國立清華大學

資訊工程學系

博 士 論 文

Efficient and Adaptive Stateful Replication in High Availability Clusters

適用於高可用度叢集之高效能且動態調整的狀態複製機制

研 究 生：馮乙軒

指 導 教 授：黃能富 教授

內容

Abstract

Kinds of stateful stream process engines (SPEs) track a large number of concurrent flow states and replicate them to backups to provide reliable functionality in high availability clusters (HACs). Under high traffic loads, existing solutions in such HACs are expensive because of precise stateful replication. In this dissertation, I study a suite of two methods to address this issue: randomization on replication messages and a replication scheme designed for when system is going to be overloaded.

Two new hierarchical structures called Flow Digest (FD) and Multi-Level Counting Bloom Filter (MLCBF) are proposed as low resource-consuming solutions of stateful replication. To the best of my knowledge, it is the first time that randomization has been introduced for stateful replication of HAC in the literature. Analysis and extensive tests are employed to evaluate performance and tradeoffs of the proposed techniques. Most importantly, MLCBF is quite as simple and practical to implement and maintain.

Furthermore, an adaptive scheme, called as *dynamic lazy insertion*, is designed to prevent replication from overloading system and optimize pass-through performance of HAC dynamically. Testbed evaluation demonstrates its feasibility and effectiveness in real situation.

1. Introduction

High Availability Clusters (HACs) are widely deployed on the highly-valuable links (e.g., the highest bandwidth links) of enterprises, campuses, and ISP networks. Fig. 1 shows that an HAC is composed of a number of stateful stream processing engines (SPEs) [1],[2] that process input stream (e.g., assembled TCP segments or URL requests) continuously, perform stateful tracking by finite state machine (FSMs), and produce output in real-time. Many stateful SPEs only need a simple *key-and-state* storage (called as *state table*) to manage the stateful results (i.e., states) of continuous tracking. If such state is lost, SPE will not possibly return an expected output.

In HAC, the SPEs of identical functionality (e.g., traffic classification and intrusion prevention) must cooperate to handle a failure and flow migration due to load balancing [3]. Because the computation of stateful SPE can be considered as deterministic (i.e., the same output by the same sequence of input [4]), SPE can produce correct output by owning the tracking result so far of a flow. Thus, SPE utilizes passive replication [5],[6] to synchronize the state changes to its backup SPEs to ensure consistent service from the point of views of end hosts.

As shown in Fig. 1, all SPEs in HAC share a LAN reserved for replication in practice and employ replication management protocol (RMP) to maintain state consistency. Yet, the efficiency of replication is critical for the performance of SPEs and HAC. My study on real testbed shows that replication using precise update messages can incur considerable resource costs, including CPU, memory, and bandwidth requirements. Under heavy traffic loads, an HAC certainly will not scale well with the maximum deployment number of SPEs because high bandwidth traffic on replication. Second, the pass-through throughput of HAC is limited to the minimum performance of a

sequence of SPEs on a pass-through link. SPE not only processes pass-through input stream, but also synchronizes its state information over network and stores incoming replicas to state table. I find that resource contention inside SPE between pass-through and replication tasks impedes the performance severely. However, pass-through processing must be optimized for the overall performance of HAC.

In this dissertation, my focus is to provide an efficient RMP for *key-and-state replication* amongst SPEs in HAC. Two kinds of *stateful replication* are considered: state-machine replication (or *state replication*) and *membership replication* (e.g., [7],[8],[9],[10]). State replication refers to the task of synchronizing the key and state transition (e.g., from 2 to 8) of an active flow (or *item*) to the backup SPEs. The flow-state change is in the range of 0 and N (state 0 indicates a deletion). In membership replication, the information whether a flow is in-set or not is propagated.

I present a new compact data structure, called *Multi-Level Counting Bloom Filter* (MLCBF), and specifically show how this data representation by randomization can be used to improve the performance of stateful replication. Analysis and extensive experiments are employed to explore the properties of my algorithms and evaluate replication efficiency by several metrics, including accuracy, maximum achievable load, search costs, resource consumption, and operational latency. The results show that my methods reduce the requirements of stateful replication on network and memory significantly, and also provide it with small and constant latency time.

Next, I propose an adaptive method to prevent system from being overloaded by the replication of TCP flows, which is the majority of the Internet traffic. The intuition is to prioritize the pass-through processing over replication at system overload to maintain optimal throughput dynamically. This self-tunable method measures system utilization and flow lifetime distribution to adjust its decision adequately.

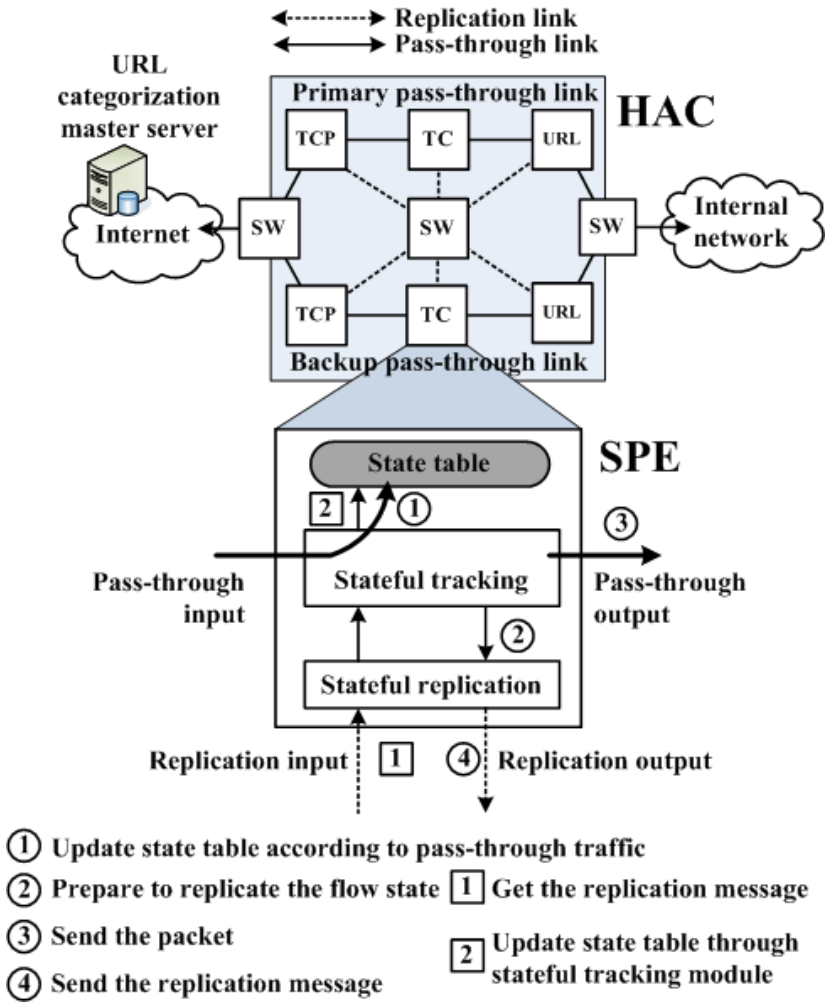


Fig. 1. A practical example of SPEs employing replication in an HAC of active/backup scheme. Primary and backup SPEs of TCP tracking, traffic classification, and URL categorization, are located on two pass-through links. Two edge switches distribute traffic according to HA scheme. For the general data flow inside SPE, the numbers in cycles and squares represent the steps of pass-through and replication processing. Notice the pass-through throughput of HAC is limited to the minimum of the

Testbed evaluation demonstrates its feasibility and effectiveness in real situation.

1.1. Background and Related Works

1.1.1 High-Availability Clusters and Operation Modes

Availability Clusters (HACs) are widely deployed on the highly-valuable links (e.g., the highest bandwidth links) of enterprises, campuses, and ISP networks. Generally, an HAC consists of pairs of stateful stream processing engines (SPEs) for functionalities such as TCP tracking and URL categorization. The goals of HACs are to automatically counter planned outages (e.g., a system upgrade) and unplanned outages (e.g., a hardware failure), avoid a possible bottleneck to optimize throughput, and, most importantly, remove single point of failure.

For redundancy, if an SPE on the pass-through link in operation is out of service, pass-through traffic (e.g., TCP flows) is passed to the backup link (i.e., a *failover*) immediately. SPEs of identical functionality in an HAC must maintain key-and-state consistency among them to ensure consistent service in case of a failure.

Figure 1 illustrates four statuses of a generic HAC which contains two sequences of SPEs which are connected by two *pass-through links*. The SPEs process pass-through traffic and replicate key-and-state information simultaneously to their backups through *replication links*.

Two distinct HA schemes are generally available, i.e., active-backup (AB) scheme and active-active (AA) scheme. Figure 2a illustrates, in AB scheme, the load-balancing switches on the boundary of HAC direct all traffic to the primary link normally. As shown in Fig. 2b, if a failure on the network link is detected, the switches then redirect the traffic to the backup pass-through link. Notably, in Figs. 2a and 2b, the SPEs of TCP state tracking and URL categorization perform stateful replication according to the input stream in real time.

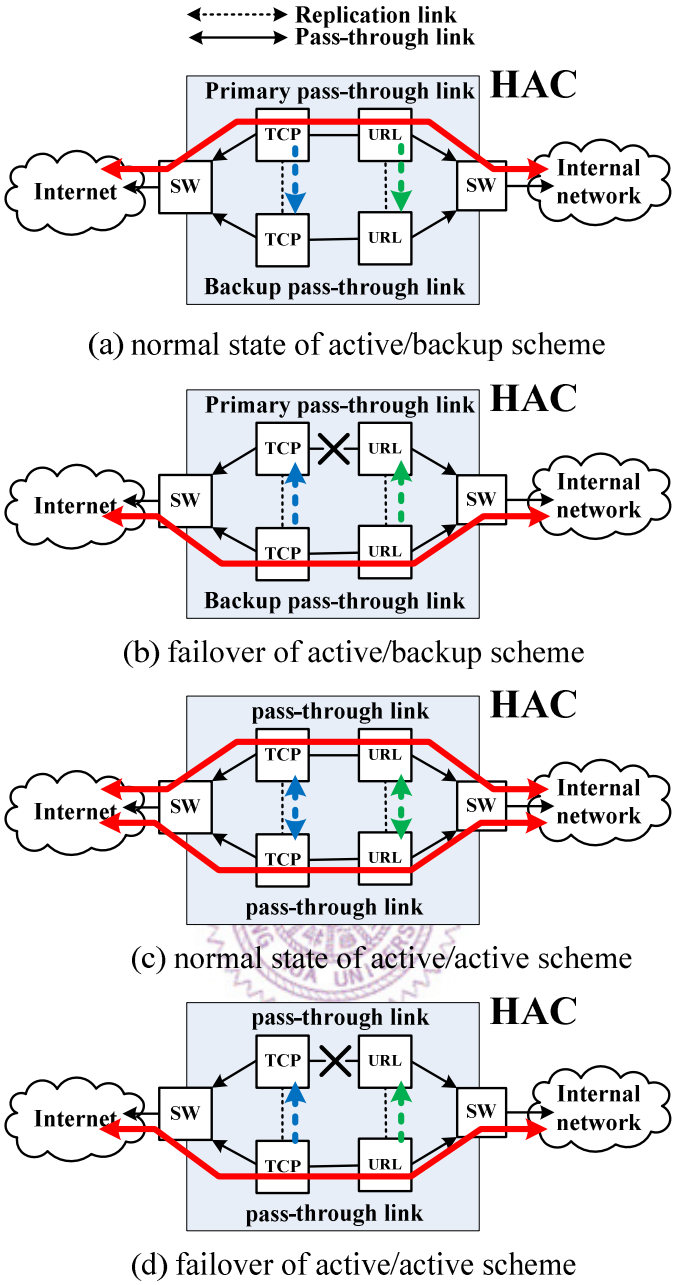


Fig. 2. Four statuses of stateful HACs consisting of dual-string pass-through links: (a) normal state of active/backup scheme, (a) failover of active/backup scheme, (c) normal state of active/active scheme, and (d) failover of active/active scheme. Notably, the blue and green dotted lines represent replication traffic. The red solid lines indicate the pass-through traffic from and to the Internet.

Figure 2c demonstrates that, in active/active (AA) scheme, the edge switches of HAC distribute traffic loads evenly across the pass-through links as well as ensure that a connection is sent to the same link bi-directionally. In both HA schemes, SPEs rely on

replication for reliable service in face of failure and flow migration by edge switches due to load balancing. Finally, the network failure on the pass-through link in Figs. 2b and 2d is recovered as soon as possible by network operators in practice. Then, the traffic loads of HAC are processed by following those shown in Figs. 2a and 2c again after the recovery.

1.1.2 Reliable Transport Solutions

In [2], the issue of fault-tolerant SPEs by active replication is discussed. In contrast, we focus on the operational performance of an HAC by passive state replication. State machine replication is a popular technique to support reliable services. OpenBSD uses the pfsync to replicate state information of the IP Filter. The ct_sync integrates tightly with the netfilter to give a Linux solution. Both the pfsync and ct_sync rely on the propagation of at least three precise messages for a flow via multicasting. To support fault-tolerant transport protocols, many solutions have been proposed such as fault-tolerant TCP [11],[12],[13],[14],[15],[16] OS mechanism [17],[18], and new protocols [19],[20]. However, achieving reliable service remains a challenge because end-to-end reliability is limited to the weakest communication segment. Our work complements these studies by focusing on the HA techniques.

1.1.3 Variants and Applications of Legacy Bloom Filters

By using a bit vector V of length m and k independent hash functions with range [1,m], a standard Bloom filter (SBF) [21],[8] yields an extremely compact and one-way data structure that supports the membership queries to a set $A = \{a_1, a_2, a_3,\ldots, a_n\}$ of n elements in constant time. The Bloom filter causes the space requirement to fall significantly below the information theoretic lower bounds for error-free data structures and can reduce the space by at least one order of magnitude. It achieves this efficiency at the cost of a small false positive rate, but has no false negatives. The term *false positive* describes the item not in the set is classified as being in the set in a query. The term *false negative* describes the item in the set is classified as not being in the set in a query. There is a tradeoff between the

size *m*, the number *k*, and the possibility of false positive *f* as the following Equation 1 and it will give a minimum value when $k = \ln 2 \times m/n$. Fig. 3 depicts the theoretical error rate with 4 or 8 hash functions and maximum active connections respectively versus different bit-vector sizes. For example, for *n* = 1M and *k* = 4, if we choose *m* = 1M×10 = 10M (bits), then the *f* will be equal to 1.2%.

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

The SBF and its variants are widely used in practice when the storage is at a premium (e.g., the memory space is too valuable to store the large volume of data) or an occasional false positive is tolerable (e.g., [7]).

DLCBF [22],[23],[24] is a simple and practical alternative to CBF. Compared to CBF, DLCBF saves a factor of two at least on memory for the same P_{FP}. For state and membership replication, motivated by Multi-level Hash Table [25],[26],[27],[28], we introduce skewness to DLCBF to improve the run-time costs, P_{FP}, and space utilization, and retain its benefits of simple construction, small filter size, and, most importantly, single message per update. To the best of our knowledge, our work is the first attempt of using MFFs to minimize the resource requirements of stateful replication. Other examples of using imprecise representation in replication are distributed metadata management [29] and resource routing on P2P networks [30],[8]. Finally, many variants of Bloom filters [21] have been presented, including filter compression (e.g., [31],[32],[33]). By contrast, symbol replacement provides another possibility by converting incremental messages in real-time, instead of compressing the filter itself.

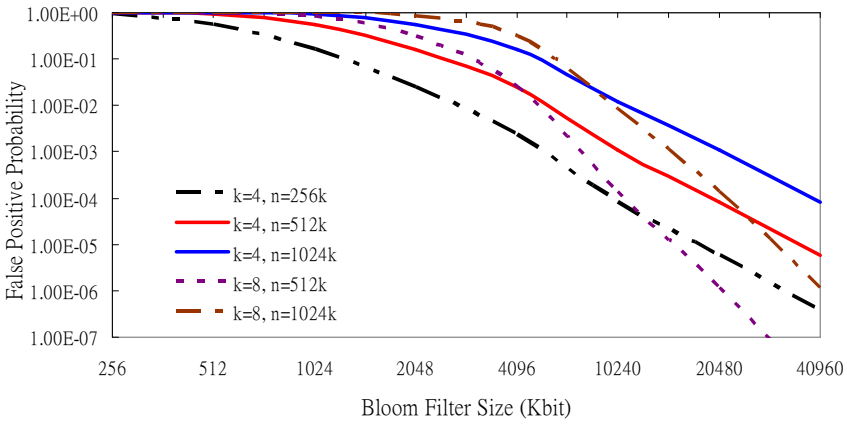


Fig. 3. False positive rates in log scale vs. bit-vector sizes under different maximum concurrent connections that a primary holds.

1.2 Motivation and Design Goals

In my preliminary tests, the performance of TCP state replication (6 states for each flow) is measured to understand the bottlenecks of stateful replication. The observations from the results can be viewed as the motivations of this work.

First, Fig. 1 shows the architecture of SPE of two existing precise replication solutions: OpenBSD pfsync and Linux ct_sync. In receiver, both replication and pass-through loads share the same state table. This makes free entries and semaphore locks of state table become another type of resources. The long-lasting flows replicated from a sender may occupy considerable entries which are only of use when necessary (e.g., a failover and flow migration). In addition, high-rate short flows aggravate resource contention of receiver, thereby interfering with performance.

Second, an intuitive example is used to explain the cost of precise replication: assume that steady flow rate is 20k connections per sec (cps), a precise replication message contains <four-tuple,state> whose size is 100 bits, and update interval is 30 sec. Then, a single update introduces 20k(cps)x6(states)x30(sec)=3,600k messages and 360Mb of memory/network costs. Obviously, precise replication incurs considerable costs into SPEs and replication links under high-rate traffic. The utilization law tells us that the reduction on resource requirements (e.g., latency per insertion or bandwidth consumption per update) increases the maximum number of task completions per time unit; namely, the capacity of SPE and scalability of HAC.

Third, Counting Bloom Filter (CBF) [7],[8] and variants are widely used by membership replication of network applications (e.g., [7],[8],[9],[10],[29],[30]). However, the bandwidth cost by CBF for state replication is higher than precise replication for some applications like traffic classification. Finally, CPU load is dominated by the number of incoming pass-through packets and replication tasks.

When system is overloaded, pass-through processing must get more resources. Replication should be de-prioritized for optimal pass-through throughput.

Motivated by above observations, my design goals are: 1) an architectural separation of pass-through and replication processing, 2) an efficient hashing structure for stateful replication at very low runtime costs. The structure has to be as simple to implement and maintain as possible for high-speed SPEs, and, 3) finally, a scheme to prioritize pass-through tasks over replication ones for optimal pass-through throughput at system overload.

2. Stateful Replication in HA Clusters Using Hashing Structures

2.1 Membership Replication by CBFs in Firewalls

In contrast to existing membership replication solutions in HACs, Flow Digest (FD) [9],[10] improves the replication procedures by two factors: (1) the new primary only references the state table when it takes over the traffic processing and (2) a new data structure is designated to save bandwidth requirement. The FD scheme can be divided into three phases: *summary*, *update* and *recovery*. They are described briefly as follows.

In the summary phase, the primary collects all active entries in state table into an FD structure which is constructed based on the Bloom filter which will be described later. During the update phase, a message is sent to the slaves by multicasting and a slave only saves the received data without any operation on its state table. The scheme shifts to the recovery phase after a failover, and the new primary reconstructs the state table in a packet-driven fashion by querying the stored FD structure (i.e., a backup SBF) to see whether an incoming TCP packet might be active classified by the old primary. If it seems true, the packet passes the recovery process. Otherwise, the new primary drops the packet.

The SBF and its variants are widely used in practice when the storage is at a premium (e.g., the memory space is too valuable to store the large volume of data) or an occasional false positive is tolerable (e.g., [7]).

2.1.1 Bloom Filters as FD Representation

To improve the bandwidth utilization, instead of notifying the slave about every

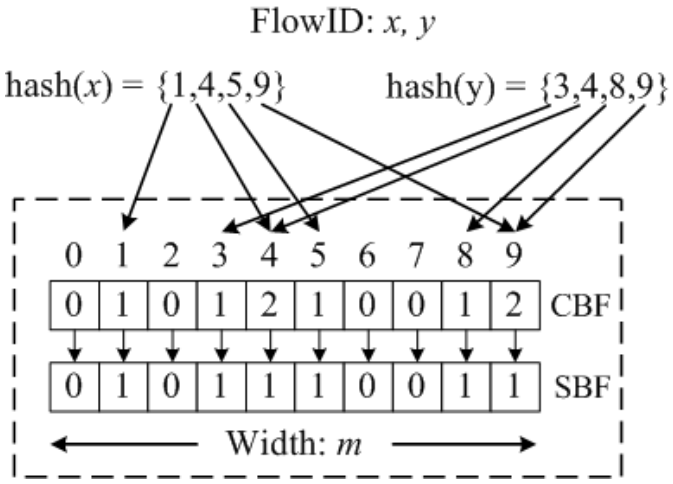


Fig. 4. An FD example to summarize the existing active connections. Two Bloom filters are used in an FD structure and the primary only sends the SBF to slaves.

change to every state entry, the main idea of FD is to provide a snapshot of the primary state table and therefore it requires much less frequent propagations of state changes. As shown in Fig. 4, the proposed method keeps a data structure which contains two filters in the memory of a primary, including a counting Bloom filter (CBF) [7],[8] and an SBF. For example, to check a flowID against CBF and SBF summary, a flowID is hashed with k hash functions (In Fig. 4, k = 4) and, after applying the modulus to resulting values by size m, the bits corresponding to the results in filters are checked. In practice, an FD structure is implemented by an m-bit array where each bit is associated with a counter (whose size is b bits). The operations to maintain an FD data during the summary phase include:

- Insertion for a new flowID. When the primary changes its state entry of a connection as established, it then inserts the connection to the CBF by hashing the flowID and incrementing all resulting counters by 1. When a counter overflow occurs, this counter stays at its maximum value. With the identical hash functions and size m, an SBF, a bit-wise array, is used to provide the

abbreviated information of the CBF and the corresponding bit is set from 0 to 1 when the associated counter is increased from 0 to 1.

- Deletion for a flowID. When a state entry is removed from the state table due to a timeout of inactivity or a connection termination, the counters of CBF touched by hashing results of the flowID are decremented by 1. When a counter deletion causes a value in the CBF to change from 1 to 0, the corresponding bit in the SBF is reset to 0.

In the update phase, two message formats are used according to update sizes. In a high connection-rate environment, the primary sends the SBF to slaves directly to achieve lower update overhead. One can thus view an update as the snapshot of a state table propagating outward from the primary. As described before, except for shifting to the recovery phase, the state table in a slave will not be accessed, clearly quite different from current SRPs, and this prevents the slave from merging every incoming message its local state table in real-time. Every update by sending entire SBF is self-contained, so that the slave just replaces the stored data with a received SBF directly.

Sending an SBF (e.g., 4,096K bits) to slaves is clearly not economical at all in a slow traffic environment (say, below 1,000 cps), because the update information contained in an SBF may be only slight different from the one before it. This makes the incurred overhead exceed the benefits of the proposed method. An alternative is to use a difference mechanism which forms an update message (called *difference message*) issuing changes. A difference message is composed of a list of 32-bit entries and every entry uses the most significant bit for specifying whether the bit should be set to 0 or 1 and the rest bits for specifying the SBF index to be modified. The choice of which message format to use will depend on what the size of an

update will be. Obviously, if the difference between two updates is small, it is more saving to use difference messages rather than entire SBF.

After a failover, a new primary comes up and enters to the recovery phase. It bases on the backup SBF from the old primary to process the incoming traffic and reconstruct its state table in a packet-driven fashion. A SYN packet will not be filtered by the backup SBF. When there is a non-SYN TCP packet arriving and it can not be found in a state table lookup, the new primary performs a membership testing on the backup SBF and the bits corresponding to the result are checked. If all bits are positive, then the new primary accepts this connection into the state table. The duration of a recovery process is equal to the default timeout of a state entry (say, 60 to 120 seconds).

Two possible errors in FD are defined:

- False hit: A connection is not active for the old primary, but the backup SBF answers a positive for the query.
- False miss: A connection is active for the old primary, but the backup SBF indicates it is not.

Two possibilities for cause of a false miss: the false negative from the backup SBF and the state inconsistency between the primary and slave. The problem of state inconsistency will be discussed later. In FD, using a counting Bloom filter as the representation of active connections incurs small false positive and false negative rates. The false miss affects the performance of pass-through TCP traffic, because the packets of a misclassified connection will be dropped continuously in general, including all re-transmissions. In order to reduce the probability of yielding an overflow event (the cause of a false negative), the counters in our array need to be

large enough to avoid the counter overflows. On the other hand, the counter size needs to be made as small as possible to save main memory. According to the analysis of [7],[8], it reveals that 4 bits per counter should be sufficient for most applications, so do the counters of a CBF.

Occasionally, a Bloom filter may return false positives. Note that the false hits do not affect the recovery of active connections. But, this means that a packet without the previous 3-way handshake procedure passes through the filtering of new primary because the SBF returns a positive answer in the recovery phase. A network attack may use this as ACK floods to exhaust the system and network resources of victims. The packet rate of attack was reported as high as 200,000 pkts/s [34],[35]. In the following, we briefly discuss how to minimize the syndrome of false positives.

First, since the packet reaches the particular endpoint through a false positive, depending on the implementation of OS, the endpoint either replies back a TCP RST packet or an ICMP unreachable packet to sender and this rogue ACK packet will be discovered eventually when the new primary receives this network error packet. Second, a DDoS prevention module is popularly equipped in a modern firewall or IPS. When the ACK flood is detected, for optimizing the utilization of state table, the new primary can enable the aggressive aging [36]. Third, it is noteworthy that false positives only exist after failing over and the duration of this potential risk is short. Finally, the failover timing and the parameters of an FD implementation should not be predictable and open to network attackers. Therefore, we believe that it is difficult for attackers to use the feature of Bloom filter to bypass the security filtering, especially when the system uses DDoS detection as a front-end to prevent the state table explosion [37].

A third kind of error introduced into HA comes from the overhead of failover process,

including the failure detection and primary election. Instead of an efficient state replication, a fast network-level failover mechanism is required to minimize failover overhead.

The number of hash functions influences two competing forces: the probability of collision and the capability of the discrimination between flowIDs. Besides, in FD, a larger number of hashes may increase update size in a difference update. We compare three hash functions. First, MD5 is an open and well established message digest algorithm and is chosen for its well distributed and fix-sized output values. Four hash values are built by calculating the MD5 checksum [38] of a flowID, which yields a 128-bit data, then dividing it into four 32-bit words. The indexes into the SBF and CBF come from applying the modulus to resulting 4-byte values by size *m* finally. Second, a modification of MD5 in [39] (called MD- by ignoring the G, H, and I functions) is used to improve the throughput and has a close result with MD5. Third, we use a "fast" hash family based on the shifting, AND, and XOR operations on flowIDs. Although this solution is less effective than MD5 and MD- in avoiding collisions, it is characterized by much lower computational overheads and can be implemented by a limited and simple instruction set (e.g., RISC-based network processor units).

2.1.2 Update Criteria

In order for the recovery phase to maximize the reconstruction performance, the backup FD at each slave must be kept most up-to-date with the primary state table of the primary. Ideally, the primary should continue to propagate the last information on itself. However, this increases the update overhead, because frequent propagations of update messages incur a non-trivial cost on system performance and should be avoided. The key to the scalability of an HA cluster is to ease the load on

the system and network. Thus, a backup SBF without updated in real-time helps the scalability; rather, the update can be triggered upon simply a periodic basis, or a threshold basis. These two methods are transparent in operation. The update can occur upon a regular time interval, or when a certain percentage of the variation on existing connections compared to the previous update is not reflected to a slave. That is, FD uses an occasional message burst for providing a table snapshot to replace the continuous small messages for updating state changes.

Because a failover is likely to be prior to an update event and results in a state inconsistency, a periodic or threshold-based update method poses a potential risk to a new primary of dropping the packets of active connections in the recovery phase. Two approaches are used in the recovery phase to solve the problem: TCP cold start [40] and the intentional block mechanism.

TCP cold start lies on the assumption that the new primary lies between a trusted network (e.g., internal network) and an untrusted network (e.g., external network). If a non-SYN packet which fails on lookups both in the state table and the backup SBF comes from the trusted network, it will pass the recovery filtering and the state entry will be instantiated into the state table. However, if the packet is from the untrusted network, the new primary forwards the packet to the destination by stripping off its payload and decreasing the sequence number in the header. In this way, if the packet is indeed from a reliable connection, then the endpoint in the trusted network will respond this “keep-alive” packet with an ACK. Then, because the ACK comes from the trusted network, the new primary instantiates the corresponding state entry and continues as usual.

Using the intentional block mechanism, for the same packet above, the new primary inserts the corresponding flowID into the backup SBF regardless of packet direction

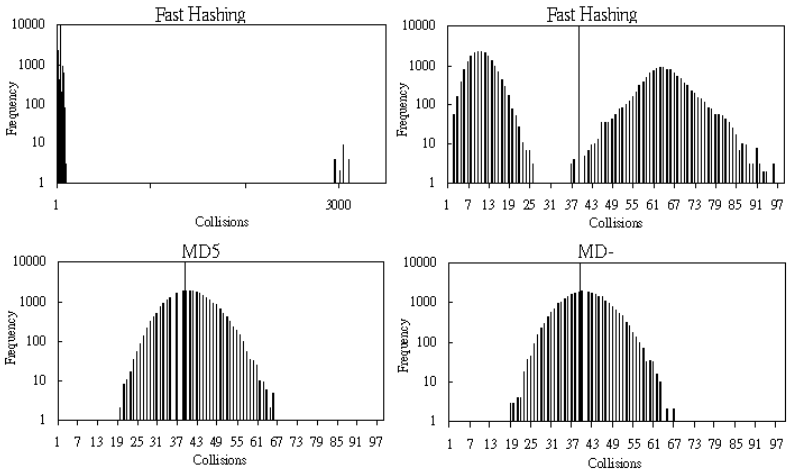


Fig. 5. Example: the collision distributions of fast hashing, MD5, and MD- (α =40).

and simply drops the packet. Again, if the connection is active, the source endpoint will initiate a retransmission. This time, the new primary will get a positive answer from the backup SBF for the retransmitted packet and forward the packet to its destination.

In summary, by taking advantage of the state held at the receiver and the sender, in the recovery phase the new primary can continue the active connections which are lost in state replication. While supporting these methods introduce additional bandwidth consumption on the pass-through path, the duration of recovery phase is short and, more importantly, these schemes relieve the load in propagating the update messages immediately for state consistency and allow a larger update interval or threshold to reduce the network overhead.

2.1.3 FD Components and Trade-offs

Besides the update strategies, the FD choices vary in many dimensions and are intertwined when determining the amount of resource requirements. In this section, the choices on *m* and hash functions and their effects are discussed.

A. Selecting Table Size

A number of factors impact the choice of the size *m*. First, in general the memory needs of FD are determined by n_c at the link. Namely, *m* should vary with the cluster maximum capacity of a real network, including pass-through and possible migrated loads. Second, despite a higher bandwidth cost, a larger *m* is preferred due to a smaller theoretical error rate as shown in Fig. 6 and a 3% *F* might be acceptable for an application of Bloom filter. However, *F* varies over time and links in practice, depending on n_a. For example, the traffic with high connection rate but low concurrent flows is common for a link before a web server. On the other hand, backbone links usually remain at loads of 35–85% and contribute both high rate and high flow number. If $n_a < n_c$, the partial fulfilled SBF will perform better than an expected *F*. Hence, we define the Table Load (*L*) as n_a/n_c to describe a temporary density of active flows in the FD structure.

Besides the false ratio, *L* also affects the amount of incremental updates. As mentioned before, a SBF-bit alternation is identical to an incremental event. This indicates that FD with a low n_a or a big *m* generates more events for a given connection rate. Next, we analyze the theoretical probability of an incremental event. As a dominating metric, the number of events can be used to approximate bandwidth cost. We consider the occurrence of events that modifies the entry value of SBF from 0 to 1 (or 1 to 0). The analysis is based on a uniform hashing distribution over the key space at random.

As observed on the SBF in [7], assume that *k* hash functions are applied to each key. After inserting *n* keys into a table of size *m*, the probability that a particular bit still remains 0 is:

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

For the case of CBF, the value of a particular entry has a range of 0 to 2^b–1 (*b* bits for each entry). The probability of an entry value from 0 to 1, and 1 to 0, is represented as P_{01} and P_{10}, respectively. For a particular entry in table of size *m*, after *n* key insertions, P_{01} = Pr (value in the entry is '0' $\cap$ hashed by next insertion) = Pr (hashed by next insertion) $\times$ Pr (value in the entry is '0')

$$\left(1-\left(1-\frac{1}{m}\right)^{k}\right)\left(1-\frac{1}{m}\right)^{kn} \quad (1)$$

While P_{10} = Pr (hashed by next deletion ∩ value in the entry is '1') = P (hashed by next deletion | value in the entry is '1') × P (value in the entry is '1')

$$\frac{1}{n}\times C_1^{kn}\frac{1}{m}\left(1-\frac{1}{m}\right)^{kn-1} \quad (2)$$

This equation for P_{10} equals to:

$$\frac{k}{m-1}\left(1-\frac{1}{m}\right)^{kn} \quad (3)$$

By above two Eqs. (2) and (3), the amount of incremental messages under different Table Loads, *L*s, and connection rates could be estimated.

B. Selecting Hash Functions

The computation throughput of hash functions are studied by many works (e.g., [39], [41], [42], [43],[44]) and FD shares similar criteria that hashing should provide an even distribution and a straightforward computation for high speed links. In FD, we study the performance of MD5, MD- [39], and *fast hashing* on x86 platforms.

Although MD5 produces collision-resistant and fix-sized hash codes, the computations are difficult to optimize [41]. On the other hand, MD-, a modification of MD5 by ignoring the G, H, and I functions, has a close distribution quality with MD5 and better throughput. Furthermore, performance can be improved if prime number calculations are avoided [45]. We then tried simple but "fast" hash functions to compute 32-bit hash values for a flowID by different combinations of shifting, XOR, and an AND operation with a prime number. This hashing is characterized by a much lower computation overhead and can be implemented by a limited and simple instruction set (e.g., RISC-based network processor units). Our preliminary experiments show that the throughput of fast hashing and MD- achieve about 5.87 times and 2.45 times higher than that of MD5.

However, the uniformity of hash distribution is an important consideration for FD to estimate the size of incremental updates by Eqs. (2) and (3). Given a CBF, the hash load factor α is defined as $n_a \times k/m$. That is, the expected collision number falling into each bin. Fig. 5 shows that fast hashing is much less effective than MD5 and MD- in hash uniformity. On the other hand, a uniform distribution also avoids unexpected false hits in the recovery phase. Therefore, we use MD- to strike a proper balance between hashing uniformity and computation speed and thus *k* is equal to 4. Finally, to reduce the hash calculations, the 128-bit digest computed by the insertion is stored into the state entry and is re-used when deleting the flow.

2.1.4 Preliminary Evaluations of FD Scheme

This section presents the results of simulation (written in C++) to evaluate network costs of FD and SRPs from low to high connection rates. We consider a stateful cluster composed of a primary and a slave and this HA cluster processes the pass-through traffic from internal and external networks. The node-to-node link is 100Mbps which is chosen to model Fast Ethernet and all updates are delivered by unicast. In this topology, the range of steady connection setup and teardown rates and active connection duration are 500 to 90,000 cps and 5 to 30 seconds, respectively. The setup delay (the time elapsed between the first SYN to the first ACK) is set as 2 seconds [37]. We study 256K, 512K, and 1M active connections and total simulation time is 3,600 seconds.

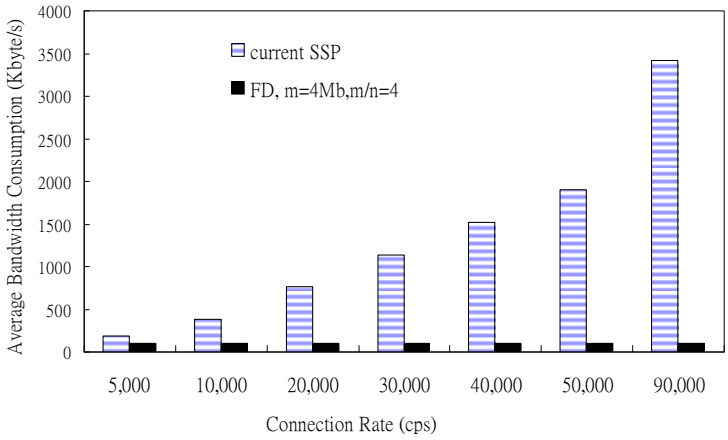


Fig. 6. The average bandwidth consumptions under different connection rates of SRP and FD (*m*=4Mb, *m*/*n*=4). The update interval is 5 seconds. Note that the update of FD is by sending entire SBF.

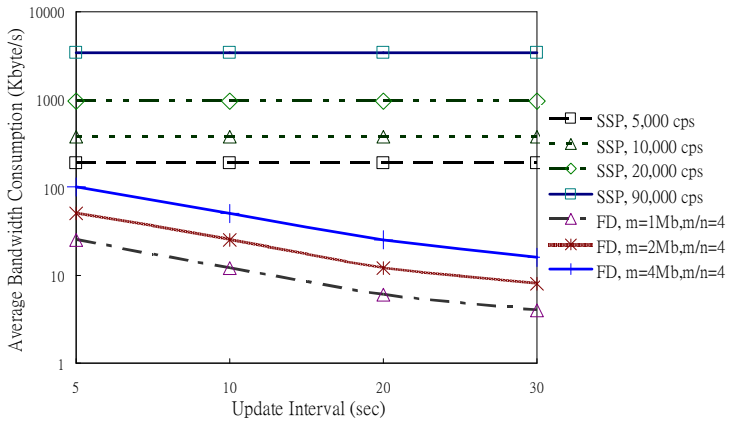


Fig. 7. The average bandwidth consumptions in log scale under different connection rates of SRP and FD by different update intervals. Note that the update of FD is done by sending entire SBF and the

We experienced with three FD configurations: 1Mb, 2Mb, and 4Mb SBF sizes for supporting 256K, 512K, and 1M connections respectively. A CBF counter takes up 4 bits. FD has 4 hash functions by calculating 128-bit MD5 outputs from an open source library. Based on the connection rate and active connection number, one can convert the thresholds to time intervals, hence, for FD, we only use a periodic update method and the update interval is from 1 to 30 seconds. Two update message formats of FD are both compared with current SRPs in the simulation.

Following the ct_sync and pfsync, the simulator for current SRPs sends messages for state insertions, changes, and deletions, and message size is 13 bytes which contains one flowID and a flag to specify update type. We do not consider the overflow of a delayed output queue and all messages of current SRPs are sent to the slave in real-time.

We first show the estimated update cost caused for different connection rates in Figs. 6 and 7 which indicate the bandwidth consumptions of current SRPs and FD by sending entire SBF. Figure 6 shows that the FD structure to support 1,000,000 connections with 5-second update interval eliminates 46%, 73%, 86%, and 97% of bandwidth consumption compared to current SRPs at 5,000, 10,000, 20,000 and 90,000 cps, respectively. In Fig. 7, we observe that FD by sending entire SBF provides a snapshot of the current state table at a moment and the message size is deterministic for every update regardless of different connection rates. By contrast, the total number of update messages under a connection rate does not change, thus current SRPs have fixed costs on network depending on the connection rate of regular traffic. Therefore, FD requires a less bandwidth as the update frequency decreases. For example, if the update interval is set to 30 seconds, FD reduces the bandwidth consumption by 91% to 99% for 1M connections. Thus, the network overheads of the proposed scheme can be reduced significantly with a larger update interval. In addition, the network overhead can be improved due to a smaller maximum connection number n with the same m/n. For example, in Fig. 6, it shows that FD reduces at least 86% of the bandwidth consumption with 5-second update interval both for supporting 256K and 512K connections. Second, we also simulate FD with difference update with low connection rates (not shown in the figure). When the primary works in a slow traffic environment or with a small update interval, e.g.,

2 seconds, a difference mechanism is used because the update size is less than *m* bits. Compared to current SRPs, the simulation results demonstrate that FD with difference messages eliminates 20% of the network bandwidth and reduces the number of update messages by 34% to 37% from 500 cps to 5,000 cps. The above results explore that the major benefit of the proposed scheme is to improve the bandwidth utilization, especially at high connection rates.

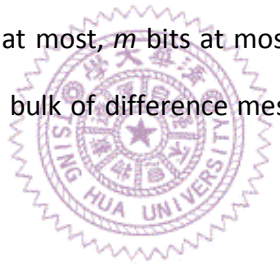
We study the throughputs of MD5, MD- and fast hash functions by getting 324M sets of hash outputs (*k* = 4) without other operations. Our results show that the throughput of fast filter and MD- filter achieve about 5.87 times and 2.45 times higher throughput than MD5 filter. Our simulation tests also demonstrate that the hash function dominates the performance of the proposed scheme.

2.1.5 Discussions

There is essentially no limit to how many nodes can participate in a cluster and the network and memory overheads introduced by maintaining state consistency determine the scalability of a replication protocol. The bandwidth needed by current SRPs scale linearly with the amount of regular traffic passing through the HA cluster and the number of update messages may grow up to many thousands per second (e.g., at 20,000 cps). By contrast, the number of bytes needed by an update is at most a constant value (*m* bits) and the number of update messages can be determined by update interval and is smaller than current SRPs. As our simulation results show, the proposed method requires relatively much less network bandwidth by sending entire SBF and less update messages by sending difference messages and therefore is more scalable for state replication than existing methods both for low and high traffic loads.

Notice the parameters for FD: the number of hash functions *k* and type (e.g., MD5 or

fast hash), a bit vector of length *m*, *b* additional bits for each CBF cell, update approach, and *n* simultaneous maximum connections. These parameters are negotiated at HA initialization time or when a new node is trying to connect to a functioning cluster. Recall that the feature of Bloom filters is that they provide a tradeoff between the storage requirement and accuracy. Thus, if one wants to run with less bandwidth consumptions for state consistency, this can be achieved by reducing the size *m* with possibly slightly increasing of entries into the state table because of more false positives in the recovery phase. The other approach to reduce network overhead is to choose an appropriate larger update interval according to the traffic conditions. Furthermore, by the above settings, FD occupies a total space of (*m* + *mb* + *m*) bits at most in memory of a primary, i.e., an FD data structure and difference messages of size *m* at most, *m* bits at most to the network for individual update by sending an SBF or a bulk of difference messages, and *m* bits in a slave to store a backup SBF.



2.1.6 Section Summary

For a stateful HA scenario, the current solutions use update messages for state replication which may use a substantial amount of network bandwidth and this extra overhead could also reduce the capacity of a cluster to process the regular pass-through traffic. Moreover, the computation requirements associated to maintain the state consistency between the cluster nodes result mainly from the processing of update messages and merging the state changes into the local state table. Flow Digest (FD) has been proposed to improve existing state replication protocols by reducing the update overhead for both low and high connection rates.

The main advantage of the proposed method is to reduce the network overheads of state replication. The simulation results show that the bandwidth consumption of the

flow digest is much less than that of current implementations. The proposed scheme by difference messages eliminates 34% to 37% of the network bandwidth and reduces the number of update messages to the slave by 20%. More importantly, at high connection rates, the bandwidth consumption can be reduced typically by at least 86% compared to current solutions.

2.2 Evaluations of TCP Membership Replication in Firewalls by Real Packet Trace Files

A stateless HA cluster can be simply achieved by using the stateless mechanisms for network redundancy (e.g., VRRP [46]) and identical configuration/ruleset. However, without state replication, all legitimate connections (a.k.a. flows), or even worse whole user requests, have to be re-established after a failover due to the loss of flow states in the firewall cluster. On the other hand, in a stateful HA scenario for firewalling, a state replication protocol (SRP) (e.g., pfsync [47] for OpenBSD [48] IP Filter [49] and ct_sync [50],[51],[52],[53] for Linux netfilter [54]) provides replication management and supports reliable connections at cluster-level by switching the active connections to the secondary firewall node transparently in a failover. Namely, an SRP is complementary to stateless failure detection by maintaining state consistency between a firewall node and its backup. Note that both the pfsync and ct_sync protocols adopt the passive replication and rely on explicit messages to replicate three in-order state types (i.e., insertion, modification, and deletion) via multicasting.

Many other solutions have been proposed to provide connection-level reliability such as fault tolerance in TCP, OS mechanism, and new transport protocols. However, although these schemes can be deployed, achieving reliable connectivity remains a challenge. For a user, there is no difference between the service outages due to the networks and due to the servers. Any near-source or near-destination single-point failure still hinders the service quality. When network failures are considered, service availability is often as low as 99%, meaning that a server is out of service for about 15 min a day on average [55]. Furthermore, Boutremans, et al. [56] find that an

availability degradation of VoIP service results from the reliability problem of routing equipment. They identify the need for more reliable hardware architecture and fast protection mechanisms against link failures.

Despite a stateful HA cluster complements flow-level protections by removing the single-point failure, an SRP also consume network and system resources to protect a connection. In this paper, we evaluate the costs of different state replication methods for TCP connections (the majority of Internet traffic) and explore the tradeoffs from varying a time-triggering parameter to replicate a connection. The packet traces (see Table I) collected from major Internet backbones and from campus are used to simulate the replication operations on a prototype implementation.

2.2.1 State Replication Methods Used in Tests

A. Eager and selective replication

In a firewall device, a state entry is used to store the information derived by TCP stateful tracking from the bi-directional traffic of a TCP flow. As a flow is initialized and terminated, the corresponding entry is inserted to and removed from the flow table. Each entry contains two types of flow sub-states: immutable and mutable. An immutable sub-state *flowID*, i.e., four-tuple <DstIP, SrcIP, DstPort, SrcPort>, remains constant and is used to identify a connection. Mutable sub-states may be changed very frequently, such as the latest packet arrival time, sequence and acknowledgement numbers, window advertisement and total flow bytes.

Two replication methods are first considered: *eager replication* and *selective replication*. Besides the immutable information, eager replication synchronizes every change on the mutable information of a flow from SYN to its completion. For example, many advanced firewalls keep track of the sequence and acknowledgement numbers and TCP flags continuously to ensure the active flows

are compliant with the TCP specification in all aspects. To meet these criteria after a failover, eager replication must be adopted by a stateful firewall cluster for synchronizing the mutable information.

Though the message sizes of the pfsync and ct_sync both exceed 100 bytes, an explicit 32-byte-long representation is used to update the following data for investigating the costs of eager replication.

- flowID
- Sequence and acknowledgement numbers
- Segment size, window scale, and TCP flag
- Timestamp
- Operation and direction flag

Another method, selective replication, synchronizes three state changes (i.e., SYN_SENT, ESTABLISHED, and flow completion) only. Actually, both the pfsync and ct_sync use a strategy similar to selective replication to optimize state copying operations. In [6], a selective mechanism is used to save the processing time of the backup server. Furthermore, note that in our evaluation, a 16-byte-long message (only flowID and operation flags) is used by selective replication to evaluate the overheads.

B. Flow Digest

The scheme *Flow Digest* (FD) [9],[10] improves the procedures of state replication through two factors: 1) an architectural improvement prevents the flow table from the access by replication traffic before a failover, and 2) a compact data structure employing randomization (i.e., Bloom filters) is designed to reflect the active flows.

For PN and AN, all established flows are collected into a terse set representation and synchronized to the backup node by sending a Bloom filter or incremental messages.

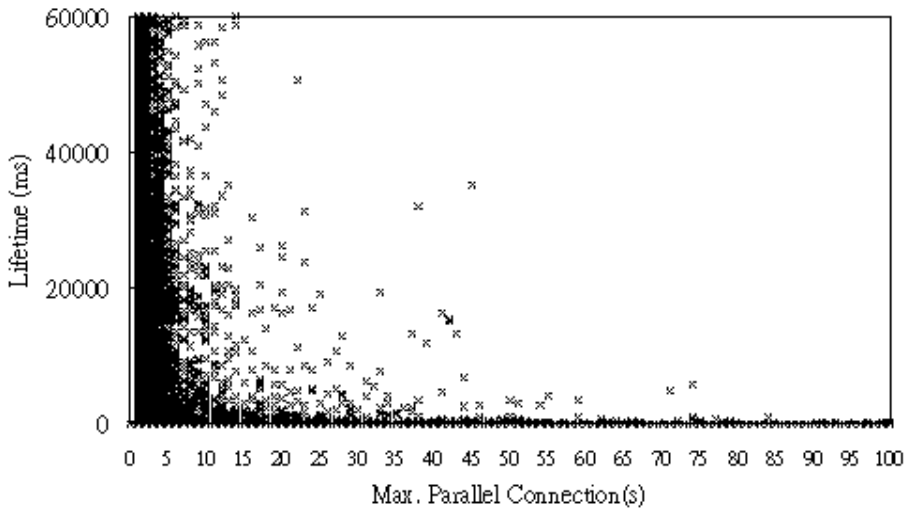


Fig. 8. The maximum number of TCP flows opened at the same time between two endpoints vs the host-level lifetime distributions (with the IPLS-3 trace)

The scheme shifts to the recovery phase after a failover and the flow table is reconstructed in a packet-driven fashion by querying the backup Bloom filter.

The memory requirement of FD can be kept small, while still achieving high accuracy. For example, for supporting 1,000,000 connections in maximum, using 10,000,000 elements, four bits per element, and four hash functions yields a false positive rate of 1.2% and requires a 7,500-KB memory space; not a concern in today's equipments. Though false positives are possible, FD never rejects active flows after a failover under accurate state replication.

We use the incremental updates to evaluate the overheads of the FD scheme. By large-size Bloom filters, FD can be viewed as a 2-state replication method to synchronize established flows and their terminations, where the message size is 32-bit.

C. Host-to-host aggregation

By ignoring the port number at the two endpoints, small replication operations at the host-level can protect the packets from different TCP flows between the same host pair. Two observations from Fig. 8 with IPLS-3 provide an initial indication of the

potential benefits of host-level aggregation. The distributions of IPLS-1 and AUCK-4 are not shown because they are similar with the results of IPLS-3.

First, Fig. 8 illustrates that parallel connections are observed for all lifetime distributions, especially parallel degrees less than 5 connections. One presumable reason is that parallel TCP connections are widely used by kinds of applications, like web transfers, multi-stream applications, and P2P. For example, web browsers open parallel connections at the same time to request various objects of a web page. In [57], the analysis of web traffic shows that nearly all web clients open 4 or fewer simultaneous TCP connections to transfer the inline contents. In Firefox 2.0 browser, the default setting for maximum parallel connections per server is increased to 8. The study also points out that as clients transfer more objects, the likelihood of using concurrent connections increases. On the other hand, the studies [58], [59] on the web workloads show that both the number of objects per web page and the number of distinct server delivering content per page are increasing over the years.

Second, as the host-level lifetime decreases, especially less than 5 sec, we observe a clear increase of the number of maximum parallel connections. Our flow analysis shows that many endpoints establish high-degree parallel connections at almost the same time. This implies the overheads of port-level replication for short flows are much higher than that of long flows due to the short burst and parallelism. By aggregating replication operations per source-destination pair, an HA cluster can counteract these potential overheads.

To evaluate the overheads of host-level aggregation, only the first establishment event and the last deletion event between the two endpoints are replicated to the backup node. The message format/size is identical to that of selective replication.

2.2.2 Flow lifetime vs State Replication Overheads

Thus far, in order to guarantee consistency, state changes in the primary firewall node are forced to be synchronized precisely. However, this approach is expensive. Measurements of the Internet traffic have shown that most TCP flows are short-running [60] and that long flows (e.g., less than 20%) carry a high proportion (e.g., 85%) of the total traffic bytes [60], [61]. Furthermore, long HTTP sessions of purchases are more profitable for web sites [62] and should be protected for successful completion. For web traffic [58], 15% of the TCP connections are persistent. However, these persistent connections deliver approximately 40% of the transferred bytes for web objects.

The lifetime and size distributions of the Internet traffic add another dimension to state replication. The above studies imply that the major costs come from short flows when doing replication, but focusing on long flows protects the majority of network traffic (in bytes) and profits. For instance, the efficiency of replicating a flow less than 50 ms is extremely low, particularly when this service is not critical for users. Because of the increased memory accesses and network operations, the cluster performance is likely to be impacted negatively due to the heavy loads from an SRP. Clearly, there is a tradeoff between good pass-through performance in failure-free duration and minimal recovery overheads after a failover.

A lazy threshold $t^{threshold}$ is used as a time-triggering parameter which refers to how long a flow has persisted before a replication operation is performed for that flow. We explore the effects of varying this parameter on the costs of four replication methods mentioned above. A $t^{threshold} = 0$ represents the precise replication which indicates the flow replication is dependent on the state replication method only.

2.2.3 Evaluations

A. Implementation and packet trace data

The experiments were performed on a testbed consisting of two identical 3-port machines (Intel Pentium-4 2.0GHz and 512MB RAM) as the cluster nodes. Two nodes are connected with a 100Mbps LAN (the replication link). As the base-line implementation, both machines run Linux 2.4.20 with our kernel module and patch installed. A tasklet implements the stateful tracking subsystem, four replication schemes, and lazy threshold in the kernel space. The flow table is implemented by a hash table. The inactivity timeouts for the idle entries in SYN_SENT, ESTABLISHED, and FIN_WAIT states are 20, 60, and 20 sec, respectively. Two kernel threads are used to send and receive the packets via UDP multicasting on the replication link.

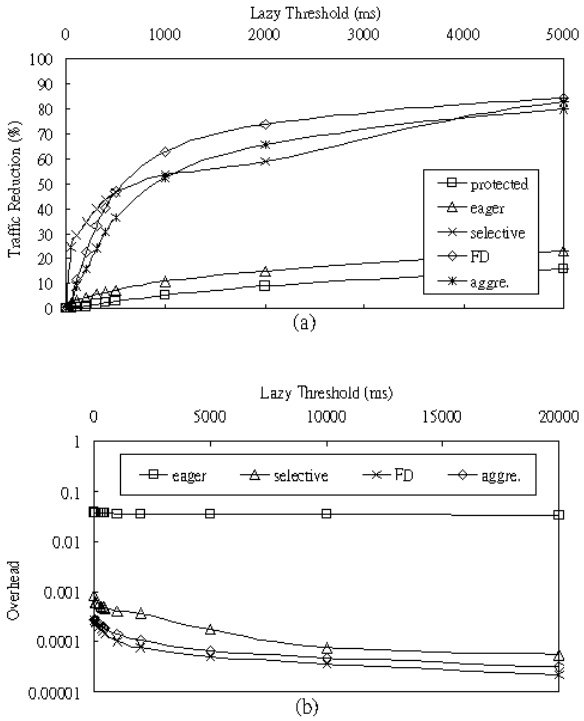


Fig. 9. (a) The traffic reduction and (b) the replication overheads (with the IPLS-1 trace).

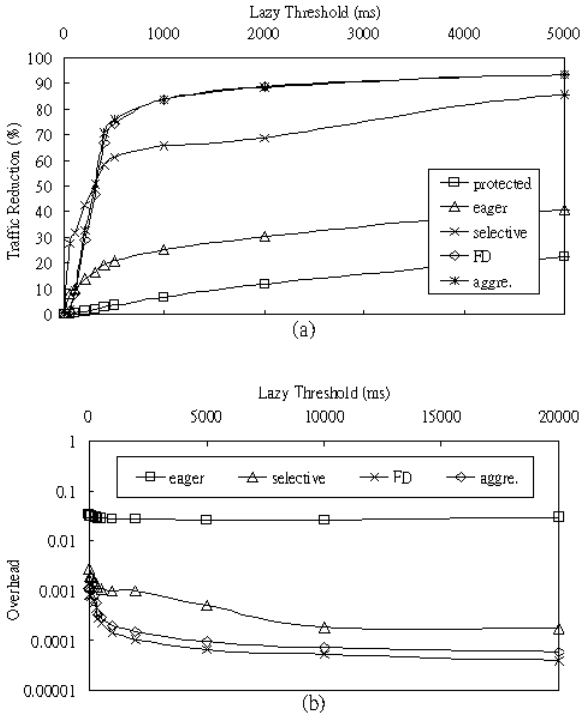


Fig. 10. (a) The traffic reduction and (b) the replication overheads (with the IPLS-3 trace).

We validate four replication methods by the trace-based simulation, which gives us the imitation of the activities of known backbone/campus networks and a practical picture of benefits and drawbacks of the given methods. The replication schemes are applied to the bi-directional 10-min traces of Abilene-I and Abilene-III (denoted as IPLS-1 and IPLS-3) which were collected from OC-48c and OC-192 links in 2002 and 2004 [63]. Another bi-directional 24-hour packet trace files (denoted as AUCK-4) from NLANR [63] were captured at the University of Auckland in 2001. All inbound and outbound packets with the corresponding metadata are read from the trace files and then sent to the kernel space sequentially as the input packets. At the end of reading trace data, all active flows are forced to complete and then passed to flow analysis.

To enable a fair comparison, we ignore purposely the replication for the flows whose SYN packets were not captured in the trace files, though this leads to an underestimation of the pass-through traffic (especially long flows) and replication costs. Furthermore, due to the fact that routes may be asymmetric at the backbone, there is a minor tuning in stateful tracking.

For setting FD parameters, the maximum number of the allocated state entries in three traces is 159,394. Therefore, we set the maximum concurrent connections supported by a cluster node as 200K and set the Bloom filter size as 2,000K elements. The MD- is used as the hashing functions and the hash number is 4.

B. Trace-based evaluation results

To understand the effects of the imprecise replication, the reductions of the protected pass-through and replication traffic are studied by tuning the parameter $t^{threshold}$ from 0 to 20,000 ms. Note that the active flows whose states are already replicated are referred to as the *protected* flows. On the other hand, the overhead of

a replication method is measured by its bandwidth costs and the protected traffic bytes (TCP payloads). Let $N_{replication}$ and $N_{protection}$ be the total bytes of transmitted replication messages and pass-through packets whose states already have been replicated, respectively. Then, the *overhead* is defined as ($N_{replication}$)/($N_{protection}$). Figs. 9 to 11 illustrate the evaluation results in AB scheme of an HAC.

First, it is observed that eager and selective methods are vulnerable to one-way flows and malicious SYN packets. For example, in IPLS-1 and IPLS-3, 9.9% and 39.2% of the recycled state entries get stuck in SYN_SENT state. In the case of IPLS-3, the flow analysis shows that 87.1% of the recycled SYN_SENT entries are allocated by one-way flows (almost sending only 1 to 3 packets), and the remaining 11% are the two-packet flows (SYN and RST). In a cluster using eager/selective replication, a short one-way flow allocates two state entries (in PN/AN and its backup node), which are recycled and deleted immediately after an inactivity timeout (20 sec in our simulations). Thus, these one-way flows significantly aggravate the contention on the system resources of two cluster nodes and bandwidth consumptions on the replication link. By contrast, because FD and host-level aggregation only replicate the established flows (namely, from ESTABLISHED state), the number of the deletion events activated by the SYN_SENT timeouts are much less than those of eager and selective methods. The measurement of one-way and two-way flows has been the subject of research in [64].

In Figs. 9a to 11a, when $t^{threshold}$ =50 ms, due to the savings of replicating one-way flows and malicious SYN packets, the reduction ratios of selective replication on the bandwidth costs are as high as 24.8%, 27.8%, and 16.7% in IPLS-1, IPLS-3, and AUCK-4. By contrast, the cost reduction ratios of FD and host-level aggregation are only 0.05%–6.3% by the same $t^{threshold}$. On the other hand, because most TCP flows of the Internet traffic are short-lasting, they dominate the state replication costs.

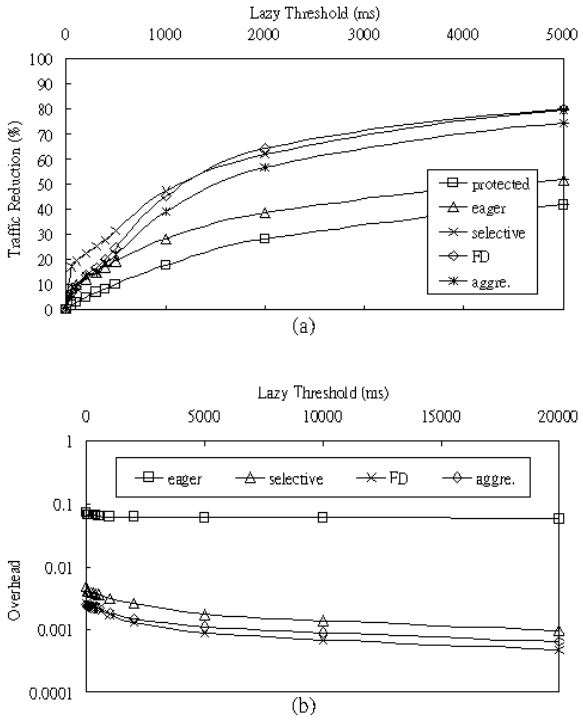


Fig. 11. (a) The traffic reduction and (b) the replication overheads (with the AUCK-4 trace).

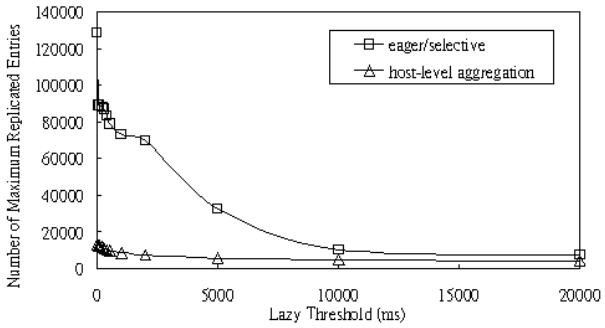


Fig. 12. The maximum replicated entry number in the backup node (with the IPLS-3 trace) in the simulation of AB scheme.

Except for eager replication, a very clear rise on the replication traffic reductions is observed when $t^{threshold}$ < 1 sec in three packet traces.

About the reductions on the protected pass-through bytes, Figs. 9a to 11a show that only 0.09–1.6% of total pass-through bytes are not protected by $t^{threshold}$=50 ms, but up to 27.8% of the replication traffic are saved for selective replication. For FD in IPLS-3, reducing 50% ($t^{threshold}$=320 ms), 74.4% ($t^{threshold}$=500 ms), and 88.9% ($t^{threshold}$=2,000 ms) of the replication traffic excludes only 1.9%, 3.4%, and 11.8% of

the pass-through bytes. The host-level aggregation has a very similar behavior. Remember that the resource savings come from the reduced replication operations. The reductions on the replication costs and protected bytes mirror the cumulative flow lifetime and size distributions in packet traces. Obviously, the efficiency of replicating flows longer than 500 ms is much higher than the precise replication. A small $t^{threshold}$ can be useful for alleviating peak system load, reducing bandwidth consumption, and protecting the majority of Internet traffic bytes.

In Figs. 9b–11b, we show the overheads of four replication methods. The overheads of eager replication do not decrease significantly as $t^{threshold}$ increases. This confirms the high costs of keeping all mutable information consistent between the cluster nodes. In Fig. 9b, when $t^{threshold}$=0 ms in IPLS-1, the overheads of selective method, FD, and host-level aggregation are only 7.9%, 2.4%, and 3.3% of the overhead of eager replication, respectively.

Figs. 9b–11b show that the overheads of the FD scheme are much less than those of eager and selective replication. For example, in IPLS-3 at $t^{threshold}$=500 ms, FD reduces 99.5%, 79.8%, and 22.9% overheads when compared to eager, selective, and host-level aggregation methods. Furthermore, though host-level aggregation avoids the operations for parallel connections, except for $t^{threshold}$ < 1 sec in AUCK-4, the overheads of host-level aggregation are slightly higher than FD. This is because the message size of the FD incremental update is 32 bits and the size of host-level aggregation is 16 bytes.

Another important metric is the number of replicated entries in the backup node. Though this metric may be not critical to an active/backup cluster, the valuable state entries of an active/active cluster are allocated both by pass-through flows and replicated flows. Thus, we perform simulations in AB mode to investigate the effects of $t^{threshold}$ on the maximum number of replicated entries in the backup node. Note

that, in FD no replicated entry is required in the flow table of the backup node.

For eager and selective methods, Fig. 12 illustrates that varying $t^{threshold}$ from 0 to 50 ms reduces the maximum replicated entry number from 128,852 to 89,425 (a 30% decrease) due to the effects of one-way flows. A $t^{threshold}$=2,000 ms reduces 45% of the maximum number of eager/selective replication (from 128,852 to 69,897), while host-level aggregation reduces it by 44.4 % (from 12,873 to 7,152) at the same $t^{threshold}$. Fig. 12 also illustrates that parallel connections exist for all lifetimes and parallelism degree increases as the lifetime decreases. When $t^{threshold}$=50 ms, the entry number of host-level aggregation is 12,818; only 14.3% of the requirements of eager/selective replication.

2.2.4 Section Summary

To improve service availability and reliability, the stateful HA firewall clusters are deployed to remove network single-point failures. In this paper, we perform the simulation tests by real backbone/campus packet traces to evaluate the costs of four state replication methods as the possible solutions for firewall clusters with a tunable time-triggering parameter. To the best of our knowledge, there have been no cost evaluation results of the flow-level state replication methods over HA clusters available. We believe that our results also give a practical view to other technologies using TCP state replication, like transparent TCP-connection migration.

We find that the precise replication overheads for short flows are high, because most TCP flows are short-running and the short flows are likely to have high-degree parallel connections. Thus, a small time delay can yield significant reductions on the bandwidth costs and cluster resources. Typically, reducing 50% and 74.4% of bandwidth costs only excludes 1.9% and 3.4% of the protection on the pass-through traffic. Moreover, the overheads of the FD scheme are lowest in nearly all the tests

we ran. For the active/active clusters, an important metric, the maximum replicated entry number in the backup node, is investigated. The results show that both the host-level aggregation and a time trigger larger than 5 sec reduce effectively the number of replicated state entries. In summary, our investigation highlights the benefits of the imprecise state replication, including the scheme employing randomization, the host-level aggregation and the time-delay policy.

Besides above results, we further suggest that an SRP should replicate a TCP flow from its ESTABLISHED state. This strategy avoids the high costs, such as a high entry-recycling rate and unnecessary bandwidth consumptions, from very short one-way flows and malicious SYN packets.

3. Multi-Level Counting Bloom Filter (MLCBF)

Fig. 20 shows a basic structure of my SPE consisting of MLCBFs. With the same functionalities of *Counting Bloom filter* (CBF), *Multi-Level Counting Bloom Filter* (MLCBF) allows item deletions and provides *membership* and *multiplicity* queries on a set S over a universe U with small error probabilities. MLCBF handles the insertions and deletions of items with keys easily to change S dynamically and features construction simplicity. A query of $x \in S$ on the filter would always get a positive answer and a membership query of $y \notin S$ could give a false positive.

As a multi-level structure, the intuition behind MLCBF is to store the most of inserted items in the largest (i.e., first) level. Then, all operations of MLCBF can be finished probably in the 1st level. To hold n items of S in maximum, MLCBF is a hierarchy of D levels ($LV_1, \ldots, LV_D, D \geq 2$) with a set of D independent and uniform hash functions ($h_1, \ldots, h_D$), wherein each level comprises different bucket numbers ($BN_1, \ldots, BN_D$). h_i at least provides a ($log_2 BN_i$)–bit long output. The level sizes are decreasing linearly by a fixed *decreasing ratio R* ($R < 1$). Let R_0 as $1 + R^1 + \cdots + R^{D-1}$. LV_1 holds $BN_1 = \lfloor n/(H \cdot R_0) \rfloor$ bucket elements (BEs), and LV_i holds $BN_i = \lfloor BN_{i-1} \cdot R \rfloor$ buckets for $i \geq 2$.

Each BE consists of *H* cells ($H \geq 1$) and a *load bitmap* (LB) of I bits to record the number and location of *active cells*, which record the information of inserted items. Each cell holds a cell counter (CC, C-bit) and a fingerprint (F-bit) from a hash function h_f. If the corresponding LB bit is not set, a cell is identified as *empty* or *non-active*, and the cell access for query and deletion can be avoided. Let the total bucket number as $BN = n/H$. This gives a total memory space bounded by $n \cdot (F + C) + BN \cdot I$ bits.

For MLCBF, I propose two insertion algorithms called *MLCBF-First Available*

Algorithm 1 Pseudo-code for FA insertion and search in MLCBF

```
Function FA-INSERT(key)
1:    if SEARCH(key) = 0 then
2:        for i ← 1 to D do
3:            pos ← LV_i[h_i(key) mod BN_i]
4:            for j ← 1 to H do
5:        if BE_pos.LB[j] = 1 then
6:                BE_pos[j].fingerprint ← h_f(key)
7:                BE_pos[j].CC ← BE_pos[j].CC + 1
8:            BE_pos.LB[j] ← 1
9:                return 1
Function SEARCH(key)
10:   for i ← 1 to D do
11:       pos ← LV_i[h_i(key) mod BN_i]
12:       if BE_pos.LB ≠ 0 then
13:           for j ← 1 to H do
14:               if BE_pos.LB[j] = 1 then
15:                   if BE_pos[j].fingerprint ≠ h_f(key) then
16:                       j ← j + 1
17:                   else return 1
18:               else j ← j + 1
19:    return 0
```

(*MLCBF-FA* or *FA*) and *MLCBF-Least Load* (*MLCBF-LL* or *LL*). For two algorithms, to insert, query, or delete item$x$, there is D possible buckets in MLCBF.

For MLCBF-FA, as shown in Algorithm 1, if the item $h_f(x)$ does not exist in MLCBF, FA simply places the item to an empty cell of BE_i indexed by $h_i(x) \bmod BN_i$ with the smallest i. Namely, FA tries to insert a new item into lower levels. An item will only be inserted into LV_{i+1} when the hashed bucket in LV_i is full. The probing is stopped until an empty cell or an overflow in LV_D. When the empty cell is found, its CC is simply incremented and the corresponding bit in LB is set to one.

For MLCBF-LL, if $h_f(x)$ of a new item x is not in MLCBF, x is placed to an empty cell of the BE whose LB stores the smallest number of bits set to one amongst D associated BEs of x. In case of a tie, I always place x to LV_1 like *d-left* scheme [24]. Thus, FA allows bucket overflows in LV_i $(1 \le i < D)$. In LL, any bucket overflow is an error condition. Notice, I check whether $h_f(x)$ already exists in any BE at first for an insertion to avoid the problem that the same $h_f(x)$ to be found in several cells when trying to delete item x. If so, I simply increase the corresponding CC in insertion.

To answer a query of "$y \in S$?", one checks whether $h_f(y)$ is found in D associated BEs by $SEARCH(key)$. If not, $y \notin S$. Thus, total $D \cdot H$ probes are required in worst case and lookup complexity is $O(1)$. In $SEARCH(key)$, because a search naturally accesses the buckets in the same order as insertion and the wanted item could be located in LV_1 probably, it starts from LV_1 and continues till LV_D, if necessary. The lookups on levels are actually independent and can be optimized by parallel executions.

In a deletion, when the inserted item is found through $SEARCH(key)$, the CC is just decremented and the LB is clear if the CC becomes 0.

3.1 Properties of MLCBF-FA

By employing d-left hashing [24], *d-left CBF* (DLCBF) [22],[23] uses D equal-sized subtables and H cells in each bucket. Thus, both MLCBF and DLCBF can be extended in two ways for a number of given cells: one extends the number of hash functions, D, and the other changes H. We unify these by considering FA, LL, and DLCBF using a two-parameter pair (D, H) to compare performance metrics and the tradeoffs. In the rest of paper, MLCBF-FA(D,H) indicates the setting (D levels, H cells per bucket) of an MLCBF by FA insertion scheme. For the sake of generality, we call FA, LL, and DLCBF as *multi-level fingerprint-based filters* (MFFs) to highlight their basic differences on the construction concept from the *Bloom filter-based filters* (e.g., legacy CBF).

The *storage utilization* or *load* of an MFF is defined as the ratio between the number of items and the total cell number. The *load distribution* of LV_i (called as LD_i) is defined as the ratio between the number of items in LV_i and the total cell number. The *load factor* α of an MFF measures the expected number of items per bucket (i.e., active cells per bucket), and α_i is the load factor of LV_i. Finally, if not specified explicitly, we set R as 0.5 and F as 20 bits in our experiments. As described later, using (4,8) and F=20-bit yield a P_{FP} less than $3.051 \cdot 10^{-5}$. We believe this is low

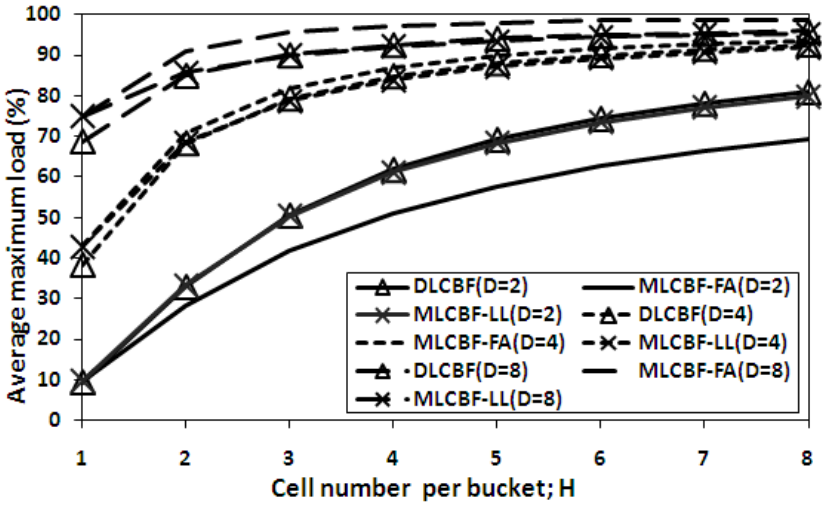


Fig. 13. Average maximum achievable loads of 10k-trail simulation with different (D, H) settings. Total cell number is 10k.

enough for many practical applications.

A. Maximum Achievable Loads

We have experimentally tried a variety of choices $(2 \le D \le 8, 2 \le H \le 8)$ for their maximum achievable load, $Load^{max}$, and Fig. 13 shows the results (the simulation setup is described in Sec. 5.1).

First, except for D=2, FA has higher $Loads^{max}$ than LL and DLCBF by the same (D, H), especially when D=8. The $Loads^{max}$ of LL and DLCBF are quite similar to each other. Second, Fig. 13 indicates an MFF needs $n/Load^{max}$ cells at least to store n items in maximum. For example, to support 10^5 items, FA, LL, and DLCBF by $(4,4)$ need at least 115,207, 119,189, and 117,994 cells, respectively. Thus, we set up the total cell number of an MFF that is slightly larger than $n/Load^{max}$ to prevent bucket overflows in test (e.g., for 115,207, we round it up to 115,300). Finally, FA$(4,8)$ is almost as space-efficient ($Load^{max}$=93.65%) as FA$(8,4)$ ($Load^{max}$=97.26%) but with four fewer hash functions. This may imply a smaller latency in a platform without hashing acceleration hardware.

B. Storage Utilization and Load Distribution

We now present the analysis of MLCBF-FA. For MLCBF-LL and DLCBF, the experimental simulations are used to investigate their properties. For simplicity, we assume the probability of the fingerprint collisions is ignored and item deletion is not considered in the subsequent analysis. The first task is to compute the expected storage utilization, LD_i, and α_i of LV_i of FA.

If n items are inserted into a hash table with separate chaining by a uniform hash function, as the number of elements n goes to infinity and the average load is μ, the fraction with load k is $\frac{1}{k!}(e^{-u}\mu^k)$. Most of the analysis on hashing is based on the above probability distribution. In MLCBF, for a level with m BEs, then the corresponding expected number of items lying in all BEs which have exactly the load of k mapped into them is:

$$km\binom{n}{k}\frac{(m-1)^{n-k}}{m^n}$$

To calculate load distribution of FA, let $n_{LV_i}^{overflow}$ as the expected number of items left from LV_i to be inserted to LV_{i+1}, $n_{LV_i}^{success}$ as the expected number of items inserted to LV_i successfully. Then:

$$n_{LV_i}^{overflow} = n_{LV_{i+1}}^{insert} = \sum_{j=H+1}^{n}(j-H)\cdot m\cdot\binom{n}{j}\frac{(m-1)^{n-j}}{m^n} \quad (3)$$

If we apply Eq. (3) recursively starting from LV_1 with $n_{LV_1}^{insert} = n$ and $n_{LV_i}^{success} = n_{LV_i}^{insert} - n_{LV_i}^{overflow}, i = 1, 2, ..., D$, we can estimate α_i as $n_{LV_i}^{success}/BN_i$, storage utilization of LV_i as $n_{LV_i}^{success}/(BN_i \cdot H)$, and LD_i as $n_{LV_i}^{success}/n$. For example, by Eq. (3), to insert 15k items into an FA(4,8) containing 20k cells, $n_{LV_i}^{success}$ of LV_1 to LV_D are 10,336, 4,237, 427, and 0; very close to the 10k-trial simulation result: 10,328, 4,237, 432, and 0 in average. Furthermore, by a total cell number n, the $n_{LV_D}^{overflow}/n$ of an FA(D,H) can be computed. Then, $Load_{FA(D,H)}^{max}$ can be estimated by increasing the load till $n_{LV_D}^{overflow} > 0$. For instance, the estimated $Loads^{max}$ of $(4,8)$ and $(8,4)$ are 94% and 97%; only 1.7% at most higher than the results of FA in Fig. 13.

TABLE I: RESULTS OF LOAD FACTORS AND LOAD DISTRIBUTION

(D,H)	Results	Method	Load = 20%				Load = 85%			
			Lv1	Lv2	Lv3	Lv4	Lv1	Lv2	Lv3	Lv4
(4,8)	Exp. AVLF	FA	2.992	0.008	0	0	7.88	7.472	4.528	0.16
		LL	1.632	1.616	1.544	1.392	6.808	6.816	6.8	6.792
		DLCBF	1.936	1.776	1.52	1.176	7.128	6.904	6.744	6.472
	Ana. load factor	FA	2.992	0.008	0	0	7.896	7.48	4.464	0.128
	Exp. AVLD	FA	0.998	0.001	0	0	0.617	0.293	0.088	0.002
		LL	0.543	0.226	0.128	0.058	0.534	0.267	0.132	0.066
		DLCBF	0.301	0.277	0.237	0.184	0.261	0.253	0.248	0.238
	Ana. load distribution	FA	0.998	0.001	0	0	0.618	0.293	0.087	0.001
(8,4)	Exp. AVLF	FA	1.564	0.06	0	0	3.856	3.736	3.304	1.776
		LL	0.804	0.796	0.796	0.796	3.48	3.476	3.332	3.128
		DLCBF	1.02	0.992	0.984	0.964	3.944	3.888	3.752	3.508
	Ana. load factor	FA	1.564	0.06	0	0	3.86	3.74	3.3	1.744
	Exp. AVLD	FA	0.98	0.019	0	0	0.57	0.276	0.122	0.032
		LL	0.505	0.249	0.124	0.062	0.514	0.256	0.123	0.057
		DLCBF	0.159	0.155	0.154	0.151	0.145	0.143	0.138	0.129
	Ana. load distribution	FA	0.981	0.019	0	0	0.569	0.276	0.122	0.032

Exp. AVLF: Experimental average load factor of 10^6 *trails,*
Exp. AVLD: Experimental average load distribution of 10^6 *trails.*

Table I lists the α_i and LD_iobtained from the analysis and simulation. It outlines the advantage of MLCBF; the majority of items are located in LV_1 (and LV_2). The skewness of LD_1 of FA is higher than LL due to insertion strategy. Interestingly, the values of α_i and $\sum \alpha_i$ of LL are similar to those of DLCBF, and $\sum \alpha_i$ of FA is smaller than those of LL and DLCBF; even their α values are identical.

C. Successful and Unsuccessful Search Costs

Now, we are ready to compute the successful and unsuccessful search costs of FA. A search in an MFF starts from LV_1, and its cost is measured in terms of the number of probing cells over levels sequentially. The analysis gives us a clear view of the

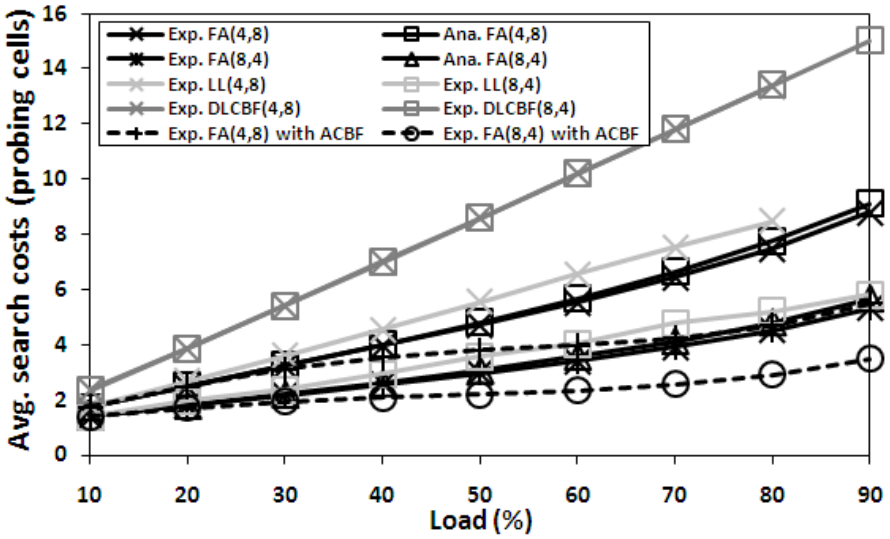


Fig. 14. Experimental and analytic successful search cost (probing cell number) of FA, LL, and DLCBF at different loads and (D,H). Total cell number is 10k. F=24 bits.

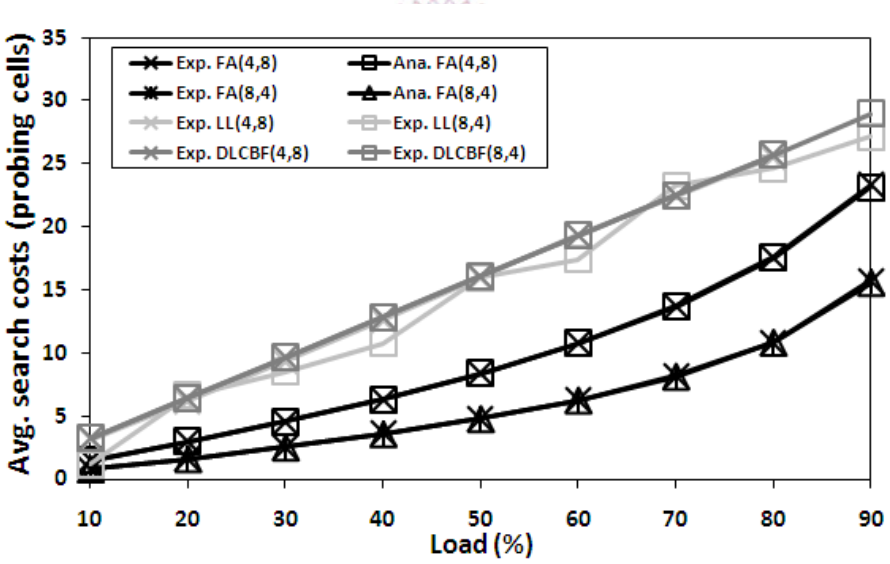


Fig. 15. Experimental and analytic unsuccessful search cost. Each experimental value is the average of 20-run measurement results. Each run contains 10^6 unsuccessful search tests. Total cell number is 10k. F=24 bits. Notice ACBF does not influence C_{FA}^u.

differences among the MFF algorithms, especially for our implementations, where there is no special parallel programming optimization.

Let C^s and C^u denote the average successful search cost and unsuccessful search

cost by measuring *cell probing length*. For a normal hashing (NH) with separate chaining in m buckets, the search costs can be shown by the closed-form expressions [65]: $C^s_{NH} = 1 + \frac{\alpha}{2} - \frac{1}{2m}$ and $C^u_{NH} = \alpha$. For an MFF, an unsuccessful search is always terminated when all levels are probed for an item not existing in the filter. Therefore, for FA, the unsuccessful search cost can be simply expressed as the sum of load factors of each level

$$C^u_{FA} = \sum_{i=1}^{D} \alpha_i \tag{4}$$

For a successful search of FA, let C^s_1 denotes the successful search cost where the wanted item is found in LV_1. Like NH by separate chaining, assume the wanted item is located in LV_1, the probing length of a successful search is the number of items appeared before the item in its bucket plus one, and $C^s_1 = 1 + \frac{\alpha_1}{2} - \frac{1}{2BN_1}$. On the other hand, if the wanted item is found at $LV_i, i > 1$, the search must experience an unsuccessful search before probing LV_i. Thus, C^s_i for an item in $LV_i, i > 1$ can be expressed as:

$$C^s_i = 1 + \frac{\alpha_i}{2} - \frac{\alpha_i}{2BN_i} + \sum_{j=1}^{i-1} \alpha_j, for \quad i > 1$$

Then, the successful search cost of FA can be modeled as:

$$C^s_{FA} = \sum_{i=1}^{D} LD_i \cdot C^s_i \tag{5}$$

By simulation and Eqs. (4) and (5), Fig. 14 shows that the C^s of FA and LL at high loads are both lower than those of DLCBF apparently, because the large part of items of FA/LL are located in LV_1. Next, the C^s_{FA} and C^s_{LL} of (8,4) are smaller than those of (4,8), because of their similar LD_i but different α_i values. In contrast, C^s_{DLCBF} of (4,8) and (8,4) are very close. Finally, in Fig. 15, notice that C^u_{FA} is clearly smaller than those of LL and DLCBF, because $\sum \alpha_i$ of FA is the lowest in the three algorithms. In summary, C^s_{LL} is slightly higher than $C^s_{FA}$and better than C^s_{DLCBF} obviously. C^u_{FA} is the lowest, and C^u_{LL} is similar to C^u_{DLCBF}.

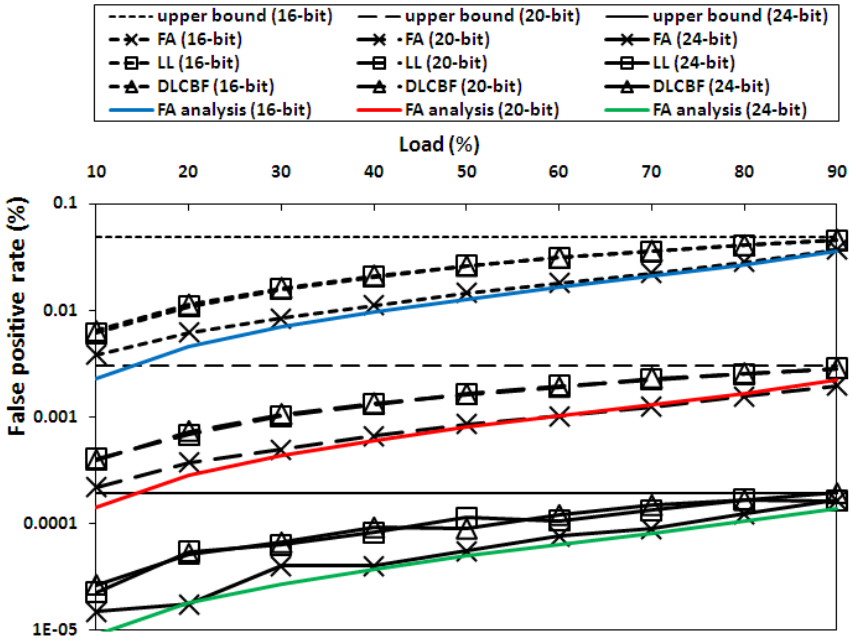


Fig. 16. The measured and expected false positive rates of FA, LL, and DLCBF with (4,8) under different F bits and filter loads. Each measured rate is the average of 20-run experiment results. Each run contains 10^7 false-positive tests. Total cell number is 10k.

D. False Positive Rates

For an MFF, a false positive occurs if and only if for a query of $y \notin S$, there exists $x \in S$ with $h_f(x) = h_f(y)$ in any BE indexed by $(h_i(y) \bmod BN_i)$. Namely, the fraction that this event occurs, called as false positive rate (P_{FP}), is calculated from the probability that one of all possible cells produces the same fingerprint for $y \notin S$. Thus, a higher D or H increases $Load^{max}_{(D,H)}$, but increase the probability of hash collisions and resulting P_{FP}.

The P_{FP} of an MFF(D,H) can be upper bounded by $D \cdot H \cdot 2^{-F}$. Thus, MFF(4,8) and F =20 bits yield a P_{FP} less than $3.051 \cdot 10^{-5}$. This P_{FP} is low enough for many applications, and $Loads^{max}$ of MFF$(4,8)$ all exceed 90%. Besides the discussion in Sec. 3.5, this is the reason of choosing $(4,8)$ and F =20-bit as default settings. Furthermore, P_{FP} of an MFF can be expressed as

$$P_{PF} = \sum_{i=1}^{D} \alpha_i \cdot 2^{-F}$$

Fig. 16 shows the P_{FP} of MFFs. Like C^u of MFFs, FA has the lowest P_{FP}, and the P_{FP} of LL and DLCBF are very similar. Finally, compared to CBF, MFFs use less memory; usually saving a factor of two *at least* for the same P_{FP}.

3.2 Example: Replication of Traffic Classification

By simulation, traffic classification (TC) is used to demonstrate the characteristics of MLCBF on state-machine replication. These results are not meant to cover the tradeoffs and real traffic mixes, but to give insight into practice and applicability of MLCBF in stateful replication.

Network traffic classification (e.g., [66],[67]) is a typical example that needs state consistency amongst SPEs. This important technique categorizes packet flows for various applications, like security, load balancing, billing, and QoS. The most common approach is to rely on deep packet inspection for searching specific characters in payloads. The patterns used for identification probably exist only in the particular segments (e.g., the packets in very beginning of a flow). Thus, by replicating classification results, these collaborated SPEs readily identify packet flows in face of SPE failure and flow migration in AA scheme.

We perform a simulation to investigate the performance of imprecise replication methods. We assume P2P, Skype, and instant messaging use application emulation by tunneling their communication over well-known ports (e.g., TCP port 80). Thus, according to a *pre-defined* state transition diagram (9 states totally), a classified flow could be reassigned to another state (e.g., from a state of “port 80” to “WWW” or “P2P”). Notice the transition diagram is not shown for the sake of brevity and a different FSM does not affect the evaluation results; a transition is processed by all replication methods for comparison.

The simulation details are as follows: the flow key is TCP four-tuple, and the maximum flow number of SPE is 20k (i.e., 20k/ $Load^{max}$ cells for MFF). The parameters of CBF are described in Sec. 5.1. A simulation consists of 180 rounds. Initially, a number of items are inserted to the filters by their insertion algorithms according to a specific filter load. Then, in each round, we insert 2k to 10k new items with random keys, update their states according to the transition diagram to the

filters, send the update messages to the network interface, and remove them from the filters finally. This emulates an SPE of TC deploying replication that processes traffic for 180 sec at the rate of 2k to 10k cps under different initial filter loads. Figs. 17 to 19 illustrate the simulation results.

Fig. 17 shows the insertion costs of FA are much lower than those of LL and DLCBF, especially with the assistance of ACBF. The cost of FA(8,4) is less than that of FA(4,8). This implies FA(8,4) may have a better performance with the support of hash hardware accelerator and parallel techniques. Furthermore, the lookup and deletion cost (not shown in the figure) of these schemes closely match their performance of successful search costs.

Figure 18 shows that the bandwidth consumption of CBF is higher than precise replication under the flow rates less than 6k cps. When the flow rates are larger than 8k cps, the bandwidth requirements of CBF are improved because table updates are activated. The bandwidth reductions of MFF methods are 53% to 72%, depending on (D, H), filter sizes, and flow rates. An MFF method is also benefit from table updates when processing high-rate traffic loading, because a smaller filter size enables the table update at a lower flow rate. In the test, the sizes of FA(4,8), CBF, and DLCBF(4,8) are 93 kBs, 195 kBs, and 95 kBs, respectively. By table update, the bandwidth consumption of an imprecise method is deterministic. MFFs meet the design goal of a scalable solution [68]; they continue to function gracefully as the load grows due to constant costs. In contrast, the cost of precise replication is proportional to the connection rate.

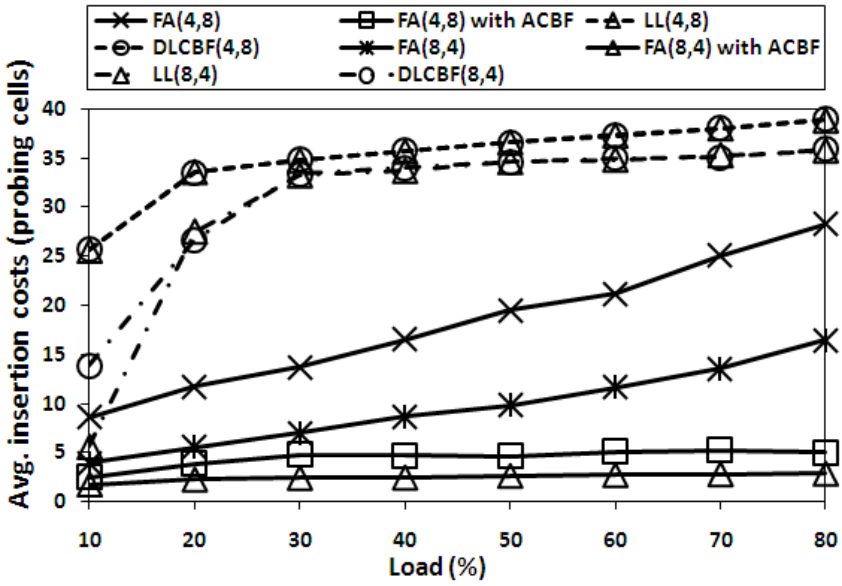


Fig. 17. Measured insertion costs in TC simulation. The flow rate is 10k cps.

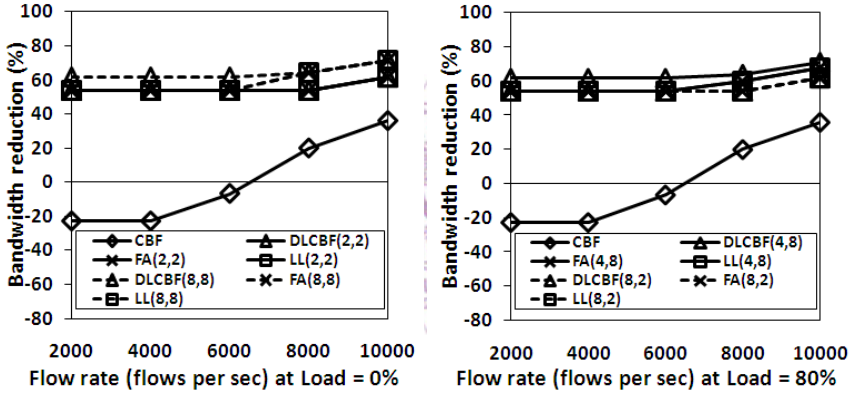


Fig. 18. Bandwidth reduction of approximate methods as compared to precise replication at different flow rates and filter loads.

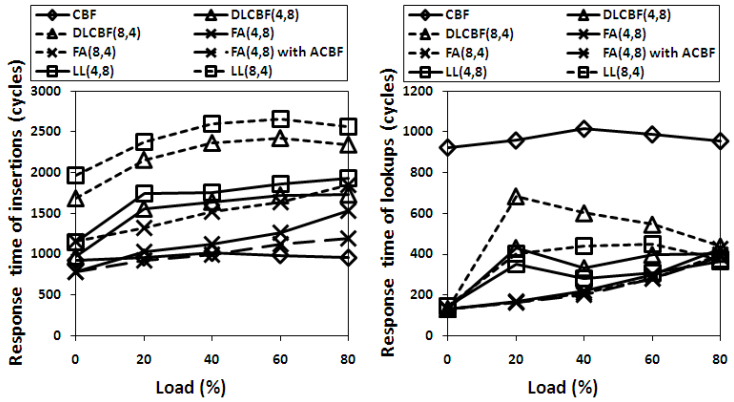


Fig. 19. Average insertion and lookup times of MFFs and CBF at different loads. Flow rate is 10k cps.

By rdtsc [69], Fig. 19 reports the measured CPU cycles of $insert()$ and $lookup()$ implementations. First, the cycles increase as the load increases for an MFF and the

performance of CBF is proportional to the number of hash functions. For any given load, the insertion cycles of FA are lower than those of DLCBF and LL and ACBF indeed enhances the performance of FA, especially for insertions. Moreover, FA provides a considerably better performance than other methods in lookups. For instance, FA yields an improvement of 2 to 7 times for CBF on lookups. For all methods, their cycle times of $modify()$ and $delete()$ are similar to those of $lookup()$.

Second, because MFFs are implemented by level-based hierarchy, increasing D drastically impedes the performance. In Fig. 19, the latency of $(4,8)$ is better than $(8,4)$ obviously. In the receiver, the costs of all methods to update the backup filters for single incremental message are below 550 cycles. In summary, FA has the same bandwidth reduction ratios with LL and DLCBF, and it outperforms other methods on the latency, especially with the support of ACBF for insertion.

In simulation, the average time for inserting an item into normal state table at 60% load is 67,875 cycles; much higher than those of imprecise methods. Though the measurement on CPU cycles highly depends on the implementation, it gives us a practical look at the differences among the replication methods and logical architectures in Figs. 1 and 20. We significantly reduce the latency of replication processing in the receiver by compact filters and the strategy of architectural separation.

Finally, for MFFs, we favor setting H to be a larger and practical value and a smaller D according to the above results. Though $(4,4)$ may be a better choice because of its lower insertion and search costs, $Loads^{max}$ of MFF(4,4) are only around 84%. We choose $(4,8)$ as the default setting due to its balance on $Load^{max}$, filter size, and latency.

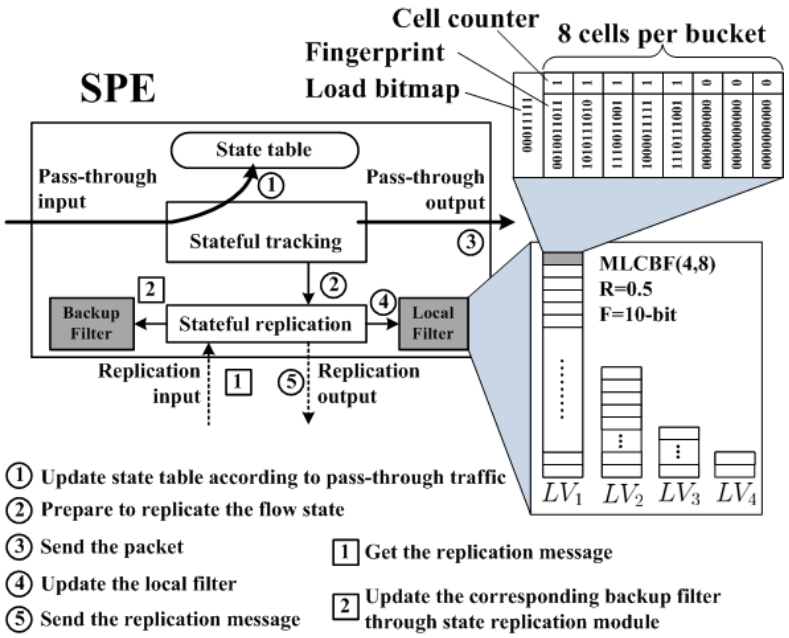


Fig. 20. New data flow inside SPE using MLCBF as replication data representation.

3.3 MLCBF as Key-and-State Representation

Fig. 20 illustrates a new design of logical SPE architecture. I improve replication through two factors: 1) a compact data structure is designated to store and replicate the state changes, and 2) an architectural separation prevents state table from the access by replication traffic. This makes state table inaccessible to RMP, alleviates resource competition, and avoids valuable table entries occupied by the other SPE before necessary.

In Fig. 20, the state table for storing $< precisekey, state >$ can be replaced further by MLCBF or *stateful Bloom filter*. This minimizes memory costs effectively at cost of introducing false error probability on the tracking of pass-through traffic. In my implementation, I still use a hash table to store precise data to verify the false rates and error conditions in all experiments.

For key-and-state access, I utilize MLCBF to support $insert(key, state)$,

$modify(key, state)$, $lookup(key)$, and $delete(key)$ operations. $2^C - 1$ is equal to the highest state number. The idea is simply to store the fingerprint $h_f(x)$ of a key x and its state into MLCBF. For example, if the state transition of an URL (i.e., key) is changed from 2 to 7, the hash signatures of the URL, including $h_1, \ldots, h_D$ of D levels, $h_f(x)$, and digital indices into ABCF, are calculated. Then, the cell is found by $h_i(URL) \bmod BN_i$ and $h_f(URL)$, and the CC is updated to 7 accordingly.

In practice, using randomization to represent the flow states may introduce some types of errors. If the lookup on a non-inserted flow returns a valid state, it is called as false positive (FP). If no valid state for an active flow, it is called as false negative (FN). If the returned state is not correct, it is called as an inaccurate state (IS). These errors are not only introduced by hash functions (i.e., P_{FP}) but probably also by bucket/counter overflows, fingerprint collisions during dynamic operations, and early recycling of active flows by memory management.

In update phase, two methods called *table-update* and *incremental* (*delta*) *update* are used as the representational units. The table-update copies entire filter and keeps the bandwidth requirement as constant, which is critical at high flow rates. However, copying entire filter is clearly not economical at small flow rate or high update frequency because the filter may be only slightly different from the previous one. An alternative method is to use *delta* or *incremental messages*. The messages of CBF and MFF are $< hashindex, state >$ and $< level, bucket, h_f(x), state >$. Though the size of single message of CBF is smaller than MLCBF, as a Bloom filter-based method, the message number of CBF is identical to the number of hash functions. Incremental update can be used in various flow rates while table update gives an upper bound of communication costs. A sender depends on the total message size at a transfer to decide whether incremental messages or entire filter should be sent.

Stateful replication can be performed immediately or based on a specific criteria (e.g.,

periodic or false-rate-control [70]). The first strategy keeps maximum consistency and the second one usually alleviates CPU loads.

MLCBF can be enhanced by the techniques to handle the counter overflows [33] and bucket overflows (e.g., by expanding D) and the techniques like item migrating or rehashing during an insertion [71],[72]). However, these techniques complicate RMP and need more network costs. Thus, I simply avoid any overflow by sufficient cells and CC bits.

3.4 Symbol Replacement for MLCBF

If the number of state changes exceeds a threshold (e.g., five), an orthogonal scheme called *symbol replacement* can be applied to incremental update for higher compression ratios for various flow rates.

The idea is similar in spirit to the IP header compression [73] which converts the constant part of an updating message to a *replication number* to remove redundant overhead in each message. There are two types of incremental messages: uncompressed and compressed. For the first message of an active flow, the sender gets the first free entry (with an entry ID) in a list to store the constant information (e.g., for MFFs, they are the fields of level number, bucket index, and fingerprint), and marks its type as uncompressed. Upon receiving an uncompressed message, the receiver also gets the first free entry to copy static data. Thus, the immutable parts of the following messages (marked as compressed) of the flow are replaced by a replication number (i.e., entry ID). In the receiver, it looks up the entry by the replication number and decompresses static data to update the backup filter.

It is necessary for symbol replacement to ensure the entry IDs allocated by a flow are identical both in the sender and receiver. Therefore, it is imperative that the order of incremental messages chosen by the sender must be followed equivalently at each receiver. For simplicity, we assume that SPEs communicate by TCP or reliable UDP. Thus, because the messages are sent in-order, the receiver can execute the same order of the sender so far.

3.5 Dynamic Lazy Insertion on Stateful Replication

Thus far, to guarantee consistency, state changes are forced to be synchronized precisely. However, this approach is expensive sometimes, especially when system is going to be overloaded. To meet the 3rd design goal in Sec. 2.3, I explore an adaptive mechanism for TCP flows to control replication based on the utilization of system bottleneck. I assume that system is CPU-bounded. Notice for the systems like intrusion prevention systems and traffic classification appliances, CPU-bound is not rare because they are usually deployed in campuses and ISPs to handle high-rate short flows and traffic over 800Mbps.

Measurements of the Internet traffic have shown that most TCP flows are short-running [74],[60] and that small long flows (e.g., less than 20%) carry a high proportion (e.g., 85%) of the total traffic [60],[61]. These studies imply that the major costs come from short flows when doing replication, but focusing on long flows protects the majority of network traffic (in bytes) and profits. Furthermore, though approximate replication performs better in bandwidth and memory requirements, it might be difficult to significantly alleviate the CPU loads from replication.

To address this issue, I differentiate between short and longer flows. *Dynamic Lazy Insertion* (*DLI*) is proposed for balancing the replication loading and the protection on pass-through flows to optimize system throughput. DLI has the advantages of fast estimation, simple to implement, and no requirement of pre-existing knowledge for network traffic.

To detect long flows, several metrics can be used, like packet number [62] and

aggressive flow [3]. For simplicity, I resort to a lifetime-based classification. The term *flow age* and *flow lifetime* are used to indicate the duration time a flow has so far existed and the total time of a flow from start to completion. The basic idea of DLI is as follows: a dynamic lazy threshold $t^{threshold}$ is used by incorporating the information of the CPU utilization and historical flow behavior over the cluster node. The CPU utilization $U_{measured}$ is measured at given time intervals. If $U_{measured}$ exceeds a pre-defined usage threshold $U_{threshold}$ below 100% (i.e., the system becomes overloaded), $t^{threshold}$ for the next interval is increased to filter more "shorter" flows. Otherwise, $t^{threshold}$ is decreased. Namely, for a "longer" flow, the replication is postponed for $t^{threshold}$. For the flows whose lifetimes are shorter than $t^{threshold}$, no operation will be invoked. For some SPEs, focusing on the long flows improves the resource utilization by fewer socket I/Os and hash calculations, thereby reducing the false positives and the frequent accesses on the remote state table. Replication with $t^{threshold}$ > 0 ms is called as *lazy replication*. Otherwise, it is called as *immediate replication*.

For smoother operations on the lifetime measurements, it would be beneficial to incorporate historical information in DLI. In general, stateful tracking keeps track of flow lifetimes since booting. The times t_{min} and t_{max} are defined as the minimum and maximum boundaries of lifetime tracking. If the age of a flow exceeds t_{max}, then its lifetime is immediately updated as t_{max} to update the longer flows in more "real-time". The maximum trigger is identical to the timeout of SYN_SENT state (e.g., 20 sec [37]). On the other hand, accurate lifetime measurement requires small time granularity and the interval of periodic clock interrupt (10 ms on most x86 systems) is set as the minimum bin size. A history array stores the number of total completed flows and counters for all lifetime bins.

Algorithm 2 Pseudo-code of DLI

Function *DLI*

wait for $T_{interval}$
if $U_{measured} > U_{threshold}$ **then**
    $p \leftarrow p + 1$
else
    $p \leftarrow p - 1$
endif
calculate the next lazy threshold by $F_p(i)$, update the latest statistics
to the past history array, and reset the array of the latest interval

---

To compute $t^{threshold}$, I use cumulative lifetime distribution to map the desired level of replication degradation. Let's consider M discrete levels that are numbered $1, \ldots, M$ from the lowest degradation to the highest one. Level 0 is denoted as the special case of immediate replication. Let p denote as a dynamic control parameter in the range $[0, M]$ and assume p is an integer. $t^{threshold}$ is supposed to reduce the replication operations by a ratio of $p/M$and then improve the system performance. I therefore suggest a periodic method, in which p is re-evaluated on a sequence of time boundaries $T_1, T_2, \ldots, T_i, \ldots$ and the length between two boundaries is denoted as $T_{interval}$ (say, each 5 sec).

Let $t_i^{threshold}$denote the minimal flow age between t_{min} and t_{max} to be replicated in the interval T_{i+1}. Also let $t_{i,j}^{measured}$ denote the threshold of jth degradation level estimated from the lifetime distribution between T_{i-1} and T_i, and t_j^{past} the threshold of jth level estimated from the distribution between T_0 and T_{i-1}, for $j = 0, 1, \ldots, M$. To balance the stability and responsiveness, the distributions in the previous interval T_i and the past history are both used to compute a $t^{threshold}$ for the next $T_{interval}$. Thus, two history arrays are used. Then I consider a function $F_p(i)$ defining $t_{i+1}^{threshold}$ at time T_i in the following way:

$$\begin{cases} F_p(1) = t_{1,0}^{measured} \\ F_p(i) = (1 - R^{DLI}) \cdot F_p(i-1) + R^{DLI} \cdot t_{i,p}^{measured} \end{cases}$$

In the above equation, $F_p(i-1)$ is equal to t_p^{past}. The factor R^{DLI} is a parameter in the range $[0, 1]$. R^{DLI} is set as 0.7 to put a higher weight on the observation of the

latest interval for better responsiveness. After computing $t_{i+1}^{threshold}$, the statistics in the interval T_i is updated to the history array for the interval T_0 to T_{i-1}. Algorithm 2 describes the self-tuning algorithm. By adjusting p, I can dynamically control the replication costs under various traffic mixes, especially CPU utilization. A lower $U_{threshold}$ (e.g., 60%) leads to a better peak throughput when overloading in price of an earlier lazy replication. Furthermore, for adapting to the locality of lifetime distribution, DLI provides more flexibility by making a smaller step size (e.g., 20 ms) when p is low and a larger size (e.g., 200 ms) when p becomes high. Note that the state changes of a flow whose replication is postponed are still transmitted in-order to keep RMP simple.

The steps to compute the lifetime threshold (denoted as $t^{threshold}$) periodically of Dynamic Lazy Insertion (DLI) is the focus in the rest of this section. Figure 21a illustrates how to compute the lifetime threshold by a mapping between M discrete levels and a cumulative lifetime distribution observed from t_{min} to t_{max}. In Fig. 21a, M is set to 10, while Level 0 represents no lazy replication. For example, in Fig. 21a, the lifetime threshold of Level 8 is thus supposed to reduce 80% of replication costs. Notably, p is the current degradation level in the range $[0, M]$, and p is re-evaluated periodically on a sequence of time boundaries $T_1, T_2, \ldots, T_i, \ldots$. When measured CPU utilization (denoted as $U_{measured}$) is larger than a pre-defined CPU threshold (denoted as $U_{threshold}$), the degradation of replication operations starts from Level 1 and continues to Level M if necessary. Restated, every $T_{interval}$, if $U_{measured} > U_{threshold}$ (i.e., overloading), p is increased to alleviate the CPU loads from replication operations. Otherwise, p is decreased.

Figure 21b depicts the concept and steps of computing lifetime threshold $t^{threshold}$ at T_i for the next interval. Assume p for the interval between T_i and T_{i+1} is increased from Level 2 to 3 due to overloading. Notably, $t_{i,3}^{measured}$ is computed by computing the lifetime threshold of Level 3 from the cumulative lifetime distribution observed between between T_{i-1} and T_i. Next, t_3^{past} is estimated by the cumulative distribution observed from T_0 to T_{i-1}. The lifetime threshold $t_i^{threshold}$ for the

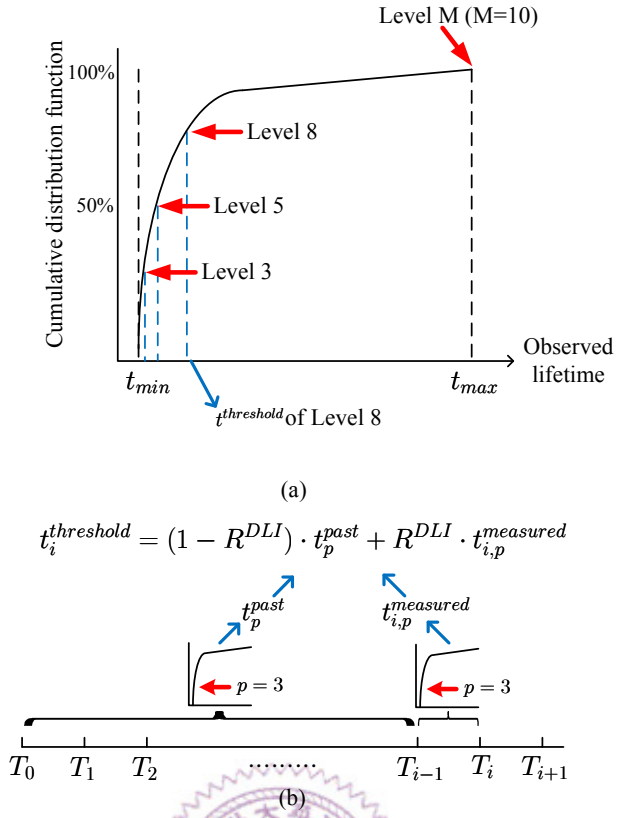


Fig. 21. a) Mapping between degradation levels and corresponding lifetime thresholds over an observed cumulative lifetime distribution, and b) an illustration of the steps to compute $t_i^{threshold}$.

interval between T_i and T_{i+1} is then computed as

$$t_i^{threshold} = (1 - R^{DLI}) \cdot t_3^{past} + R^{DLI} \cdot t_{i,3}^{measured}$$

In this work, the factor R^{DLI} is set as 0.7 for better responsiveness.

TABLE II: IP PACKET TRACES FROM NLANR [63].

IP packet trace name and time	Max. active TCP flows	Avg. SYN pkts per sec
IPSL-1 (2001/08/14/09:00 – 09:10)	159,394	1199.12
IPLS-3 (2004/06/01/19:40 – 19:50)	159,210	5796.61
AUCK-4 (2001/04/03)	38,604	308

TABLE III: SIMULATION RESULTS OF URL CATEGORIZATION BY REAL URL COLLECTIONS FROM NLANR [63].
MEMORY AND NETWORK BANDWIDTH REQUIREMENTS OF IMPRECISE REPLICATION METHODS.

Access list name and time	Total access logs	Total unique URLs	Mean URL size (bytes)	URL string size (kBs)	CBF				MLCBF-FA(4,8), F=20-bit			
					Mem (kBs)	Reduction	Net (kBs)	Reduction	Mem (kBs)	Reduction	Net (kBs)	Reduction
bo2(2007/01/09)	241,173	144,852	57	13,534	2,355	82.6%	2,247	83.4%	1,271	90.6%	848	93.7%
bo2(2007/01/10)	207,704	133,420	56	11,384	2,028	82.2%	2,069	81.8%	1,095	90.4%	781	93.1%
rtp(2007/01/09)	3,176,785	1,653,579	59	184,066	31,023	83.2%	32,100	82.6%	16,752	90.9%	9,688	94.7%
rtp(2007/01/10)	2,986,122	1,501,494	58	169,990	29,161	82.9%	29,153	82.9%	15,747	90.7%	8,797	94.8%
sd(2007/01/09)	1,426,885	879,114	55	77,341	13,934	82.0%	13,638	82.4%	7,534	90.3%	5,151	93.3%
sd(2007/01/10)	1,497,891	933,756	55	81,420	14,627	82.0%	14,484	82.2%	7,899	90.3%	5,471	93.2%

3.6 Implementation and Testbed Setup for MLCBF

The experiments were performed on a testbed consisting of two identical 3-port machines (Intel Pentium-4 2.0 GHz, 512-kB L2 cache, and 1024 MBs RAM) as the SPEs in HAC. Two SPE nodes are connected with a 100 Mbps LAN (the replication link) and the external and internal ports are connected with Gigabit Ethernet networks (the pass-through link). To measure CPU utilization, Linux facility dstat is executed with 1-sec bin. Other statistics, such as update message size, are collected with 5-sec bins. By the design of Fig. 20 and C language, replication methods, DLI, and symbol replacement are implemented as a kernel-space module in Linux 2.4.20. As a state table, a hash table by separate chaining is used to store precise keys, (states) counters, and metadata.

In this dissertation, four types of traffic generators are used: powerful IXIA machines, simulator of traffic classification, simulator to analyze the filters, and real traffic traces. Notice the simulation to study the filter properties is also performed by my prototype platform. The random keys in test are read from Linux /dev/urandom. The

TABLE IV: DEFAULT PARAMETER LIST IN EXPERIMENTAL TESTS

	Parameters	Description	Value
MFF	(D, H)	(level, height) of an MFF	$(4, 8)$
	$Load^{max}$	Maximum achievable load	93.65% for FA$(4, 8)$
	R	Decreasing ratio of MLCBF	0.5
	F	Fingerprint size	20-bit
Dynamic Lazy Insertion	$t^{threshold}$	Lazy threshold	50 – 2k ms
	$U_{threshold}$	Boundary CPU load to change $t^{threshold}$	90%
	p	Current lazy level	0 – M
	M	Maximum lazy level	10
	t_{max}	Maximum flow lifetime	20k ms
	t_{min}	Minimum flow lifetime	10 ms
	$T_{interval}$	Interval to compute next $t^{threshold}$	5 sec
	R^{DLI}	Responsiveness ratio	0.7

prototype of SPE synchronizes its receiver by reliable UDP. For TCP flows, inactivity timeouts for the entries stayed in SYN_SENT, ESTABLISHED, and FIN_WAIT are 20, 60, and 20 sec.

For CBF, the number of hash functions is 4, and *load factor of CBF* (i.e., m/n, the ratio between the number of filter slots and the number of inserted items in maximum) is 10. The P_{FP} is about 1.2% in theory.

The hashing throughput is studied by many works. SHA-1 is chosen because of its available fast implementation. I use an open-source SHA-1 with modification to support the requirements of D hash functions, fingerprint and signatures of CBF.

In practice, network traffic is time and link dependent. This makes it impossible to evaluate all possible traffic mixes. To overcome this limitation, I validate the proposed methods by trace-based simulation on IP packets and URL access logs. The environment and parameters of trace-based simulation are identical to those of testbed experiments. After processing all logs, the FP rates are verified by a round of

10^6 items with the random keys (as URLs or TCP four tuples) that do not exist in state table. The FN and IS rates are verified by comparing the precise key and state stored in state table with the data in filters.

For URL applications, six one-day collections of HTTP requests downloaded from NLANR [63] are used in analysis. Table I lists the information of traces in detail. URL string size is the sum of all distinct URL string lengths.

For SPEs on TCP flows, the replication schemes are applied to the bi-directional 10-min traces of Abilene-I, Abilene-III, and the University of Auckland (denoted as IPLS-1, IPLS-3, and AUCK-4). Table I shows the maximum concurrent TCP flows and packet arrival rates. For fair comparison, I ignore purposely the replication of the flows whose SYNs were not captured, though this leads to an underestimation of pass-through traffic (especially long flows) and replication costs. Furthermore, due to the fact that routes may be asymmetric at backbone, there is a minor-tuning in TCP tracking.

4. Evaluations of Stateful Replication Using MLCBFs

4.1 Replication for State-Machine Tracking

A. URL categorization

In URL categorization, there are numerous master servers to collect and classify URLs through the technology of web content classification. As shown in Fig. 1, the categorization result enables the SPEs in gateway to classify HTTP traffic by URLs. An operator can thus establish management policy by useful categories, such as malicious threats and adult content. There are 50 to 90 categories usually, which are represented by integers.

Usually, an URL request received by an SPE is sent to one of master servers for classification. A service provider reported that they receive over 100 million requests for categorization per day. Thus, the caching of categorization results in SPEs can accelerate pass-through web traffic, alleviate the loading of master servers, and reduce bandwidth costs among the SPEs and masters.

To study replication performance, a state number, which follows a Poisson distribution between 1 and 127, is assigned to each distinct URL of a trace file. All request logs are inserted to state table to find out unique URLs at first, and no URLs are removed during the experiment. The distinct URLs are then inserted to the filters to trigger update messages. The number of cells of an MFF is set up according to the number of total access logs.

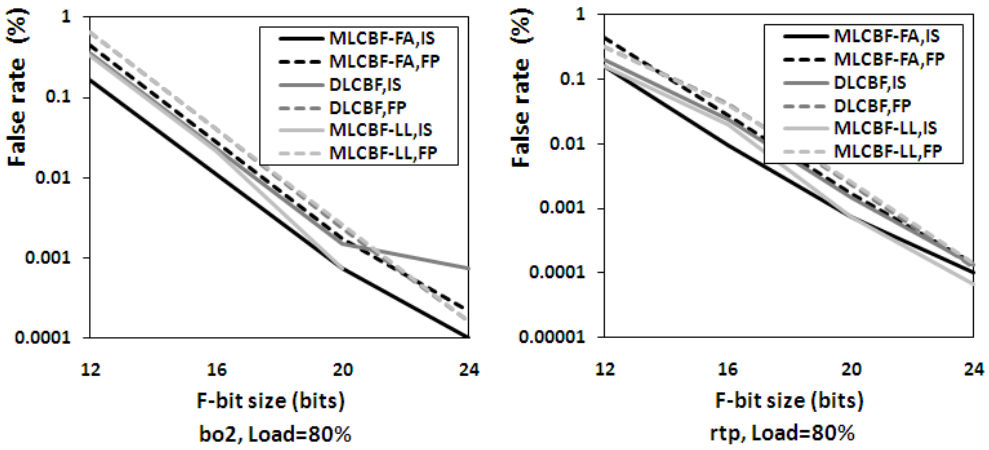


Fig. 22. FP and IS rates of MFF-based methods in bo2(2007/1/10) and rtp(2007/1/10) at 80% load for URL categorization.

Table III shows the resource reduction ratios achieved by FA and CBF. We use “URL string size” as the size of a key to compute memory/network requirements of precise replication. The memory and bandwidth reduction rates of FA are as high as 90.9% and 93.1% at least. The results indicate the approximate methods reduce the resource consumptions of URL categorization significantly.

By F=20-bit and (4,8), the range of FP rates of MFFs for six collections is from 0.0003% to 0.0012%. The IS rates are all smaller than 0.0014%. The FP and IS rates of CBF at a loaf factor of 10 are 0.637% and 14.96%. By a load factor of 40 for CBF, they are 0.0012% and 1.5% at a cost of 4 times memory requirements. The FN rates are zeros, because of no URL removal and no overflow.

Fig. 22 shows the resulting false rates of MFFs by varying F. With an F of 24 bits, the FP and IS rates are about 0.0001% and the filter size is lower than 21 MBs for rtp(2007/1/10) at 80% load. The resource requirements and false rates observed in the tests are likely to be reduced in practice, because it is expected that an SPE and its backups would not contain so many URLs. It is probably that only a set of the most frequently accessed URLs would be stored. Finally, the average times of the insertions and lookups on the state table take 31,566 and 1,759 cycles. By contrast,

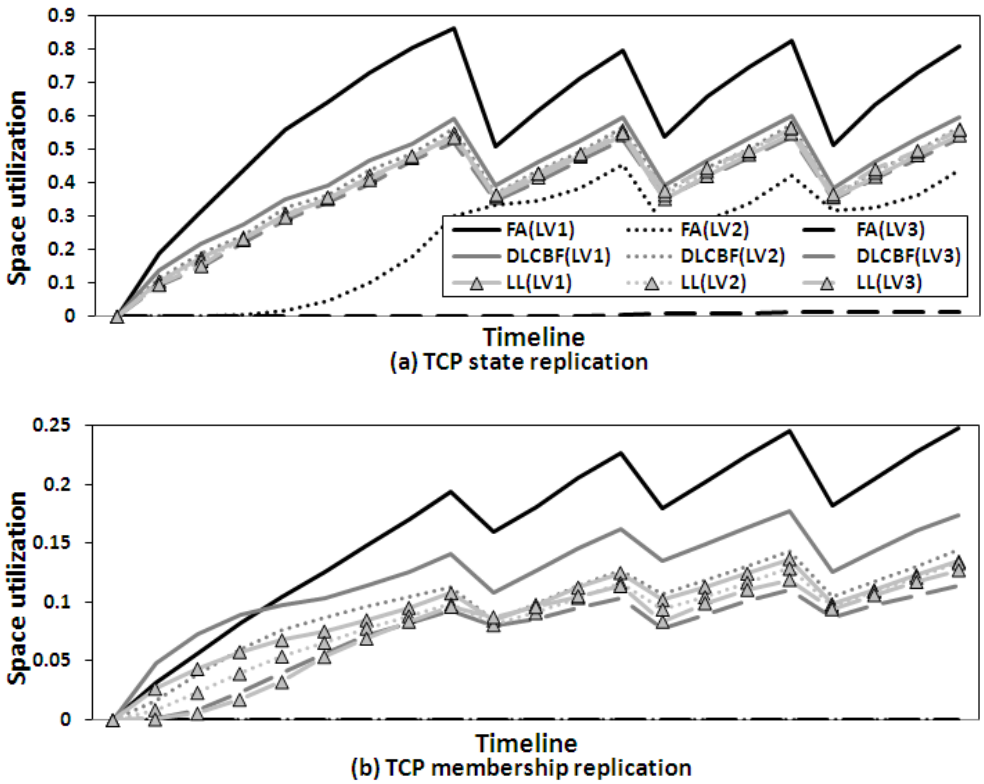


Fig. 23. Storage utilization of MFF(4,8) in IPLS-3 for a) TCP state replication and b) TCP membership replication. Only first three levels are illustrated.

they are only 1,211 and 425 cycles for FA, and 2,426 and 1,210 cycles for DLCBF.

B. TCP State Replication

We tune $t^{threshold}$ from 0 to 2k ms in simulation to understand the performance of imprecise methods and the effects of lazy replication. Driven by TCP flows from the traces, replication methods synchronize six state changes, including SYN_SENT, SYN_ACK_RCV, EST, WAIT_CLS, HALF_CLS, and flow completion, by incremental update. By Table III, for IPLS-1 and IPLS-3, the total cell numbers of MLCBF, DLCBF and CBF are 170,848, 173,900 and 2,000,000. For AUCK-4, they are 42,712, 43,400, and 400,000 cells, respectively. In IPLS-1 and IPLS-3, the sizes of FA, DLCBF, and CBF are 750 kBs, 764 kBs, and 1,953 kBs; all not a concern for modern equipments.

Figures 23 to 25 illustrate the results of TCP replication in AB scheme. Fig. 23 shows

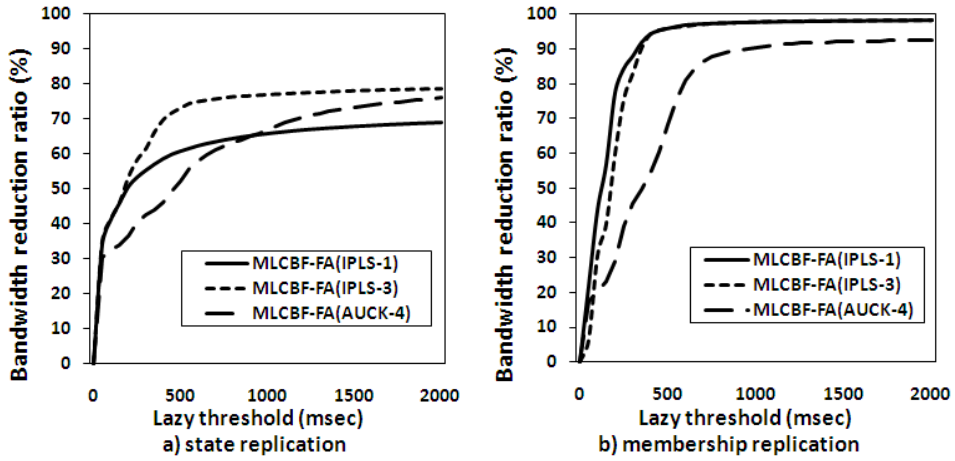


Fig. 24. The effect of $t^{threshold}$ on FA for a) state replication and b) membership replication.

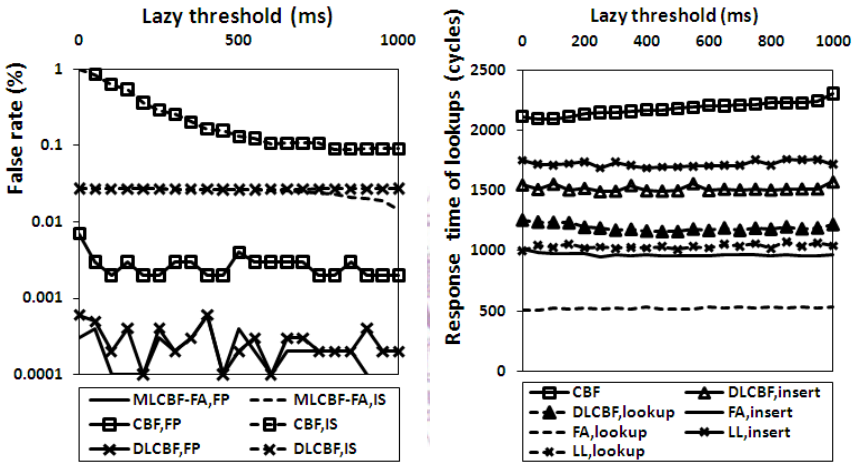


Fig. 25. False rates and operation overheads of CBF and MFF-based methods in IPLS-1 trace for TCP state replication.

the storage utilization of MFF in IPLS-3. The oscillation of utilizations in Fig. 23 comes from inactivity timeouts. Obviously, LV_1 and LV_2 of FA fill up with the most part of flows.

In state replication, we found $t^{threshold}$ does not obviously influence the bandwidth savings of MFFs and CBF. The reduction rates of CBF in AUCK-4 for all $t^{threshold}$ are around -15.14% as compared to precise replication. By contrast, the reduction rate of FA is 61.54% and is improved to 71.23% by symbol replacement.

Because most TCP flows are short-lasting, they dominate replication cost. Fig. 24 shows the effects of TCP flow lifetimes on state and membership replication. Fig. 24a

shows that when $t^{threshold}$=50 ms, due to the savings of replicating one-way flows [64] and malicious SYN packets, the reduction ratio of FA is as high as 35.11% in IPLS-3. In IPLS-3, flow analysis shows that 87.1% of the recycled SYN_SENT entries are allocated by one-way flows. These one-way flows significantly consume replication bandwidth unnecessarily.

A clear rise on the replication traffic reductions is observed when $t^{threshold}$ < 500 ms. For IPLS-3 in Fig. 24a, reducing 53.21% ($t^{threshold}$=200ms), 75.25% ($t^{threshold}$=650 ms), and 78.52% ($t^{threshold}$=2k ms) of replication traffic only excludes 12% of pass-through bytes in maximum. Thus, a small $t^{threshold}$ like 500 ms (or ignore the flows less than six packets [74]) can be useful for alleviating system load, reducing bandwidth consumption, and still protecting the majority of traffic bytes for TCP replication.

Fig. 25 shows the false rates and operational overheads of TCP state replication in IPLS-1. Like URL categorization, the false rates of MFFs are much lower than those of CBF which is not suitable for TCP state replication.

4.2 URL and TCP Membership Replication

A. Comparison to Summary Cache

Summary Cache (SC) [7] can be viewed as a kind of stateful replication; it summarizes a snapshot of incoming URLs for a proxy by a CBF from scratch and keeps the filters consistent between its local cache and neighbors as URLs are inserted and deleted. With the same parameters of URL categorization, we compare the performance of SC and MFF(4,8) for propagating a snapshot of URL collection.

By incremental updates for rtp(2007/1/10), SC using load factor of 10 transmits 21,252 kBs versus 8,797 kBs of FA. The reduction rates of SC for six collections are between 81.9% and 88.86% for memory and bandwidth requirements compared to

precise replication. The reduction rates of MFFs on memory requirements range from 90.27% to 90.9% and the bandwidth reduction rates are between 93.28% and 94.82%. In bo2(2007/1/10), with a load factor of 10 and 4 hash functions, the average P_{FP} of SC is 0.026305%, but the P_{FP} of FA, LL and DLCBF with an F of 20 bits are 0.001015%, 0.00173%, and 0.001595%. To achieve a comparable P_{FP} (say, 0.0015%), SC needs a load factor of 40 with 4 hash functions. However, the memory and bandwidth requirements are increased to 7.40 and 2.32 times of FA.

For SC, more hash functions decrease P_{FP}, but increase the memory accesses and bandwidth consumption by incremental update. By contrast, FA can reduce P_{FP} effectively by increasing F, and the bandwidth cost only increases a little. As a useful feature of MFF, this avoids the change of (D, H) and $Load^{max}$ for a smaller P_{FP}.

B. TCP Membership Replication

In [9],[10], Flow Digest (FD) is used to replicate the membership information of pass-through TCP flows (namely, TCP tracking in Fig. 1). To compare performance, all methods generate two events (i.e., EST and flow deletion).

Compared to precise replication, the bandwidth reduction rates of FD in the three traces for $t^{threshold}$ between 50 and 2k ms range from 7.86% to 8.35%, and those of MFF-based methods are 60.07% to 61.73%. Fig. 24b illustrates, at $t^{threshold}$=500 ms, the bandwidth reduction ratio of FA is as high as 95.95% in IPLS-3. Finally, for IPLS-3 without lazy replication, the FP rates of MFFs are all below 0.0008% due to low utilization as shown in Fig. 23b. Since only the established flows are inserted to the filters, the overhead from one-way flows is avoided completely.

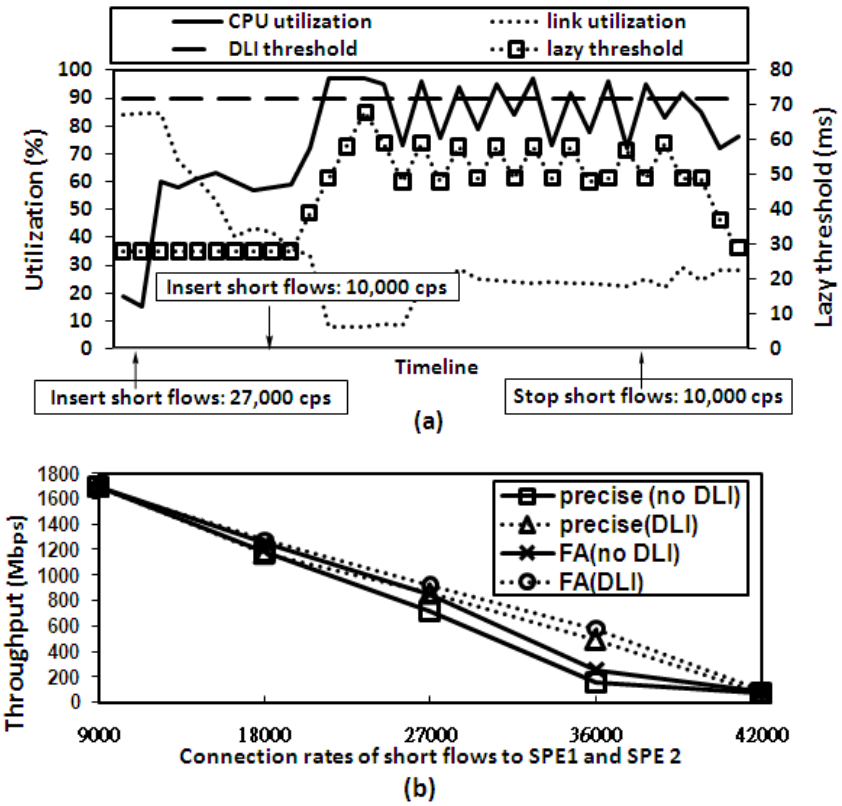


Fig. 26. a) Behavior of primary SPE with TCP state replication and DLI in AB scheme, and b) The aggregated throughput of two SPEs of HAC in AA scheme under high-rate short flows.

4.3 Testbed Study for DLI and SPEs in AA Scheme

To this end, the testbed study is used to demonstrate feasibility and effectiveness of DLI. We also show that imprecise replication increases the aggregated throughput of SPEs on pass-through traffic in AA scheme.

Two connection types, short and long flows, are generated by IXIA machines to stress the primary SPE. A short flow completes its establishment and termination quickly (mainly within 10 to 50 ms). The initial values of $t^{threshold}$ and p are set to 0. Fig. 26a illustrates the system behavior with precise TCP state replication and DLI. Initially, HTTP traffic (long flows) is used to measure pass-through (link) throughput. Then, two sets of short flows (27k and 10k cps) are inserted. With the 1st set, the throughput degrades dramatically, but CPU does not surpass $U_{threshold}$. After 2nd set

insertion, system exhibits saturation. Then, DLI quickly brings CPU to oscillate around $U_{threshold}$ by adjusting $t^{threshold}$ around 48 to 68 ms.

Because the high-rate short flows almost complete within 50 ms and become indivisible in degradation, DLI exhibits over-degradation behaviors; over 98% of replication traffic is eliminated in a $T_{interval}$, where p = 3 only and $t^{threshold}$>50 ms. The problem can be solved by restricting the reduced replicating operations in a $T_{interval}$ in the range $[0, p/M]$ when p > 0. Though this heuristic takes more steps to the optimal point, it avoids over-degradation effectively. In Fig. 26a, DLI controls replication effectively to alleviate CPU load, thereby enhancing the throughput from 77 to 224 Mbps. In contrast, CPU without DLI exhibits saturation by two short-flow sets. Finally, without replication, the maximum throughput under two sets of short flows is 259 Mbps in average.

Next, we measure the end-to-end throughput of our HAC in AA scheme consisting of two SPEs using TCP state replication. In test, two pass-through links of HAC are stressed by the same rates of short flows at first and the aggregated throughput (i.e., pass-through throughput of HAC) of HTTP traffic on two links are measured under these high-rate flows. Fig. 26b reports that SPEs using imprecise replication outperforms those by precise one, especially at short-flow rates over 18k cps. For example, at 27k cps and without DLI, FA improves the aggregated throughput from 716 to 850 Mbps. Moreover, Fig. 26b shows DLI apparently increases pass-through throughput of HAC at 27k and 36k cps. Finally, at 42k cps, two SPEs are almost saturated by incoming pass-through packets; even without any replication.

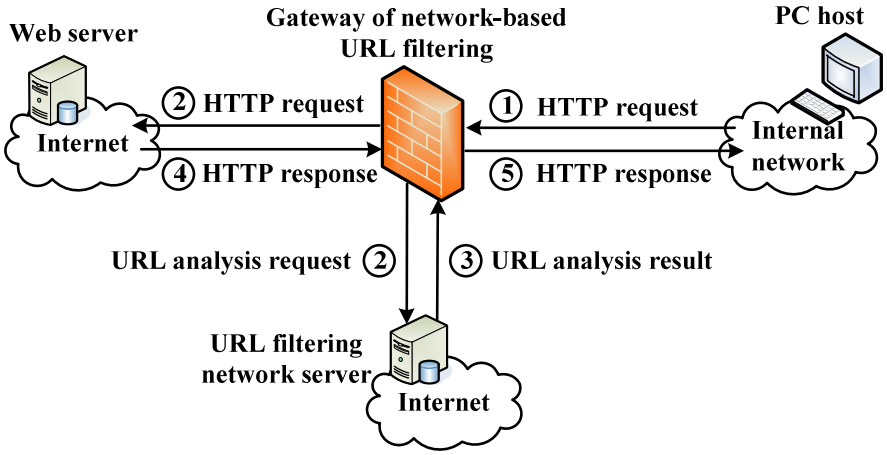


Fig. 27. The general overview of network-based URL filtering.

4.4 Other Potential Applications

4.4.1 Using MLCBF as Data Representation for Stateful Replication

Stateful SPEs using replication are widely deployed in kinds of network environments. Thus, stateful replication using randomization may benefit an application which employs a set of peers to share information. An example is intrusion detection and traffic classification on SCTP [20]. SCTP provides fault tolerance by *multi-homing* which connects to the Internet over multiple network access links.

The SPEs deployed on different links must synchronize for inspecting SCTP associations. Our methods can naturally be used in conjunction with the techniques like cooperative intrusion detection [75] and may be also suitable for other fault-tolerant SPEs, (e.g., WWW firewalls [76] and VoIP intrusion detection [77]).

4.4.2 Using MLCBF as Local Caching of Network-based URL Filtering

In ISP, enterprise, and SOHO networks, URL filtering is widely used to prevent users

to access unwanted and malicious web sites. Several service and device providers like Cisco, Websense, Surfcontrol, Blue Coat, and Gemtek provide network-based URL filtering (NUF) as a solution to classify, monitor, and control web traffic. Figure 27 illustrates the schematic procedure of NUF.

1. The end user browses a page on the web server, and the browser sends an HTTP request to the web server.
2. After the gateway receives HTTP request, it extracts the URL from the HTTP request. The URL is then sent to the network servers for analysis, while the HTTP request is forwarded to the web server simultaneously. For safety, each gateway needs to be authenticated before sending requests to the network server.
3. After the network server receives the analysis request, it checks its database to classify the web site represented by the URL. It then returns an integer representing the category of URL. Restated, the analysis result of a URL is just an integer (i.e., categorization ID) which represents a specific category. For example, "P2P" is represented by integer 2 and "online news" is 30. Notably, there are 50 to 90 categories usually, which are all represented by integers.
4. The HTTP response from the web server is queued for waiting for the decision by the gateway.
5. After getting the analysis result from the network server, the gateway sends or blocks the corresponding HTTP response according to the analysis result and management policy. For example, assume that although the P2P access is not allowed in an enterprise, the management policy allows for P2P access from an internal testing laboratory. Consider the classification result of a specific URL is a web site of P2P forum. If the source IP is in the range of the testing laboratory, the gateway sends the HTTP response to the end user. Otherwise, it sends a warning page to the end user.

NUF provides two important benefits over *gateway-based URL filtering* (GUF) which analyzes the URLs by simply comparing them with the local database in a gateway and updating the database continuously.

1. NUF can employ a cluster of powerful servers to quickly analyze extracted URLs using multiple complex techniques, including blacklist, web content inspection, and intelligent behavioral analysis. Moreover, the network servers are able to cooperate for detecting new malicious URLs more quickly because they collect and analyze URLs in a bird's eye view. Such comprehensive capability can significantly increase the detection coverage and accuracy of URL classification.
2. Compared to GUF, the task of a gateway of NUF is only to extract URLs from HTTP requests, send them to the network server, and wait for the responses. This lowers the implementation complexity of a gateway engine of NUF, while the gateways no longer need to continually update local databases, thereby reducing administrative cost. Next, simplifying the engine allows the service to be used in resource-limited devices (e.g., mobile phones) that lack sufficient computing power but remain as a target of malicious web sites.

However, a service provider of NUF reported that they receive over 100 million requests for URL categorization per day. The bandwidth consumption amongst the gateways and network servers is therefore the key factor of the capacity and the maintenance cost of the service. Furthermore, waiting for the response from the network server indeed introduces processing delay to web traffic. In our preliminary tests, the average network latency from our laboratory to three network servers of a service provider is between 100 to 500 ms. This motivates us to design an efficient model for NUF to reduce the bandwidth cost between the gateways and network servers, while accelerate the processing time in the gateways of NUF. The first idea is the local caching of URL analysis results in the gateways. The second idea is to use a hashing structure as the data representation of local caching.

To minimize the resource requirements of NUF, MLCBF can be introduced to address this issue, and specifically we show that how to integrate MLCBF into the gateway of NUF as local caching. Based on the idea of using Counting Bloom filter (CBF) to store state machine [23], MLCBF is used to cache the URL classification results to minimize the memory requirements.

Based on the idea of using CBF to store state machine [23], MLCBF is used to cache the URL classification results. Restated, the cell counter CC of MLCBF is used to

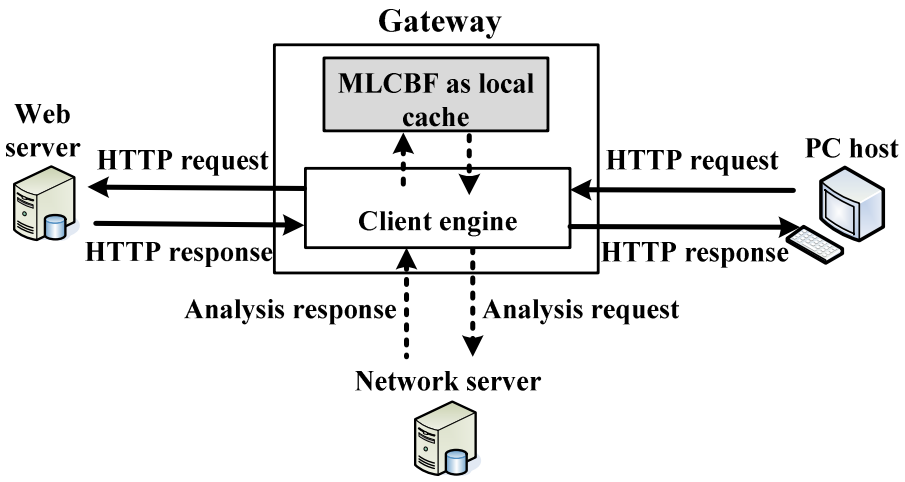


Fig. 28. The model of network-based URL filtering. The basic ideas are to use caching for analysis results and hashing structure as data representation.

store the classified integer of $h_f(URL)$ directly, not used in the way of its original design; as a counter of a specific $h_f(key)$.

Fig. 28 illustrates the proposed model of using MLCBFs in a client engine of NUF. Initially, when a URL is extracted by the client engine from HTTP request, it is searched in the local MLCBF by $h_f(URL)$. If the URL is not found, the client engine sends the URL as an analysis request to the network server for classification. The analysis response is <URL, integer>. After the client engine receives the result, it inserts <$h_f(URL)$,integer> into the local MLCBF. Next time when the same URL gets into the engine, the classification result can be found locally.

About the related works of GUF and NUF, a wide range of techniques have been proposed for enhancing web applications, like web access security, URL forwarding and lookup engine [78], and web proxy caching [7]. Web content filtering is one of popular approaches to provide web access security. The key function of this method is the classification on web pages. In [79], it provides a hierarchical structure for classifying a large collection of web content. In the works of [80],[81],[82], different machine-learning-based methods are used to perform web content filtering. Although those methods provide accurate filtering results, it seems to take too much time to process each web page by multiple intelligent techniques. In contrast, NUF and GUF are more appropriate for ISP, enterprise, and SOHO networks.

URL blacklist is another common method to implement web filtering engine. Allowing HTTP access or not depends on comparing the URL of an HTTP request to

the URLs in the blacklist. In [83], URL filtering is performed based on caching mechanism. In [84], a Wu-Manber-like matching algorithm with a support of CRC32 is used in a URL filtering system. In [85], two functions are proposed for hashing the signatures of URLs which can get efficient URL lookup performance. In sum, similar to GUF, the above works store blacklist in the local filtering engines, and they therefore have to update the databases periodically.

5. Conclusions

This paper has explored how to efficiently replicate key-and-state information of stateful SPEs in an HAC to provide service consistency. We propose a new compact data representation, called Multi-Level Counting Bloom Filter (MLCBF), to employ the effect of insertion distribution over MLCBF levels for storing and synchronizing stateful data of a large number of active flows. Our extensive experiments show that MLCBF considerably reduces the amount of resource requirements in terms of bandwidth/memory costs and low constant operation latency as compared to precise replication. Especially, MLCBF-FA outperforms other imprecise replication methods in the areas of false rates, search costs, and operational time.

The proposed replication methods have been implemented by Linux as a real platform of HAC. Both our analysis and extensive experiments indicate that the performance of MLCBF is very promising under heavy traffic loads. Trace-based simulation shows that MLCBF reduces network and memory requirements typically by 94.7% and 90.9% for URL categorization, and reduces 61.54% of bandwidth consumption for TCP state replication. For URL membership replication, the resource reduction rates of MLCBF range from 90.27% to 94.82%.

We have also proposed three improvements on replication. First, a supporting structure can be used to not only reduce the number of unsuccessful searches significantly of MLCBF-FA but also reduce the costs of successful searches. Second, a self-tuning scheme for TCP flows has been introduced to control replication cost. The testbed and trace-based experiments have shown that adaptation by flow lifetime and CPU utilization can alleviate the loading from short-flow replication, protect the majority of the Internet traffic, and offer an optimal throughput dynamically. At last, an orthogonal scheme used to compress incremental messages improves bandwidth

consumption of multi-state replication.

References

[1] M. Stonebraker, U. Cetintemel, and S. Zdonik, “The 8 Requirements of Real-time Stream Processing,” ACM SIGMOD Record, 2005.

[2] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, “Fault-tolerance in the Borealis Distributed Stream Processing System,” ACM Transactions on Database Systems, 2008.

[3] W. Shi, M. H. MacGregor, and P. Gburzynski, “Load Balancing for Parallel Forwarding,” IEEE/ACM Transactions on Networking, 2005.

[4] F. Schneider, “Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial,” ACM Computing Surveys, 22(4), 1990.

[5] P. Felber and P. Narasimhan, “Experiences, strategies, and challenges in building fault-tolerant CORBA systems,” IEEE Trans. Comput., 53(5):497–511, 2004.

[6] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, “Distributed Systems,” Addison-Wesley, 1993.

[7] 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, 2000.

[8] A. Broder and M. Mitzenmacher, “Network Applications of Bloom Filter: A Survey,” Allerton, 2002.

[9] Yi-Hsuan Feng, Nen-Fu Huang, Rong-Tie Liu, and Meng-Huan Wu, “Flow Digest: A State Replication Scheme for Stateful High Availability Cluster,” IEEE ICC, June 2007.

[10] Yi-Hsuan Feng, Nen-Fu Huang, and Yen-Min Wu, “Evaluation of TCP State Replication Methods in Cluster-based Firewall,” IEEE Globecom, 2008.

[11] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov, “Wrapping Server-Side TCP to Mask Connection Failures,” Proc. IEEE INFOCOM, pp. 329-337, 2001.

[12] N. Aghdaie and Y. Tamir, “Client-transparent fault-tolerant web service,” 20th IEEE International Performance, Computing, and Communications Conference, pp. 209–216, 2001.

[13] R. Zhang, T. F. Abdelzaher, and J. A. Stankovic, “Efficient TCP Connection Failover in Web Server Clusters,” Proc. IEEE INFOCOM, 2004.

[14] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan, “Fine-grained failover using connection migration,” Proc. 3rd USENIX Symposium on Internet Technologies and Systems, March 2001.

[15] R. R. Koch, S. Hortikar, L. E. Moser, and P. M. Melliar-Smith, “Transparent TCP connection failover,” Proc. the IEEE Int. Conf. on Dependable Systems and Networks (DSN’03), June 2003.

[16] M. Marwah, S. Mishra, and C. Fetzer, “TCP server fault tolerance using connection migration to a backup server,” Proc. IEEE Int. Conf. on Dependable Systems and Networks (DSN’03), June 2003.

[17] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode, “Migratory TCP: Highly available Internet services using connection migration,” Proc. the International Conference on Distributed Computing Systems (ICDCS’02), pp. 469-470, 2002.

[18] F. Sultan, A. Bohra, and L. Iftode, “Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions,” Proc. the Symposium on Reliable Distributed Systems, Oct. 2003.

[19] A. Shieh, A. Myers, and E. G. Sirer, “Trickles: A Stateless Network Stack for Improved Scalability, Resilience and Flexibility,” Proc. the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI’05), pp. 175–188, May 2005.

[20] R. Stewart and C. Mets, “SCTP: New Transport Protocol for TCP/IP,” IEEE Internet Comput., 2001.

[21] B. Bloom, “Space/time tradeoffs in hash coding with allowable errors,” CACM 13, 1970.

[22] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, “An Improved Construction for Counting Bloom Filters,” LNCS 4168, 14th Annual European Symposium on Algorithms, pp. 684–695, 2006.

[23] F. Bonomi, M. Mitzenmacher, R. Panigraphy, S. Singh, and G. Varghese, “Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines,” Proc. ACM SIGCOMM, Sept. 2006.

[24] A. Broder and M. Mitzenmacher, ”Using Multiple Hash Functions to Improve IP Lookups,” Proc. IEEE INFOCOM, 2001.

[25] Z. Broder and A. R. Karlin, “Multilevel adaptive hashing,” in ACM-SIAM SODA, 1990, pp. 43–53.

[26] Kirsch and M. Mitzenmacher, “Simple summaries for hashing with choices,” IEEE/ACM Trans. Networking, vol. 16, no. 1, pp. 218–231, 2008.

[27] S. Kumar, J. Turner, and P. Crowley, “Peacock hashing: Deterministic and Updatable Hashing for High Performance Networking,” Proc. IEEE INFOCOM, 2008, pp. 556–564.

[28] Yossi Kanizo, David Hay, and Isaac Keslassy, “Optimal Fast Hashing,” Proc. IEEE INFOCOM, 2009.

[29] Y. Zhu, H. Jiang, J. Wang and F. Xian, “HBA: Distributed Metadata Management for Large Cluster-Based Storage Systems,” IEEE Transactions on Parallel and Distributed Systems, 2008.

[30] Kumar, J. Xu and E. Zegura, “Efficient and Scalable Query Routing for Unstructured Peer-to-Peer Networks,” Proc. IEEE INFOCOM, 2005.

[31] M. Mitzenmacher, “Compressed bloom filters,” Proc. ACM PODC, pp. 144–150, 2001.

[32] S. Cohen and Y. Matias, “Spectral bloom filters,” ACM SIGMOD, pp. 241–252, 2003.

[33] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, “Multilayer Compressed Counting Bloom Filters,” Proc. IEEE INFOCOM, 2008.

[34] Z. Morley Mao, Vyas Sekar, Oliver Spatscheck, Jacobus van der Merwe, Rangarajan Vasudevan, “Analyzing Large DDoS Attacks Using Multiple Data Sources,” Proceedings of ACM SIGCOMM Workshop on Large-Scale Attack Defense (LSAD), 2006.

[35] P. Vixie, G. Sneeringer, M. Schleifer, Events of 21-Oct-2002. Available: http://f.root-servers.org/october21.txt

[36] S. Gill, “Maximizing Firewall Availability,” Available: http://www.qorbit.net/documents/maximizing-firewall-availability.htm

[37] Hyogon Kim, Jin-Ho Kim, Inhye Kang, Saewoong Bahk, “Preventing Session Table Explosion in Packet Inspection Computers,” IEEE Transactions on Computers, 2005.

[38] R. Rivest, “The MD5 Message-Digest Algorithm,” RFC 1321, April 1992.

[39] Kang Li and Zhenyu Zhong, “Fast Statistical Spam Filter by Approximate Classifications,” ACM SIGMETRICS Performance Evaluation Review, June 2006.

[40] M. Handley, V. Paxson, C. Kreibich, “Network Intrusion Detection: Evasion, Traffic Normalization and End-to-End Protocol Semantics,” In Proceedings of the USENIX Security Symposium, August 2001.

[41] J.D. Touch, “Performance analysis of MD5,” Computer Communication Review, vol. 25, no. 4, pp. 77–86, Oct. 1995.

[42] Z. Genova and K. Christensen, “Efficient Summarization of URLs using CRC32 for Implementing URL Switching,” *Proc. the 27th IEEE Conference on Local Computer Networks (LCN)*, pp. 343-344, November 2002.

[43] O. Erdogan and P. Cao, “Hash-av: fast virus signature scanning by cache-resident filters,” *Globecom*, November 2005.

[44] R. Rivest, “The MD5 Message-Digest Algorithm,” RFC 1321, April 1992.

[45] B. Jenkins, “A hash function for hash table lookup,” Available: http://burtleburtle.net/bob/hash/doobs.html

[46] R. Hinden, “Virtual Router Redundancy Protocol (VRRP),” RFC 3768, April 2004.

[47] Ryan McBride, “Firewall Failover with pfsync and CARP,” Available: http://www.countersiege.com/doc/pfsync-carp/

[48] OpenBSD project. Available: http://www.openbsd.org/index.html

[49] D. Hartmeier, “Design and Performance of the OpenBSD Stateful Packet Filter (pf),” In Proceedings of the USENIX Annual Technical Conference, 2002.

[50] The High-availability Linux project. Available: http://www.linux-ha.org/

[51] The keepalived project. Available: http://www.keepalived.org/

[52] H. Welte, “ct_sync: state replication of ip_conntrack,” Linux Symposium, 2004.

[53] Alan Robertson, “Linux-HA Heartbeat System Design,” ALS 2000.

[54] Netfilter. Available: http://www.netfilter.org

[55] M. Dahlin, B. Chandra, L. Gao, and A. Nayate, “End-to-end WAN Service Availability,” IEEE/ACM Transactions on Networking, 2003.

[56] C. Boutremans, G. Iannaccone, and C. Diot, “Impact of link failures on VoIP performance,” in Proc. NOSSDAV, pp. 63–71, 2002.

[57] M. Allman, “A Web Server's View of the Transport Layer,” ACM Computer Communication Review, October 2000.

[58] F. D. Smith, F. H. Campos, K. Jeffay and D. Ott, “What TCP/IP Protocol Header Can Tell Us About the Web,” Proc. ACM SIGMETRICS, June 2001.

[59] F. Hernandez-Campos, K. Jeffay, and F. Donelson-Smith, “Tracking the Evolution of Web Traffic: 1995-2003”, Proceedings of the 11th IEEE/ACM MASCOTS Conference, pp. 16-25, October 2003.

[60] N. Brownlee and K.C. Claffy, “Understanding Internet Traffic Stream: Dragonflies and Tortoises,” IEEE Communications, Vol. 40, No. 10, pp. 110-117, 2002.

[61] Shaikh, J. Rexford, and K.G. Shin, “Load-Sensitive Routing of Long-Lived IP Flows,” Proc. ACM SIGCOMM, September 1999.

[62] L. Guo and I.Matta, “The War between Mice and Elephants,” In Proc. IEEE Int. Conf. Network Protocols (ICNP), 2001.

[63] NLANR PMA Trace. Available: http://pma.nlanr.net/

[64] DongJin Lee and Nevil Brownlee, “Passive Measurement of One-way and Two-way Flow Lifetimes,” Proc. ACM SIGCOMM, 2007.

[65] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, “Introduction to Algorithms,” Prentice Hall, 1 edition, 1990.

[66] T. Karargiannis, A. Broido, M. Faloutsos, and K.C. Claffy, “BLINC: Multilevel Traffic Classification in the Dark,” ACM SIGCOMM, 2005.

[67] S. Sen, O. Spatscheck, and D. Wang. Accurate, “Scalable In-network Identification of P2P Traffic Using Application Signatures,” Proc. WWW, 2004.

[68] A.B. Bomdi, “Characteristics of Scalability and Their Impact on Performance,” Proc. the second international workshop on Software and performance, 2000.

[69] Intel, “Using the RDTSC Instruction for Performance Monitoring,” Technical report, Intel Corporation, 1997.

[70] Yifeng Zhu, Hong Jiang, **“**False Rate Analysis of Bloom Filter Replicas in Distributed Systems,” ICPP, 2006.

[71] Kirsch and M. Mitzenmacher, “The Power of One Move: Hashing Schemes for Hardware,” Proc. IEEE INFOCOM, 2008.

[72] R. Pagh and F. Rodler, “Cuckoo Hashing,” Journal of Algorithms, 51(2), pp. 122-144, 2004.

[73] V. Jacobson, “Compressing TCP/IP Headers,” RFC 1144, February 1990.

[74] N. Brownlee, “Some Observations of Internet Stream Lifetimes,” Proc. the Passive and Active Measurement Workshop (PAM), 2005.

[75] M. Colajanni, D. Gozzi, and Mirco Marchetti, “Enhancing Interoperability and Stateful Analysis of Cooperative Network Intrusion Detection Systems,” Proc. ACM ANCS, 2007.

[76] L. Desmet, F. Piessens, W. Joosen, and P. Verbaeten, “Bridging the Gap between Web Application Firewalls and web applications,” Proc. ACM FMSE, 2006.

[77] H. Sengar, D. Wijesekera, H. Wang and S. Jajodia, “VoIP Intrusion Detection Through Interacting Protocol State Machines,” Proc. IEEE Dependable Systems and Networks (DSN), 2006.

[78] B. Michel, K. Nikoloudakis, P. Reiher, and L. Zhang, “URL Forwarding and Compression in Adaptive Web Caching,” in *IEEE INFOCOM*, pp.670-678, March 2000.

[79] Susan Dumais and Hao Chen, “Hierarchical Classification of Web Content,” ACM SIGIR conference, July 2000

[80] P. Y. Lee, S. C. Hui, and A. C. M. Fong, IEEE “An Intelligent Categorization Engine for Bilingual Web Content Filtering,” IEEE Transactions on Multimedia, vol. 7, no. 6, 2005.

[81] P. Y. Lee, S. C. Hui and A. C. M. Fong, ”Neural Networks for Web Content Filtering,” IEEE Intelligent Systems, 2002.

[82] Mohamed Hammami, Youssef Chahir, and Liming Chen, “WebGuard: A Web Filtering Engine Combining Textual, Structural, and Visual Content-Based Analysis,” IEEE Transactions on Knowledge and Data Engineering, Vol. 18, No. 2, 2006.

[83] Wang Hui-chang , Ruan Shu-hua and Tang Qi-jie, “The Implementation of a Web Crawler URL Filter Algorithm Based on Caching,” International Workshop on Computer Science and Engineering, 2009.

[84] Zhou, Z., Song, T. and Jia, Y.,”A High-Performance URL Lookup Engine for URL Filtering Systems,” Proc. IEEE ICC, 2010.

[85] Xiaoming Li and Wangsen Feng, “Two Effective Functions on Hashing URL,” Journal of Software, vol.14, pp. 177-192, 2004.