cs 591B1: ALGORITHMIC ASPECTS OF COMPUTER NETWORKS		Spring 2002
	Lecture 12 — April 30	
Lecturer: See sections	*BOSTON UNIVERSITY*	*Scribe: Scott Russell*

This is the summary of the first session of course project presentations.

12.1 Dynamic Bloom Filters

Presenters: Scott Russell

Problem: How can we alter Bloom Filters to allow them to support the usual dynamic table expand and contract operations? Also, can this be done without access to elements previously inserted into the filter?

Proposed Solution: To expand a filter, instead of rehashing the elements in the filter into a larger filter, create an Aggregate filter. This is done by appending a new independent filter to the original. Insert new elements into the new filter until it becomes full and repeat as necessary. An element is in the Aggregate filter and therefore in the set if it is found in any sub-filter.

Results: Aggregate filters have an amortized insertion time equal to standard Bloom filters' while straight forward rehashing leads to amortized insertion time that's approximately 3 times as much. Membership queries for Aggregate filters are slower than those for a standard filter, but can be sped up by searching the set of sub-filters making up the Aggregate from largest to smallest. Aggregate filters are also susceptible to wasted space due repeated insertions of the same element over a period of time. A "smart" insertion policy can be used, but then each insertion takes as long an aggregate membership test plus a standard insert. The presenter claims a bound on the wasted space of 1/2 the total filter capacity when filters are doubled at each expansion. The false positive rate for aggregate filter is approximately the product of its sub-filters' false positive rates.

To contract a filter, employ consistent hashing when creating or expanding a filter. Using binary strings for the array labels we can merge buckets whose labels differ in the last bit. Here also, there is no need to access the inserted elements themselves.

Future Work: Support for deletions, simulations, alternate constructions, and controlling aggregate false positive rate

12.2 An Indexing Mechanism with a Worst Case Guarantee

Presenters: Anna Karpovsky and Xun Yuan

Problem: The Chord data structure distributes data efficiently, but is balanced with respect to identifiers not the number of nodes and provides no worst case search guarantee. The presenters try to modify Chord in order to bound the worst case while keeping the average search time, retained state, and update times logarithmic in the number of nodes.

Proposed Solution: Each node creates a 2-3-4 tree for all the nodes it knows about. Instead of the usual Chord finger table the node stores the leftmost path or subtree of its 2-3-4 tree as its finger table. The motivation is that since changes to 2-3-4 trees are local and trees for different nodes will overlap updates should be inexpensive.

Results: Before trying 2-3-4 trees, they first attempted to make finger table entries depend on the number of intermediate nodes rather than the distance between node identifiers. This gave linear update times, and was abandoned.

For the 2-3-4 idea, inserts and deletes are efficient and the worst case search is now logarithmic due to the properties of the tree. Unfortunately, tree updates are more complex than updates to the original Chord finger table structure. Tree construction is currently based on the order of node arrivals which is assumed to be known.

Simulations revealed that number of nodes that needed to update their tables is greater than the $\log N$ goal, but better than Chord's worst case $\log^2 N$ result. There are three types of tree updates: node level changes, node additions, node deletions. Of these, the level changes impact the greatest number of nodes and lead to the greatest number of updates. The presenters believe periodic updates with more frequent updates for the node departure case can help reduce the number of nodes that need to update their tables.

Future Work: Try building random trees to reduce update times and do away with need to know the order of the nodes arrivals. Also a more detailed set of rules for constructing nodes' finger tables seems necessary.

12.3 Unbiased Bloom Filters for Compression of Comparative Statistical Data

Presenters: Ryan Mahon

Problem: Using Bloom filters to perform statistical queries to determine which elements or combinations of elements are common across a number of sets is problematic because the error for a particular query can deviate significantly from the mean error. We might want to do this kind of comparison to formulate a Web history scan or for a music file survey.

Proposed Solution: Use a group of Bloom filters or hash sets for each set being represented to evenly distribute the inaccuracies. To perform a statistical query, choose a random filter from the group of filters and use this filter for all of the sets for that query.

Results: The idea can be implemented by using an additional $\log S$ header giving the choice of filter, where S is the number of filters per set.

Simulations were intentionally performed using non optimal parameters. 500 sets of elements were created with set elements drawn from a universe of size 256. Uniform, exponential, and heavy tail distributions were used to generate the sets. For ease of simulation, linear congruential hash functions were used. All single and two element combination queries were performed for a single filter and for a group of 15 filters and compared to the actual query results performed on the sets themselves (instead of their filter representations). The results were good for the uniform and exponential distributions. The error for particular queries showed a much smaller deviation from the average error when a group of 15 filters was used. For the heavy tailed distribution, there were still some sizable deviations from the mean error for certain queries.

Future Work: Use of better hashing in simulations, examining the effect of using a heterogeneous group of filters, i.e. different size, capacity and number of hashes, rather than a homogeneous set to represent each set.

12.4 Active Network Accounting Using Bloom Filters

Presenters: Manish Sharma and Dhiman Barman

Problem: How to accurately identify large, heavy, or non-responsive flows in a scalable fashion for the purposes of pricing, bandwidth provisioning, denial of service attack detection, and queue management? Current state of the art routers are slow, memory intensive, and inaccurate.

Results: Current approaches include random sampling with serial and parallel multistage filters [3] and probabilistic packet marking for detection of non-responsive flows using Bloom Filters[4].

The work by Estan and Varghese[3] uses filters in parallel, each with a different set of hash functions. The size of each packet is inserted into a location determined by hashing that packet's flow identifier. Upon insertion of a packet, if all of the buckets corresponding its flow ID exceed a certain prescribed threshold, then the flow is classified as a large flow. The presenters instead use a 2 stage serial Bloom filter where the size, threshold, and number of associated hash functions for each filter are tunable parameters. Flows are first hashed into several buckets in the first stage filter and when all the buckets corresponding to the flow ID overflow, then the packets of that flow are hashed into several buckets in the second stage. If a packet of a flow cause all of the buckets in the second stage to overflow, the flows is then identified as being a large flow.

Simulations were performed comparing the presenters' approach to the Estan Varghese approach. Parameters were chosen for the presenters' scheme so that the memory requirements were equal to those for the Estan Varghese scheme. Packet sizes were generated according to Pareto and Exponential distributions. In both cases, only flows which occupied at least 4% of the link bandwidth were tracked. The presenters' approach has the same success rate for large flows as the Estan Varghese approach, but *misclassifies* only half as many of the smaller flows. For the Pareto distribution, the 2 stage serial filtering far outperformed the 3 stage parallel filtering. Misclassification of small and medium flows due to false positives of the Bloom Filters can be reduced by using more memory (i.e. using larger filters).

Future Work: Determining appropriate memory allocation and measurement intervals to get better classification performance, simulations using actual traffic traces, and attempting to achieve finer grained flow identification.

Bibliography

[1] Bloom, B. “Space/time trade-offs in hash coding with allowable errors,” in Communications of the ACM, 13(7):422-426, 1970. Proceedings of STOC 1997.

[2] Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. “Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications,” in Proceedings of ACM SIGCOMM 2001.

[3] Estan, C. and Varghese, G. “New Directions in Traffic Measurement and Accounting,” in IMW 2001.

[4] Feng, W.C., Kandlur, D., Saha, D., Shin, K. “Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness,” in INFOCOM 2001.