An Examination of Bloom Filters and their Applications

Jacob Honoroff

March 16, 2006

Outline

- Bloom Filter Overview
- Traditional Applications
- Hierarchical Bloom Filters Paper
- Less Traditional Applications & Extensions

Outline

- **Bloom Filter Overview**
- Traditional Applications
- Hierarchical Bloom Filters Paper
- Less Traditional Applications & Extensions

Bloom Filter Overview

"Space/Time Trade-offs in Hash Coding with Allowable Errors", Burton Bloom, *Communications of the ACM*, 1970.

Application example: Program for automatic hyphenation in which 90% of words can be hyphenated using simple rules, and 10% require dictionary lookup.

Bloom Filter Principle

“Network Applications of Bloom Filters: A Survey” A. Broder, M. Mitzenmacher, *Allerton Conference on Communication, Control, and Computing*, 2002

“Whenever a list or set is used, and space is consideration, a Bloom filter should be considered. When using a Bloom filter, consider the potential effects of false positives.”

Notation

S is a set of n elements.

Set of k hash functions with range $\{1 \ldots m\}$ (or $\{0 \ldots m-1\}$).

m-long array of bits initialized to 0.

Families of Hash Functions

k hash functions $h_1 \ldots h_k$

We could use SHA1, MD5, etc.

How could we get a family of size k?

$h_i(x) =$ MD5$(x + i)$ or MD5$(x \parallel i)$ would work.

Example

We insert and query on a Bloom filter of size $m = 10$ and number of hash functions $k = 3$.

Let $H(x)$ denote the result of the three hash functions which we will write as a set of three values $\{h_1(x), h_2(x), h_3(x)\}$

We start with an empty 10-bit long array:

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

Insert x_0:

$H(x_0) = \{1, 4, 9\}$

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	0	0	0	0	1

Insert x_1:

$H(x_1) = \{4, 5, 8\}$

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

$H(x_0) = \{1, 4, 9\}$

$H(x_1) = \{4, 5, 8\}$

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

Query y_0:

$H(y_0) = \{0, 4, 8\} \longrightarrow$ No

Query y_1:

$H(y_1) = \{1, 5, 8\} \longrightarrow$ Yes (False Positive)

A Little Math (Broder & Mitzenmacher)

After n elements inserted into bloom filter of size m, probability that a specific bit is still 0 is

$$\left(1 - \frac{1}{m}\right)^{kn} \approx \mathrm{e}^{-\frac{kn}{m}}$$

(The useful approximation comes from a well-known formula for calculating e):

$$\lim_{x \to \infty} \left(1 - \frac{1}{x}\right)^{-x} = \mathrm{e}$$

Thus the probability that a specific bit has been flipped to 1 is

$$1 - \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - \mathrm{e}^{-\frac{kn}{m}}$$

Useful Approximation

x	$\left(1 - \frac{1}{x}\right)^{-x}$
4	3.160494
16	2.808404
64	2.739827
256	2.723610
1024	2.719610
4096	2.718614
16384	2.718365
65536	2.718303
262144	2.718287
1048576	2.718283
4194304	2.718282

A Little Math

A false positive on a query of element x occurs when **all** of the hash functions $h_1 \ldots h_k$ applied to x return a filter position that has a 1.

We assume hash functions to be **independent**.

Thus the probability of a false positive is

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - \mathrm{e}^{-\frac{kn}{m}}\right)^k$$

Choose k To Minimize False Positives

We are given m and n, so we choose a k to minimize the false positive rate.

Let $p = \mathrm{e}^{-\frac{kn}{m}}$. Thus we have

$$\begin{aligned} f &= \left(1 - \mathrm{e}^{-\frac{kn}{m}}\right)^k \\ &= (1-p)^k \\ &= \mathrm{e}^{k \ln(1-p)} \end{aligned}$$

So we wish to minimize $g = k \ln(1-p)$.

Choose k To Minimize False Positives

We could use calculus. Less messy, we notice that since

$$\ln\left(\mathrm{e}^{-\frac{kn}{m}}\right) = -\frac{kn}{m}$$

we have

$$\begin{aligned} g &= k\ln(1-p) \\ &= -\frac{m}{n}\ln(p)\ln(1-p) \end{aligned}$$

and by symmetry, we see that g is minimized when $p = \frac{1}{2}$

Choose k To Minimize False Positives

Since $p = e^{-\frac{kn}{m}}$, when $p = \frac{1}{2}$ we have

$$k = \ln 2 \cdot \left(\frac{m}{n}\right)$$

Plugging back into $f = (1-p)^k$, we find the minimum false positive rate is

$$\left(\frac{1}{2}\right)^k \approx (.6185)^{\frac{m}{n}}$$

Caveat: k must be an integer.

Optimal Filter Structure

Recall $p = \mathrm{e}^{-\frac{kn}{m}}$ is the probability than any specific bit is still 0.

So $p = \frac{1}{2}$ corresponds to a half-full Bloom filter array.

m, n, k Examples

From `http://www.cs.wisc.edu/~cao/papers/summary-cache/`

False positve rates for choices of k given m/n

m/n	k	k=1	k=2	k=3	k=4	k=5
2	1.39	0.393	0.400			
3	2.08	0.283	0.237	0.253		
4	2.77	0.221	0.155	0.147	0.160	
5	3.46	0.181	0.109	0.092	0.092	0.101
6	4.16	0.154	0.0804	0.0609	0.0561	0.0578
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347
8	5.55	0.118	0.0489	0.0306	0.024	0.0217

Outline

- Bloom Filter Overview
- **Traditional Applications**
- Hierarchical Bloom Filters Paper
- Less Traditional Applications & Extensions

Application: Weak Password Dictionary

“Opus: Preventing Weak Password Choices”, E. Spafford, *Computer and Security*, 1991

Store dictionary of easily guessable passwords as bloom filter, query when users pick passwords.

Can add new entries (e.g. previously used passwords).

Application: Weak Password Dictionary

What is a false positive in this context?

Application: Weak Password Dictionary

What is a false positive in this context?

A strong password that happens to hit. No big deal, just ask user for another one.

Weak Password Dictionary Caveat

Normally, we don't care about cryptographically strong hash functions.

But if we store sensitive data (previously used passwords), we do care, given attacker can see change in filter.

Solution: Use strong hash functions, or encrypt words before entering.

Application: Traceback

"Hash-Based IP Traceback", A. Snoeren et al., *SIGCOMM*, 2001

Developed "Source Path Isolation Engine" (SPIE)

600.424 Week 5 (Oct 10) related reading, remember?????

SPIE Traceback

Different goal than HBF authors:

“In an IP framework, the packet is the smallest atomic unit of data. Any smaller division of data (a byte for instance) is contained within a unique packet. Hence an optimal IP traceback system would precisely identify the source of an arbitrary IP packet”.

SPIE Traceback

Hashes “invariant” fields of IP header and first 8 bytes of payload into Bloom filter.

Explicitly handles fragmentation, NAT, ICMP messages (?), IP-in-IP tunneling (?) and IPsec (?) using additional 64-bit data structure

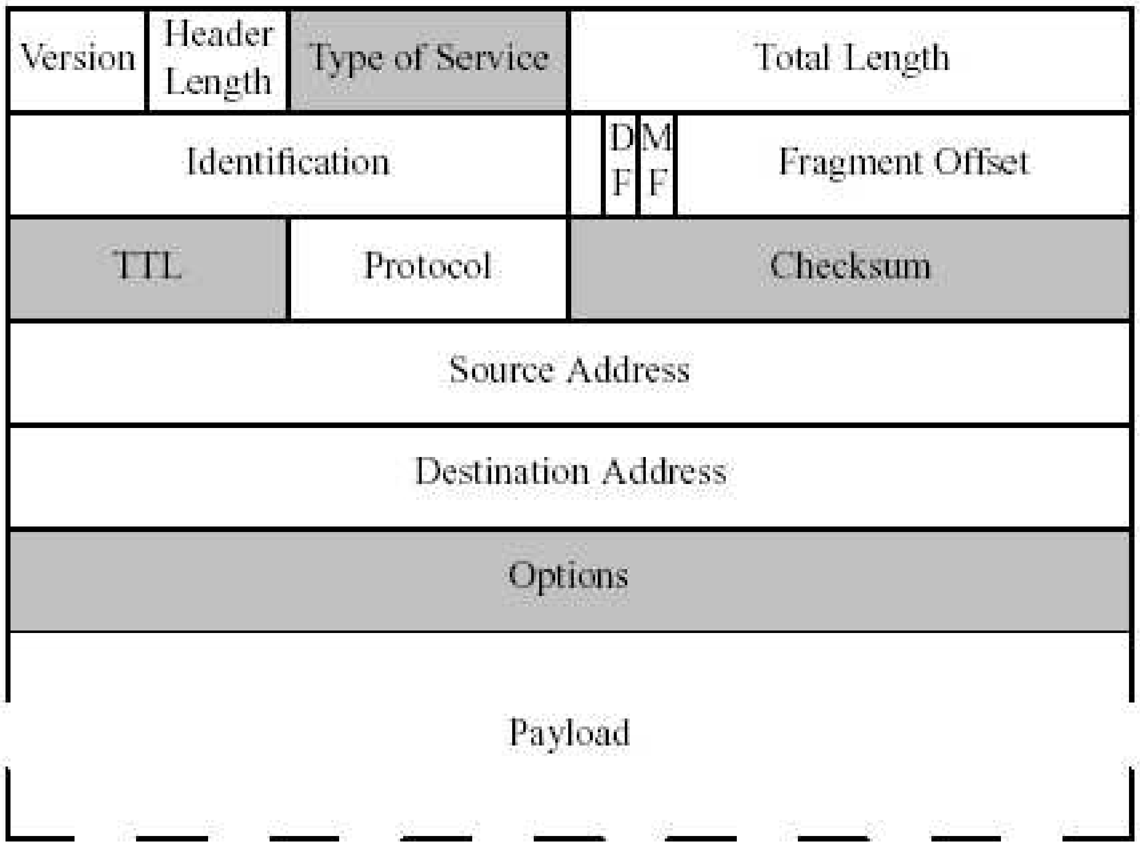
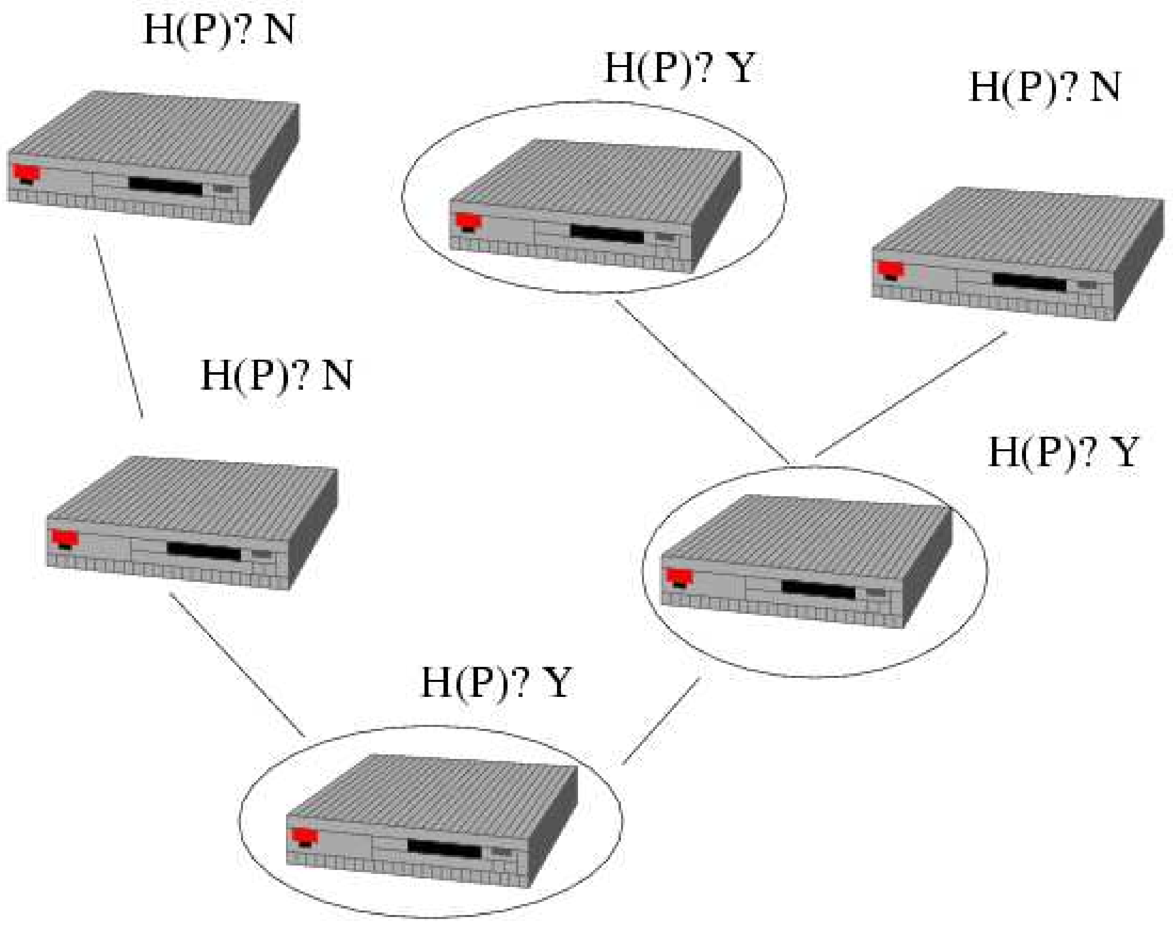


Figure 2: The fields of an IP packet. Fields in gray are masked out before digesting, including the Type of Service, Time to Live (TTL), IP checksum, and IP options fields.

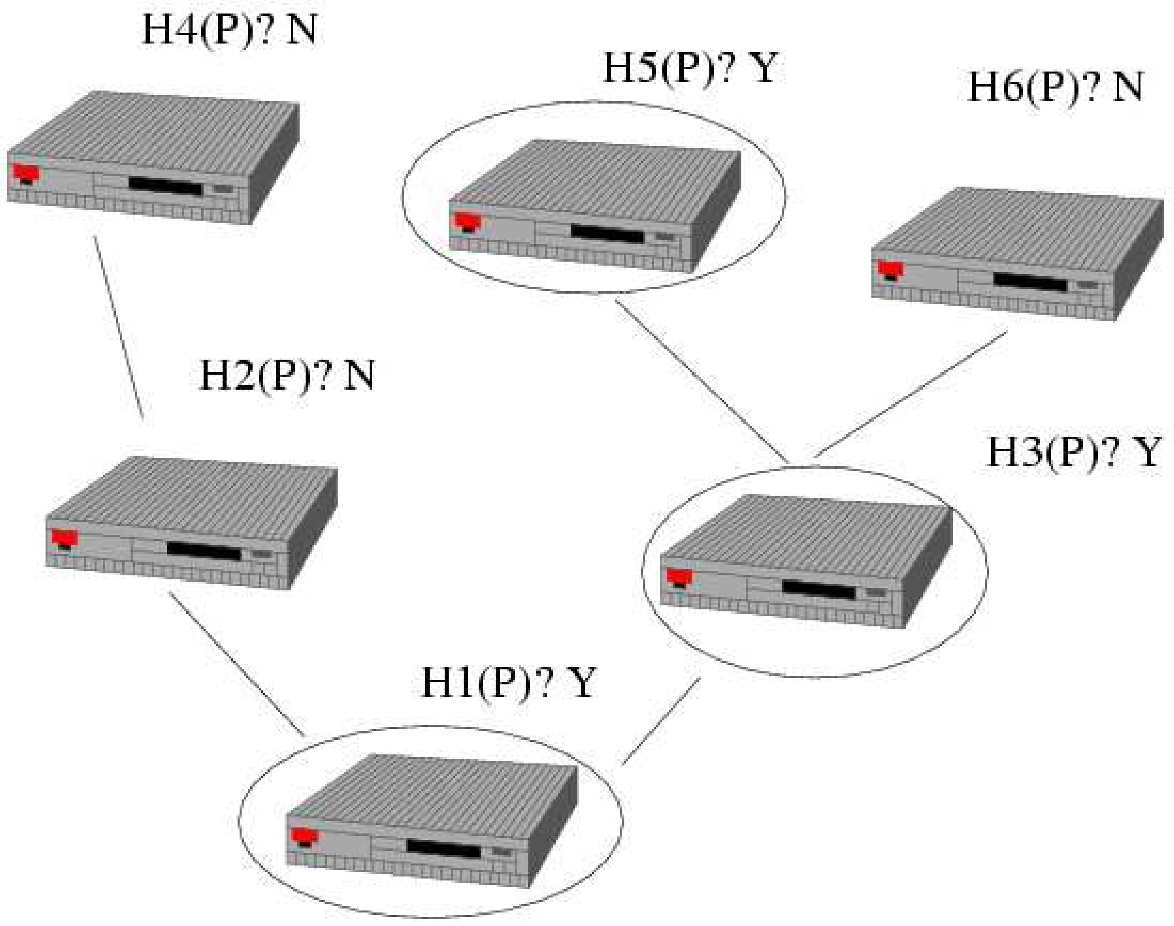

H(P)? N
H(P)? Y
H(P)? N
H(P)? N
H(P)? Y
H(P)? Y

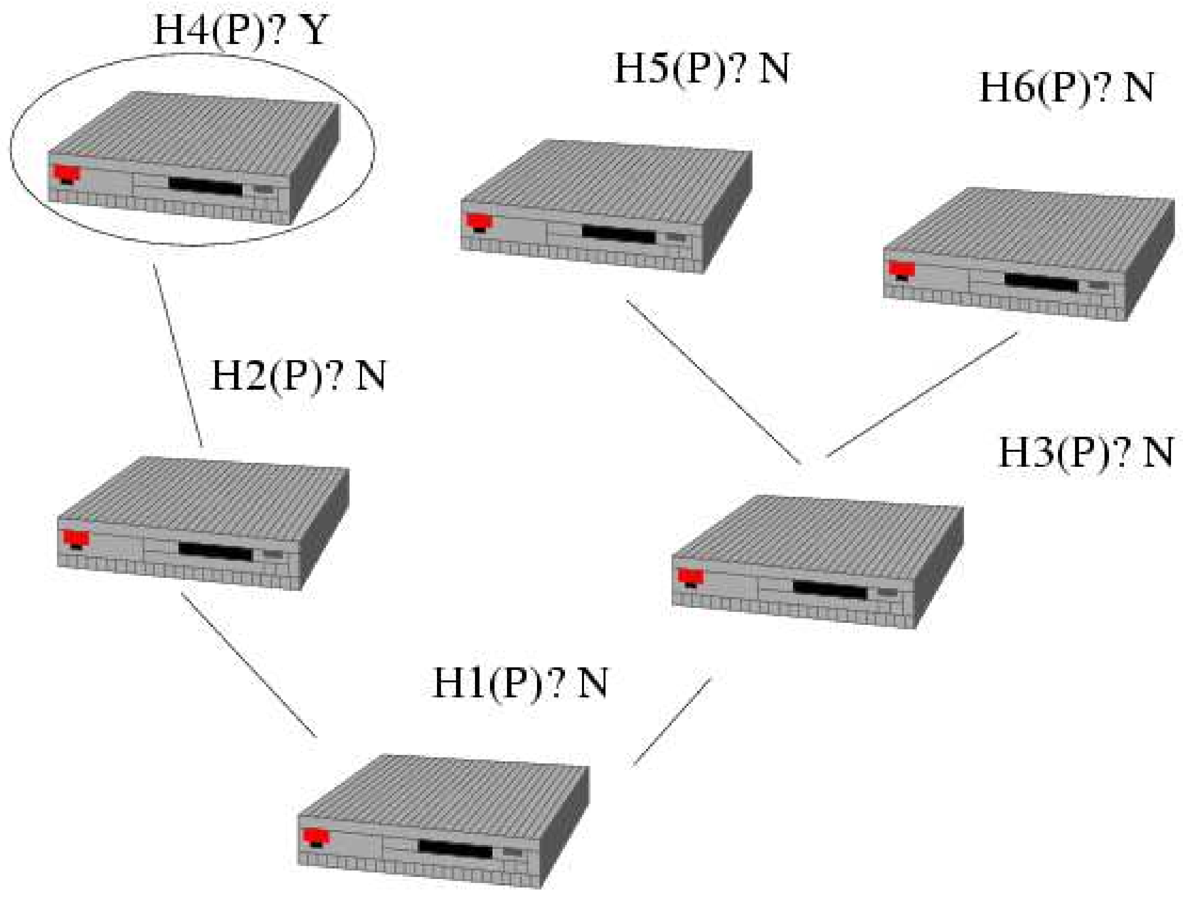
SPIE Bloom Filter Usage

Each router in path hashes packet digests into Bloom filters which are paged and stored locally for some amount of time.

Key point: Each router's hash functions are **independent**. They are based on RNG seeded at each router and changed every page out.

So false positives are **independent** of false positives at other routers or at other time periods.


H4(P)? N
H5(P)? Y
H6(P)? N
H2(P)? N
H3(P)? Y
H1(P)? Y


H4(P)? Y
H5(P)? N
H6(P)? N
H2(P)? N
H3(P)? N
H1(P)? N

Application: Cache Sharing

"Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol", L. Fan et al., *IEEE/ACM Transactions on Networking*, 2000.

Proxies on same side of network bottleneck share their caches.

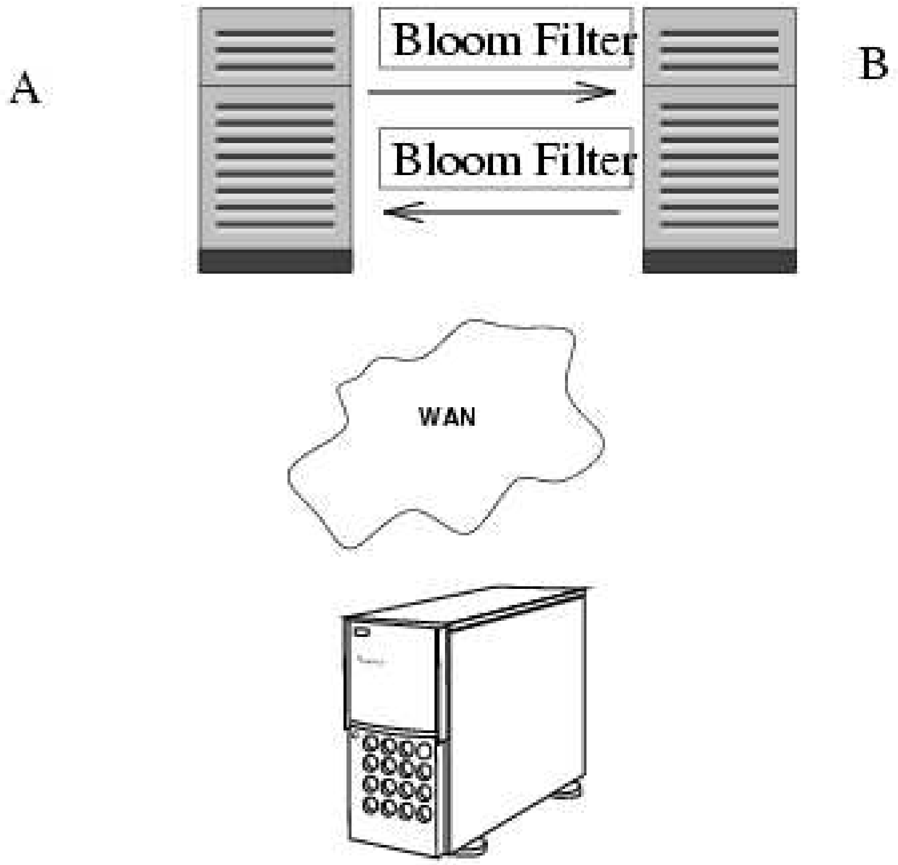
Proxies use Internet Cache Protocol (ICP). Messages sent out to all other proxies on cache misses.

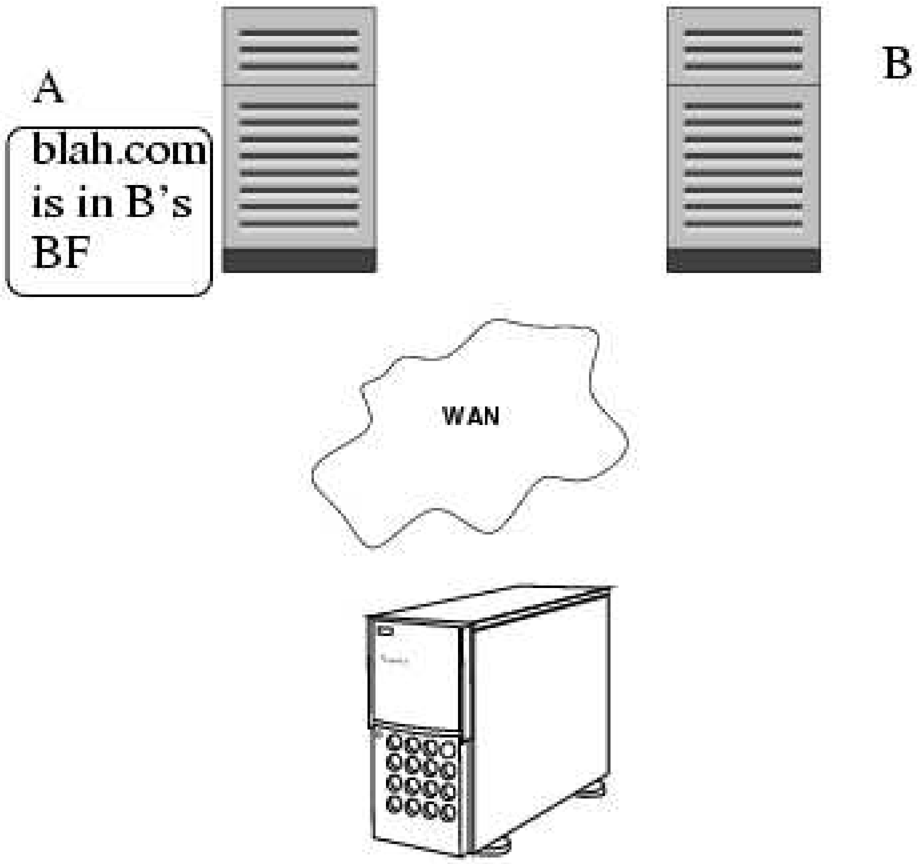
Involves a lot of interproxy communication, adding to network load.

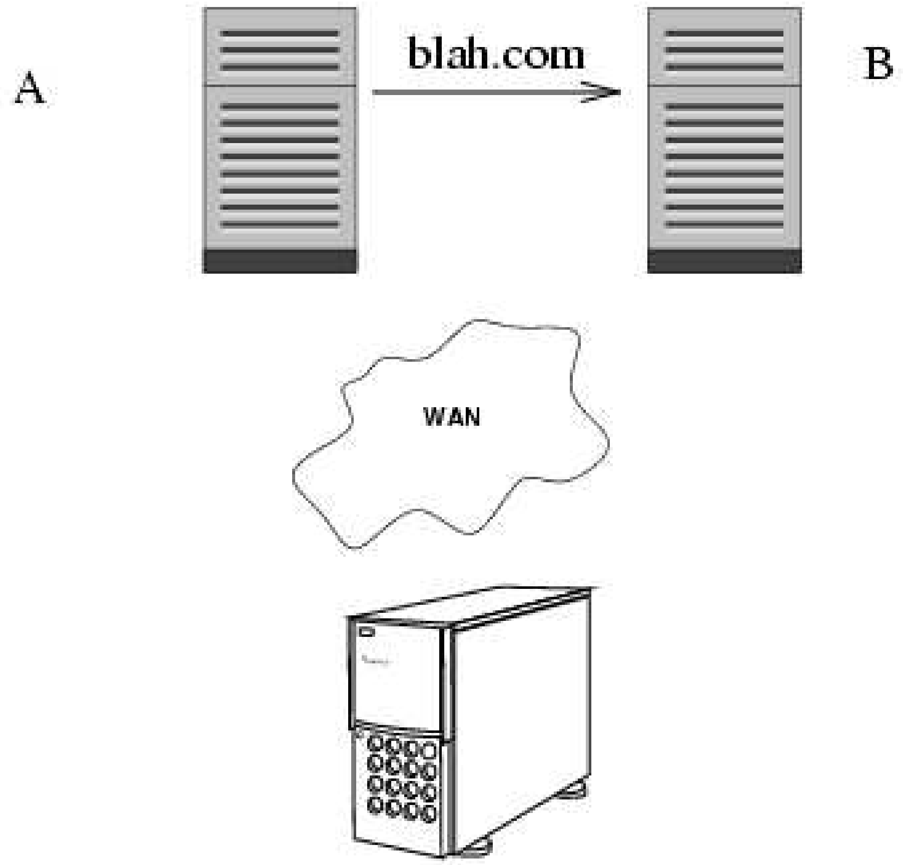
Application: Cache Sharing

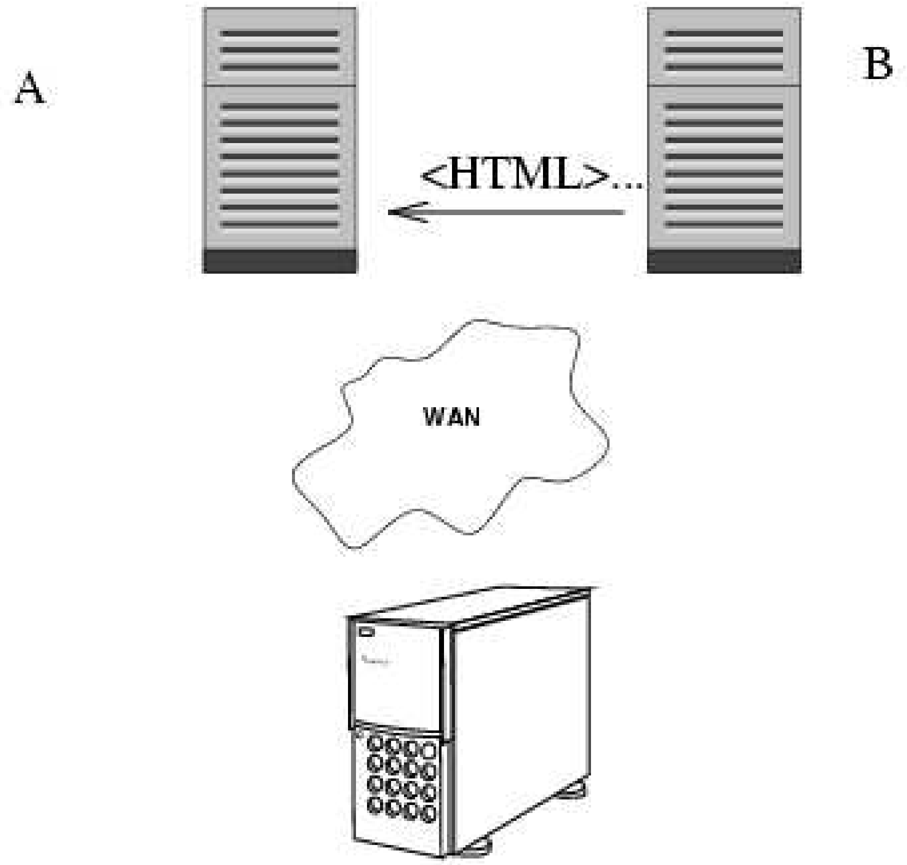
Proxy hashes all of the URLs in its cache into Bloom Filter.

Proxies periodically exchange Bloom filters, so queries of other caches can be made **locally** without sending ICP message.


A
Bloom Filter
Bloom Filter
B
WAN


A
blah.com is in B's BF
B
WAN


A
blah.com
B
WAN


A
B
<HTML>...
WAN

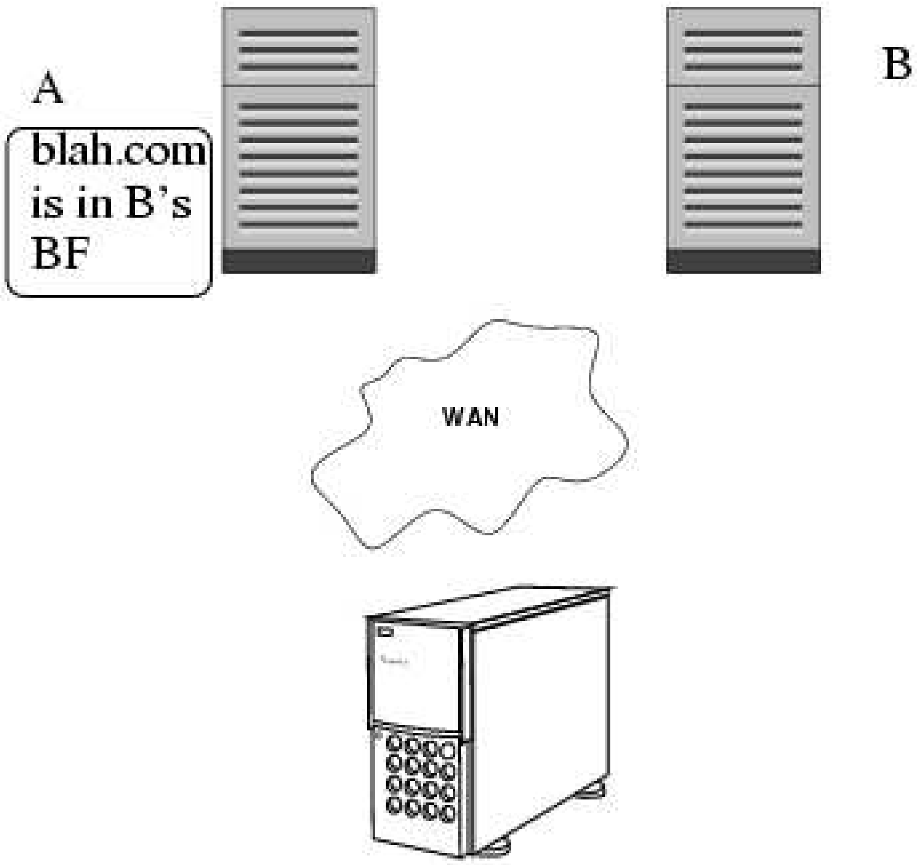
Application: Cache Sharing

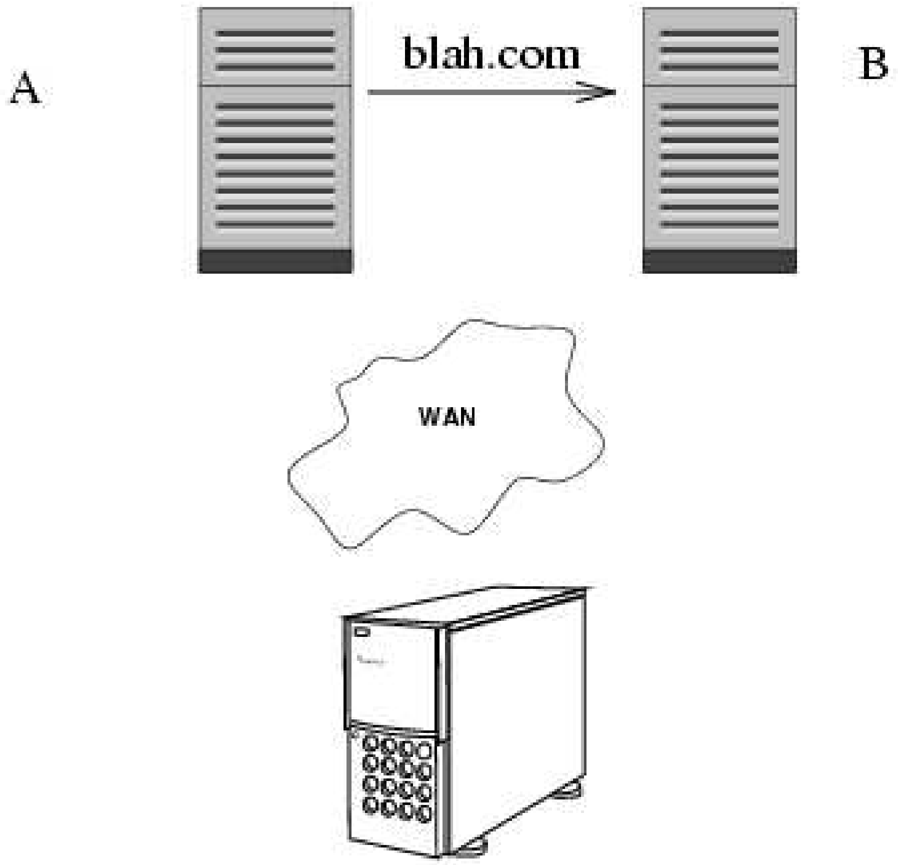
Proxy hashes all of the URLs in its cache into Bloom Filter.

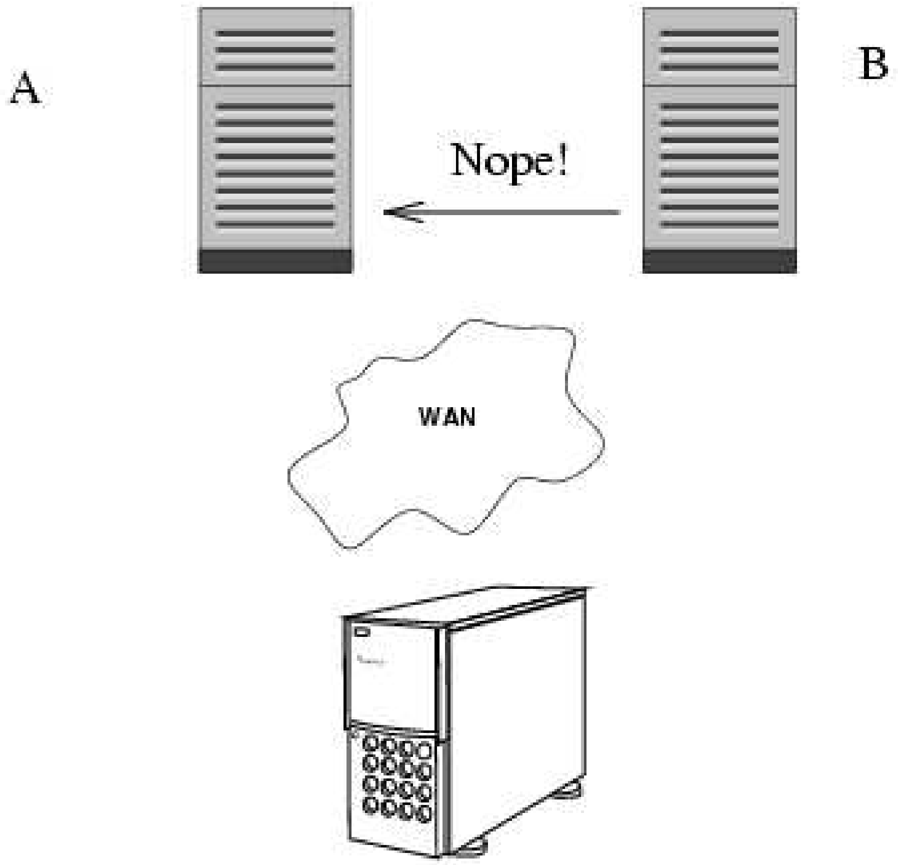
Proxies periodically exchange Bloom filters, so queries of other caches can be made **locally** without sending ICP message.

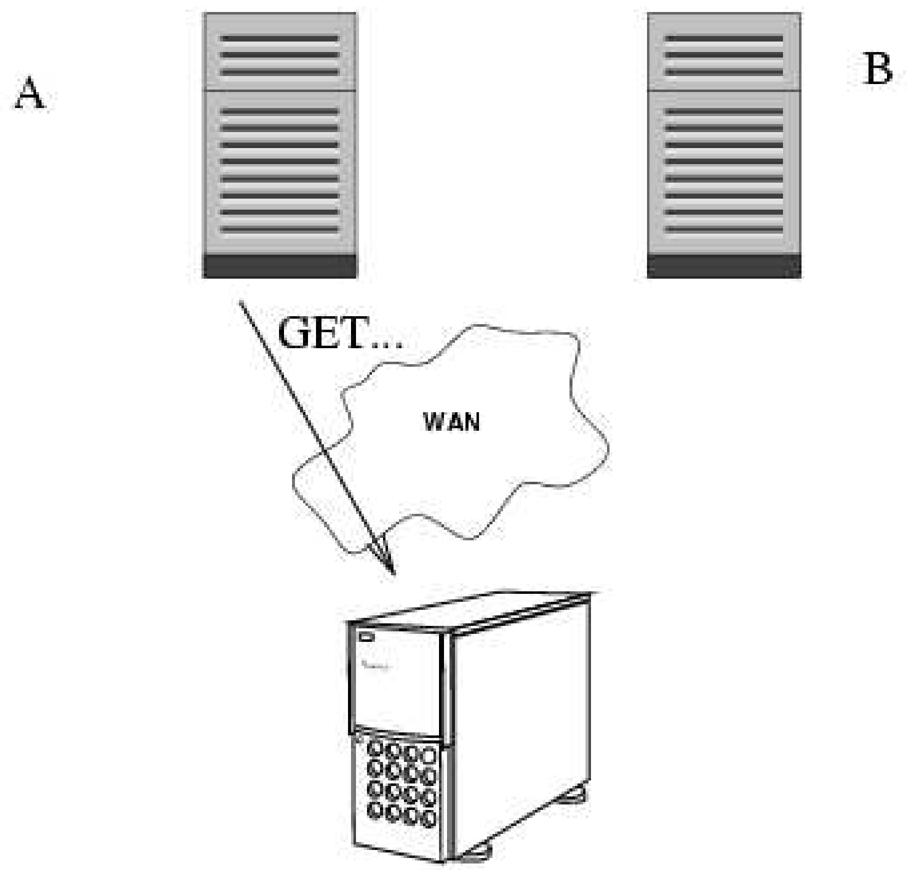
What's a false positive? Is it a big deal?

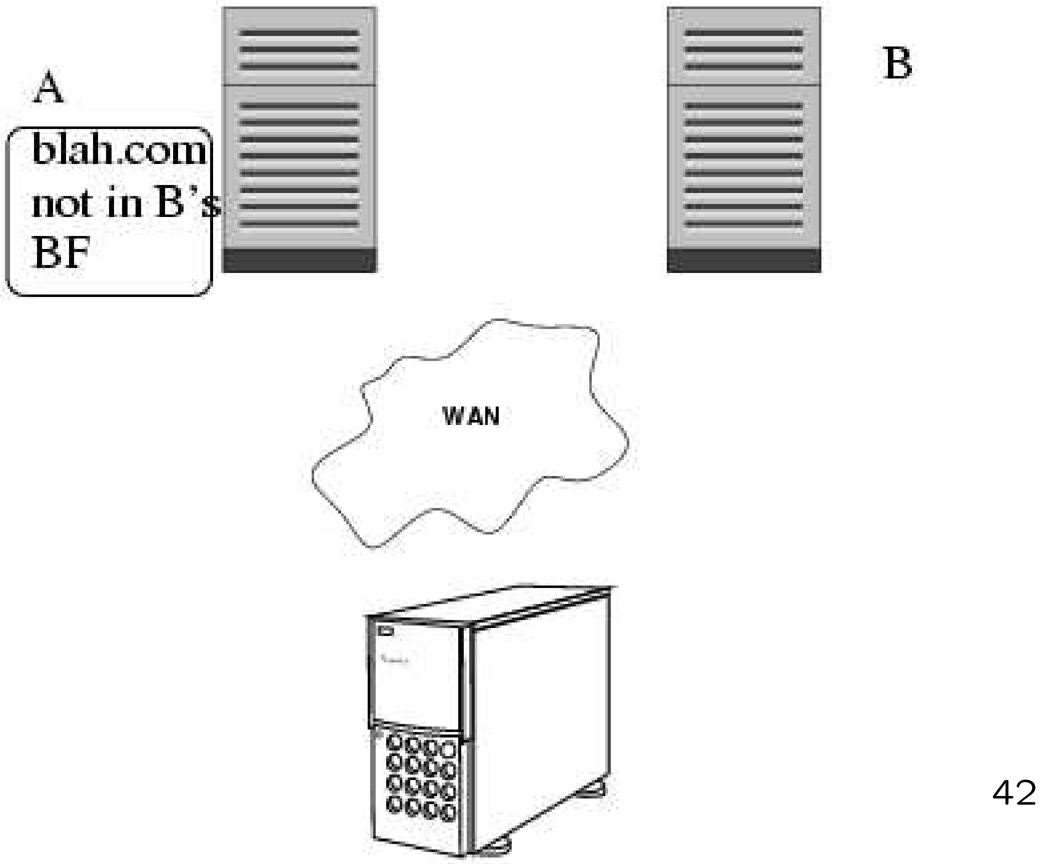
What's a false *negative*? Is it a big deal?

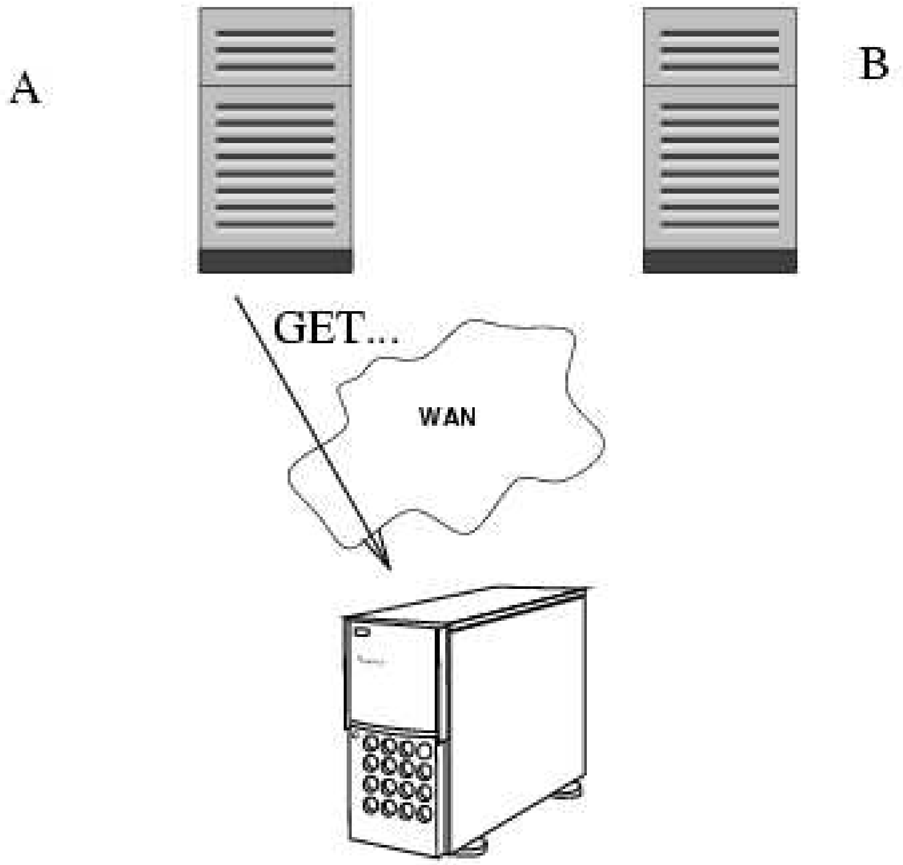

A
blah.com is in B's BF
B
WAN


A
blah.com
B
WAN


A
B
Nope!
WAN


A
B
GET...
WAN


A
blah.com not in B's BF
B
WAN


A
B
GET...
WAN

False Positives / Negatives

False positive: Proxy A thinks Proxy B has URL U cached. A asks for cached U, B responds back with “no”, A goes to actual website.

False negative: Proxy A thinks nobody has URL U cached, so it goes directly to website.

Result: a little extra traffic.

Problem: Deleting Items

Proxies remove pages from their cache, so they need to remove items from the Bloom filter.

How do we do this?

No Deletion Support

Recall our example Bloom filter of two items:

$H(x_0) = \{1, 4, 9\}$

$H(x_1) = \{4, 5, 8\}$

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

Can't delete one without clobbering the other since they share address 4.

No Deletion Support

Delete x_0:

$H(x_0) = \{1, 4, 9\}$

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	1	0	0	1	0

Query x_1:

$H(x_1) = \{4, 5, 8\} \longrightarrow$ No (False *Negative)*

Solution: Counting Filters?

In addition to storing bit at each address of filter, we store counter for each position.

Counter is incremented on insertion and decremented on deletion.

Bit in filter flipped when counter changes from 0 to 1 or 1 to 0.

In our example, we'd have the following counter array:

0	1	2	3	4	5	6	7	8	9
0	1	0	0	2	1	0	0	1	1

Counting Filters Issues

Counter Overflow? No chance! (kind of)

- Authors propose 4-bit counters enough
- In technical paper, with lots of math, show with 4-bit counters and $k < \ln 2 \cdot \left(\frac{m}{n}\right)$, probability of overflow

$$\leq 1.37 \times 10^{-15} \times m$$

- If counter overflows, just keep it at max value

Counting Filters: More Generally

What if we insert and delete multiple copies of the same item into a counting Bloom Filter? Can we reliably count the instances of items in the filter?

Counting Filters: More Generally

What if we insert multiple copies of the same item into a Bloom Filter? Can we use counting filters to count the instances of items in the filter? **NO!**

We insert ≥ 16 times and delete 15 times, and have a resulting **false negative**.

Recall: Summary Cache authors don't care so much about false negatives anyway.

We'll see a cooler use of Bloom Filters as counters later.

Outline

- Bloom Filter Overview
- Traditional Applications
- **Hierarchical Bloom Filters Paper**
- Less Traditional Applications & Extensions

Hierarchical Bloom Filterss

“Payload Attribution via Hierarchical Bloom Filters”, K. Shanmugasundaram, et al., *ACM CCS*, 2004.

Use Bloom Filter extension to store portions of packets for the purposes of payload attribution.

While SPIE is “packet digesting scheme”, their proposal is a “payload digesting scheme”.

Applications

- We possess piece of virus, shellcode, etc., and want to see if it was in any packets.
  - "Fornet: A Distributed Forensics Network"
- Track unauthorized disclosure of sensitive information from own network.

First Critique: Bad LaTeX

Variables typeset like these are lame: $offset$, $loffset$

BBFs

To support substring matching in Bloom filters, the Block-Based Bloom Filter (BBF) is introduced.

Payloads are broken into blocks of size s

Blocks are inserted along with their offset in payload: (content||offset).

BBF example from paper

0	1	2	3	4	5	6
ABR	ACA	DAB	RAC	ADA	RAC	ABA

We query BRACADAB, giving three alignments of 2 blocks each

- BRA CAD: not found
- ACA DAB: found at offset 1
- RAC ADA: found at offset 3, half at 5

? “double false positive of the BBF” at offset 2 for RAC ADA ?

BBF Drawback

Two packets made up of blocks $S_0S_1S_2S_3S_4$ and $S_0S_2S_3S_1S_4$.

Query for S_2S_1 would be a false hit.

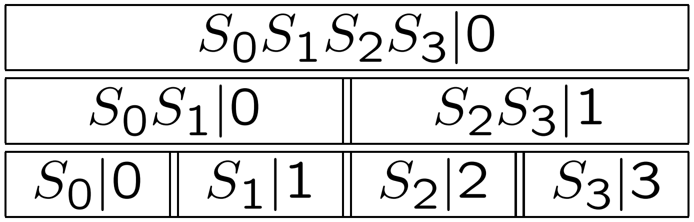
HBF example

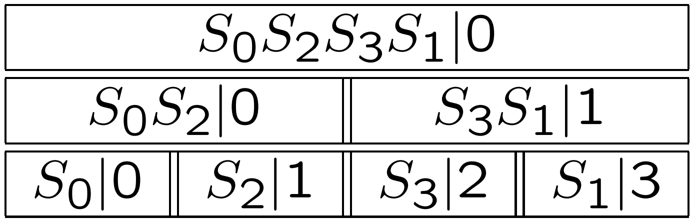
$S_0S_1S_2S_3 \mid 0$			
$S_0S_1 \mid 0$		$S_2S_3 \mid 1$	
$S_0 \mid 0$	$S_1 \mid 1$	$S_2 \mid 2$	$S_3 \mid 3$

We get additional check to limit false positives when searching for multiblock strings.

HBF: Small string drawbacks

For some small strings we still appear to have BBF style false hit:





We still get false hit on S_1S_3 since hierarchy doesn't capture two-block strings at odd offsets.

Several HBFs Used to make Payload Attribution System

- **Block Digest** (optional): hashes of (content) only

- **Offset Digest**: hashes of (content||offset). This is what was described above

- **Payload Digest**: hashes of (content||offset||hostID).

Attribution

- Destination Attribution: “not affected by spoofing”.
  - OK, but could get a lot of hits for internal worm/virus trying to propagate out of network

- Local Source Attribution: Can be accurate up to local subnet that HBF is in front of.
  - OK, I buy this

Attribution

- Foreign Source Attribution: Handwaves at using other forms of payload attribution that don't rely on source IP address
  - For connection oriented sessions, claim is can trust source IPs.
  - It seems that they don't really deal with spoofed source IPs: "...PAS suffers from denial of service attack as an attacker can overflow the list of host IDs used for full attribution".

Attacks on PAS

- Splitting payload into packets smaller than blocksize
  - Could make PAS stateful
- Stuffing payload with `nops` or equivalent
  - HBFs make PAS more robust than packet digesting
- Some other less interesting issues are mentioned

Experimental Results: FP_e and FP_o

	Basic False Positive Rates (FP_o)				
Blocks	.3930	.2370	.1550	.1090	.0804
1	1.00000	.999885	.996099	.976179	.933179
2	.063758	.064569	.048981	.036060	.026212
3	.012081	.002620	.000744	.000275	.000172
4	.000820	.000230	.000060	.000020	-
> 4	-	-	-	-	-

Experimental Results: FP_e and FP_o

	Basic False Positive Rates (FP_o)				
Blocks	.3930	.2370	.1550	.1090	.0804
1	**1.00000**	**.999885**	**.996099**	**.976179**	**.933179**
2	.063758	.064569	.048981	.036060	.026212
3	.012081	.002620	.000744	.000275	.000172
4	.000820	.000230	.000060	.000020	-
> 4	-	-	-	-	-

Don't use HBF to attribute blocks of length one!

BBF vs. HBF Under “Identical Memory Footprint”

Query Blocks	2	3	4	5
BBF	.049621	.035129	.000560	.000088
HBF	.016547	.000720	.000110	0.0

Presumably, BBF is better for one-block strings (this makes sense).

Tracking MyDoom

Searched for substrings of MyDoom virus in five days of traffic from large network of thousands of hosts.

Block size of 32 bytes used.

"Incorrect attributions" given total of 25,328 actual attributions:

Length	96	128	160	192	224	256
Incorrect	1375	932	695	500	293	33

Useful Data?

The number of incorrect per correct is **meaningless** since Bloom Filters do not allow false negatives

What about false positive rate? Disparity in charts?:

Length	96	128	160	192	224	256
Incorrect	1375	932	695	500	293	33

	Basic False Positive Rates (FP_o)				
Blocks	.3930	.2370	.1550	.1090	.0804
1	1.00000	.999885	.996099	.976179	.933179
2	.063758	.064569	.048981	.036060	.026212
3	.012081	.002620	.000744	.000275	.000172
4	.000820	.000230	.000060	.000020	-
> 4	-	-	-	-	-

Useful Data?

The number of incorrect per correct is **meaningless** since Bloom Filters do not allow false negatives

What about false positive rate? Disparity in charts?:

Length	96	128	**160**	**192**	**224**	**256**
Incorrect	1375	932	695	500	293	33

	Basic False Positive Rates (FP_o)				
Blocks	.3930	.2370	.1550	.1090	.0804
1	1.00000	.999885	.996099	.976179	.933179
2	.063758	.064569	.048981	.036060	.026212
3	.012081	.002620	.000744	.000275	.000172
4	.000820	.000230	.000060	.000020	-
> 4	-	-	-	-	-

Comments on HBF paper

Fairly simple construction for including varying length substrings in Bloom Filter.

Lots of handwaving about false positives.

Payload attribution not robust as long as it trusts source IPs.

Outline

- Bloom Filter Overview
- Traditional Applications
- Hierarchical Bloom Filters Paper
- **Less Traditional Applications & Extensions**

Using Bloom Filters to Measure Traffic Flow

“Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement”, A. Kumar, et al.,*IEEE INFOCOM*, 2004

We want to measure traffic flows. Flows can be defined by any combination of features, such as:

- IP address
- Ports
- Protocols

Measuring Flows

How can we measure **both small and large** traffic flows accurately?

- Counters? Does not scale for large flows and high link speeds.

- Random Sampling (like 1%)? Innacurate, especially for small flows.

Space-Code Bloom Filters

Measure **approximate** sizes of flows.

Note: Assume flow information is unencrypted.

We extend Bloom Filters, accepting some false positives in favor of speed and memory savings.

Of course, we don't use counting filters a la "Summary Cache"!

Space-Code Bloom Filters

Traditionally, we have set of hash functions $h_1, h_2, \ldots h_k$

A SCBF has l sets of k hash functions

$$\begin{array}{c} h_1^1, h_2^1, \ldots h_k^1 \\ h_1^2, h_2^2, \ldots h_k^1 \\ \vdots \\ h_1^l, h_2^l, \ldots h_k^l \end{array}$$

When inserting element x, we choose one of l sets **at random** and do normal BF insertion.

Space-Code Bloom Filters

When inserting element x, we choose one of l sets at random.

When querying element y, we iterate through all l sets of hash functions, and count number that hit, yielding multiplicity value $\widehat{\theta}, 0 \leq \widehat{\theta} \leq l$

We then use Maximum Likelihood Estimation (MLE) or Mean Value Estimation (MVE) to estimate multiplicity of y.

Coupon Collector's Problem

Given set of N elements, how many random samples do we expect before we hit all N?

Given that we've seen i elements, we will see a new element with probability $\frac{N-i}{N}$. So we expect to need $\frac{N}{N-i}$ samples before we get the $(i+1)$st element.

$$\frac{N}{N} + \frac{N}{N-1} + \frac{N}{N-2} + \cdots + \frac{N}{1}$$

$$N \sum_{i=1}^{N} \frac{1}{i} \approx N \ln N$$

How do we Choose l

We expect all l sets of hash functions hit after $\approx l \ln l$ insertions of same element x.

For example $l = 32$, $l \ln l \approx 111$. So how do we differentiate 200 vs. 400 insertions?

Can't make l arbitrarily large

Solution: Use Many l's: MRSCBF!

Multi-Resolution SCBF.

We use r filters, each an SCBF. We associate probability of insertion into each filter p_i where $p_1 > p_2 > ... > p_r$.

High p_i are high-resolution filters, capture small flow information.

Low p_i are low-resolution filters, capture large flow information.

Paper uses $l = 32$, $p_i = (.25)^{(i-1)}$

MRSCBF querying

Given a flow identifier, we compute **all** l functions on all r filters, yielding the set of multiplicities $\widehat{\theta}_1, \widehat{\theta}_2, \ldots, \widehat{\theta}_r$,

Doing MVE or MLE is too computationally complex

So we use "most relevant" filters

Most Relevant Example

Let actual multiplicity of x be 1000.

Filter at resolution 1 will have $\widehat{\theta} = l$

Filter at resolution $\frac{1}{1024}$ will have $\widehat{\theta}$ tiny, like 0 or 1

Probably best to use filter around $\frac{1}{16}$ or so.

Formalize Most Relevant Filter

If x matches θ hash groups, it would take about $\frac{l}{l-\theta}$ to match another hash group

The expected number of insertions given θ matches is

$$\left(\frac{l}{l} + \frac{l}{l-1} + \cdots + \frac{l}{l-\theta+1}\right)$$

Define *relative incremental inaccuracy* as

$$\frac{\frac{l}{l-\theta}}{\left(\frac{l}{l} + \frac{l}{l-1} + \cdots + \frac{l}{l-\theta+1}\right)}$$

and choose filter with smallest inaccuracy

SCBF Takeaway

Very cool way of using Bloom filters as counters.

Addresses the problem of “Summary Cache” counting filters which couldn’t effectively deal with multiple copies of the same data item.

Fabian's Extension: Privacy Preserving Observations

Interesting applications when many people have access to approximate counts of items.

Alice is interested in Bob's count of item X, but doesn't want to reveal her interest in X.

From Bob's count of a different, uninteresting item Y she can estimate his count of X.

So she asks for the count on Y and then deduces an approximate count for X.

Bloomier Filters

"The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables", B. Chazelle, et al., *ACM/SIAM Symposium on Discrete Algorithms (SODA)*, 2004

Associate a function value with f with each element in domain D of size N such that:

- Range R of f is size $2^r = \{\perp, 1, \ldots, 2^r - 1\}$ where $\perp$ means undefined.

- Subset $S \subseteq D$ of size n such that f is defined for $x \in S$ and $f(x) = \perp$ for $x \notin S$

Bloomier Filters: More Concretely

Let x_i be a set of elements separated into non-intersecting subsets A_i. For example:

$$\begin{aligned} A_0 &= x_0, \ldots, x_9 \\ A_1 &= x_{10}, \ldots, x_{19} \\ A_2 &= x_{20}, \ldots, x_{29} \\ &\vdots \end{aligned}$$

A Bloomier filter allows us to query an element y and **guarantees** the correct subset A_i if $y \in A_i$ for some i. If $y \notin A_i$ for all i, we **should** get $\perp$ unless we hit a false positive.

Extra Notation

Any element of range R can be encoded as a q-bit binary number in the additive group $Q = \{0, 1\}^q$. It is important that $2^q > |R|$.

We still have k hash functions $h_1, \ldots, h_k$ which return a value in range $1, \ldots, m$. In addition, we have one additional q-bit "masking value" M returned by hashing.

We define the "neighborhood" $N(t)$ of $t \in S$ as the results of the k hash functions, $\{h_1(t), \ldots, h_k(t)\}$

Let Π be a total ordering on the elements of S.

Immutable Table

The idea is to store $f(t)$ in the addresses of the table

$$\{h_1(t), \ldots, h_k(t)\}$$

such that

$$f(t) = M \oplus \bigoplus_{i=1}^{k} \text{Table}\left[h_i\left(t\right)\right]$$

The trick will be to figure out which address $h_i\left(t\right)$ to update for each element t so that we don't clobber another element's stored value.

Walk Through Ordering Example

We'll work through an example of creating an Immutable Bloom filter for the following parameters:

$$\begin{aligned} k &= 4 \\ m &= 10 \\ q &= 8 \\ n &= 4 \\ |R| &= 4 \end{aligned}$$

Where the range of f is the four values `0x11`, `0x22`, `0x44`, `0x88`

We will call the four elements of S $\{A, B, C, D\}$

Walk Through Ordering Example

	f	M	Neighborhood	τ	Π
A	`0x11`	`0x54`	1,3,6,7	?	?
B	`0x22`	`0xeb`	1,3,8,9	?	?
C	`0x44`	`0x07`	1,6,8,9	?	?
D	`0x88`	`0x2c`	2,3,8,7	?	?

f: Function value we want to store

M: Radom Mask computed from hashing

Neighborhood: The set of addresses computed from hashing

τ: The member of the Neighborhood which we will update

Π: The order in which we insert

Walk Through Ordering Example

	f	M	Neighborhood	τ	Π
A	`0x11`	`0x54`	1,3,6,7	?	?
B	`0x22`	`0xeb`	1,3,8,9	?	?
C	`0x44`	`0x07`	1,6,8,9	?	?
D	`0x88`	`0x2c`	**2**,3,8,7	?	?

f: Function value we want to store

M: Radom Mask computed from hashing

Neighborhood: The set of addresses computed from hashing

τ: The member of the Neighborhood which we will update

Π: The order in which we insert

Walk Through Ordering Example

	f	M	Neighborhood	τ	Π
A	`0x11`	`0x54`	1,3,6,7	?	?
B	`0x22`	`0xeb`	1,3,8,9	?	?
C	`0x44`	`0x07`	1,6,8,9	?	?
D	`0x88`	`0x2c`	~~2,3,8,7~~	2	4

f: Function value we want to store

M: Radom Mask computed from hashing

Neighborhood: The set of addresses computed from hashing

τ: The member of the Neighborhood which we will update

Π: The order in which we insert

Walk Through Ordering Example

	f	M	Neighborhood	τ	Π
A	`0x11`	`0x54`	1,3,6,**7**	?	?
B	`0x22`	`0xeb`	1,3,8,9	?	?
C	`0x44`	`0x07`	1,6,8,9	?	?
D	`0x88`	`0x2c`	~~2,3,8,7~~	2	4

f: Function value we want to store

M: Radom Mask computed from hashing

Neighborhood: The set of addresses computed from hashing

τ: The member of the Neighborhood which we will update

Π: The order in which we insert

Walk Through Ordering Example

	f	M	Neighborhood	τ	Π
A	`0x11`	`0x54`	~~1,3,6,7~~	7	3
B	`0x22`	`0xeb`	1,3,8,9	?	?
C	`0x44`	`0x07`	1,6,8,9	?	?
D	`0x88`	`0x2c`	~~2,3,8,7~~	2	4

f: Function value we want to store

M: Radom Mask computed from hashing

Neighborhood: The set of addresses computed from hashing

τ: The member of the Neighborhood which we will update

Π: The order in which we insert

Walk Through Ordering Example

	f	M	Neighborhood	τ	Π
A	`0x11`	`0x54`	~~1,3,6,7~~	7	3
B	`0x22`	`0xeb`	1,**3**,8,9	?	?
C	`0x44`	`0x07`	1,**6**,8,9	?	?
D	`0x88`	`0x2c`	~~2,3,8,7~~	2	4

f: Function value we want to store

M: Radom Mask computed from hashing

Neighborhood: The set of addresses computed from hashing

τ: The member of the Neighborhood which we will update

Π: The order in which we insert

Walk Through Ordering Example

	f	M	Neighborhood	τ	Π
A	`0x11`	`0x54`	~~1,3,6,7~~	7	3
B	`0x22`	`0xeb`	~~1,3,8,9~~	3	1
C	`0x44`	`0x07`	~~1,6,8,9~~	6	2
D	`0x88`	`0x2c`	~~2,3,8,7~~	2	4

f: Function value we want to store

M: Radom Mask computed from hashing

Neighborhood: The set of addresses computed from hashing

τ: The member of the Neighborhood which we will update

Π: The order in which we insert

Building the Bloomier Filter

0	1	2	3	4	5	6	7	8	9
`0x00`	`0x00`	`0x00`	`0x00`	`0x00`	`0x00`	`0x00`	`0x00`	`0x00`	`0x00`

	f	M	Neighborhood	τ	Π
A	`0x11`	`0x54`	1,3,6,7	7	3
B	`0x22`	`0xeb`	1,3,8,9	3	1
C	`0x44`	`0x07`	1,6,8,9	6	2
D	`0x88`	`0x2c`	2,3,8,7	2	4

Building the Bloomier Filter

0	1	2	3	4	5	6	7	8	9
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00

	f	M	Neighborhood	τ	Π
A	0x11	0x54	1,3,6,7	7	3
B	0x22	0xeb	1,3,8,9	3	1
C	0x44	0x07	1,6,8,9	6	2
D	0x88	0x2c	2,3,8,7	2	4

$$\begin{aligned}\text{Table}\,[\tau(B)] &= f(B) \oplus M(B) \oplus \bigoplus_{i=1}^{4} \text{Table}\,[h_i\,(B)] \\ \text{Table}\,[3] &= \texttt{0x22} \oplus \texttt{0xeb} \\ &= \texttt{0xc9}\end{aligned}$$

Building the Bloomier Filter

0	1	2	3	4	5	6	7	8	9
0x00	0x00	0x00	0xc9	0x00	0x00	0x00	0x00	0x00	0x00

	f	M	Neighborhood	τ	Π
A	0x11	0x54	1,3,6,7	7	3
B	0x22	0xeb	1,3,8,9	3	1
C	0x44	0x07	1,6,8,9	6	2
D	0x88	0x2c	2,3,8,7	2	4

$$\begin{aligned}\text{Table}[\tau(C)] &= f(C) \oplus M(C) \oplus \bigoplus_{i=1}^{4} \text{Table}[h_i(C)] \\ \text{Table}[6] &= \texttt{0x44} \oplus \texttt{0x07} \\ &= \texttt{0x43}\end{aligned}$$

Building the Bloomier Filter

0	1	2	3	4	5	6	7	8	9
0x00	0x00	0x00	0xc9	0x00	0x00	0x43	0x00	0x00	0x00

	f	M	Neighborhood	τ	Π
A	0x11	0x54	1,3,6,7	7	3
B	0x22	0xeb	1,3,8,9	3	1
C	0x44	0x07	1,6,8,9	6	2
D	0x88	0x2c	2,3,8,7	2	4

$$\begin{aligned} \text{Table}\,[\tau(A)] &= f(A) \oplus M(A) \oplus \bigoplus_{i=1}^{4} \text{Table}\,[h_i\,(A)] \\ \text{Table}\,[7] &= \texttt{0x11} \oplus \texttt{0x54} \oplus \texttt{0xc9} \oplus \texttt{0x43} \\ &= \texttt{0xcf} \end{aligned}$$

Building the Bloomier Filter

0	1	2	3	4	5	6	7	8	9
0x00	0x00	0x00	0xc9	0x00	0x00	0x43	0xcf	0x00	0x00

	f	M	Neighborhood	τ	Π
A	0x11	0x54	1,3,6,7	7	3
B	0x22	0xeb	1,3,8,9	3	1
C	0x44	0x07	1,6,8,9	6	2
D	0x88	0x2c	2,3,8,7	2	4

$$\begin{aligned} \text{Table}\,[\tau(D)] &= f(D) \oplus M(D) \oplus \bigoplus_{i=1}^{4} \text{Table}\,[h_i\,(D)] \\ \text{Table}\,[2] &= \texttt{0x88} \oplus \texttt{0x2c} \oplus \texttt{0xc9} \oplus \texttt{0xcf} \\ &= \texttt{0xa2} \end{aligned}$$

Building the Bloomier Filter

0	1	2	3	4	5	6	7	8	9
0x00	0x00	0xa2	0xc9	0x00	0x00	0x43	0xcf	0x00	0x00

	f	M	Neighborhood	τ	Π
A	0x11	0x54	1,3,6,7	7	3
B	0x22	0xeb	1,3,8,9	3	1
C	0x44	0x07	1,6,8,9	6	2
D	0x88	0x2c	2,3,8,7	2	4

Why do we Need M?

M, the “masking value” is used to eliminate false positives by effectively randomizing lookup misses.

We have $2^q > |R|$, so that $f(y) = \perp, y \notin S$ with probability $\frac{|R|}{2^q}$

Easy example: $0 \in R$, and we don’t use a mask M. With a table initialized to 0, lookups that hit addresses with no values (or values that sum to 0) would be false positives.

Why we Need M

0	1	2	3	4	5	6	7	8	9
0x00	0x00	0x01	0x02	0x00	0x00	0x03	0x04	0x00	0x00

$R = \{0, 1, 2, 3, 4\}$

$N(y) = \{1, 4, 5, 9\}$ for $y \notin S$

y is now a false positive.

“Mutable” Bloomier Filter

We use an extra table and one level of redirection so that we can update $f(t)$ for any $t \in S$.

Instead of storing $f(t)$ in the first table, we store the value $i \in \{1, \ldots, k\}$ for which $h_i(t) = \tau(t)$.

Then in second table, we store $f(t)$ in TABLE2 $[\tau(t)]$.

So What's the Catch?

There is (at least) one major downside to Bloomier filters. Did you catch it?

Major Downside of Bloomier Filters

"Mutable" Bloomier filter refers to being able to update $f(t)$ for $t \in S$. Membership in S cannot change; the set S itself is **immutable**.

- Must know entire set S upon creation
- Cannot update filter with new elements after its creation

This would seem to severely limit its practical use.

Conclusions

Bloom Filters and their extensions are useful tools for a variety of applications in the field of security.

I happen to think HBFs are not among the coolest applications.

Again, the **Bloom Filter Principle**: "Whenever a list or set is used, and space is consideration, a Bloom filter should be considered. When using a Bloom filter, consider the potential effects of false positives."

Matchings: More Notation

We say a matching τ respects (S, π, N) if

- for all $t \in S, \tau(t) \in Nt$
- if $t_i >_{\sqcap} t_j$ then $\tau(t_i) \notin N(t_j)$

In English, if we have $\tau(t)$ for all $t \in S$, then this gives an address in every element's neighborhood for which we can update the table without clobbering any previously entered data!

Finding the Ordering Π and Matching τ

We say $h \in \{1 \ldots m\}$ is a *singleton* if $h \in N(t)$ for a unique $t \in S$.

We use the term TWEAK$(t, S,$ HASH$)$ to refer to the smallest singleton of t (so that we have a defined one to pick in case t has multiple singletons).

If every t had a singleton, we'd be fine, we could use $\tau(t) =$ TWEAK$(t, S,$ HASH$)$.

Find Singletons Recursively

1. For all t with a singleton, set $\tau(t) = \text{TWEAK}(t, S, \text{HASH})$. .

2. Remove all these elements with singletons from S.

3. Put these elements in Π after those still in S but before those currently in Π

4. Repeat until all elements are ordered. We fail if their are elements left with no singletons.