functional requirements

The main task of the location service is to find a specified number of PFNs given a LFN. Requests might contain multiple LFNs, and thus the location system should also handle efficiently requests for PFNs associated with ad-hoc sets of LFNs.

Below, we briefly enumerate other functional requirements (documented extensively in [5]):

- *Autonomy*: Failure of various components (RLNs, network outages) should not prevent the remaining healthy parts of the system operate correctly.
- *Best-effort consistency* [11]: RLNs might have incomplete and/or outdated views of the system. The system tries to be eventually consistent, but only does the best it can without impeding performance.
- *Adaptiveness*: Nodes get overloaded, networks get congested, sometimes users get frantic and all submit queries at the same time. Overall system performance should degrade smoothly when facing bursts in demand or the quality of resources used decays.

2.3. Scale requirements

The total numbers of files/replicas, the numbers of storage sites and their geographical distribution, as well as aggregated query and update rates ultimately determine the design of the location mechanism. We use requirements for high energy physics (HEP) data-analysis projects [12, 13] as a realistic starting point.

The HEP community estimates an initial target of 500 million replicas to be kept track of by our system. (To put things into perspective, note that Google indexes 2 billion documents, so we are on approximately the same scale as the Web). Query and update rates estimates are more likely to change but the current estimates are: aggregate query rates 100,000 queries/sec (peak) and 10,000 queries/sec (average), with update rates one order of magnitude lower.

The number of participating organizations and their need to 'own' individual RLN for latency-hiding and/or security or administrative reasons determines the total number of RLNs. HEP projects estimate around two hundred RLNs, but this number could easily grow larger.

3. Related work

Distributed search and indexing mechanisms have been a topic of extensive research. The most relevant to our current work are CAN [14], Chord [15], Tapestry [16], Past [17], and Gnutella [18], in which queries are routed through an overlay network. CAN, Chord, Tapestry, and Past build structured, search-efficient indexing structures that provide good scalability and

search performance, although increasing the cost of file and node insertion and removal. Gnutella does not use indexing mechanisms; its relatively good search performance (as measured in number of hops) is offset by intensive network usage.

Compared to the above location schemes based on query routing and forwarding, our system, based on location information dissemination, makes an explicit tradeoff: reduced query latency for increased memory requirements at nodes. Additionally, one scheme or another will generate more network traffic depending on the ratio of query rates vs. system dynamic behavior (replica add and delete rates).

Structured query routing schemes, similar to CAN or Chord, are unsuitable mechanisms for our replica location service for an additional reason: site autonomy requires that a RLN and its associated local SSs serve local users even when network outages disconnect the site from the external Internet.

Cooperative Internet proxy-caches [1-3, 19] offer comparable functionality to our location mechanism. A cooperative proxy-cache receives requests for an URL and locates *one* cached replica at other collaborating proxies (unlike our replica location mechanism that might locate a specified number of replicas). Hierarchical caching in proxy servers has been extensively analyzed [2]. Two distinct solutions that do not use hierarchies are Summary Cache and the Cache Array Routing Protocol. Summary Cache [19] uses Bloom filters (dubbed "cache summaries"): each cache periodically broadcasts its summary to all members of the distributed cache. We use a similar scheme but, unlike nodes in a Summary Cache that uses fixed communication schedules, RLNs use an adaptive soft-start protocol to distribute node summaries (or digests). The Cache Array Routing Protocol [20] uses consistent hashing [21, 22] to partition the name space and route requests. Site autonomy requirements prevent us from using a similar solution in this context.

Although apparently similar, *Content Distribution Networks* (CDNs) [23] face a different problem: in this case individual nodes are not autonomous, a single authority controls replication decisions and the request forwarding mechanism.

4. Building blocks

This section details the three techniques on which our replica location service design is built: (1) an overlay network to obtain decentralization and reliability, (2) compressed probabilistic representation of sets that bring important space and bandwidth savings, and (3) soft-state mechanisms to decouple node states.

4.1. Overlay network

Overlay networks have been used to implement features unavailable at lower network layers, such as multicast [24, 25] and security [26, 27]. A recent project [28] shows that aggressively optimized custom routing over an overlay network can significantly improve performance and availability of network paths between Internet hosts. The proven versatility and achieved scale of P2P systems based on overlay networks (e.g., Gnutella, FastTrack) provide another argument for using this technique.

An additional argument for an overlay network is reduced security overhead. In a traditional system, nodes might need to authenticate at each message exchange. In an overlay, nodes need to authenticate only when entering the overlay and establishing a connection. Then, if necessary, all traffic can be secured through an application-level encryption scheme.

RLNs organize into an overlay network and distribute compact, probabilistic representations of the set of LFNs registered locally.

4.2. Compact set representation with Bloom filters

Bloom filters [29, 30] are compact data structures used for probabilistic representation of a set in order to support membership queries ("Is element *x* in set *Y*?"). The cost of this compact representation is a small rate of *false positives*: the structure sometimes incorrectly recognizes an element as a set member. We describe Bloom filters in

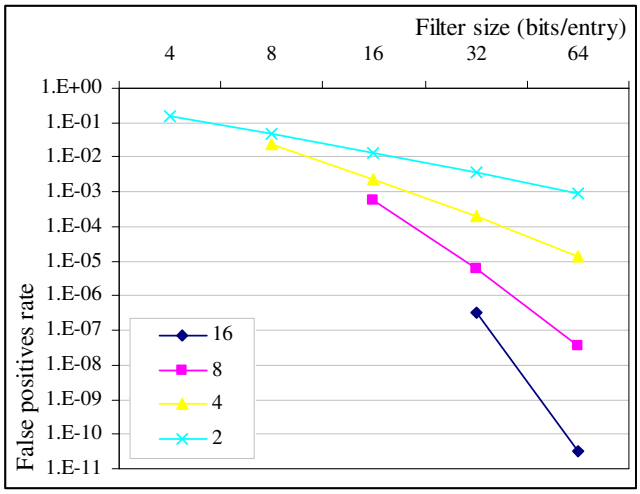

Figure 1: Bloom filter's false positive rate vs. the size of the filter. The plot presents (analytically obtained data for) the variation of false positive rates with size of the filter (bits/entry) for 2, 4, 8 or 16 perfect hash functions used. Note that filter size can be traded for a higher false positive rate.

detail in the Appendix. However, to give a taste of achievable compression rates, we note that a set of N elements can be represented using $2N$ bytes with less than 0.1% false positive rates and lookup time of about 100μs.

In the rest of this paper, we borrow a term from the cooperative Internet proxy caching community [1] and use the term '*node digest'* for the Bloom filter compressed representation of the set of LFNs registered at a RLN. Initially, RLNs multicast their digests. Updates are handled via a combination of periodic multicasts of both complete digests and shorter update messages. An additional useful feature is that, when a node determines that the available bandwidth is insufficient to keep up with offered traffic, or when memory at a node is scarce, digests can be further compressed (although with precision loss) at intermediary nodes within the multicast network (Figure 1). This strategy offers an elegant mechanism for smooth QoS degradation when facing bursts in demand.

In summary we use Bloom filters for their efficient compression, low query overhead, incremental update ability, and tunable size vs. false positive rates.

4.3. Soft-state updates

Generally in soft-state mechanisms [31, 32] a state producer sends its state to one or more receivers over a (lossy) communication channel. Receivers maintain copies of this state together with associated timeouts. State is deleted if not refreshed within a timeout interval. This mechanism is used in a number of Internet (RSVP, RIP) and Grid protocols (MDS-2) and works well in practice.

Soft-state mechanisms have two main advantages:

- *'Eventual' state*. As information introduced by failed nodes is eventually eliminated through timeouts, there is no need for explicit failure detection and state removal. Similarly, new nodes do not set up an explicit state-gathering protocol when joining; state simply flows through the network and a long-lived node eventually collects all available state. (One could argue that making full state available only in time creates an incentive for nodes to participate longer in the system).
- *Adaptiveness*. Traditionally, soft state systems have used fixed, empirically determined send rates. However, state producers can also obey more complex policies that save network bandwidth (e.g. "generate an update each hour and each time the set of files stored locally has changed by 10%, but make sure not to employ more than 1% of the available network bandwidth"). In this case, receivers have to estimate timeouts dynamically [32]. In order to further reduce generated network traffic the soft-state mechanism could combine full state announcements with incremental updates.

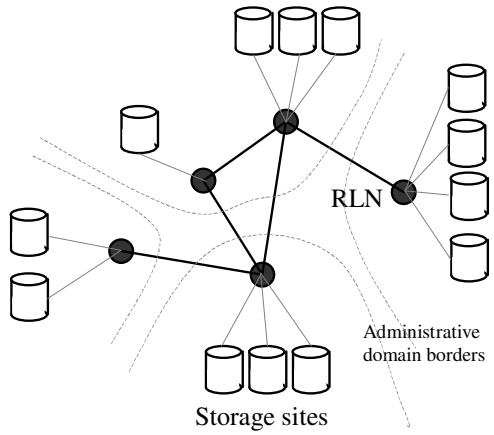

Figure 2: Replica location service organization. Storage sites publish LFN to PFN mappings for files stored locally to Replica Location Nodes (RLNs). RLNs (black circles in the figure) store these mappings and compute *digests*: compressed representations for the set of LFNs stored at local SSs. RLNs organize into an overlay network and distribute digests using a soft-state protocol.

5. Assembling the pieces

In a nutshell, RLNs organize into a flat overlay network and distribute their digests using a soft-state mechanism (Figure 2).

A broader description of the location mechanism starts with the three functionalities offered by a RLN:

- *Storage site (SS) registration*. An RLN allows SS(s) to register/delete pairs of (LFN, PFN) with the replica location service. The details of the SS-RLN interaction are outside the scope of this paper. However, we note that a soft-state mechanism can be used here as well.
- *Querying*. When receiving a query (an LFN for which the associated PFN is required), a RLN checks first to see if a LFN mapping is stored locally. If it is, then the RLN returns the associated PFN(s). Otherwise, it checks the digests available locally to see which remote RLN might store mappings for the requested LFN (remember digests allow for false positives). If one is found, the local RLN contacts the remote RLN to obtain the associated PFN(s) that are subsequently sent back to the client. Alternatively the remote RLN address(es) could be returned, and the querying client could contact each of them and obtain PFN(s). The latter scheme takes some burden off the RLN, but prevents caching at the RLN level.

  This mechanism extends intuitively to requests for many (exactly N) or all PFNs associated with a given LFN. It also handles requests for PFNs associated with sets of LFNs.
- *Digest distribution network*. This component is the core of the location system and its performance is key to

overall performance. RLNs distribute digests through the overlay using the soft-state mechanism. Nodes have hard upper limits on the network traffic that they generate into the overlay and soft lower (in)consistency limits (e.g., "generate a soft-state update each time a X% of the content has been modified with at most Y% inaccuracy. Generated traffic should not be above Z bps").

We use a bootstrap mechanism similar to that used in Gnutellas: a new node obtains, through an out-of-band channel, the location of a node already in the network. It then connects to the network and uses the information flowing in the overlay to discover other nodes and create new connections if necessary.

Undoubtedly, the topology of the overlay network determines its efficiency in using the underlying networking infrastructure [18]. Creating a self-organizing overlay that matches the underlying network topology is a challenging topic that we are currently investigating.

5.1. Overall benefits

The mix of techniques that we use brings important benefits:

- *Low query latency*. Looking up one PFN implies, with high probability, at most two calls across the network. This cost compares favorably with other distributed index systems (CAN, Tapestry) that route queries and thus incur larger query latencies.
- *Adaptive*. When the system is overloaded, source nodes reduce their update rate while intermediary nodes may combine update messages.
- *Robust, high availability*. There is no single point of failure in the system. As long as the overlay network remains connected, node failures do not prevent the remaining parts of the system from operating correctly.
- *Manageability*. Manual configuration and maintenance of even medium-sized sets of resources becomes quickly a daunting task. Our solution based on a self-configuring overlay network reduces administration overhead to a minimum.
- *High throughput*. Searches for sets of files (as opposed to searches for one file at a time) are likely to be the norm. In this case query routing protocols do not help much as a multiple-LFN query generates a large number of individual queries to be routed in the network. Our system benefits from the (approximate) global image available at each node and can process requests in batches, taking advantage of request locality characteristics (i.e., a single request issued for LFNs that are mapped on the same remote RLN).

5.2. Rough resource consumption evaluation

Our decentralized system is designed to provide low latency, robustness and high-availability in an unreliable, wide-area environment. There are two types of resources it consumes 'at large': system memory and network bandwidth.

Each node maintains a compressed image of the whole system: in the limit, one digest for each node in the system. Assuming 500 million replicas and digests sized at 2 bytes per entry, one node needs 1 GB to represent the whole system state with a false positive rate of 0.05%.

We provide a crude evaluation for the generated traffic when using the maximal requirements outlined in Section 2. Assume 500 RLNs, each maintaining information about 1 million replicas. Assume an average add/drop rate of 2 replica/sec at each node and a 1% change trigger to generate a soft-state update message (i.e., an update message is generated if 1M * 1% = 10,000 replicas have been added or deleted). Assume further that one in ten generated messages describes the state completely; the others are incremental updates. In this case a RLN has to support digest traffic of about 25 kBps for each of its overlay links. Generating one complete digest for every 100 updates reduces the traffic to 4 kBps. Assuming 20 queries/sec per node and 200 bytes per query, a further 8 kBps are necessary to support local query traffic. Thus, we see that a mechanism based on query forwarding will not fare significantly better: assuming an average of 10 forwarding hops, the generated traffic is about the same.

We stress that we do not claim that the approach described here scales better than current distributed hash table solutions (CAN, Chord, Tapestry, etc.) based on structured overlays. Rather, we contend that for a realistic class of problems this approach generates comparable (if not less) traffic and allows for significantly lower query latencies. The class of problems for which our approach works well is precisely those envisaged by GriPhyN's data intensive projects: hundreds to thousands of RLNs and query rates one order of magnitude greater than replica add/delete rates. In addition, we note that our use of unstructured overlays should allow our solution to cope better with node and network failures.

6. Implementation and performance data

We currently use Python to implement a proof-of-concept prototype of the replica location service described here. We have tested the main components of the system separately and on small-scale deployments. We present below our preliminary performance results.

- *Bloom Filters*. We have experimented with various configurations of Bloom filters and obtained

compression and false positive rates close to those predicted by theory (see Appendix A for details), and lookup times in the order of 100μs per lookup.

- *Replica Location Node (RLN)*. A single client querying, over a LAN, an isolated RLN storing 10 million LFN to PFN mappings achieved over 300 queries per second (without authentication); with multiple clients, throughput to the RLN peaked at 3,000 queries per second. While we still have to evaluate the cost of authentication, these initial results are encouraging.
- *Overlay Network*. We have experimented with small-scale overlays (24 RLNs, 50 million replicas) with manually configured topologies. In this configuration, our system achieves 2,000 query/sec rates concurrently with 1,200 updates/sec. While these setups do not test the reliability of our system, or its ability to adapt, they do provide an initial (and encouraging) idea of achievable overall performance. We stress that to date we have been more concerned with implementing RLN functionality than optimizing for performance.

Currently we are adding authentication (using Grid Security Infrastructure [33]) and self-configuring overlays, and experimenting with various configurations of the soft-state update mechanism.

7. Summary

This paper describes a decentralized, adaptive mechanism for replica location in wide-area distributed systems. Unlike traditional, hierarchical (e.g, DNS) and more recent (e.g., CAN, Chord, Gnutella) distributed search and indexing schemes, nodes in our location mechanism do not route queries, instead, they organize into an overlay network and distribute location information. We contend that this approach works well in environments where replica location queries are prevalent but the dynamic component of the system (e.g., node and network failures, replica add/delete operations) cannot be neglected.

This paper argues that a replica location mechanism that combines probabilistic representations of replica location information with soft-state protocols and a flat overlay network of nodes brings important benefits: genuine decentralization, resilience when facing network and node failures, low query latency, and flexibility to introduce adaptive communication schedules.

We support these claims with two arguments. First, we provide a rough resource consumption evaluation: we show that, for environments similar to GriPhyN's large data-analysis projects, generated network traffic is limited and, more importantly, is comparable to the traffic generated by a request routing scheme. Second, we provide encouraging early performance data from our prototype implementation.

We are currently investigating hybrid approaches that combine query forwarding with information dissemination in self-organizing, unstructured overlays. One possible approach [34] benefits from data sharing patterns in scientific collaborations and organizes the overlay network so as follow the small-world sharing patterns that emerge in these collaborations. Nodes within a small-world (a cluster) use an information dissemination mechanism similar to that discussed in this paper to serve requests for files available within the cluster. Requests for other files are forwarded to different clusters.

8. Appendix: Bloom filters

Bloom filters [29] are compact data structures for probabilistic representation of a set in order to support membership queries (i.e., queries that ask: "Is element *x* in set *Y*?"). This compact representation is achieved at the cost of a small rate of *false positives* in membership queries; that is, queries might incorrectly recognize an element as a set member.

8.1. Usage

Since their introduction in [29], Bloom filters have seen various uses:

- *Web cache sharing*. Collaborating Web caches use Bloom filters (called "cache digests" or "cache summaries") as compact representations for the local set of cached files. Each cache periodically broadcasts its summary to all other members of the distributed cache. Using all summaries received, a cache node has a (partially outdated, partially wrong) global image about the set of files stored in the aggregated cache.
- *Query filtering and routing* ([35-37]) The Secure Discovery Service [35] subsystem of the Ninja project [38] organizes service providers in a hierarchy. Bloom filters are used as summaries for the set of services offered by a node. Summaries are sent upwards in the hierarchy and aggregated. A query is a description for a specific service, also represented as a Bloom filter. Thus, when a member node of the hierarchy generates/receives a query, it has enough information at hand to decide where to forward the query: downward, to one of its descendants (if a solution to the query is present in the filter for the corresponding node), or upward, toward its parent (otherwise).
- *Compact representation of a differential file* [39]. A differential file contains a batch of database records to be updated. For performance reasons, the database is updated only periodically (e.g., at midnight) or when the differential file grows above a certain threshold. However, in order to preserve integrity, each reference/query to the database has to access the

differential file to see if a particular record is scheduled to be updated. To speed up this process, with little memory and computational overhead, the differential file is represented as a Bloom filter.

- *Free text searching* [40]. The set of words that appear in a text is succinctly represented using a Bloom filter

8.2. Constructing Bloom filters

Consider a set $A = \{a_1, a_2, ..., a_n\}$ of n elements. Bloom filters describe membership information of A using a bit vector V of length m. For this, k hash functions, $h_1, h_2, ..., h_k$ with $h_i : X \rightarrow \{1..m\}$, are used. The following procedure builds an m bits Bloom filter, corresponding to a set A using $h_1, h_2, ..., h_k$ hash functions:

```
Procedure BloomFilter (set A,
                hash_functions_set h, int m)
  returns filter
  filter = new m bit vect. initialized to 0
  foreach a_i in A:
       foreach hash function h_j:
            filter[h_j(a_i)] = 1
       end foreach
  end foreach
return filter
```

Therefore, if a_i is member of a set A, in the resulting Bloom filter V all bits obtained corresponding to the hashed values of a_i are set to 1. Testing for membership of an element *elm* is equivalent to testing that all corresponding bits of V are set:

```
Procedure MembershipTest (elm,
           filter, hash_functions h)
 returns yes/no
 foreach hash function h_j:
  if filter[h_j(elm)] != 1 return No
 end foreach
return Yes
```

An important feature of the algorithm is that filters can be built incrementally: as new elements are added to a set the corresponding positions are computed through the hash functions and bits are set in the filter. Moreover, the filter expressing the reunion of two sets is simply computed as the bit-wise OR applied over the two corresponding Bloom filters.

8.3. Bloom filters: the math

In this section we follow the same lines of reasoning as [41]. One prominent feature of Bloom filters is that there is a clear tradeoff between the size of the filter and the rate of false positives. Observe that after inserting n keys into a filter of size m using k hash functions, the probability that a particular bit is still 0 is:

$$p_0 = \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - e^{-\frac{kn}{m}}. \qquad (1)$$

Note that we assume perfect hash functions that spread the elements of A evenly throughout the space $\{1..m\}$. In practice, good results have been achieved using MD5 and other hash functions [40].

Hence, the probability of a false positive (the probability that all k bits have been previously set) is:

$$p_{err} = (1 - p_0)^k = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \qquad (2)$$

In (2) p_{err} is minimized for $k = \frac{m}{n} \ln 2$ hash functions. In practice however, a smaller number of hash functions are used. The reason is that the computational overhead of each hash additional function is constant while the incremental benefit of adding a new hash function decreases after a certain threshold (Figure 3).

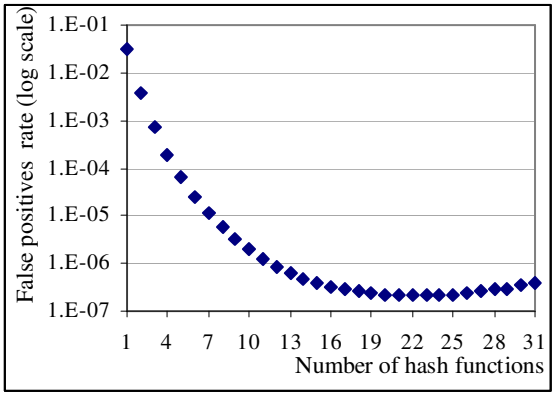

Figure 3: False positive rate as a function of the number of hash functions used. The size of the Bloom filter is 32 bits per entry (m/n=32). In this case using 22 hash functions minimizes the false positive rate. Note however that adding a hash function does not significantly decrease the error rate when more than 10 hashes are already used

The central formula for engineering Bloom filters, (2), helps us computing minimal memory requirements (filter size) and number of hash functions given the maximum acceptable false positive rate and number of elements in the set (as we detail in Figure 4).

$$\frac{m}{n} = \frac{-k}{\ln\left(1 - e^{\frac{\ln p_{err}}{k}}\right)} \text{ (bits per entry)} \qquad (3)$$

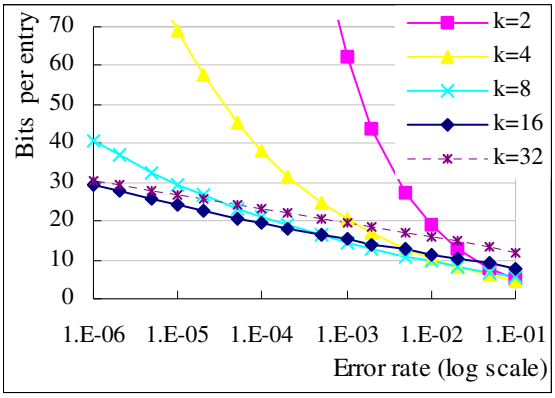

Figure 4: Size of Bloom filter (bits/entry) as a function of the error rate desired. Different lines represent different numbers of hash keys used. Note that, for the error rates considered, using 32 keys does not bring significant benefits over using only 8 keys.

To summarize: Bloom filters are compact data structures for probabilistic representation of a set in order to support membership queries. The main design tradeoffs are the number of hash functions used (driving the computational overhead), the size of the filter and the error (collision) rate. Formula (2) is the main formula for tuning parameters according to application requirements.

8.4. Compressed Bloom filters

Applications that use Bloom filters may need to communicate them across a network. In this case, besides the three performance metrics we have seen so far: (1) the computational overhead to look up a value (related to the number of hash functions used), (2) the size of the filter in memory, and (3) the error rate, a fourth metric can be used: the size of the filter transmitted across the network. Mitzenmacher shows that compressing Bloom filters might lead to significant bandwidth savings at the cost of higher memory requirements (larger uncompressed filters) and some additional computation time to compress the filter that is sent across the network [30].

Acknowledgements

We are grateful to Ann Chervernak, Carl Kesselman, Wolfgang Hoschek, Peter Kunszt, Adriana Iamnitchi, Heinz Stockinger, Kurt Stockinger, and Brian Tierney for discussions and support. This work was supported by the National Science Foundation under contract ITR-0086044 (GriPhyN).

References

[1] "Squid Web Proxy Cache", http://www.squid-cache.org/.

[2] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy, "On the scale and performance of cooperative Web proxy caching," presented at 17th ACM Symposium on Operating Systems Principles (SOPS'99), Kiawah Island Resort, SC, USA, 1999.

[3] P. S. Yu and E. A. MacNair, "Performance study of a collaborative method for hierarchical caching in proxy servers," presented at 7th International World Wide Web Conference (WWW7), 1998.

[4] M. v. Steen, P. Homburg, and A. S. Tanenbaum, "Globe: A Wide-Area Distributed System," *IEEE Concurrency*, vol. 7, pp. 70-78, 1999.

[5] A. Chervenak, I. Foster, A. Iamnitchi, C. Kesselman, W. Hoschek, P. Kunszt, M. Ripeanu, H. Stockinger, K. Stockinger, and B. Tierney, "Giggle: A Framework for Constructing Scalable Replica Location Services," presented at Global Grid Forum, Toronto, Canada, 2001.

[6] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke, "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Data Sets," *J. Network and Computer Applications*, pp. 187-200, 2001.

[7] R. Moore, C. Baru, R. Marciano, A. Rajasekar, and M. Wan, "Data-Intensive Computing," in *The Grid: Blueprint for a New Computing Infrastructure*, I. Foster and C. Kesselman, Eds.: Morgan Kaufmann, 1999, pp. 105-129.

[8] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the Impossibility of Group Membership," presented at 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96), New York, USA, 1996.

[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, vol. 32, pp. 374-382, 1985.

[10] P. Avery and I. Foster, "The GriPhyN Project: Towards Petascale Virtual Data Grids," Technical Report GriPhyN-2001-15, 2001.

[11] J. Yin, L. Alvisi, M. Dahlin, and C. Lin, "Hierarchical Cache Consistency in WAN," presented at The 1999 USENIX Symposium on Internet Technologies and Systems (USITS99), Boulder, Colorado, 1999.

[12] K. Holtman, "CMS Data Grid System Overview and Requirements," Technical Report GriPhyN-2001-1, 2001.

[13] M. Wilde, "Replica Catalog Performance and Capacity Requirements (v.6)," MCS-ANL, 2001.

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," presented at SIGCOMM 2001, San Diego USA, 2001.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," presented at SIGCOMM 2001, San Diego, USA, 2001.

[16] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," UC Berkeley, Technical Report CSD-01-1141, 2001.

[17] P. Druschel and A. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility," presented at 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Elmau/Oberbayern, Germany, 2001.

[18] M. Ripeanu, I. Foster, and A. Iamnitchi, "Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design," *IEEE Internet Computing Journal special issue on peer-to-peer networking*, vol. 6, 2002.

[19] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol," *IEEE/ACM Transactions on Networking*, vol. 8, pp. 281-293, 2000.

[20] V. Valloppillil and K. W. Ross, "Cache array routing protocol v1.0," in *Internet Draft*, 1988.

[21] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi, "Web Caching with Consistent Hashing," presented at The Eighth International World Wide Web Conference (WWW8), Toronto, Canada, 1999.

[22] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," presented at Symposium on Theory of Computing, 1997.

[23] B. Krishnamurthy, C. Wills, and Y. Zhang, "On the Use and Performance of Content Distribution Networks," presented at ACM SIGCOMM Internet Measurement Workshop, San Francisco, CA, USA, 2001.

[24] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, "Overcast: Reliable Multicasting with an Overlay Network," presented at 4th Symposium on Operating Systems Design and Implementation (OSDI 2000), San Diego, California, 2000.

[25] Y.-h. Chu, S. G. Rao, S. Seshan, and H. Zhang, "Enabling Conferencing Applications on the Internet Using an Overlay Multicast Architecture," presented at SIGCOMM 2001, San Diego, CA, 2001.

[26] J. Touch, "Dynamic Internet Overlay Deployment and Management Using the X-Bone," *Computer Networks*, vol. 36, pp. 117-135, 2001.

[27] J. Touch and S. Hotz, "The X-Bone," presented at Third Global Internet Mini-Conference at Globecom, Sydney, Australia, 1998.

[28] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris, "Resilient Overlay Networks," presented at 18th ACM SOPS, Banff, Canada, 2001.

[29] B. Bloom, "Space/time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, pp. 422-426, 1970.

[30] M. Mitzenmacher, "Compressed Bloom Filters," presented at Twentieth ACM Symposium on Principles of Distributed Computing (PODC 2001), Newport, Rhode Island, 2001.

[31] S. Raman and S. McCanne, "A Model, Analysis, and Protocol Framework for Soft State-based Communication," *Computer Communication Review*, vol. 29, 1999.

[32] P. Sharma, D. Estrin, S. Floyd, and V. Jacobson, "Scalable Timers for Soft State Protocols," presented at IEEE Infocom '97, 1997.

[33] R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch, "Design and Deployment of a National-Scale Authentication Infrastructure," *IEEE Computer*, vol. 33, pp. 60-66, 2000.

[34] A. Iamnitchi, M. Ripeanu, and I. Foster, "Locating Data in (Small-World?) P2P Scientific Collaborations," presented at 1st International Workshop on Peer-to-Peer Systems, Cambridge, MA, USA, 2002.

[35] T. D. Hodes, S. E. Czerwinski, B. Zhao, A. D. Joseph, and R. H. Katz, "An Architecture for Secure Wide-Area Service Discovery," *Wireless Networks*, 2001.

[36] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler, "Scalable, Distributed Data Structures for Internet Service Construction," presented at Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000), San Diego, CA, 2000.

[37] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "OceanStore: An Architecture for Global-Scale Persistent Storage," presented at 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), Cambridge, MA, 2000.

[38] S. D. Gribble, M. Welsh, R. v. Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. D. Joseph, R. H. Katz, Z. Mao, S. Ross, and B. Zhao, "The Ninja Architecture for Robust Internet-Scale Systems and Services," *Special Issue of Computer Networks on Pervasive Computing*, 2001.

[39] J. K. Mullin, "A second look at Bloom filters," *Communications of the ACM*, vol. 26, pp. 570-571, 1983.

[40] M. V. Ramakrishna, "Practical performance of Bloom filters and parallel free-text searching," *Communications of the ACM*, vol. 32, pp. 1237-1239, 1989.

[41] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," presented at ACM SIGCOMM'98, Vancouver, Canada, 1998.