High-Speed Event-Counting and -Classification Using a Dictionary Hash Technique*

Paul E. McKenney
Information Sciences and Technology Division
SRI International
pmckenne@us.ibm.com

Jacob N. Sarvela
Department of Mathematics
San Jose State University

March 13, 1999

Abstract

This paper presents a "dictionary hash" technique that, when presented with a large group of labeled events, can stochastically determine the number of unique labels quickly and with a small amount of memory. This technique is applicable to areas such as signal processing, process monitoring and control, and computer communications network monitoring and control.

As a specific example, this paper focuses on application of this technique to congestion-avoidance algorithms in high-speed computer-communications networks. In this application, the events are packet[1] arrivals at a particular network node, and the labels consist of the source and destination addresses in the packets. The set of all packets with a particular source/destination address pair constitutes a "session"; the more sophisticated congestion-avoidance algorithms require knowledge of the number of active sessions. This knowledge can be provided in an effective and timely manner by the dictionary hash technique presented in this paper.

The technique is configurable to any desired degree of accuracy and lends itself to a simple realization in high-speed parallel hardware. This paper will describe how to optimize performance of both hardware and software implementations of the dictionary hash technique.

*This work was supported by the Defense Advanced Research Projects Agency (DARPA) under contract NOO140-87-C-8907 and by SRI IR&D.

[1]A "packet" is a block of information that is switched through the network as a unit.

Keywords: high-speed networks, congestion avoidance, network monitoring, process control, probabilistic set membership.

1 Introduction

In this section, we will focus on a specific application (congestion avoidance in high-speed networks), and show how that application can benefit from event counting and classification.

Queueing theory predicts that equilibrium queue lengths will increase without bound as the load offered to a store-and-forward network approaches the capacity of the network [1]. Furthermore, many protocols will inflict a sudden increase in offered load upon the network (due to retransmissions) when the round-trip delay rises above a certain level [2]. This can cause the load carried by the network to *decrease* as the offered load increases, leading to congestive collapse.

This effect has been observed in production networks, for example, in ARPANET [3]. Congestive collapse causes very inefficient use of network resources, unreliable data transfer (i.e., a large fraction of the data entering the network fails to reach its destination), unfair allocation of resources, and extremely long queueing delays. Since these effects are clearly undesirable, there is an incentive to develop algorithms that control networks in a way that avoids this problem.

To date, only the simplest congestion-avoidance algorithms have been seriously considered for use in production networks [3, 4, 5]. Two of these base their action on simple speed-up/slow-down signals sent from intermediate nodes to the source node, and thus must strike a compromise between steady-state oscillation and slow response to changes in network state. The third avoids this trade-off by use of direct rate control of sessions; however, it is subject to making inappropriate adjustments of the session rates because it lacks the information needed to estimate the effects of its adjustments.

More sophisticated algorithms [6, 7, 8] use knowledge of the number of sessions flowing through each node to avoid these problems. The dictionary hash technique presented in this paper provides this knowledge in an effective and timely manner.

This paper presents the dictionary hash algorithm and compares it to similar algorithms used in probabilistic spelling-checkers. The algorithm is analyzed to show the mean and variance of the error probability for both single-hash and multiple-hash variants. This analysis is used to determine the optimal number of hash functions for both software and hardware implementations of the algorithm. Finally, a relaxation method that promises to further increase the accuracy of the algorithm is presented.

The dictionary hash algorithm will be seen to provide an effective method for counting classes of events. This algorithm may be implemented either in software in real-time systems or as high-speed parallel hardware, and may be tuned to any desired degree of accuracy.

1.1 Alternative Techniques

This section examines alternatives to the dictionary hash technique and shows how each is deficient for high-speed applications.

The simplest and fastest way of remembering which labels have already been counted is to use a simple array, indexed by a number that uniquely identifies the things being counted. For example, the Internet has relatively small 32-bit addresses, so that a session is uniquely identified by a 64-bit quantity. Unfortunately, this results in an infeasible 2^{64} element array.

Various types of search trees are heavily used in database applications [9]. These methods are relatively slow (requiring numerous memory references) and require complex control circuitry, making them unsuitable for use in switches operating in gigabit-per-second networks. Furthermore, these methods require a memory-allocation scheme, which makes them prone to sudden failure when memory is exhausted.

If the events can be marked and if all events carrying a particular label are produced by a single source, then each source can specially mark one of the events carrying a particular label during each measurement period. The number of these specially marked events can then be simply counted. However, this scheme is subject to large inaccuracies if events can be lost (e.g., due to bit errors or congestion in communications networks) or if there are random delays in the system (e.g., due to queueing). Furthermore, this scheme requires that the entire system agree on a single value for the measurement period.

2 Description of Dictionary Hash Technique

The dictionary hash technique is based on an algorithm described and analyzed by Carter et al. [10].[2] Carter's algorithm is summarized here, as it provides a good base for understanding our algorithm.

Carter's algorithm maintains a bit-vector that is indexed by multiple hash functions.[3] The bit vector is initialized by first setting the entire vector to zero, then setting to one those bits that are indexed by one of the hash functions when applied to the words in the dictionary. Later, any word that hashes to a zero-bit is known to be misspelled. A word that hashes to all one-bits has probability $(1-\epsilon)$ of being in the dictionary, where ϵ depends on the size of the vector, the number of hash functions, and the number of words in the dictionary, as will be shown in Section 3.

This algorithm can be used to count stochastically the number of sessions passing through a node in a high-speed network during a predetermined measurement period. As

[2]This algorithm is used by some spelling-checkers.

[3]See Sedgewick [11] for a definition of hash functions and a description of how they are used.

before, the bit vector is initially zeroed. As each packet arrives at the node, a string consisting of that packet's source and destination address is hashed by each of the hash functions. If the bit indexed by any of the hash functions is a zero-bit, the packet is known to belong to a session that has not yet been counted. If all the bits indexed by the hash functions are one-bits, the packet has some probability $(1-\epsilon)$ of belonging to a session that has already been counted. (Section 3 describes how to make this probability arbitrarily close to 1.) Any zero-bits indexed by a hash function are set to one, thus a subsequent packet belonging to the same session will be recognized as having already been counted.

2.1 Modifications for Real-Time Use

Carter's algorithm requires that the entire bit-vector be set to zero at the beginning of each measurement period. This operation causes delays that are intolerable in real-time systems such as our high-speed network application. Thus, the algorithm must be modified to avoid this delay by replacing the bit-vector with a vector of sequence numbers. A global sequence number is incremented at the beginning of each measurement period; where Carter's algorithm would check for bits being set, the new algorithm checks for equality to the current global sequence number. If at least one of the sequence numbers in the vector differs from the current global sequence number, the packet belongs to a session that has not yet been counted, otherwise, the packet has some probability $(1-\epsilon)$ of belonging to a session that has already been counted.

Two important boundary conditions must be accounted for: overflow of the global sequence number and persistence of errors from one measurement period to the next.

In order to prevent old sequence numbers from persisting for a full cycle of the global sequence number, a roving pointer[4] into the vector must be maintained. At the beginning of each measurement period, the entry currently pointed to is set to an illegal sequence number (e.g., if 16-bit unsigned sequence number ranges from zero to 32767, then 65535 is an illegal sequence number). The number of legal values for the sequence number must be greater than the number of entries in the vector (and thus the word size of the sequence number must be at least $\lceil \lg(T) \rceil + 1$, where "lg" is the base-2 logarithm and T is the size of the vector).

Persistent errors are caused by distinct source/destination address pairs hashing to the same values. These errors can be rendered temporary by including the current value of the sequence number in the quantity to be hashed.

A software implementation of the modified algorithm with five hash functions would execute the pseudo-code in Appendix A.1 upon receipt of each packet. The pseudo-code

[4] A "roving pointer" is an index into the vector of sequence numbers that is incremented at the end of each measurement period. This index thus "roves" through the entire vector.

in Appendix A.2 would be executed between measurement periods to manage the sequence numbers.

2.2 Modifications for Implementation in Parallel Hardware

Since the dictionary hash technique is composed of simple operations and is of time complexity $O(1)$, it is an ideal candidate for implementation in hardware in very-high-speed networks.

However, to allow a parallel implementation of the technique, each hash function must be given its own private RAM vector (otherwise, costly and slow multiport RAMs would be required). A block diagram of an implementation using three hash functions is shown in Figure 1.

The source and destination addresses are extracted from each packet by the "Address Extraction" unit. The addresses are passed in parallel to the "Hardware Hash" units, where they are combined with the current sequence number by hashing functions to yield indexes.[5] Possible candidates for the hashing function include cyclic redundancy check, checksums, and linear combinations.

The indexes and the current sequence number are passed to the "RAM Lookup" units, each of which performs a read-modify-write cycle to the RAM location addressed by its index. If the value read from the RAM matches the current sequence number, the unit asserts its "Found" line, and in any case writes the value of current sequence number to the RAM location addressed by the index.

The three-input NAND gate at the bottom of the figure will assert the "Not In Table" line if any of the "Found" lines are not asserted. Thus, the "Not In Table" line will be asserted if the current packet belongs to a session that has not yet been counted. This line may be used to control a counter that will produce the total number of sessions, or it might feed into additional logic that classifies the packets by some criterion (thus producing the number of sessions within each such class).

Each "Hardware Hash" unit must also implement a roving pointer into its RAM.[6] Each time the sequence number is incremented, the RAM location addressed by the roving pointer must be set to an illegal value, and the roving pointer must be incremented. This practice scrubs old sequence numbers from the RAM.

Assuming that the RAM lookup for a packet is overlapped with the hashing of the next packet,[7] the only performance requirement is that the hardware must be able to complete a lookup in the time it takes for a minimum-sized packet to be received. Internet Protocol

[5]Each "Hardware Hash" unit must use a distinct hash function that is statistically independent from that of the other units.

[6]Although a single roving pointer could be shared by all of the "Hardware Hash" units.

[7]This is possible since there is no feedback from the "RAM Lookup" units to the "Hardware Hash" units.

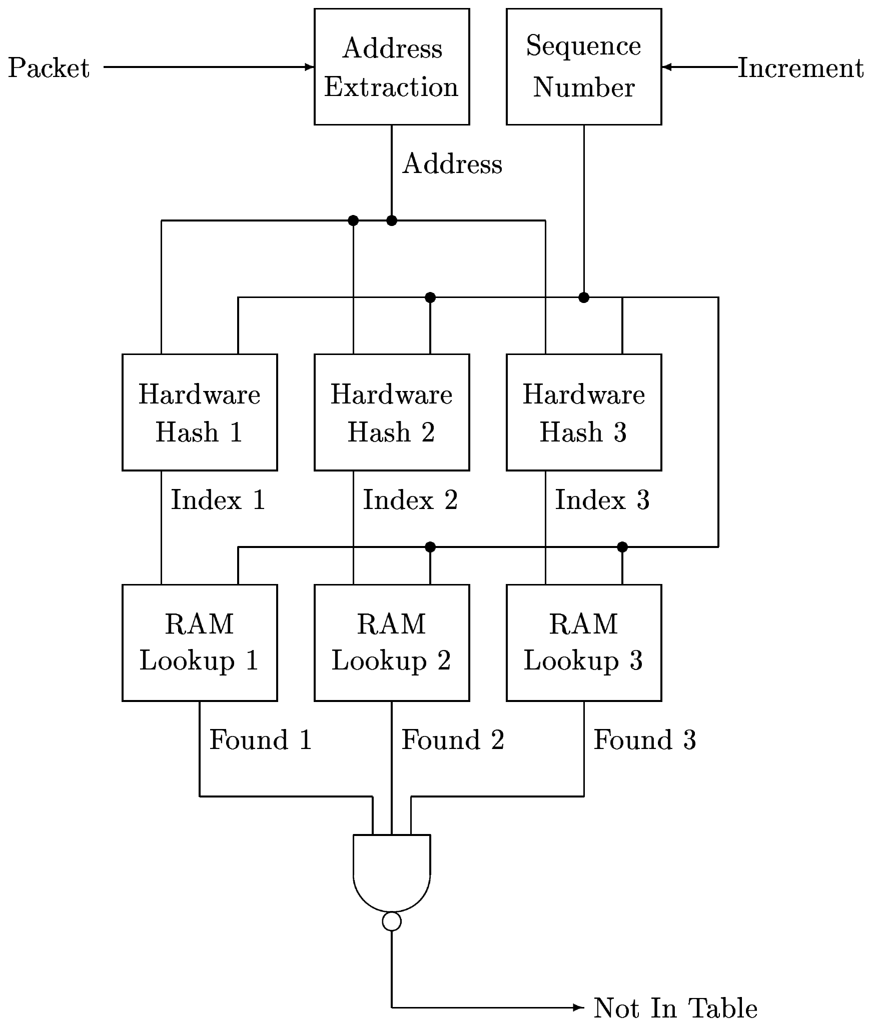

Figure 1: Hardware implementation.

(IP) packets will contain at least an IP header and a link-level header, each containing about 160 bits, for a total of about 320 bits, or about 267 nanoseconds on a 1.2 gigabit-per-second communications line. This is well within the capabilities of current static RAM technology.

3 Analysis

The dictionary hash technique has a finite probability of error, i.e., of failure to recognize a new session as distinct from other sessions.

The following sections analyze this error, first for single hash functions, then for multiple hash functions. Design rules to aid in determining the optimal number of hash functions for a given configuration are presented. Finally, the use of the relaxation technique to decrease error is introduced.

3.1 Single Hash Function

Ideal hash functions are assumed throughout this paper; each hash function is assumed to map the elements of its domain into its range randomly with a uniform probability distribution [12]. This assumption reduces the analysis of the algorithm to that of a variant of the well-known "occupancy problem" from probability theory [13]. This section outlines an approach to this problem that produces variances as well as expected values of the probability of error.

Defining $p_{n,k}$ to be the probability that exactly k entries of a table of size T will be filled in after n distinct items have been added to the table, we obtain

$$p_{n,k} = \frac{k}{T} p_{n-1,k} + \frac{T-k+1}{T} p_{n-1,k-1}. \tag{1}$$

The first term of the right-hand side is the probability that item n collided with a previous item (and thus did not fill in an additional entry), and the second term is the probability that item n did *not* collide with a previous item (and thus *did* fill in an additional entry). The boundary conditions are $p_{0,k} = 0$ for k greater than 0, $p_{n,0} = 0$ for n greater than 0, and $p_{0,0} = 1$. In other words, no entries will be filled in until at least one item has been added to the table.

The moments of this probability distribution may be found through use of moment generating functions (see Appendix B.1). These moments may be used to find exact values for the expected value ($E(k)_n$) and variance ($\sigma^2(k)_n$) of k after n distinct items have been added to the table:

$$E(k)_n = T\left[1 - \left(1 - \frac{1}{T}\right)^n\right] \tag{2}$$

$$\sigma^2(k)_n = T(T-1)\left(1-\frac{2}{T}\right)^n + T\left(1-\frac{1}{T}\right)^n - T^2\left(1-\frac{1}{T}\right)^{2n}. \tag{3}$$

The expected value and variance of the probability of error ϵ (i.e., the probability of not counting the next item added to the table) can be obtained by dividing Equations 2 and 3 by T and T^2, respectively.

By defining a "fill factor" f equal to $\frac{n}{T}$ and assuming large T, we obtain the following simpler expression for the expected value of the probability of error $E(\epsilon)$:

$$E(\epsilon) = 1 - e^{-f}. \tag{4}$$

The error in this approximation will be less than 1% for T greater than one hundred[8].

3.2 Multiple Hash Functions

An implementation that uses m hash functions will have m vectors, each of size T. The probability of error is the probability that *all* of them err (assuming large T):

$$E(\epsilon)_n = \left(1-\left(1-\frac{1}{T}\right)^n\right)^m. \tag{5}$$

The variance for m hash functions, $\sigma^2_m(\epsilon)_n$, can be defined in terms of the first and second moments for a single hash function (see Appendix B.2):

$$\sigma^2_m(\epsilon)_n = \left(1+\frac{(T-1)}{T}\left(1-\frac{2}{T}\right)^n - \frac{(2T-1)}{T}\left(1-\frac{1}{T}\right)^n\right)^m - \left(1-\left(1-\frac{1}{T}\right)^n\right)^{2m}. \tag{6}$$

Note the decrease in the variance for large m.

For optimization purposes, it is helpful to assume large T and to redefine T to be the sum of the sizes of the individual vectors, instead of the size of each of the vectors. Applying these modifications to Equation 5 yields

$$E(\epsilon) = \left(1 - e^{-mf}\right)^m. \tag{7}$$

The following sections present design rules for selecting m.

[8]The temptation to approximate Equation 3 with $\sigma^2(\epsilon) = \frac{e^{-f}}{T}$ must be resisted, since $(1-1/T)^T$ converges to e^{-1} at a rate of only $1/T$. This makes it impossible to cancel the first and third terms of Equation 3, as the magnitude of the remaining term would be on the order of the error in these two terms.

3.2.1 Software Implementation

Software implementations can increase the number of hash functions freely with insignificant additional memory. Therefore, the optimal number of hash functions may be determined by finding the minimum of Equation 7, which is located at

$$m = -\frac{\log(\epsilon)}{\log(2)}. \tag{8}$$

This expression, derived for multiple hash functions over multiple vectors, is identical to the expression derived by Carter et al. [10] for multiple hash functions over a single vector because the performance of the single-vector and multiple-vector variants converges to identical values for large table size [14].

Once m has been determined, T may be computed by solving Equation 7, yielding

$$T = -\frac{mn}{\log(1 - \epsilon^{\frac{1}{m}})}. \tag{9}$$

As an example, assume that a network node needs to count up to 1000 sessions to an accuracy of 1%. Substituting $\epsilon = 0.01$ into Equation 8, we obtain $m = 6.644$. Since m must be an integer,[9] we substitute $m = 6$ and $m = 7$ into Equation 9, yielding 9617 and 9593 words, respectively. Thus, $m = 7$ and $T = 9597$ (rounding 9593 up to the nearest multiple of seven) are optimal for this example.

3.2.2 Hardware Implementation

Hardware implementations should consider the cost of circuitry needed to implement each hash function. If the cost of each word of memory is α and the cost of the circuitry implementing each hash function is β, then the optimal number of hash functions may be found by minimizing

$$\alpha T + \beta m, \tag{10}$$

subject to [from Equation 7]

$$\epsilon \leq \left(1 - e^{-\frac{mn}{T}}\right)^m, \tag{11}$$

where ϵ is the desired expected error rate. Since the objective function is linear, the minimum will be located on the boundary of the feasible region at a point where the slope of the boundary is equal to the slope of the lines of equal cost. This point may be located by

[9]Thus, the two possible values for m may be found from Figure 2 by finding 0.01 on the "Expected Error" axis and noting that the $m = 6$ and $m = 7$ curves intersect the axis on either side of this point.

solving the constraint for T, substituting the result into the objective function, and finding the minimum. This procedure will yield the following:

$$\frac{\beta}{\alpha n} = \frac{1}{\log(1-\epsilon^{\frac{1}{m}})} - \frac{\epsilon^{\frac{1}{m}}\log(\epsilon)}{m(\log(1-\epsilon^{\frac{1}{m}}))^2\left(1-\epsilon^{\frac{1}{m}}\right)}. \tag{12}$$

A graphical representation of this expression is shown in Figure 2.

To use this figure, find the point $(\epsilon, \frac{\beta}{\alpha n})$ on the figure (where ϵ is the expected error, β is the cost per unit for hash function circuitry, α is the cost per word of memory, and n is the expected number of entries in the table). The optimal number of hash functions will be given by the curve closest to this point. The vector size T may then be determined using Equation 9.

It is important to remember that α is the cost per *word* of memory and that the word size must be at least $\lceil \lg(T/m) \rceil + 1$ bits.

For example, assume that the network node of the previous section (that needed to count up to 1000 sessions within 1% accuracy) has a cost of \$2000 per megabyte of memory and \$300 per hash function. Assuming that each memory word will require two bytes, we get a cost of \$0.00381 per word. The memory cost ratio will thus be 78.6, so we find the point (.01, 78.6) on Figure 2, which lies between the $m = 1$ and $m = 2$ curves. Evaluating Equation 9 for $m = 1$ and $m = 2$ yields 99499 words and 18982 words, respectively, and when these values are substituted into Equation 10 we obtain costs of \$679.55 and \$672.41, respectively. Thus, the optimal design has two hash functions and 18982 words of memory (split into two banks of 9491 words apiece), and costs \$672.41.

Note that the word size (16 bits) is greater than $\lceil \lg(18982/2) \rceil + 1 = 15$, as required.

3.3 Relaxation

The fact that the expected error can be obtained in closed form suggests application of the relaxation technique from numerical analysis. For example, Equation 2 may be solved for n, giving

$$n = \frac{\log(1-\frac{k}{T})}{\log(1-\frac{1}{T})}. \tag{13}$$

Since the value of T is known in advance for a particular implementation, this expression could be cast into tabular form to allow speedy evaluation.[10] Additional work is needed to determine the conditions under which relaxation is most effective.

[10]It is tempting to simplify this expression by assuming large T, but this reduces to $n = k$.

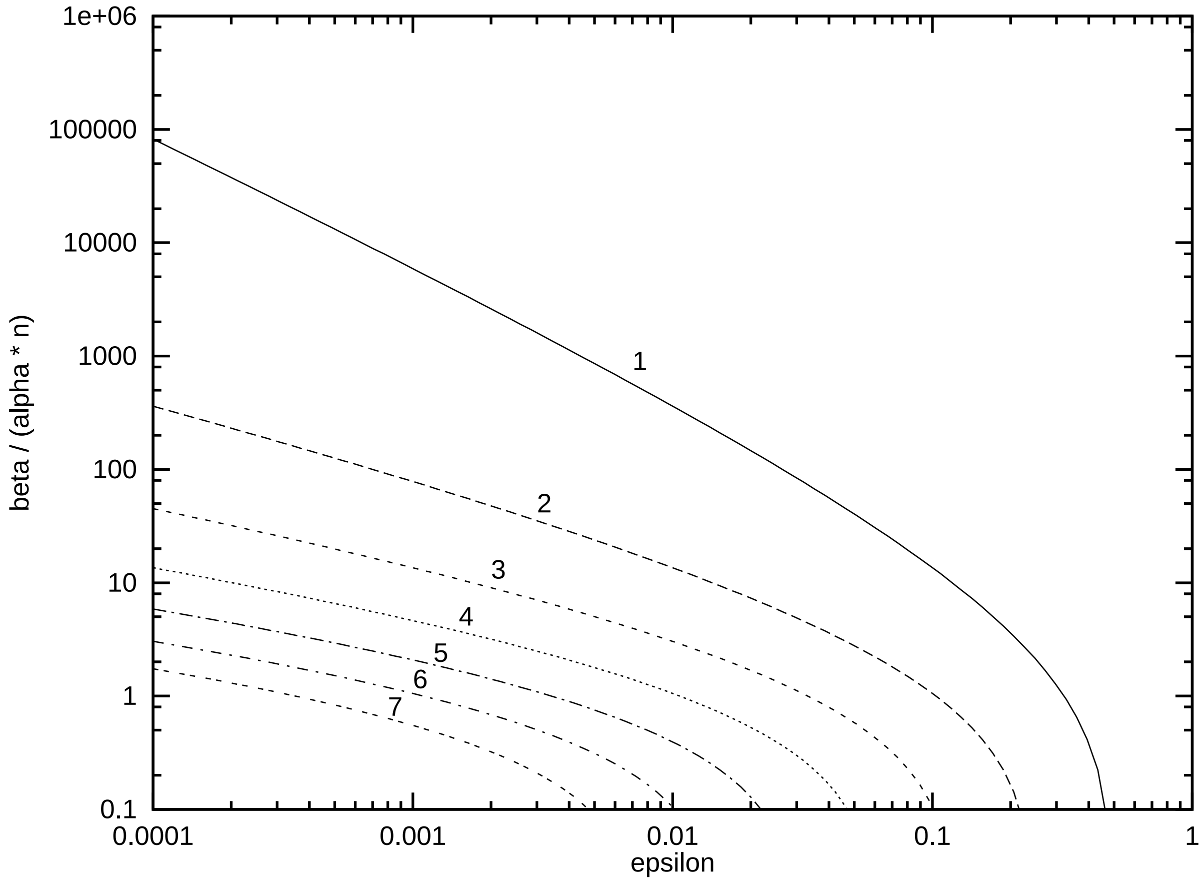

Figure 2: Design rule for hardware implementation.

4 Conclusions

This paper has presented and analyzed a stochastic technique for counting classes of events. The expected value and variance of the error has been presented, along with expressions that may be used to find the optimal number of hash functions given the desired error rate and (in the case of hardware) memory and circuitry costs.

This technique may be used for signal processing, process monitoring and control, and computer communications network monitoring and control. In particular, when implemented in hardware, it allows sophisticated congestion-avoidance algorithms to be applied to gigabit-per-second computer networks.

References

[1] Leonard Kleinrock. *Queueing Systems vII.* John Wiley and Sons, 1976.

[2] J.B. Postel. Transmission Control Protocol. Technical Report RFC793, Network Information Center, SRI International, September 1981.

[3] Van Jacobson. Congestion avoidance and control. In *SIGCOMM '88*, pages 314–329, August 1988.

[4] Raj Jain and K.K. Ramakrishnan. Congestion avoidance in computer networks with a connectionless network layer. Technical Report DEC-TR-506, Digital Equipment Corporation, Maynard, Massachusetts, August 1987.

[5] J. Zavgren. Congestion control in the DDN with applications to SURAN. In *SURAN Working Group Meeting*, Menlo Park, California, February 1988.

[6] H. Hayden. Voice flow control in integrated packet networks. Technical Report LIDS-TH-1152, MIT Laboratory for Information and Decision Systems, 1981.

[7] D. Bertsekas and R. Gallager. *Data Networks.* Prentice-Hall, Inc., 1987.

[8] Paul E. McKenney. Congestion avoidance and control. Technical Report SRNTN61, SRI International, In preparation.

[9] Donald Knuth. *The Art of Computer Programming.* Addison-Wesley, 1973.

[10] L. Carter, R.W. Floyd, J. Gill, G. Markowsky, and M.N. Wegman. Exact and approximate membership testers. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 59–65, May 1978.

[11] Robert Sedgewick. *Algorithms.* Addison-Wesley, 1984.

[12] A.C. Bajpai, I.M. Calus, and J.A. Fairley. *Statistical Methods for Engineers and Scientists.* John Wiley, 1978.

[13] W. Feller. *An Introduction to Probability Theory and its Applications.* John Wiley, 1958.

[14] R.W. Floyd. Large-table-size approximations for set membership algorithms. Private Correspondence, November 1988.

A Software Implementation

A.1 Test if Session Already Counted

```
if ((seqvec[hash1(sessionID, curseqno)] == curseqno) &&
    (seqvec[hash2(sessionID, curseqno)] == curseqno) &&
    (seqvec[hash3(sessionID, curseqno)] == curseqno) &&
    (seqvec[hash4(sessionID, curseqno)] == curseqno) &&
    (seqvec[hash5(sessionID, curseqno)] == curseqno))
        newsession = FALSE;
else
        {
        newsession = TRUE;
        seqvec[hash1(sessionID, curseqno)] = curseqno;
        seqvec[hash2(sessionID, curseqno)] = curseqno;
        seqvec[hash3(sessionID, curseqno)] = curseqno;
        seqvec[hash4(sessionID, curseqno)] = curseqno;
        seqvec[hash5(sessionID, curseqno)] = curseqno;
        }
```

A.2 Incremental Zeroing of Vector

```
/* Increment sequence number with wrap-around.                  */

if (++curseqno > 32767)
        curseqno = 0;

/* Invalidate one of the entries -- this guarantees that the    */
/* entire seqvec will be scrubbed before the current sequence   */
/* number is reused.                                            */

seqvec[rovingptr] = 65535;
if (++rovingptr >= SEQVEC_LEN)
        rovingptr = 0;
```

B Derivations

B.1 Derivation of Moments

This section derives moments of the recurrence equation:

$$p_{n,k} = \frac{k}{T} p_{n-1,k} + \frac{T-k+1}{T} p_{n-1,k-1} \tag{14}$$

for fixed n using moment generating functions.

B.1.1 Moment Generating Function Definition

Moment generating functions are defined as follows:

$$G_n(t) = \sum_k p_{n,k} e^{kt}. \tag{15}$$

Taking the first two derivatives:

$$G_n^{'}(t) = \sum_k k p_{n,k} e^{kt} \tag{16}$$

$$G_n^{''}(t) = \sum_k k^2 p_{n,k} e^{kt}. \tag{17}$$

The value $G_n^{(m)}(0)$ is the m^{th} moment of p_n, thus, the mean of p_n is $G_n^{'}(0)$ and the variance of p_n is $G_n^{''}(0) - (G_n^{'}(0))^2$.

B.1.2 Derivation of Moment Difference Equations

Multiplying both sides of Equation 14 by Te^{kt} and summing over k:

$$T \sum_k p_{n,k} e^{kt} = \sum_k k p_{n-1,k} e^{kt} - \sum_k k p_{n-1,k-1} e^{kt} + (T+1) \sum_k p_{n-1,k-1} e^{kt}. \tag{18}$$

The left-hand side of this equation is equal to $G_n(t)$ and the first term of the right-hand side is equal to $G_n^{'}(t)$.

The second and third terms of the right-hand side may be re-indexed, substituting $k+1$ for k, yielding (after rearrangment):

$$TG_n(t) = (1-e^t)G_{n-1}^{'}(t) + Te^t G_{n-1}(t). \tag{19}$$

Differentiating twice:

$$TG_n^{'}(t) = (1-e^t)G_{n-1}^{''}(t) + (T-1)e^t G_{n-1}^{'}(t) + Te^t G_{n-1}(t) \tag{20}$$

$$TG_n^{''}(t) = (1-e^t)G_{n-1}^{'''}(t) + (T-2)e^t G_{n-1}^{''}(t) + (2T-1)e^t G_{n-1}^{'}(t) + Te^t G_{n-1}(t). \tag{21}$$

B.1.3 Derivation of the First Moment

The first moment (or mean) may be found by substituting $t = 0$ into Equation 20:

$$TG_n'(0) = (T-1)G_{n-1}'(0) + T.$$

The solution of this difference equation is

$$G_n'(0) = T\left(1 - \left(1 - \frac{1}{T}\right)^n\right) = \mu. \tag{22}$$

B.1.4 Derivation of the Second Moment

The second moment may be found by substituting $t = 0$ into Equation 21:

$$TG_n''(0) = (T-2)G_{n-1}''(0) + (2T-1)G_{n-1}'(0) + T. \tag{23}$$

Substituting equation 22,

$$G_n''(0) = \left(1 - \frac{2}{T}\right) G_{n-1}''(0) + 2T + (2T-1)\left(1 - \frac{1}{T}\right)^{n-1}. \tag{24}$$

The solution is

$$G_n''(0) = T^2 + (T^2 - T)\left(1 - \frac{2}{T}\right)^n - (2T^2 - T)\left(1 - \frac{1}{T}\right)^n. \tag{25}$$

B.1.5 Derivation of Variance

The variance of the distribution is the second moment minus the square of the first moment:

$$V_n = G_n''(0) - \left(G_n'(0)\right).$$

Substituting Equations 22 and 25 and simplifying gives the desired result:

$$V_n = T(T-1)\left(1 - \frac{2}{T}\right)^n + T\left(1 - \frac{1}{T}\right)^n - T^2\left(1 - \frac{1}{T}\right)^{2n}. \tag{26}$$

B.2 Variance for Multiple Hash Functions

The variance of the error for m hash functions can be obtained in terms of the first two moments for a single hash function using the identity

$$E(\Pi_{i=1}^m (x_i)^2) - (E(\Pi_{i=1}^m x_i))^2 = \left(E(x_i^2)\right)^m - (E(x_i))^{2m}, \tag{27}$$

where the x_i are m independent random variables taken with replacement from distributions with identical means and variances.

Substituting Equations 22 and 25 and dividing by T^2 to obtain error probability yields

$$\sigma_m^2(\epsilon)_n = \left(1 + \frac{(T-1)}{T}\left(1 - \frac{2}{T}\right)^n - \frac{(2T-1)}{T}\left(1 - \frac{1}{T}\right)^n\right)^m - \left(1 - \left(1 - \frac{1}{T}\right)^n\right)^{2m}. \quad (28)$$