cs 591B1: ALGORITHMIC ASPECTS OF COMPUTER NETWORKS **Spring 2002**

Lecture 5 — February 18, 2002

Lecturer: John Byers *BOSTON UNIVERSITY* *Scribe: Peng Xie*

In this lecture, we are going to study two algorithms to solve the problem associated with "hot spots" in the Internet. "Hot spots" occur when a large number of web clients send a high volume of requests for popular content to a single server concurrently. Random Tree algorithm disperses the loading by routing the requests through a distributed set of virtual paths through the caches. Popular files are actually stored at many caches, distributing the load and alleviating hot spots. The second algorithm, consistent hashing, balances the load among the set of caches, each cache in the set has been assigned roughly equal load. In addition, the change of the set of caches doesn't cause the total change in the mapping. Consistent hashing even works well when the clients have different views about the set of caches. The last two properties are very important for the applications in the Internet today where machines come and leave when they are down or brought into the Internet.

5.1 Random Tree

In order to solve the "hot spots" problem, one approach is to set up caches for the hot server. The random tree adopts a tree of caches, which is called abstract tree. With each page, we associate an abstract tree. Generally, we use the same structure for all the abstract trees. For different pages, the nodes of the tree are mapped to different caches. The number of the nodes in the tree equals to the number of the caches. We identify the nodes of the tree by their order in the breadth-first search algorithm. If the degree of the tree is d, the total number of the nodes is C , then the height of the tree is $\log_d C$. An abstract tree for a single page is shown in Fig 5.1.

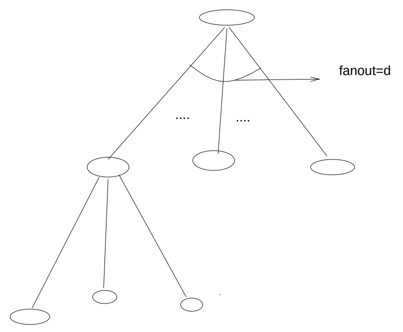

Figure 5.1. an abstract d-ary tree

We create a hash function as follows: $h : P \times [1 \cdots C] \rightarrow \mathbf{C}$, where P is the set of page IDs , C is the set of index of the tree nodes and $\mathbf{C}$ is the set of the caches. The root of the tree is always mapped to the server of that page. Fig 5.2 is an example of the hashing function for page 1. In this hashing function, the node 1 in the abstract tree and page 1 are mapped to the cache C_4, i.e., $h(P_1, 1) = C_4$.

To request a page, requester selects a random leaf on the tree (a random path on the tree) and issues a request. When a cache receives a request, it first checks if it has a copy of that page, if so, it sends the page to the requester. Otherwise, it sends the request to the next machine and increases the counter for that page. When the counter is greater than q, the cache will ask the server for a copy of that page. The server will send a copy of this page along the path down to this cache. If any caches along the path are qualified

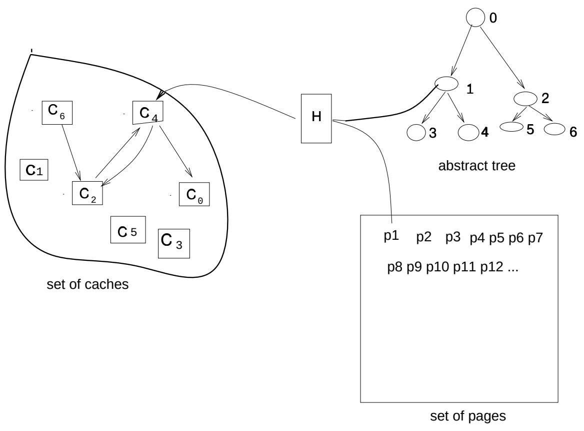

Figure 5.2. Hashing URLs to search paths

for keeping copy of that page, they will keep copies of that page. When this cache gets requests for this page again, it will serve the requesters without sending its parent request anymore.

We are going to analyze this algorithm to show that the random tree algorithm prevents any cache from being swamped with high probability. we are considering two cases: the leaf nodes and internal nodes.

The analysis proceeds in two parts:
1) ensuring that there are not too many requests at leaves, and
2) ensuring there are not too many requests at internal nodes.

We make some assumptions here. The maximum number of the request is $R = \rho C$. where ρ is a constant, and C is the number of the caches. The height of the tree is $\log_d C$, a request may span $\log_d C$ caches. So the total requests are $\rho C \log_d C$. The average number of the request each cache receives is $\rho \log_d C$.

First of all, we analyze the request to the leaf nodes.

There are C nodes in an abstract tree and $O(C)$ leaves and there are C caches to map to the leaves. This leads to a balls-in-bins type of analysis. For example, there are $\frac{C}{2}$ leaves when $d = 2$. For the balls-in -bins model, there are $C/2$ balls and C bins. The worst case(fullest bin) load is expected to be $O(\frac{\log C}{\log \log C})$. A possible mapping of the leaves is shown in Fig 5.3. In the example, the number beside the node is the identifier for the caches. Note that cache 4 appears in the leaves twice while cache 0 doesn't appear in the mapping at all.

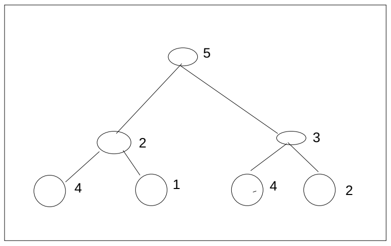

Figure 5.3. a possible mapping of the leaves to caches for a given page

This analysis is with respect to a single abstract tree, i.e.,URL. In a targeted adversary situation, under which all the R request are made for one page, requests for that page would averagely cause $\frac{R}{C}$ load on each leaf. Some cache would get $\frac{R \log C}{C \log\log C}$ requests. Therefore, the expected number of hits to a cache in the leaf level is $\frac{\rho \log C}{\log\log C}$ in a targeted adversary situation.

We use a Chernoff bound to give the lower bound and upper bound for the numbers of one cache is being hitted by the leaf nodes.

The following definition and theorems are taken from [4].

Definition: Poisson trials: Let $X_1, \ldots, X_n$ be independent events, and for $1 \leq i \leq n$, $Pr[X_i = 1] = p_i$, and $Pr[X_i = 0] = 1 - p_i$, X *(sum of the a sequence of n trials)* $= \sum_{i=1}^{n} X_i$, μ *(expected sum)* $= \sum_{i=1}^{n} p_i$.

Theorem4.1: Let $X_1, X_2, \ldots, X_n$ be independent Poisson trials such that , for $1 \leq i \leq n$, $P_r[X_i = 1] = p_i$, where $0 < p_i < 1$. Then , for$X = \sum_{x=1}^{n} X_i, \mu = E[X] = \sum_{i=1}^{n} p_i$,and any $0 < \delta$,
$P_r[X > (1+\delta)\mu] < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu}$

Theorem4.2: Let $X_1, X_2, \ldots, X_n$ be independent Poisson trials such that , for $1 \leq i \leq n$, $P_r[X_i = 1] = p_i$, where $0 < p_i < 1$. Then , for $X = \sum_{x=1}^{n} X_i, \mu = E[X] = \sum_{i=1}^{n} p_i$,and $0 < \delta \leq 1$,
$P_r[X < (1-\delta)\mu] < e^{\frac{-\mu\delta^2}{2}}$

In our case, $p_1 = p_2 = \ldots = p_C = \frac{1}{C}$, $\mu = 1$, we can apply $\delta \approx \frac{1.5 \log C}{\log\log C}$ in theorem 4.1, and get $P_r[X > (1+\delta)\mu] < \frac{1}{C^2}$

Therefore, for the caches in the leaf level, with probability at most $\frac{1}{C^2}$,the number of requests a cache in the leaf node gets is more than

$\frac{\rho 1.5 \log C}{\log\log C} = \frac{\rho \log C}{2 \log\log C} + \frac{2\rho \log C}{2 \log\log C} \in O(\frac{\rho \log C}{\log\log C})$

We get the following probability:
P_r[each cache hitted by the leaf node gets requests more than $O(\frac{\rho \log C}{\log\log C})$]$\leq \frac{1}{C^2}$

We can see that the probability that a machine at the leaf level gets fullest load is very low.

We now analyze the requests for internal nodes.

For the internal nodes, no abstract node can get more than dq requests because each child of this node can send at most q requests for a page and there are at most d children. We use n_j to denote the number of nodes in the abstract tree that gets between 2^j and 2^{j+1} requests, where $0 \leq j \leq \log dq - 1$. Mapping these n_j nodes to the caches, some caches get hit possibly more than once. Let X denote the event that a cache gets more than z hits. Because all the caches have the same chance to get hitted, we get the following probability:

$P_r[X] = \sum_{i=z}^{n_j} \binom{n_j}{z} (\frac{1}{C})^z \leq \sum_{i=z}^{n_j} (\frac{en_j}{Cz})^z$.

In order to make the P_r is smaller than $\frac{1}{C}$, we must have
$z \in \Omega(\frac{n_j}{C} + \frac{\log C}{\log(\frac{C}{n_j} \log C)})$.

There are total $\log dq$ such nodes (n_j), so with the probability at least $1 - \frac{\log dq}{C}$, the total requests received by one cache hitted by the interval nodes will be the order:

$\sum_{j=0}^{\log dq-1} 2^{j+1}(\frac{n_j}{C} + \frac{\log C}{\log(\frac{C}{n_j}\log C)}) \in 2\rho \log_d C + O(\frac{dq \log C}{\log \frac{dq}{\rho} \log C})$

By analyzing the internal nodes and the leaf nodes, we can see that with high probability, the random tree algorithm can prevent the cache from being be swamped.

5.2 Consistent Hashing

The second algorithm is the consistent hashing. This algorithm assigns a set of items to a set of buckets such that each bucket receives almost equal number of items. Additionally, a bucket joining the set or leaving the set doesn't cause total change of mapping.This algorithm localizes the small change in the set of buckets. Let I be the set of items and B be the set of buckets. A view, V is any subset of B. The function $\gamma_B(b)$ maps buckets randomly to the unit interval and $\gamma_I(i)$ maps the items in the same way.

Definition: The hash function $f_V(i)$ is defined to be the bucket $b \in V$ that minimizes $|\gamma_B(b) - \gamma_I(i)|$. An example of hashing function is shown in Fig5.4.

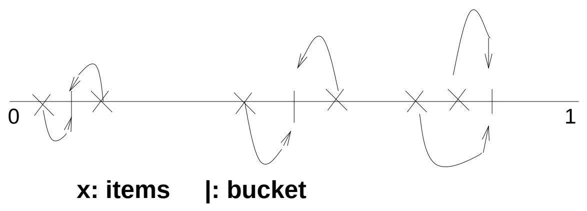

Figure 5.4. a consistent hashing function

For the consistent hashing function, we define the following concepts:

ranged hash function is a function of the form $f : 2^B \times I \to B$. A ranged hash family is a family of ranged hash functions. Obviously, the consistent hashing function family is the ranged hash function family.

balance A ranged hash family is balanced if, given a particular view V, a set of items, and a randomly chosen function selected from the hash family, with high probably, the fraction of items mapped to each bucket is $O(1/|V|)$. V is a view (a set of buckets)

monotonicity A ranged hash function f is monotone if for all views $V_1 \subseteq V_2 \subseteq B$, $f_{V_2}(i) \in V_1$ implies $f_{V_1}(i) = f_{V_2}(i)$. A ranged hash family is monotone if every ranged hash family in it is monotone. This property means that adding/removing buckets perturbs only local data.

spread : Let $V_1, V_2, \ldots, V_V$ be a set of views, altogether containing C distinct buckets and each individually containing at least $\frac{C}{t}$ buckets. For a ranged function and a particular item i,the spread $\sigma(i)$ is the quantity$|\{f_{V_j}(i)\}_{j=1}^{V}|$. The spread of a hash function $\sigma(f)$ is the maximum spread of an item. The spread of a hash family is σ if with high probability, the spread of a random hash function from the family is σ. This property is used to bound the number of distinct bin hits by a single item over all views,assuming view sees a constant fraction of bins $(\frac{1}{t})$. A good consistent function should have low spread over all items.

load For a range hash function f and bucket b, the load $\lambda(b)$ is the quantity $|\bigcup_v f_v^{-1}(b)|$.The load of a hash function is the maximum load of a bucket. The load of a hash family is λ if with high probability, the randomly chosen hash function has load λ. This property is used to measure how many items a bucket is thought to hold in a particular view. For a given view, load per bucket is near uniform in term of the the number of bins in the bucket. A good hash function should has low load.

We can get the following theorem[1].

theorem: The consistent hash function family F has the following properties:

1.F is monotone .

2.balance : For a fixed view V, $P_r[f_V(i) = b] \leq \frac{O(1)}{|V|}$ for $i \in I$ and $b \in V$, and conditioned on the choice of γ_B, the assignments of items to buckets are $\log(C)$-way independent.

3.spread : If the number of views $V = \rho C$,for some constant ρ , and the number of items $I = C$, then for $i \in I, \sigma(i)$ is $O(t \log(C))$ with probability greater than $1 - \frac{1}{C^{\Omega(1)}}$

4.Load : if V and I are as above, then for$b \in B$,$\lambda(b)$is $O(t \log(C))$ with probability greater than $1 - \frac{1}{C^{\Omega(1)}}$.

From this theorem, we can see that consistent hashing function algorithm allows different machines in the group have different view of the network and agree on the location for an item with low communication.

Bibliography

[1] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin,and Rina Panigrahy. "*Consistent Hashing and Random Trees: Distributed caching Protocol for Releiving Hot Spots on the World Wide Web*," in Proceedings of STOC '97.

[2] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp,and Scott Shenker. "*A Scalable Content-Addressible Network*," in Proceedings of ACM SIGCOMM '01.

[3] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek,and Hari Balakrishnan. "*Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*," in Proceedings of IEEE INFOCOM'99.

[4] Rajeev Motwani and Prabhakar Raghavan. "*Randomized Algorithms* , " Cambridge University Press. p67-73.